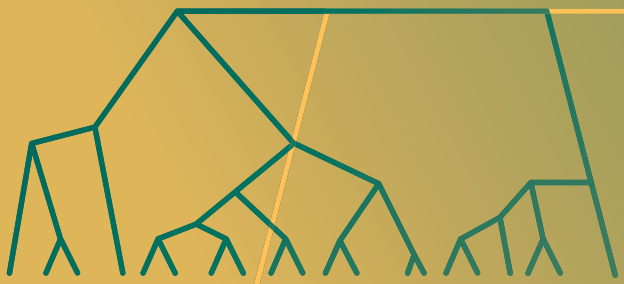


Bernhard Haubold
Angelika Börsch-Haubold

Bioinformatics for Evolutionary Biologists

A Problems Approach

Second Edition

Springer

Bioinformatics for Evolutionary Biologists

Bernhard Haubold · Angelika Börsch-Haubold

Bioinformatics for Evolutionary Biologists

A Problems Approach

Second Edition

Springer

Bernhard Haubold
Department of Evolutionary Genetics
Max-Planck-Institute
for Evolutionary Biology
Plön, Schleswig-Holstein, Germany

Angelika Börsch-Haubold
Plön, Schleswig-Holstein, Germany

ISBN 978-3-031-20413-5 ISBN 978-3-031-20414-2 (eBook)
https://doi.org/10.1007/978-3-031-20414-2

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Like all of biology, evolutionary biology is rapidly turning digital as we are surrounded by more and more cheap computers and cheap computerized machines for gathering data, especially DNA sequence data. This book is designed to help biologists analyze these data using classical bioinformatics techniques.

We teach bioinformatics as a practical skill focused on the analysis of DNA sequences on the Unix command line. We do this by posing a series of graded and solved Problems, most of which are meant to be tackled on the Unix command line. The Unix command line is our interface of choice, because it is malleable like no graphical interface.

The malleability of the Unix command line takes some practice to unlock, and opportunities for getting this practice have grown in leaps and bounds since the publication of the first edition of *Bioinformatics for Evolutionary Biologists* in 2017. In 2019 Microsoft released an improved version of their Linux subsystem that runs on Windows 10, which was in turn improved for Windows 11 two years later. In 2019 Google's chromeOS joined the club of Unix-capable systems with the relase of Linux for Chromebooks. Given that the Unix command line has been part of Apple's macOS since 2001, this means that all four major computer operating systems—Windows, macOS, chromeOS, and Linux—now allow users easy access to the command line. Out of these four, we have tested on two, Windows 11 with Ubuntu 20.04 and macOS Big Sur.

This book is intended for biologists with no prior knowledge of Unix. We cover the fundamentals of interacting with the computer on the command line in Chapter 1. In Chapter 2 we align pairs of protein and DNA sequences and score the resulting alignments. The alignment techniques explored here can be sped up substantially by using exact matching, the topic of Chapter 3. In Chapter 4, we mix exact matching with the alignment methods of Chapter 2 to arrive at the fast algorithms of modern molecular biology underlying such popular tools as Blast. Up to this point, we have only aligned pairs of sequences, but at the end of Chapter 4 we generalize this to align multiple sequences. Multiple sequence alignments are the starting point for phylogenies, the topic of Chapter 5. Phylogenies describe the evolution of multiple species. If we are interested in the evolution of individual species, we can use

concepts from population genetics, the topic of Chapter 6. A central model of population genetics traces a sample of genes from the present back in time to their most recent common ancestor. This results in a tree called *coalescent*, which looks similar to the phylogenies of the previous chapter. In the last chapter, Chapter 7, we cover two topics not necessarily based on sequences, but still ubiquitous in modern biology, the statistics of multiple testing and the fundamentals of databases.

The first edition of this book was well received and we have used it regularly in courses taught at the Max-Planck-Institute in Plön and at Lübeck University. Some of our students were so kind as to point out errors—thank you again!—which we have tried our best to fix. Some errors probably remain, and we are bound to have introduced new ones, so if you catch a bug, please drop BH a line at

`haubold@evolbio.mpg.de`

We will add it to the errata list maintained at

`guanine.evolbio.mpg.de/beb2`

and, we hope, fix it in a future edition. While fixing errors, we also strove to clarify and in places expand the narrative. As a result, the total number of Problems has grown from under 500 to over 800.

However, the biggest change concerns the Unix tools written for the course. We've rewritten them in Go and distribute them as a biological toolbox, a "biobox", at

`sn.pub/dy6S42`

While rewriting the programs, we also added a few new ones, for example `plotLine`, to quickly convert pairs of x, y values into line graphs.

Finally, let us reiterate a point from the first edition. Half of this book consists of Answers, which you find in the yellow pages in the back. These are designed to be read even if you do find your own solution to a particular Problem. An Answer might expand on the Problem, might inspire you to improve your answer, and rarely, we hope, might give you the satisfaction of having come up with a better one. In that case we'd be particularly keen to hear from you.

Plön, August 2022

Bernhard Haubold
Angelika Börsch-Haubold

Contents

Part I Problems

1 The Unix Command Line 3
1.1 Getting Started 4
1.2 Files, Directories, and Programs 11
1.3 Scripts 18

2 Optimal Alignment 29
2.1 Keeping Score 29
2.2 Construction 39
2.3 Application 51

3 Exact Matching 57
3.1 Keyword Trees 57
3.2 Suffix Trees 64
3.3 Suffix Arrays 70
3.4 Text Compression 81

4 Fast Alignment 93
4.1 Global 93
4.2 Local 98
4.3 Glocal 110
4.4 Assembly 122
4.5 Multiple Sequences 132

5 Evolution Between Species: Phylogeny 141
5.1 Trees of Life 141
5.2 Rooted Trees 148
5.3 Unrooted Trees 154

6 Evolution within Populations 161
6.1 Descent from One or Two Parents 161
6.2 The Coalescent 172

7 Interrogating and Storing Data 183
7.1 Statistics 183
7.2 Relational Databases 190

Part II Answers

1 The Unix Command Line 207
1.1 Getting Started 207
1.2 Files, Directories, and Programs 211
1.3 Scripts 217

2 Optimal Alignment 229
2.1 Keeping Score 229
2.2 Construction 239
2.3 Application 250

3 Exact Matching 257
3.1 Keyword Trees 257
3.2 Suffix Trees 262
3.3 Suffix Arrays 266
3.4 Text Compression 274

4 Fast Alignment 287
4.1 Global 287
4.2 Local 291
4.3 Glocal 303
4.4 Assembly 312
4.5 Multiple Sequences 320

5 Evolution Between Species: Phylogeny 329
5.1 Trees of Life 329
5.2 Rooted Trees 333
5.3 Unrooted Trees 339

6 Evolution within Populations 345
6.1 Descent from One or Two Parents 345
6.2 The Coalescent 351

7 Interrogating and Storing Data 361
7.1 Statistics 361
7.2 Relational Databases 367

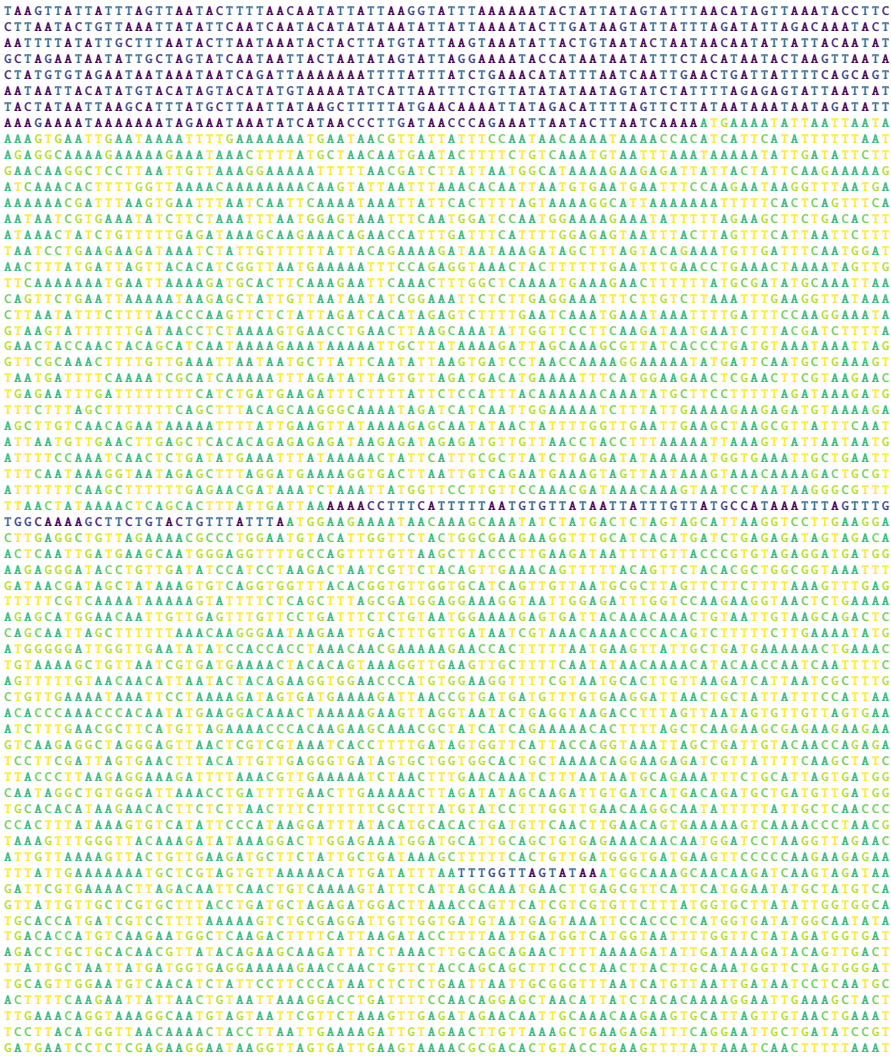
A Unix Guide 377

B Programs 397
B.1 Own 397
B.2 Biobox 398
B.3 Third-Party 398

References 401

Index 405

Part I
Problems



5808 nucleotides, roughly one percent of the genome of the pathogenic bacterium *Mycoplasma genitalium*, which at 580 kb has one of the smallest genomes of any free-living organism; genes are light, intergenic regions dark

Chapter 1
The Unix Command Line

Almost all commercial software comes with attractive graphical user interfaces that allow us to work and play by touching and mousing. This is great for deleting a file by dragging it into a trash can, renaming a file by clicking onto its name, or editing text by mouse selection. However, in biology we might like to check the three billion nucleotides of the human genome for the occurrence of a PCR primer, or compute averages from thousands of expression values distributed across dozens of files. Such operations are hard to perform using click-driven programs. This is because graphical user interfaces are excellent for carrying out the tasks their creators deem important, but they lack the universality that makes learning about computers so fascinating. Computers are universal machines in the sense that they can perform any precisely specified operation. All that is necessary is an interface that lets us communicate every possible operation, not just a finite set, however large it may be.

To illustrate the importance of being able to communicate an infinite number of possible operations, think of the communication system we all know best, our language. Take any sentence that comes to mind and search the world-wide-web with it. Unless you were quoting from memory, chances are, your sentence is unique. This is because we do not memorize sentences, but use rules to construct new ones. These rules are so fundamental, we all know them without even being aware of them. This leaves us free to think about what to say while saying it. Moreover, the words we utter have a curiously vague relationship to what we mean. If someone says: "John is my friend.", the word "friend" neither looks nor sounds like a friend. Nevertheless, we know immediately what "friend" signifies. Taking our cue from language, we expect all powerful communication systems to be characterized by a set of rules and an arbitrary mapping between words and meaning. Communicating effectively with a computer is no different.

The Unix command line, also known as the *shell*, is the de facto standard for text-based, rather than graphics-based, computer communication. It has been around since the late 1960's and has proved flexible enough to adapt rather than go extinct like so may other programs over the years. Its behavior is governed by a standard, the POSIX standard. This means that once you have mastered the Unix shell on one type of computer, you have mastered it on all. If you have never used it, now is a good

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_1

time to start by working through this chapter. Even if you have used the command line before, we recommend you work through the following material to make the most of later chapters.

We assume you are now sitting in front of a computer with an open terminal displaying a blinking cursor like this:

```
jdoe@unixbox:~$ |
```

1.1 Getting Started

New Terms

append (>>)	`head`	`pwd`
auto completion	`history`	`rmdir`
`bash`	home directory	`rm`
`bc`	`ls`	shell
bit	`man`	`tail`
`cd`	`mkdir`	`touch`
code chunk	path	wildcard (∗)
`echo`	pipe	`zsh`
`export`		

1.1 Make a directory for this book by entering

⟨*cli*⟩≡
```
mkdir beb
```

Any command to be entered on the command line is typeset as the special code chunk, ⟨*cli*⟩, for *command line*. Change directory, `cd`, to `beb`.

⟨*cli*⟩+≡
```
cd beb
```

This directory contains all our computational work, so we'd like to find it easily. We do this by giving it a special name, `BEB`, and assigning the current directory to that name.

⟨*cli*⟩+≡
```
BEB=$(pwd)
```

The command `pwd` *prints which directory*. Which is yours?

1.2 What happens when you `echo` the value of `BEB`?

⟨*cli*⟩+≡
```
echo $BEB
```

1.3 Which directory do you reach when you enter

⟨*cli*⟩+≡
```
cd
```

1.4 What happens when you do

⟨*cli*⟩+≡
```
cd $BEB
```

1.5 What happens when you do

⟨*cli*⟩+≡
```
cd ~
```

1.6 Change again into the book directory.

⟨*cli*⟩+≡
```
cd $BEB
```

The variable `BEB` is useful and we'd like to remember it between shell sessions. Whenever we try to remember something, we write it down. The place for writing down shell variables is the resource file in the home directory, `.bashrc` on the Bourne-again shell, `bash`, and `.zshrc` on the z shell, `zsh`. To find out which shell you are using, query the `SHELL` variable by entering

⟨*cli*⟩+≡
```
echo $SHELL
```

Which is yours?

1.7 We'd like to save our `BEB` variable to our resource file. But resource files can be difficult to fix when broken. So we first save a copy of `.bashrc` or `.zshrc`. The copy command, `cp`, has the structure `cp source destination`. Here the destination is the current directory, which is called dot.

⟨*cli*⟩+≡
```
cp ~/.bashrc .
```

There are systems without an initial resource file. If you happen to be working on such a system, the copy command fails, but don't worry, you can just continue regardless and append (>>) one line to the resource file.

⟨*cli*⟩+≡
```
echo "export BEB=$(pwd)" >> ~/.bashrc
```

If the resource file doesn't exist yet, it is constructed *and* appended to. What do you get when you print the tail end (`tail`) of your resource file?

⟨*cli*⟩+≡
```
tail ~/.bashrc
```

1.8 If the last line of your resource file looks fine, test it by opening a new terminal window and enter

⟨*cli*⟩+≡
```
echo $BEB
```

If you don't get the expected result, revert to the backup copy of the resource file and try again.

This book consists of chapters, and we bundle the computations they contain in the directory `ch`, which we make with `mkdir`.

⟨*cli*⟩+≡
```
mkdir ch
```

We can call a directory anything we like, but Unix is case sensitive, so `ch`, `CH`, `Ch`, and `cH` would be four distinct names. What happens when you do

⟨*cli*⟩+≡
```
ls
```

1.9 What happens when you list *all* the contents of your current directory?

⟨*cli*⟩+≡
```
ls -a
```

1.10 We change into `ch`.

⟨*cli*⟩+≡
```
cd ch
```

We are currently in Chapter 1, Section 1. Can you make directories to mirror that structure and change into the directory for Section 1.1?

1.11 Inside `beb/ch/1/1/`, make two more directories, `td1` and `td2`, and list them.

1.12 Change into `td1` and back into its parent directory.

1.13 To minimize typos, the shell supports *auto completion* of names. Change again into `td1`, but this time type only `cd t` followed by `Tab`. This completes the common prefix of the two directories, `td`. To get the two possible suffixes, press `Tab` again. Type `1`, once more followed by `Tab`, to ensure correct completion.

This technique of mixing typing and tabbing is very useful once you get the hang of it. Practice by changing in and out of the current directory. The command `rmdir` removes directories. What happens if you enter

⟨*cli*⟩+≡
```
rmdir td*
```

1.14 Remake the test directories `td1` and `td2` and change into `td1`. Then `touch`, or create, a test file and return to the parent directory.

⟨*cli*⟩+≡
```
mkdir td1 td2
cd td1/
touch tf
cd ..
```

Notice the slash after `td1` from the auto completion. What happens if you now apply `rmdir` to `td1`?

1.15 Can you empty `td1` and then delete it?

1.16 The *recursive* switch, `-r`, allows `rm` to delete a directory and its contents. Remake `td1`, add `tf`, and remove both with `rm`.

1.17 Recreate `td1` and enter it. Create two test files, `tf1` and `tf2`. How would you copy both to `td2`?

1.18 Remove both files with one command and copy them back from `td2`.

1.19 File `tf1` is renamed `tf3` by *moving* it to the new name with `mv`.

⟨*cli*⟩+≡

```
mv tf1 tf3
```

Move `tf2` to `tf4`. What happens when you move `tf3` to `tf4`?

1.20 As you've probably noticed, the cursor cannot be positioned using the mouse. This seems to leave the left and right arrow keys as the only navigation tools. However, to use them, we have to move our right hand from the home row on the keyboard, which we try to avoid. Instead of using the arrow keys, we can press the control key and while keeping it pressed, press `f`, to move the cursor forward by one position. So if `C-f` moves the cursor one position forward, can you guess how to move it one position back?

1.21 When trying to get to the beginning of a line, we can keep `C-b` pressed until we get there. But we might as well jump with `C-a`. Type

```
might as well jump
```

and jump to the beginning of the line. Then race to the end by keeping `C-f` pressed and repeat a few times. But again, we might as well jump to the end. Can you guess how that's done?

1.22 Apart from jumping to the beginning and end of a line, we might like to jump one word at a time. This is done through key combinations based on the meta key, `M`. By default this is mapped to `Esc`. It may also be mapped to `Alt`, which again means you can reach it without moving your left hand from the home row. `M-f` moves forward one word, `M-b` backward. Try this again on "might as well jump". What does `M-d` do?

1.23 While editing the command line, we might make a mistake, which we can undo with `C-_`. What happens when you undo repeatedly?

1.24 The combination `C-k` deletes from the cursor to the end of the line. Try this. What happens when you now press `C-y`?

Table 1.1 Key combinations for moving the cursor and editing the command line

Key Combination	Explanation
`C-a`	position cursor at beginning of line
`C-e`	position cursor at end of line
`C-y`	insert deleted text
`C-b`	move cursor back by one position
`C-f`	move cursor forward by one position
`C-d`	delete character right of cursor
`C-k`	delete right to end of line
`C-_`	undo
`M-b`	jump back by one word
`M-f`	jump forward by one word
`M-backspace`	delete word to the left of cursor
`M-d`	delete word to the right of cursor

1.25 Moving the cursor with key combinations is a bit awkward at first, but once you get used to it, working with the command line becomes much easier than it might be right now. Experiment with the key combinations summarized in Table 1.1. How would you use them to transform "the dog bit the man" into "the man bit the dog"?

1.26 The command line remembers commands and `history` lists them. However, this list can be rather long, so we just look at its tail end by making the output of `history` the input of `tail`.

⟨*cli*⟩+≡
```
history | tail
```

The command for switching output to input, `|`, is called *pipe*, as in, the pipe dream of a pipeline under the Baltic Sea, which n'est pas une pipe. Can you look up the head of the command history? How many items does your command history contain?

1.27 We've seen that the entries in the command history are numbered and we can repeat a command by entering its number preceded by an exclamation mark. Try this. What happens when you enter

⟨*cli*⟩+≡
```
!!
```

1.28 What happens when you enter

⟨*cli*⟩+≡
```
!-3
```

1.29 Like when moving horizontally along the command line, when moving vertically along the command history, we can replace the up and down arrows by key combinations. `C-p` moves to the *previous* command and when repeated to the next one up. Again, this has the advantage compared to the up arrow key that your right hand stays near the home row when navigating the command history. Can you guess how to get the *next* command down the list?

1.30 We can keep `C-p` pressed until we get to the first command stored. But we might as well jump to the beginning with the meta key, `M-<`. What happens when you now try to go to the "previous" command?

Table 1.2 Key combinations for navigating the command history

Key Combination	Explanation
`C-p`	previous command
`!!`	previous command
`C-n`	next command
`!n`	command number `n`
`!-2`	command two steps up the list
`C-r`	reverse-serarch history
`C-g`	quit history search
`M-<`	start of history
`M->`	end of history

1.31 Table 1.2 summarizes the key combinations for navigating the command history. If `M-<` gets us to its beginning, how do we jump to its end?

1.32 A long list of commands is best navigated by searching, `C-r`. Search for commands containing `cd`. This retrieves the last command with `cd`. What happens when you press `C-r` again?

1.33 What happens when you press `C-g`?

1.34 If you'd like to learn more about features of the shell, read the user manual. This is accessed with the program `man`, which itself has an entry in the manual.

⟨*cli*⟩+≡
```
man man
```

Navigate the `man` page with the arrow keys, `q` to quit. Take a look at the EXAMPLES section in the manual entry for `man`. How would you find `man` pages for the keywords `bash` or `zsh`?

1.35 To read up on the navigation commands listed in Tables 1.1 and 1.2, open the `man` page for `bash`—the commands for `zsh` are essentially identical, so under both shells we can do

⟨*cli*⟩+≡
```
man bash
```

Now activate the `man` help function by pressing `h`. How would you look for the pattern *Commands for Moving* and *History*?

1.36 We're working on a computer, but haven't computed anything so far. So let's calculate the number of codons, 4^3.

⟨*cli*⟩+≡
```
echo $((4**3))
```

What do you observe? And what happens if you drop the dollar?

1.37 Instead of printing the result of our computation, we can assign it to the variable `nc`, for *number of codons*.

⟨*cli*⟩+≡
```
((nc=4**3))
```

Can you print the result of this computation?

1.38 For more detailed output, we can write a message containing our variable `nc`.

⟨*cli*⟩+≡
```
((nc=4**3)); echo "The number of codons is $nc"
```

Here we use a semicolon to separate the computation from the `echo` command. What happens if you replace the double quotes by single quotes?

1.39 Apart from exponent (`**`), we can multiply (`*`), add (`+`), and subtract (`-`), as expected. But what happens when we divide (`/`) 22 by 7?

1.40 To calculate fractions on the command line, we can use the basic calculator, `bc`. Start it with the math library (`-l`),

⟨*cli*⟩+≡
```
bc -l
```

To quit, enter `quit`. What is 22/7? Do you recognize this number?

1.41 A convenient way to use `bc` is to pipe the output of `echo` into it.

⟨*cli*⟩+≡
```
echo '22/7' | bc -l
```

What happens if `bc` is run without `-l`?

1.42 We currently use 64-bit computers. This means a "word" in computer memory consists of 64 bits. How many distinct 64-bit words are there? We might be tempted to compute this on the shell, but

⟨*cli*⟩+≡
```
echo $((2**64))
```

gives zero. Can you think of a reason, why this computation fails? Attempt it with `bc`, where n^x is expressed as `n^x`. Can you gesstimate the result (hint: $2^{10} = 1024 \approx 10^3$)?

1.43 How long a DNA sequence do we need to store as much information as a 64-bit word?

1.2 Files, Directories, and Programs

New Terms		
byte	`grep`	`plotLine`
`cat`	`gunzip`	`printf`
`cres`	`gzip`	redirect (>)
`cutSeq`	Homebrew	significance
`cut`	`less`	`tar`
`drawGenes`	`make`	`translate`
FASTA format	octal number	`tr`
`fold`	`od`	`wc`
`git`	`PATH`	`which`

1.44 To start the session, make a new directory for this section and change into it.

1.45 Files are kept inside directories, which may contain further directories. This hierarchy of directories forms a tree, like the one in Fig. 1.1, where the root directory (`/`) contains the six directories `bin` through `var`. What files and directories are contained in your root directory?

1.46 The program `wc` prints the number of lines, words, and bytes for a file. How many directories does your root directory contain (hint: use pipe)?

1.47 Often we are just interested in one of the three numbers listed by `wc`. How can you print only the number of lines (hint: `man`)?

1.48 The directories directly hanging from the root of the directory tree in Fig. 1.1 can only be changed by the system administrator; for example, when installing a program. The `/bin` directory contains quite a few of these programs. Can you find out whether it contains `ls` on your system?

1.49 The file `/bin/ls` might not be the only `ls` on the system. We can find out `which ls` we're currently using with

⟨*cli*⟩+≡
```
which ls
```

Which `which` are you using?

1.50 In Fig. 1.1 the directories rooted on `home` are the home directories. Every user has one and we've already played with making and deleting files and directories in it. We also said that its symbol is tilde, ~. How many files and directories does yours contain?

1.51 Bioinformatics is centered on sequence data, so let's look at a short sequence,

```
>dnaN
ATGAAAATATTA
```

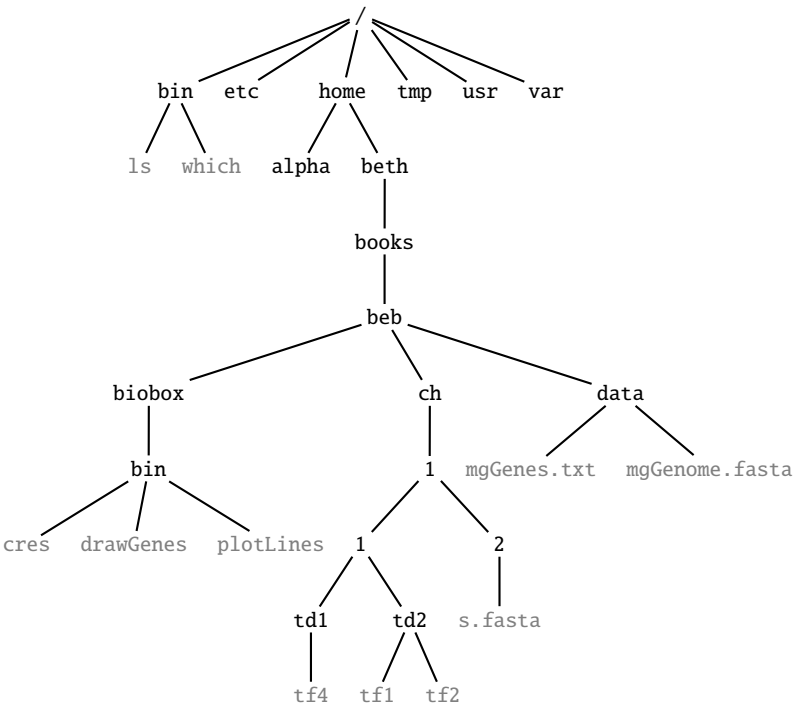


Fig. 1.1 The Unix directory tree, slightly abridged; directories black, files gray

This data is in FASTA format, which consists of one header line followed by potentially many data lines. The header line starts with *greater than*, >, and contains an optional name, *dnaN* in our case. We could try to print this with `echo`, but for multiline text `printf` is more appropriate, as it interprets `\n` as newline. How would you print the above data with `printf`?

1.52 To put the output of `printf` into a file, we redirect (>) it into the file `s.fasta`.

⟨*cli*⟩+≡
```
printf ">dnaN\nATGAAAATATTA\n" > s.fasta
```

We could have called the file anything, but the extension `fasta` reminds us that it's a FASTA file. How many bytes does `s.fasta` contain?

1.53 Look at the file we just created,

⟨*cli*⟩+≡
```
cat s.fasta
```

and think of what you see as the file's *phenotype*. What, then, is a file's *genotype*? It consists of bytes. And since one byte is eight bits, bytes are represented by octal numbers. In octal we count 1, 2, 3, 4, 5, 6, 7, 10, 11, and so on. So to get at the genotype of `s.fasta`, we generate its so-called "octal dump" with `od`,

⟨*cli*⟩+≡
```
od -b s.fasta
```
where each byte is shown as a three-digit octal number:

```
0000000 076 144 156 141 116 012 101 124 107 101 101 101 101 124 101 124
0000020 124 101 012
0000023
```

The first number in each line is the offset, followed by the actual bytes. Why do our 19 bytes amount to an offset of 23?

1.54 Codons and their amino acids are often presented as parallel sequences, for example,

```
 M   K   I   L
ATG AAA ATA TTA
```

Similarly, we can format our sequence file to show both the characters and the underlying triplet octal numbers

⟨*cli*⟩+≡
```
od -c -b s.fasta
```

```
0000000   >   d   n   a   N  \n   A   T   G   A   A   A   A   T   A   T
          076 144 156 141 116 012 101 124 107 101 101 101 101 124 101 124
0000020   T   A  \n
          124 101 012
0000023
```

What are the octal codes for `>`, `a`, `A`, and `\n`?

1.55 The character codes are summarized as the ASCII code, which can be looked up in octal, decimal, and hexadecimal notation in section 7 of the manual.

⟨*cli*⟩+≡
```
man 7 ascii
```
What is the decimal representation of `A`?

1.56 The ASCII code is a 7 bit code. How many characters does it specify?

1.57 `od` can also print the decimal representations of characters if we set the format (`-t`) to unsigned integers encoded by 1 byte (`u1`). Among other things, decimal character codes are used to encode sequencing errors, as we shall see later.

⟨*cli*⟩+≡
```
printf "A" | od -t u1
```
What is the decimal representation of `ACGT`?

1.58 We can append a second sequence to our file.

⟨*cli*⟩+≡
```
printf ">gyrB\nATGGAAGAAAAT\n" >> s.fasta
```

What happens if you redirect (>) instead of append (>>)?

1.59 How many bytes does `s.fasta` now contain?

1.60 With sequence files we are usually more interested in counting the residues rather than the bytes. To do this, we need a program from our bioinformatics toolbox, the biobox. This is contained in the `git` repository biobox, which we clone into the root of our book tree and change into.

⟨*cli*⟩+≡
```
cd $BEB
git clone sn.pub/dy6S42
cd biobox
```

To make the programs, follow the instructions at the bottom of the package page at,

`sn.pub/dy6S42`

and eventually run the program `make`.

⟨*cli*⟩+≡
```
make
```

The new programs are now listed in `progs.txt` and should all be contained in the directory `./bin`. If you get stuck, ask for help from a more experienced unixer. Does `bin` contain all the programs listed in `progs.txt`?

1.61 Table B.2 lists the biobox programs used throughout this course. Which start with `plot`?

1.62 To make the system aware of the new programs, we adjust the set of directories where the system looks for programs when it receives a command. It is stored in the variable `PATH`. What does your path currently look like?

1.63 We reset `PATH` to `$BEB/biobox/bin`, followed by all the current entries in `PATH`.

⟨*cli*⟩+≡
```
export PATH=$BEB/biobox/bin:$PATH
```

What do you get when you now enter

⟨*cli*⟩+≡
```
cres -h
```

1.64 To make the new path permanent, we follow a similar procedure we used for setting `BEB`. First we make a backup copy of the resource file, `~/.bashrc` or `~/.zshrc`, then we append (>>) the new path. However, note the single rather than double quotes in the argument of `echo`. Can you explain them?

⟨*cli*⟩+≡
```
cp ~/.bashrc .
echo 'export PATH=$BEB/biobox/bin:$PATH' >> ~/.bashrc
```

1.65 We test the new path in a new terminal window

⟨*cli*⟩+≡
```
cres -h
```

If the command isn't found, replace the resource file by its backup and try again. What do you get when you eventually run `cres` on `s.fasta`?

1.66 Let's now look at some real data, which we get from the web using `wget`. To be more specific, we get the data from the book website and place it into the root directory of our book tree.

⟨*cli*⟩+≡
```
cd $BEB
wget http://guanine.evolbio.mpg.de/beb2/data.tgz
```

If you're on a Mac, `wget` might not be installed. In that case, use your package manager for installing `wget`. If you don't have a package manager yet, install Homebrew as explained at the web site

`https://brew.sh`

and install `wget`, which is also explained there.

The extension `tgz` in `data.tgz` means the file is a "tape archive" (`t`), and has been gzipped (`gz`), or compressed. The expression "tape archive" here means that potentially many files have been concatenated into one file. Compressed files are *binary files*, which means their bytes are not all text characters as you find in the text files we ordinarily use. Count the bytes in `data.tgz`, then uncompress it with `gunzip` and count the characters in the uncompressed file. What is the compression factor achieved by `gzip`?

1.67 We use `tar` to extract (`-x`) verbosely (`-v`) the files concatenated in the archive file (`-f`) `data.tar`.

⟨*cli*⟩+≡
```
tar -x -v -f data.tar
```

What do you observe?

1.68 We don't need the `tar` file any more; can you remove it?

1.69 How many data files have we just extracted? How many of them are FASTA files?

1.70 Change back into the directory for the current Section, `$BEB/ch/1/2`, and copy the genome of *Mycoplasma genitalium* in `mgGenome.fasta` from the data directory. How long is the genome of *M. genitalium*?

1.71 Even though *M. genitalium* has one of the shortest genomes of any free-living organism, it is still too long to print usefully to screen with `cat`. Instead, we just look at its head. What do you see?

1.72 A raw genome sequence is not very enlightening, so we annotate it with the genes it encodes. The genes of *M. genitalium* are contained in `mgGenes.txt`. Copy it from `data` and look at its head. What do you see?

1.73 What does the tail of `mgGenes.txt` look like?

1.74 As we just saw when we looked at the head of `mgGenes.txt`, the first gene in the list, *dnaN*, has coordinates 686–1828 on the plus strand. Use `cutSeq` to cut out this interval from the genome and translate it with `translate`. How long is the resulting protein sequence?

1.75 The second gene in `mgGenes.txt` has coordinates 1828–2760. By how much do the first and the second genes overlap?

1.76 How many genes does *M. genitalium* have?

1.77 We could use `cat` to look at the full gene file, but then its lines would just fly across the screen. Instead, we use the pager `less`.

⟨*cli*⟩+≡
```
less mgGenes.txt
```

Try leafing through the genes. For help with navigating, press `h`. Do you notice anything about the help page?

1.78 The number of genes on the forward and reverse strands should be roughly equal. To investigate this, let's begin by counting the number of genes on the forward and reverse strands. So we use `cut` to cut the strand column, the fourth, from `mgGenes.txt`, and grab the lines containing plus with `grep`.

⟨*cli*⟩+≡
```
cut -f 4 mgGenes.txt | grep + | head
```

How many genes are on the forward strand, how many on the reverse strand? Do these two numbers add up to the total number of genes?

1.79 Do you think the discrepancy between the number of genes on the forward and reverse strands is significant? How could you find out?

1.80 From our quick glances at `mgGenes.txt` with `head` and `tail`, it looks as though genes at the beginning of the list are mainly on the forward strand, genes at the end on the reverse. Since the list is ordered according to starting position rather than strand, this is intriguing. Do genes on the 5' half of the genome really *prefer* the plus strand, those on the 3' half the minus strand? To get a visual impression of strandedness along the genome, we plot the pluses and minuses we just counted in a single line. So we delete the newline characters using another translation program, `tr` and print the result.

⟨*cli*⟩+≡
```
cut -f 4 mgGenes.txt | tr -d '\n' | awk '{print}'
```

What do you see?

1.81 We can leave out the `awk` part in the last command.

⟨*cli*⟩+≡
```
cut -f 4 mgGenes.txt | tr -d '\n'
```

What happens then?

1.82 The long line of pluses and minuses can be folded using `fold`. Can you fold it into lines length 50?

1.83 In our plot of pluses and minuses the genes look as if they are all equally long, but we know that's not the case. In order to draw the genes to scale, we convert their coordinates to small boxes using the program `drawGenes`. To see how this works, pipe two toy genes, one on the forward, the other on the reverse strand, through `drawGenes`.

⟨*cli*⟩+≡
```
printf "100 400 +\n600 1500 -\n" | drawGenes
```

What happens?

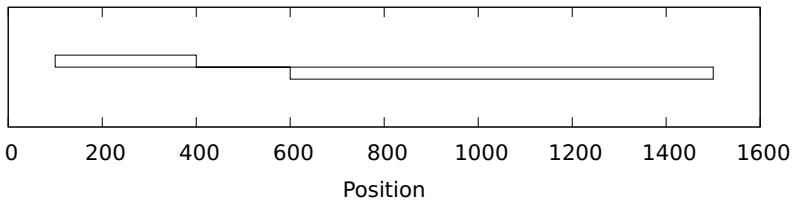


Fig. 1.2 The plot of our two toy genes

1.84 To actually draw our toy genes, we pipe the output of `drawGenes` through the program `plotLine`.

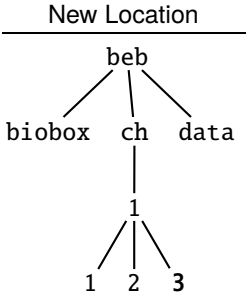
⟨*cli*⟩+≡
```
printf "100 400 +\n600 1500 -\n" | drawGenes | plotLine
```

Can you modify the call to `plotLine` to generate the graph shown in Fig. 1.2 (hint: `-h`)?

1.85 Draw the genes of *M. genitalium*. Is the 5'/3' bias in strandedness still visible?

1.3 Scripts

Scripts are short programs, often written to be thrown away like a shopping list. We look at two types of scripts, shell scripts and Awk scripts.



1.86 Can you make a new directory for the current section?

Shell

New Terms		
ampersand (&)	`for` loop	`seq`
`apt`	`histogram`	`sort`
`emacs`	remainder (%)	

1.87 In Answer 1.79 we sketched a test of the idea that 525 genes could by chance split into 299 on the forward strand and 226 on the reverse. We said this test would involve a lot of coin tossing, so we postponed it until now, when we give it to the machine. To begin with, we need a source of random numbers. Try a few times

⟨*cli*⟩+≡
```
echo $RANDOM
```

Can you use these numbers and remainder (%) to simulate coin tossing?

1.88 We'd like to repeat our coin tosses many times. In the shell, repetition is typically done with `seq`. Can you print the numbers from 1 to 3 with it (hint: `man`)?

1.89 We can capture the results of `seq` in a variable.

⟨*cli*⟩+≡
```
s=$(seq 3)
```

What does `s` contain?

1.90 We can also use `seq` to drive a `for` loop.

⟨*cli*⟩+≡
```
for a in $(seq 3); do echo $a; done
```

How would you flip a coin three times?

1.91 Our scripts are becoming a bit too unwieldy for the command line, time to put them into files. For this we need an editor. We recommend `emacs` as it strikes a nice balance between the needs of beginners and power users. Command line programs like `emacs` are distributed as packages and administered by a package manager. If you're on macOS, you might already have used Homebrew, for example, to install `wget`. The different flavors of Unix come with their own package managers. What is the package manager on your system and how can you use it to install `emacs` (google)?

1.92 You might be wondering how often we need to use our package manager. That depends. Table B.3 shows the extra programs required on Ubuntu and the corresponding packages. Table B.4 does the same for Homebrew. So at this point you can take a break from the course, and install all extra programs in one step. Or you can install individual packages whenever an unknown program crops up. How do you proceed?

1.93 The freshly installed `emacs` is either a console program that runs in the current terminal, or a graphical program that runs in its own window. Start `emacs`,

⟨*cli*⟩+≡
```
emacs
```

Did your package manager install console `emacs` or graphical `emacs`?

1.94 We open a script for coin tossing, `ct.sh`. On graphical `emacs` that's

⟨*cli*⟩+≡
```
emacs ct.sh &
```

An ampersand (`&`) after a command sends it from the default foreground to the background. So on console `emacs` just leave out the ampersand (`&`). What happens if you start graphical `emacs` without the ampersand, or console `emacs` with the ampersand?

1.95 Here is our script for coin tossing, `ct.sh`.

Prog. 1.1 (`ct.sh`)

⟨*ct.sh*⟩≡
```
for a in $(seq $1)
do
    echo $(($RANDOM % 2))
done
```

`ct.sh` is the first of 64 programs we write together throughout the book, Table B.1 lists them all. Notice in the first line of `ct.sh` the variable `$1`. This is the first value on the command line. So we can run

⟨*cli*⟩+≡
```
bash ct.sh 3
```

to toss a coin three times. How would you count the number of heads (1's) among 525 coin tosses (hint: `grep` & `wc`)?

1.96 Write a `for` loop to repeat the coin tossing experiment 20 times. Do you find a result with 299 or more heads?

1.97 Again, the script just generated is a bit unwieldy for the command line, time to return to `emacs`. It recognizes the same key combinations as the shell for navigation and editing. In fact, the commands for moving horizontally along the command line and for editing it listed in Table 1.1 are all preserved. What about the "vertical" commands for moving up and down the command history in Table 1.2?

1.98 There are a few additional `emacs` commands that make editing easier, and the best way to learn about them is to work through the `emacs` tutorial, which you can start with `C-h t`. What is the key combination for ending the tutorial?

1.99 Can you remember the key combination for ending an `emacs` session?

1.100 Write a script for iterated coin tossing, `ict.sh`. Make it read the number of iterations and the number of coin tosses per iterations from the command line.

⟨*cli*⟩+≡
```
bash ict.sh 30 525
```

Do you find 299 heads or more in 30 iterations?

1.101 It's a bit tedious to compare random numbers with 299, so we draw a histogram instead. Fig. 1.3 shows the results of 1000 iterations and we see that 299 is on the extreme right of this distribution. Can you reproduce it with `histogram` and `plotLine`?

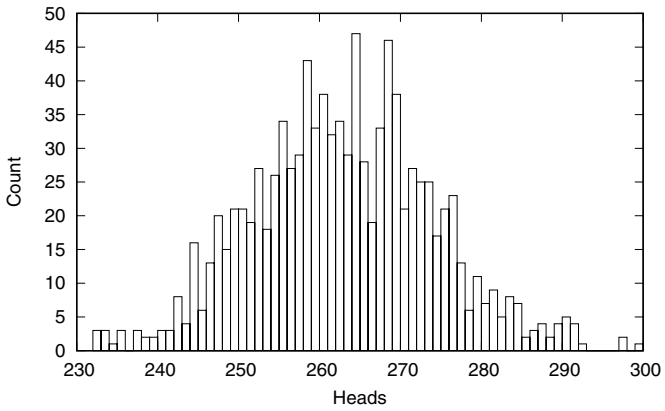


Fig. 1.3 Histogram of 1000 experiments tossing a coin 525 times

1.102 For actually counting the experiments with at least 299 heads, numerical sorting again helps.

⟨*cli*⟩+≡
```
bash ict.sh 100 525 | sort -n | tail
```

In case you're wondering what happens with default `sort`, try

⟨*cli*⟩+≡
```
printf "2\n10\n" | sort -n
```

and omit `-n`. What do you observe?

1.103 How often do you find 299 or more heads in 100, 1000, and 10,000 trials? Are the genes in *M. genitalium* distributed randomly between the forward and the reverse strands?

1.104 Next, we investigate the bias in strandedness along the genome. We already saw this in our earlier plots of pluses and minuses, but now we'd like to test the null hypothesis of no such bias. Can you think of a way to do this (no programming, just thinking)?

1.105 Copy the file `mgGenes.txt` from `data` to your current directory. How many genes are on the plus strand among the first $525/2 \approx 263$ genes?

1.106 The `-R` option of `sort` randomizes the result. What do you observe when you apply it to the strand column in `mgGenes.txt`?

1.107 The option `-n` adds line numbers to the output of `cat`. What do you observe when you add line numbers to the strand column, followed by `sort -R`?

1.108 Write a script that shuffles the strands and counts the number of pluses in the first half of the *M. genitalium* genome. Call the script `ss.sh`, for *shuffle strand*, and pass the number of iterations via the command line. Run it a few times—do you ever find the original number of pluses or more by chance alone?

1.109 Run `ss.sh` with 1000 iterations and save the results in `ss.dat`. Then plot the corresponding histogram and make the x range large enough to include the observed number of forward genes. We can even mark the observed number of forward genes by drawing an arrow with the `-g` option of `plotLine`. This allows us to pass arbitrary code to the program `gnuplot`, which does the actual plotting for `plotLine`.

```
-g "set arrow from x1,y1 to x2,y2"
```

How significant is the observed bias in strandedness?

Awk

New Terms

action	`END` block	regular expression
`BEGIN` block	literate programming	variance
`diff`	pattern	

1.110 Instead of directly programming the shell, we can use a programming language that integrates well with the shell. Awk has traditionally been that language. Like in the shell, `$1`, `$2`, and so on refer to positions in input, but with a twist: What do you observe when you enter

⟨*cli*⟩+≡
```
awk '{print $2, $3}' mgGenes.txt | head
```

1.111 What happens when you leave out the argument of `print`?

⟨*cli*⟩+≡
```
awk '{print}' mgGenes.txt | head
```

1.112 To print the lengths of genes, we can subtract the start from the end and add one.

⟨*cli*⟩+≡
```
awk '{print $3 - $2 + 1}' mgGenes.txt | head
```

Can you plot a histogram of gene lengths?

1.113 Use Awk to print gene length, accession, and name. What is the shortest gene in *M. genitalium*? The longest?

1.114 We can also sum the gene lengths and print the result after the last line of input has been read.

⟨*cli*⟩+≡
```
awk '{s = s + $3 - $2 + 1}END{print s}' mgGenes.txt
```

Which fraction of the genome is covered by genes?

1.115 What is the average gene length?

1.116 Apart from the average gene length, we'd also like to calculate its variance, the average squared difference from the average. If we have n lengths, ℓ_i, their variance is

$$s_\ell^2 = \frac{1}{n-1} \sum_{i=1}^{n} \left(\ell_i - \overline{\ell}\right)^2, \tag{1.1}$$

where $\overline{\ell}$ is the average length. According to equation (1.1), we first calculate the average length, then the variance, which means we have to store the lengths. In our next program, `varLen.awk`, we do this using the array `lengths`.

Prog. 1.2 (`varLen.awk`)

⟨*varLen.awk*⟩≡
```
BEGIN {
  n = 0
}
{
  l = $3 - $2 + 1
  lengths[n] = l
```

```
    n++
  }
  END {
    ⟨Calculate variance of gene lengths, Prog. 1.2⟩
  }
```

This is a good point to explain our notation for code. We write it in *chunks*. A chunk is referred to by its name, here ⟨*varLen.awk*⟩. This is followed by (≡) if the chunk is defined or (+≡) if it is appended to. Chunks can contain chunks, here ⟨*varLen.awk*⟩ contains ⟨*Calculate variance of gene lengths, Prog. 1.2*⟩. This chunk is filled in later. For the time being, you can mark it as a comment in your code.

```
# Calculate variance of gene lengths.
```

In the book we disambiguate code chunks by program numbers, but this is not necessary when you write your own code. This style of chunk-wise programming in the context of plenty of ordinary prose was invented in the early 1980's by Donald Knuth, who calls it *literate programming* [27, ch. 4].

We begin to fill in the chunk ⟨*Calculate variance of gene lengths, Prog. 1.2*⟩. Can you print the first three entries of the array `lengths`?

1.117 What happens if we omit the `BEGIN` block?

1.118 The standard method to iterate over the entries of an array is a `for` loop.

```
for (i = 0; i < n; i++) {
  ...
}
```

The first argument of `for`, the initialization, `i = 0`, is executed once. Then the test in the second argument, `i < n`, is carried out. If it is true, the action block is executed. Then the third argument is executed by incrementing the running variable, `i++`. At this point the little waltz test-action-count is repeated, until the test fails. If the block consists of a single line, the curly brackets can be omitted, for example

```
for (i = 0; i < n; i++)
  print lengths[i]
```

We continue working on our chunk ⟨*Calcultate variance of gene lengths*⟩ and calculate the average gene length, which we save to the variable `avg`. Can you do that and print `avg`, just to make sure?

1.119 What is the variance of gene lengths in *M. genitalium*?

1.120 The opposite of the `END` block we just worked on is the `BEGIN` block, which we've also seen already. It is executed *before* any input is read. For example, Awk gives us another handy calculator.

⟨*cli*⟩+≡

```
  awk 'BEGIN{print 540447/580076}'
```

Can it calculate 2^{64}?

1.121 `BEGIN` and `END` are examples of *patterns*, one of the two building blocks of Awk programs. The second are *actions*. So Awk programs have the structure

```
pattern {action}
pattern {action}
...
```

Whenever a pattern is true, the action is applied to the current line of input. As we've already seen in programs like

⟨*cli*⟩+≡
```
awk '{print $1}' mgGenes.txt | head
```

when we leave out the pattern, the action is applied to every input line. We can also leave out the action, for example,

⟨*cli*⟩+≡
```
awk '/\+/' mgGenes.txt | head
```

prints lines with a plus, so the default action is `print $0`. The two slashes enclose a regular expression, a notation for sets of strings. In the regular expression library used by some Awk implementations, + is a quantifier. To ensure its literal meaning, we prefix the plus with a backslash. What happens when we match (`~`) plus in the fourth column without any action?

⟨*cli*⟩+≡
```
awk '$4 ~ /\+/' mgGenes.txt | head
```

1.122 Given that minus isn't part of the regular expression syntax, how would you print all the genes on the minus strand?

1.123 You might be wondering whether it's really necessary to specify the fourth column for matching; why not match the whole line? But it turns out there is a difference between matching just the fourth column and matching the whole line.

⟨*cli*⟩+≡
```
awk '$4 ~ /-/' mgGenes.txt > minus1.txt
awk '/-/' mgGenes.txt > minus2.txt
```

The program `diff` compares two files line by line. What is the difference between `minus1.txt` and `minus2.txt`?

1.124 The number of lines, or number of records (`NR`), is a built-in Awk variable. So

⟨*cli*⟩+≡
```
awk 'END{print NR}' mgGenes.txt
```

prints the number of lines in the input. Copy the file `mgGenome.fasta` to your current directory. How many lines does it contain? Compare your result with `wc -l`.

1.125 The number of fields (`NF`) is also a built-in Awk variable, `NF`. Can you print the last field of every line in `mgGenes.txt`?

1.126 Some genes have names, others don't. The genes with names have lines with five fields, the others lines with four. So by filtering for lines with five fields we filter for the genes with names.

⟨*cli*⟩+≡
```
awk 'NF == 5' mgGenes.txt | head
```

What percentage of genes has names?

1.127 FASTA files consist of one or more headers and data. We can filter for the headers by looking for lines with >.

⟨*cli*⟩+≡
```
awk '/>/' mgGenome.fasta
```

Here the > can appear anywhere in the line. But header markers only appear at the beginning of lines. The regular expression for the beginning of a string is ^, so we can write

⟨*cli*⟩+≡
```
awk '/^>/' mgGenome.fasta
```

The file `mgProteome.fasta` contains the proteome of *M. genitalium*. Copy it to your directory. How many entries does it contain?

1.128 Instead of filtering for headers, we can filter for data by negating (`!`) the match for header.

⟨*cli*⟩+≡
```
awk '!/^>/' mgGenome.fasta | head
```

The function `length` gives the length of a string.

⟨*cli*⟩+≡
```
printf "ACGT\n" | awk '{print length($1)}'
```

Can you calculate the genome length of *M. genitalium*?

1.129 We've already seen that the sequence in `mgGenome.fasta` consists of over 8000 lines. How can we convert this into one contiguous sequence inside `awk`? This is called *concatenation*. It is done by writing two strings next to each other.

⟨*cli*⟩+≡
```
head -n 3 mgGenome.fasta | awk '{t = t $1}END{print t}'
```

Can you avoid including the first field of the header?

1.130 The function `split` splits a string at a delimiter into an array, `a`, of substrings, which contains the first substring at `a[1]`. `split` returns the length of the array.

⟨*cli*⟩+≡
```
printf "axb\n" |
    awk '{n=split($1,a,"x");for(i=1;i<=n;i++)print a[i]}'
```

```
a
b
```

What happens when you split at `a` or `b` instead of `x`?

1.131 We often try to avoid empty lines in output. But how do we recognize empty lines? In an empty line the beginning is directly followed by the end. We've already seen the regular expression for the beginning of a string, `^`. The regular expression for the end is `$`. Can you filter out the empty lines generated in Problem 1.130?

1.132 What happens when you use the empty string as delimiter in `split`?

1.133 The first token in the header of `mgGenome.fasta` has the form

```
>a|b|c|d|
```

How would you split it into a, b, c, and d? Beware that a command like

```
grep > mgGenome.fasta
```

would be interpreted as a redirect and overwrite `mgGenome.fasta`. To get a verbatim >, place it in single quotes, `'>'`.

1.134 We've already seen integers as indexes for arrays. But strings can also index arrays, which is handy for counting strings.

```
printf "A\nA\nA\nC\nC\nT\n" | awk '{counts[$1]++}'
```

To iterate over an array with an unknown set of indexes, we use a version of the `for` loop that visits all of them.

```
for (i in array)
  print i, array[i]
```

Can you print the nucleotide counts we just generated?

1.135 Let's combine some of the Awk tricks we've seen so far and write a program for counting residues, `cres.awk`. First, we count the residues in each line. Then we print the total residue count followed by the count of each individual residue.

Prog. 1.3 (`cres.awk`)

```
⟨cres.awk⟩≡
  !/^>/ {
    ⟨Count residues, Prog. 1.3⟩
  }
  END {
    ⟨Print total residue count, Prog. 1.3⟩
    ⟨Print residue counts, Prog. 1.3⟩
  }
```

We now fill in the three chunks of `cres.awk`, one chunk per Problem. Can you first count the residues?

1.136 Can you print the total residue count?

1.137 So far, we've used the `print` command for printing. This is like the shell command `echo`. If we'd like more control over the formatting, we can use `printf`, in the shell and in Awk, too. For example, we can print a floating point number with `%f` and a tab with `\t`.

⟨*cli*⟩+≡

```
awk 'BEGIN{r=22/7; printf "22/7\t%f\n", 22/7}'
```

To learn more about `printf`, take a look at the `man` page for `awk` and search for *The printf Statement*. Can you print a table of the count and frequency of each residue?

1.138 What are the nucleotide frequencies in the genome of *M. genitalium*? Is there anything unusual about them?

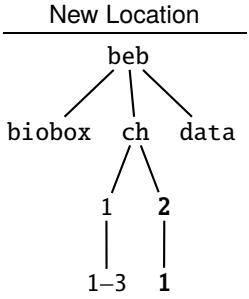
1.139 Repeat this analysis for the proteome. What is the least frequent amino acid? The most frequent? To answer these questions, it helps to cut off the first two lines of the output. Read the `tail` `man` page to see how to use `-n` for this. Also, it's helpful to sort by the third column, rather than by the default first. The `man` page for `sort` explains under `-k`, how.

Chapter 2
Optimal Alignment

When we align two sequences, we in fact propose an evolutionary history for them. A history, where we account for three kinds of events, mutation, insertion, and deletion. These events are not all equally important, which we express by giving them individual scores summarized in a score scheme. Given such a scheme, we can look for the most likely history of two sequences, their optimal alignment.

2.1 Keeping Score

When scoring alignments, we aim to give high scores to *true* alignments, which means, alignments that reflect the actual evolutionary history of their sequences. The overall score of an alignment is composed of the score for residue pairs and the gap score. Historically, more attention has been paid to scoring residues than gaps. And among the residues it is particularly the amino acids whose evolution has been studied in connection with scoring alignments. We begin by looking at how alignments relate to evolution, then investigate the relationship between amino acids and the genetic code, before we study how the evolution of amino acids is simulated to obtain scores for pairs of amino acids.



2.1 Can you make a new directory for this section and change into it?

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_2

Alignments and Evolution

New Terms		
allele	gap opening	open reading frame (ORF)
`al`	homology	overlap alignment
coding sequence (CDS)	indel	synonymous mutation
gap extension	`keyMat`	

2.2 Consider a short example sequence, S = `ACCGT`, which is passed from parent to child to grandchild, and so on. If replication were perfect, nothing would ever change. However, we only need to look at the delightful diversity around us to remind ourselves that mutations do occur. Say, the `G` at position 4 in our example sequence changes into a `C`. Now the ancestral sequence has split into two versions, or alleles, which we visualize in Fig. 2.1A.

The alignment at the bottom of Fig. 2.1A summarizes this scenario by writing nucleotides with a common ancestor on top of each other. Such nucleotides are called "homologous". Use the program `al` to align the two example sequences. Can you recapitulate the alignment score you get (hint: `al -h`)?

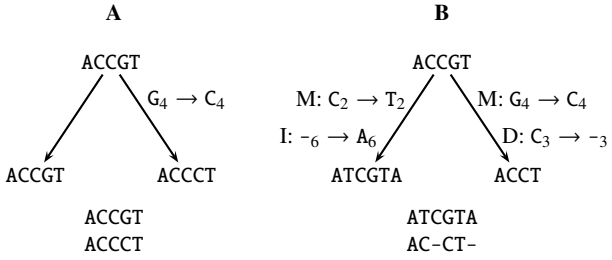


Fig. 2.1 Evolution of one sequence into two with one mutation (**A**), and with two mutations (*M*), an insertion (*I*), and a deletion (*D*) (**B**); the evolutionary histories are followed by the true alignment

2.3 Fig. 2.1B shows another evolutionary history and the corresponding alignment. This one contains not only mutation, but also insertion and deletion. Use `al` to align the sequences. Do you get the true alignment?

2.4 An insertion in one sequence implies a deletion in its partner. Since the two events cannot be distinguished as long as we only align two sequences, insertions and deletions are often collectively called *indels*. Indels are denoted by gaps. Gaps of length l are traditionally scored as

$$g(l) = g_o + (l - 1)g_e,$$

where g_o is the score for opening a gap and g_e for extending it. This means a gap of length 1 has score g_o. A popular alternative gap scoring scheme is to count every

gap position as an extension,

$$g(l) = g_o + l g_e.$$

Which gap-scoring scheme is implemented in `al`?

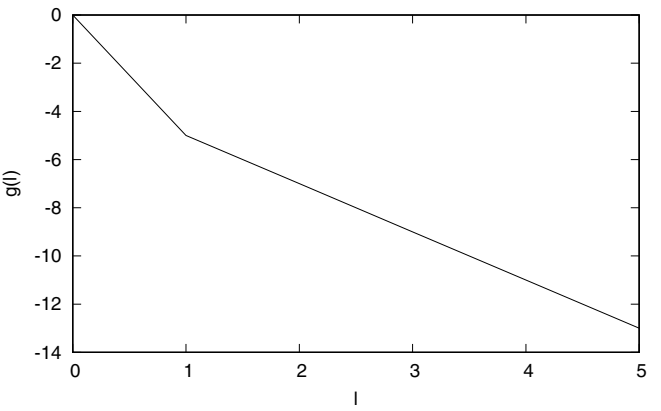


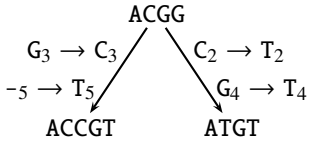
Fig. 2.2 Gap score, $g(l)$ as a function of gap length, l

2.5 Write an Awk program, `gapScore.awk`, to print l and $g(l)$ for gaps of lengths 0 to 5 with $g_o = -5$ and $g_e = -2$. Fig. 2.2 shows the resulting plot with `plotLine`. Can you reproduce it?

2.6 With few exceptions, we can only observe contemporary sequences, while ancestral sequences remain unknown. Given two contemporary sequences, S_1 = `ACCGT` and S_2 = `ATGT`, we wish to infer their evolutionary history by aligning them. One possible alignment is

```
ACCGT
ATGT-
```

Here is an evolutionary scenario compatible with that alignment. It consists of three mutations and one insertion:



Draw an alternative evolutionary scenario leading to S_1 and S_2.

2.7 Consider the two sequences S_5 = `ACAGTTC` and S_6 = `AGTTC`. Without much thinking, write down what seems to you their most natural alignment. Then align them with `al`. What do you observe?

2.8 There is an alignment method where flanking gaps always have score zero called *overlap alignment*. Use `al` in overlap mode to align S_5 and S_6 again.

2.9 Instead of playing with toy sequences, we now align two real sequences contained in `hbb1.fasta` and `hbb2.fasta`. Copy these files from `$BEB/data` to your current directory. How long are the sequences they contain and what are their functions?

2.10 Align the two β-globin mRNA sequences using `al`. Where do they differ?

2.11 What is the position of the mutation in the two sequences (hint: `-L`)?

2.12 The mutation might be in the 5' untranslated region (UTR) or inside the coding sequence (CDS). To find out, use the program `translate` to translate `hbb1.fasta`. Proteins start with a start codon, `ATG`, encoding methionine, `M`, and end with a stop codon, denoted by an asterisk `*`. `translate` takes the reading frame as argument. Which reading frame gives the longest uninterrupted protein sequence, also called the longest *open reading frame*, ORF?

2.13 The program `keyMat` looks up patterns in a sequence, for example all stop codons in the translation of `hbb1.fasta`.

⟨*cli*⟩+≡
```
translate -f 3 hbb1.fasta | keyMat '*'
```
What is the start and end position of the human β-globin in the translation?

2.14 Is the mutation we observed in the CDS?

2.15 In the following steps we find out, whether the mutation leads to an amino acid change or not. Use the program `cutSeq` to cut out the human β-globin and save it in `hbb1p.fasta`.

2.16 Cut out the chimp β-globin and save it in `hbb2p.fasta`.

2.17 Next we'd like to align the β-globin sequences from human and chimp. We've already seen that for nucleotide sequences we only distinguish between matches and mismatches. For proteins we make finer distinctions, which are summarized in matrices of scores for pairs of amino acids. One of these is PAM70, shown in Fig. 2.3. PAM70 is symmetrical; what does that mean?

2.18 PAM70 is contained in the file `pam70.txt`, which is part of the Blast software package. Copy this from the data directory to your current directory. How many entries does `pam70.txt` contain? Compare that to the 20 standard amino acids. What are the extra entries in `pam70.txt`?

2.19 Write an Awk script to print the match scores for the canonical 20. What are the smallest and largest match scores?

2.20 Edit your script to print out only the mismatch scores. Again, what are the smallest and largest values? Use the difference between these values as the number of bins for a histogram of mismatch values drawn with `histogram` and `plotLine`.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	5	-4	-2	-1	-4	-2	-1	0	-4	-2	-4	-4	-3	-6	0	1	1	-9	-5	-1
R	-4	8	-3	-6	-5	0	-5	-6	0	-3	-6	2	-2	-7	-2	-1	-4	0	-7	-5
N	-2	-3	6	3	-7	-1	0	-1	1	-3	-5	0	-5	-6	-3	1	0	-6	-3	-5
D	-1	-6	3	6	-9	0	3	-1	-1	-5	-8	-2	-7	-10	-4	-1	-2	-10	-7	-5
C	-4	-5	-7	-9	9	-9	-9	-6	-5	-4	-10	-9	-9	-8	-5	-1	-5	-11	-2	-4
Q	-2	0	-1	0	-9	7	2	-4	2	-5	-3	-1	-2	-9	-1	-3	-3	-8	-8	-4
E	-1	-5	0	3	-9	2	6	-2	-2	-4	-6	-2	-4	-9	-3	-2	-3	-11	-6	-4
G	0	-6	-1	-1	-6	-4	-2	6	-6	-6	-7	-5	-6	-7	-3	0	-3	-10	-9	-3
H	-4	0	1	-1	-5	2	-2	-6	8	-6	-4	-3	-6	-4	-2	-3	-4	-5	-1	-4
I	-2	-3	-3	-5	-4	-5	-4	-6	-6	7	1	-4	1	0	-5	-4	-1	-9	-4	3
L	-4	-6	-5	-8	-10	-3	-6	-7	-4	1	6	-5	2	-1	-5	-6	-4	-4	-4	0
K	-4	2	0	-2	-9	-1	-2	-5	-3	-4	-5	6	0	-9	-4	-2	-1	-7	-7	-6
M	-3	-2	-5	-7	-9	-2	-4	-6	-6	1	2	0	10	-2	-5	-3	-2	-8	-7	0
F	-6	-7	-6	-10	-8	-9	-9	-7	-4	0	-1	-9	-2	8	-7	-4	-6	-2	4	-5
P	0	-2	-3	-4	-5	-1	-3	-3	-2	-5	-5	-4	-5	-7	7	0	-2	-9	-9	-3
S	1	-1	1	-1	-1	-3	-2	0	-3	-4	-6	-2	-3	-4	0	5	2	-3	-5	-3
T	1	-4	0	-2	-5	-3	-3	-3	-4	-1	-4	-1	-2	-6	-2	2	6	-8	-4	-1
W	-9	0	-6	-10	-11	-8	-11	-10	-5	-9	-4	-7	-8	-2	-9	-3	-8	13	-3	-10
Y	-5	-7	-3	-7	-2	-8	-6	-9	-1	-4	-4	-7	-7	4	-9	-5	-4	-3	9	-5
V	-1	-5	-5	-5	-4	-4	-4	-3	-4	3	0	-6	0	-5	-3	-3	-1	-10	-5	6

Fig. 2.3 PAM70 amino acid score matrix; match scores are shown in red

2.21 Align the two β-globins using PAM70. Is there an amino acid difference between human and chimp β-globin?

Mutations and the Genetic Code

New Terms		
`geco`	polarity	random genetic code
mutational space		

2.22 Fig. 2.4 shows the genetic code, which maps the $4^3 = 64$ codons to 20 amino acids. The first codon position is occupied by the nucleotides in the first column of the code table, the second codon position by the nucleotides in the top row, and the third codon position by the nucleotides in the last column. What is the smallest and the largest number of codons encoding the same amino acid?

2.23 Mutations at the third codon position are often synonymous, leading to the blocks of four identical amino acids in the genetic code, which correspond to given first and second codon positions. Are there also synonymous mutations at these two positions?

2.24 Amino acids are separated by one, two, or three nucleotide mutations. Can you find an example for each starting with phenylalanine (`F`)?

First Position	Second Position: T	C	A	G	Third Position
T	Phe/F	Ser/S	Tyr/Y	Cys/C	T
	Phe/F	Ser/S	Tyr/Y	Cys/C	C
	Leu/L	Ser/S	Ter/*	Ter/*	A
	Leu/L	Ser/S	Ter/*	Trp/W	G
C	Leu/L	Pro/P	His/H	Arg/R	T
	Leu/L	Pro/P	His/H	Arg/R	C
	Leu/L	Pro/P	Gln/Q	Arg/R	A
	Leu/L	Pro/P	Gln/Q	Arg/R	G
A	Ile/I	Thr/T	Asn/N	Ser/S	T
	Ile/I	Thr/T	Asn/N	Ser/S	C
	Ile/I	Thr/T	Lys/K	Arg/R	A
	Met/M	Thr/T	Lys/K	Arg/R	G
G	Val/V	Ala/A	Asp/D	Gly/G	T
	Val/V	Ala/A	Asp/D	Gly/G	C
	Val/V	Ala/A	Glu/E	Gly/G	A
	Val/V	Ala/A	Glu/E	Gly/G	G

Fig. 2.4 The genetic code with three-letter and single-letter amino acid designations

2.25 Amino acids are not just separated by one, two, or three nucleotide mutations, they also differ greatly in their chemical properties. One of these properties is polarity and Fig. 2.5 shows the structures of the 20 amino acids color-coded and ordered by polarity. What is the least polar amino acid? The most polar?

2.26 The polarities of the amino acids shown in Fig. 2.5 are listed in the file `polarity.dat` [18]. Copy it to your current directory. What is the average polarity of amino acids (`awk`)?

2.27 Next, we plot the polarities.

⟨*cli*⟩+≡
```
awk '!/^#/{print $2, 0, $1}' polarity.dat | sort -n |
    plotLine -Y -0.2:2 -u y -P -x Polarity
```

Try running this code. Can you explain what it does?

2.28 Fig. 2.6A shows the genetic code with the amino acids color-coded by polarity. It looks as though amino acids are clustered in blocks of similar color. To test this, we shuffle the amino acids among the codons, only the stop codon is left unchanged. Fig. 2.6B shows one such random code. Do you think its color distribution is less homogeneous than that of the natural genetic code?

2.29 Instead of comparing the two codes in Fig. 2.6 by eye, we can quantify the effect of mutations on them. We do this by calculating the mean squared difference in polarity between the amino acids in the given genetic code and the amino acids reached by all of its one-step mutations [19]. Let's call this quantity d. The program `geco` can either iterate to compute d for multiple random codes, or, without iterations, compute d for the natural code. What is d for the natural code?

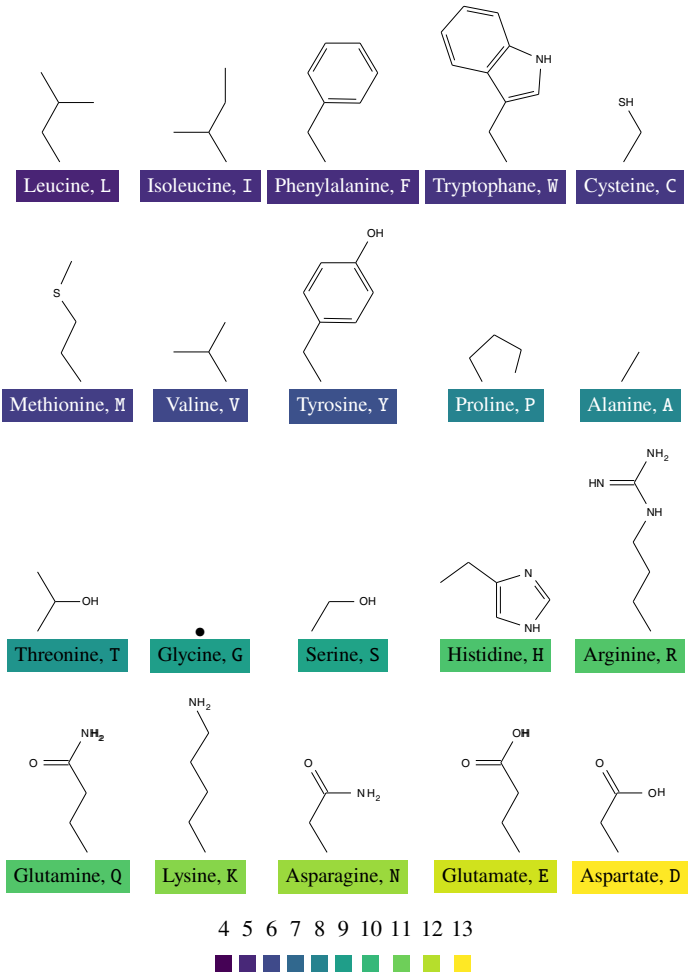


Fig. 2.5 The side chains of the 20 amino acids specified by the genetic code color-coded by their polarity, which ranges from 4.9 to 13. Glycine is merely bound to a hydrogen atom, the dot, •

2.30 To find out whether the d value of the natural code is different from that of random codes, we might want to look at all possible codes and calculate d for each one. The number of possible codes is the same as the number of possible arrangements of 20 books on a shelf, 20 factorial,

$$20! = 2 \times 3 \times \ldots \times 20$$

Can you use Awk to calculate 20!?

A

	T	C	A	G	
T	Phe/F	Ser/S	Tyr/Y	Cys/C	T
	Phe/F	Ser/S	Tyr/Y	Cys/C	C
	Leu/L	Ser/S	Ter/*	Ter/*	A
	Leu/L	Ser/S	Ter/*	Trp/W	G
C	Leu/L	Pro/P	His/H	Arg/R	T
	Leu/L	Pro/P	His/H	Arg/R	C
	Leu/L	Pro/P	Gln/Q	Arg/R	A
	Leu/L	Pro/P	Gln/Q	Arg/R	G
A	Ile/I	Thr/T	Asn/N	Ser/S	T
	Ile/I	Thr/T	Asn/N	Ser/S	C
	Ile/I	Thr/T	Lys/K	Arg/R	A
	Met/M	Thr/T	Lys/K	Arg/R	G
G	Val/V	Ala/A	Asp/D	Gly/G	T
	Val/V	Ala/A	Asp/D	Gly/G	C
	Val/V	Ala/A	Glu/E	Gly/G	A
	Val/V	Ala/A	Glu/E	Gly/G	G

B

	T	C	A	G	
T	Ser/S	Val/V	Pro/P	Met/M	T
	Ser/S	Val/V	Pro/P	Met/M	C
	Ala/A	Val/V	Ter/*	Ter/*	A
	Ala/A	Val/V	Ter/*	Thr/T	G
C	Ala/A	Arg/R	Tyr/Y	Cys/C	T
	Ala/A	Arg/R	Tyr/Y	Cys/C	C
	Ala/A	Arg/R	His/H	Cys/C	A
	Ala/A	Arg/R	His/H	Cys/C	G
A	Lys/K	Gly/G	Leu/L	Val/V	T
	Lys/K	Gly/G	Leu/L	Val/V	C
	Lys/K	Gly/G	Asp/D	Cys/C	A
	Phe/F	Gly/G	Asp/D	Cys/C	G
G	Trp/W	Glu/E	Asn/N	Ile/I	T
	Trp/W	Glu/E	Asn/N	Ile/I	C
	Trp/W	Glu/E	Gln/Q	Ile/I	A
	Trp/W	Glu/E	Gln/Q	Ile/I	G

4 5 6 7 8 9 10 11 12 13

Fig. 2.6 The standard genetic code (**A**) and a randomized version (**B**) color-coded by polarity

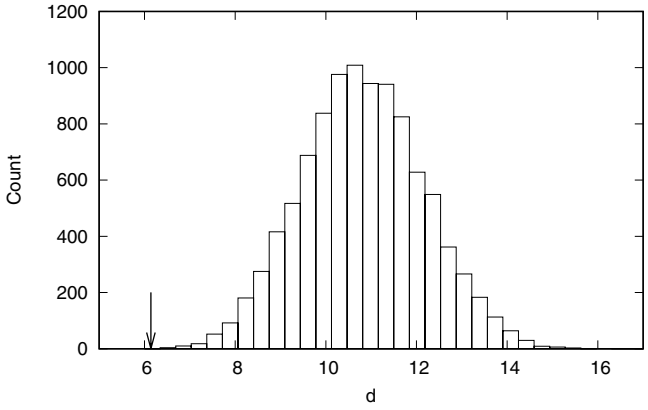


Fig. 2.7 Distribution of the d values for polarity from 10^4 random genetic codes, the arrow marks the polarity of the natural genetic code

2.31 Fig. 2.7 shows the d values from 10^4 random genetic codes generated with `geco`. Can you reproduce it? Is the d value of the natural code unusual?

2.32 Let's formally test our impression that the genetic code evolved to minimize the effect of mutations on polarity. So we calculate the frequency with which a random code appears that has a d value less or equal to that of the natural code. What is the error probability when rejecting the null hypothesis that the structure of the natural code is random?

2.33 Apart form polarity, amino acids also differ according to hydropathy, volume, and charge. These quantities are stored in the files `hydropathy.dat`, `volume.dat`, and `charge.dat`. Is the genetic code optimized with respect to them, too?

Scoring Amino Acids using PAM Matrices

New Terms		
background frequencies	odds ratio	`paste`
divergence time	`pam`	percent accepted mutations
`getSeq`		

2.34 When scoring amino acids, time is taken into account. The more time elapses, the more likely an amino acid has mutated. Time is itself measured in mutations, the unit being *percent accepted mutations*, or PAM [9]. To get an idea what 1% protein divergence might mean in years, let's look at two homologous proteins, one from human (P49792), the other from our closest extant relative, the chimp (H2QII6). Use `getSeq` and the accessions to obtain the two sequences from `uniprot_sprot.fasta` and align them with `al` and PAM70. What is the percent mismatch between them?

2.35 To get the PAM scoring system off the ground, Margaret Dayhoff and her colleagues looked up the mutation probabilities for pairs of amino acids in multiple sequence alignments and normalized them to 1% mismatch [9]. Their results are given in file `pam1.txt`. Copy this to your working directory, `$BEB/ch/2/1`. An entry m_{ij} in `pam1.txt` indicates the probability that the amino acid in column j has mutated into the amino acid in row i after 1 PAM has elapsed. What is the mutation probability of alanine to serine? Serine to alanine?

2.36 On the main diagonal of `pam1.txt`, we find the match probabilities after 1 PAM. What is the average match probability (hint: `tail`)? Can you make sense of the result?

2.37 The program `pam` can multiply the PAM1 matrix n times with itself to simulate amino acid evolution over the time interval PAMn. Calculate the percent mismatch between protein sequences for $n = 2, 5, 10, 20, 50, 100$. What do you observe?

2.38 Instead of calculating individual values for percent mismatch, we can loop over the values we are interested in and plot the result. Write the script `pm.sh` that reads

n from the command line and prints the percent mismatch as a function of PAM*n*. Fig. 2.8 shows the resulting plot. Can you reproduce it? If you find `pm.sh` slow, try incrementing the `for` loop by two (or more) using a version of `seq` that includes the increment, for example,

⟨*cli*⟩+≡
```
seq 1 2 10
```

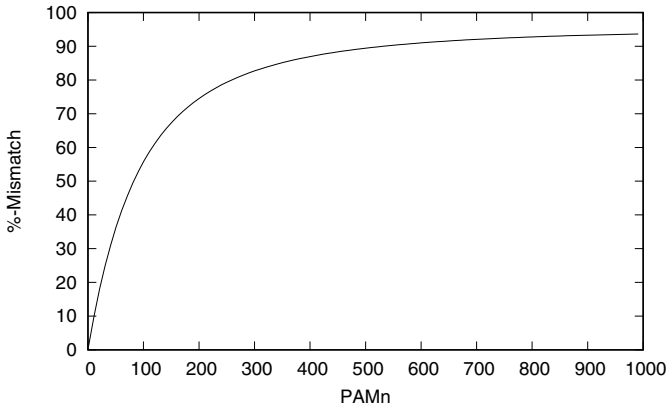


Fig. 2.8 The percent mismatch as a function of the percent accepted mutations, PAM*n*

2.39 The file `aa.txt` contains the background frequencies of amino acids [9]. Copy it to your working directory. What is the least frequent amino acid? The most frequent?

2.40 We've already seen that the match probability is on the main diagonal of the probability matrix. Save these entries for $n = 70$ to the file `matchProb.txt`. The ratio of match probabilities to background frequencies is a measure of amino acid conservation. What is the most conserved amino acid (hint: `paste`)? Take a look at the amino acid side chains in Fig. 2.5. Can you make sense of your result?

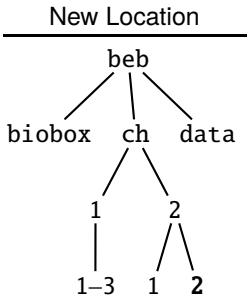
2.41 Division by the background frequency can also be applied to the whole probability matrix using `pam`. Can you recapitulate the probability ratios we just calculated to measure conservation?

2.42 The probability ratios just computed are also called *odds ratios*. These are usually expressed as logarithm base two, thus giving an information measure in bits. The final match score is traditionally expressed in half bits rounded to the nearest integer. What is the match score of the most conserved amino acid for PAM70?

2.43 The logarithm can be applied to all odds ratios using `pam`. This gives the final PAM70 score matrix we already saw in Fig. 2.3. What happens to the score of the most conserved amino acid when you double the divergence time to PAM140?

2.2 Construction

Given that we now know how to score an alignment, we might be tempted to pick the best one from all possible alignments. But as we shall see, the number of possible alignments is very large indeed, and we spend some time calculating exactly how large. So instead of going through all possible alignments, we could draw a rectangle with the query sequence along the vertical axis, the subject sequence along the horizontal, and mark the matching parts. Such dot plots are already quite useful, but it turns out that calculating the full alignment is very similar to calculating the number of possible alignments.



2.44 Can you make a new directory for this section and change into it?

The Number of Possible Alignments

New Terms

bottom up solution	`numAl`	recursion
`dot`	programming matrix	top down solution

2.45 Given two sequences of a single nucleotide, S_1 = `A` and S_2 = `T`, how many possible alignments are there?

2.46 Let $f(m, n)$ be the number of possible alignments of two sequences of lengths m and n. We've seen that their alignment can end in three possible ways: residue/gap, gap/residue, and residue/residue. A residue reduces the remaining sequence by one, a gap doesn't. We can thus write the number of possible alignments as the sum of the number of alignments when we exclude the last column,

$$f(m,n) = f(m-1,n) + f(m,n-1) + f(m-1,n-1). \tag{2.1}$$

If we apply this to our example of two sequences length 1, we get

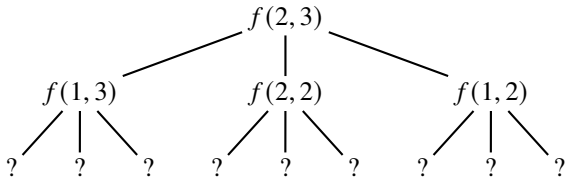
$$f(1,1) = f(0,1) + f(1,0) + f(0,0)$$

We could go on applying equation (2.1), but we shouldn't, as there are no sequences of negative length. Moreover, there is only one way to align a sequence of any length to a "null" sequence consisting only of gaps. In other words,

$$f(i,0) = f(0,j) = f(0,0) = 1. \tag{2.2}$$

What is the number of possible alignments of two sequences of lengths 1 and 2?

2.47 An equation that refers to itself, like equation (2.1), is called a *recursion*. Recursions have a close connection to one of our favorite structures in evolutionary biology, trees. If, for example, we wanted to know the number of alignments of two sequences of lengths 2 and 3, we can draw a tree with $f(2,3)$ as its root, $f(1,3)$, $f(2,2)$, and $f(1,2)$, as its children, and so on.



Can you complete the tree? What is the number of possible alignments for two sequences lengths 2 and 3?

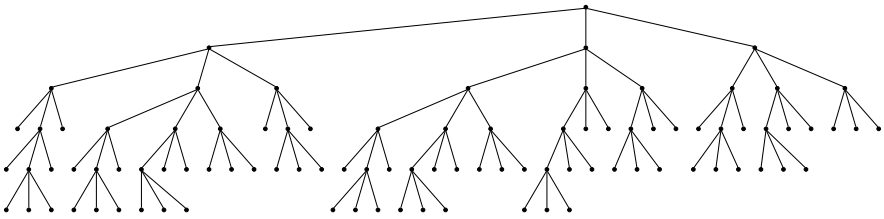


Fig. 2.9 The recursion tree for computing the number of possible alignments of two sequences length 3

2.48 Fig. 2.9 shows the recursion tree tree for two sequences of length 3. How many alignments are there?

2.49 The direct approach to solving equation 2.1 is also called *top down*, as we walk from the root at the top of the recursion tree down to its leaves. This is a bit tedious,

so we get the computer to do it for us. Fig. 2.9 was drawn using the program `numAl` in top down mode (`-t`), printing the tree (`-p`), and rendering it with `dot`. `dot` prints to the X11 terminal (`-T`); if this doesn't work on your computer, you can print to PNG instead and open the resulting file in a viewer.

⟨*cli*⟩+≡
```
numAl -t -p 3 3 | dot -T x11
```

If we concentrate on sequences of equal length, what are the longest for which `numAl` can draw the recursion tree (`C-c` aborts)?

2.50 If we let `numAl` count the leaves for us, we can calculate the number of possible alignments conveniently.

⟨*cli*⟩+≡
```
numAl -t 3 3
```

To simplify things, we stick to sequences of equal length. Write a loop to measure the run times of top down computation with pairs of sequences of lengths $1, 2, \ldots$. Plot your results (don't overdo it). Do you recognize the shape of the function you get? You might find it useful to change the number format on the y axis to *engineering* with zero significant digits using the `gnuplot` command

```
set format y '%.0e'
```

2.51 The top down approach uses roughly constant time per possible alignment. How long is that on your computer?

2.52 Without `-t`, `numAl` can quickly determine the number of possible alignments for longer sequences. What is the number of possible alignments between two sequences length 100?

2.53 How long would it take to compute the number of possible alignments of length 100 top down?

2.54 The default mode of `numAl` is called *bottom up*, were we invest a little memory and in return get a huge increase in speed. Let's again calculate the number of possible alignments between two sequences length 3. We start by writing down the 4×4 matrix like in Fig. 2.10A. Then we apply equation (2.2) and initialize the first column and the first row to 1 (Fig. 2.10B). The rest of the matrix we fill by summing their three neighbors according to equation (2.1). Can you finish calculating the number of alignments of two sequences length 3?

2.55 The program `numAl` can also print the programming matrix (`-p`). Take a look at a few short sequences to see how the numbers grow very quickly. How many possible alignments are there between two sequences length 9?

2.56 The number of possible alignments grows so quickly that the numerical system of `numAl` eventually overflows. What is the longest pair of sequences for which `numAl` still works?

A

	0	1	2	3
0				
1				
2				
3				

B

	0	1	2	3
0	1	1	1	1
1	1			
2	1			
3	1			

C

	0	1	2	3
0	1	1	1	1
1	1	3	?	
2	1			
3	1			

Fig. 2.10 Bottom up computation of the number of possible alignments of sequences length 3 using a matrix (**A**), that we initialize (**B**) and fill in (**C**)

Dot Plots and Match Plots

New Terms

alcohol dehydrogenase (*Adh*)	gene duplication	`plotSeg`
dot plot	match plot	`rep2plot`
Drosophila guanche	ortholog	`repeater`
Drosophila melanogaster	paralog	

```
  A T A T T A C T A T
A * * * * *
T * * * * * *
A * * * * *
T * * * * * *
T * * * * * *
A * * * * *
C         *
T * * * * * *
A * * * * *
T * * * * * *
```

Fig. 2.11 Dot plot for comparing `ATATTACTAT` to itself

2.57 Fig. 2.11 shows a matrix with the same short sequence, `ATATTACTAT`, written along both axes. Each matrix entry is either blank for mismatch, or a dot for match. Notice first of all the stretch of dots along the main diagonal. You can also see crosses of three dots for `ATA`, which is the same read backwards. A dot plot like Fig. 2.11 can be drawn quickly in an editor once you realize there are only four possible rows, one for each nucleotide. So once you've figured out a row for a nucleotide, the corresponding rows can be filled by copy and paste. What is the dot plot for `GATATAGATATA`?

2.58 While it is reasonably simple to draw dot plots in an editor, we'd like to make it even simpler. The program `dot.awk` automatically draws the dot plots we just

typed. First, we split the query and subject into character arrays, then we print the header of the dot matrix, and finally the rest of it.

Prog. 2.1 (`dot.awk`)

⟨*dot.awk*⟩≡
```
BEGIN {
  ⟨Split query and subject, Prog. 2.1⟩
  ⟨Print header of dot matrix, Prog. 2.1⟩
  ⟨Print rest of dot matrix, Prog. 2.1⟩
}
```

We run `dot.awk` by passing a query sequence, `q`, and a subject sequence, `s`, from the command line.

⟨*cli*⟩+≡
```
awk -f dot.awk -v q=ATATTACTAT -v s=ATATTACTAT
```

Inside `dot.awk`, the variables `q` and `s` can now be treated like any other variable. Can you split `q` and `s` into character arrays (hint: `split`)?

2.59 Let's write the subject sequence along the header of the dot plot. Can you do that?

2.60 Can you print the rest of the dot plot?

2.61 Many of the dots in a dot plot are part of longer matches and it would be helpful if we could just plot these longer matches rather than all the dots they consist of. The program `repeater` finds matches that cannot be extended—just the kind we need. For the example sequence in Fig. 2.11 we can extract the matches of minimum length 1 (`-m`) and print all their positions (`-p`).

⟨*cli*⟩+≡
```
printf ">s1\nATATTACTAT\n>s2\nATATTACTAT\n" |
    repeater -m 1 -p
```

We can pipe this through `rep2plot` and `plotSeg` to generate a segment plot of the matches, which we call "match plot". What do you observe when you compare your match plot to the dot plot in Fig. 2.11?

2.62 Fig. 2.12 shows the match plot of mRNA for human β-globin (`hbb1.fasta`) and chimp β-globin (`hbb2.fasta`) for a minimal match length of 5. Depending on the match length you choose, this plot can be quite crowded due to random matches. How long are the longest of these random matches?

2.63 With the β-globin sequences you might have wondered, which one is written against which axis. Can you think of a simple test to decide?

2.64 Fig. 2.13 shows the alcohol dehydrogenase (*Adh*) region of *Drosophila melanogaster* compared to the same region in *D. guanche*. One of the two regions contains a transposon indicated by the jump in matches from one diagonal to another. Can you tell which fruit fly got the transposon?

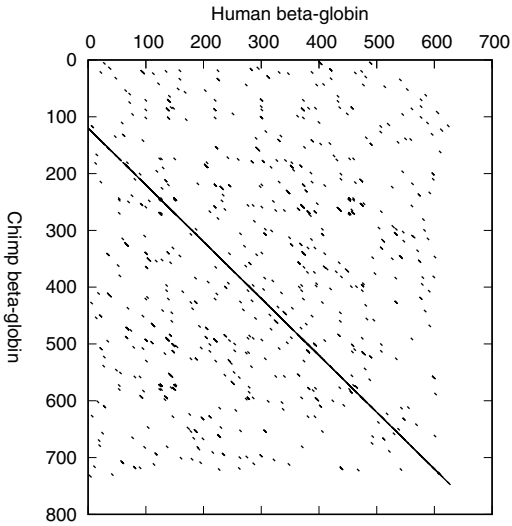


Fig. 2.12 Match plot of β-globin mRNA from chimp and human with minimal match length 5

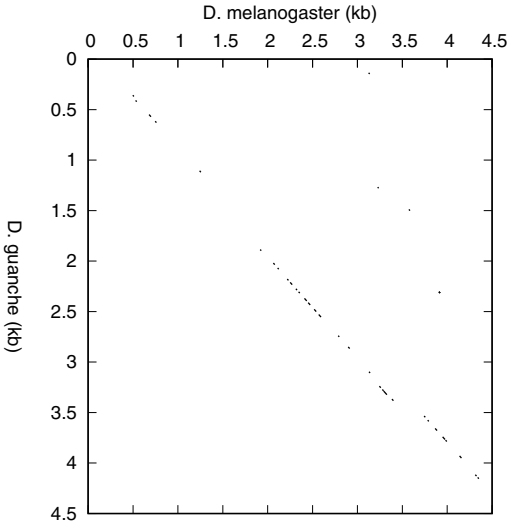


Fig. 2.13 Match plot of the *Adh* region from *D. melanogaster* and *D. guanche*

2.65 The files `dmAdhAdhdup.fasta` and `dgAdhAdhdup.fasta` contain the sequences plotted in Fig. 2.13. Can you copy them and then reproduce that figure?

2.66 When used with `-r`, `repeater` also includes the reverse strands in the analysis. For example, compare the two `repeater` runs with or without `-r`.

⟨*cli*⟩+≡
```
printf ">s1\nAACCT\n>s2\nAAGGT\n" | repeater -m 2 -p
printf ">s1\nAACCT\n>s2\nAAGGT\n" | repeater -m 2 -p -r
```

Produce a match plot for the *Adh* region with minimum match length 12. What happens when you include the reverse strand (hint: `diff`)?

2.67 The *Drosophila Adh* region contains two genes, *Adh* and *Adh-dup*. The coordinates of their coding sequences, CDSs, are listed in Table 2.1, split into exons. These data are also contained in the Genbank files `dmAdhAdhdup.gb` and `dgAdhAdhdup.gb`. Copy them to your current directory. Can you extract the CDS coordinates to make sure the entries in Table 2.1 are correct?

Table 2.1 Coordinates of the coding sequences (CDSs) in the *Adh* region of *D. melanogaster* and *D. guanche*

Organism	Gene	Exon 1	Exon 2	Exon 3
D. melanogaster	*Adh*	2021–2119	2185–2589	2660–2926
D. melanogaster	*Adh-dup*	3226–3321	3748–4152	4204–4521
D. guanche	*Adh*	1984–2076	2145–2549	2613–2879
D. guanche	*Adh-dup*	3221–3316	3540–3944	4007–4345

2.68 Cut out the four CDSs listed in Table 2.1 with `cutSeq` and translate them just to double check the coordinates. Does a CDS contain the stop codon?

2.69 Fig. 2.14 shows the match plot with the CDSs and their exons marked by extra lines. Notice the denser matches inside the CDSs compared to outside. We'd like to reproduce this figure. The first step is to write a script, `exex.sh` that extracts the exon start positions from the CDS lines in the Genbank files. One way to do this, is by repeated application of `tr` in the script `exex.sh`.

Prog. 2.2 (`exex.sh`)

⟨*exex.sh*⟩≡
```
grep CDS |
    tr -d 'a-zA-Z() ' |
    tr ',' '\n' |
    tr -s '.' ' '
```

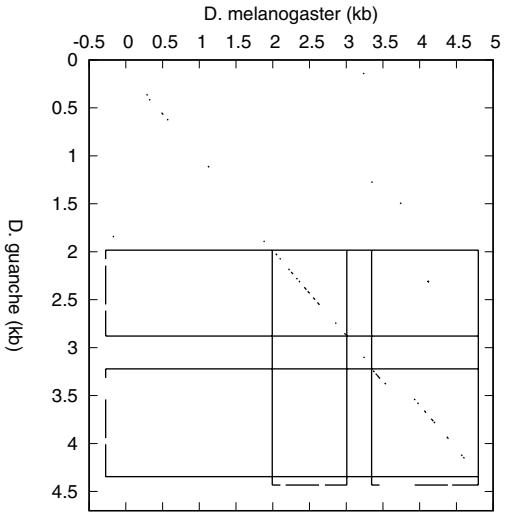


Fig. 2.14 Match plot of the *Adh* region of *D. melanogaster* and *D. guanche* with the coding sequences (CDSs) marked by long horizontal and vertical lines, and the exons within the CDSs by short lines near the axes

Do you understand what it does?

2.70 We save the annotations for our dot plot in `annot.txt`. Can you use `exex.sh` to construct the exons for *D. melanogaster*?

2.71 Can you construct the exons for *D. guanche*?

2.72 What remains, is to draw the long vertical lines to delineate Adh_{dm} and $Adh\text{-}dup_{\mathrm{dm}}$, and the long horizontal lines to delineate Adh_{dg} and $Adh\text{-}dup_{\mathrm{dg}}$. So we write the program `lines.awk` and save the start and end coordinates of the exons in arrays `s` and `e`. Each set of lines depends on the same four array entries, the start and end positions of the two genes. For the vertical lines these positions are interpreted as x coordinates with the y coordinates supplied from outside; for the horizontal lines these positions are interpreted as y coordinates with the x coordinates supplied from outside. Depending on which coordinates are supplied, we know which lines to draw.

Prog. 2.3 (`lines.awk`)

⟨*lines.awk*⟩≡

```
{
  s[NR] = $1
  e[NR] = $2
}
END {
```

```
    ⟨Extract gene positions, Prog. 2.3⟩
    if (y1) {
      ⟨Draw vertical lines, Prog. 2.3⟩
    } else {
      ⟨Draw horizontal lines, Prog. 2.3⟩
    }
  }
```

Can you extract the gene positions?

2.73 Can you draw the vertical lines?

2.74 Can you draw the horizontal lines?

2.75 Can you use `lines.awk` to draw the vertical lines?

2.76 Can you use `lines.awk` to draw the horizontal lines?

2.77 Can you now reproduce Fig. 2.14?

2.78 Did the transposon insert into an exon or an intron?

2.79 With four genes in a pairwise plot, there are six homology relationships. As shown in Fig. 2.15, two of these homology relationships are between the pairs of orthologs $Adh_{\mathrm{dm}}/Adh_{\mathrm{dg}}$ and $Adh\text{-}dup_{\mathrm{dm}}/Adh\text{-}dup_{\mathrm{dg}}$, while the remaining four are between pairs of paralogs. Which of these six homology relationships are visible in the match plot Fig. 2.13?

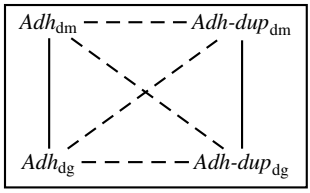


Fig. 2.15 All genes are characterized by homology (box); the solid lines connect orthologs, the dashed lines paralogs

2.80 In our match plots of *Drosophila Adh* we used a minimum match length of 12. How likely is it to find random matches of this length? To find out, we randomize one of the sequences and repeat the analysis.

⟨*cli*⟩+≡

```
  randomizeSeq dgAdhAdhdup.fasta > r.fasta
  cat r.fasta dmAdhAdhdup.fasta | repeater -p -r -m 12 |
      rep2plot | wc -l
```

Write a script, `ranAdh.sh`, to randomize `dgAdhAdhdup.fasta` and look up the longest match with `dmAdhAdhdup.fasta`. How many random matches length 12 or longer do you find on average in 100 reshuffles?

Global and Local Alignment

New Terms		
alignment matrix	local alignment	trace-back
global alignment		

2.81 In an alignment we stitch together the matches in a match plot like that of the *Drosophila Adh* region in Fig. 2.13. To see how this is done, let's start with a pair of short query and subject sequences, q = `AG` and s = `ACG`. Fig. 2.16 shows them written along the edges of an alignment matrix, which is similar to the programming matrix we used for computing the number of possible alignments. And again, each sequence is preceded by a null element, a gap. A matrix entry, v_{ij}, is the score of the best alignment between the partial sequences up to that point, $q[1...i]$ and $s[1...j]$. The very first entry is zero. The first column is then filled with the scores of `A/-` and `AG/--`. Let's set gap opening and gap extension to -1. Can you fill in the first column of the matrix?

	-	A	C	G
-	0			
A				
G				

Fig. 2.16 Initial alignment matrix for a pair of query and subject sequences, q = `AG` and s = `ACG`

2.82 Rather than computing the entries in each cell of the first column from scratch, we can picture them as an extension of the alignment in the preceding cell by a gap:

$$v_{ij} = v_{i-1,j} + g_e.$$

We emphasize this way of thinking by placing a little arrow pointing to the cell of the alignment we extended:

	-	A	C	G
-	0			
A	↑ −1			
G	↑ −2			

Can you fill in the first row of the alignment matrix?

2.83 The remainder of the alignment matrix is filled in by picking the best extension from among the three neighboring cells.

$$\begin{array}{ll} \uparrow & v_{i-1,j} + g_e \\ \leftarrow & v_{i,j-1} + g_e \\ \nwarrow & v_{i-1,j-1} + \text{score}(q[i], s[j]) \end{array}$$

Let's score a match as 1 and a mismatch as -1. Can you fill in the rest of the matrix?

2.84 The program `al` can print the matrix of scores with `-P s`. Can you do that for our score scheme and example sequences?

2.85 The score of the optimal alignment is now written in the lower right hand corner of the matrix. To get the corresponding alignment, we trace the arrows we left behind from the bottom right back to the top left. This generates the alignment from right to left, which we invert to get our customary left-right notation. Can you reconstruct the optimal alignment between `AC` and `ACG` in this way?

2.86 Fig. 2.17 shows the alignment matrix for a new pair of query and subject sequences, q = `TACAGTCC` and s = `TTCAGGGTCC`. What is their optimal alignment?

```
    -   T   T   C   A   G   G   G   T   C    C
-   0 <-1 <-2 <-3 <-4 <-5 <-6 <-7 <-8 <-9 <-10
T ^-1  \1  <0 <-1 <-2 <-3 <-4 <-5 <-6 <-7  <-8
A ^-2  ^0  \0 <-1  \0 <-1 <-2 <-3 <-4 <-5  <-6
C ^-3 ^-1 ^-1  \1  <0 <-1 <-2 <-3 <-4 \-3  <-4
A ^-4 ^-2 ^-2  ^0  \2  <1  <0 <-1 <-2 <-3  <-4
G ^-5 ^-3 ^-3 ^-1  ^1  \3  <2  <1  <0 <-1  <-2
T ^-6 ^-4 \-2 ^-2  ^0  ^2  \2  <1  \2  <1   <0
C ^-7 ^-5 ^-3 \-1 ^-1  ^1  ^1  \1  ^1  \3   <2
C ^-8 ^-6 ^-4 ^-2 ^-2  ^0  ^0  ^0  ^0  ^2   \4
```

Fig. 2.17 Alignment matrix of S_1 = `TACAGTCC` and S_2 = `TTCAGGGTCC` with match score 1, mismatch -1, $g_o = g_e = -1$

2.87 Fig. 2.18 shows the trace-back for the *Adh* locus in *D. melanogaster* and *D. guanche*. When you compare it to the match plot of the same region in Fig. 2.13, you can see in what sense an alignment stitches together matches. Fig. 2.18 was generated using `al -P t` and the default score scheme, and plotted with `plotSeg`. Can you reproduce Fig. 2.18?

2.88 In a global alignment we assume homology across both sequences as sketched in Fig. 2.19A. But in many cases homology is only local like in Fig. 2.19B. To find such local regions of homology, we again start with our alignment matrix, only this time the first row and column are initialized to zero to stop the trace-back if it reaches any of these cells. Why should the trace-back of a local alignment stop when it reaches the first row or column?

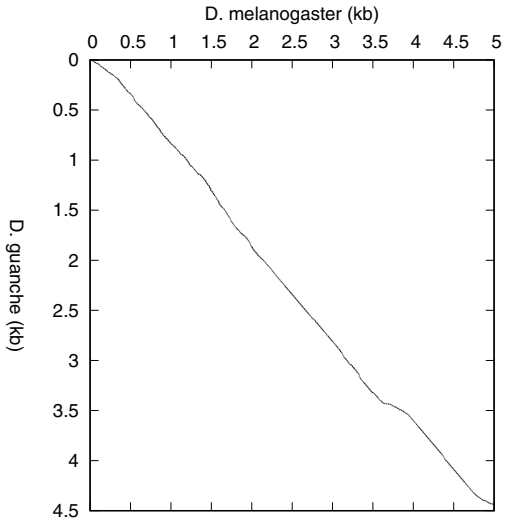


Fig. 2.18 Trace-back of the alignment of the *Adh* region of *D. melanogaster* and *D. guanche*

2.89 After initialization, the local alignment matrix is filled in by taking the maximum over the three possible extensions and no extension, which has score zero. Fig. 2.20 shows such an alignment matrix for the example sequences we already aligned in Fig. 2.17. The trace-back starts at the maximum entry of the matrix, which happens to coincide with the maximum for the global matrix, 4, in the bottom right corner. It stops at the first 0. What is the optimal local alignment?

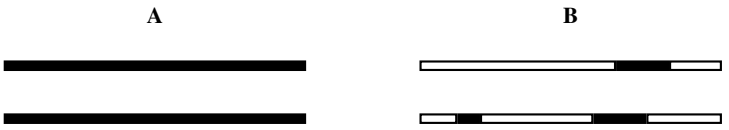


Fig. 2.19 Global (**A**) and local (**B**) homology between pairs of sequences; homologous regions are shown in black

2.90 There is generally only one global alignment. But for local alignments we are often also interested in the second best, the third best, and so on. What is the second best local alignment in Fig. 2.20 that does not intersect the best path?

2.91 Calculate the best local alignment between the *Adh* regions of *D. melanogaster* and *D. guanche* and compare the coordinates you get with the CDS coordinates listed in Table 2.1. Which exon, or exons, in which genes contain the best local alignment?

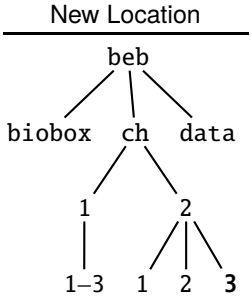
2.92 Where is the second best local alignment?

```
   -  T  T  C  A  G  G  G  T  C  C
-  0  0  0  0  0  0  0  0  0  0  0
T  0 \1 \1  0  0  0  0  0 \1  0  0
A  0  0  0  0 \1  0  0  0  0  0  0
C  0  0  0 \1  0  0  0  0  0 \1 \1
A  0  0  0  0 \2 <1  0  0  0  0  0
G  0  0  0  0 ^1 \3 <2 <1  0  0  0
T  0 \1 \1  0  0 ^2 \2 <1 \2 <1  0
C  0  0  0 \2 <1 ^1 ^1 \1 ^1 \3 <2
C  0  0  0 ^1 \1  0  0  0  0 ^2 \4
```

Fig. 2.20 Local alignment matrix of S_1 = TACAGTCC and S_2 = TTCAGGGTCC with match score 1, mismatch -1, $g_o = g_e = -1$

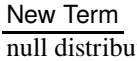
2.3 Application

Optimal alignment is slow when applied to long sequences. But many interesting questions in biology can be tackled with sequences so short, a program like `al` has all the speed we need. For example, we've already noticed that the match plot of the *Adh* region of *Drosophila* in Fig. 2.13 contains no trace of the homology between *Adh* and *Adh-dup*. Can we use optimal alignment to detect that homology? And if so, can we reconstruct the evolutionary history of the *Adh* locus?



2.93 Can you make a new directory for this section and change into it?

Detecting Homology



2.94 To reacquaint ourselves with the *Drosophila Adh* region, take a look at Fig. 2.21, which shows the exons of the two coding sequences, CDSs, whose coordinates are contained in the Genbank files `dmAdhAdhdup.gb` and `dgAdhAdhdup.gb`. The program `plotLine` can draw several data sets when presented with three columns,

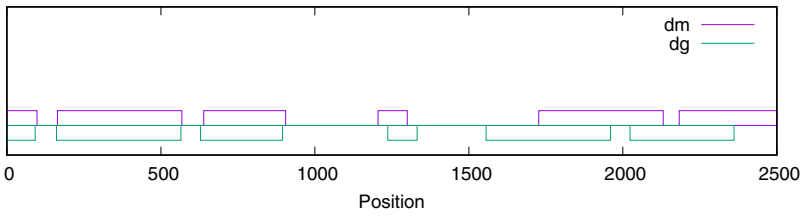


Fig. 2.21 The exons of the coding sequences of the two *Adh* genes in *D. melanogaster* (*dm*) and *D. guanche* (*dg*); the first three exons belong to *Adh*, the second three to *Adh-dup*

x, y, and category. Fig. 2.21 was generated using `exex.sh` and `drawGenes`. Can you reproduce the part for *D. melanogaster*?

2.95 Can you now add the exons for *D. guanche*?

2.96 Local alignment already told us that exon 2 in *Adh* and *Adh-dup* is the most conserved region of the *Adh* locus. So let's concentrate on that exon and save its four copies in separate files called, for example, `dmAdhE2.fasta`. Can you do that (`cutSeq`)?

2.97 How long are the four copies of exon 2?

2.98 There are six possible comparisons between the four *Adh* genes marked by the bullets:

	Adh_{dm}	$Adh\text{-}dup_{\mathrm{dm}}$	Adh_{dg}	$Adh\text{-}dup_{\mathrm{dg}}$
Adh_{dm}		•	•	•
$Adh\text{-}dup_{\mathrm{dm}}$			•	•
Adh_{dg}				•
$Adh\text{-}dup_{\mathrm{dg}}$				

We'd like to replace the bullets with scores, so in our program `scores.sh` we write two nested loops to generate all comparisons between distinct sequences. We've already seen that `do` blocks are closed by `done`. Here we see an `if` block, which is closed by `fi`.

Prog. 2.4 (`scores.sh`)

⟨*scores.sh*⟩≡

```
for i in gA gD mA mD
do
    for j in gA gD mA mD
    do
        if [ ${i} != ${j} ]
        then
            ⟨Calculate score, Prog. 2.4⟩
        fi
```

```
    done
done
```

Can you finish the script?

2.99 Can you replace the bullets by scores?

2.100 As shown in Fig. 2.15, some exon pairs have arisen due to gene duplication (paralogy), others by speciation (orthology). How do our alignment scores differ for paralogs and orthologs?

2.101 Clearly, the scores between orthologous pairs are much larger than between paralogous pairs. The question is, are the low scores between paralogous pairs still significant? To test this, we repeatedly generate random alignments using `ral.sh`.

Prog. 2.5 (`ral.sh`)

⟨*ral.sh*⟩≡

```
for a in $(seq $1)
do
    randomizeSeq $2 |
          al $3 |
          grep Score
done
```

Fig. 2.22 shows the distribution of 1000 such scores for Adh_{dm}/$Adh\text{-}dup_{\mathrm{dm}}$. Can you reproduce this plot?

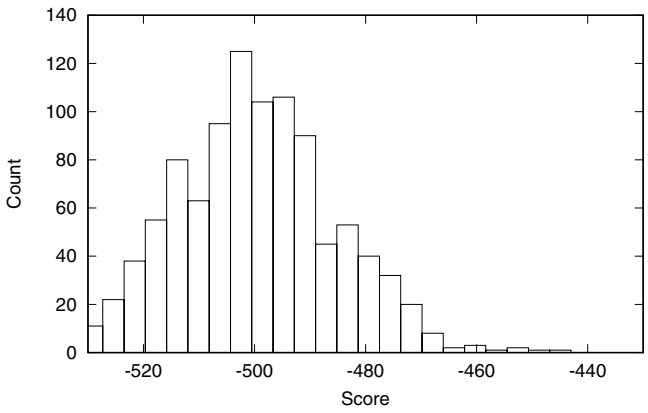


Fig. 2.22 The null distribution of alignment scores for exon 2 from Adh_{dm} and $Adh\text{-}dup_{\mathrm{dm}}$

2.102 What is the significance of the score of Adh_{dm}/$Adh\text{-}dup_{\mathrm{dm}}$?

2.103 What is the significance of the remaining three comparisons between duplicated genes? What do you conclude about the relative sensitivity of match plots *vs* alignments?

The Evolutionary History of the *Adh* Locus

New Terms		
`midRoot`	`nj`	rooted tree
`mutator`	`plotTree`	unrooted tree

2.104 Given that there is appreciable homology between all four copies of exon 2, let's quantify it. A good quantifier of homology, or rather, lack thereof, is the number of mismatches per site. What is the number of mismatches per site between exon 2 from Adh_{dm} and Adh_{dg}?

2.105 The script `mism.sh` calculates the mismatches per site between the four copies of exon 2. Can you finish it?

Prog. 2.6 (`mism.sh`)

⟨*mism.sh*⟩≡

```
echo 4
for i in dmAdh dmDup dgAdh dgDup
do
    printf "%s " $i
    for j in dmAdh dmDup dgAdh dgDup
    do
        ⟨Calculate mismatches per site, Prog. 2.6⟩
    done
    printf "\n"
done
```

2.106 Mismatches are distances and distance matrices can be summarized into a tree using the program `nj`. Can you make sense of its output?

2.107 Plot the output of `nj` with `plotTree`. What do you get?

2.108 Fig. 2.23A shows the rooted tree based on mismatches. We added the root with `midRoot`. Can you reproduce Fig. 2.23A?

2.109 The number of mismatches tends to be smaller than the number of mutations because a mutation can affect a site more than once, but each mismatch is only counted once. To see the difference between mismatches and mutations, we can

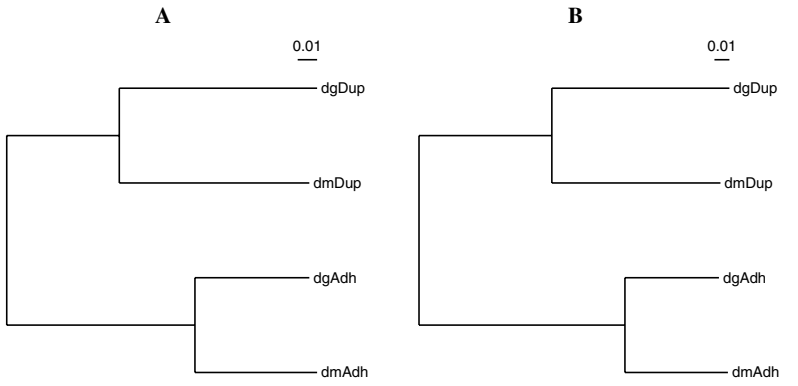


Fig. 2.23 Rooted mismatch (**A**) and mutation (**B**) tree of *Drosophila Adh* exon 2; the two trees are almost identical

artificially mutate a sequence with the program `mutator` and then count its mismatches. Since the original sequence and its mutated version are aligned, we prevent gap insertion by giving gap opening a large negative score of, say, −500.

⟨*cli*⟩+≡
```
mutator -m 0.5 dmAdhE2.fasta | al -p -500 dmAdhE2.fasta |
    awk '/^E/{print $5/405}'
```

What do you observe when you try this a couple of times?

2.110 The number of mutations per site, k, can be estimated using the Jukes-Cantor equation [22],

$$k = \frac{-3}{4} \log\left(1 - \frac{4m}{3}\right), \tag{2.3}$$

where m is the number of mismatches per site. What is the number of mutations corresponding to the number of mismatches you got?

2.111 Given a mismatch matrix as input, the program `mut.awk` converts this to mutations using equation (2.3).

Prog. 2.7 (`mut.awk`)

⟨*mut.awk*⟩≡
```
NF == 1 {
  print
}
NF > 1 {
  printf $1
  ⟨Calculate row of mutations, Prog. 2.7⟩
  printf "\n"
}
```

Can you complete `mut.awk`?

2.112 Fig. 2.23B shows the mutation tree of *Adh* corresponding to the mismatch tree on the left. Can you reproduce it?

2.113 Given that the divergence time between *D. melanogaster* and *D. guanche* is approximately 32 million years [16], how old is the duplication of *Adh*?

Chapter 3
Exact Matching

We have already seen that alignments contain exact matches. So far we haven't made use of this—even when comparing two identical sequences, we filled in the entire alignment matrix to find what amounts to one long exact match. It might be quicker to first look for exact matches and build alignments from them. As we shall see in Chapter 4, this is the approach used in modern alignment tools like Blast. This means that exact matching, the topic of this chapter, is eminently useful for inexact matching.

The basic task in exact matching is to look up a short pattern, say P = `ACA`, in a potentially long text, say T = `CACAGACACAT`. In this case, P starts in T at positions 2, 6, and 8:

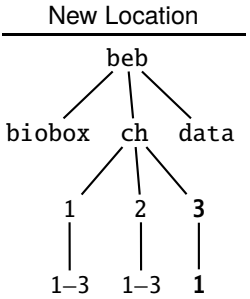
```
 P        T
123 12345678901
ACA CACAGACACAT
```

To find P in T, we can either use both as given, or preprocess them to speed up the search. Preprocessing is done with tree structures, a perhaps surprising reappearance of evolution's central metaphor. In this section we look at preprocessing the pattern, in Sections 3.2 and 3.3 we look at preprocessing the text. In Section 3.4, finally, we use text preprocessing for text compression.

3.1 Keyword Trees

We begin our exploration of exact matching with what's known as the *naïve* algorithm for exact matching. It is actually quite fast, but does have its limitations. These limitations are overcome by using a keyword tree, which efficiently matches not only one but an arbitrarily large set of patterns. This is called *set matching*.

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_3



3.1 Can you make a directory for the new Section 3.1?

New Terms		
`drawKt`	keyword tree	`ps2pdf`
`dvips`	`latex`	`revComp`
`evince`	naïve matching	set matching

3.2 Regardless of the kind of matching we do, our sequence data comes in FASTA files, and we begin by reminding ourselves how to parse them. Let's generate a FASTA file with two entries and two lines of sequence data each.

⟨*cli*⟩+≡
```
printf ">s1\nTATTC\nTCTTC\n>s2\nAGTTA\nCTAAT\n" > s.fasta
cat s.fasta
```

```
>s1
TATTC
TCTTC
>s2
AGTTA
CTAAT
```

Next, we write the program `readFasta.awk` that separates the headers from the data and concatenates the data. It is not a particularly useful program by itself, but we shall use ideas from it in our implementation of the naïve algorithm.

⟨*cli*⟩+≡
```
awk -f readFasta.awk s.fasta
```

```
>s1
TATTCTCTTC
>s2
AGTTACTAAT
```

Inside `readFasta.awk` we generate the concatenated sequence by dividing the work into dealing with headers, data, and the last sequence.

Prog. 3.1 (`readFasta.awk`)

⟨*readFasta.awk*⟩≡
⟨*Deal with headers, Prog. 3.1*⟩
⟨*Deal with data, Prog. 3.1*⟩
⟨*Deal with last sequence, Prog. 3.1*⟩

Dealing with the data is a bit easier than dealing with the headers, so we start with that. Can you implement it?

3.3 FASTA headers open the subsequent sequence *and* close the previous one. So when we encounter a header, it might be the first one, in which case we just print it. Otherwise, we print the previous sequence, `t`, reset it, and then print the new header. Can you implement the chunk of code dealing with the headers?

3.4 Having already printed the last header, the end of file prompts us to print the corresponding sequence. Can you implement the chunk of code dealing with the last sequence?

Step	1	2	3
T	CACAGACACAT	CACAGACACAT	CACAGACACAT
P	ACA	ACA	ACA
Match	0	111	0
Step	4	5	6
T	CACAGACACAT	CACAGACACAT	CACAGACACAT
P	ACA	ACA	ACA
Match	10	0	111
Step	7	8	9
T	CCACAGACACAT	CCACAGACACAT	CCACAGACACAT
P	ACA	ACA	ACA
Match	0	111	0

Fig. 3.1 Naïve matching algorithm; 1 is match, 0 mismatch

3.5 In the naïve matching algorithm we start at the first position of P and T, and march P as far as we can. If we get to the end of P, we report a match, otherwise we stop. Then we move P one position with respect to T and start again. Fig. 3.1 illustrates this idea, which we implement in the program `naive.awk`. The heart of the program is the function `naive`, which prints all starting positions of the pattern in the text. But first things first, can you deal with the data?

Prog. 3.2 (`naive.awk`)

⟨*naive.awk*⟩≡
⟨*Deal with headers, Prog 3.2*⟩
⟨*Deal with data, Prog. 3.2*⟩
⟨*Deal with last sequence, Prog. 3.2*⟩

```
function naive(p, t) {
  m = split(p, pa, "")
  n = split(t, ta, "")
  ⟨Find matches, Prog. 3.2⟩
}
```

3.6 When we find a header, it might be the first, in which case we just print it. Otherwise, we call `naive` and then print it. Can you implement the code chunk dealing with headers?

3.7 Can you implement the code chunk dealing with the last sequence?

3.8 The most interesting part of `naive.awk` is, of course, the chunk for finding matches. In it we need to break out of a `for` loop on mismatch. This is done by saying `break`, as in

```
if (a[i] != b[j])
  break
```

Can you implement the chunk for finding matches? Where in `s.fasta` does `AT` appear?

3.9 Let's look again at the example sequences in Fig. 3.1. Where does `ACA` appear in `CACAGACACAT`?

3.10 Where does `ACGTCG` occur in the genome of *M. genitalium*? Where does its shorter variant `ACGTC` occur?

3.11 So far, we have only looked at the forward strand of the *M. genitalium* genome. The program `revComp` computes the reverse complement of a sequence. Is `ACGTCG` unique across the entire genome of *M. genitalium*?

3.12 The naïve algorithm becomes slow when applied to a pattern that matches everywhere, for example, P = `AAA`, T = `AAAAAAAAAA`. In that case the run time is expected to be of the order of the product of the lengths of P and T, $O(|P| \times |T|)$. Let's measure the run time of `naive.awk` when searching for 20 `A`s followed by a `G` in ten million `A`s. We generate a random sequence 10 Mb long, change `C`, `T`, and `G` to `A`, and save it to `ran.fasta`. Then we do a timed search.

⟨*cli*⟩+≡

```
ranseq -l 10000000 | tr 'CGT' 'A' > ran.fasta
time awk -f naive.awk -v p=AAAAAAAAAAAAAAAAAAAG ran.fasta
```

The command `time` returns three time measurements; the actual time elapsed (`real`), the time taken by the user's process (`user`), and the time taken by system processes (`sys`). How long does `naive.awk` take?

3.13 How long does it take to search for only the `G` in the 10 Mb of `A`? Can you explain the difference?

3.14 Scripting languages like Awk or Python tend to be slower than compiled languages like C or Go. The program `naiveMatcher` is written in Go and implements the same algorithm as `naive.awk`. What are the run times when searching for the 20 `A`s followed by `G` in `ran.fasta`?

3.15 The naïve matching algorithm outlined in Fig. 3.1 can also be understood as a graph consisting of nodes and edges as shown in Fig. 3.2A. Each node represents a state in the matching procedure and each edge a response to match or mismatch. Match is illustrated by gray lines, mismatch by red arrows. The defining characteristic of the naïve algorithm is that upon every mismatch, or "failure", matching resumes at the beginning of P; hence all red arrows in Fig. 3.2A point to the first node. However, a match to P = `ACA` implies an additional match to the first character of P. Therefore, instead of returning to the beginning of P, the algorithm only needs to compare the characters from $P[2]$ = `C` onward. This is illustrated in Fig. 3.2B. What are the failure links for P = `AAA`?

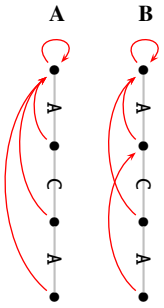


Fig. 3.2 A pattern to be matched shown as a graph. Grey lines are followed upon match, red arrows are "failure links" that are followed upon mismatch. Naïve failure links (**A**) always return to the beginning of the pattern. Better failure links (**B**) incorporate the fact that after the last `A` has been matched, the first `A` has also already been matched

3.16 To systematically construct failure links, we begin with a self-referential arrow to the root. This means, don't move in the tree if there's a mismatch. If the first character matches, but is followed by a mismatch, we return to the root (Fig. 3.3A). After this initialization, we work our way from top to bottom. For each node, we go to its parent and follow its failure links until we find a match. The node reached by the match is the target of the next failure link. For example, in Fig. 3.3B we look for a match to `T`; after following the parent's failure link, we follow the match link and have thus found the target for the failure link. Next, two existing failure links are followed without a match, hence the new failure link points to the start node (Fig. 3.3C). Then one failure link is followed before we find a match in Fig. 3.3D,

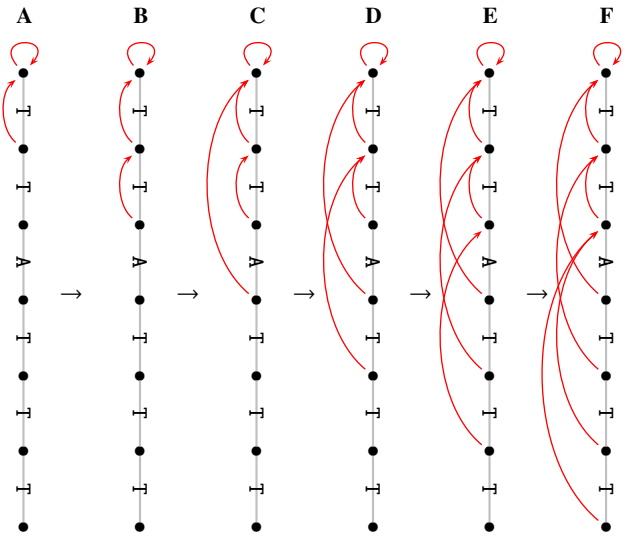


Fig. 3.3 Systematic construction of failure links going from the initialization (**A**) through to the fully preprocessed pattern (**F**)

and this move is repeated in Fig. 3.3E. Finally, two failure links need to be followed before a match is found leading to the complete set of failure links in Fig. 3.3F. Can you construct the failure links for P = ATATAT?

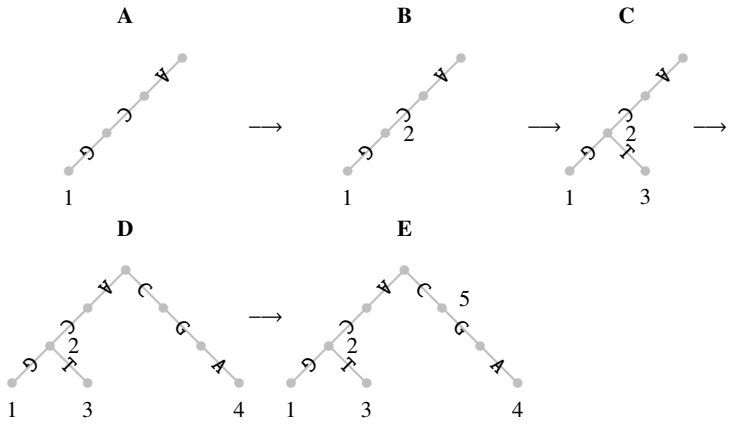


Fig. 3.4 Sequential construction of a keyword tree in five steps for the five patterns P_1 = ACG, P_2 = AC, P_3 = ACT, P_4 = CGA, and P_5 = C

3.17 Keyword trees were originally developed for matching sets of patterns, say all entries in a dictionary for spell checking. We start a bit smaller and match five patterns, or *keywords*: P_1 = ACG, P_2 = AC, P_3 = ACT, P_4 = CGA, and P_5 = C. Notice that P_2 is contained in P_1 and P_3, P_5 in all others, and P_1 and P_3 have the matching prefix AC. To make searching efficient, matching prefixes are summarized as common paths in the keyword tree. Its construction is shown in Fig. 3.4, where the patterns are sequentially fitted into a growing tree. The first partial tree for P_1 = ACG in Fig. 3.4B should look familiar from our graphs for single patterns, except for the label on the terminal node indicating the pattern just matched.

Repeat the keyword tree construction using paper and pencil. Can you enter the failure links?

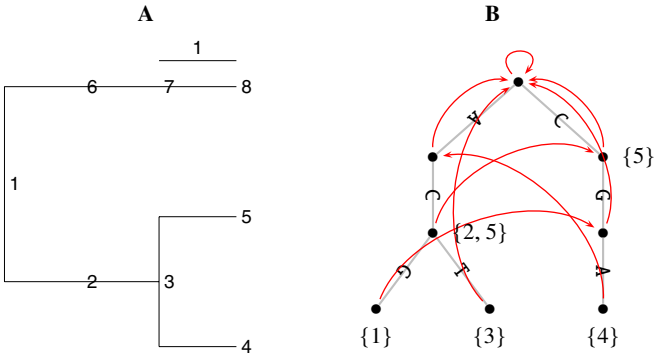


Fig. 3.5 Drawing keyword trees with drawKt as "phylogenies", without failure links (**A**), and as more conventional graphs with failure links(**B**)

3.18 Let's use the keyword tree from Fig. 3.27 to search for the five patterns in T = ACGC. We start at the root and walk into the tree and into the text. Whenever we find a match, we advance in the text and in the tree. If we find a mismatch, or run out of matches, we just follow the failure link and don't move in the text. But there is a problem. Can you spot it?

3.19 We can use the program drawKt to draw keyword trees as text in Newick format.

⟨*cli*⟩+≡

```
drawKt -t ACG AC ACT CGA C
```

```
(((4[G->7{1}],5[T->1{3}])3[C->6{2,5}],
((8[A->2{4}])7[G->1])6[C->1{5}])2[A->1])1[->1];
```

Fig. 3.5A shows the visualization of this tree with plotTree. Can you reproduce it?

3.20 As a more conventional alternative to the "phylogeny" notation of a keyword tree, `drawKt` can also draw the graph with labeled edges shown in Fig. 3.5B in LaTeX. Can you reproduce that figure?

3.21 If you haven't used LaTeX before, take a look at `ktWrapper.tex` and add three lines after the command `\begin{document}`.

```
\title{Our Keyword Tree}
\author{Alpha \& Beth}
\maketitle
```

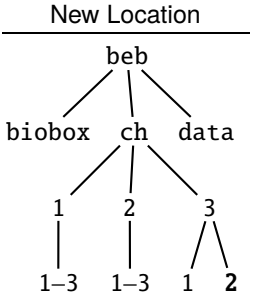
What do you observe when you typeset the page again?

3.22 The program `keyMat` looks for exact matches using a keyword tree. What is the run time of `keyMat` when searching for a string of 20 `A` followed by a `G` in `ran.fasta`? Compare your result to the corresponding run time of `naiveMatcher`. What happens when you double the number of `A`s in the pattern?

3.23 The program `sblast` implements a simple version of the popular search tool Blast. It relies on exact matching and `sblast` can either use a keyword tree or the naïve method for exact matching. The file `ecoliK12.fasta` contains the genome of the K12 strain of the bacterium *Escherichia coli*, which lives in the gut of warm-blooded animals like us. The K12 strain is widely used in genetic engineering. Cut out the first kb of that genome and search for it with `sblast` in the genome of the pathogenic *E. coli* strain O157H7 in `ecoliO157H7.fasta`. How long does this take? How long does it take with naïve matching?

3.2 Suffix Trees

Exact matching is about finding patterns in a text. This can be sped up by preprocessing. With keyword trees we preprocessed the patterns, now we preprocess the text. To search a text efficiently, we need an index, like the index of this book. However, a suffix tree is a special index as it references every substring of the text, not just a list of words like a book index.



3.24 Can you first construct and then go to the working directory for this section?

New Terms

`date` `shustring` suffix tree
`drawSt`

3.25 A suffix is the end of a word. What are the suffixes of `CATGGCAT`?

3.26 Fig. 3.6 shows the suffix tree of t = `CATGGCAT`. Every suffix is represented as a path from the root to a leaf labeled with the starting point of the suffix. To search for, say, p = `CAT` in t, start matching at the root and simultaneously walk into the tree and the pattern, until the pattern is found. The leaves in the subtree now reached indicate the starting positions of p in t. Where is p = `AT` in t? Where p = `X`?

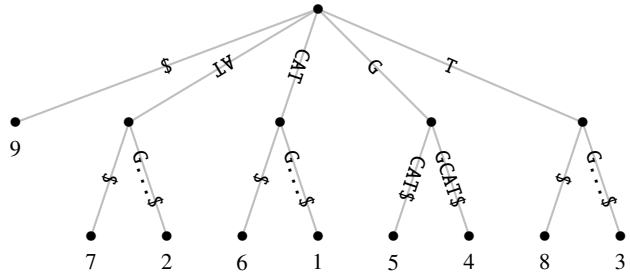


Fig. 3.6 Suffix tree of `CATGGCAT$`

3.27 A suffix tree is constructed by first drawing a root, a leaf, and a connecting edge. The edge is labeled with the first suffix, to which we add an end marker, or "sentinel" character, `$`. The leaf is labeled "1" to refer to the first suffix. The result of this first construction step is shown in Fig. 3.7A. Then the second suffix, `ATGGCAT$`, is taken and fitted into the partial tree. Its first character, `A`, mismatches the initial `C` of the first suffix, so we make a new branch to get Fig. 3.7B. This is repeated for suffixes 3 and 4 to get Fig. 3.7C and D. The fifth suffix, `GCAT$`, matches the existing suffix `GGCAT$` for one character; the next mismatch breaks the branch to give Fig. 3.7E. The same thing happens when we add `CAT$` to get Fig. 3.7F. Can you finish the suffix tree?

3.28 Can you construct the suffix tree for `GTTCAAAT` by hand?

3.29 A path label in a suffix tree consists of the concatenated labels starting at the root and ending somewhere in the tree. Now, any path label that ends above an internal node is a repeat. In particular, paths labels that end just above nodes connected only to leaves, cannot be extended to the right without losing the match. What are the path labels of such "frontier" nodes in Fig. 3.6? What is the longest repeat in `CATGGCAT`?

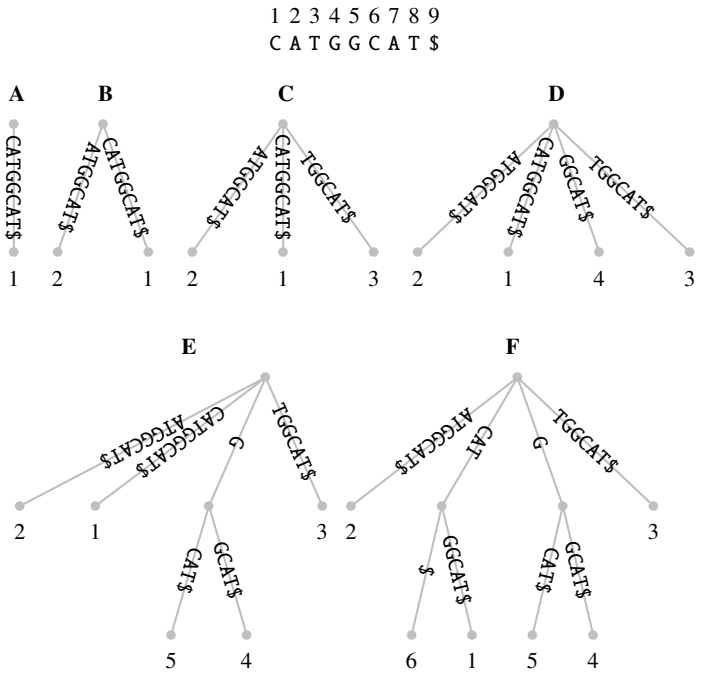


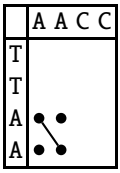
Fig. 3.7 The first six steps in the construction of the suffix tree for `CATGGCAT$`

3.30 The program `repeater` uses suffix trees to find repeats. What is the longest repeat in the genome of *M. genitalium*? Does the result change if you include the reverse strand?

3.31 Is the repeat we just found particularly long? To answer, we compute the length of the longest repeat expected in a random version of the *M. genitalium* genome. We begin by calculating the probability of a match between two random nucleotides. Since there are four possible matches, `AA`, `CC`, `GG`, and `TT`, the probability of randomly drawing two identical nucleotides from a sequence where each nucleotide has frequency $1/4$ is $(1/4)^2 \times 4 = 1/4$. However, in real sequences, the nucleotides usually do not occur with equal frequencies. Use the program `cres` to compute the nucleotide composition of *M. genitalium* in `mgGenome.fasta`. What is the probability of drawing `AA` when picking two random nucleotides from the genome of *M. genitalium*?

3.32 What is the probability of drawing a pair of identical nucleotides from the genome of *M. genitalium*?

3.33 To get from the match probability, P_m, to the expected length of the longest repeat in the genome of *M. genitalium*, consider a toy dot plot with three matches, two of length 1, one of length 2:



The probability of drawing a dot is P_{m}. The probability of drawing a diagonal of length l, and hence a match of length l, is P_{m}^{l}. The expected number of such diagonals is their probability times the number of cells in the dot plot. When comparing two sequences of length L, this is $(L-l)^2$, which is approximately L^2. Hence the expected number of l-mer matches, n_{e}, is

$$n_{\mathrm{e}} = P_{\mathrm{m}}^{l} \times L^2.$$

Since we are looking for the *longest* such match, we set $n_{\mathrm{e}} = 1$. What is the expected length of the longest repeat in the genome of *M. genitalium*? How does this compare to the observed longest repeat?

3.34 To check the expected match length just computed, randomize the sequence of *M. genitalium* with `randomizeSeq` and compute the longest repeat in that randomized version. What is its average length from, say, 100 repeats?

3.35 The program `drawSt` draws suffix trees. By default it draws them without a sentinel and assumes that the tree is contained in file `st.tex`. So we can calculate the suffix tree without sentinel for our example sequence, `CATGGCAT`.

⟨*cli*⟩+≡
```
printf ">s\nCATGGCAT\n" | drawSt -w st1.tex > st.tex
latex st1
dvips st1
ps2pdf st1.ps
evince st1.pdf &
```

We can also include a sentinel (`-s`), which gives Fig. 3.6.

⟨*cli*⟩+≡
```
printf ">s\nCATGGCAT\n" | drawSt -s -w st2.tex > st.tex
```

What's the difference?

3.36 Most books come without an index. One reason for this is that making an index is a lot of work. We have already seen that `repeater` is quick when applied to the genome of *M. genitalium*. However, with just half a megabase, this genome is tiny compared to, say, ours with 3.1 gigabases. So the question is, how time-consuming is it to construct suffix trees in general? Consider the naïve construction method depicted in Fig. 3.7 and apply it to a sequence consisting only of one kind of nucleotide, for example `AAAA`. How does the run time of naïve suffix tree construction scale as a function of sequence length?

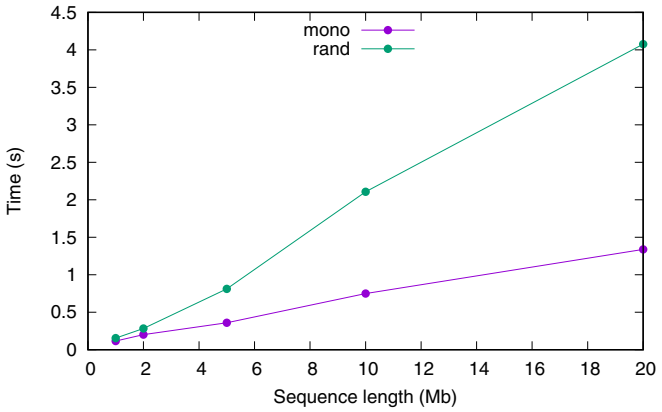


Fig. 3.8 Run time of `repeater` as a function of sequence length; the program was either given mononucleotide (*mono*) or random sequences (*rand*) as input

3.37 The program `repeater` uses a more efficient algorithm for constructing suffix trees than the naïve construction in Fig. 3.7. Fig. 3.8 shows its run time as a function of sequence length for mononucleotide and random sequences. On Windows-Ubuntu we found that parsing mononucleotide sequences is much quicker than parsing random sequences, while on macOS the effect was less pronounced. `repeater` relies on the library `libdivsufsort`. Benchmarks for this library show that text over a single character is analyzed more quickly than text over the full alphabet[1]. So the general trend of our results is in line with this earlier observation, but the magnitude of the effect seems to be system-dependent. This just goes to show that it is always a good idea to make specific measurements. Fig. 3.8 was constructed using the script `repeater.sh`, where we iterate over five sequence lengths between 1 and 20 Mb and generate the corresponding random sequence. Then we measure the run time for a mononucleotide version of the random sequence and for the original random sequence. After the time measurements, we delete the sequence files again.

Prog. 3.3 (`repeater.sh`)

⟨*repeater.sh*⟩≡

```
for a in 1 2 5 10 20
do
    ranseq -l ${a}000000 > r.fasta
    tr 'CGT' 'A' < r.fasta > m.fasta
    ⟨Measure time for mononucleotide sequence, Prog. 3.3⟩
    ⟨Measure time for random sequence, Prog. 3.3⟩
    rm r.fasta m.fasta
done
```

[1] `https://github.com/y-256/libdivsufsort/blob/wiki/SACA_Benchmarks.md`

To measure the time for the mononucleotide sequences, we convert the random sequence to all `A` and time `repeater`. The command `time` without any qualifications refers to a built-in `bash` command. This is a bit fickle to use as part of a pipeline. Instead, we use the program `date`. If you're on macOS, the default `date` command differs from our description here, but `gdate` corresponds to our `date`. What is the default output of `date` (or `gdate`)?

3.38 `date` can print the date in various formats, which are prefixed by a plus. For example, `date` can print the seconds since the Unix epoch started on 1st of January 1970.

⟨*cli*⟩+≡
```
date +%s
```

How many seconds have elapsed since then?

3.39 We'd like to use `date` to measure run times, for which seconds often don't give us enough resolution. So let's measure the run time of `ls` with nanosecond precision.

⟨*cli*⟩+≡
```
date +%N; ls; date +%N
```

How many nanoseconds does `ls` take on your computer?

3.40 To measure the run time of a command, we store its start and end time, and subtract start from end. We also append the category `mono` to the time measurement to distinguish it from the time for random sequences. Since we are not interested in the actual output of `repeater`, we redirect it to the "null device", `/dev/null`. This is actually a file, you can picture it as a bottomless bin.

⟨*Measure time for mononucleotide sequence, Prog. 3.3*⟩≡
```
st=$(date +%s.%N)
repeater m.fasta > /dev/null
en=$(date +%s.%N)
rt=$(echo "$en - $st" | bc -l)
echo $a $rt "mono"
```

Can you measure the time for the random sequence?

3.41 Can you reproduce Fig. 3.8?

3.42 The same logic used to find repeats can be used to find unique substrings, as any path label that ends on a terminal branch is unique. Can you look up the shortest unique substring or substrings in `CATGGCAT` from its suffix tree Fig. 3.6?

3.43 The program `shustring` finds shortest unique substrings. How long are the shortest unique substrings in the genome of *M. genitalium*?

3.44 As with the longest repeat, we'd like to know whether the shortest unique is due to chance or not. So we write the script `shustring.sh` to find out.

Prog. 3.4 (shustring.sh)

⟨*shustring.sh*⟩≡

```
for a in $(seq $1)
do
    randomizeSeq mgGenome.fasta |
          shustring -r |
          tail -n +3 |
          head -n 1
done
```

How often do you find a shortest unique substring of length 6 or less in 100 random versions of the *M. genitalium* genome?

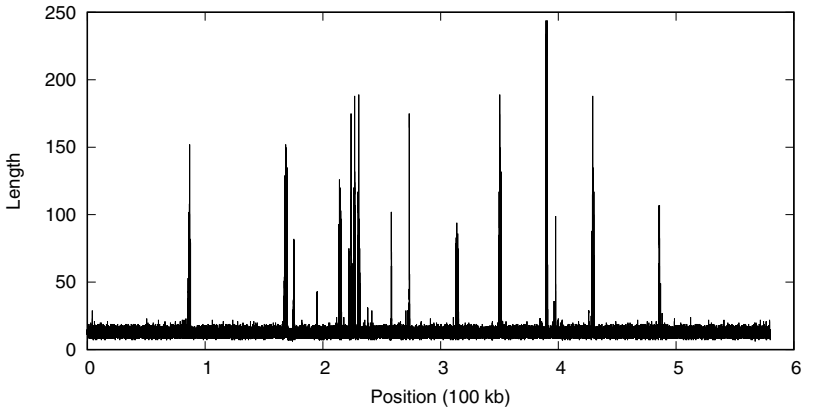
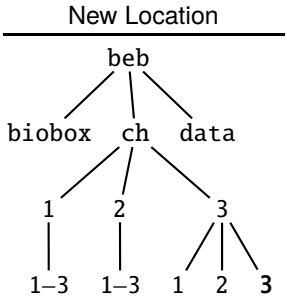


Fig. 3.9 The shortest unique substrings at every position along the genome of *M. genitalium*

3.45 Apart from the globally shortest unique substrings, the program `shustring` can also print the local shustrings at every position. Fig. 3.9 shows the lengths of these local shustrings as a function of sequence position. Can you reproduce it?

3.3 Suffix Arrays

The nodes of suffix trees consume a lot of computer memory. This becomes a problem when computing suffix trees for very long sequences such as mammalian genomes with their billions of nucleotides. Hence a space-saving alternative to suffix trees has been developed, the suffix array [34]. It consists of the alphabetically sorted suffixes of the input sequence.



3.46 Can you make and change into the directory for this section?

New Terms		
enhanced suffix array	lcp interval tree	string depth
inverse suffix array	longest common prefix	suffix array

3.47 Like suffix trees, suffix arrays are built from suffixes. So we begin by writing the program `suf.awk` that converts a sequence into its suffixes. It concatenates the sequence data and then prints the starting position and sequence of each suffix.

Prog. 3.5 (`suf.awk`)

⟨*suf.awk*⟩≡

```
!/^>/ {
  t = t $1
}
END {
  n = length(t)
  for (i = 1; i <= n; i++)
    print i, substr(t, i)
}
```

Can you run `suf.awk` on `CATGGCAT`?

3.48 A list of suffixes isn't a suffix array yet, but it's close. Fig. 3.10A shows the suffix array for `CATGGCAT$`. This time we include the sentinel for easier comparison with the corresponding suffix tree in Fig. 3.10B. The actual suffix array is just the column of integers headed sa. So a suffix array is an array of suffix starting positions after the suffixes have been sorted alphabetically. Compare the order of suffixes in the suffix array and the suffix tree. Do you notice anything?

3.49 Can you reproduce the suffix array in Fig. 3.10A?

3.50 Fig. 3.11A shows the suffix array for another example sequence, `ACCCA$`. Notice again the parallel order of suffixes in the suffix array and leaves in the suffix tree next to it, Fig. 3.11B. Apart from the leaves, this order also implies the positions of the internal nodes. Take for example the internal node with path label

A

i	sa[i]	suf[i]
1	9	$
2	7	AT$
3	2	ATGGCAT$
4	6	CAT$
5	1	CATGGCAT$
6	5	GCAT$
7	4	GGCAT$
8	8	T$
9	3	TGGCAT$

B

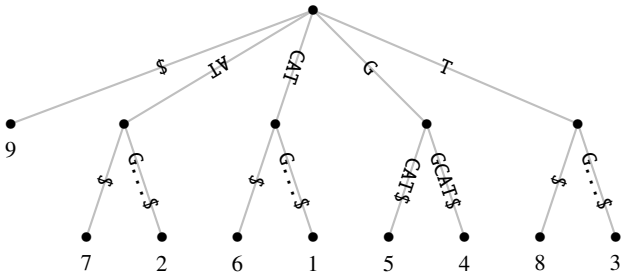


Fig. 3.10 Suffix array (**A**) and suffix tree (**B**) of CATGGCAT$

A

i	sa	suf
1	6	$
2	5	A$
3	1	ACCCA$
4	4	CA$
5	3	CCA$
6	2	CCCA$

B

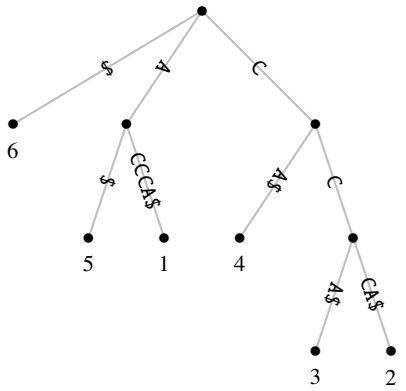


Fig. 3.11 Suffix array of ACCCA$ (**A**) and corresponding suffix tree (**B**)

A in Fig. 3.11B. The leaves connected to that node refer to suffixes 5 and 1. These are neighbors in the suffix array, where they occupy entries sa[2] = 5 and sa[3] = 1. We denote this interval sa[2...3]. There are three more intervals that correspond to the three nodes we haven't characterized yet. Can you write down the suffix array intervals for them?

3.51 To further illustrate the close relationship between suffix array and suffix tree, we add to our suffix array the array for "common prefixes", cp, where cp[i] is the longest common prefix between suf[i] and its left neighbor, suf[$i-1$]. Fig. 3.12A shows the first three common prefixes. The very first suffix has no left-hand neighbor, hence its common prefix is "not defined", nd. The second suffix has no common prefix, but the third one has the common prefix A. This is also marked in red in both the suffix array and tree in Fig. 3.12. Can you fill in the missing three common prefixes?

A

index	sa	suf	cp
1	6	$	nd
2	5	A$	-
3	1	ACCCA$	A
4	4	CA$	?
5	3	CCA$	?
6	2	CCCA$	?

B

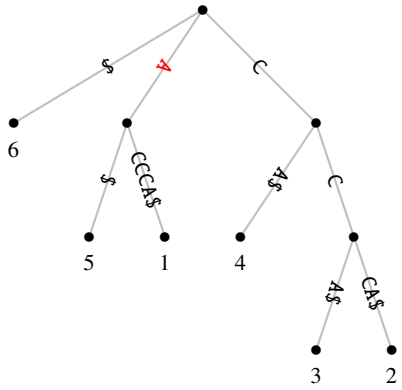


Fig. 3.12 Partial table of common prefixes (**A**) and their occurrence in the suffix tree (**B**)

Table 3.1 Incomplete table of longest common prefix lengths, the lcp array

index	sa	suf	cp	lcp
1	6	$	nd	-1
2	5	A$	-	0
3	1	ACCCA$	A	1
4	4	CA$	-	?
5	3	CCA$	C	?
6	2	CCCA$	CC	?

3.52 A suffix array is a much simpler structure than a suffix tree—a list of suffix starting positions compared to a set of branching nodes. However, the suffix tree is merely hidden in the suffix array. To uncover it, we need the *lengths* of the common prefixes, rather than the prefixes themselves. So in Table 3.1 we add the lcp array to the suffix array, where lcp[i] is the length of cp[i]. As the first entry in cp is undefined (there is no suffix to the left of it), we give it a length less than the smallest entry in the lcp array, -1. Can you fill in the missing lcp values?

3.53 The path label of a node in a suffix tree starts at the root and ends just above the node. Its length is also called the *string depth* of the node. Consider again the suffix tree in Fig. 3.11B. What are the string depths of its internal nodes?

3.54 Compare the string depths of the suffix tree in Fig. 3.11B with the longest common prefix lengths in Table 3.5. What do you notice?

3.55 The close relationship between suffix tree and suffix array means it is possible to reconstruct the tree from the array. To be more precise, for this reconstruction we need the suffix array enhanced by its lcp array. This combination of suffix array and lcp array is called an "enhanced suffix array". Given an enhanced suffix array, we first extend the lcp array by a "stop" entry, -1, as shown in Fig. 3.13A. In addition,

we write next to the lcp array an empty table with the three distinct string depths 2, 1, and 0 as headers.

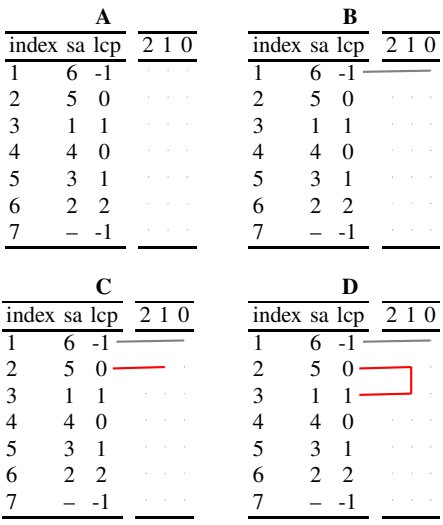


Fig. 3.13 The enhanced suffix array with an auxiliary table for reconstructing the corresponding suffix tree from lcp intervals

Now we traverse the lcp array to find the intervals in the sa array that correspond to the suffix tree. Starting from the top, we check at each position the relationship between the current entry in the lcp array, lcp$[i]$ and the next entry, lcp$[i+1]$ [35, p. 85ff]:

- If lcp$[i]$ = lcp$[i+1]$, do nothing.
- If lcp$[i]$ < lcp$[i+1]$, open one or more intervals. The number of intervals to open is lcp$[i+1]$ − lcp$[i]$. The string depths of the opened intervals are the values between lcp$[i]+1$ and lcp$[i+1]$. In our example, lcp$[1] = -1$ and lcp$[2] = 0$; since lcp$[1]$ < lcp$[2]$, we open $0 - -1 = 1$ interval with string depth $-1+1 = 0$. To denote this, we draw the gray line in Fig. 3.13B. Similarly, in the next step we observe lcp$[2]$ < lcp$[3]$ and open another lcp interval, as shown by the red line in Fig. 3.13C.
- If lcp$[i]$ > lcp$[i+1]$, close one or more intervals where string depth is greater than lcp$[i+1]$ and has occurred in the interval to be closed. In our example, we observe in Fig. 3.13D that lcp$[3]$ > lcp$[4]$. Since the string depth of the red, but not of the gray interval, is greater than lcp$[4]$ and it occurs in the interval under consideration, the red interval is closed, while the gray interval remains open.

Can you draw the remaining lcp intervals in Fig. 3.13?

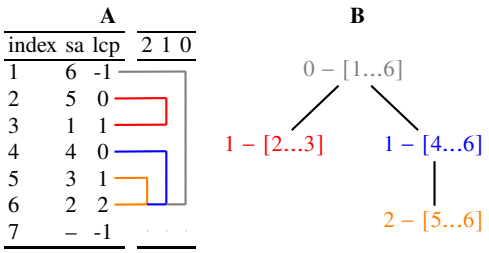


Fig. 3.14 Lcp intervals (**A**) and lcp interval tree (**B**) of `ACCCA$` using matching colors

3.56 The nested structure of the lcp intervals in Fig. 3.14A can be converted to a stripped version of the suffix tree, the lcp interval tree shown in Fig. 3.14B. Each node in the lcp interval tree has the format $\ell - [i...j]$, where ℓ is the string depth, i the start index, and j the end index in the suffix array. This lcp interval tree is the suffix tree in Fig. 3.11 stripped of its leaves. And while it is a bit tedious to go from an enhanced suffix array to an lcp tree, and hence to a suffix tree, it is easier to move from a suffix tree to the lcp interval tree. This is not a transformation we would carry out in practice, but it does help us get a feeling for the relationship between suffix trees and arrays. Can you draw the lcp interval tree of `CATGGCAT$`?

3.57 The program `drawSt` can automatically draw lcp interval trees. Can you draw the lcp interval tree of `CATGGCAT$`?

3.58 Let's study the relationship between enhanced suffix arrays and suffix trees with a longer example sequence, S = `GTAAACTATT$`. Can you compute its suffix array?

3.59 Can you add—by hand—the lcp values to the suffix array?

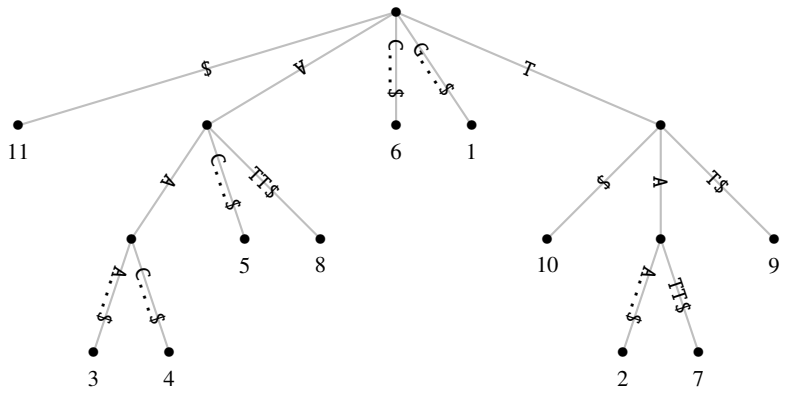


Fig. 3.15 Suffix tree of `GTAAACTATT$`

3.60 Fig. 3.15 shows the suffix tree of GTAAACTATT$. Can you reproduce it by hand and then with drawSt?

3.61 Can you construct the nested lcp intervals for your enhanced suffix array of GTAAACTATT$? Looking at the suffix tree in Fig. 3.15 might help.

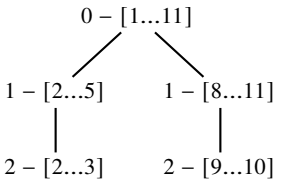


Fig. 3.16 Lcp interval tree of GTAAACTATT$

3.62 Fig. 3.16 shows the lcp interval tree for GTAAACTATT$. Can you reproduce it by hand and using drawSt?

3.63 The point to take away from what we have seen so far is that enhanced suffix arrays imply suffix trees. The efficient computation of lcp arrays is therefore central to the application of suffix trees. In the following couple of problems we see how to do this. The crucial insight is that the lcp value for a suffix $S[i...]$ imposes a lower bound of the lcp value of the suffix one step to the right, $S[i+1...]$. Consider, for example, the sequence ACCCACG. Its first suffix matches AC at position 5; from this we can conclude that its second suffix matches at least the C at position 6 from the previous match. In other words, if ℓ is the length of the common prefix of $S[i...]$, then $\ell - 1$ is the lower bound of the lcp for $S[i+1...]$. The emphasis here is on *lower bound*; in our example, the CC at the beginning of the second suffix matches the CC at the beginning of the third rather than just the first C guaranteed by the lower bound of 1. To use the lower bound insight, we traverse the suffix array in the same order in which the suffixes appear in the input sequence. The mapping between sa and S is called the inverse suffix array, isa, which is defined as

$$\text{isa}[\text{sa}[i]] = \text{sa}[\text{isa}[i]] = i.$$

Can you add the isa of ACCCA$ to the left of its suffix array in Fig. 3.14A?

3.64 The program isa.awk takes as input the sorted output from suf.awk, which it stores, together with the isa. Then it prints the suffix array with isa as an additional column.

Prog. 3.6 (isa.awk)

⟨*isa.awk*⟩≡

```
{
  n = NR
```

```
    sa[n] = $1
    suf[n] = $2
    ⟨Construct isa, Prog. 3.6⟩
  }
  END{
    printf "# i\tsa\tisa\tsuf\n" # print the output table
    for (i = 1; i <= n; i++)
      printf "%d\t%d\t%d\t%s\n", i, sa[i], isa[i], suf[i]
   }
```

Can you finish `isa.awk` and run it on `ACCCA$`?

Algorithm 1 Algorithm for computing lcp array [23]

Require: t {input sequence}
Require: n {length of t}
Require: sa {suffix array}
Ensure: lcp {array of lengths of longest common prefixes}
1: **for** $i \leftarrow 1$ to n **do**
2: isa[sa[i]] $\leftarrow i$ {construct inverse sa}
3: **end for**
4: lcp[1] $\leftarrow -1$ {initialize lcp}
5: $\ell \leftarrow 0$
6: **for** $i \leftarrow 1$ to n **do**
7: $j \leftarrow$ isa[i]
8: **if** $j > 1$ **then**
9: $k \leftarrow$ sa[$j-1$] {$t[k\ldots]$ is left-hand neighbor of $t[i\ldots]$ in sa}
10: **while** $t[i+\ell] = t[k+\ell]$ **do**
11: $\ell \leftarrow \ell + 1$
12: **end while**
13: lcp[j] $\leftarrow \ell$
14: $\ell \leftarrow \max(\ell - 1, 0)$ {ℓ cannot become negative}
15: **end if**
16: **end for**

3.65 Algorithm 1 shows how to compute the lcp array in linear time [23]. The inverse suffix array, isa, is computed in lines 1–2. Then we iterate over the positions in the input sequence, t. For each suffix $t[i\ldots]$, we require the suffix with the longest matching prefix, $t[k\ldots]$. We look up k in two steps. First, we look up the position of $t[i\ldots]$ in the suffix array. This is isa[i] and we call it j. The suffix most similar to $t[i\ldots]$ is the left hand neighbor of sa[j], so k is sa[$j-1$]. Now we can compare $t[i+\ell\ldots]$ and $t[k+\ell\ldots]$, where ℓ is the length of the matching prefix from the previous round, minus one. When we've found the first mismatch, we set lcp[j] to ℓ. We finish this round by decrementing ℓ by one.

We implement this algorithm in the program `esa.awk`, which takes as input the sorted output of `suf.awk`. It stores the suffixes and extracts the input sequence from them. Having parsed the input, we compute the isa and the lcp array, before we print the output, the enhanced suffix array.

Prog. 3.7 (`esa.awk`)

```
⟨esa.awk⟩≡
  {
    ⟨Store sa and suf, Prog. 3.7⟩
    ⟨Find input sequence, t, Prog. 3.7⟩
  }
  END {
    ⟨Compute isa, Prog. 3.7⟩
    ⟨Compute lcp, Prog. 3.7⟩
    ⟨Print enhanced suffix array, Prog. 3.7⟩
   }
```

Can you store the suffix array, sa, and the suffixes, suf?

3.66 Algorithm 1 requires access of the input sequence. Can you extract it from the suffixes?

3.67 Can you compute the isa?

3.68 To compute the lcp array, we split the input sequence into its character array and initialize the computation, which consists of a loop over all suffixes in t. For each suffix $t[i...]$, we look up its most similar partner in the text, $t[k...]$. Then we calculate the lcp value as the length of the match between these two suffixes.

```
⟨Compute lcp, Prog. 3.7⟩≡
  ⟨Split t into character array ta, Prog. 3.7⟩
  ⟨Initialize lcp computation, Prog. 3.7⟩
  for (i = 1; i <= n; i++) {
    ⟨Find suffix most similar to t[i...], t[k...], Prog. 3.7⟩
    ⟨Calculate lcp value, Prog. 3.7⟩
  }
```

Can you split the input sequence into its character array?

3.69 Can you initialize the lcp computation?

3.70 Next we look up the suffix most similar to $t[i...]$, $t[k...]$, via the inverse suffix array, as outlined in lines 7–9 of Algorithm 1. Here the `continue` command is useful, which says, "skip the rest of the loop". This skipping only makes sense if some condition is true:

```
if (cond) {
   continue
}
```

Since the action block of the if-clause consists of a single statement, we could leave out the curly brackets.

Can you find the suffix that is most similar to $t[i...]$?

3.71 Now we match the prefixes of $t[i...]$ and $t[k...]$ to get the desired lcp value, $\text{lcp}[j]$. This is done in a `while` loop:

```
while (cond) {
   do something
}
```

Like in the `if`-clause, action blocks for `while` with more than one statement must be in curly brackets, action blocks with one statement can be. The minimum of $\text{lcp}[j]$ is ℓ, the match length from the previous round minus one. Can you calculate the lcp value?

3.72 Can you print the enhanced suffix array?

3.73 Can you apply `esa.awk` to `ACCCA$`?

3.74 Now we apply our programs `suf.awk` and `esa.awk` to real sequences to identify longest repeats. We do this using the script `lrep.sh`.

Prog. 3.8 (`lrep.sh`)

⟨*lrep.sh*⟩≡

```
awk -f suf.awk $1 |
    sort -k 2 |
    awk -f esa.awk |
    tail -n +2 |
    sort -k 3 -n -r |
    head -n 1
```

Can you explain what it does?

3.75 What is the longest repeat in the *Adh* region of *D. guanche*?

3.76 The program `randomizeSeq` randomizes an input sequence, which is useful to judge whether a repeat might be due to chance. Is the longest repeat we just found longer than expected by chance alone? Can you formally test the null hypothesis that the observed longest repeat is merely due to chance?

3.77 We've seen that the longest repeat just identified occurs at position 988 of the *Adh* region in *D. guanche*. Where else does it occur?

3.78 We've already used the program `repeater` to find longest repeats. Can you use it to double-check the longest repeat in the *Adh* region of *D. guanche*?

3.79 We have established that our suffix array pipeline works as intended. Can you use it to find the longest repeat in the *Adh* region of *D. melanogaster*?

3.80 Where does the longest repeat just identified in the *Adh* region of *D. melanogaster* occur apart from 3908?

3.81 Next, we search for the longest repeat in the combined *Adh* regions of *D. melanogaster* and *D. guanche*. But before we run `lrep.sh`, can you place a lower bound on the length of the repeat we'll find?

3.82 What is the longest repeat in the combined *Adh* regions of *D. melanogaster* and *D. guanche*? Can you explain your result?

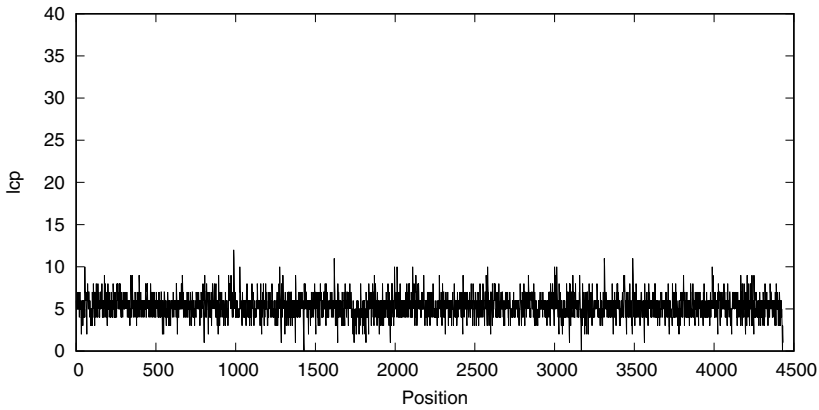


Fig. 3.17 Lcp values along the *Adh* region of *D. guanche*

3.83 The entries in an lcp array tell us at every position in the sequence how far we can walk without becoming unique. Let's plot them with the script `plotLcp.sh`. The expression `$@` in its first line means every token on the command line. Can you explain what the rest of `plotLcp.sh` does?

Prog. 3.9 (`plotLcp.sh`)

⟨*plotLcp.sh*⟩≡
```
awk -f suf.awk $@ |
    sort -k 2 |
    awk -f esa.awk |
    tail -n +3 |
    cut -f 2,3 |
    sort -n
```

3.84 Fig. 3.17 shows the lcp values of the *Adh* region of *D. guanche*. The y range is adjusted to include the maximum repeat length we have observed. None of the values seems to be particularly large, which agrees with our conclusion from statistics that the longest match may well be due to chance. Can you reproduce Fig. 3.17?

3.85 Fig. 3.18 shows the lcp values along the *Adh* region of *D. melanogaster*. Can you reproduce it?

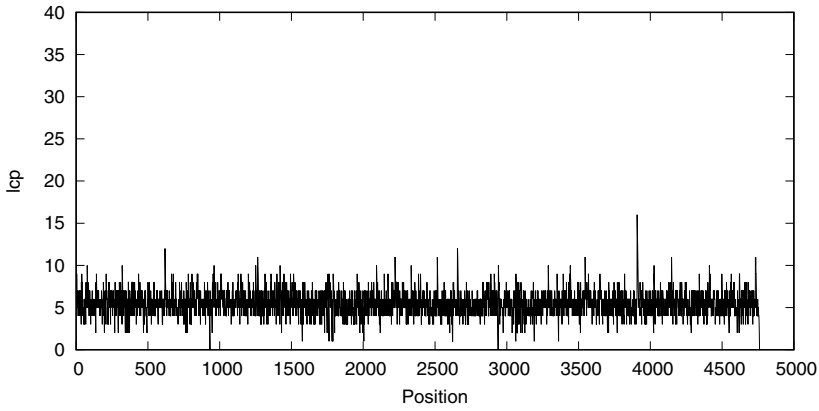


Fig. 3.18 Lcp values along the *Adh* region of *D. melanogaster*

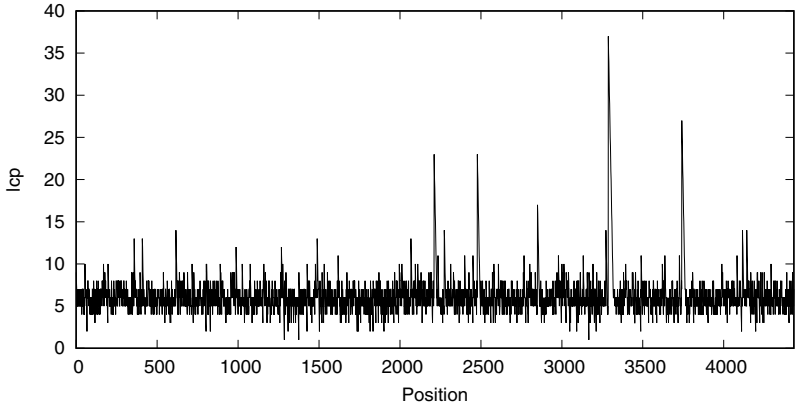


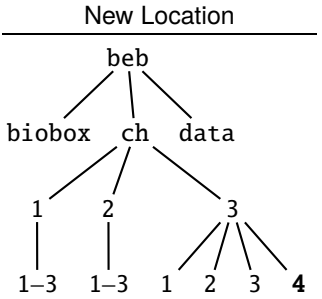
Fig. 3.19 Lcp values along the *Adh* region of *D. guanche* on the background of *D. melanogaster*

3.86 Fig. 3.19 shows the plot we've been working towards, the lcp values along the *Adh* region of *D. guanche* on the background of *D. melanogaster*. Notice the peak of 37. Can you reproduce Fig. 3.19?

3.4 Text Compression

The ability to compress data underlies computer applications ranging from mp3 players to read mappers. Of the many methods for data compression, we look at a classic combination of three steps, sorting, reordering, and encoding [6]. The

sorting is intimately connected to a sorted structure we already know, the suffix array. The reordering converts runs of any character into runs of zeros, and the encoding squeezes unused bits out of bytes.



3.87 Can you make the directory for this section and change into it?

Sorting

New Terms		
Burrows-Wheeler transform	runs of characters	string rotation
`bwt`		

Compressed		Original		Sorted		Compressed
$A_4C_2T_5C_2$	↔	AAAACCTTTTTCC	→	AAAACCCCTTTTT	↔	$A_4C_4T_5$

Fig. 3.20 Compression through run length encoding

3.88 Consider the sequence AAAACCTTTTTCC marked *original* in Fig. 3.20. It can be compressed by looking for runs of individual characters and encoding the initial four As, for example, as A_4. This kind of compression is easy to reverse, hence the double-headed arrow to the left in Fig. 3.20. Run encoding becomes more effective as the number of characters in runs increases. The ultimate list of runs would be obtained by sorting the characters, but unfortunately this is irreversible, as indicated by the arrow pointing only to the right in Fig. 3.20. However, there is a way to *almost* sort a version of the text that turns out to be highly compressible and reversible. We begin with a text,

tobeornottobe$

and write its "rotations" by shifting the text one position to the left and attaching the character that falls off at the beginning, t, to the end, as if the text were written on a ring:

```
obeornottobe$t
```

Can you write down the first five string rotations of `thatisthequestion$`?

3.89 Fig. 3.21A shows all the string rotations of `tobeornottobe$`. It was generated using the program `rotate.awk`.

Prog. 3.10 (`rotate.awk`)

⟨*rotate.awk*⟩≡

```
    !/^>/ {
    t = t $0
  }
  END {
    n = split(t, ta, "")
    for (i = 1; i <= n; i++) {
      ⟨Print string rotation, Prog. 3.10⟩
    }
  }
```

Can you print a string rotation and thus finish `rotate.awk` (hint: `%`)?

A	B
tobeornottobe$	$tobeornottobe
obeornottobe$t	be$tobeornotto
beornottobe$to	beornottobe$to
eornottobe$tob	e$tobeornottob
ornottobe$tobe	eornottobe$tob
rnottobe$tobeo	nottobe$tobeor
nottobe$tobeor	obe$tobeornott
ottobe$tobeorn	obeornottobe$t
ttobe$tobeorno	ornottobe$tobe
tobe$tobeornot	ottobe$tobeorn
obe$tobeornott	rnottobe$tobeo
be$tobeornotto	tobe$tobeornot
e$tobeornottob	tobeornottobe$
$tobeornottobe	ttobe$tobeorno

Fig. 3.21 Rotations (**A**) and sorted rotations (**B**) of `tobeornottobe$`

3.90 Once we've got the rotations, we sort them to get Fig. 3.21B. Can you reproduce it?

3.91 In the sorted rotation in Fig. 3.21B, the first and last columns are of particular interest. The first column contains the characters of the text alphabetically sorted. We would like to submit them to run length encoding, but have already realized that we can't convert that back into the original text. But the last column *can* be converted back into the original, as we shall see later. This last column is known

as the Burrows-Wheeler transform in honor of its two discoverers [6]. What is the Burrows-Wheeler transform of `tobeornottobe$`?

3.92 How is the Burrows-Wheeler transform useful? To find out, we need to analyze longer texts. For longer texts it is inconvenient to read off the last column of a set of sorted rotations. So we look again at the rotations in Fig. 3.21A and realize that up to the sentinel, they are the suffixes. And after sorting in Fig. 3.21B, they form a suffix array. Now, the last column in the rotation is the position just to the left of the start of the suffix. In other words, we can read the Burrows-Wheeler transform of text t off the suffix array of t:

$$\mathrm{bwt}[i] = t[\mathrm{sa}[i] - 1] \tag{3.1}$$

for all $\mathrm{sa}[i] > 1$. Can you encode `tobeornottobe$` into its Burrows-Wheeler transform using `suf.awk` and `sort`?

3.93 The program `bwt` computes the Burrows-Wheeler transform of an input sequence. Let's take for now as our "sequence" Shakespeare's *Hamlet* in the file `hamlet.fasta`. Browse through it with `less`. Then count its characters. What is the most frequent and the least frequent character in *Hamlet*?

3.94 Use `bwt` to transform *Hamlet* and browse the transformation with an eye for runs of characters. What do you notice?

3.95 Let's quantify our impression that more characters end up in runs with the Burrows-Wheeler transform. Our example sequence in Fig. 3.20 contains two runs of length 2, one run of length 4, and one run of length 5. The program `runs.awk` counts the number of runs of each length.

Prog. 3.11 (`runs.awk`)

⟨*runs.awk*⟩≡
```
!/^>/ {
  t = t $0
}
END {
  n = split(t, ta, "")
  ⟨Count run lengths, Prog. 3.11⟩
  ⟨Print run lengths and counts, Prog. 3.11⟩
}
```

Look at the sequence in Fig. 3.20 and think about how you'd count the lengths of its runs. Can you implement this part of `runs.awk`?

3.96 Can you print the run lengths and their counts?

3.97 What is the length of the longest run in *Hamlet*?

3.98 What is the length of the longest run in the Burrows-Wheeler transform of *Hamlet*?

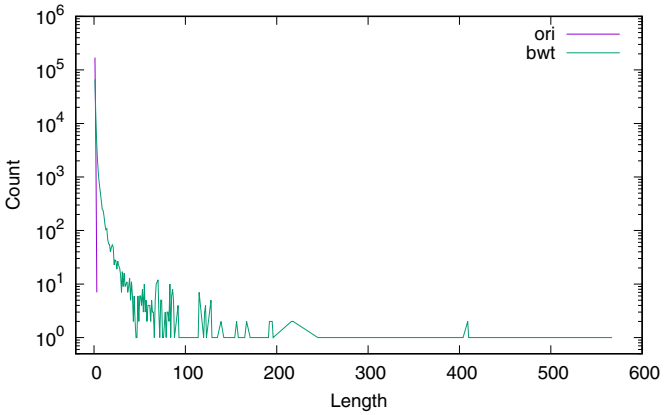


Fig. 3.22 The distribution of run lengths in the original version of *Hamlet* (*ori*) and after Burrows-Wheeler transformation (*bwt*)

3.99 Fig. 3.22 shows the distribution of run lengths in the original *Hamlet* and in its Burrows-Wheeler transform. Clearly, the transform has a huge effect on the amount of sequence located in runs of characters. Can you think of why this might be?

3.100 Can you reproduce Fig. 3.22?

3.101 Let's abolish the context preference of characters by randomizing *Hamlet* using `randomizeSeq`. What is now the effect of the Burrows-Wheeler transform?

Table 3.2 Decoding the Burrows-Wheeler transform `eoobbrttenot$o`. Sorted characters give the start, S, of the rotation (**A**); juxtaposition of S with the ends of the rotation, E (**B**); track the character occurrences in S and E (**C**)

A	**B**	**C**
S	S E	S E
$	$ e	$_1$ e$_1$
b	b o	b$_1$ o$_1$
b	b o	b$_2$ o$_2$
e	e b	e$_1$ b$_1$
e	e b	e$_2$ b$_2$
n	n r	n$_1$ r$_1$
o	o t	o$_1$ t$_1$
o	o t	o$_2$ t$_2$
o	o e	o$_3$ e$_2$
o	o n	o$_4$ n$_1$
r	r o	r$_1$ o$_3$
t	t t	t$_1$ t$_3$
t	t $	t$_2$ $$_1$
t	t o	t$_3$ o$_4$

3.102 Say, we are given `eoobbrttenot$o` and all we know about it is that it's a Burrows-Wheeler transform. How can we decode it? The answer is, by sorting and counting. First, we sort the transform's characters and write them in one column (Table 3.2A). These are the starting characters of the sorted rotations, hence we label that column *S*. Then we write the transform, which consists of the ends of the sorted rotations, in a column next to it, column *E* (Table 3.2B).

Now we count the occurrences of each character in each column. So in Table 3.2C the first `$` in *S* becomes $\texttt{\$}_1$, the first `b` becomes $\texttt{b}_1$, the second `b` $\texttt{b}_2$, and so on for the rest of column *S*. Then we count the character occurrences in column *E*. To decode the first character, we look up the rotation that ends in `$`; it starts with $\texttt{t}_2$. Then we look for the rotation that ends with $\texttt{t}_2$, it starts with $\texttt{o}_2$. Then we look for the rotation that ends with $\texttt{o}_2$, $\texttt{b}_2$, and so on, until we find `$`. Can you decode `nhhuttttoieie$ssaq`? You can check your result with the decoding option of `bwt`.

Reordering with Move to Front

New Terms	
move to front	`mtf`

3.103 We've seen that the Burrows-Wheeler transform can reversibly increase the number of runs of characters in a sequence. So let's think a bit more about runs of characters. Fig. 3.20 suggests that we can summarize a run like `AAAA` as $\texttt{A}_4$. But, there is also another way to take advantage of runs. We could encode the example in Fig. 3.20 by just writing the first character of each run, followed by a place holder, say a zero:

`AAAACCTTTTTCC` → `A000C0T0000C0`

This would be particularly useful if we could encode zeros using less space than an ordinary character, which occupies eight bits, or one byte. In fact, there is a way to squeeze bits from bytes, as we shall see later. For now we concentrate on the placeholder notation. To use it, we begin by mapping each character in the alphabet of the input to a number,

Character	Number
`A`	`0`
`C`	`1`
`T`	`2`

Can you write down such a character mapping for `tobeornottobe`?

3.104 To encode our example sequence `AAAACCTTTTTCC` (Fig. 3.20), we take its first character, `A`, look up its number in the alphabet, `0`, and write down the `0`. We repeat this until we get to the `C`. Its position in the alphabet is `1`, so our encoding now becomes

```
0 0 0 0 1
```

As C is not at the front of the alphabet, we move it to the front to get

Character	Number
C	0
A	1
T	2

The next C is 0. Then we have the T, a 2, and we move it to the front to get

Character	Number
T	0
C	1
A	2

The next four Ts are zeros. At the end we have a 1 for the first C and a 0 for the second. So our result is

```
0 0 0 0 1 0 2 0 0 0 0 1 0
```

Can you encode TATAAATTTT?

3.105 Our encoding into small integers relies on moving characters to the front of the alphabet, hence its official name "move to front" [6]. It is implemented in the program `mtf`. Can you use `mtf` to encode `thatisthequestion` and its Burrows-Wheeler transform?

3.106 What is the most frequent move to front code in *Hamlet*?

3.107 What is the most frequent move to front code in *Hamlet* after Burrows-Wheeler transform?

3.108 Fig. 3.23 shows the frequency distribution of move to front codes in the original *Hamlet* and in its Burrows-Wheeler transform. As we might expect, the two distributions are very different. Can you write a script to format the codes, `fc.sh`, and use it to reproduce Fig. 3.23?

3.109 Like the Burrows-Wheeler transform, move to front is only useful if reversible. Decoding starts with the encoded sequence, say

```
0 0 0 0 1 0 2 0 0 0 0 1 0
```

and the alphabet used,

Character	Number
A	0
C	1
T	2

The first four codes are converted to A, followed by a C, which is moved to front, and so on. What is the decoding of

```
0 0 0 1 0 1 0 1 0 1
```

if C is 0 and G is 1? You can check your result with the decoding option of `mtf`.

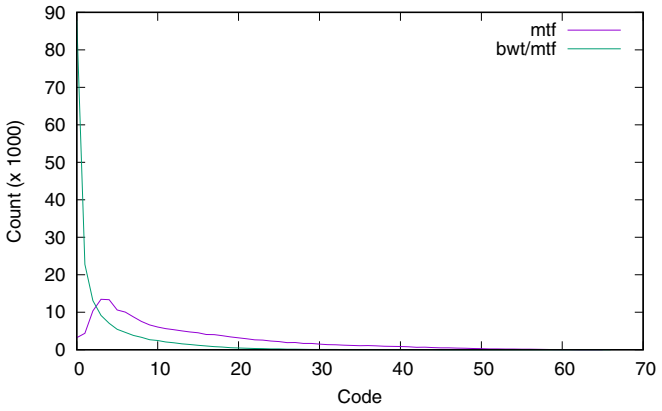


Fig. 3.23 Frequency distribution of code counts in *Hamlet* after move to front only (*mtf*) or after Burrows-Wheeler transform and move to front (*bwt/mtf*)

Squeezing Bits from Bytes

New Terms		
`bowtie`	fixed length code	`huff`
`bunzip2`	`gunzip`	`hut`
`bwa`	`gzip`	prefix code
`bzip2`	Huffman code	variable length code

3.110 When storing characters, each residue is usually stored as a byte, which occupies four bits. A code where each element occupies the same amount of space is called a "fixed length code". How many bits are required to store the nucleotides of the *M. genitalium* genome with the fixed length code of one byte per character?

Table 3.3 Fixed length code of the DNA nucleotides

Character	Code
`A`	`00`
`C`	`01`
`G`	`10`
`T`	`11`

3.111 Genomes consist of only four characters, which can be encoded by 2 bits each, as shown in Table 3.3. How many bits does the genome of *M. genitalium* occupy with this new fixed length code?

3.112 Consider this run of bits,

```
11011110000011101011
```

Can you decode it using Table 3.3?

Table 3.4 Two variable length codes for DNA; **A** is a prefix code, **B** isn't

A	
Character	Code
A	0
C	100
G	101
T	11

B	
Character	Code
A	0
C	000
G	101
T	11

3.113 We can save even more space if we give up the requirement that each code has the same length. Table 3.4A shows such a variable length code for DNA. How much space does the genome of *M. genitalium* now occupy? How much space do we save compared to two bits per nucleotide?

3.114 Consider this run of bits

```
11001011111011110
```

Can you decode it with Table 3.4A?

3.115 The codes in Table 3.4A and B differ by one bit in the code for C. This makes the code for A a prefix of the code for C. To see the problems that causes, try decoding

```
11001100011
```

Where does it break down?

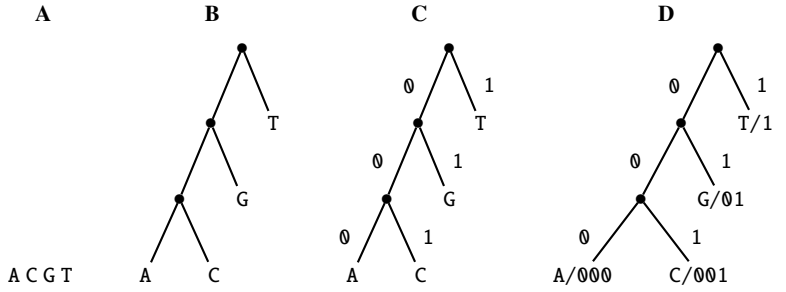


Fig. 3.24 Construction of a prefix code for DNA

3.116 Variable length codes only work if no code is prefix of another. Such codes are called "prefix codes". The construction of prefix codes is based on binary trees. We start with the characters we wish to encode as leaves (Fig. 3.24A). Then we construct an arbitrary tree (Fig. 3.24B), and mark its left branches 0, its right branches 1 (Fig. 3.24C). Now we read the code of each character as its path label (Fig. 3.24D). Can you construct an alternative prefix code for DNA?

3.117 Not all prefix codes are equally effective. How many bits are required by the code in Fig. 3.24D to store the genome of *M. genitalium*? Compare that to the code in Table 3.4A.

3.118 Can you come up with an even worse code using the tree notation?

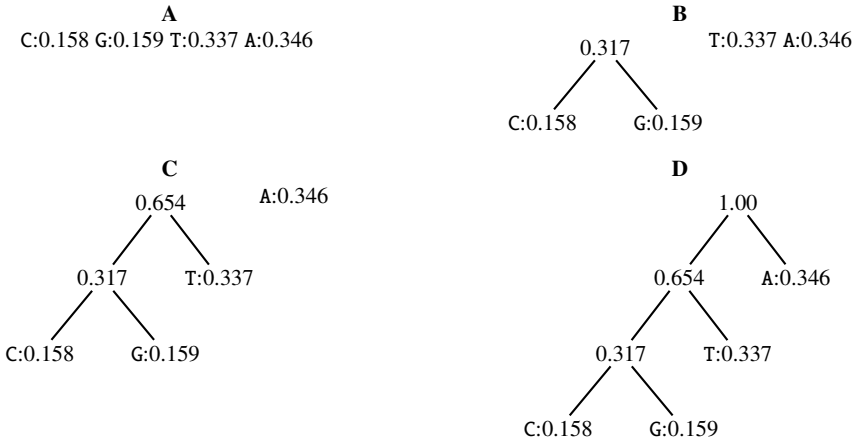


Fig. 3.25 Construct a Huffman tree of DNA from weighted nodes; **A** shows just the leaves, **B** the first clustering, **C** the second, and **D** the last

3.119 To find an optimal code, we construct the tree from weighted nodes, where the weight of a node is the frequency of its character as shown in Fig. 3.25A. Then we merge the two lightest nodes, C and G, into a node weighing the sum of their weights (Fig. 3.25B). We now consider the remaining three nodes with weights 0.317, 0.337, and 0.346. As before, we join the lightest pair with weights 0.317 and 0.337 (Fig. 3.25C). The remaining two nodes are joined in the root to complete tree construction (Fig. 3.25D). A tree like this can now be labeled and the codes read off, as we saw earlier (Fig. 3.24D). This method of finding optimal codes was published in 1952 by David Huffman [21] and the codes are thus called Huffman codes [8, p.385ff]. Can you construct the Huffman codes for the *Adh* region of *D. melanogaster*?

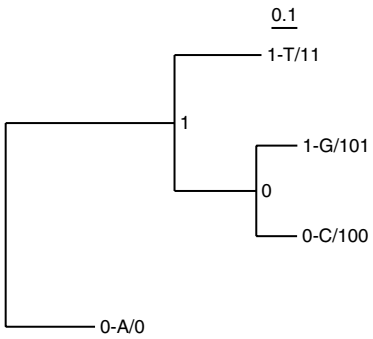


Fig. 3.26 Huffman tree for the genome of *M. genitalium*

3.120 The program `hut` constructs the trees underlying Huffman codes, let's call them Huffman trees. Fig. 3.26 shows the Huffman tree for the genome of *M. genitalium* with branch lengths that are proportional to node weights. Can you reproduce Fig. 3.26?

3.121 Huffman trees are curious structures. Can you plot the Huffman tree for the proteome of *M. genitalium* (`mgProteome.fasta`)?

3.122 A Huffman tree is constructed from residue frequencies. Can you determine the most frequent amino acid or amino acids in the proteome of *M. genitalium* from its Huffman tree? You can check your answer with `cres`.

3.123 The program `huff` takes as input a Huffman tree and a sequence. It then prints a stream of zeros and ones to show the corresponding bit encoding. Can you compute the bit encoding of the *M. genitalium* genome? How many bits does it consist of?

3.124 If we are just interested in the number of bits in a Huffman encoding, we can run `hut -b`. This allows us to calculate the compression ratio of Huffman encoding by dividing the number of bits in the original sequence by the number of bits in the encoding. What is that ratio for *Hamlet*?

3.125 As with any encoding, we need to also be able to decode a Huffman encoding. Consider this sequence of bits encoded under the Huffman tree in Fig. 3.26:

```
0100111111111001000
```

The first `0` in the sequence leads to `A` in the Huffman tree. Then we parse `100` for a `C`. Can you decode the rest of the sequence? You can check your result with the decoding option of `huff`.

3.126 The methods we have covered, Burrows-Wheeler transform with `bwt`, move to front with `mtf`, and Huffman encoding with `hut`, are meant to be applied in that order. So we'd like to write a pipeline like

```
bwt hamlet.fasta | mtf | hut -b
```

However, we've already seen that the output of `mtf` is a blank-delimited list of numbers, while `hut` takes a stream of characters as input. The gap between these two programs is filled by `num2char`. Can you calculate the compression ratio of *Hamlet* when subjected to Burrows-Wheeler transform, move to front, and Huffman encoding?

3.127 Our compression scheme does not include explicit encoding of repeats, a component of the popular compression program `gzip`. Still, let's find out the compression ratio of `gzip` and compare it to the hypothetical ratio achieved with our combination of Burrows-Wheeler transform, move to front, and Huffman encoding. To make this a fair comparison, we first remove the FASTA header and the newlines.

⟨*cli*⟩+≡

```
tail -n +2 hamlet.fasta | tr -d '\n' > hamlet.txt
```

What is the compression ratio of `gzip` on *Hamlet*? `gunzip` unzips the result of `gzip`.

3.128 While `gzip` does not use the Burrows-Wheeler transform, `bzip2` does. In fact, a number of programs with prominent "b" and "w" in their names are based on the Burrows-Wheeler transform, for example the read mappers `bowtie` [32] and `bwa` [33]. What is the compression ratio of `bzip2` on `hamlet.txt`? `bunzip2` unzips the result of `bzip2`.

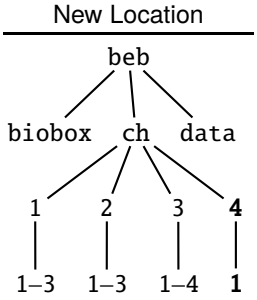
3.129 We've played with a play quite a bit, so let's return from *Hamlet* to the genome of *M. genitalium*. What are the compression ratios of `gzip`, `bzip2` and our approach?

Chapter 4
Fast Alignment

We've seen that optimal alignment is slow but accurate while exact matching is fast but cannot account for mutations. In this chapter we tweak exact matching to get fast alignments. We begin with the easiest case, global alignment between a query and a subject of similar lengths. Then we move to the best known application of fast alignment, local alignment between a short query and a long subject as implemented in Blast. Sometimes it is useful to align only the query in full, for example, when aligning a PCR primer to its target genome, and we show how to do such global/local or "glocal" alignment efficiently. The reads of a shotgun sequencing run are assembled by overlapping them, and overlap alignment is the next kind of alignment we encounter. Finally, we generalize from aligning pairs of sequences to aligning multiple sequences.

4.1 Global

We've used the optimal global alignment method implemented in `al` to align the *Adh* regions of *D. melanogaster* and *D. guanche*, sequences that are a few kilobases long. But how can we align, say, bacterial genomes that are a few megabases long? By using exact matching.



4.1 Can you make a new directory for this section?

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_4

New Terms		
data streams	`mummer`	`nucmer`
`dnaDist`		

4.2 We've said optimal alignment is slow, so let's measure how slow using `date` (or `gdate` on macOS). Can you recall how to time `ls`?

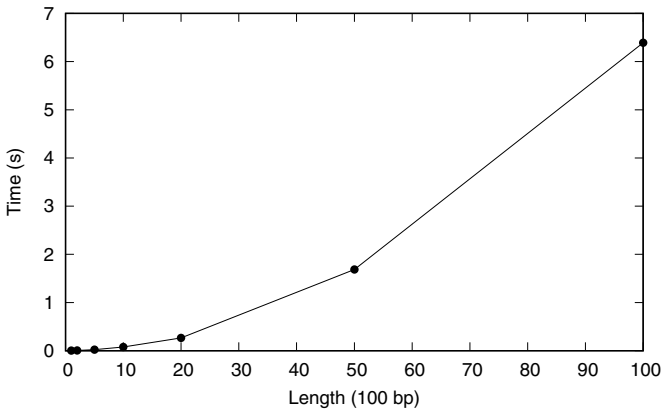


Fig. 4.1 Run time of optimal global alignment implemented in `al` as a function of sequence length

4.3 Fig. 4.1 shows the run time of our optimal aligner `al` as a function of the length of both input sequences. Can you write the script `rtAl.sh` to reproduce it?

4.4 Mummer is a software package for quickly aligning pairs of genome sequences. The acronym MUM in its name stands for maximal unique match. As we saw when working with `repeater`, maximal matches cannot be extended to the left or the right. But in contrast to `repeater`, `mummer` looks for unique maximal matches.

Fig. 4.2 shows the MUMs found by `mummer` for the *Adh* region of *D. melanogaster* and *D. guanche*, which should by now look familiar. The output of `mummer` can be converted to `plotSeg` input with `mum2plot`. Can you reproduce Fig. 4.2?

4.5 Let's look at the first five MUMs in the *Adh* region.

⟨*cli*⟩+≡

```
mummer dmAdhAdhdup.fasta dgAdhAdhdup.fasta | head -n 5
```

```
> DGADHDUP
     679       549        22
    2251      2211        23
    2410      2370        26
    2452      2412        23
```

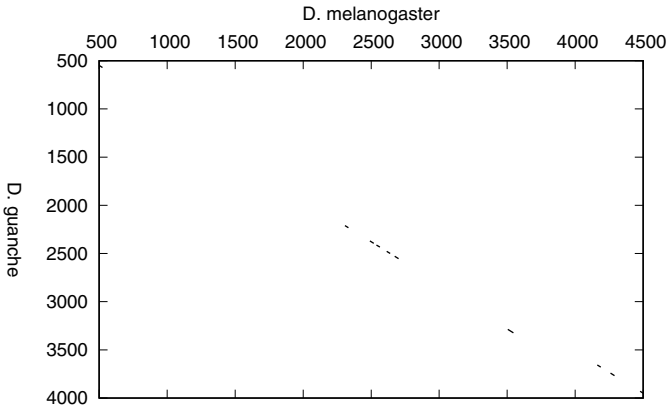


Fig. 4.2 `mummer` plot of the *Adh* region in *D. melanogaster* and *D. guanche*

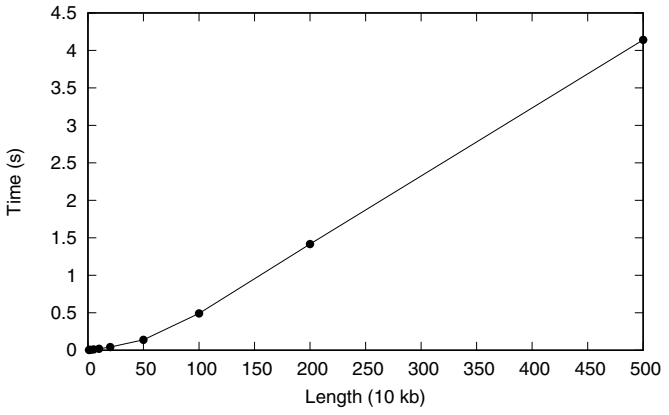


Fig. 4.3 Run time of `mummer` as a function of sequence length

The three columns indicate the start in the first sequence, the start in the other sequence, and the length of the MUM. Can you cut out the very first MUM and find it in the other sequence?

4.6 How long are the shortest and longest MUMs in the *Adh* region?

4.7 `mummer` writes copious output, which we don't always need. But what happens when you try to throw it away by redirecting it to the null device?

⟨*cli*⟩+≡

```
mummer r.fasta r.fasta > /dev/null
```

4.8 The data streams of the shell are numbered. Standard input is 0, standard output is 1, and standard error is 2. By default, > redirects the standard output, but we can redirect the standard error instead.

⟨*cli*⟩+≡
```
mummer r.fasta r.fasta 2> /dev/null
```

What happens now?

4.9 We can also redirect standard error *and* standard out.

⟨*cli*⟩+≡
```
mummer r.fasta r.fasta &> /dev/null
```

What do you observe now?

4.10 Fig. 4.3 shows the run time of `mummer` as a function of sequence length. For 10 kb `al` takes 6.9 s, `mummer` a mere 0.006 s. In other words, on a pair of 10 kb sequences `mummer` is 1000 times faster than `al`. Can you write a script `rtMummer.sh` to reproduce Fig. 4.3?

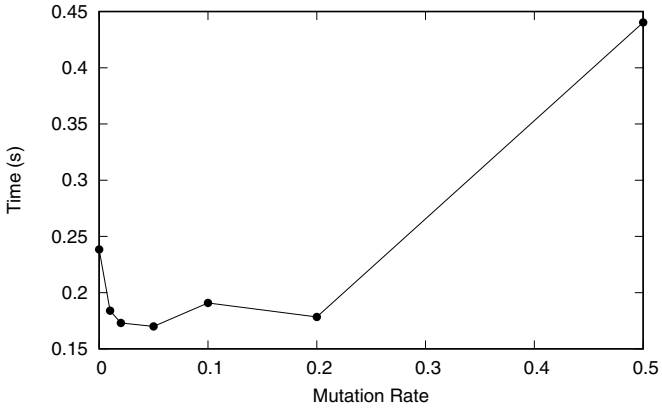


Fig. 4.4 The run time of `mummer` on pairs of 500 kb sequences as a function of mutation rate

4.11 We've already seen that `mummer` works with exact matches. In fact, when aligning two identical sequences, there is only a single match. What happens to the run time as we increase the number of matches by mutating the sequence? Fig. 4.4 shows that only very high mutation rates have a significant effect on run time. The program `mutator` mutates a sequence. Can you use it in a script `rtMummer2.sh` to reproduce Fig. 4.4?

4.12 In addition to drawing dot plots, the Mummer package contains programs for counting the number of single nucleotide polymorphisms, or SNPs, in an alignment. This goes well beyond the capabilities of ordinary dot plots. The program for counting

SNPs is `nucmer`, which is used in conjunction with `show-snps`. Let's start from our 500 kb random sequence, and mutate 1% of its positions, in other words, we expect 5000 mutations. Then we run `nucmer`, which generates the file `out.delta`, from which `show-snps` extracts the SNPs. The output of `show-snps` is a table with one line per SNP. If we cut off the table header, we can directly count the SNPs.

⟨*cli*⟩+≡

```
mutator r1.fasta > r2.fasta
nucmer r1.fasta r2.fasta
show-snps out.delta | tail -n +6 | wc -l
```

How many SNPs do you find between `r1.fasta` and `r2.fasta`?

4.13 We can exactly count the number of mismatches between two sequences using the program `dnaDist`. What is the number of SNPs `nucmer` should have found between `r1.fasta` and `r2.fasta`?

4.14 What happens to the SNP count of `nucmer` as you ratchet up the mutation rate in a script `mutate.sh`?

4.15 Clearly, mutation affects the number of MUMs. Fig. 4.5 shows the number of MUMs as a function of mutation rate. Without mutation, there is only a single MUM. With very high mutation, there are only a few, and in between there are many. Can you write a script `numMum.sh` to reproduce Fig. 4.5?

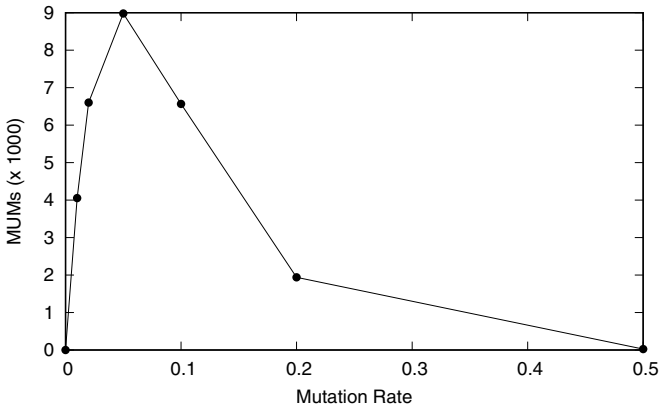


Fig. 4.5 The number of MUMs as a function of mutation rate

4.16 We close our investigation of the Mummer package by aligning the genomes of two strains of *E. coli*, K12 and O157H7. These are contained in files `ecoliK12.fasta` and `ecoliO157H7.fasta`, and Fig. 4.6 shows their `mummer` dot plot. Notice the large inversion at 1.5 Mb in K12 and 2 Mb in OH157H7. Can you reproduce this plot?

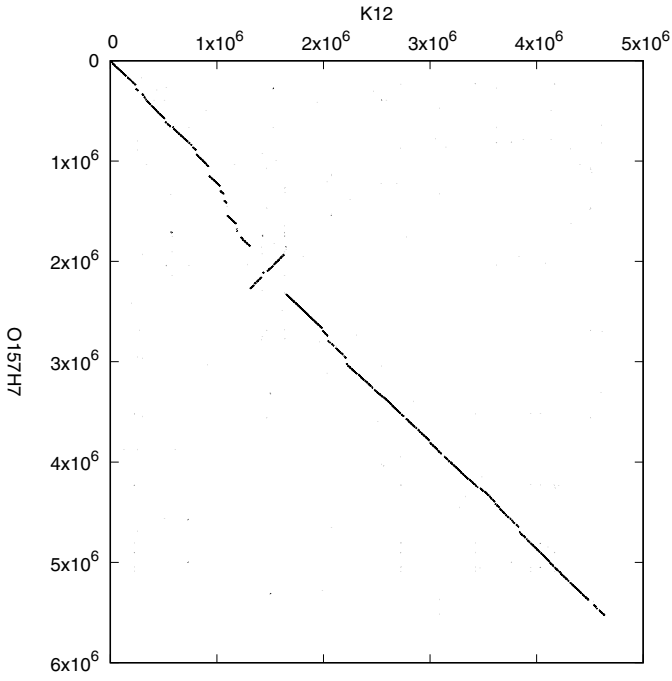
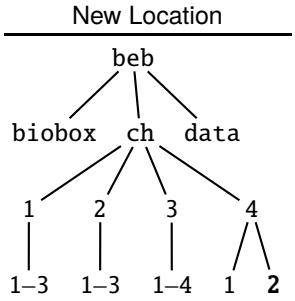


Fig. 4.6 `mummer` plot of two *E. coli* strains, K12 and O157H7

4.2 Local

Fast local alignment is epitomized by Blast, the "basic local alignments tool", which we already encountered when we talked about keyword trees. We start our exploration of Blast with the simple version we used previously, `sblast`, which produces ungapped alignments. Then we look at the Blast program maintained by the NCBI. This is designed for searching potentially very large databases. In such a search for needles in haystacks it becomes important to assess the significance of an alignment, for which Blast supplies its own statistics.



4.17 Can you make the directory for this section and change into it?

Simple Blast

New Terms		
extension steps	high-scoring pair	word list

4.18 Blast is a fast version of the optimal local alignment implemented in `al`. So let's start with `al` and use it to calculate something we've seen before, the optimal local alignment between the *Adh* regions of *D. melanogaster* and *D. guanche*. Where is it located again, and what is its score?

4.19 What is the best alignment of the *Adh* region returned by the simple Blast algorithm implemented in `sblast`? Is it identical to that returned by `al`?

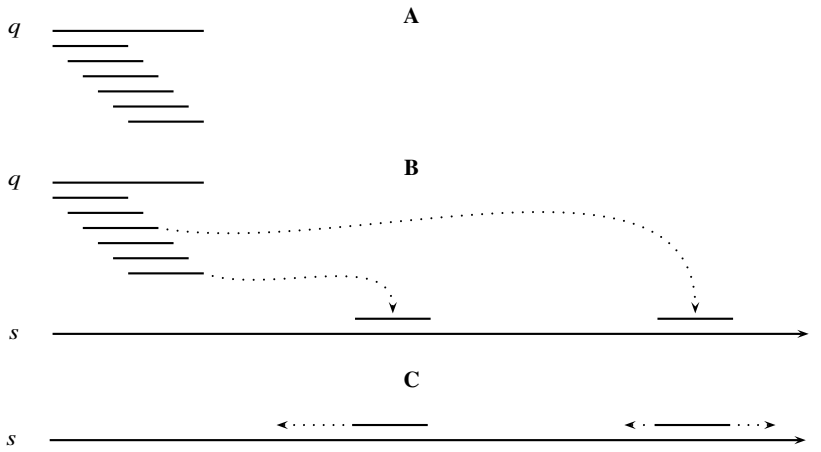


Fig. 4.7 Blast algorithm. Preprocess the query sequence, q, into overlapping words (**A**); search the words in the subject, s, (**B**); extend matches until the score cannot be improved any further to yield high-scoring pairs (**C**)

4.20 The first step in the Blast algorithm is to break the query into overlapping words (Fig. 4.7A). The words are then searched for in the subject (Fig. 4.7B) and we already saw that keyword trees make this step fast. The matches are extended to give the set of alignments, or "high-scoring pairs" in Blast nomenclature (Fig. 4.7C). They are filtered according to score and perhaps other criteria and those that pass are returned as the final set of alignments. Let's look in more detail at the first step,

the conversion of the query into a word list. How long are the words used by `sblast` and how many of them are there when using the *D. melanogaster Adh* as query?

4.21 `mummer` uses unique matches. Are the words used by `sblast` unique when searching for *D. melanogaster Adh*?

4.22 After breaking the query into words, the words are searched for in the subject (Fig. 4.7B). This search is carried out on the forward and the reverse strand of the subject. We've already seen that the program `keyMat` can search multiple patterns simultaneously. These can be supplied either on the command line or as a FASTA file, whichever is more convenient. How many matches of the words in *melanogaster Adh* are found in *guanche*?

4.23 `sblast` returns three alignments for the *Adh* region. Are they the same as the top three alignments returned by `al`?

4.24 Clearly, it is possible that the extension step (Fig. 4.7C) breaks off too early. Why might that be? The extension is governed by the number of steps allowed that don't improve on the last best score. If this parameter is too small, alignments may get truncated. If this parameter is too large, the algorithm is slowed down unnecessarily. By increasing the number of idle extension steps we can recover the full second alignment with `sblast`. What is the smallest number of idle extensions that gives the optimal result?

4.25 Blast is all about speed. How long does `sblast` take to find the top three *Adh* alignments compared to `al`?

4.26 The files `dmChr*.fasta` contain the genome sequence of *D. melanogaster*. How long is it?

4.27 How many matches to the *melanogaster* query words are contained in the *melanogaster* genome?

4.28 Where is the *Adh* region in the genome of *D. melanogaster*?

4.29 How long does `sblast` take to find the *Adh* region in the genome of *D. melanogaster*?

4.30 What happens if we take the *guanche Adh* as query?

NCBI Blast

New Terms
`blastn` `makeblastdb`

4.31 NCBI Blast consists of a set of programs, we start with nucleotide Blast implemented in `blastn`. Which alignments does it return for the *Adh* regions of *D. melanogaster* and *D. guanche*?

4.32 The default mode of `blastn` is called megablast. It is very fast, but—as we just saw—not particularly sensitive. A more sensitive mode is somewhat confusingly called `blastn` and invoked with the `-task` option.

⟨*cli*⟩+≡
```
blastn -task blastn -query dmAdhAdhdup.fasta \
       -subject dgAdhAdhdup.fasta
```

What are the best three results returned by this? How does this compare to the three best results returned by `sblast`?

4.33 By default, `blastn` prints the resulting alignments nucleotide by nucleotide. However, in many instances a table of alignment coordinates is all we need and easier to understand. Such a table also has the advantage that it can be parsed by downstream programs. The `-help` option of `blastn` gives a number of options for formatting the output. Can you generate output in tabular format?

4.34 Let's turn to the more typical Blast application of searching a short query in a long subject. Where does `blastn` locate the *melanogaster Adh* region in the *melanogaster* genome? How does that compare to the `sblast` result?

4.35 What happens when you use the *guanche* sequence as query?

4.36 How long does the `blastn` search of the *D. melanogaster* genome take compared to `sblast`?

4.37 `blastn` can work on a binary, compressed version of the subject, a Blast database. This is constructed with the program `makeblastdb`, which takes as mandatory arguments a file of input sequences (`-in`), the name of the database (`-out`), and the type of the database (`-dbtype`). To make individual sequences accessible by their accessions, we also opt to parse the sequence IDs (`-parse_seqids`).

⟨*cli*⟩+≡
```
makeblastdb -in subject.fasta -out dm -dbtype nucl \
            -parse_seqids
```

How long does database construction take?

4.38 What is the compression ratio of `makeblastdb`?

4.39 How long does it take to search the new database for the *melanogaster Adh* using the default megablast mode?

4.40 How long does the database search take with the more sensitive `blastn` task?

4.41 By default, `blastn` runs in a single thread. It can also use multiple parallel threads with the option `-num_threads`. What is the run time of the `blastn` task with eight threads?

4.42 We've seen that we can query a Blast database much more efficiently than the unprocessed subject, but is database construction reversible? In other words, can we get the subject sequences back? In that case we wouldn't need the subject sequences any more, which take up four times more disk space. The program `blastdbcmd` is designed to query Blast databases, its `-info` option gives basic information.

⟨*cli*⟩+≡
```
blastdbcmd -db dm -info
```

```
Database: subject.fasta
          8 sequences; 137,567,484 total bases
...
```

How long is the longest sequence in the `dm` database?

4.43 With `blastdbcmd` we can also retrieve the entries in the database. For example, we can retrieve *all* entries.

⟨*cli*⟩+≡
```
blastdbcmd -entry all -db dm | head
```

How many nucleotides does the output contain compared to the input?

4.44 So, database construction is reversible. But it can do more than allow quick searching on a smaller file. For example, we can list just the titles of the sequences and their lengths by setting the output format, `-outfmt`.

⟨*cli*⟩+≡
```
blastdbcmd -db dm -entry all -outfmt "%t %l"
```

How long is chromosome 2L? Can you retrieve it and measure its length?

4.45 The ability of Blast to find the query in the subject depends on the divergence between query and subject. To look at the relationship between Blast searches and divergence, we cut out a 5 kb region from chromosome 2L of *D. melanogaster* at coordinates that are easy to recognize, 10,000,001–10,005,000. Then we mutate the fragment with `mutator` and search for the mutated sequence.

⟨*cli*⟩+≡
```
cutSeq -r 10000001-10005000 dmChr2L.fasta > frag1.fasta
mutator -m 0.01 frag1.fasta > frag2.fasta
blastn -query frag2.fasta -db dm -outfmt 6
```

What happens for mutation rates of 0.03 and 0.3? Try a couple of runs.

4.46 Let's write a program, `megablast.sh`, to determine the number of successful runs out of 100 for a given mutation rate. First we generate the 100 randomized fragments. These serve as the query for `blastn`. A successful run might return more than one fragment, so to prevent over-counting, we set the maximum number of high-scoring pairs to 1. To speed things up, we use eight threads.

Prog. 4.1 (`megablast.sh`)

⟨*megablast.sh*⟩≡

```
printf "" > frag2.fasta
for a in $(seq 100)
do
    mutator -m $1 frag1.fasta >> frag2.fasta
done
blastn -max_hsps 1 -num_threads 8 -query frag2.fasta \
         -db dm -outfmt 6 |
    wc -l
```

How many successful runs do you get for a mutation rate of 0.3?

4.47 Let's write `blastn.sh`, which does the same thing as `megablast.sh`, except that it uses `blastn` as *task*. It also sets the E-value to 10^{-20}, which you can think of as the P-value for an alignment.

Prog. 4.2 (`blastn.sh`)

⟨*blastn.sh*⟩≡

```
printf "" > frag2.fasta
for a in $(seq 100)
do
    mutator -m $1 frag1.fasta >> frag2.fasta
done
blastn -max_hsps 1 -num_threads 8 -task blastn \
         -query frag2.fasta -db dm \
         -evalue 1e-20 -outfmt 6 |
    wc -l
```

How many successful `blastn` searches do you now get with a mutation rate of 0.3?

4.48 Fig. 4.8 shows the number of successful Blast runs as a function of the mutation rate for `megablast` and `blastn`. The fraction of successful runs is also called the *sensitivity*. `blastn` is more sensitive than `megablast` (we also saw that it is slower). Can you reproduce Fig. 4.8? The required script, call it `sens.sh`, might run for a while.

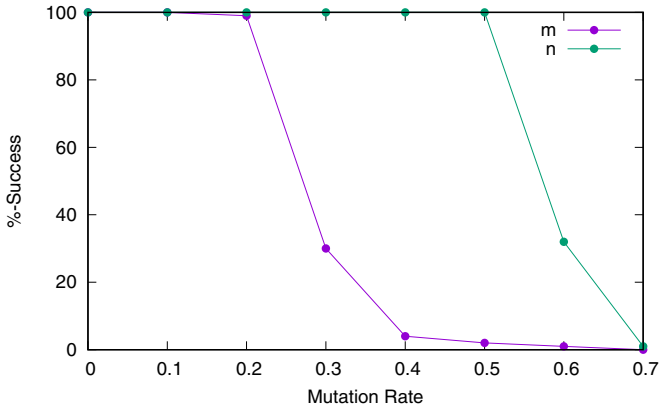


Fig. 4.8 Percent successful Blast runs as a function of the mutation rate for `megablast` (*m*) or `blastn` (*n*)

Blast Statistics

New Terms

E-value *P*-value

4.49 In our script `blastn.sh` we already used the *E*-value and we said at the time that you can think of it as an alignment's *P*-value. To be more precise, the *E*-value is the number of alignments with a score as good as that observed or better *expected* by chance alone. It is also called the "expectation value". To give an example, let's cut the interval 3101–3200 from the *guanche Adh*, align it with task `blastn`, and save the result in the file `blast.out`

⟨*cli*⟩+≡

```
cutSeq -r 3101-3200 dgAdhAdhdup.fasta > dgFrag.fasta
blastn -max_hsps 1 -task blastn -query dgFrag.fasta \
        -db dm > blast.out
```

Fig. 4.9 shows an abridged version of `blast.out`. The first line gives the score, or rather two scores, the bit score of 41.9 and the raw score of 45. The raw score corresponds to the scores seen with `al` and `sblast`. The underlying scoring scheme is shown at the bottom of the Blast output. Can you recapitulate the raw score?

4.50 The raw score depends on the scoring scheme. To have a score that is independent of the scoring scheme, the authors of Blast devised the bit score, S'. It is a function of the raw score and two parameters extracted from the scoring scheme, λ and K [30, p. 100f],

$$S' = \frac{\lambda S - \log(K)}{\log(2)}.$$

```
...
>NT_033777.3 Drosophila melanogaster chromosome 3R
Length=32079331

 Score = 41.9 bits (45),  Expect = 0.005
 Identities = 27/30 (90%), Gaps = 0/30 (0%)
 Strand=Plus/Minus

Query  8         TAATGCCAGTGGCAGTGGCAGGGGCACTGG  37
                 ||||||||||||| |||||| |||| |||
Sbjct  9098607   TAATGCCAGTGGCAATGGCAGTGGCATTGG  9098578
...
Gapped
Lambda      K        H
   0.625    0.410    0.780
...
Matrix: blastn matrix 2 -3
Gap Penalties: Existence: 5, Extension: 2
```

Fig. 4.9 Abridged output from a `blastn` run

Can you recapitulate our bit score of $S' = 41.9$?

4.51 The E-value is a function of the bit score, the length of the query, m, and the length of the subject, n,

$$E = mn2^{-S'}.$$

Can you recapitulate our E-value of 0.005?

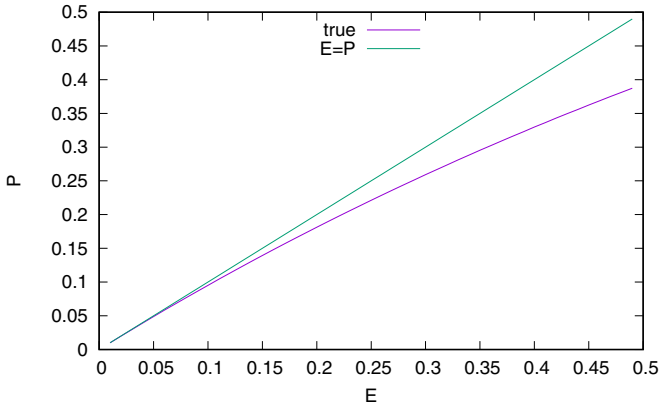


Fig. 4.10 The P-value of statistics as a function of the E-value returned by Blast

4.52 In statistics we are used to P-values rather than E-values. P-values are related to E-values through the function

$$P = 1 - e^{-E}.$$

Fig. 4.10 shows the P-value as a function of the E-value. Notice that for small E-values the two are indistinguishable. So in practice, we can think of small E-values as P-values. But we shouldn't forget that P-values cannot become larger than 1, while E-values can. Can you write a program `eval.awk` to reproduce Fig. 4.10?

4.53 Let's recapitulate the P-value through simulation. We write the program `simStats.sh` on the pattern of `megablast.sh` starting with an empty file that holds the randomized sequences we need. Then we run `blastn` and print the results with a score at least as large as our bit score of 41.9.

Prog. 4.3 (`simStats.sh`)

⟨*simStats.sh*⟩≡

```
printf "" > ran.fasta
for a in $(seq $1)
do
    randomizeSeq dgFrag.fasta >> ran.fasta
done
blastn -num_threads 8 -task blastn -query ran.fasta \
            -db dm -outfmt 6 |
    awk '$12>=41.9'
```

What is the simulated P-value of our alignment? How does that compare to the theoretical P-value?

Discover Protein Families

New Terms		
`blastp`	`neato`	reciprocal Blast hits
`circo`	`ranDot`	unidirectional Blast hits

4.54 We've used `sblast` to compare the *melanogaster Adh* as query to the *guanche Adh* as subject. What happens when we switch query and subject?

4.55 Fig. 4.11 shows a graph of Blast hits among ten yeast proteins. Arrows point from query to subject. Most arrows have two heads, which means they are reciprocal hits where homology is found regardless of the query/subject labeling. Can you spot the one unidirectional hit?

4.56 Fig. 4.11 was drawn using the program `neato`. It takes as input a graph file in dot notation. We can think of the dot notation as a programming language for graphs that is interpreted by `neato`. So here is our first dot program, `g1.dot`.

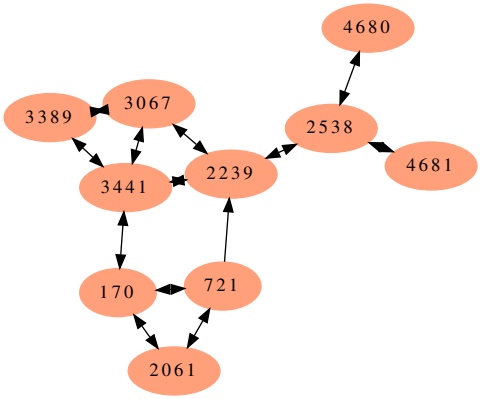


Fig. 4.11 Proteins (nodes) and their Blast hits (edges), which are either reciprocal Blast hits of the form query ↔ subject, or unidirectional Blast hits between query → subject

Prog. 4.4 (`g1.dot`)

⟨*g1.dot*⟩≡
```
graph G {
    a -- b
    b -- c
    c -- a
}
```

We visualize this.

⟨*cli*⟩+≡
```
neato -T x11 g1.dot
```

In case `x11` isn't recognized on your system, try one of the output formats listed by `neato`, for example portable network graphics, png, which you can save to file and then open with your system's viewer.

⟨*cli*⟩+≡
```
neato -T png g1.dot > g1.png
```

Can you write `g2.dot` to specify the graph in Fig. 4.12?

4.57 We draw Fig. 4.11 with prgram `yeast.dot`, where we first declare the nodes as filled in with the color lightsalmon. Then we declare the edges as having arrows in both directions. This is followed by the actual edges; if we write more than one edge per line, we separate them by semicolons. At the end we specify the single edge with just a forward arrow. So a local edge annotation overrides the global declaration of double heads.

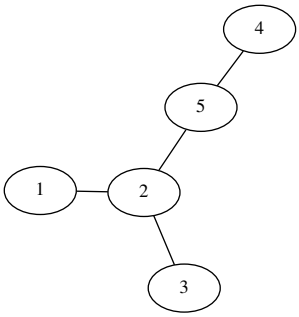


Fig. 4.12 Example graph

Prog. 4.5 (`yeast.dot`)

⟨*yeast.dot*⟩≡

```
graph G {
    node [style=filled, color=lightsalmon]
    edge [dir=both]
    170 -- 721; 170 -- 2061; 170 -- 3441
    721 -- 2061; 2239 -- 2538; 2239 -- 3067
    2239 -- 3441; 2538 -- 4680; 2538 -- 4681
    3067 -- 3389; 3067 -- 3441; 3389 -- 3441
    721 -- 2239 [dir=forward]
}
```

What happens when we layout `yeast.dot` with the program `circo` instead of `neato`?

4.58 The program `ranDot` draws random graphs in dot notation. Can you draw a graph with ten nodes where all nodes are connected to all other nodes?

4.59 How many edges could in theory be drawn between the nodes (proteins) in Fig. 4.11? What proportion of these is actually found?

4.60 If a is homologous to b and b is homologous to c, then we infer that a is also homologous to c, even if the direct Blast hit doesn't exist. Are the proteins in Fig. 4.11 all homologous to each other?

4.61 Sets of homologous proteins are called *protein families* and their underlying genes form *gene families*. Fig. 4.11 shows a yeast gene family, and we've already seen the pairs of *Adh* genes in *Drosophila*. But what about *Mycoplasma genitalium*, which has one of the smallest genomes of any organism? Does it also contain gene families? We investigate this question on the level of proteins. The proteome of *M. genitalium* is contained in `mgProteome.fasta`. How many proteins does it consist of?

4.62 Protein families are discovered through all-against-all Blast searches. How many pairwise comparisons can be carried out between the proteins of *M. genitalium*?

4.63 The headers in `mgProteome.fasta` consist of the prefix `lcl|MG_` followed by a number and a brief description of the protein function.

⟨*cli*⟩+≡
```
grep '^>' mgProteome.fasta | head -n 3
```

```
>lcl|MG_002 DnaJ domain protein
>lcl|MG_003 DNA gyrase, B subunit
>lcl|MG_004 DNA gyrase, A subunit
```

The prefix is redundant. We delete it and redirect the result to `mgProteome2.fasta`.

⟨*cli*⟩+≡
```
awk -F '_' '/^>/{print ">" $2}!/^>/{print}' \
    mgProteome.fasta > mgProteome2.fasta
```

Can you explain this Awk code?

4.64 We use protein Blast implemented in the program `blastp` to find all pairwise hits among the *M. genitalium* proteome. Our E-value is 10^{-5} and we restrict the output to no more than one hit per query.

⟨*cli*⟩+≡
```
blastp -query mgProteome2.fasta -subject mgProteome2.fasta \
         -evalue 1e-5 -max_hsps 1 -outfmt 6 > mgp.bl
```

How many Blast hits are there?

4.65 The number of times a protein appears among the queries is the number of homologs it has in the proteome. Which proteins have the largest number of homologs?

4.66 What are the functions of the proteins with most homologs?

4.67 How many proteins have at least one homolog?

4.68 Fig. 4.13 shows the protein families in the proteome of *M. genitalium*. It is drawn using `blast2dot` and `circo`. Can you reproduce it?

4.69 The central circle in Fig. 4.13 contains the ABC transporters 410, 180, and 179. How many entries does the circle contain in total? Can you write the program `entr.sh` to list their functions?

4.70 To learn more about ABC transporters, we look them up in the Prosite database. This consists of two files, `prosite.dat` and `prosite.doc`. `prosite.dat` contains protein motifs, `prosite.doc` documentation on the protein families characterized by the motifs. By searching the documentation for "ABC transporter" we get to the right entry. Can you find it? What is the major function of ABC transporters?

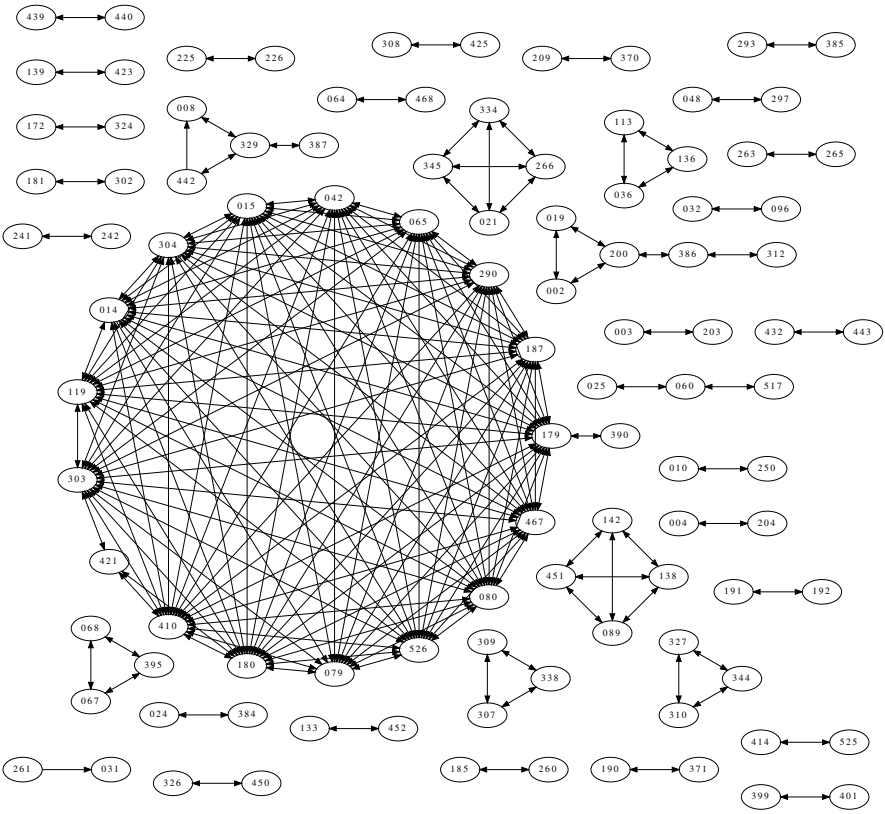


Fig. 4.13 Protein families in the proteome of *M. genitalium*

4.71 By default, `blast2dot` excludes singletons, but it can include them with `-s`. Fig. 4.14 shows the protein families as colorful islands in a sea of gray singletons. Can you reproduce it?

4.3 Glocal

We saw that in Blast, we chop up the query into overlapping words and use them to locate promising regions in the subject (Fig. 4.7). However, when we align, say, the *Adh* region from *D. melanogaster* with the genome of the same species, we *know* that the query can be aligned globally. Similarly, a sequencing read obtained from a given organism can be aligned globally to its genome sequence. We call such global/local alignment *glocal alignment*. We look at two cases, the global alignment

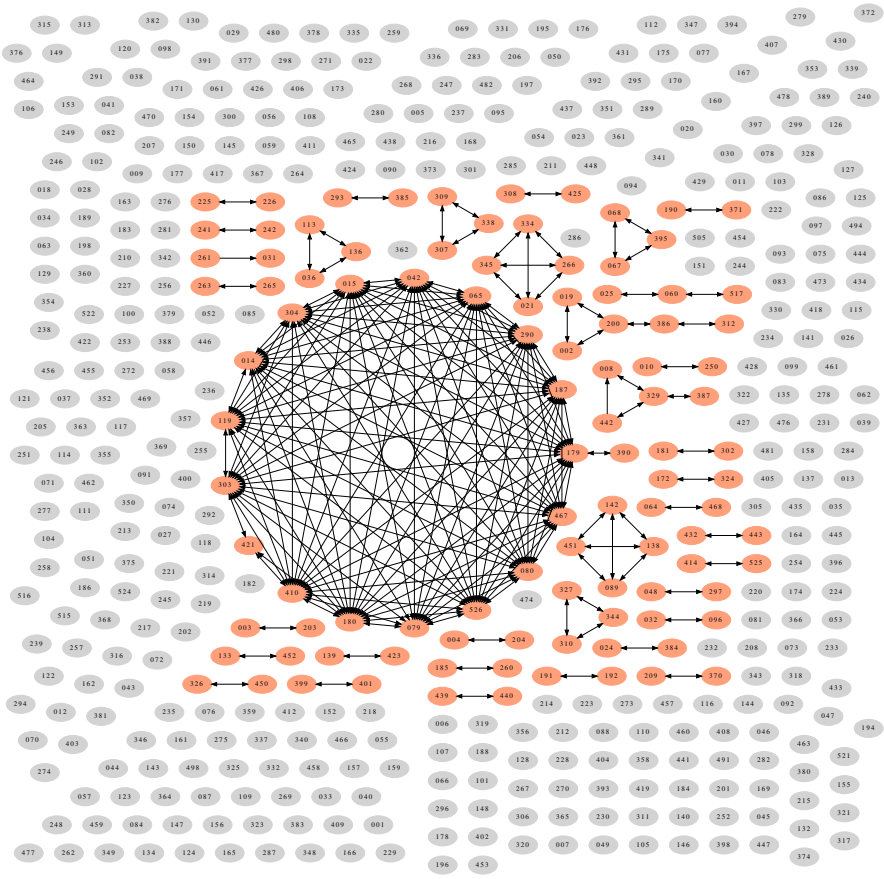
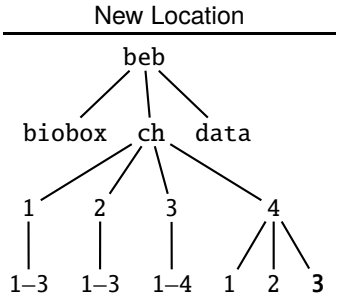


Fig. 4.14 The protein families of *M. genitalium* as colorful islands in a sea of gray singletons

of a gene-sized query to a longer subject, and the global alignment of sequencing reads to their genome of origin.



4.72 We start by creating a working directory for this section. Can you do that?

k-Error Alignment

New Term
kerror

4.73 The simplest alignment is exact matching, so let's begin with that, using a 100 bp fragment from the *melanogaster Adh* region, which we first copy to our working directory.

⟨*cli*⟩+≡
```
cp $BEB/data/dmAdhAdhdup.fasta .
cutSeq -r 2301-2400 dmAdhAdhdup.fasta > dmFrag.fasta
```

We should be able to locate the fragment in the parent sequence with an exact matcher like `keyMat`, but also with an inexact matcher like `al`. Can you do both?

4.74 In biology, sequences are always subject to mutation. So we mutate a single random position in our fragment.

⟨*cli*⟩+≡
```
mutator -n 1 dmFrag.fasta > dmFrag2.fasta
```

Can you convince yourself that inexact matching still works, while exact matching fails?

4.75 We've already seen that `al` is too slow for large subjects. So we use the same trick as for Blast and chop up our fragment into "words". In Blast the words are used to pick a matching region from both the query and the subject (Fig. 4.7). In glocal alignment, there is no need to pick a region from the query, only from the subject, so we rethink the partition strategy. Two aspects matter, the number of words and their lengths. A small set of words is easier to handle than a large set of words, and long words create fewer spurious matches than short words. In other words, the ideal set of words has a few long entries. Can you think of such a set of words for our mutated fragment?

4.76 Our current example contains a single mismatch, but more generally, we can think of k errors, where an error is either a mismatch or a gap. In k-error alignment, we first set k as the maximum number of errors allowed. Then we divide the query into $k+1$ contiguous fragments of equal length, knowing that one of them will form an exact match to the relevant subject region (Fig. 4.15A). The program `kerror` can print the fragment list. What does that look like for `dmFrag2.fasta` compared to the word list of `sblast`?

4.77 Having constructed the fragments, `kerror` searches for them in the subject (Fig. 4.15B). For each match, it constructs an alignment matrix and calculates the alignment in the traditional way (Fig. 4.15C). How long does `kerror` take to locate our 100 bp fragment in the genome of *D. melanogaster*?

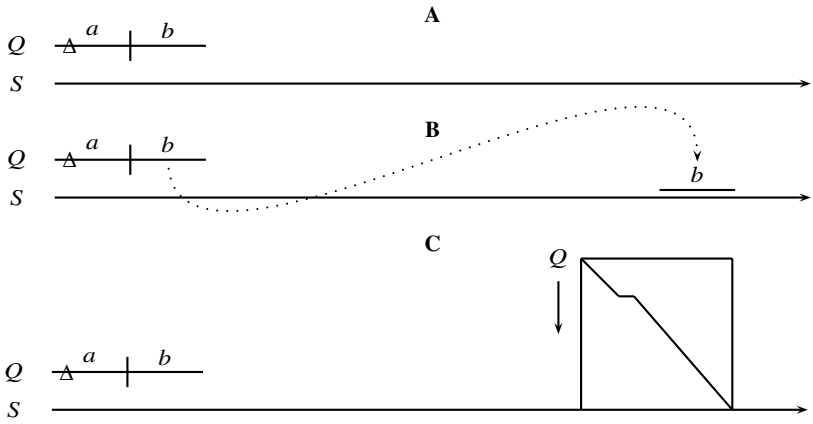


Fig. 4.15 The k-error alignment method comprises three steps: Division of the query into $k + 1$ contiguous fragments (**A**), exact search for the fragments (**B**), and checking whether or not a k-error alignment has been found by filling in the alignment matrix anchored by the exact match (**C**)

4.78 Our implementation of `kerror` searches only the forward strand of the subject, while `sblast` does the more sensible thing searching the forward and reverse strands. Bearing this in mind, which of the two programs is faster?

4.79 It is surprising that in our example `kerror` is faster than `sblast`, because aligning by matrix traversal, as done by `kerror`, is much more time-consuming than the Blast strategy of aligning by extension. So let's investigate this further. How many matches do the two fragments used by `kerror` produce on the forward strand of the genome of *D. melanogaster*?

4.80 How many matches do the 90 words used by `sblast` generate on the forward strand of the *melanogaster* genome?

4.81 `kerror` returns gapped alignments, while `sblast` is restricted to ungapped alignments. To see the difference, we construct a version of our fragment that contains a single nucleotide deletion.

⟨*cli*⟩+≡
```
cutSeq -r 1-49,51-100 -j dmFrag.fasta > dmFrag3.fasta
```
Can you check whether your fragment contains the intended gap?

4.82 How does `kerror` handle the fragment with the deletion (`dmFrag3.fasta`) compared to `sblast`?

4.83 What about a single nucleotide insertion? We begin again by constructing a fragment.

⟨*cli*⟩+≡
```
cutSeq -r 1-50,50-100 -j dmFrag.fasta > dmFrag4.fasta
```

Can you convince yourself that the query now contains a single nucleotide insertion?

4.84 How does `kerror` handle the fragment with the insertion (`dmFrag4.fasta`) compared to `sblast`?

4.85 We now graduate from carefully constructed fragments to real sequences. Can you find the full *melanogaster Adh* region in the genome? You might like to write a script `kerror.sh` with an escalating number of possible errors like $k = 1, 2, 4, ...$

4.86 `sblast` also found parts of the *guanche Adh* region in the *melanogaster* genome. We can establish the minimum number of errors necessary for replicating this with `kerror` using `al`.

⟨*cli*⟩+≡
```
cp $BEB/data/dgAdhAdhdup.fasta .
al dmAdhAdhdup.fasta dgAdhAdhdup.fasta | head
```

Do you think a search for the *guanche Adh* in the *melanogaster* genome with `kerror` is feasible?

Read Mapping

New Terms		
BAM file	Phred score	`samtools`
coverage	SAM file	

4.87 Mapping sequencing reads is by far the most common application of glocal alignment. Mapping is global for the read and local for the template. The program `sequencer` simulates a DNA sequencing machine. The number of reads it generates is specified via the *coverage*, which is the number of nucleotides sequenced divided by the template length. To start small, we sequence the *Adh* region of *D. melanogaster*.

⟨*cli*⟩+≡
```
sequencer dmAdhAdhdup.fasta > reads.fasta
```

What is the coverage you get?

4.88 We map the reads with `blastn`, ask for tabular output, and redirect it to the file `adh.bl`.

⟨*cli*⟩+≡
```
blastn -query reads.fasta -subject dmAdhAdhdup.fasta \
         -outfmt 6 > adh.bl
```

Now we can think about an individual position and ask, how many reads are stacked on top of it. Fig. 4.16 shows the coverage per position along the *melanogaster Adh* from our sequencing simulation with overall coverage 1 and 15. The figure was generated with the program `cov.awk`, which we write now. At the beginning we initialize its variables. In the central block we look at the lines of Blast results and

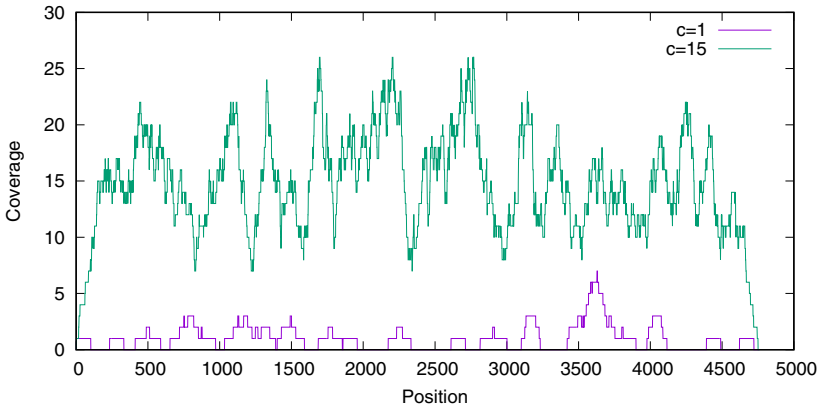


Fig. 4.16 Coverage per position along the *melanogaster Adh* region for total coverage of 1 and 15

store the subject start and end in columns 9 and 10. A match on the reverse strand is marked by a start greater than the end, in which case we switch the coordinates. At the end we print the coverage at each position.

Prog. 4.6 (`cov.awk`)

⟨*cov.awk*⟩≡
```
⟨Initialize variables, Prog. 4.6⟩
{
  s = $9
  e = $10
  if (s > e) {
    ⟨Switch start and end, Prog. 4.6⟩
  }
  for (i = s; i <= e; i++)
    cov[i]++
}
⟨Print coverage per position, Prog. 4.6⟩
```

There are two variables to initialize in the `BEGIN` block, the template length, n, and the array of coverages, `cov`.

⟨*Initialize variables, Prog. 4.6*⟩≡
```
BEGIN {
  ⟨Get template length, n, Prog. 4.6⟩
  ⟨Initialize array of coverages, cov, Prog. 4.6⟩
}
```

The template length is provided by the user. Can you make sure (s)he has actually done so?

4.89 At the beginning, the coverage of every position is zero. Can you initialize the array of coverages?

4.90 To switch the start and end position, one of them is best saved to a temporary variable first. Can you implement the switch?

4.91 Can you print the coverage per position?

4.92 Can you now generate your version of Fig. 4.16?

4.93 Fig. 4.16 shows that with overall coverage 1 many sites remain unsequenced, while—at least in our simulation—some sites have a coverage greater than 5. Can you count the sites with coverage 0, 1, and so on, for overall coverage 1?

4.94 Fig. 4.17 shows the site coverages for our sequencing simulation for overall coverages 1 and 15. The maximum for $c = 15$ is at 15, the maximum for $c = 1$ is, in fact, zero. What do the distributions from your simulation look like?

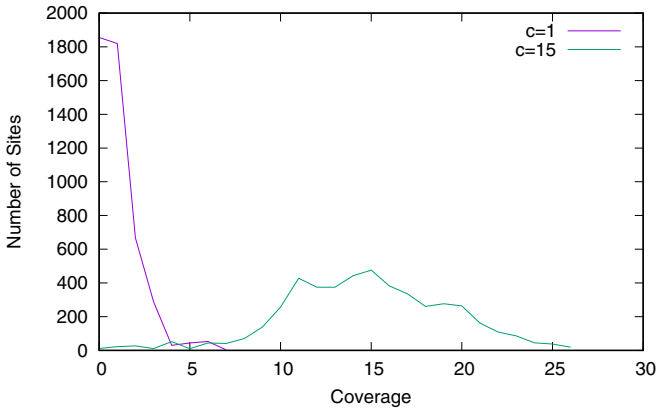


Fig. 4.17 The distribution of site coverages in our sequencing simulation with overall coverage 1 and 15

4.95 How many sites remained unsequenced with coverage 15?

4.96 Just to see how far we can take this, let's insist we sequence every last nucleotide. What is the coverage we need for that? To explore this systematically, we write the script `simCov.sh`, which iterates across coverages 1 to 100 and reports "all sequenced" for every winning coverage.

Prog. 4.7 (simCov.sh)

⟨*simCov.sh*⟩≡

```
for a in $(seq 100)
do
    printf "%s " $a
    sequencer -c $a dmAdhAdhdup.fasta > reads.fasta
    blastn -query reads.fasta -subject dmAdhAdhdup.fasta \
            -outfmt 6 |
        awk -f cov.awk -v n=4761 |
        awk '{c[$2]++}END{for(a in c)print a, c[a]}' |
        sort -n |
        head -n 1 |
        awk '/^0/{print}!/^0/{print "all sequenced"}'
done
```

For which coverages is every nucleotide sequenced?

4.97 Read mapping is usually done with a dedicated mapping program, for example bwa. We've already mentioned bwa in the context of the Burrows-Wheeler transform, but haven't used it yet. It requires an index of the template to run.

⟨*cli*⟩+≡

```
bwa index -p dmAdh dmAdhAdhdup.fasta
```

Do you recognize any of the acronyms printed during index construction?

```
@SRR006041.1 FGR9I4U01AIGBG length=54
CTCGAGAATTCTGGATCCTCCATACATACTGCAACAATTTGTAACTTTACTTCC
+SRR006041.1 FGR9I4U01AIGBG length=54
;;;;;;;;;;;;;;;;9;;;;;;8::9:9:99:::;9;;;;9;:9888887788
```

Fig. 4.18 Example read in FASTQ format taken from data published by The International Genome Sample Resource [13]

4.98 Sequencing machines return for each base not just its identity, but also its quality. This information is stored in FASTQ format, a variant on the FASTA format. Fig. 4.18 shows an example, which is one of the myriad reads published by The International Genome Sample Resource, a consortium focusing on human genetic diversity [13]. Like every read, it consists of four lines, a header starting with @, the sequence, a second header starting with +, and a quality score for each base in the sequence in line 2. The quality score, Q, is written as the numerical value of the character. For example, the first base, C, has a semicolon as quality score. What is the value of Q denoted by semicolon?

4.99 The ASCII values in FASTQ files are shifted by 33 to give the final quality score, $Q' = 59 - 33 = 26$. This is also called a "Phred score" because it was

first introduced in the Phred software package for analyzing sequencing reads. The corresponding error probability is

$$P_e = 10^{-Q/10},$$

which in our case is ≈ 0.0025.

⟨*cli*⟩+≡
```
echo 'e(-2.6 * l(10))' | bc -l
```

What is the error probability corresponding to the remaining distinct scores, `:`, `9`, `8`, and `7`?

4.100 We generate a fresh batch of reads with coverage 15 and store them in `reads.fasta`.

⟨*cli*⟩+≡
```
sequencer -c 15 dmAdhAdhdup.fasta > reads.fasta
```

Now we'd like to convert these reads from FASTA to FASTQ before we map them with `bwa`. This is a bit of an *ad hoc* conversion, as no real differences in quality are involved, we'd just like to play with the FASTQ format. So we write an *ad hoc* program to generate it for us. We've already seen that files with multiple FASTA entries can be tricky to deal with, so we simplify them first with `fasta2tab`. What is its output when applied to `reads.fasta`?

4.101 We now need a program, `tab2fastq.awk`, to convert `fasta2tab` output to FASTQ. Can you write it? You can use the same, arbitrary quality value for each nucleotide, for example, semicolon.

4.102 Can you convert `reads.fasta` to `reads.fastq`?

4.103 Now we can map our reads.

⟨*cli*⟩+≡
```
bwa mem dmAdh reads.fastq > adh.sam
```

`adh.sam` is a SAM file, which consists of a header and a body.[1] Header lines start with `@`. So we can filter for the header lines.

⟨*cli*⟩+≡
```
grep '^@' adh.sam
```

```
@SQ     SN:DMADH        LN:4761
@PG     ID:bwa  PN:bwa  VN:0.7.15-r1142-dirty \
   CL:bwa mem dmAdh reads.fasta
```

Our header consists of two lines, a line for the reference sequence (`@SQ`), and a line for the program used (`@PG`). The sequence line contains the sequence name, DMADH, and its length, 4761. The program line contains the program identifier (`ID`), its name (`PN`), the version (`VN`), and the command line (`CL`). What does your header look like?

[1] The SAM format is specified at `http://samtools.github.io/hts-specs/SAMv1.pdf`

Table 4.1 The mandatory fields of a SAM file

Col.	Meaning
1	Query
2	Comment flag; 0: none, 4: unmapped, 16: reverse-complement
3	Subject
4	Position
5	Mapping quality
6	Match string; M: match, D: deletion, I: insertion, S: soft clipping from read
7	Name of read mate
8	Position of read mate
9	Template length
10	Read sequence
11	Base quality

4.104 A SAM file consists of eleven mandatory columns, which might be followed by additional columns. The mandatory columns are listed in Table 4.1. Column 2 indicates whether a read is mapped to the reverse strand. How are reverse reads stored in `adh.sam`?

4.105 SAM files are often kept in their more compact binary representation, BAM, which we generate with the `view` command of `samtools`.

⟨*cli*⟩+≡
```
samtools view -b adh.sam > adh.bam
```

What is the compression ratio between our SAM and BAM files?

4.106 We can convert a BAM file back into SAM.

⟨*cli*⟩+≡
```
samtools view adh.bam | head
```

Is the compression lossless?

4.107 BAM files are often kept in sorted format.

⟨*cli*⟩+≡
```
samtools sort adh.bam > adhS.bam
```

In what way is `adhS.bam` sorted?

4.108 Sorting again reduces the file size—by how much?

4.109 A sorted BAM file can be indexed.

⟨*cli*⟩+≡
```
samtools index adhS.bam
```

What is the new index file called?

4.110 An indexed BAM file can be visualized. For this, we apply "text view", or `tview`, to the BAM file, which displays the reads with respect to the reference sequence, the *melanogaster Adh* region.

⟨*cli*⟩+≡
```
samtools tview --reference dmAdhAdhdup.fasta adhS.bam
```

At the top we see a row of coordinates, followed by the reference sequence and the underlined consensus sequence. The remaining lines are forward reads indicated by dots and reverse reads indicated by commas. To quit the viewer, press `q`. Press `?` for help, `Esc` to cancel (for example the help screen). Can you color the reads according to base quality?

4.111 Positions in an alignment are denoted by the sequence name and the position separated by colon,

```
name:position
```

Can you jump to position 2000?

4.112 Let's scale up our simulation and sequence chromosome 2L of *D. melanogaster* with a coverage of 15. How many nucleotides are now contained in the reads?

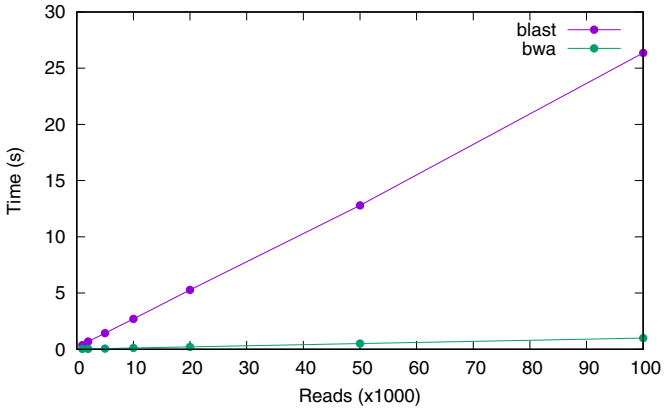


Fig. 4.19 Run times of `blastn` and `bwa` as a function of the number of 100 bp reads mapped onto chromosome 2L of *D. melanogaster*

4.113 Fig. 4.19 shows the run times of `blastn` and `bwa` as a function of the number of reads mapped. Let's begin with `blastn`. It requires a Blast database to run efficiently. Can you construct it for chromosome 2L?

4.114 We write a script, `timeBlast.sh`, to measure the run times of `blastn` as a function of the number of reads analyzed. The number of lines we retrieve with `head` is twice the number of reads.

Prog. 4.8 (`timeBlast.sh`)

⟨*timeBlast.sh*⟩≡
```
for a in 1 2 5 10 20 50 100
do
    ((nl=$a*1000*2))
    head -n $nl reads2L.fasta > query.fasta
    ⟨Time Blast run, Prog. 4.8⟩
    echo $a $rt
done
```

Can you time the Blast run (`date`)?

4.115 It's time to run `timeBlast.sh` and save the results for later use. Can you do that?

4.116 What happens to the Blast run times if we set the number of threads to 8?

4.117 How long would Blast take to map all the reads?

4.118 `bwa` requires an index to run. Can you construct it?

4.119 Now we write the script `timeBwa.sh`. In a FASTQ file the number of lines is four times the number of reads.

Prog. 4.9 (`timeBwa.sh`)

⟨*timeBwa.sh*⟩≡
```
for a in 1 2 5 10 20 50 100
do
    ((nl=$a*1000*4))
    head -n $nl reads2L.fastq > reads.fastq
    ⟨Time bwa run, Prog. 4.9⟩
    echo $a $rt
done
```

`bwa` runs much faster with multiple threads than without, so we timed it with 8 threads using `-t 8`. In addition, we minimized the verbosity with `-v 1`. Can you also time `bwa` like that?

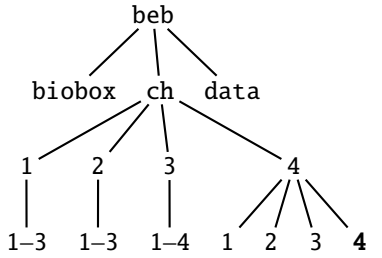
4.120 How long would `bwa` take to map all reads?

4.121 Can you reproduce Fig. 4.19?

4.4 Assembly

Over 40 years after its invention, shotgun sequencing remains the standard method for sequencing DNA [38]. Shotgun sequencing consists of two steps, sequencing of random reads in the lab and assembly of these reads in the computer. Assembly is essentially a puzzle with potentially millions of pieces and the way we piece together the pieces is through fast alignment.

New Location



4.122 Can you construct the directory for this section and change into it?

Overlapping and Merging Reads

New Terms

`olga`	overlap graph

4.123 Assembly is done by overlapping pairs of reads. Fig. 4.20A illustrates such an overlapping pair of reads, where a suffix of read a matches a prefix of read b. To make this concrete, we consider the pair a = `GATAC` and b = `TACAG`. What is their overlap?

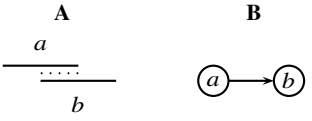


Fig. 4.20 Cartoon of overlap between reads a and b; **A** shows the overlap alignment, **B** the overlap graph

4.124 The program `al` implements alignment in overlap mode, where flanking gaps are ignored in the score. Can you use `al` to overlap a and b?

4.125 Just to remind ourselves—what does the global alignment of a and b look like? And the local alignment?

4.126 In assembly, overlapping reads are merged into contigs. What is the contig resulting from merging a and b?

4.127 The program `sass` is a simple assembler. Can you use it to assemble a and b?

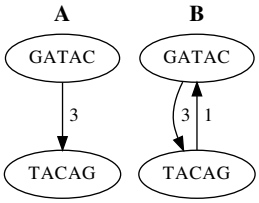


Fig. 4.21 Overlap of minimum length $k = 3$ between oligos `GATAC` and `TACAG` (**A**); including the additional overlap without threshold (**B**)

4.128 An overlap between two reads can also be visualized as a graph, where the nodes are the sequences, and the edges point from matching suffix to matching prefix (Fig. 4.20B). Can you draw the overlap graph for the reads in Fig. 4.22?

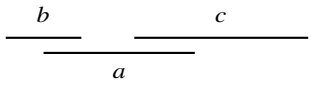


Fig. 4.22 Three overlapping reads

4.129 Nodes in overlap graphs are often labeled with the actual read sequence rather than the read name. In that notation the edges are usually labeled with the overlap length. Fig. 4.21A shows the optimal overlap we have already identified. However, there is another overlap in the opposite direction if we also include overlaps of length 1 (Fig. 4.21B). Can you draw the complete overlap graph for the two reads r_1 = `CCTTG` and r_2 = `CTTGA`?

4.130 The program `olga` calculates the overlap graph between reads. The resulting graph is in dot notation, which we can plot with `dot`. This uses the same rendering system we already saw in `neato` and `circo`; so again, if `-T x11` doesn't work on your system, try `-T png`. Can you use `olga` to draw the overlap graph for r_1 and r_2?

4.131 The file `pentamers.fasta` contains oligos of length 5. How many pentanucleotides does it contain? Are r_1 and r_2 among them?

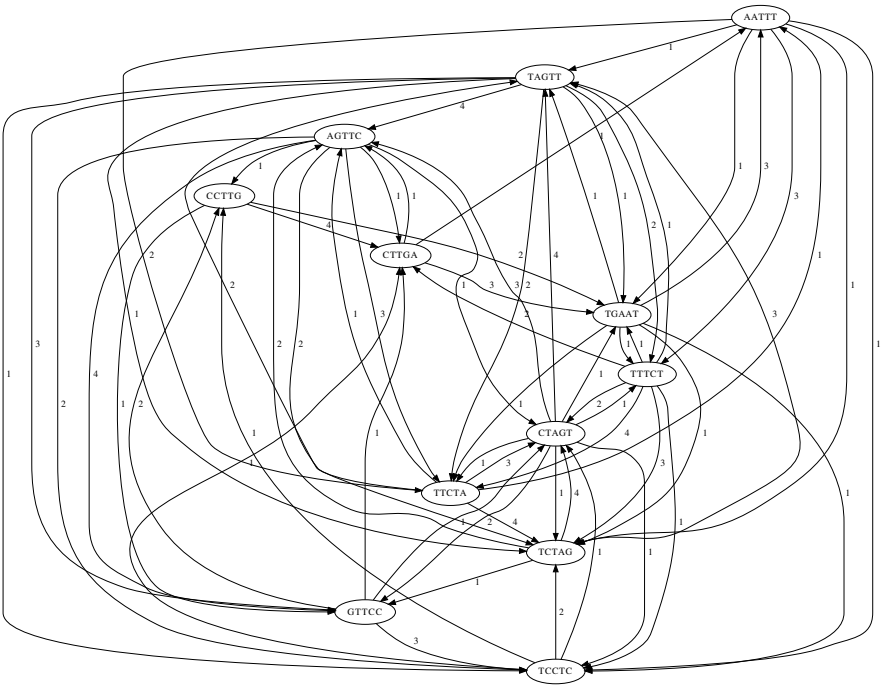


Fig. 4.23 Complete overlap graph between twelve nucleotides of length five

4.132 Fig. 4.23 shows the complete overlap graph of the oligos in `pentamers.fasta`. Can you reproduce it and spot r_1 and r_2?

4.133 Fig. 4.24 shows the overlap graph for our pentamers restricted to overlaps of 3 or more. Can you reproduce it?

4.134 The overlap graph Fig. 4.24 gives an idea of how to assemble sequencing reads: Start at the node that has no incoming edge as this must be the leftmost read, and work your way to the last edge. If there is more than one outgoing edge, choose the "heavier" one. If there are several outgoing edges with the same maximum weight, pick an arbitrary one among them. Can you reconstruct the sequence from which the twelve reads in Fig. 4.24 were sampled? You can check your answer with `sass`.

4.135 We can save our assembly into a template sequence.

⟨*cli*⟩+≡

```
sass -t 3 pentamers.fasta > template.fasta
```

Now we try to sequence the template using `sequencer` in *shredder* mode (`-S`), which means it only samples the forward strand. We also abolish sequencing errors

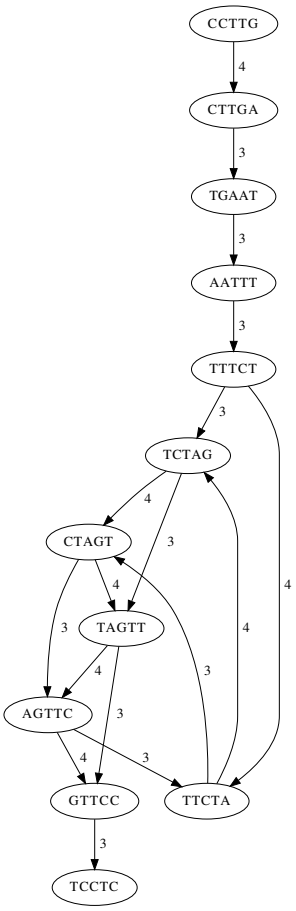


Fig. 4.24 Overlap graph for the same dozen oligos shown in Fig. 4.23, this time with minimum overlap 3

(`-e 0`) and set the seed for the random number generator (`-s 2`) to make sure you get the same reads as us in this instance.

⟨*cli*⟩+≡

```
sequencer -s 2 -r 5 -S -e 0 -c 4 template.fasta > reads.fasta
```

What happens when you assemble these reads with `sass` and minimum overlap of 3 or minimum overlap of 2?

4.136 Fig. 4.25A shows the overlap graph for our reads with a minimum overlap of 3. The two graphs for the two contigs are clearly visible. However, there are a number of redundant reads in the input, which make the graph messy. They have been removed in Fig. 4.25B. Can you write a script, `reduce.sh`, to remove redundant reads, and use it to reproduce Fig. 4.25B?

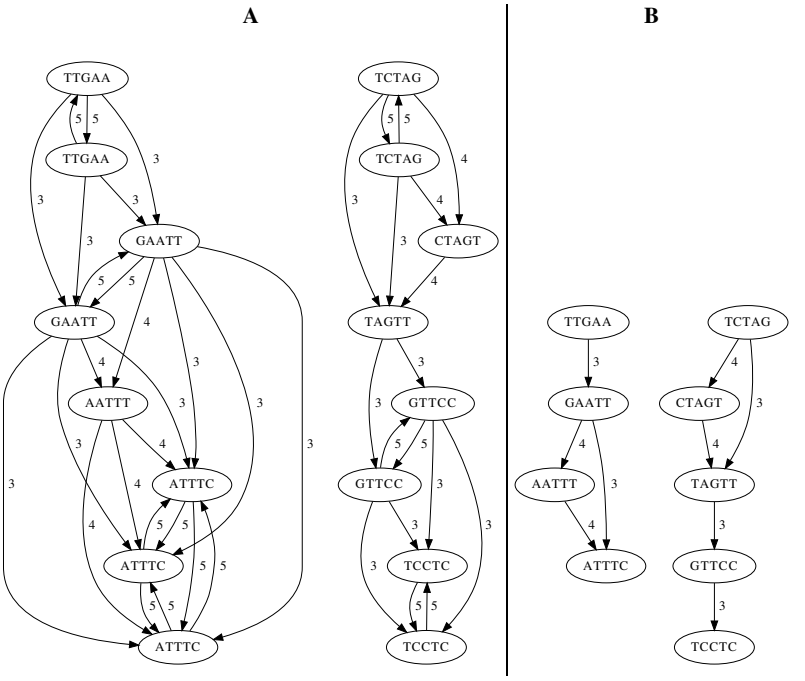


Fig. 4.25 An overlap graph with all reads (**A**), and reduced to the unique reads (**B**)

4.137 Can you draw the overlap graph for the single contig?

4.138 We've seen the problem that there might not be enough links in the graph to cover the length of the template. Here is a set of reads to illustrate a different type of problem.

⟨*cli*⟩+≡

```
sequencer -s 18 -r 5 -S -e 0 -c 4 template.fasta |
    bash reduce.sh > reads.fasta
```

What do you observe when you assemble these reads with minimum overlap 3?

Scaling Up

New Terms		
N_{50}	`velvetg`	`velveth`
hashing		

4.139 To assemble whole genomes, we need more sophisticated tools than our simple assembler, `sass`. The programs `velveth` and `velvetg` together form the

popular assembler *velvet* used in real-world genome projects [43]. To explore it, we'd like to start with a sequence without the complications of real sequences, but still of realistic size. So we copy the genome of *M. genitalium*, randomize it, and sequence the randomized version with coverage 1 and no errors.

⟨*cli*⟩+≡
```
cp $BEB/data/mgGenome.fasta .
randomizeSeq mgGenome.fasta > rg.fasta
sequencer -e 0 rg.fasta > reads.fasta
```

To assemble the reads, we first hash them. Hashing converts a read—more precisely a substring of a read—into a number. This number is used to look up the read in a read table. In velvet, hashing is carried out by `velveth`. It takes as main parameters an output directory, `assem`, and the length of the words, or k-mers, to be hashed, 21. There are various additional options, in our case to tell the program we're using short reads (`-short`) in FASTA format (`-fasta`).

⟨*cli*⟩+≡
```
velveth assem 21 -short -fasta reads.fasta
```

What do you observe?

4.140 `velveth` has now generated the directory `assem`. What does it contain?

4.141 What happens when you run `velveth` with a hash length of 32?

4.142 We revert to hash length 21 and assemble the reads with `velvetg`. This takes as argument the expected coverage (`-exp_cov`).

⟨*cli*⟩+≡
```
velveth assem/ 21 -short -fasta reads.fasta
velvetg assem/ -exp_cov 1
```

The resulting contigs are now contained in

`assem/contigs.fa`

How many contigs do you get?

4.143 What happens when you double the coverage to 2?

4.144 Let's write a program to summarize the steps of our shotgun simulation, `simShot.awk`. We begin by interacting with the user before we sequence the template, hash the reads, assemble the reads, and count the contigs.

Prog. 4.10 (`simShot.awk`)

⟨*simShot.awk*⟩≡
```
BEGIN {
    ⟨User interaction, Prog. 4.10⟩
    ⟨Sequence template, Prog. 4.10⟩
    ⟨Hash reads, Prog. 4.10⟩
    ⟨Assemble reads, Prog. 4.10⟩
    ⟨Count contigs, Prog. 4.10⟩
}
```

In the user interaction, we require a coverage and a template. If we don't get both, we print a usage message and exit.

⟨*User interaction, Prog. 4.10*⟩≡
```
if (!c || !t) {
  ⟨Print usage message, Prog. 4.10⟩
  exit 1
}
```

In the usage message we request the coverage and a template. The blank at the beginning of the string in the second line is significant.

⟨*Print usage message, Prog. 4.10*⟩≡
```
m = "Usage: awk -f simCov.awk"
m = m " -v c=<cov> -v t=<template>"
```

The usage message still lacks switches to collect the arguments for the three programs we will run, `sequencer` (`s`), `velveth` (`h`), and `velvetg` (`g`). Can you add these to the message and then print it?

4.145 To sequence the template, we first construct the command for `sequencer` from the coverage, its additional arguments, and the template. The reads are redirected to `reads.fasta`. Then we send the command to the system with the Awk function `system`. Execution of `sequencer` might take some time, so we give ourselves feedback about what's going on.

⟨*Sequence template, Prog. 4.10*⟩≡
```
cmd = "sequencer -c %s %s %s > reads.fasta"
cmd = sprintf(cmd, c, s, t)
printf("# running sequencer...")
system(cmd)
print "done"
```

Next, we hash the reads with `velveth`. It writes messages to the screen, which we don't need here. Can you run `velveth` "quietly"?

4.146 Can you now quietly assemble the reads?

4.147 The contigs are assembled, so we count them and print their count as a function of the coverage.

⟨*Count contigs, Prog. 4.10*⟩≡
```
printf("%s\t", c)
cmd = "grep -c '^>' assem/contigs.fa"
system(cmd)
```

How many contigs do you get with error-free sequencing of the randomized genome with coverage 5?

4.148 Fig. 4.26 shows the number of contigs as a function of the coverage for our randomized genome when sequenced without error. Can you construct your own version of Fig. 4.26?

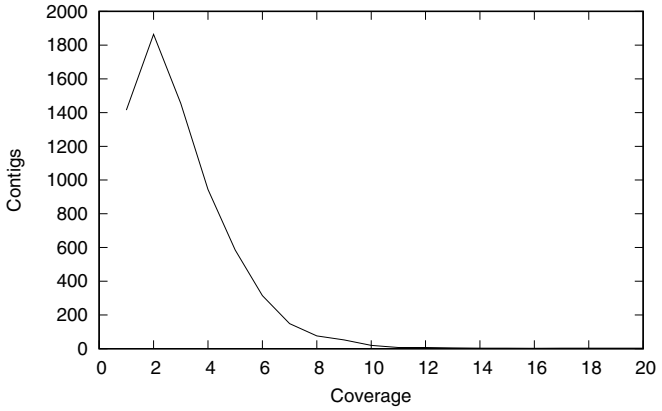


Fig. 4.26 The number of contigs as a function of the coverage in simulated shotgun runs on the shuffled genome of *M. genitalium*

4.149 What is the smallest coverage resulting in a single contig in your simulations?

4.150 Clearly, the more we sequence, the better our assembly, but let's be a bit more specific about the relationship between the coverage, c, and the probability of sequencing a particular nucleotide. If we picture shotgun sequencing as randomly drawing individual nucleotides from a genome of length L, the probability of getting a particular one is $1/L$ and the probability of not getting it is $1 - 1/L$. The probability of not sequencing a nucleotide in s trials is

$$P_0 = \left(1 - \frac{1}{L}\right)^s.$$

To simplify this, we first rewrite

$$\left(1 - \frac{1}{L}\right)^s = e^{s \ln(1 - \frac{1}{L})},$$

and use the approximation $\ln(1 + x) \approx x$ to get

$$P_0 \approx e^{-s/L} = e^{-c}.$$

How many nucleotides of our randomized *M. genitalium* genome are expected to be left unsequenced if it is shotgunned with coverage 10?

4.151 How many nucleotides are expected to be left unsequenced with a coverage of 15, or of 20?

4.152 What is the theoretical coverage necessary to achieve an expected combined gap length of 1?

4.153 What is the theoretical coverage necessary to achieve a combined gap length of 0?

4.154 The default sequencing error in `sequencer` is 0.1%, one base per kb. How many contigs do you get in ten trials with coverage 20 and the default sequencing error rate?

4.155 We've seen that sequencing errors wreck our single contig result. Does greater coverage help?

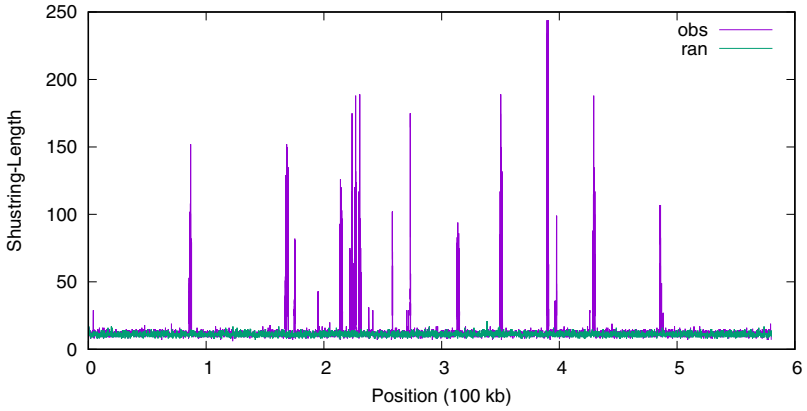


Fig. 4.27 Shustring lengths along the observed genome of *M. genitalium* (*obs*) and its randomized version (*ran*)

4.156 The program `sequencer` simulates sequencing errors as random changes that are indistinguishable from true nucleotides. We've already seen that real sequencing data comes with quality scores for every nucleotide in every read (Fig. 4.18). We won't emulate this here and hence continue to pretend the sequencing data is perfect, while acknowledging that it never is. However, we now turn from the shuffled *M. genitalium* genome to the real thing. To remind ourselves of the difference, look at Fig. 4.27, where the shortest unique substring lengths are plotted along the randomized and the observed genome of *M. genitalium*. Out of the roughly six hundred thousand shustring lengths, we plot only those greater than 24 and 1% of the rest to keep plotting nimble.

⟨*cli*⟩+≡

```
shustring -l mgGenome.fasta | tail -n +3 |
    awk 'NR%100 == 0 || $2 > 24 {print $1, $2, "obs"}' \
        > sl.dat
```

Can you reproduce Fig. 4.27?

4.157 Let's simulate error-free sequencing of *M. genitalium* with coverage 20. How many contigs do you get?

4.158 So far we have simulated shotgun sequencing by picking random reads. In real shotgun sequencing experiments, random fragments of mean length, say, 500 bp are picked and sequenced from both ends. This approach is called *paired-end sequencing*. The information about read pairing is passed on to the assembly program. So we repeat the sequencing experiment, but this time generate paired-end reads in `sequencer` (`-p`) and use the option `-shortPaired` in `velveth`. In `velvetg` we set the "insert length", that is, the length of the fragment sequenced from both ends, to 500 (`-insert_len`).

⟨*cli*⟩+≡

```
awk -f simShot.awk -v c=$c -v s="-e 0 -p" \
    -v t=mgGenome.fasta \
    -v h="-shortPaired -fasta" \
    -v g="-exp_cov $c -ins_length 500"
```

How many contigs and nucleotides do you get?

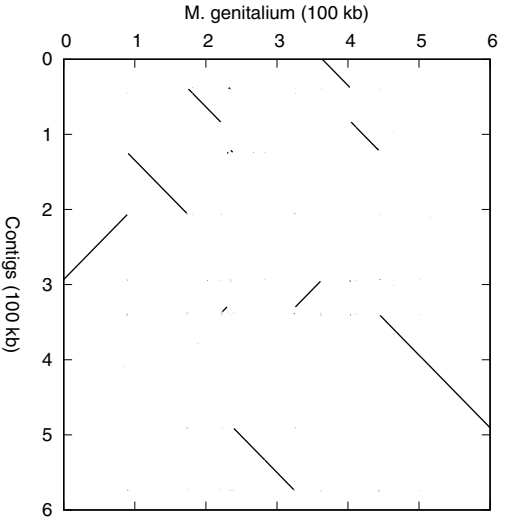


Fig. 4.28 Comparison between the genome of *M. genitalium* and a simulated shotgun sequence with coverage 20

4.159 Let's compare the contigs we just got to the ideal result, the complete forward or reverse strand of the *M. genitalium* genome. For this we concatenate the contigs we got and plot their matches with the genome of *M. genitalium* using `mummer`.

⟨*cli*⟩+≡
```
echo ">contigs" > contigs.fa
grep -v '^>' assem/contigs.fa >> contigs.fa
mummer -b -c mgGenome.fasta contigs.fa | mum2plot |
    awk '{f=100000;print $1/f, $2/f, $3/f, $4/f}' |
    plotSeg -x "M. genitalium (100 kb)" \
            -y "Contigs (100 kb)"
```

Fig. 4.28 shows the plot of our particular contigs. Can you interpret it? What do your contigs look like?

4.160 Apart from the number of contigs, a second popular measure of assembly quality is the so-called N_{50}. This is related to the median contig length, which in turn is similar to the mean contig length. So before we define the N_{50}, we remind ourselves of median and mean. The median is the midpoint of a sorted set of numbers, the mean its average. Consider eleven toy contigs with lengths

$$31, 66, 74, 6, 5, 79, 83, 52, 10, 90, 28$$

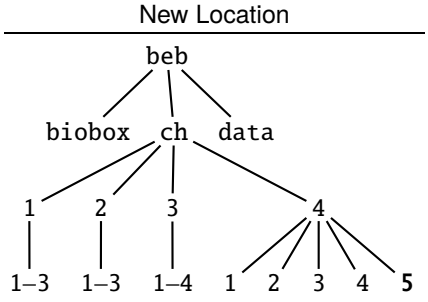
What is the median and the mean contig length?

4.161 The N_{50} is found in three steps. First, we calculate the total contig length. Then we sort the contig lengths and calculate their cumulative length. Finally, we search for the point at which the cumulative length is greater or equal to half the total length. What is the N_{50} of our toy contigs?

4.162 What is the N_{50} of your last assembly?

4.5 Multiple Sequences

We've spent quite some time on aligning pairs of sequences. But that's just a special case of aligning multiple sequences. Multiple sequence alignments are the starting point of many types of analyses in molecular evolution, most prominently among them perhaps phylogeny reconstruction. As we shall see, multiple sequence alignments themselves are calculated from phylogenies.



4.163 Can you make the directory for this section and change into it?

Optimal Multiple Sequence Alignment

New Term
multi-dimensional matrix

4.164 To optimally align n sequences, an n-dimensional alignment matrix is needed. To construct multi-dimensional matrices, we start from zero dimensions, a mere dot in Fig. 4.29A. By doubling this dot and drawing a connecting edge, a one-dimensional matrix is generated, which can accommodate a single sequence, S_1, written from left to right as indicated by the arrow in Fig. 4.29B. In the next round of doubling we get the familiar two-dimensional matrix for aligning pairs of sequences, S_1 and S_2 in Fig. 4.29C. The trace-back is indicated by the red arc. How large is the two-dimensional alignment matrix?

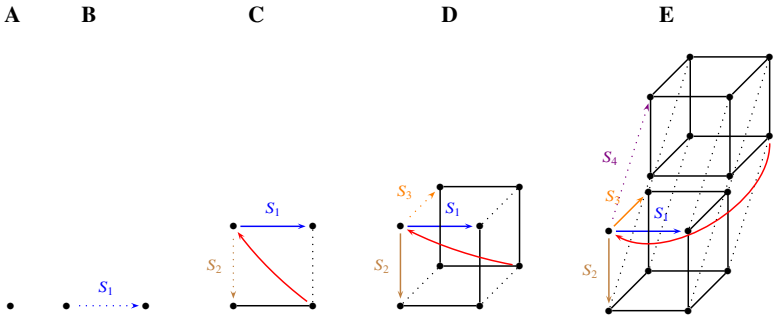


Fig. 4.29 Building multi-dimensional matrices to optimally align multiple sequences. The number of dimensions ranges from zero (**A**) to four (**E**) and corresponds to the number of sequences, S_i, that can be written along its edges and hence aligned. Sequences are indicated by colored arrows labeled S_i. They all start at the same node. The red arc indicates the trace-back

4.165 We can again double the two-dimensional alignment matrix in Fig. 4.29C to get a cube for aligning three sequences. By doubling this, we get the hyper-cube in Fig. 4.29D for aligning four sequences. How large is the matrix required for aligning n sequences length ℓ?

4.166 Do you think optimal multiple sequence alignment is feasible?

Aligning to a Reference

New Terms		
anchor alignment	`sops`	sum-of-pairs score

```
>a
GAGCTCAACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCGCGTC
>b
GAGGTCAGACCGGTTGGTAGGCGTTCATTATTGGGAGGCAGGCGTATGTC
>c
GAGCTCAGACGGGTTGGTGCGTTCATTAATGGGAGGCTGGCGTACGTC
>d
GAGCTCAACCGGTTGGTAGGCGTTCATTATTAGGACGGAGACGCGTC
```

Fig. 4.30 The four sample sequences in `sample.fasta` for multiple sequence alignment

4.167 We've seen that pairwise optimal alignment cannot in practice be generalized to multiple sequences. Instead, the problem of multiple sequence alignment is solved by reducing it to pairwise alignment. The simplest version of this is to align all sequences to a reference, a technique also called *anchor alignment* as the sequences are anchored on the reference. Consider the four sequences a, b, c, and d in Fig. 4.30, which are contained in `sample.fasta`. To carry out anchor alignment, we need all four sequences in separate files, `a.fasta`, `b.fasta`, and so on. Can you split `sample.fasta` into its component sequences (`getSeq`)?

4.168 Let's make sequence a our reference and align sequence b to it with our optimal aligner `al`. Can you do that?

4.169 We'd like to save the output of `al` in FASTA format. For this we write the program `al2fasta.awk`, where we interact with the user and print the sequence.

Prog. 4.11 (`al2fasta.awk`)

⟨*al2fasta.awk*⟩≡
  ⟨*Interact with user, Prog. 4.11*⟩
  ⟨*Print sequence, Prog. 4.11*⟩

Sometimes we'd like to save the query, sometimes the subject. So we tell the user that we are converting a *target*, which can be either the query or the subject.

⟨*Interact with user, Prog. 4.11*⟩≡

```
BEGIN {
  if (target != "Subject" && target != "Query") {
    u = "Usage: awk -f al2fasta.awk "
    u = u "-v target=<Subject|Query>"
```

```
      print u
      exit
    }
  }
```

When printing the sequence, we either print the header, or the sequence data.

⟨*Print sequence, Prog. 4.11*⟩≡
  ⟨*Print header, Prog. 4.11*⟩
  ⟨*Print sequence data, Prog. 4.11*⟩

The name of the sequence is marked by the first "Query" or "Subject" line we encounter.

⟨*Print header, Prog. 4.11*⟩≡

```
$1 == target && NR < 3 {
  printf ">%s\n", $2
}
```

Can you print the sequence data?

4.170 Now we are in a position to save our nascent anchor alignment in, say, `anc.fasta`. Can you do that?

4.171 Any gap introduced in the reference is carried into the next round. So the reference sequence might accumulate gaps as we go along. Since `al` cannot align gap characters, we replace them by `N` and store the new reference sequence in `r.fasta`.

⟨*cli*⟩+≡

```
al a.fasta b.fasta | tail -n +7 | grep Q | tr '-' 'N' |
    awk '{printf ">r\n%s\n", $3}' > r.fasta
```

What does our multiple sequence alignment look like after we've anchored sequence c?

4.172 Can you add sequence d to our alignment?

4.173 Now that we've got an alignment, we'd like to score it. A popular score for multiple sequence alignments is the sum-of-pairs score. It is calculated by iterating over the alignment columns. For each column, all pairs of residues are scored, pairs of gaps are ignored, and a residue paired with a gap is given a gap score. There is no gap opening. The grand total of these pair scores is the sum-of-pairs score. Consider the alignment

```
A-
A-
TT
```

and let match be 1, mismatch -3, and gap -2. What is the sum-of-pairs score of this mini-alignment? You can test your answer with `sops`, a program for calculating the sum-of-pairs score.

4.174 What is the sum-of-pairs score of our anchor alignment?

4.175 We picked sequence *a* as our reference, but we might as well have picked another one. What does the alignment anchored on *c* look like?

4.176 What is the sum-of-pairs score of the alignment anchored on sequence *c*?

4.177 We've seen that the sum-of-pairs score changes when we switch from sequence *a* as anchor to sequence *c*. Can you explain why?

```
a GAGCTCA-ACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCGCGTC
d .......-..........A.................C....A......
b ...G...G..........A.............G.....C.....
```

Fig. 4.31 Blast anchor alignment of the four sequences in Fig. 4.30

4.178 Blast has an anchor mode, where the query serves as the anchor. If we take sequence *a* as the query and the rest as the subject, we can set the output to one of the four anchor formats, 1–4. We find format 3 the most useful.

⟨*cli*⟩+≡

```
cat b.fasta c.fasta d.fasta > subject.fasta
blastn -task blastn -query a.fasta \
         -subject subject.fasta -outfmt 3
```

Fig. 4.31 shows a slightly edited version of the result. Can you make sense of it?

4.179 What does output format 4 look like?

4.180 Output formats 1 and 2 make the same distinction between matches as formats 3 and 4. But there is another difference. Can you spot it?

Aligning without Reference

New Terms		
guide tree	progressive alignment	`upgma`
`mafft`	`sed`	

4.181 Without an anchor to pile our sequences on, we can still calculate a multiple sequence alignment efficiently. This is done in three steps: Calculate pairwise distances (Fig. 4.32A and B), cluster distances into a guide tree (Fig. 4.32C), and align sequences in the order implied by the guide tree (Fig. 4.32D–F). This procedure is called *progressive* alignment as opposed to optimal alignment in multiple dimensions, which is too slow. Consider again our four example sequences in `sample.fasta` (Fig. 4.30). We begin constructing their progressive alignment by calculating their distances as entries in a distance matrix,

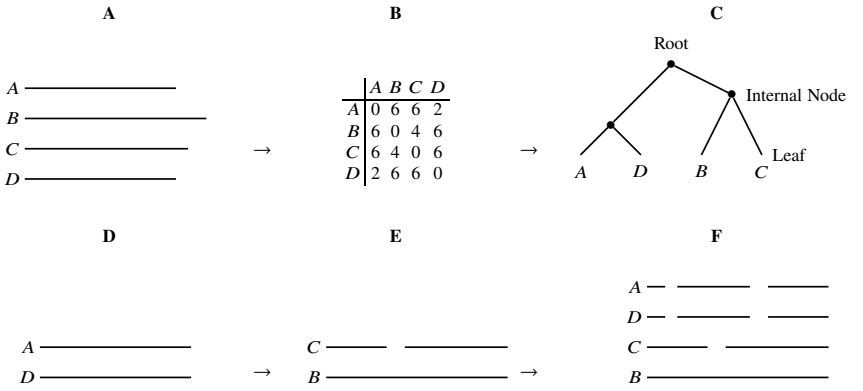


Fig. 4.32 Alignment of multiple sequences as progressive pairwise alignment along a guide tree: Four sequences (**A**) are aligned pairwise and their distances stored in matrix **B**, which is summarized as tree **C**; traversal of this tree from the leaves to the root guides the alignment (**D**–**F**)

	a	*b*	*c*	*d*
a	0			
b		0		
c			0	
d				0

As distances we use the number of errors returned by `al`. Can you fill in the distance matrix?

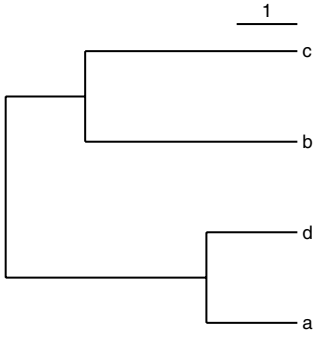


Fig. 4.33 The guide tree for the four sequences in Fig. 4.30

4.182 In progressive alignment, the distance matrix (Fig. 4.32B) is converted into a tree (Fig. 4.32C), which guides the subsequent construction of the multiple sequence alignment. The tree is thus called *guide tree*. We get to know the details of methods

for tree construction from distances in the next chapter. For now, we just take the program `upgma` to convert our distance matrix into a tree and plot it with `plotTree`. The format of the distance matrix read by `upgma` is the Phylip[2] format, which consists of the number of taxa, 4 in our case, followed by four rows of data. Each data row consists of the sequence name, followed by the distances. Fig. 4.33 shows our guide tree. Can you reproduce it?

4.183 The multiple sequence alignment we're after is constructed by traversing the guide tree in Fig. 4.33 from the leaves to the root. The first internal node we encounter is the cluster (a, d), so we align sequences a and d first. Can you save their alignment in `prog.fasta`?

4.184 The second pair of sequences we reach walking up the guide tree in Fig. 4.33 is (b, c). Can you add their alignment to `prog.fasta`?

4.185 The last internal node we reach on our way up the guide tree in Fig. 4.33 is the root, $((a, d), (b, c))$. So we align the two alignments (a, d) and (b, c). We do this by inserting gaps into alignments (a, d) and/or (b, c) in `prog.fasta`. The first gap we insert is into (a, d) and converts `CAAC` into `CA-AC`. We do this using the stream editor `sed`.

⟨*cli*⟩+≡
```
sed 's/CAAC/CA-AC/' prog.fasta | fasta2tab
```

The `sed` command `s/a/b/` means substitute `a` by `b`. Can you replace `prog.fasta` with its new version?

4.186 Alignment (a, d) is still two residues shorter than alignment (b, c). Can you insert another gap to fix this?

4.187 What is the sum-of-pairs score of the alignment in `prog.fasta`?

4.188 `mafft` is a popular program for computing multiple sequence alignments [24, 25]. It's name stands for Multiple sequence Alignment using Fast Fourier Transform. Fast Fourier transform is a computational technique that we only mention here to explain the double f in `mafft`. Let's apply `mafft` to our sample sequences in default mode.

⟨*cli*⟩+≡
```
mafft sample.fasta
```

What is the sum-of-pairs score of the resulting multiple sequence alignment?

4.189 `mafft` tells us that its default alignment strategy, the authors call it FFT-NS-2, is "fast but rough". We have just seen that there is room for improvement of the alignment score, so we take the authors' advice and use the `--auto` option.

⟨*cli*⟩+≡
```
mafft --auto sample.fasta
```

[2] `evolution.genetics.washington.edu/phylip.html`

What is the score of the alignment you get now?

4.190 The strategy just picked by `mafft`, L-INS-i, is described by the authors as "probably most accurate", but "very slow". Speed is not an issue with our short toy sequences, but we know that accuracy could still be improved. Here is another variant of `mafft`, which uses an algorithm the `mafft` authors call "parttree" to calculate the guide tree.

⟨*cli*⟩+≡

```
mafft --parttree sample.fasta
```

What is the score of the alignment this time?

4.191 Let's turn from toy sequences to real data. We have already looked at the primate mitochondrial genomes in `primates.fasta`. Here we concentrate on the simians among the primates, *P. troglodytes*, *P. paniscus*, *H. sapiens*, *G. gorilla*, *P. pygmaeus*1, *P. pygmaeus*2, and *H. lar*. Can you extract their sequences into `simians.fasta`?

4.192 What is the sum-of-pairs score of the simians alignment with the `--auto` option?

4.193 With our toy sequences we found that the alignment quality could be improved by using the `--parttree` option and nothing else. Is this also the case with the simians data?

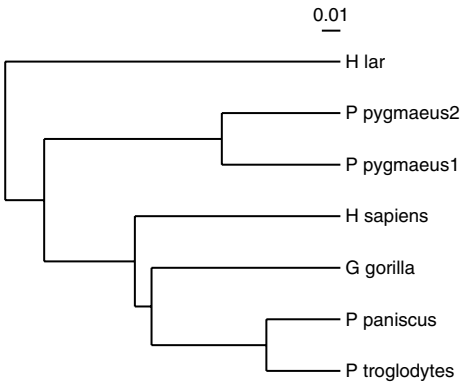


Fig. 4.34 The `mafft` guide tree for our seven simian sequences

4.194 `mafft` can save its guide tree to a file called `input.tree`; and since at this point we are only interested in the guide tree, we throw away the rest of the output by redirecting it to the null device.

⟨*cli*⟩+≡

```
mafft --auto --treeout simians.fasta > /dev/null
```

The tree is now in `simians.fasta.tree`. Fig. 4.34 shows it, for which we reformatted the taxon labels with `sed`.

⟨*cli*⟩+≡

```
sed 's/^[0-9]_//' simians.fasta.tree | plotTree
```

As we've said, the `sed` command `s/a/b/` means substitute `a` by `b`. What are `a` and `b` in this case?

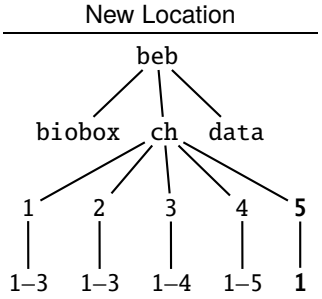
4.195 The guide tree in Fig. 4.34 differs from the standard simian phylogeny. Can you see how?

Chapter 5
Evolution Between Species: Phylogeny

Sometime in the summer of 1837, Charles Darwin (1809–1882) wrote in his notebook "I think", and continued by drawing a phylogeny. When he published his thinking in 1859, *The Origin of Species* contained a single figure: a phylogeny. We already saw that phylogenies in the form of guide trees are useful for computing multiple sequence alignments. But beyond clever computing, phylogenetic trees embody biologists' thinking. Taxonomies are trees and evolution is pictured as a tree. Trees consist of trees, and this hierarchical structure is explored further in the following section. Phylogenetic trees may or may not have a root, and we show the distinct methods to calculate rooted and unrooted trees in two later sections.

5.1 Trees of Life

Trees are by no means new to us at this point in the course, and Fig. 5.1 shows another one with 10 taxa, whose common ancestor is the root. The tips of the branches are all lined up at time zero indicated by the vertical dotted line; left is the past, right the future. In this section we show how to write, traverse, and count trees.



5.1 Can you make a directory for this section?

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_5

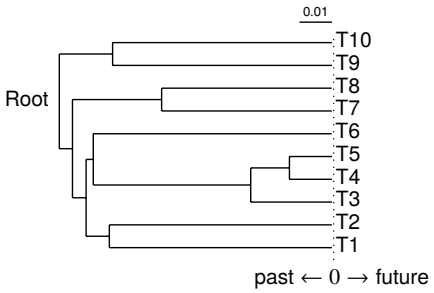
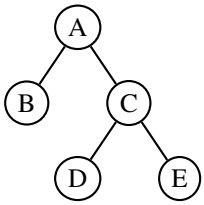


Fig. 5.1 A random tree of 10 taxa with the *Root* marked; all branches end at time zero, the present, indicated by the dotted line

New Terms

`genTree`	molecular clock	preorder traversal
inorder traversal	postorder traversal	`travTree`

5.2 Phylogenies are trees, which are often written using nested parentheses [29, p. 312]. For example, `(A(B)(C(D)(E)))` corresponds to the tree



Each node has a name and all branches have the same length. Phylogenies are often written in a specialized parenthesis notation, the Newick format[1]. In Newick format, the tree above is

```
(B,(D,E)C)A;
```

Can you spot the differences between Newick and plain parentheses?

5.3 Phylogenies typically only contain leaf labels, interpreted as extant species, while the internal nodes—the common ancestors—usually remain anonymous. Can you write the leaves-only version of `(B,(D,E)C)A;`?

5.4 In addition to the topology of a phylogeny, its branch lengths are meaningful. In Newick format, branch lengths are delimited by colons, for example

```
(B:1,(D:1,E:1));
```

The program `plotTree` converts this string to the tree in Fig. 5.2. Can you reproduce it?

[1] `evolution.genetics.washington.edu/phylip/newick_doc.html`

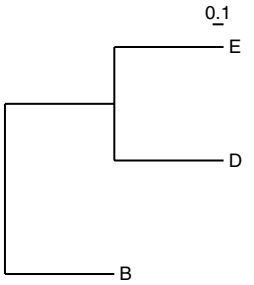


Fig. 5.2 Graphical representation of `(B:1,(D:1,E:1));`

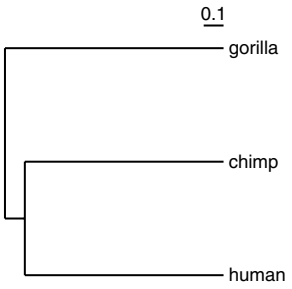


Fig. 5.3 Schematic phylogeny of human, chimp, and gorilla

5.5 Fig. 5.3 shows a fanciful phylogeny of human, chimp, and gorilla. Can you reproduce it?

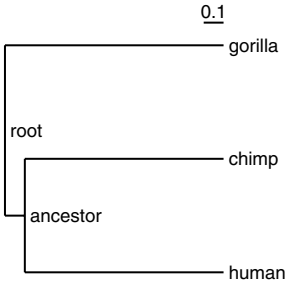


Fig. 5.4 Schematic phylogeny of human, chimp, and gorilla with internal nodes labeled *ancestor* and `root`

5.6 We just said that in phylogenies the internal nodes are usually not labeled. However, they can be labeled, as shown in Fig. 5.4. Can you reproduce it?

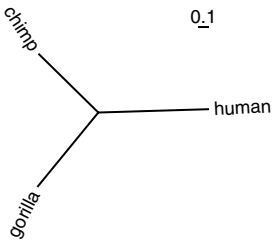


Fig. 5.5 Unrooted tree of human, chimp and gorilla

5.7 In mathematics and computer science, all trees have a root. But in biology, we often talk about "unrooted" trees, for example Fig. 5.5 is the unrooted version of our human/chimp tree. Can you reproduce it?

5.8 Strictly speaking, there is no such thing as an unrooted tree. Can you explain in what sense the tree in Fig. 5.5 is, in fact, rooted, but also "unrooted"?

5.9 A classical method to root a tree is to look for the most distant pair of taxa on the tree and place the root mid-distance. This is called "midpoint rooting" and is implemented in the program `midRoot`. What do you get when you apply it to the unrooted tree in Fig. 5.5?

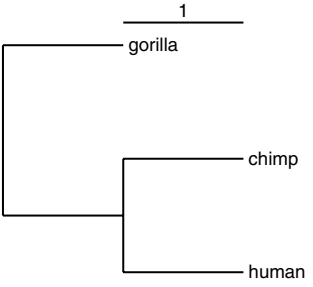


Fig. 5.6 Plot of `((human,chimp),gorilla);`

5.10 The program `plotTree` always shows a scale bar, even if we don't put explicit branch lengths in the underlying Newick string. We can set the scale with `-c`.

⟨*cli*⟩+≡
```
printf "((human,chimp),gorilla);\n" | plotTree -c 1
```

Fig. 5.6 shows the tree we just generated. What are the assumed branch lengths if we don't state any?

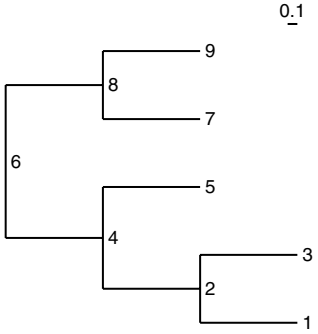


Fig. 5.7 A tree with all nine nodes labeled

5.11 Fig. 5.7 shows a tree with all nine nodes labeled. Can you reproduce it?

5.12 As we've already said, a tree consists of trees. So in Fig. 5.7 the tree rooted on node 6 consists of subtrees rooted on nodes 8 and 4, and so on. This hierarchical, or recursive, structure leads to a method for traversing a tree, where the function `traverse` calls itself:

```
traverse(node)
   visit(node)
   traverse(leftChild(node))
   traverse(rightChild(node))
```

Since the root is always visited first, then each one of the two children in turn, the procedure is called *preorder traversal*. In what order are the nodes of Fig. 5.7 visited if we start at the root? You can check your result with `travTree`.

5.13 Instead of preorder, a tree can also be traversed inorder by first visiting the left child, then the root in the middle, and finally the right child:

```
traverse(node)
   traverse(leftChild(node))
   visit(node)
   traverse(rightChild(node))
```

In what order do we now visit the nodes of Fig. 5.7?

5.14 Lastly, we can also visit the root last, which is called postorder traversal:

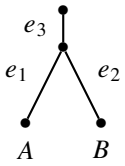
```
traverse(node)
   traverse(leftChild(node))
   traverse(rightChild(node))
   visit(node)
```

In what order does this visit the nodes in our example tree Fig. 5.7?

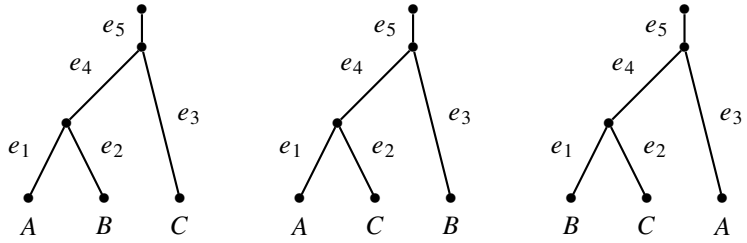
5.15 The example tree right at the beginning of this chapter, Fig. 5.1, shows a random tree of ten taxa generated with `genTree` and plotted with `plotTree`. Draw a few such trees yourself. What difference does the `-a` option make?

5.16 When reconstructing phylogenies, we often talk about a *molecular clock*. Under the molecular clock model, mutations occur with constant rate along all branches of a phylogeny. Trees generated by `genTree` have this property. But in contrast to the clocks of everyday life, the molecular clock is stochastic, which is why the branches fluctuate around the zero line, rather than end there exactly. Stochastic processes tend to converge with the number of trials, in our case the number of mutations. Their expected number can be set with `-t`. What do you observe with `-t` values of 100 compared to, say 10,000?

5.17 We like drawing random trees, it's fun. But however many random trees we draw with, say, ten taxa on an idle Friday afternoon, they all have distinct branching patterns. That's because there are so very many possible trees with ten taxa. Exactly how many, can be computed based on the following consideration [14, p. 20ff]: Two taxa are connected by a single rooted tree:



The third taxon, C, can be added to any of the three edges e_1, e_2, and e_3, giving three trees:



The fourth taxon can be added to any one of the five edges $e_1, e_2, ..., e_5$, yielding $3 \times 5 = 15$ rooted trees. How many trees with five taxa are there?

5.18 In general, the number of rooted, bifurcating trees for n taxa is

$$3 \times 5 \times ... \times (2n - 3).$$

We write the program `numTrees.awk` to compute the number of rooted phylogenies as a function of n.

Prog. 5.1 (`numTrees.awk`)

⟨*numTrees.awk*⟩≡

```
BEGIN {
  if (!n) {
    print "Usage: awk -f numTrees.awk -v n=<n>"
    exit
  }
  ⟨Calculate number of trees, Pr. 5.1⟩
  ⟨Print number of trees, Pr. 5.1⟩
}
```

Can you calculate the number of trees?

5.19 Can you print the number of trees as a function of the number of taxa?

5.20 What is the number of trees with ten taxa?

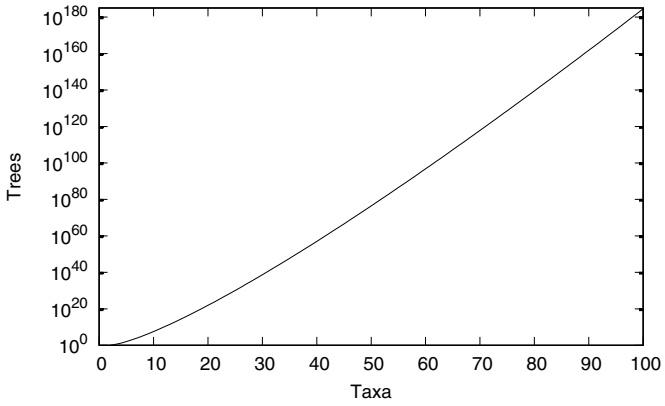


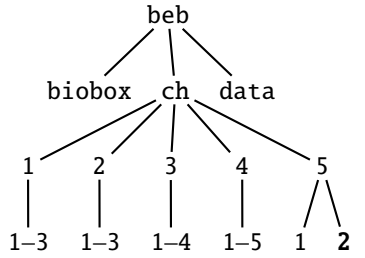
Fig. 5.8 The number of possible trees as a function of the number of taxa

5.21 Fig. 5.8 shows the number of possible trees as a function of the number of taxa, which quickly grows very large indeed. Can you reproduce Fig. 5.8?

5.2 Rooted Trees

How can we calculate phylogenetic trees from data rather than just simulate them as we did in the previous section? Given the huge number of possible phylogenies, finding the correct one might seem impossible. Fortunately, it is well possible once we understand a tree as the summary of pairwise distances. This means we can work our way from distances to trees, and the simplest method for this returns rooted trees.

New Location



5.22 Can you make a directory for this section and change into it?

New Terms

`phylonium`	three point criterion	ultrametric distances
`pps`		

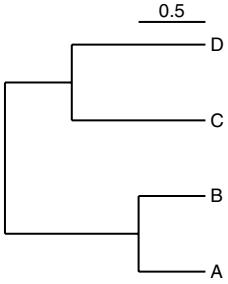


Fig. 5.9 Example phylogeny with four taxa

5.23 Fig. 5.9 shows a phylogeny with a scale bar. This implies all pairwise distances between taxa. For example, the distance between A and B is $0.5 + 0.5 = 1$. We can enter such distances in a distance matrix.

	A	B	C	D
A	-			
B	1	-		
C	?	?	-	
D	?	?	?	-

Can you enter the missing five distances?

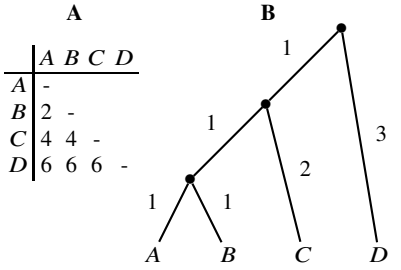


Fig. 5.10 A distance matrix (**A**) and the corresponding tree with branch lengths (**B**)

5.24 Our example matrix contains only the lower triangle. Why can we omit the upper triangle?

5.25 Let's work the other way round, from the distance matrix to the tree. Fig. 5.10 shows another distance matrix and the corresponding tree with all branch lengths marked. The first step in constructing this tree is to pick the two entries with the smallest distance and construct the corresponding partial tree from them. Can you do that?

5.26 Picking a pair of taxa means we merge them in a new node, their ancestor. Say, we merge taxa x and y into (x, y). The distance between some other taxon, w, and (x, y), is the average distance between w and x and w and y. Since we merged taxa A and B, the distance between (A, B) and C is $(4 + 4)/2 = 4$; similarly, the distance between (A, B) and D is $(6 + 6)/2 = 6$. So the new distance matrix is

	(A, B)	C	D
(A, B)	-		
C	4	-	
D	6	6	-

We repeat picking a pair of taxa with the smallest distance to construct the next partial tree. Can you do that?

5.27 After the merge, our new distance matrix is

	(A, B, C)	D
(A, B, C)	-	
D	6	-

Now there is no choice which pair to pick, and we end with Fig. 5.10B. Distance matrices are stored in Phylip format, which we've already seen, and which turns the matrix in Fig. 5.10A into

```
4
A 0 2 4 6
B 2 0 4 6
C 4 4 0 6
D 6 6 6 0
```

The program `upgma` converts this distance matrix into the corresponding tree. Can you reproduce Fig. 5.10B?

5.28 Our program is called `upgma`, because the tree-building algorithm it implements is called "Unweighted Pair-Group Method using an Arithmetic average", or UPGMA, which we shall write Upgma [41, p.359f]. The name of this method is perhaps its most challenging aspect. As we've seen, Upgma works by repeatedly picking the smallest entry from a distance matrix and adjusting the matrix accordingly. The program `upgma` can also print these intermediate matrices. Can you do that?

Table 5.1 Another distance matrix

	A	*B*	*C*	*D*
A	-			
B	6	-		
C	2	6	-	
D	6	4	6	-

5.29 Table 5.1 shows another set of distances. Can you calculate the Upgma tree from them? You can check your intermediate matrices with `upgma`.

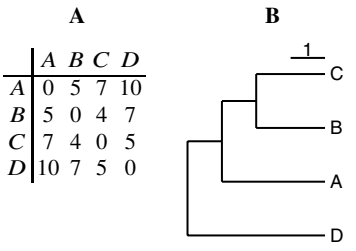
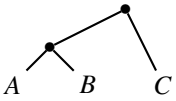


Fig. 5.11 Distance matrix (**A**) and corresponding Upgma tree (**B**)

5.30 In a Upgma tree, all terminal branches end exactly at the implied zero line. We've already seen that this means the underlying sequence data mutates with a constant rate along all branches, in other words, the molecular clock applies. But Upgma doesn't even allow for stochastic wobble around the zero line. What does that imply about the mutation rates along the tree branches?

5.31 Fig. 5.11 shows another distance matrix and the corresponding tree. What do you notice when you try to recover the input distances from the tree?

5.32 We've seen there can be discrepancies between a Upgma tree and the distances from which it was constructed. Perfect agreement between tree and distances is called "ultrametricity". Any three ultrametric distances fit on a tree like this



In other words, two of the distances are equal, in this case $d_{A,C} = d_{B,C}$, and the other distance cannot be greater than them, $d_{A,B} \leq d_{A,C} = d_{B,C}$. This is called the "three point criterion". It ensures that the leaves in an ultrametric tree fall in line. Ultrametricity also implies that for n taxa there are no more than $n-1$ distinct entries in the distance matrix. How many distinct entries are in Fig. 5.11A?

5.33 Having no more than $n-1$ distinct entries is necessary for ultrametricity, but it isn't sufficient. Can you think of an example to show this?

5.34 Let's construct a phylogeny from some real sequence data. The file

`hominidae.fasta`

contains the D-loop of the mitochondrial genome. The D-loop is a 900 bp region of the mitochondrial genome that mutates rapidly. The sequences were obtained from the four great apes, human, chimp (*Pan*), gorilla, and orangutan (*Pongo*). These four apes are also known as the *Hominidae*, hence the name of the file. How many nucleotides is each sequence long (`cres`)?

5.35 The program `pps` prints the polymorphic sites in an alignment. How many polymorphic sites are in the great apes alignment? Is the gapped site counted as polymorphic?

5.36 Distances are calculated by counting the number of mismatches between sequences. Let's consider the first ten polymorphisms of the great apes and format them for easy eyeballing. A dot indicates a match to the human sequence in the first row.

⟨*cli*⟩+≡

```
pps -d hominidae.fasta | cutSeq -r 1-10 | getSeq -c Pos |
    fasta2tab | tr -d 'a-z'
```

Can you fill in the corresponding mismatch matrix?

	Hom	Pan	Gor	Pon
Hom				
Pan				
Gor				
Pon				

You can check your result with `dnaDist`.

5.37 Next, we look at all the mismatches between the four sequences. Do they form ultrametric distances?

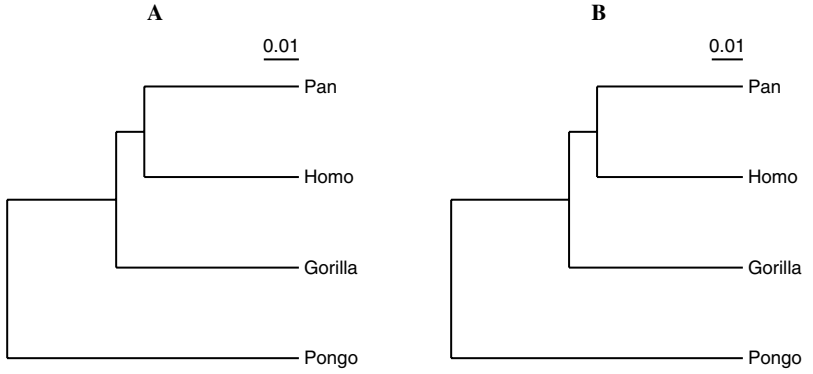


Fig. 5.12 Phylogenies of the great apes based on the D-loop of the mitochondrial genome; uncorrected mismatches (**A**), Jukes-Cantor distances (**B**)

5.38 Molecular distances are usually expressed on a per-site basis. The simplest molecular distance is the number of mismatches per site. Fig. 5.12A shows the corresponding tree. Can you reproduce it? What is our closest relative among the great apes?

5.39 As the number of mutations grows, the probability that a mutation affects the same position twice also grows. This means that the number of mismatches is a lower bound on the number of mutations. The relationship between the number of mutations per site, k, and the number of mismatches per site, m, is summarized in the Jukes-Cantor equation, which we already saw a while back, equation (2.3). What is the largest value of m for which k is defined?

5.40 What is the expected number of mismatches per site for two random sequences? You can check your answer with `ranseq` and `dnaDist`.

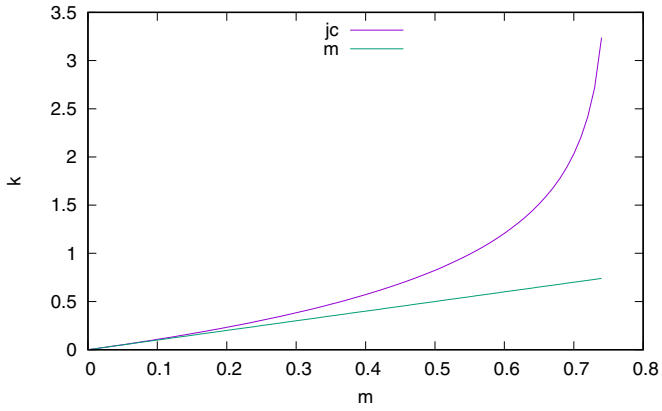


Fig. 5.13 The number of mutations per site, k, as a function of the number of mismatches per site, m, when calculated with the Jukes-Cantor equation, *jc*, or without correction, m

5.41 Fig. 5.13 shows the number of mutations as a function of the number of mismatches. Clearly, the number of mismatches can be a strong underestimate of the number of mutations. We generated Fig. 5.13 with the program `jc.awk`.

Prog. 5.2 (`jc.awk`)

⟨*jc.awk*⟩≡

```
BEGIN {
  for (i = 0; i < 0.75; i += 0.01) {
    jc = -3/4 * log(1 - 4/3 * i)
    print i, jc, "jc"
    print i, i,  "m"
  }
}
```

Can you reproduce Fig. 5.13?

5.42 Fig. 5.12B shows the great apes tree with Jukes-Cantor distances. Can you draw it? Do you think the difference to the uncorrected tree is important?

5.43 The file `primates.fasta` contains the full mitochondrial genome sequences of primates. How many sequences does it contain and how long are they on average?

5.44 Fig. 5.14 shows the Upgma trees of primates based on two methods to compute the distances. The first, in Fig. 5.14A, is based on a `mafft` alignment. How long does it take to reproduce this tree?

5.45 Instead of aligning sequences, we can estimate distances between them directly, which is often much quicker. The program `phylonium` implements a fast method to calculate alignment-free distances based on exact matches [26]. It takes as input

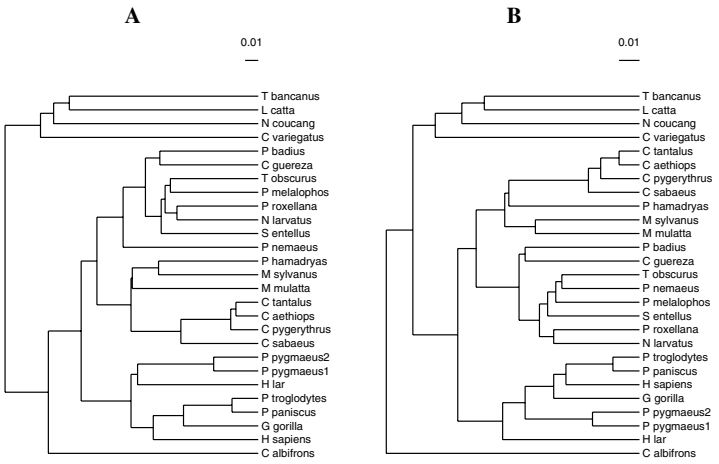


Fig. 5.14 Upgma tree of primates from their mitochondrial genomes; based on alignment (**A**), based on fast distance estimation (**B**)

genomes in separate files, so we split `primates.fasta` into one file per sequence. To do this, we use the neat property of Awk that we can use redirection as if we were working directly on the shell. Then we create a new directory `primates`, and move the sequences into it.

⟨*cli*⟩+≡

```
fasta2tab primates.fasta |
    awk '{f=$1 ".fasta";printf ">%s\n%s\n", $1, $2 > f}'
mkdir primates
mv *_*.fasta primates
```

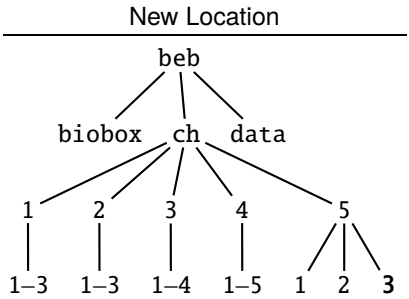
How long does it take to reproduce Fig. 5.14B (we ignore `phylonium`'s warnings about lack of homology)?

5.46 Do the two primate trees in Fig. 5.14 have the same topology?

5.3 Unrooted Trees

All groups of organisms have a common ancestor, which means that their phylogenetic trees are, in fact, rooted. As we've seen, the Upgma algorithm quickly generates rooted phylogenies. Its underlying assumption that a real clock was used to measure branch lengths fits our intuition that the leaves of a phylogeny should all be located in the present. However, branch lengths are usually not measured with conventional clocks, but estimated from the number of mutations, which form a molecular clock.

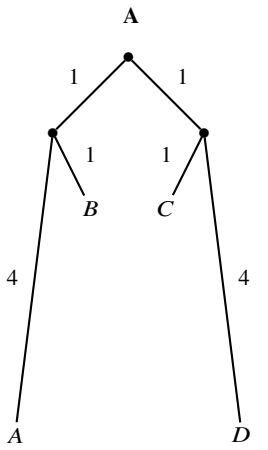
The molecular clock is stochastic and hence won't fit a Upgma tree. If the discrepancy is small, we still get the correct tree, but as we shall see, this is not guaranteed. Fortunately, there is a slightly more involved method of tree reconstruction from distances that works where Upgma fails. This method is called neighbor-joining and gains accuracy by losing rootedness. We can add a root afterwards, but a more pressing problem is to quantify the uncertainty surrounding the placement of any node in the tree, not just the root. We solve this problem by pulling ourselves up by our bootstraps.



5.47 Can you construct the directory for this section and change into it?

New Terms

additive distances	`clac`	midpoint rooting
bootstrap	four point criterion	neighbor-joining



B

	A	*B*	*C*	*D*
A	-	5	7	10
B	5	-	4	7
C	7	4	-	5
D	10	7	5	-

Fig. 5.15 Tree with unequal terminal branches (**A**) and the corresponding distance matrix (**B**)

5.48 Consider the tree in Fig. 5.15A with its drastically unequal terminal branches. Fig. 5.15B shows the distances read off the tree. Evidently, the distances are not ultrametric. This means they don't conform to the three point criterion, according to which among the three distances between any triple of taxa, two are equal and the third not greater. Can you find three taxa that contradict this?

5.49 What is the first pair of taxa joined by Upgma when analyzing the distances in Fig. 5.15B? How does that compare to the true tree?

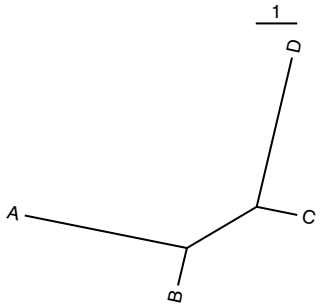


Fig. 5.16 Unrooted version of the phylogeny in Fig. 5.15A

5.50 Distances that fit a tree, like those in Fig. 5.15B, are called *additive*. Additivity is based on four taxa. Fig. 5.16 shows such a tree, which is in fact the tree we started off with in Fig. 5.15A after unrooting. There are six distances defined by this tree, d_{AB}, d_{AC}, d_{AD}, d_{BC}, d_{BD}, and d_{CD}. However, the tree only has five branches, four terminal branches and one central branch. So the tree is *overdetermined* by the distances. Two of the distances only involve terminal branches. Which are they?

5.51 The remaining four distances, d_{AC}, d_{AD}, d_{BC}, and d_{BD} traverse the stem of the tree. We can divide them into two pairs that each cover the full tree. Can you spot these pairs?

5.52 The sum of distances that cover the terminal branches cannot be greater than the sum of distances that cover the whole tree,

$$d_{AB} + d_{CD} \leq d_{AD} + d_{BC}, d_{AB} + d_{CD} \leq d_{AC} + d_{BD}.$$

On the other hand, the pairs of distances that cover the whole tree must be equal,

$$d_{AC} + d_{BD} = d_{AD} + d_{BC}.$$

Combining these two observations, we arrive at the four point criterion of additivity,

$$d_{AB} + d_{CD} \leq d_{AD} + d_{BC} = d_{AC} + d_{BD}.$$

Can you convince yourself that the four point criterion holds for the distances in Fig. 5.15B?

5.53 The neighbor-joining algorithm recovers the correct tree from the distances in Fig. 5.15B. We begin by just writing down their top triangle:

	A	B	C	D	r_i
A	-	5	7	10	
B		-	4	7	
C			-	5	
D				-	

The last column, r_i, is reserved for the row sums. Compute by hand the values of r_i, remembering that the distance matrix is symmetrical.

5.54 In the next step of neighbor-joining we prepare the later selection of a pair of neighbors. For this we compute for each pair of taxa, i, j, the difference between its distance, d_{ij}, and the normalized sum of the corresponding row sums, r_i, r_j:

$$S_{ij} = d_{ij} - \frac{r_i + r_j}{n-2},$$

where n is the number of taxa. Can you compute the S_{ij}-values by hand and write them in the lower triangle of the distance matrix? You can check your results with `nj` in matrix-printing mode.

5.55 Instead of clustering the pair of taxa with the smallest distance as in Upgma, neighbor-joining clusters the taxa with the smallest S_{ij}. If the new cluster is called c, the distance between c and some other cluster, k, is

$$d_{kc} = (d_{ik} + d_{jk} - d_{ij})/2.$$

The other distances are unchanged. Let's cluster (A, B). Can you adjust the distance matrix accordingly? You can again check your result with `nj`.

5.56 We still need to know the lengths of the branches connecting the new cluster c and leaves i and j:

$$d_{ic} = \frac{(n-2)d_{ij} + r_i - r_j}{2(n-2)},$$

and

$$d_{jc} = \frac{(n-2)d_{ij} + r_j - r_i}{2(n-2)}.$$

Compute (by hand) the branch lengths for the new cluster.

5.57 To summarize, the neighbor-joining algorithm consists of four steps; given distances d_{ij},

- compute the row sums

$$r_i = \sum_j d_{ij}$$

- compute the matrix for neighbor selection

$$S_{ij} = d_{ij} - (r_i + r_j)/(n-2)$$

- identify neighbors as taxa with smallest S_{ij} and cluster them in node c with

$$d_{kc} = (d_{ik} + d_{jk} - d_{ij})/2$$

- calculate branch lengths

$$d_{ic} = \frac{(n-2)d_{ij} + r_i - r_j}{2(n-2)}$$
$$d_{jc} = \frac{(n-2)d_{ij} + r_j - r_i}{2(n-2)}$$

This procedure is repeated until there are only three clusters left, i, j, k, which is the stage we have reached in our example. These are joined to the pseudo-root r, by branches with the following lengths

$$d_{ri} = (d_{ij} + d_{ik} - d_{jk})/2$$
$$d_{rj} = (d_{ji} + d_{jk} - d_{ik})/2$$
$$d_{rk} = (d_{ki} + d_{kj} - d_{ij})/2$$

What are the lengths of the last three branches added to our example tree?

5.58 Now we know how neighbor-joining works. Can you use its implementation in `nj` to reproduce Fig. 5.16?

5.59 It is often more convenient to read rooted trees than unrooted trees. A popular method for rooting a tree is to place the root midway between the most distant taxa on a tree. This is called "midpoint rooting". Can you midpoint root our example tree? You can check your answer with `midRoot`.

5.60 Let's turn from toy data to the *Hominidae* sequences in `hominidae.fasta`. Fig. 5.17 shows the distances between the sequences next to the neighbor-joining tree. Can you reproduce the tree?

5.61 Are the *Hominidae* distances additive?

5.62 Given that biological data is always noisy, we'd like to gauge the reliability of the trees we calculate. A classical method for doing this is called the bootstrap. This is a general statistical method to answer the question, how much would our result vary if we sampled again? Applied to our phylogeny problem, the question becomes, how much would our phylogeny vary, if we repeatedly sampled other sets of 896

A

	Homo	Pan	Gorilla	Pongo
Homo	0.00	0.10	0.11	0.18
Pan	0.10	0.00	0.11	0.19
Gorilla	0.11	0.11	0.00	0.19
Pongo	0.18	0.19	0.19	0.00

B

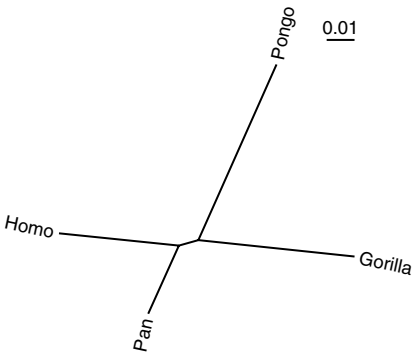


Fig. 5.17 Distances (**A**) and tree (**B**) for the *Hominidae* data set

homologous nucleotides from the genomes of our *Hominidae*? Collecting additional samples is often difficult. The answer provided by bootstrap is to resample the original sample with replacement [12]. Compared to actually going to the lab and collecting more data, this solution is so simple, it almost amounts to pulling yourself up by your bootstraps.

The bootstrap method is widely used and our program `dnaDist` implements the classical version where the 896 columns in our *Hominidae* alignment are sampled with replacement to generate a new pseudo-sample. The distance matrix is computed from this pseudo-sample, and the procedure is repeated. Can you run `dnaDist` with ten bootstrap iterations?

5.63 We can pipe the distance matrices through `nj` to get the corresponding trees. Does their topology vary?

5.64 What we'd like to know is the frequency of, say, the (Homo, Pan) clade in a large set of trees. The program `clac` is a clade counter. What is the frequency of the (Homo, Pan) clade in 1000 bootstrap samples of the *Hominidae* sequence data?

5.65 Given a reference tree, `clac` can also enter the bootstrap values. This is how we generated Fig. 5.18. Notice that only one of the two internal nodes is labeled with a bootstrap percentage. The other node is the (effective) root and since the clade defined by the root always has 100% bootstrap support, `clac` omits that value. Can you reproduce Fig. 5.18?

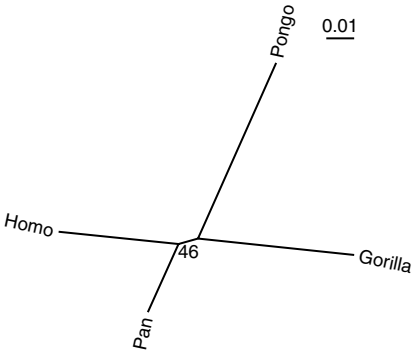


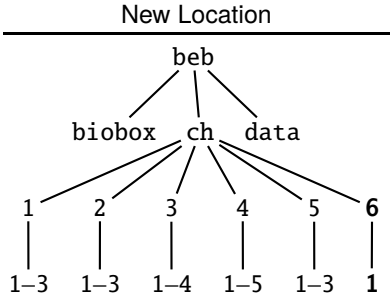
Fig. 5.18 The *Hominidae* tree with one bootstrap value for the (Homo, Pan) clade

Chapter 6
Evolution within Populations

Distinct species originate from differences between members of one species. Over time these differences can lead to the formation of distinct populations, which might eventually become unable to interbreed. One species has split in two. The differences within species are the subject of population genetics. Here the descent of genes takes center stage. This is often modeled using yet another tree, the coalescent.

6.1 Descent from One or Two Parents

We all have two parents, but each of our chromosomes only comes from one of them. Men even know which parent gave which sex chromosome, the father the Y, the mother the X. Here we explore the ramifications of individual descent from two parents and genetic descent from one parent.



6.1 Can you make the new directory for this section and change into it?

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_6

Two Parents

New Terms		
`drag`	partial ancestor	universal ancestor

6.2 Sexual organisms like us usually have two parents, four grandparents, and eight great-grandparents. How many great-great-great-grandparents do you have?

6.3 How many ancestors did you have 33 generations back? How does that compare to the world population 33 generations back of roughly 400 million people[1]?

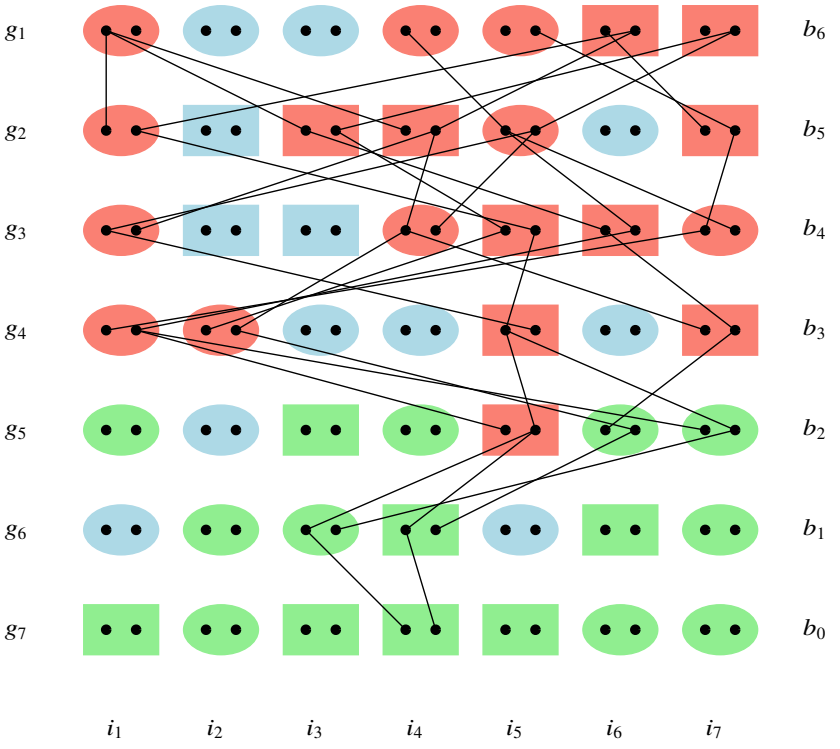


Fig. 6.1 Simulation of the ancestors of a single individual in a population of seven individuals. Ellipses and boxes indicate two sexes, the dots two genes; g_i stands for generation i, b_j for j generations back

[1] `https://en.wikipedia.org/wiki/World_population`

6.4 Fig. 6.1 shows a simulation of the ancestry of a single individual, i_4. The population consists of seven individuals and their lines of descent that end in i_4 traced over seven generations from g_1 at the top to g_7 at the bottom. When we go back one step from the present in b_0 to b_1, we find two parents of distinct shape, a mother and a father. This figure was generated using a program for drawing genealogies, `drag`. Can you draw your own version of this figure?

6.5 So we can fill in the first line of a table of observed and expected ancestors of individual i_4 moving back in time.

Back	Observed	Expected
b_1	2	2
...		

Can you fill in the rest?

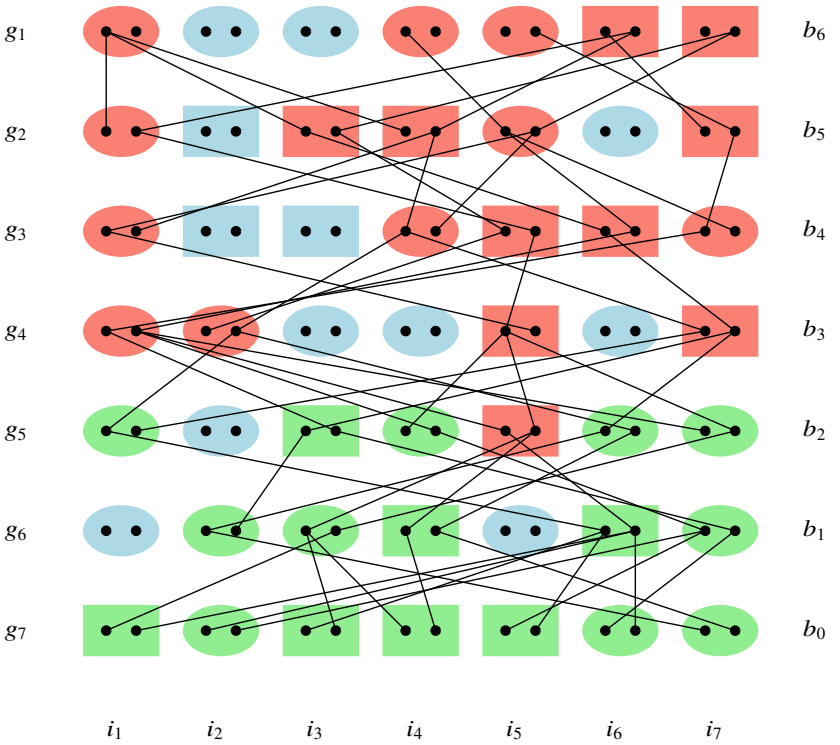


Fig. 6.2 Same as Fig. 6.1 but with all lines of descent included

6.6 Fig. 6.2 shows the same simulation as Fig. 6.1, but this time all lines of descent are included, not only those leading to i_4. To get such a pair of figures, use the same

seed for the random number generator between runs of `drag`. Can you draw your own pair of matching figures?

6.7 In Fig. 6.2 we can walk from any individual forward in time to visit all its descendants. By doing so, can you explain the color coding of green, blue, and red individuals?

6.8 As we go back in time in Fig. 6.2, partial ancestors go extinct leaving only universal ancestors and non-ancestors [37]. Let's concentrate on the point in time where partial ancestors vanish. Do you notice anything when you compare Fig. 6.1 and Fig. 6.2?

6.9 As you go back further in time, do the partial ancestors reappear? Can you explain your observation?

6.10 `drag` also has a statistical mode without graphics output, `-a`. So we can simulate the time to the first universal ancestor and the time to the disappearance of partial ancestors.

⟨*cli*⟩+≡
```
drag -a
```

If one of the results is zero, there weren't enough generations in the simulation to reach that point. How long does it take until the first universal ancestor appears in a population of size 1000? How variable is this number between runs?

6.11 How many generations does it take until the first universal ancestor appears in a population of size 2000?

6.12 What is the average number of generations until partial ancestors disappear from a population of 1000?

6.13 The expected number of generations until the disappearance of partial ancestors in a population size N is $1.77 \log_2(N)$ [37]. For $N = 1000$ this is 17.6, which fits our simulation result of 17.8 quite well.

⟨*cli*⟩+≡
```
echo '1.77 * l(1000) / l(2)' | bc -l
```

Let's explore the relationship between population size and no partial ancestors a bit more systematically through the script `nopanc.sh`.

Prog. 6.1 (`nopanc.sh`)

⟨*nopanc.sh*⟩≡
```
# Usage: bash nopanc.sh <N> <iterations>
for a in $(seq $2)
do
    ⟨Sum number of generations, Prog. 6.1⟩
done
o=$(echo "$sum / $2" | bc -l)
e=$(echo "1.77 * l($1) / l(2)" | bc -l)
printf "%d %.2f %s\n" $1 $o obs
printf "%d %.2f %s\n" $1 $e exp
```

Can you sum the number of generations?

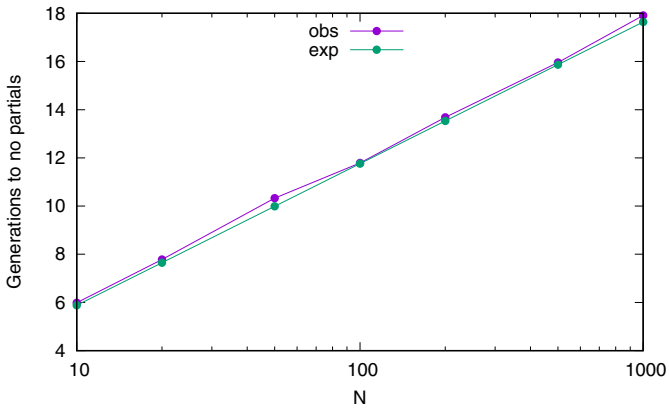


Fig. 6.3 Time to no partial ancestors as a function of population size, N; the observed values (*obs*) are averages of 100 simulation runs

6.14 Fig. 6.3 shows the time to the loss of partial ancestors as a function of population size. The observed and expected curves are quite close. Can you generate your own version of this figure?

One Parent

New Terms		
`drawf`	waiting time	Wright-Fisher model
most recent common ancestor		

6.15 In contrast to individuals, who have two parents, genes only have one. Fig. 6.4 shows a third version of Fig. 6.1, where the lines of descent are restricted to those of the two genes in individual i_4. Can you use `drag` to generate your own version of this figure?

6.16 In Fig. 6.4 the two genes find their most recent common ancestor two steps back. We say *most recent*, because all nodes on the genealogy further back are also common ancestors, but not most recent. In our example, the most recent common ancestor is also the first universal ancestor to appear. Is that always the case?

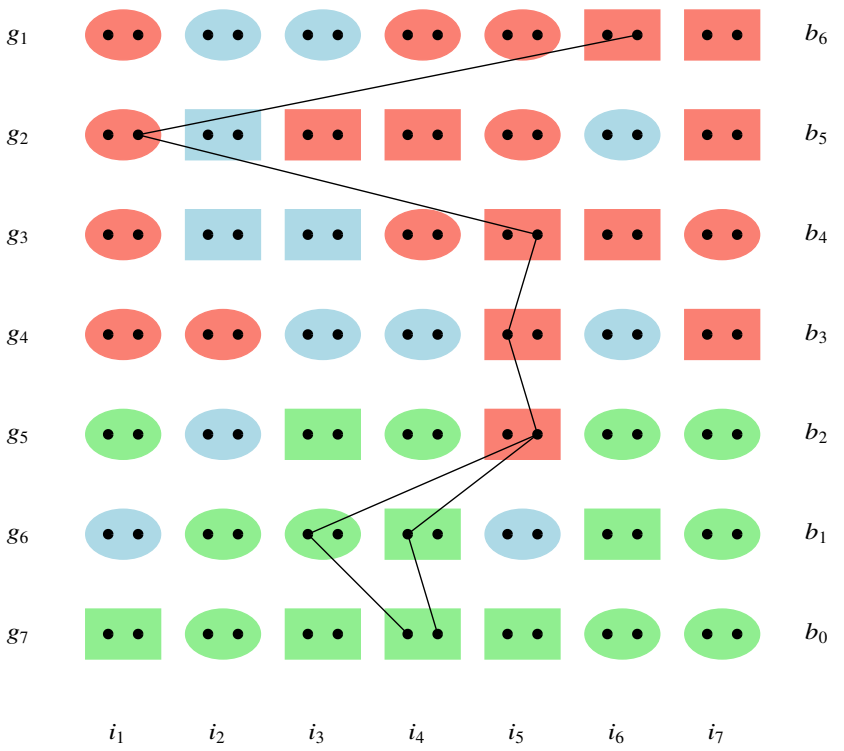


Fig. 6.4 Tracing the genes of individual i_4

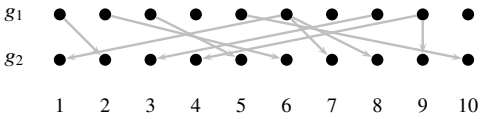


Fig. 6.5 Two generations of a population size 10 under the Wright-Fisher model

6.17 Populations are often modeled as consisting of genes without reference to individuals or gender. This abstract model of a population is called the Wright-Fisher model in honor of two founding figures of population genetics, Ronald A. Fisher (1890–1962) and Sewall Wright (1889–1988). Fig. 6.5 shows two generations in the evolution of 10 genes under the Wright-Fisher model. To get from one generation to the next, ancestors are simply picked at random, as indicated by the arrows. To see how this works, we can generate random numbers between 1 and 10.

⟨*cli*⟩+≡

```
echo "$RANDOM % 10 + 1" | bc
```

If you are on `zsh`—most likely under macOS—this won't work, but here's a workaround.

⟨*cli*⟩+≡

```
v="$RANDOM % 10 + 1"; echo $v | bc
```

Can you generate ten random numbers between 1 and 10 to extend Fig. 6.5 for one more generation?

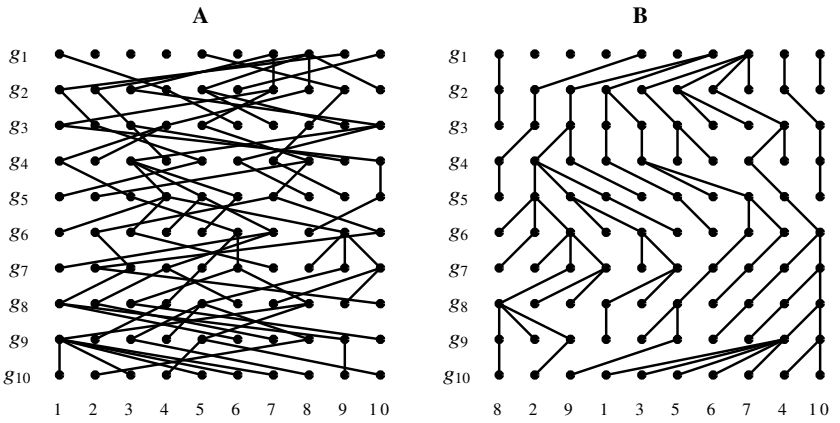


Fig. 6.6 Tangled (**A**) and untangled (**B**) versions of the same Wright-Fisher population

6.18 Fig. 6.6A depicts ten generations of a Wright-Fisher population consisting of ten genes. It was generated with a program for drawing Wright-Fisher populations, `drawf`. Can you generate your own version of this tangled image?

6.19 With all the crisscrossing ancestral lines connecting the genes between generations, it is a bit hard to see what is going on. `drawf` can also draw untangled lines of descent. Fig. 6.6B shows the untangled version of Fig. 6.6A. This makes it much simpler to trace ancestry back in time. Can you draw your own pair of tangled/untangled figures?

6.20 Does Fig. 6.6 contain a common ancestor of all genes?

6.21 The program `drawf` can mark the common ancestor of the entire population (`-m`). With a population size of $N = 10$, how many generations are required to be reasonably certain that a common ancestor is found?

6.22 When going one generation back in time, the number of lineages that eventually end up in the common ancestor either stays the same, or decreases. For example, in Fig. 6.7 that number goes from originally ten in b_0 to six in b_1. You can think of each gene (dot) in Fig. 6.7 as randomly picking its ancestor in the preceding generation. Occasionally two genes pick the same ancestor. What is the probability of this occurring?

Fig. 6.7 Two generations of a population under the Wright-Fisher model consisting of ten genes

6.23 If two genes pick an identical ancestor, their lineages fuse and the number of ancestral lineages is reduced by one. Let's write a program for picking random ancestors, `panc.awk`. It takes as arguments a seed for the random number generator and a population size.

Prog. 6.2 (`panc.awk`)

⟨*panc.awk*⟩≡
```
BEGIN {
  ⟨Set usage, Prog. 6.2⟩
  ⟨Pick ancestors, Prog. 6.2⟩
  ⟨Print ancestors, Prog. 6.2⟩
}
```

Can you set the usage?

6.24 Ancestors are random integers between 1 and N. We draw N of them and store them.

⟨*Pick ancestors, Prog. 6.2*⟩≡
```
srand(seed)
for (i = 0; i < N; i++) {
  r = int(rand() * N + 1)
  anc[r] = 1
}
```

Can you print the ancestors we just generated?

6.25 Let's run `panc.awk` ten times in a loop and print the number of distinct ancestors at the end of the output.

⟨*cli*⟩+≡
```
for a in $(seq 10)
do
    awk -f panc.awk -v N=10 -v seed=$RANDOM
done | awk '{print $0, NF}'
```

Do you get any runs with ten distinct ancestors?

6.26 We've seen that the probability of two genes having the same ancestor in the previous generation is $1/N$. So the probability of two genes *not* having a common ancestor is $1 - 1/N$. The probability of three genes not having a common ancestor is

$$\left(1-\frac{1}{N}\right)\left(1-\frac{2}{N}\right)$$

and hence the probability of N genes not having the same ancestor is

$$P_0(N)=\left(1-\frac{1}{N}\right)\left(1-\frac{2}{N}\right)\ldots\left(1-\frac{N-1}{N}\right). \tag{6.1}$$

Let's write the program `pn.awk` to compute $P_0(N)$.

⟨*pn.awk*⟩≡

```
BEGIN {
  if (!N) {
    print "Usage: awk -f pn.awk -v N=<N>"
    exit
  }
  pn = 1
  for (i = 1; i < N; i++)
    pn *= (1 - (N - i) / N)
  print pn
}
```

What is $P_0(10)$?

6.27 Our program `panc.awk` simulates one generation of ancestor picking. Can you use it to simulate $P_0(10)$?

Fig. 6.8 The Wright-Fisher population of Fig. 6.7 traced back one more generation

6.28 We are interested in the most recent common ancestor of a group of genes. Fig. 6.8 shows two steps on our way to the most recent common ancestor rather than just the single step in Fig. 6.7. The number of ancestral lines in green now declines from six to three. To model this situation, we distinguish the number of active lineages, n, from the population size, N. The calculation of the probability of no ancestor is now a function of the population size N and the sample size n, but the shape of equation (6.1) doesn't change

$$P_0(N,n)=\left(1-\frac{1}{N}\right)\left(1-\frac{2}{N}\right)\ldots\left(1-\frac{n-1}{N}\right).$$

Since N is usually large, we can ignore terms with N^{-2} or smaller to get

$$P_0(N,n) \approx 1 - \frac{1}{N} - \frac{2}{N} - \ldots - \frac{n-1}{N}.$$

The complement of this is the probability of an ancestor event,

$$P_\mathrm{a}(N,n) = \frac{1+2+\ldots+n-1}{N} = \frac{n(n-1)}{2N}.$$

What is the probability of observing an ancestor event if $N = 1000$ and $n = 10$?

6.29 If some event has probability p, then the expected waiting time until the event occurs is $1/p$. What is the expected number of generations we have to wait in a population of size $N = 1000$ for the number of active lineages to decline from $n = 10$ to $n = 9$?

6.30 Let's simulate the time to an ancestor event under the Wright-Fisher model. And since we don't have a preference for a particular ancestor event, let's just simulate all of them until we reach the most recent common ancestor. So our program, `tmrca.awk`, takes as input a population size, N, a sample size, n, and a seed for the random number generator. Given that we've got these variables set, we seed the random number generator and print the sample size as a function of the zeroth generation. Now we iterate until there is only one lineage left. In the loop we count the generations, pick n lineages out of N, determine the new number of lineages we got, and report any change in the number of lineages. Then the new number of lineages becomes the current number of lineages.

Prog. 6.3 (`tmrca.awk`)

⟨*tmrca.awk*⟩≡
```
BEGIN {
  ⟨Set usage, Prog. 6.3⟩
  srand(seed)
  print 0, n
  while (n > 1) {
    g++
    ⟨Pick n lineages out of N, Prog. 6.3⟩
    ⟨Count lineages, Prog. 6.3⟩
    ⟨Report change in lineages, Prog. 6.3⟩
    n = new_n
  }
}
```

In the usage we make sure the user has set N, n, and the seed for the random number generator. Can you do that?

6.31 We pick n lineages out of a population of N genes.

⟨*Pick n lineages out of N, Prog. 6.3*⟩≡
```
for (i = 0; i < n; i++) {
  r = int(rand() * N)
```

```
    lineages[r] = 1
  }
```

Can you count the distinct lineages we picked?

6.32 We are done with the lineages, so we delete them again.

⟨*Count lineages, Prog. 6.3*⟩+≡

```
  for (lineage in lineages)
    delete lineages[lineage]
```

Since the waiting time to ancestor events scales with the population size, it is usually expressed in units of N generations. Can you check if there's been a change in the number of lineages and print a message if so?

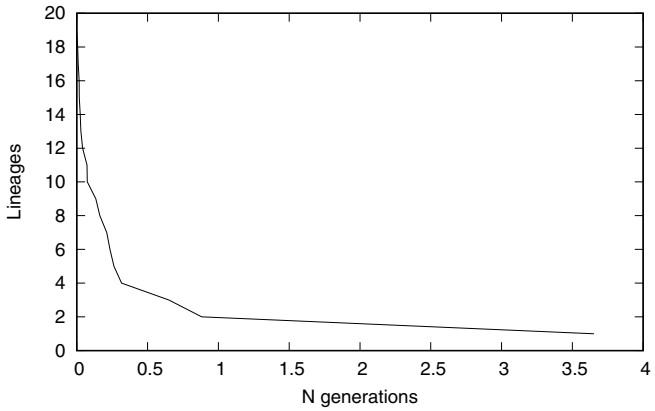


Fig. 6.9 One simulation of the number of lineages as a function of N generations

6.33 Fig. 6.9 shows the number of lineages as a function of N generations for an initial sample of $n = 20$ and a population size of $N = 10^4$. We simulated this with `tmrca.awk`. Can you generate you own version of Fig. 6.9?

6.34 The expected time to the most recent common ancestor measured in N generations is [40, p. 76]:

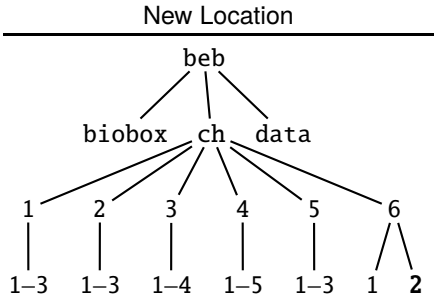
$$E\left\{T_{\mathrm{MRCA}}(n)\right\} = 2\left(1 - \frac{1}{n}\right). \tag{6.2}$$

What is the expected T_{MRCA} for $n = 2$ and $n \to \infty$?

6.35 According to equation (6.2), the expected time to the most recent common ancestor for a sample of size $n = 10$ is 0.9. What is the average time to the most recent common ancestor in 1000 simulation runs with $N = 10^4$?

6.2 The Coalescent

A sample of genes under the Wright-Fisher model is connected by lines of descent that we like to trace to their most recent common ancestor. These lines of descent form a tree, the root of which is the most recent common ancestor. This tree is called *coalescent*, and we explore its construction and use in this section.



6.36 Can you make the directory for this section and change into it?

New Terms		
coalescent	Poisson distribution	VCF format
`curl`	population mutation rate	Watterson's equation
exponential distribution	segregating sites	`watterson`
`ms`		

6.37 The coalescent is based on the Wright-Fisher model. So to get us started, Fig. 6.10A shows a population of ten genes evolving for ten generations under the Wright-Fisher model. We've already drawn such figures with the program `drawf`. Can you do it again?

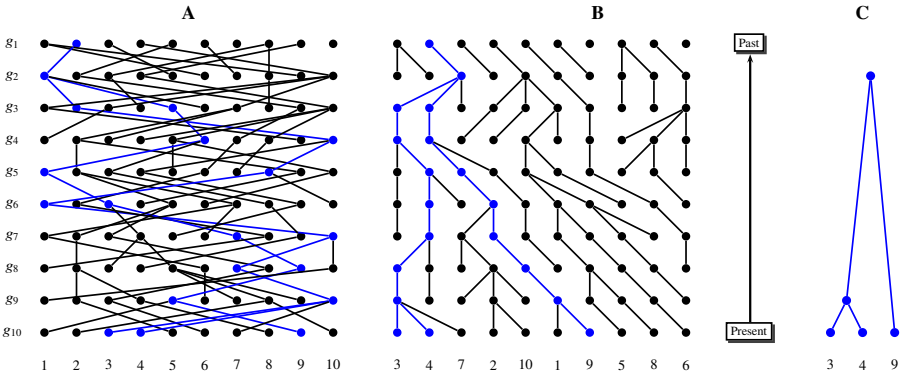


Fig. 6.10 A population under the Wright-Fisher model (**A**), its untangled version (**B**), and the coalescent for three of its lineages (**C**)

6.38 If you look carefully at Fig. 6.10A, you should find three blue lineages. These are more apparent when we untangle the lines of descent in Fig. 6.10B. Investigations of real genes are usually restricted to small samples, the three blue genes might be such a sample. Their lines of descent form a tree. If we just concentrate on this tree, we can further reduce it to the nodes where two lines of descent collide as we move from the present into the past. A different way of looking at such a collision is to say that two lines of descent merge, or "coalesce", into one. The collection of such coalescence events is depicted in Fig. 6.10C and is called the "coalescent". It describes the descent of a sample of genes evolving under the Wright-Fisher model from the present to the most recent common ancestor. Can you draw the coalescent for genes 5, 6, and 8?

6.39 As we've said, coalescents are random trees or genealogies. To construct a coalescent, we represent it as an array, where nodes 1–4 in green are leaves, nodes 5–7 in black are inner nodes.

Index	1	2	3	4	5	6	7
Node	1	2	3	4	5	6	7

Draw an example tree with these nodes. What is the sample size, n, for this coalescent?

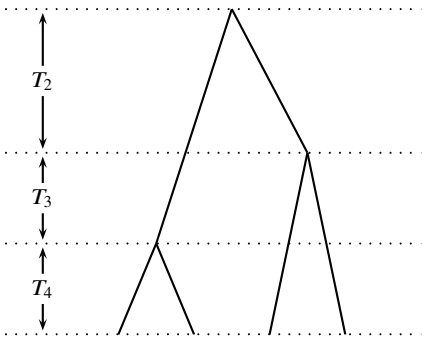


Fig. 6.11 Coalescent with time intervals T_i indicating how long the tree consists of i lines

6.40 All nodes of a coalescent are annotated with coalescence times. The leaves are in the present, which is time zero:

Index	1	2	3	4	5	6	7
Node	1	2	3	4	5	6	7
Time	0	0	0	0			

To compute the times of the three inner nodes, we divide the coalescent into intervals T_i, where i is the number of lines of descent in the tree during that time (Fig. 6.11). Where would T_1 be in this graph, and why is it not shown?

6.41 To compute T_i, start from the probability of a coalescence event we've already seen,

$$P_{\mathrm{a}}(N, n) = \frac{n(n-1)}{2N}.$$

This means the time to the next coalescence event is an exponentially distributed random variable with mean

$$1/P_a(N, n) = 2N/n/(n-1).$$

Since we usually only know the sample size n and not the population size N, we measure time in units of N generations. So the mean of T_i becomes $\lambda = 2/i/(i-1)$. If r is a uniformly distributed random variable, then an exponentially distributed random variable with mean λ is calculated as $t = -\lambda \log(1-r)$. Let's write a program, `ti.awk`, that calculates T_i. It takes as input the i and a seed for the random number generator. After seeding, we calculate λ and then the random time interval T_i. Can you set the usage?

Prog. 6.4 (`ti.awk`)

⟨*ti.awk*⟩≡
```
BEGIN {
  ⟨Set usage, Prog. 6.4⟩
  srand(seed)
  la = 2 / i / (i-1)
  t = -la * log(1 - rand())
  print t
}
```

6.42 Can you fill in the three missing node times in our tree construction table?

6.43 We now write a program to calculate coalescence times, `coat.awk`. Its central statement should look familiar from `ti.awk`, but this time we wrap the computation of T_i in a loop and sum the times in the variable `t`.

Prog. 6.5 (`coat.awk`)

⟨*coat.awk*⟩≡
```
BEGIN {
  ⟨Set usage, Prog. 6.5⟩
  srand(seed)
  t = 0
  for (i = n; i >= 2; i--) {
    ⟨Calculate coalescence time, Prog. 6.5⟩
  }
}
```

`coat.awk` takes as input the sample size, n, and a seed for the random number generator. Can you set its usage?

6.44 Can you calculate the coalescence time and print it?

6.45 What is the average time to the most recent common ancestor from 100 iterations with $n = 5$? You can compare your result to equation (6.2).

6.46 Having seen how to assign times to nodes, we next construct a random tree from them—the coalescent. This requires shuffling a set of nodes. Consider the five nodes 1, 2, 3, 4, 5; after shuffling, they might have the order 4, 5, 1, 2, 3. Can you think of an effective way to shuffle the numbers 1, 2, ..., 5 without using the computer?

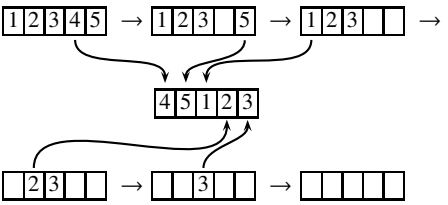


Fig. 6.12 Shuffling the numbers 1, 2, ..., 5 into 4, 5, 1, 2, 3

6.47 In a computer, we keep the five nodes to be shuffled in an array. Then we pick random nodes from that array as shown in Fig. 6.12. By not replacing the nodes we've picked, we get a shuffled version of the original array. Unfortunately, this isn't very efficient. Can you see why?

6.48 Let's say we have an array, a, of n nodes. We can shuffle them efficiently by combining picking nodes with swapping nodes:

1. Pick a random number r between 1 and n
2. Swap $a[r]$ and $a[n]$
3. Reduce n by 1
4. Repeat

This method of sampling without replacement depends on distinguishing between the value of an index, i, and the value of an array, a, at that index, $a[i]$. Can you shuffle 1, 2, 3, 4, 5 using the random indexes 1, 3, 1, 2?

6.49 To construct a coalescent we need to a apply the shuffling procedure to the array of leaves and internal nodes. For this we add three auxiliary rows to our table so we can transparently overwrite node labels. We also need rows for the first and second child of the internal nodes. Table 6.1 shows the augmented table for constructing the coalescent. We begin with the first internal node, 5, and pick a random first child among the four leaves.

⟨*cli*⟩+≡

```
echo "$RANDOM % 4 + 1" | bc
```

Table 6.1 Table for constructing the coalescent

Index	1 2 3 4 5 6 7
Node	1 2 3 4 5 6 7
Child1	
Child2	
Time	0 0 0 0

Say, we pick a 1. Can you make the corresponding entry in Table 6.1?

6.50 Having assigned the first child, we make the move from shuffling, that is, we replace node 1 by the rightmost leaf in this round, 4. Can you enter this in Table 6.1?

6.51 We draw another random number, but this time only the first three leaves are candidates

⟨*cli*⟩+≡
```
echo "$RANDOM % 3 + 1" | bc
```

Say, this is 2; then the second child of 5 is node 2, which is in turn replaced by node 5. So in the next round, node 5 is among the children to chose from. Can you enter these two changes in Table 6.1?

Algorithm 2 Construct coalescent

Require: n {sample size}
Require: tree {Array: n leaves, $n-1$ internal nodes}
Ensure: Tree topology
1: **for** $i \leftarrow n$ **to** 2 **do**
2: $p \leftarrow 2 \times n - i + 1$ {Parent}
3: $c \leftarrow i \times \text{ran}() + 1$ {Draw first child, $1 \le c \le i$}
4: tree$[p]$.child1 $\leftarrow$ tree$[c]$
5: tree$[c] \leftarrow$ tree$[i]$ {Replace first child}
6: $c \leftarrow (i-1) \times \text{ran}() + 1$ {Draw second child, $1 \le c \le i-1$}
7: tree$[p]$.child2 $\leftarrow$ tree$[c]$
8: tree$[c] \leftarrow$ tree$[p]$ {Replace second child by parent}
9: **end for**

6.52 As we've seen, in each round of coalescent construction the first child of some node v is replaced by the rightmost child available in that round, and the second child is replaced by v. Algorithm 2 summarizes these steps. Say, the remaining child indexes drawn are all 1. Can you finish constructing the coalescent?

6.53 Can you plot the coalescent we've just constructed?

6.54 Let's write a program for automatically picking the children, `pick.awk`

Prog. 6.6 (`pick.awk`)

⟨*pick.awk*⟩≡
```
BEGIN {
  ⟨Set usage, Prog. 6.6⟩
  srand(seed)
  for (i = n; i >= 2; i--) {
        ⟨Pick children, Prog. 6.6⟩
  }
}
```

`pick.awk` takes as input the sample size, n, and a seed for the random number generator. Can you set the program's usage?

6.55 Now we pick the children and print them in a table.

⟨*Pick children, Prog. 6.6*⟩≡
```
print "# Pa\tC1\tC2"
for (i = n; i >= 2; i--) {
  c1 = int(rand() * i + 1)
  c2 = int(rand() * (i-1) + 1)
  no = 2 * n - i + 1
  printf "%d\t%d\t%d\n", c1, c2, no
}
```

Can you use `pick.awk` together with `coat.awk` to generate another coalescent for $n = 4$?

6.56 At this stage, we have coalescence times and a branching order. To achieve biological relevance, we still need mutations. Mutations are rare events and the probabilities of rare events are modeled by the Poisson distribution. So we generate mutations as Poisson-distributed random variables for each branch length t with expectation

$$\lambda = t\theta/2,$$

where $\theta = 2N\mu$ and μ is the number of mutations per generation for the region considered. The program `rpois` generates Poisson-distributed random numbers given a mean. Let $\theta = 10$; can you use `rpois` to generate a random number of mutations for $t = 0.5$?

6.57 Fig. 6.13 shows an example coalescent. Can you draw for each of its six branches a random number of mutations ($\theta = 10$)?

6.58 The number of mutations across all branches of a coalescent is given by Watterson's equation [42],

$$E\{S\} = \theta \sum_{i=1}^{n-1} \frac{1}{i}, \tag{6.3}$$

where $E\{S\}$ is the expected number of mutations, or *segregating sites*. This equation is implemented in the program `watterson`. What is the expected number of

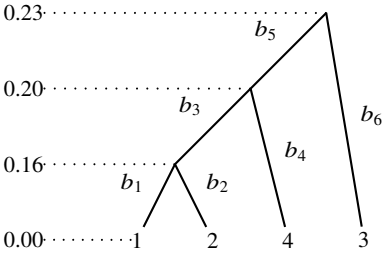


Fig. 6.13 Example coalescent with coalescence times and branches labeled

segregating sites for $n = 4$ and $\theta = 10$? How does this compare to the number of mutations you just simulated?

6.59 The program `ms` [20] is a popular coalescent simulator and we can use it to generate one sample of size 4 with $\theta = 10$ and tree printing.

⟨*cli*⟩+≡

```
ms 4 1 -t 10 -T
```

```
ms 4 1 -t 10 -T
60977 30522 51696

//
(2:0.209,(3:0.042,(1:0.030,4:0.030):0.012):0.167);
segsites: 6
positions: 0.0675 0.1627 0.2495 0.2952 0.3512 0.4482
010111
101000
010111
010111
```

Row by row this output means:

1. Repetition of the command
2. Initialization of the random number generator
3. Blank
4. Start of the first sample
5. Coalescent tree in Newick format
6. Number of segregating sites (mutations): 6
7. Positions of the mutations along the interval $(0, 1)$
8. Start of four haplotypes, each six segregating sites long, 0 indicates ancestral state, 1 mutant

Can you plot ten coalescents with $n = 4$ and $\theta = 10$? What happens when you change θ?

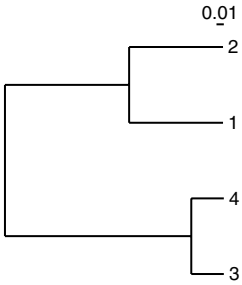


Fig. 6.14 Coalescent for a sample of size 4

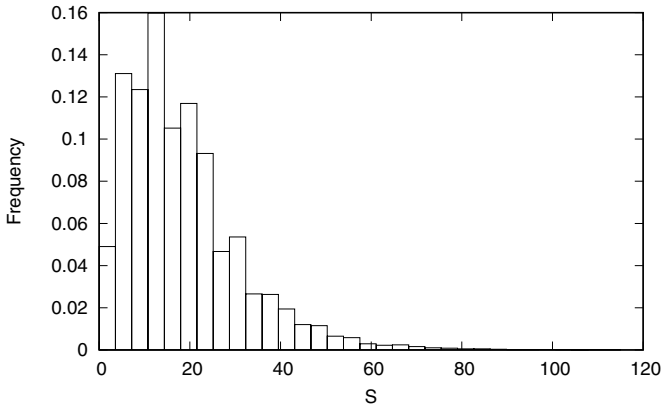


Fig. 6.15 Distribution of the number of segregating sites in 10^4 simulated samples of size 4 with $\theta = 10$

6.60 Fig. 6.14 shows a coalescent for four homologous genes. What does the scale bar mean?

6.61 Fig. 6.15 shows a histogram of the number of segregating sites for 10^4 simulated samples with $n = 4$ and $\theta = 10$. Can you generate your own version of Fig. 6.15?

6.62 Say, you observe 50 segregating sites in a sample of four genes and the population mutation rate is $\theta = 10$. We've already calculated with `watterson` that the expected number of segregating sites for this combination of sample size and mutation rate is 18.3. So you might think, observing 50 segregating sites is a lot compared to the expected 18.3. Can you test the null hypothesis that the observed 50 segregating sites arose under the Wright-Fisher model?

6.63 Next, we'd like to investigate some real mutations in mice. A great source of mouse mutation data is provided by the mouse genomics group at the Wellcome Sanger Institute. We can list the files in the Sanger mouse data repository using

`curl`, a program to transfer URLs. Please note that the slash at the end of the URL is significant.

⟨*cli*⟩+≡
```
url=https://ftp.ebi.ac.uk/pub/databases/mousegenomes/
curl  $url | less
```

What do you see?

6.64 To suppress progress messages from `curl`, we run it silently.

⟨*cli*⟩+≡
```
curl -s $url | less
```

Mutation database releases have names starting with `REL`. Can you count the number of releases posted by the Sanger Institute?

6.65 As of this writing, the directory for the current mutation data is

`REL-1505-SNPs_Indels`

and we adjust our URL accordingly.

⟨*cli*⟩+≡
```
url=${url}REL-1505-SNPs_Indels/
```

Can you list the contents of this directory?

6.66 Mutation data is stored in "variant call format", or VCF, files. Of the two types of mutations, single nucleotide polymorphisms (SNPs) and indels, we concentrate on the SNPs, as they are the only type of mutation modeled by the coalescent. VCF files can be quite large. On the right hand side of the listing returned by `curl` you find the file size in bytes. Our SNP file is called

`mgp.v5.merged.snps_all.dbSNP142.vcf.gz`

How large is it?

6.67 We now concentrate on our target SNP file, so we adjust the URL one more time for convenient handling.

⟨*cli*⟩+≡
```
url=${url}mgp.v5.merged.snps_all.dbSNP142.vcf.gz
```

The program `tabix` allows us to query VCF files over the net without downloading them. Can you list the chromosome names contained in our VCF file (`man`)?

6.68 The `tabix` query resulted in the index file `*.tbi` in your current directory. How large is that compared to the file it indexes (`ls -l`)?

6.69 We'd like to work on the shortest mouse chromosome. A VCF file is described in its header and ours contains contig, or chromosome, lengths. Now, by convention chromosomes are ordered by descending length, so chromosome 1 is the longest followed by 2, and so on. Mice have 19 autosomes, which would make chromosome 19 the shortest mouse chromosome. How long is chromosome 19, and is it the shortest autosome?

6.70 The last line of the header section is the header of the subsequent SNP table. Its first nine columns are mandatory. Can you list them and guess what they mean?

6.71 The columns beyond the first nine describe the samples that make up the data set. Can you count the samples in our data set?

6.72 Let's look at the first two SNPs on chromosome 19. What are their positions and alleles?

⟨*cli*⟩+≡
```
tabix $url 19 | head -n 2
```

6.73 `tabix` can be used to download slices of SNP data, for example the 1 kb region 50,900,001–50,901,000 on chromosome 19.

⟨*cli*⟩+≡
```
tabix  $url 19:50,900,001-50,901,000
```

Notice that it's ok to use commas in numbers, which makes them more legible. How many SNPs does this region contain?

6.74 We would like to compare the number of SNPs we just observed to an expectation. For this we need two things, sample size, n, and the population mutation rate, θ. What is the sample size?

6.75 To estimate θ, we count the SNPs on chromosome 19.

⟨*cli*⟩+≡
```
tabix $url 19 | wc -l
```

This took us 2.5 minutes, working with remote data does have its disadvantages. How many SNPs are on chromosome 19?

6.76 Can you estimate θ per site using Watterson's equation (`watterson`)?

6.77 How many SNPs are expected in our 1 kb region?

6.78 Is the difference between observed and expected SNPs significant?

Chapter 7
Interrogating and Storing Data

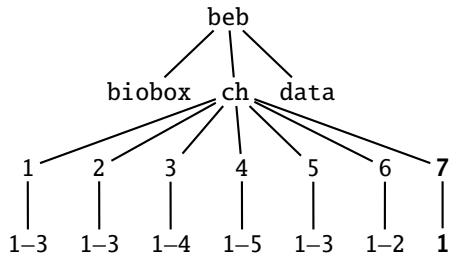
When analyzing data, statistics is never far off. In fact, statistics *is* a structured approach to data analysis. Such a structured approach becomes especially important when analyzing the multiple simultaneous experiments that are characteristic of genomics. One aspect of data analysis rarely mentioned is that we can only compute with data stored in a suitable format. The emphasis on human-readable text in Unix is already very helpful, but as we shall see, next to text, tables are the second great idea in digital data storage.

7.1 Statistics

Compared to other branches of mathematics, statistics is a young discipline. Take Student's *t* test, which assesses the hypothesis that two small samples are drawn from the same population by comparing their means. It was published in 1908 by William S. Gosset (1876–1937), who used "Student" as a pseudonym for his work in statistics [39]. Gosset trained as a chemist and worked all his life in the management of the Guinness brewery, first in Dublin and later in London. A central aim of the company leadership at the time was to make brewing scientific. This required experimentation on such things as the effect of the resin content of hops on beer quality. However, once the relevant measurements had been made, a structured approach to their interpretation was also needed; how large a difference in hop resin content made a significant difference to the shelf life of stout? Today we call the investigation of such questions "statistics" and Gosset was one of its pioneers. Rather than the resin content of hops, we take as our example an investigation of the effect of acute amebic colitis on gene expression in humans [5].

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_7

New Location



7.1 Can you create the directory for this section and change into it?

Single Experiments

New Terms

`efetch`	Monte Carlo method	`testMeans`
`esearch`	Student's *t* test	

7.2 The study on acute amebic colitis we are using as our example was published with document identifier (doi) [36]

`10.1016/j.parint.2011.04.005`

We can search for this doi in Pubmed, a database of medical literature. Pubmed is part of the Entrez database collection, which can be accessed through programs of the edirect software suite. Here we use `esearch` to search for the document and `efetch` to fetch it in XML format.

⟨*cli*⟩+≡

```
esearch -db pubmed -query "10.1016/j.parint.2011.04.005" |
    efetch -format xml | less
```

How many patients were included in this study?

7.3 The expression data collected in the colitis study was deposited in the Geo database, from where we can download and unpack it.

⟨*cli*⟩+≡

```
url="ftp://ftp.ncbi.nlm.nih.gov/geo/datasets/"
url=$url"GDS4nnn/GDS4374/soft/GDS4374.soft.gz"
curl $url -o GDS4374.soft.gz
gunzip GDS4374.soft.gz
```

What does the file contain (`less`)?

7.4 The colitis study is based on measurements from day 1 of the infection and from day 60, long after recovery through treatment with an antibiotic. Can you find out which columns correspond to which of these two categories?

7.5 To simplify the data analysis, we split the data into two files, `d1.txt` and `d60.txt` for day 1 and day 60. Each file contains just an identifier and eight columns of the expression data. The identifier is constructed by concatenating the probe name and the locus name in the first two columns of the original data via a dollar.

⟨*cli*⟩+≡

```
awk '!/^[!^#]/' GDS4374.soft |
    tail -n +2 |
    awk '{printf "%s$%s", $1, $2;
    for(i=3;i<=10;i++)printf "\t%s",
    $i; printf "\n"}' > d1.txt
```

Can you construct `d60.txt`?

7.6 What do you observe when you look at the head of `d1.txt`?

7.7 What do you observe when you look at the tail of `d1.txt`?

7.8 Each line in `d1.txt` and `d60.txt` corresponds to a distinct expression probe. How many expression probes were assayed?

7.9 All entries in `d1.txt` (and `d60.txt`) not called `chr*` or `control` are presumably genes. A gene may have more than one probe. How many distinct genes were assayed?

7.10 What are the genes with the largest number of probes?

7.11 We start by investigating a single gene, *ACKR2*, which stands for "atypical chemokine receptor 2". Chemokines are a critical part of the inflammation response and amebic colitis leads to inflammation of the intestinal mucosa. What are the expression levels of *ACKR2* on day 1 and day 60?

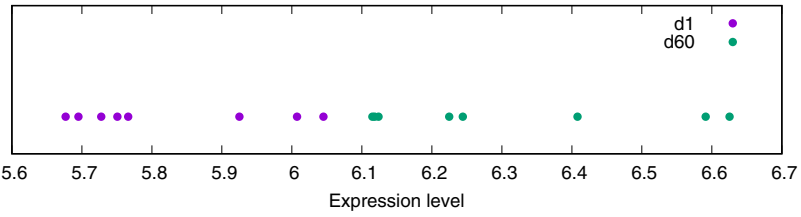


Fig. 7.1 The *ACKR2* expression levels on day 1 and day 60 plotted along the x axis

7.12 Fig. 7.1 shows the expression levels of *ACKR2* on day 1 and day 60 plotted along the x axis of a graph. Colitis appears to reduce *ACKR2* expression. We generated Fig. 7.1 with the help of `expr.sh`, which takes as input the name of a gene and writes its expression values ready for plotting.

Prog. 7.1 (`expr.sh`)

⟨*expr.sh*⟩≡

```
for a in d1 d60
do
    grep $1 ${a}.txt |
          tr '\t' '\n' |
          tail -n +2 |
          awk -v c=${a} '{print $1, 0, c}'
done
```

Can you reproduce Fig. 7.1?

7.13 We'd like to know whether the expression of *ACKR2* decreases significantly due to acute amebic colitis. A simple way to summarize our sets of eight measurements are their averages. What are the average expression levels of *ACKR2* on day 1 and day 60?

7.14 The expression values are given as $\log_2$. In other words, an expression value of 2 corresponds to $2^2 = 4$ units of expression, a value of 3 to $2^3 = 8$ units of expression. The fold change between these two expression values is $2^{3-2} = 2$. What is the fold change in average expression of *ACKR2* between day 1 and day 60?

7.15 The fold change we just determined might seem a bit small, but our question remains, is it significant? We answer this by testing whether the averages we just calculated are significantly different. The program `testMeans` implements Student's t test to answer this question. Do you think the two averages differ?

7.16 For Student's t test, t is defined as

$$t = \frac{m_1 - m_2}{s_p\sqrt{1/n_1 + 1/n_2}}, \tag{7.1}$$

where m_1 and m_2 are the sample means, n_1 and n_2 the sample sizes, and s_p the pooled standard deviation,

$$s_p = \sqrt{\frac{(n_1 - 1)v_1 + (n_2 - 1)v_2}{n_1 + n_2 - 2}},$$

where v_1 and v_2 are the sample variances. The null hypothesis tested by the t test is that $m_1 = m_2$, in which case $t = 0$. Can you recapitulate the t you just computed (`var`)?

7.17 The t test is based on the assumption that the variables are drawn from a normal distribution. This may or may not be the case. Can we test the means of *ACKR2* expression without making this assumption? A popular strategy for answering this question is to shuffle the expression values among day 1 and 60 and to recalculate their averages. To see how this works, we save the *ACKR2* expression values in the

file `ackr2.dat` using our script `expr.sh`. Then we generate 16 random numbers and save them in file `r.txt`. These numbers are used as prefixes of the expression values and sorted, which shuffles the expression values. Now we divide the data into two equal halves with `head` and `tail` and calculate the average of each half.

⟨*cli*⟩+≡
```
bash expr.sh ACKR2 > ackr2.txt
for a in $(seq 16); do echo $RANDOM; done > r.txt
for a in head tail
do
    paste r.txt ackr2.txt |
          sort -n |
          ${a} -n 8 |
          awk '{s+=$2}END{print s/8}'
done
```

What's the difference between the averages after shuffling compared to the original difference?

7.18 The shuffling test we just outlined belongs to the class of "Monte Carlo" methods. Can you explain their peculiar name?

7.19 To actually carry out a hypothesis test with our Monte Carlo method, we'd have to repeat the shuffling many times and calculate the frequency with which we find a difference between the two shuffled means that is at least as great as the difference between the original means. The program `testMeans` implements this Monte Carlo procedure when used with option `-m`. What is the *P*-value for *ACKR2* computed with Monte Carlo? How does this compare to the *P*-value with the *t* test formula?

7.20 What happens if you carry out the test with much fewer, say 100, iterations?

7.21 We've investigated a single gene, and we'd now like to apply our test to all 33,297 expression probes. Why might this be problematic?

Multiple Experiments

New Terms		
Benjamini-Hochberg correction	false discovery rate	false positive rate
Bonferroni correction	false negative rate	`simNorm`

7.22 The program `simNorm` generates samples drawn from a normal distribution. We use it to simulate two experiments with $m = 100$ samples of size $n = 8$ with arbitrary mean 12 and look at the top of one of them.

⟨*cli*⟩+≡
```
simNorm -i 100 -m 12 > exp1.txt
```

```
  simNorm -i 100 -m 12 > exp2.txt
  head exp1.txt

# ID    x_1    x_2    x_3    x_4    x_5    x_6    x_7    x_8
s_1     12     13     15.2   13     11.5   11.3   12.2   12.4
s_2     11.8   12.3   10.6   11.2   11.5   12.4   12.2   12.5
s_3     11.8   13.2   14.1   11.3   11.9   12.5   11     13
...
```

The files we just generated contain the values for sample 1, s_1, followed by the values for sample 2, s_2, and so on. We can think of the samples in `exp1.txt` as control, the samples in `exp2.txt` as experiment for genes x_1, x_2, and so on. Since we've generated the two data files from the same normal distribution, there can be no significant differences between them. Still, when we apply `testMeans` to the two files we find positive results. How many such positive values do you get if the cutoff value of P for calling a result positive is the customary $\alpha = 0.05$?

7.23 Pseudo-positive, or false-positive results occur with an expected frequency α. What is the observed false positive rate if we repeat the simulation with 10^4 experiments?

7.24 False-positive results occur more frequently the more tests we carry out. In fact, the probability of getting at least one false positive in m tests with threshold α is

$$f_{\mathrm{p}}(\alpha, m) = 1 - (1 - \alpha)^m.$$

What is $f_{\mathrm{p}}(0.05, 100)$?

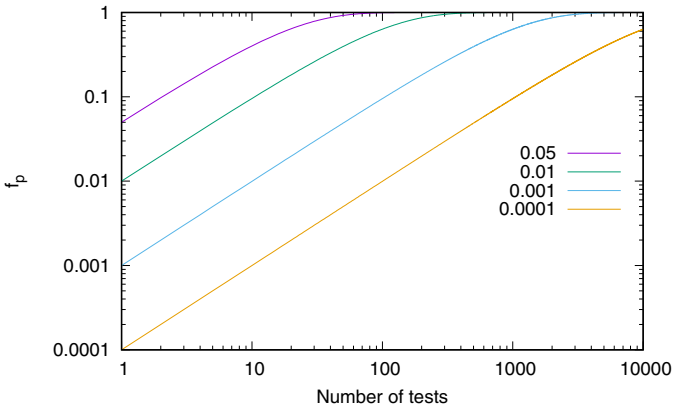


Fig. 7.2 The probability of obtaining at least one false-positive result, f_{p}, as a function of the number of hypothesis tests for α-values ranging from 0.05 to 10^{-4}

7.25 As illustrated in Fig. 7.2, the more tests we carry out, the greater the probability that we obtain at least one false-positive. Can you write a script `fp.sh` to reproduce Fig. 7.2?

7.26 In statistics the false positive rate is also known as the type I error. So far, we set this to $\alpha = 0.05$ per experiment. However, we can also regard the 10^4 experiments as a single unit. Then we can divide the original α by the number of tests carried out to assess the null hypothesis that the ensemble contains no sample with a significant difference. This division of α by the number of tests was proposed by Carlo Emilio Bonferroni (1862–1960) and is thus called *Bonferroni correction*. What is the false positive rate if we analyze our 10^4 experiments using the Bonferroni correction?

7.27 Now we simulate 10^4 experiments with different means, say, 12 and 11.

⟨*cli*⟩+≡

```
simNorm -i 10000 -m 12 > exp1.txt
simNorm -i 10000 -m 11 > exp2.txt
```

Since we have drawn the samples from distinct distributions, any P-value greater than α would indicate a false negative test. What is the false negative rate, β, if we leave $\alpha = 0.05$?

7.28 If the difference in means, also called the "effect size", is small, we need bigger samples to detect it. What happens to the false negative rate if you increase the sample size from 8 to 16? to 32?

7.29 The false negative rate is also known as the type II error. We've seen that the Bonferroni correction is used to minimize the type I error. What happens to the type II error if you re-analyze `exp1.txt` and `exp2.txt` with Bonferroni correction?

7.30 As we've seen, minimizing the type I error can lead to a large type II error. Hence the concept of false discovery rate, fdr, has been developed. The fdr is the fraction of false-positive results among the *rejected* hypotheses, rather than among all hypotheses. In order to set the fdr to some level δ, the original P-values are sorted in ascending order, $P_1 \leq P_2 \leq \ldots \leq P_m$; then a particular P-value at position j is significant if it is less than δ divided by the number of tests multiplied by the position, $P_j \leq \delta j/m$. This method is due to Yoav Benjamini and Yosef Hochberg and hence also known as the Benjamini-Hochberg correction [4]. What is the type II error, or β, with $\delta = 0.05$?

7.31 To explore the robustness of the Benjamini-Hochberg correction with respect to type I error, we return to a data set without any significant differences, but keep the large sample sizes for now.

⟨*cli*⟩+≡

```
simNorm -n 32 -i 10000 -m 12 > exp1.txt
simNorm -n 32 -i 10000 -m 12 > exp2.txt
```

What is the type I error in your simulation with $\alpha = 0.05$?

7.32 How large is the type I error with the Benjamini-Hochberg correction, $\delta = 0.05$?

7.33 What happens when you set δ to the more permissive value of 0.1?

7.34 Let's now return to the real expression data from amebic colitis. Benjamini-Hochberg corrections are often carried out with the relatively permissive threshold of $\delta = 0.1$. How many distinct loci are deemed significant under that parameter? Is *ACKR2* among them?

7.2 Relational Databases

In the late 1960's the British mathematician Edgar F. Codd (1923–2003) proposed a new model for storing and accessing data. This model, called the relational data model, has become the standard way of dealing with large data sets [7]. Originally used mainly in government and business, relational databases are now also ubiquitous in genomics [17]. In this section we learn how to construct and query relational databases.

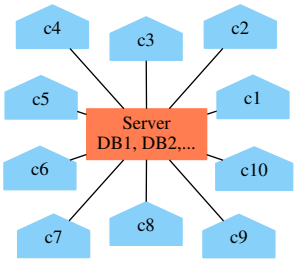
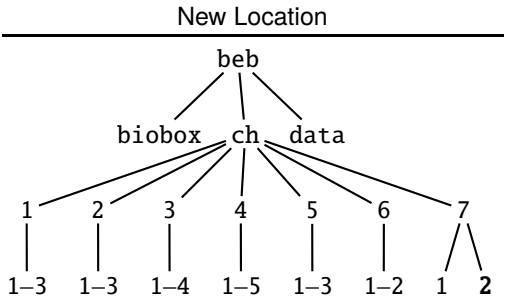


Fig. 7.3 A database server hosting several databases in red connected to ten clients in blue

There are a number of software systems available for doing this and they all implement the query language SQL. However, there are differences, and the most important distinction is between systems with a client-server structure and those without. Systems with a client-server structure, such as Oracle, Mysql, and Postgresql, are usually centered on a server hosting one or more databases (Fig. 7.3). A potentially large number of clients connects via the internet to this server. As an example, we look at the Ensembl database collection, which contains genome data on vertebrates and is managed by Mysql [3].

Server-client systems are powerful and can be challenging to construct and administer, as opposed to mere querying. For single-user databases there are also simpler

systems available, where the database is just a local file. As an example for this kind of system we experiment with the Sqlite database management system.



7.35 Can you construct the directory for this section and change into it?

A Database of Colitis Expression Data

New Terms		
Ensembl	Mysql	SQL
entity relation (ER) model	primary key	`sqlite3`
foreign key	relational database	`xtract`

7.36 We continue working with the colitis data. Can you copy the files `d1.txt` and `d60.txt` from the last session?

7.37 Fig. 7.4 shows an entity relation (ER) model of the database we wish to construct. Boxes are *entities*, ellipses *attributes*, and the diamond denotes a *relationship*. It's a one-to-one relationship, where each entry in `d1` has a corresponding entry in `d60`. The underlined attribute, `Probe`, is a unique *primary key*. We construct the database with the SQL script `colitis.sql`. It consists of two major chunks, one for each of the two tables in `colitis.db`.

Prog. 7.2 (`colitis.sql`)

⟨*colitis.sql*⟩≡
  ⟨*Create d1, Prog. 7.2*⟩
  ⟨*Create d60, Prog. 7.2*⟩

Table `d1` consists of attributes—the table columns—and a unique identifier of each row, the primary key. In our case the probe name serves as primary key.

⟨*Create d1, Prog. 7.2*⟩≡
```
create table d1 (
        ⟨Attributes, Prog. 7.2⟩
        primary key(Probe)
);
```

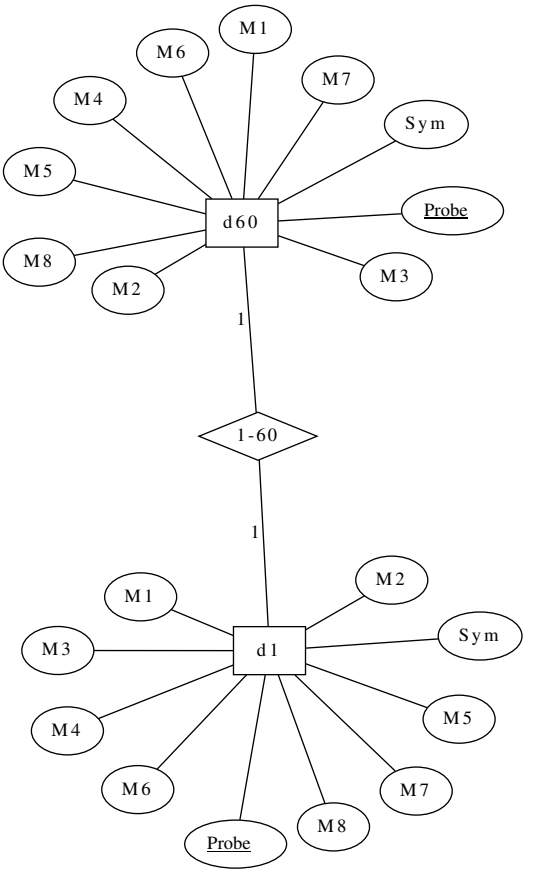


Fig. 7.4 Entity-relation (ER) model for `colitis.db`

We declare the ten attributes, which are common to both tables: A probe name of up to 255 characters, a gene symbol of up to 255 characters, and eight measurements.

⟨*Attributes, Prog. 7.2*⟩≡
```
Probe varchar(255),
Sym varchar(255),
M1 float,
```

Can you finish the attribute declarations?

7.38 Table `d60` has the same attributes and primary key as `d1`. In addition, it has a foreign key. A foreign key connects some attribute `x` in the current table with some attribute `y` in a different table. It is declared on the pattern

```
foreign key(x) references some_table(y)
```

In our case, `Probe` in `d60` refers to `Probe` in `d1` to make sure that `d60` contains no entry without a corresponding entry in `d1`. Can you construct `d60`?

7.39 We can now create the database `colitis.db` and list its tables. Commands starting with a dot are "meta commands"; they are not part of SQL, but specific to `sqlite3`.

⟨*cli*⟩+≡
```
sqlite3 colitis.db < colitis.sql
sqlite3 colitis.db .tables
```

What do you get?

7.40 Next, we look at the table *schema*.

⟨*cli*⟩+≡
```
sqlite3 colitis.db .schema
```

What are the table schema?

7.41 Time to fill our two tables, `d1` and `d60`, with data. For that we still need to split the first entry in each row into the probe ID and the locus symbol. Can you do that?

7.42 There is one more formatting step before we can import the data, `sqlite3` expects pipe symbols, `|`, as column delimiters. Can you replace the tabs in `d1.txt` and `d60.txt` by pipes?

7.43 Now we can import the data and query it.

⟨*cli*⟩+≡
```
sqlite3 colitis.db ".import d1.txt d1"
sqlite3 colitis.db ".import d60.txt d60"
sqlite3 colitis.db "select * from d1 limit 10"
```

What do you see?

7.44 We count the 33,297 entries in table `d1`.

⟨*cli*⟩+≡
```
sqlite3 colitis.db "select count(*) from d1"
```

Can you check that `d60` also has 33,297 entries?

7.45 We select the expression data for *ACKR2* and build our query from a *select* part, a *from* part, and a *where* part—the building blocks of many queries.

⟨*cli*⟩+≡
```
s="select *"
f1="from d1"
w="where sym like 'ACKR2'"
q="$s $f1 $w"
sqlite3 colitis.db "$q"
```

```
8079117|ACKR2|5.72763|6.04518|5.67683|5.69512|...
```

Can you get the corresponding expression data from day 60?

7.46 In the statistics section we concentrated on average expression values. So let's calculate the average expression of *ACKR2* on day 1. The code is not particularly elegant, because SQL lacks loop constructs, but there we are. You might be wondering why we write `m1` rather than `M1` as shown in Fig. 7.4. This is to prompt us to point out that capitalization in attribute names is ignored, so writing `m1` is identical to writing `M1`. As we saw before, the average expression of *ACKR2* on day 1 is approximately 5.82.

⟨*cli*⟩+≡

```
s="select (m1+m2+m3+m4+m5+m6+m7+m8)/8"
q="$s $f1 $w"
sqlite3 colitis.db "$q"
```

Can you repeat this computation for day 60?

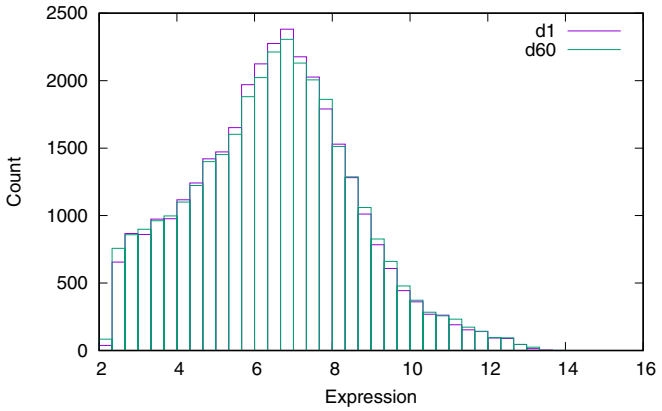


Fig. 7.5 Histograms of average expression values in the colitis study from day 1 and day 60

7.47 Fig. 7.5 shows the histograms of average expression values from days 1 and 60. At this resolution, the two distributions look very similar. Can you reproduce Fig. 7.5?

7.48 The SQL function `max` gives the maximum of a set of values. For example, we can select the gene with the highest average expression on day 1. This is an interval starting at position 50,320,334 on chromosome 14.

⟨*cli*⟩+≡

```
s="select sym, max((m1+m2+m3+m4+m5+m6+m7+m8)/8)"
q="$s $f1"
sqlite3 colitis.db "$q"
```

```
chr14:50320334-50320632|13.5845
```

Such intervals are used to monitor background hybridization, so we exclude them. The percent symbol in the `like` phrase matches any string. Now we get a control region.

⟨*cli*⟩+≡

```
w="where sym not like 'chr%'"
q="$s $f1 $w"
sqlite3 colitis.db "$q"
```

```
control|13.528025
```

When we exclude chromosome intervals and controls, we find the gene *RNA45S5*. The 45S ribosomal DNA encodes the precursor for the three rRNAs, 18S, 5.8S, and 28S, that are an essential part of the ribosomes in eukaryotes.

⟨*cli*⟩+≡

```
w="$w and sym not like 'control'"
q="$s $f1 $w"
sqlite3 colitis.db "$q"
```

```
RNA45S5|13.2305125
```

Can you find the most highly expressed gene on day 60?

7.49 The SQL function `min` gives the minimum of a set of values. It works like `max`. Which gene had the smallest average expression on day 1? on day 60?

7.50 Expression values are often discussed in terms of fold change. For this we need information from the two tables, `d1` and `d60`. We obtain this by joining them on their primary key, `probe`. The query we're about to write is a bit longer, so it gets its own script, `fc.sql`. By writing `as fc`, we declare the variable `fc`, for fold change. It allows us to sort the result by the fold change. In the joined table, all attribute names are duplicated, so they are disambiguated with the dot notation.

Prog. 7.3 (`fc.sql`)

⟨*fc.sql*⟩≡

```
select d1.sym, (d1.m1+d1.m2+d1.m3+d1.m4+
          d1.m5+d1.m6+d1.m7+d1.m8)/8-
          (d60.m1+d60.m2+d60.m3+d60.m4+
          d60.m5+d60.m6+d60.m7+d60.m8)/8 as fc
from d1 join d60
where d1.probe=d60.probe
and d1.sym not like 'chr%'
and d1.sym not like 'control'
order by fc
```

We run this query.

⟨*cli*⟩+≡

```
sqlite3 colitis.db < fc.sql | head
```

What do you observe?

7.51 Let's reformat the exponents we've just calculated into actual fold changes using the program `fc.awk`. It interacts with the user, reads the data, calculates the fold change, and prints it.

Prog. 7.4 (`fc.awk`)

⟨*fc.awk*⟩≡
```
BEGIN {
  ⟨User interaction, Prog. 7.4⟩
}
{
  ⟨Read data, Prog. 7.4⟩
  ⟨Calculate fold change, Prog. 7.4⟩
  ⟨Print fold change, Prog. 7.4⟩
}
```

In the user interaction, we respond to a call for help by showing how `fc.awk` is intended to be used.

⟨*User interaction, Prog. 7.4*⟩≡
```
if (h) {
  u = "Usage: sqlite3 colitis.db < fc.sql | "
  u = u "tr '|' '\\t' | awk -f fc.awk [-v h=1]"
  print u
  exit
}
```

How would you call for help?

7.52 We read the symbol and the exponent.

⟨*Read data, Prog. 7.4*⟩≡
```
sym = $1
ex = $2
```

Now we calculate the fold change, which is either an increase or a decrease. We make a note of the type of fold change and turn negative exponents positive.

⟨*Calculate fold change, Prog. 7.4*⟩≡
```
type = "incr"
if (ex < 0) {
  type = "decr"
  ex *= -1
}
```

Can you print the fold change and its type?

7.53 Which genes are the most upregulated in acute amebic colitis?

7.54 We can look up the function of genes in the Entrez database collection, which we have already encountered. It is almost certainly built from relational databases at some level, but these are not exposed to external users. Instead, we can use command line programs like `esearch` and `efetch` to find gene functions in the `gene` database. The function reports are in XML, from which we can extract tagged elements using `xtract`. Here we extract the summary.

⟨*cli*⟩+≡
```
q="REG1B [GENE] AND Homo sapiens [ORGN]"
esearch -db gene -query "$q" |
    efetch -format docsum |
    xtract -pattern DocumentSummary -element Summary
```

What is the function of the most up regulated gene, *REG1B*?

7.55 Which genes are the most down regulated in acute colitis?

7.56 What is the function of the most down regulated gene, *AQP8*?

Go

7.57 We have seen that SQL—albeit very useful—lacks easy branching and looping. As a result, SQL is often embedded in a proper programming language. As an example, we look at a program in the programming language Go [11]. Go is developed at Google, where it is used for building web server infrastructure. The programs in the biobox are also written in Go. Our program for demonstrating SQL embedding is called `fc` and prints fold changes. The Go code for `fc` is contained in the file `fc.go`, which consists of some 50 lines in its final form. This might seem like a lot, but don't worry, we'll go slow and show you every line of code. The outline of `fc.go` has hooks for imports and the logic of the main function. As before, a good way to use these hooks is to enter them as comments, which start with two forward slashes in Go, rather than the hashes we used in Awk.

```
// Hooks make good comments.
```

Prog. 7.5 (`fc.go`)

⟨*fc.go*⟩≡
```
package main

import (
        ⟨Imports, Prog. 7.5⟩
)

func main() {
        ⟨Main function, Prog. 7.5⟩
}
```

In the main function we open the database connection, construct the query, run the query, and print the result.

⟨*Main function, Prog. 7.5*⟩≡
```
⟨Open database connection, Prog. 7.5⟩
⟨Construct query, Prog. 7.5⟩
⟨Run query, Prog. 7.5⟩
⟨Print result, Prog. 7.5⟩
```

We open a connection to our colitis database. If this doesn't work, we throw a fatal error with a message.

⟨*Open database connection, Prog. 7.5*⟩≡
```
cd := "colitis.db"
db, err := sql.Open("sqlite3", cd)
if err != nil {
          log.Fatalf("couldn't open database %q", cd)
}
```

`sql` and `log` are packages, which we have to import to make the functions `sql.Open` and `log.Fatal` available.

⟨*Imports, Prog. 7.5*⟩≡
```
"database/sql"
"log"
```

We also import a package for connecting an `sqlite3` database. However, we won't directly refer to the package `go-sqlite3` and Go prohibits importing superfluous packages. To signal the compiler we are using the package even though we don't refer to any of its members, we mark it as a blank import by the underscore.

⟨*Imports, Prog. 7.5*⟩+≡
```
_ "github.com/mattn/go-sqlite3"
```

We use the same query we used before, only in Go strings are concatenated with the plus operator.

⟨*Construct query, Prog. 7.5*⟩≡
```
q := "select d1.sym, (d1.m1+d1.m2+d1.m3+d1.m4+" +
          "d1.m5+d1.m6+d1.m7+d1.m8)/8-" +
```

Can you complete the query?

7.58 We run the query.

⟨*Run query, Prog. 7.5*⟩≡
```
rows, err := db.Query(q)
```

Can you check whether we've had an error?

7.59 We defer the closure of the results rows we've just created until the end of the current function, `main`.

⟨*Run query, Prog. 7.5*⟩+≡
```
defer rows.Close()
```

We print the result in an output table, which we prepare before we iterate over the table rows.

⟨*Print result, Prog. 7.5*⟩≡
```
⟨Prepare output table, Prog. 7.5⟩
for rows.Next() {
        ⟨Print a row, Prog. 7.5⟩
}
```

Which columns does our output table contain?

7.60 We layout our table with a tab writer. This writes to the standard output stream—the screen—and uses at least one blank as column delimiter. A tab writer needs to be flushed after everything has been written to it, so we defer flushing ours. The first thing we write is a fenced-off table header.

⟨*Prepare output table, Prog. 7.5*⟩≡
```
w := tabwriter.NewWriter(os.Stdout, 1, 1, 1, ' ', 0)
defer w.Flush()
fmt.Fprint(w, "#Sym\tFC\tType\n")
```

We import the `tabwriter` and `os`.

⟨*Imports, Prog. 7.5*⟩+≡
```
"text/tabwriter"
"os"
```

The table generated by our query contains two columns. Can you remember what they are?

7.61 We prepare two variables for storing the data we read, a string for the gene symbols and a floating point number for the exponent of the fold change.

⟨*Prepare output table, Prog. 7.5*⟩+≡
```
var sym string
var e float64
```

When printing a row, we first scan its entries into the two variables we just declared. For this we supply the addresses, or pointers, of our variables, which we access with a prefixed ampersand.

⟨*Print a row, Prog. 7.5*⟩≡
```
err = rows.Scan(&sym, &e)
```

Can you check the error?

7.62 By default, we set the type of change to *increase*. But if the exponent is negative, we set the type to *decrease* and make the exponent positive. Then we calculate the fold change with the power function from the package `math`.

⟨*Print a row, Prog. 7.5*⟩+≡
```
t := "incr"
if e < 0 {
        t = "decr"
        e *= -1
```

```
}
f := math.Pow(2, e)
```

Can you import the package `math`?

7.63 We print the final output.

⟨*Print a row, Prog. 7.5*⟩+≡

```
fmt.Fprintf(w, "%s\t%.6f\t%s\n", sym, f, t)
```

Can you import `fmt`?

7.64 Can you guess the meaning of `%s` and `%.6f` in our printing command?

7.65 Before we can compile our program, we initialize the current directory as a Go module. This can have any name, we call ours `alpha.beth`.

⟨*cli*⟩+≡

```
go mod init alpha.beth
```

What do you observe?

7.66 We tidy up our module.

⟨*cli*⟩+≡

```
go mod tidy
```

What do you observe?

7.67 Now we can compile and run `fc`.

⟨*cli*⟩+≡

```
go build fc.go
./fc | head
```

What happens?

7.68 Which genes changed least between the two measurements?

Esembl

7.69 In contrast to Entrez, the Ensembl database collection *does* expose its SQL interface, so that anybody can query its Mysql database management system. We store the connection details in variable `c` and then redirect the names of all Ensembl databases into the file `dbs.txt`.

⟨*cli*⟩+≡

```
c="-h ensembldb.ensembl.org -u anonymous"
mysql $c -e "show databases" > dbs.txt
```

This works on `bash`, because it splits words on white space. On `zsh`, this behavior is invoked by first entering

```
setopt sh_word_split
```

How many databases make up Ensembl?

7.70 We are interested in the databases for human (*Homo sapiens*). How many are there?

7.71 The core databases of an organism are called `*_core_*`. How many core databases are there for human?

7.72 The core databases have version numbers. What is the highest version number for human-core you can find?

7.73 We list the tables in `homo_sapiens_core_106_38`.

⟨*cli*⟩+≡
```
c="$c homo_sapiens_core_106_38"
mysql $c -e "show tables"
```

How many are there?

7.74 The table `seq_region` contains information on the human genome sequence. We list its attributes.

⟨*cli*⟩+≡
```
mysql $c -e "describe seq_region"
```

How many are there?

7.75 We'd like to look at the contents of `seq_region`. We could just list the whole table, but let's be cautious and count its rows first.

⟨*cli*⟩+≡
```
mysql $c -e "select count(*) from seq_region"
```

How many rows does `seq_region` have?

7.76 We can just peek at the first ten rows of `seq_region`.

⟨*cli*⟩+≡
```
mysql $c -e "select * from seq_region limit 10"
```

The coordinate system IDs are all 1 in our result. So we ask, how many distinct coordinate system IDs are there?

⟨*cli*⟩+≡
```
q="select count(distinct(coord_system_id))"
q="$q from seq_region"
mysql $c -e "$q"
```

How many do you find?

7.77 The attribute `coord_system_id` in table `seq_region` suggests that there is a table `coord_system`, and there really is. Can you list its attributes?

7.78 How many entries does `coord_system` have?

7.79 What is the coordinate system ID for chromosomes from the GRCh38 version of the human genome?

7.80 We take a look at the names of the chromosomal sequence regions in GRCh38 to find that the names of the human chromosomes are, not surprisingly, 1–22, X, and Y.

⟨*cli*⟩+≡
```
q="select * "
q="$q from seq_region"
q="$q where coord_system_id = 4"
q="$q limit 30"
mysql $c -e "$q"
```

We've seen that `seq_region` contains lengths. So we list the lengths of the chromosomes for GRCh38 and guess their names are those that don't start with CHR or MT for mitochondrion.

⟨*cli*⟩+≡
```
q="select name, length"
q="$q from seq_region"
q="$q where coord_system_id = 4"
q="$q and name not like 'CHR%'"
q="$q and name not like 'MT'"
mysql $c -e "$q"
```

Is that true?

7.81 Fig. 7.6 shows the lengths of human chromosomes. Let's reproduce it. We've seen that the names of autosomes are numbers, so they are easy to plot along the x axis. The sex chromosomes are characters, so we write a program to convert them to numbers. The program is called `fcl.awk` for "format chromosome lengths".

Prog. 7.6 (`fcl.awk`)

⟨*fcl.awk*⟩≡
```
/^X/ { print 23 "\t" $2 }
/^Y/ { print 24 "\t" $2 }
/^[0-9]/
```

Can you reproduce Fig. 7.6?

7.82 The SQL function `sum` sums a range of numbers. Can you use it to calculate the total length of the human genome?

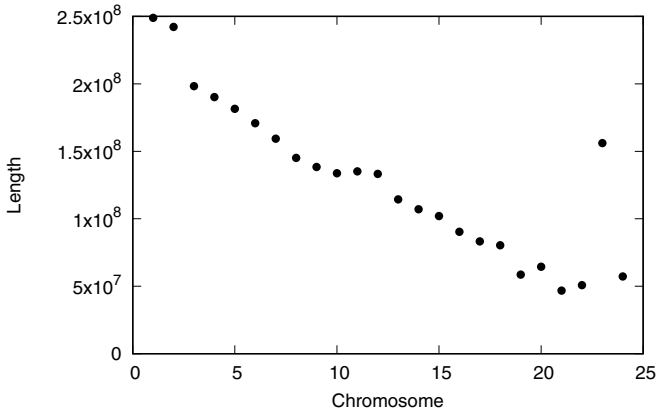


Fig. 7.6 Human chromosome lengths; chromosome 23 is the X, chromosome 24 the Y

7.83 Chromosomes are numbered by size. Their relative size was originally determined by looking at condensed chromosomes through the microscope. However, the DNA in chromosomes is complexed with proteins, mainly the histones that form the nucleosome cores. This means that chromosome length is not perfectly correlated with DNA content. Can you spot the "misorderings" in Fig. 7.6? You might also like to take another look at `chrLen.dat`.

7.84 We turn to genes. Can you list the attributes of table `gene`?

7.85 How many genes are known for the human genome?

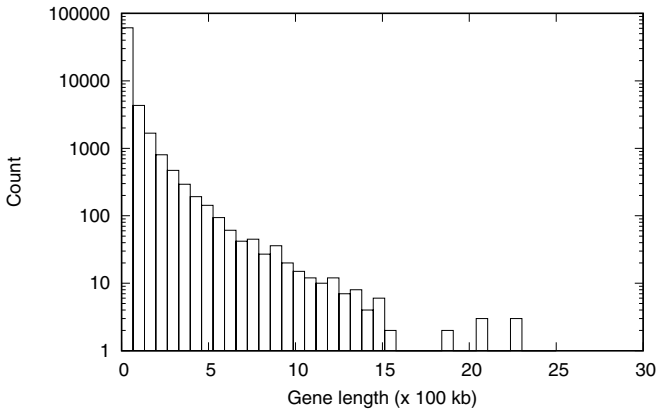


Fig. 7.7 The length distribution of human genes

7.86 Fig. 7.7 shows the distribution of gene lengths on a logarithmic y axis. The vast majority of genes are less than half a megabase long, but a few go up to 2.5 Mb. Can you use table `gene` to reproduce the histogram of gene lengths?

7.87 Which portion of the human genome is covered with genes?

7.88 The 2.2 Gb of "genes" might suggest that most of the genome is coding. However, genes consist of expressed parts, the exons, and unexpressed parts in between, the introns. The exons are stored in table `exon`. Can you list its attributes?

7.89 Exons contained in all transcripts of a gene are called *constitutive*. So we can sum the nucleotides contained in constitutive exons.

⟨*cli*⟩+≡
```
q="select sum(seq_region_end - seq_region_start + 1)"
q="$q from exon"
q="$q where is_constitutive=1"
mysql $c -e "$q"
```

What portion of the human genome is covered by constitutive exons?

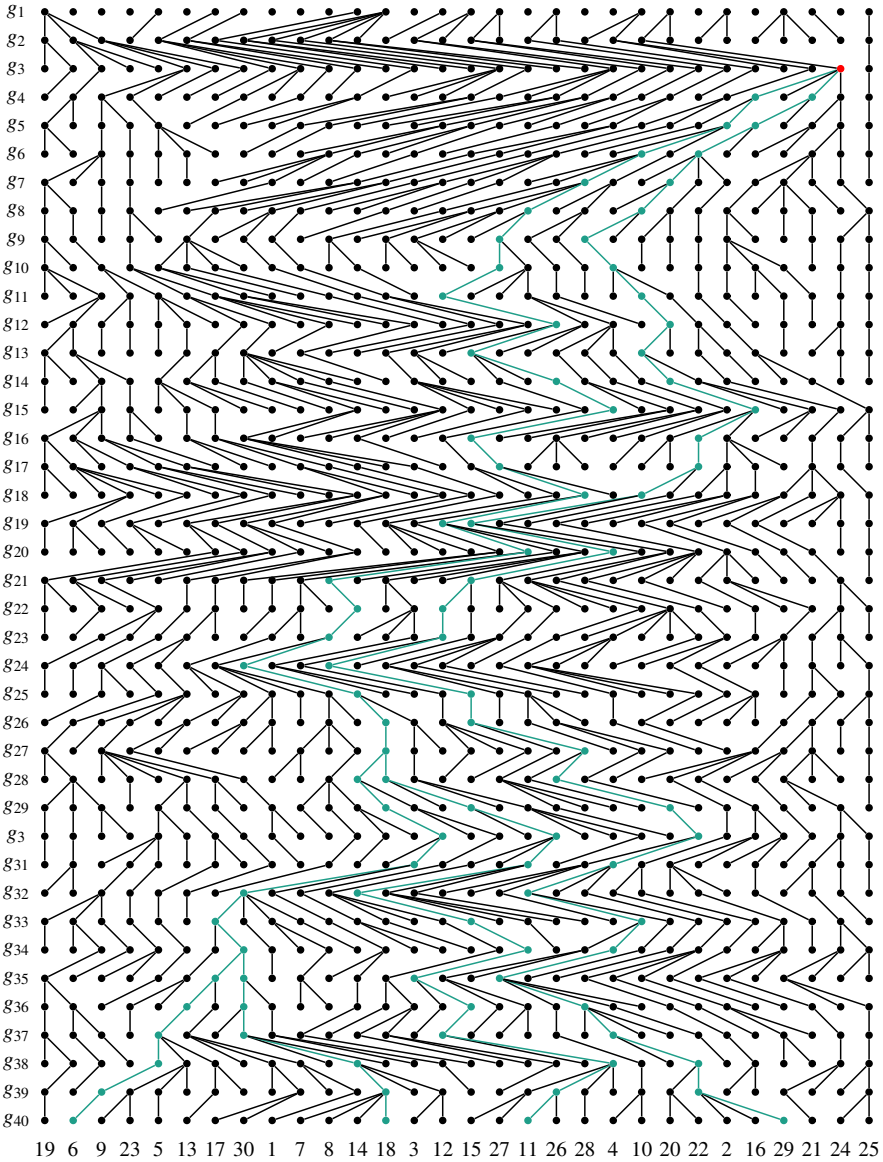
7.90 If we'd like to inquire about a particular gene, we need be able to query for its name. Table `xref` contains the attribute `display_label`, which corresponds to the gene names we used in the colitis expression analysis. For each gene, `xref` also contains a `description` of its function. One of the genes we singled out in the colitis study was *ACKR2*. Can you list its description?

7.91 Important entities in Ensembl such as genes or transcripts are labeled with a stable ID, `stable_id`. A stable ID can, for example, be entered on the Ensembl web site (`www.ensembl.org`) to quickly and unambiguously look up information on a particular gene. We'd like to look up the stable ID of *ACKR2*. For this we need to join `gene` and `xref` via `gene.display_xref_id` and `xref.xref_id`.

⟨*cli*⟩+≡
```
q="select display_label, gene.stable_id"
q="$q from xref join gene"
q="$q where display_xref_id = xref_id"
q="$q and display_label like 'ACKR2'"
mysql $c -e "$q"
```

What is the stable ID of *ACKR2*?

Part II
Answers



Wright-Fisher population of 30 genes over 40 generations; the red dot in g_3 indicates the most recent common ancestor of genes 6, 18, 11, and 29; their genealogy is shown in green

Chapter 1
The Unix Command Line

1.1 Getting Started

1.1 Ours happens to be,

```
/home/beth/books/beb
```

your's is bound to be different.

1.2 We get the *path* of the current directory.

1.3 We find out with

⟨*cli*⟩+≡
```
pwd
```

which in our case is `/home/beth`. This directory is called the *home* directory.

1.4 We return to the `beb` directory.

1.5 We again change into the home directory, which is denoted by tilde (`~`).

1.6 Ours is `/bin/bash`.

1.7 Right at the end you should see the `export` line we just produced, on our test environment this is

```
export BEB=/home/beth/books/beb
```

If you get something quite different, copy your backup resource file to the home directory and try again.

1.8 `ls` *lists* files and directories, so we get `ch`, the one directory contained in our current directory, `beb`.

1.9 In addition to `ch`, we get `.bashrc`, a dot, and a double dot. A file preceded by a dot like `.bashrc` is a hidden file, which is why we didn't see `.bashrc` with plain `ls`. The single and double dots are special hidden "files". The single dot is the name of the current directory, the double dot the name of the parent directory.

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_8

1.10 We make directory 1, change into it, and repeat.

⟨*cli*⟩+≡
```
mkdir 1
cd 1
mkdir 1
cd 1
```

1.11 We make our directories and list them.

⟨*cli*⟩+≡
```
mkdir td1 td2
ls
```

1.12 We change into `td1` and back into its parent.

⟨*cli*⟩+≡
```
cd td1
cd ..
```

1.13 The star, or wildcard (∗), matches any completion of `td`, so we delete both `td1` and `td2`, which we can see with `ls`.

1.14 This causes the error message

```
rmdir: failed to remove ‘td1’: Directory not empty
```

1.15 We delete the file `td1/tf` and remove the empty directory `td1`.

⟨*cli*⟩+≡
```
rm td1/tf
rmdir td1/
```

1.16 We make the directory `td1`, add a file, and remove `td1` recursively.

⟨*cli*⟩+≡
```
mkdir td1
touch td1/tf
rm -r td1/
```

1.17 We make `td1`, change into it, and touch `tf1` and `tf2`. Then we use the wildcard notation to copy the two new files to the neighboring directory.

⟨*cli*⟩+≡
```
mkdir td1
cd td1/
touch tf1 tf2
cp * ../td2/
```

1.18 We remove the files and copy them from the neighboring directory to the current directory using the wildcard notation.

⟨*cli*⟩+≡
```
rm *
cp ../td2/* .
```

1.19 We move `tf2` to `tf4` and then `tf3` to `tf4`.

⟨*cli*⟩+≡
```
mv tf2 tf4
mv tf3 tf4
```

The system has now overwritten the old `tf4` by `tf3` without checking whether that's really what you intended to do. The same thing would happen when moving files containing valuable information, so beware.

1.20 `C-b` moves the cursor one position back.

1.21 `C-e` jumps to the end of a line.

1.22 `M-d` is a command for editing the command line, it deletes the word to the right of the cursor.

1.23 Repeated undo rolls back the steps taken so far.

1.24 `C-y` pastes (*yanks*) what was just deleted.

1.25 Our solution is to delete *man* (`M-backspace`), move in front of *dog* by three `M-b`, insert *man* (`C-y`), delete *dog* (`M-d`), jump to the end of the line (`C-e`), and insert *dog* (`C-y`).

1.26 We use `head` to look at the head of the command history.

⟨*cli*⟩+≡
```
history | head
```

The command numbering tells us there are 1000 items in our list. Your result may well differ.

1.27 Two exclamation marks repeat the last command.

1.28 The command `!-3` repeats to the current command minus 3.

1.29 `C-n` gets the next command down the list.

1.30 The command history doesn't wrap around, so `C-p` on the first command doesn't do anything. We can, of course, walk to the next command down with `C-n`.

1.31 `M->` is the opposite of `M-<` and gets us to the most recent command line, which may well be empty.

1.32 Repeated `C-r` walks up the matching commands in the history.

1.33 `C-g` exits the history search.

1.34 The command for finding `man` pages for a keyword like `bash` is

⟨*cli*⟩+≡
```
man -k bash
```

1.35 The help page contains a section on searching, which explains that

```
/pattern
```

generates a search; it also tells us to quit with `q`. So we quit the help page

```
q
```

This means that

```
/Commands for Moving
```

gets us to the section on moving along the command line and

```
/History
```

to the section on moving along the command history.

1.36 The expected answer, 64, is printed—or rather *echoed*—to the screen. When we drop the dollar, we get a syntax error as echo expects a *value*, which is referred do by the dollar, rather than a computation, which is started by the double parentheses.

1.37 We access the value of a variable by prefixing it with a dollar.

⟨*cli*⟩+≡
```
echo $nc
```

1.38 Strings in single quotes are echoed verbatim, so `$nc` is not replaced by its value.

1.39 The shell can only compute with integers, so it tells us 22/7 is 3.

⟨*cli*⟩+≡
```
echo $((22/7))
```

1.40 The answer given by `bc` is 3.142..., which is roughly the ratio of a circle's circumference to its diameter, π.

1.41 Without `-l`, `bc` reverts to integer division.

1.42 The computation presumably fails because the result is too large; but the command

⟨*cli*⟩+≡
```
echo '2^64' | bc
```

gives the result 18,446,744,073,709,551,616. This is more difficult to read than its approximation,

$$2^{64} = \left(2^{10}\right)^{6.4} \approx \left(10^{3}\right)^{6.4} \approx 10^{19}.$$

1.43 32 nucleotides is all we need, since

$$4^{32} = 2^{64}.$$

1.2 Files, Directories, and Programs

1.44 We change into the directory for Chapter 1, make the directory for Section 2, and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/1/
mkdir 2
cd 2/
```

1.45 We list the contents of the root directory.

⟨*cli*⟩+≡
```
ls /
```

1.46 There are 24 directories in our root directory, yours might be different.

⟨*cli*⟩+≡
```
ls / | wc
```

1.47 The `-l` option gets the lines.

⟨*cli*⟩+≡
```
ls / | wc -l
```

1.48 We try to list `/bin/ls`,

⟨*cli*⟩+≡
```
ls /bin/ls
```

to find that—at least on our system—`ls` is contained in `/bin`, it may be located somewhere else on yours.

1.49 On our system, the current `which` command is located in the same directory as the current `ls` command, `/bin`.

⟨*cli*⟩+≡
```
which which
```

1.50 We find out by listing and counting.

⟨*cli*⟩+≡
```
ls ~ | wc -l
```

1.51 We print the header line and the data line, each terminated by a newline.

⟨*cli*⟩+≡
```
printf ">dnaN\nATGAAAATATTA\n"
```

1.52 `s.fasta` contains 19 bytes, the five characters of the header `>dnaN`, the twelve nucleotides `ATGAAAATATTA`, and two newlines.

⟨*cli*⟩+≡
```
wc -c s.fasta
```

1.53 An octal 23 corresponds to a decimal $2 \times 8^1 + 3 \times 8^0 = 19$.

1.54 From the output of `od` we take

Octal Code	Character
`076`	`>`
`141`	`a`
`101`	`A`
`012`	`\n`

You can think of octal `012` as the stop codon of a line of text.

1.55 `A` corresponds to decimal 65.

1.56 The ASCII code comprises $2^7 = 128$ characters, which is also the number of entries in the ASCII table of the manual.

1.57 `ACGT` corresponds to decimal 65, 67, 71, and 84.

⟨*cli*⟩+≡
```
printf "ACGT" | od -t u1
```

```
0000000  65  67  71  84
0000004
```

1.58 The redirect (>) overwrites a file—so beware.

1.59 It contains 38 bytes.

⟨*cli*⟩+≡
```
wc -c s.fasta
```

1.60 The number of lines in `progs.txt` and the number of programs in the biobox should agree.

⟨*cli*⟩+≡
```
wc -l progs.txt
ls bin/ | wc -l
```

1.61 There are three plotting programs in Table B.2, `plotLine`, `plotSeg`, and `plotTree`.

1.62 It's a list of directories separated by colons, which includes, for example, the directory `/bin` we've already looked at.

⟨*cli*⟩+≡
```
echo $PATH
```

1.63 You get the help message for `cres`. If this fails, try again setting the path, perhaps in a new terminal window.

1.64 A string in single quotes is printed verbatim, while in double quotes variables are replaced. In our case variable replacement would lead to an unnecessarily verbose entry in the resource file.

1.65 We run `cres` on our sequence file, which is located in the directory for Section 1.2, to get the total nucleotide count of 24 and the nucleotide frequencies.

⟨*cli*⟩+≡
```
cres $BEB/ch/1/2/s.fasta
```

```
Total: 24
Residue Count Fraction
A       14    0.583
G       4     0.167
T       6     0.25
```

1.66 There are 190 million bytes in the compressed file, 600 million characters when uncompressed. So the compression ratio is a bit better than 3.

⟨*cli*⟩+≡
```
wc -c data.tgz
gunzip data.tgz
wc -c data.tar
```

1.67 We get a report on which files were extracted into the new directory `data`.

```
data/
data/dmChr3R.fasta
data/dmChr3L.fasta
...
```

1.68 We remove `data.tar` with `rm`.

⟨*cli*⟩+≡
```
rm data.tar
```

1.69 We've extracted 47 data files, of which 27 are FASTA files.

⟨*cli*⟩+≡
```
ls data | wc -l
ls data/*.fasta | wc -l
```

1.70 We change into our working directory, copy the genome sequence and find it is roughly 580 kb long.

⟨*cli*⟩+≡
```
cd $BEB/ch/1/2/
cp $BEB/data/mgGenome.fasta .
cres mgGenome.fasta
```

```
Total: 580076
Residue Count  Fraction
A       200544 0.346
C       91515  0.158
G       92306  0.159
T       195711 0.337
```

1.71 We get the FASTA header and nine lines of DNA sequence.

⟨*cli*⟩+≡

```
head mgGenome.fasta
```

```
>gi|84626123|gb|L43967.2| Mycoplasma genitalium G37, complete genome
TAAGTTATTATTTAGTTAATACTTTTAACAATATTATTAAGGTATTTAAAAAATACTATTATAGTATTTA
ACATAGTTAAATACCTTCCTTAATACTGTTAAATTATATTCAATCAATACATATATAATATTATTAAAAT
ACTTGATAAGTATTATTTAGATATTAGACAAATACTAATTTTATATTGCTTTAATACTTAATAAATACTA
CTTATGTATTAAGTAAATATTACTGTAATACTAATAACAATATTATTACAATATGCTAGAATAATATTGC
TAGTATCAATAATTACTAATATAGTATTAGGAAAATACCATAATAATATTTCTACATAATACTAAGTTAA
TACTATGTGTAGAATAATAAATAATCAGATTAAAAAAATTTTATTTATCTGAAACATATTTAATCAATTG
AACTGATTATTTTCAGCAGTAATAATTACATATGTACATAGTACATATGTAAAATATCATTAATTTCTGT
TATATATAATAGTATCTATTTTAGAGAGTATTAATTATTACTATAATTAAGCATTTATGCTTAATTATAA
GCTTTTTATGAACAAAATTATAGACATTTTAGTTCTTATAATAAATAATAGATATTAAAGAAAATAAAAA
```

1.72 We copy `mgGenes.txt` and list its first ten lines. Each line describes one gene with an identifier, a start and an end position, the strand, and a name, if available.

⟨*cli*⟩+≡

```
cp $BEB/data/mgGenes.txt .
head mgGenes.txt
```

```
MG_001  686     1828    +       dnaN
MG_002  1828    2760    +
MG_003  2845    4797    +       gyrB
MG_004  4812    7322    +       gyrA
MG_005  7294    8547    +       serS
MG_006  8551    9183    +       tmk
MG_007  9156    9920    +
MG_008  9923    11251   +
MG_009  11251   12039   +
MG_010  12068   12724   +
```

1.73 The layout is the same as for the head, one line per gene, three or four fields per line.

⟨*cli*⟩+≡

```
tail mgGenes.txt
```

```
MG_462  566186  567640  -       gltX
MG_463  567627  568406  -
MG_464  568399  569556  -
MG_465  569528  569914  -       rnpA
MG_466  569883  570029  -       rpL34
MG_467  570055  570990  -
MG_468  570994  576345  -
MG_526  576351  577205  -
MG_469  577268  578581  -
MG_470  579224  580033  -
```

1.74 `cres` tells us there are 381 residues, but one of these is the stop codon, so the protein consists of 380 amino acids.

⟨*cli*⟩+≡
```
cutSeq -r 686-1828 mgGenome.fasta | translate | cres | head
```

1.75 The first gene ends at 1828, so the last nucleotide of its stop codon is the first nucleotide of the subsequent gene. So the first and second genes overlap by one nucleotide. Overlapping genes are a feature of small genomes.

1.76 *M. genitalium* has 525 genes.

⟨*cli*⟩+≡
```
wc -l mgGenes.txt
```

1.77 `less` has the same navigation commands as `man`. That's because `man` uses `less` as pager. By repeatedly pressing the space bar, we can move forward through the list of genes.

1.78 *M. genitalium* has 299 genes on the forward strand and 226 genes on the reverse. Since 299 + 226 = 525, it all adds up.

⟨*cli*⟩+≡
```
cut -f 4 mgGenes.txt | grep + | wc -l
cut -f 4 mgGenes.txt | grep - | wc -l
```

1.79 299 seems quite different from 226. One way to test the significance of the difference of 73 genes would be to repeatedly toss a coin 525 times and determine the frequency with which at least 299 heads are found. Repeatedly tossing a coin 525 times—that'd be an awful lot of coin tosses, so we postpone this until we know how to get the computer to do it for us.

1.80 We see that pluses are clustered 5', minuses 3'.

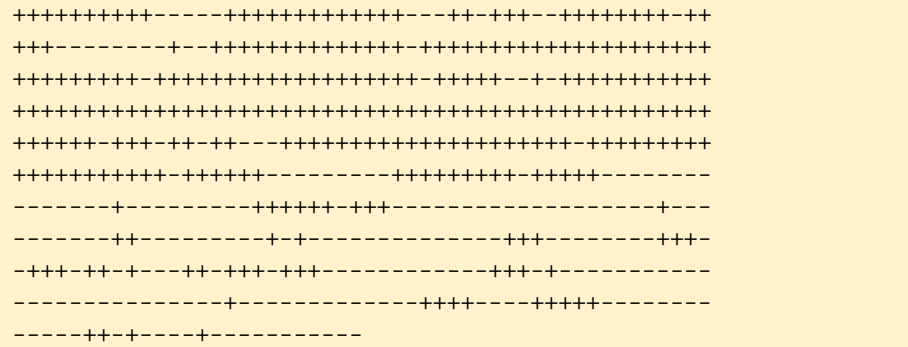
```
++++++++++----++++...++-+----+----------
```

1.81 We lose the newline at the end of the string, it is now concatenated with the prompt. Not so nice.

1.82 From the `man` page we learn about the `-w` switch of `fold`.

⟨*cli*⟩+≡
```
cut -f 4 mgGenes.txt |
    tr -d '\n' |
    awk '{print}' |
    fold -w 50
```

The uneven distribution of strandedness is now even more noticeable than in the stretched-out string.



1.83 We get two columns of numbers, pairs of x and y values marking the corners of boxes for genes. Upward boxes on the positive strand and downward boxes on the negative.

```
100     0
100     1
400     1
400     0
600     0
600     -1
1500    -1
1500    0
100     0
```

1.84 We list the options, from which we learn how to adjust the y range (`-Y`), set the x axis label (`-x`), and unset the y axis (`-u y`). Since the y range isn't shown in Fig. 1.2, we experiment with a few ranges to find that -5:5 looks about right.

⟨*cli*⟩+≡

```
plotLine -h
printf "100 400 +\n600 1500 -\n" | drawGenes |
    plotLine -Y -5:5 -x Position -u y
```

1.85 We draw the genes to get Fig. 1.4, where the bias in strandedness is immediately visible.

⟨*cli*⟩+≡

```
cut -f 2-4 mgGenes.txt | drawGenes |
    plotLine -Y -5:5 -x Position -u y
```

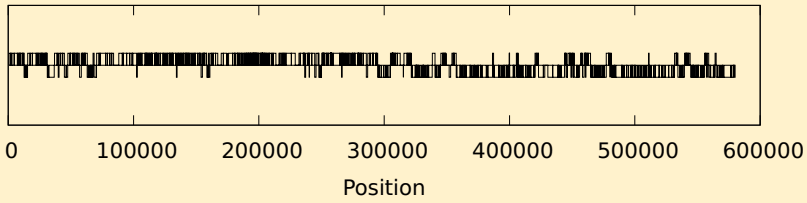


Fig. 1.4 The genes along the genome of *M. genitalium*

1.3 Scripts

1.86 We change into the directory for this chapter, make the section directory, `3`, and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/1/
mkdir 3
cd 3/
```

1.87 The remainder of dividing by 2 is always either zero or one. So our source of random 1 and 0 is the remainder (or modulo) operation, `%`, applied to random integers.

⟨*cli*⟩+≡
```
echo $(($RANDOM % 2))
```

1.88 With `man seq` we find we can do

⟨*cli*⟩+≡
```
seq 3
```

1.89 The variable `s` contains the numbers 1, 2, and 3 in a row.

⟨*cli*⟩+≡
```
echo $s
```

```
1 2 3
```

1.90 We loop the coin tossing step.

⟨*cli*⟩+≡
```
for a in $(seq 3); do echo $(($RANDOM % 2)); done
```

1.91 On Ubuntu, which is the most widely used Unix on Windows, the package manager is called `apt`. Its installation function is usually coupled with an update

and a search. The result of a search for `emacs` is a bit overwhelming, so we reduce it with `grep` to entries that start with `emacs`.

⟨*cli*⟩+≡
```
sudo apt update
apt-cache search emacs | grep ^emacs
sudo apt install emacs
```

On macOS, `brew` also has a search function, and it is usually a good idea to search for a package before installing it. Installing `emacs` with `brew` is thus

```
brew search emacs
brew install emacs
```

1.92 It's up to you. If you opt for install now, this takes a bit of time. If you opt for install later, remember to do so when a command isn't found.

1.93 Homebrew installs by default console `emacs`. `apt` installs by default graphical `emacs`. Graphical `emacs` can be converted into console `emacs` by starting it with no window.

⟨*cli*⟩+≡
```
emacs -nw
```

1.94 Graphical `emacs` without ampersand blocks the command line until it is stopped by entering `C-c` on the command line. `C-c` is a general mechanism for interrupting a running program.

Console `emacs` with ampersand doesn't start the editor at all.

1.95 We grab the 1's and count them.

⟨*cli*⟩+≡
```
bash ct.sh 525 | grep 1 | wc -l
```

1.96 We didn't get a positive result, but you might have.

⟨*cli*⟩+≡
```
for a in $(seq 20)
do
    bash ct.sh 525 | grep 1 | wc -l
done
```

1.97 The commands in Table 1.2 are also preserved. So key combinations like `C-p`, `C-n`, `M-<`, and `M->` move up and down the screen. `C-r` searches backward (and `C-s` forward).

1.98 It says in the tutorial that `C-x k` followed by Enter terminates the tutorial. In fact, this is the general mechanism for *killing* a buffer, in this case the tutorial buffer.

1.99 The tutorial promised that `C-x C-c` gets you out.

1.100 Our script, `ict.sh`, calls `ct.sh`.

Prog. 1.4 (`ict.sh`)

⟨*ict.sh*⟩≡
```
for i in $(seq $1)
do
    bash ct.sh $2 |
         grep 1 |
         wc -l
done
```

We ran it, but in 30 iterations never found at least 299 heads.

1.101 We carry out the simulation and save the results to the file `ict.dat`. Then we sort *numerically* (`-n`) the results to find the minimum and maximum, 231 and 299 in our case. We use the difference between 231 and 299, 68, as the number of bins in the final histogram.

⟨*cli*⟩+≡
```
bash ict.sh 1000 525 > ict.dat
sort -n ict.dat | head -n 1
sort -n ict.dat | tail -n 1
histogram -b 68 ict.dat | plotLine -x Heads -y Count
```

1.102 The default alphabetical order flips the numerical order of 2 and 10.

1.103 Here are our results:

Iterations	≥ 299
100	0
1000	2
10000	8

So our error probability when rejecting the null hypothesis of random distribution between strands is about $P \approx 10^{-3}$. Since this is much lower than the usual rejection threshold of $\alpha = 0.05$, we would reject the null hypothesis and conclude that the distribution of genes between strands is biased in *M. genitalium*. But we find this puzzling as the strand labels are arbitrary. So perhaps it *is* chance, after all?

1.104 We came up with this:

- Determine the number of + observed in the first half of the strand list and call it n_o.
- Shuffle the strand list and again count the + in the first half of the list, call it n_1.
- Repeat the shuffling and counting.
- Determine the frequency with which $n_i \geq n_o$.

1.105 We copy the file `mgGenes.txt` to our working directory and find there are 227 genes on the positive strand in its upper half.

⟨*cli*⟩+≡
```
cp $BEB/data/mgGenes.txt .
head -n 263 mgGenes.txt | grep + | wc -l
```

1.106 We run the command

⟨*cli*⟩+≡
```
cut -f 4 mgGenes.txt | sort -R | less
```

to find a block of + followed by a block of -, or the other way round. That's because `sort -R` clusters identical elements in a list; so the only variation in its result is the order of the blocks of plus and minus.

1.107 `sort -R` randomizes the numbers in the first column and hence also the column of plus and minus.

⟨*cli*⟩+≡
```
cut -f 4 mgGenes.txt | cat -n | sort -R | less
```

1.108 Here is our `ss.sh`.

Prog. 1.5 (`ss.sh`)

⟨*ss.sh*⟩≡
```
for a in $(seq $1)
do
    cut -f 4 mgGenes.txt |
          cat -n |
          sort -R |
          head -n 263 |
          grep + |
          wc -l
done
```

When we ran it, we got 156, 149, 153, 153, 150, and so on, but never anything close to 227.

⟨*cli*⟩+≡
```
bash ss.sh 10
```

1.109 We run the simulation, find an appropriate number of bins for the histogram, 38 in our case, and plot the results to get the distribution in Fig. 1.5. We set the x range such that it includes the observed value of 227, draw an arrow pointing to 227 on the x axis, and observe that its position is far to the right of the histogram.

⟨*cli*⟩+≡
```
bash ss.sh 1000 > ss.dat
sort -n ss.dat | head -n 1
sort -n ss.dat | tail -n 1
histogram -b 38 ss.dat |
    plotLine -X 125:230 -x "Forward Genes" -y Count \
               -g "set arrow from 227,20 to 227,2"
```

In other words, the bias in strandedness is very significant and had already been observed in the original publication of the *M. genitalium* genome, where the authors suggest it has to do with the movement of the replication fork around the circular

genome [15]. The *M. genitalium* genome is arranged such that the start and end of the sequence coincide with the origin of replication.

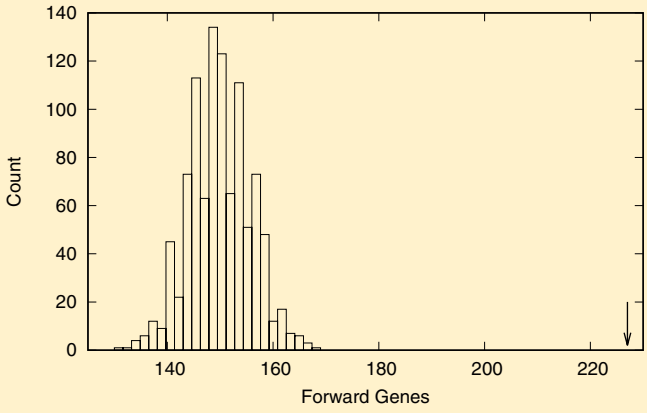


Fig. 1.5 Distribution of the number of *M. genitalium* genes found on the first half of the forward strand by chance alone; the observed number of 227 forward genes is marked by the arrow

1.110 We print the second and third column of the input.

```
686 1828
1828 2760
2845 4797
...
```

1.111 Without an argument, we print the entire line. The full line is called `$0`, so we get the same result with

⟨*cli*⟩+≡
```
awk '{print $0}' mgGenes.txt | head
```

1.112 We build a pipeline for plotting.

⟨*cli*⟩+≡
```
awk '{print $3 - $2 + 1}' mgGenes.txt | histogram |
    plotLine -x "Gene Length (bp)" -y Count
```

This gives us the distribution of gene lengths in Fig. 1.6, which is quite heavy-tailed.

1.113 We assign the gene length to a variable for convenient printing, then we sort numerically and look at the top of the list.

⟨*cli*⟩+≡
```
awk '{l = $3 - $2 + 1; print l, $1, $5}' mgGenes.txt |
    sort -n | head
```

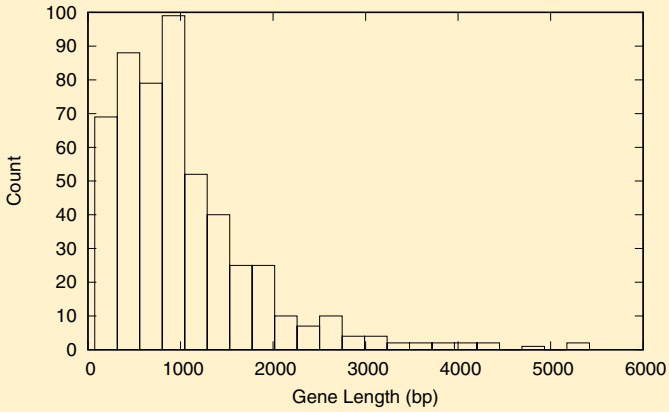


Fig. 1.6 Distribution of gene lengths in *M. genitalium*

We find five unnamed genes length 74. We can safely assume they don't code for proteins as 74 is not even a multiple of 3.

```
74 MG_479
74 MG_489
74 MG_493
74 MG_496
74 MG_499
...
```

At the end of the list we find the gene *hmw2*, which is involved in cell adherence [10] and is 5418 bp long.

```
5418 MG_218 hmw2
```

1.114 We've already looked up the genome length of 580,076 bp, but there's no harm in doing it again. The combined length of the genes is 540,447 bp. So $580076/540447 \times 100 \approx 93\%$ of the genome is covered by genes.

⟨*cli*⟩+≡

```
cres $BEB/data/mgGenome.fasta
awk '{s = s + $3 - $2 + 1}END{print s}' mgGenes.txt
echo '540447/580076*100' | bc -l
```

```
93.16830898020259414200
```

1.115 We divide the sum of gene lengths by the number of lines to get the average gene length, approximately 1 kb.

⟨*cli*⟩+≡

```
awk '{s += $3 - $2 + 1; c++}END{print s / c}' mgGenes.txt
```

```
1029.42
```

Here we abbreviated the expression `s=s+x` to `s+=x` and `c=c+1` to `c++`.

1.116 The first entry in `lengths` is `lengths[0]`, the second `lengths[1]`, and the third `lengths[2]`. So we program

⟨*Calculate variance of gene lengths, Prog. 1.2*⟩≡
```
print lengths[0], lengths[1], lengths[2]
```

and run it.

⟨*cli*⟩+≡
```
awk -f varLen.awk mgGenes.txt
```

```
1143 933 1953
```

1.117 To omit the `BEGIN` block, we can either delete it or comment it out with hashes. Working with comments is easily reversible, so that's our preferred method when experimenting with code. Without the `BEGIN` block, we lose the first entry.

```
933 1953
```

In Awk, a variable is declared and initialized to its null value at run time. Awk guesses the type of the variable from its context and if there is nothing to go by, the default type is string. So without the `BEGIN` block, the first time `n` is used, it's a string; then we add to it, at which point it becomes a number. But that's *after* we've read the first length. That number is not lost, we can access it via its index, the empty string,

```
print lengths[""]
```

But it's simpler to just initialize `n` to zero.

1.118 We sum the gene lengths and divide by their number to find the average.

⟨*Calculate variance of gene lengths, Prog. 1.2*⟩+≡
```
for (i = 0; i < n; i++)
  s += lengths[i]
avg = s / n
print "avg:", avg
```

1.119 We repeat the iteration across the lengths and apply equation (1.1).

⟨*Calculate variance of gene lengths, Prog. 1.2*⟩+≡
```
for (i = 0; i < n; i++)
  ss += (lengths[i] - avg)^2
var = ss / (n - 1)
print "var:", var
```

So we compute that the variance of gene lengths is roughly 6.6×10^5.

⟨*cli*⟩+≡
```
awk -f varLen.awk mgGenes.txt
```

```
1143 933 1953
avg: 1029.42
var: 662399
```

1.120 Yes, `awk` can calculate $2^{64} \approx 1.8 \times 10^{19}$.

⟨*cli*⟩+≡
```
awk 'BEGIN{print 2^64}'
```

```
18446744073709551616
```

1.121 We print all genes on the plus strand.

```
MG_001  686     1828    +       dnaN
MG_002  1828    2760    +
MG_003  2845    4797    +       gyrB
...
```

1.122 We just switch the character in the match field from plus to minus. Since minus is not part of the regular expression syntax, we can safely omit the backslash, but there's also no harm in leaving it in.

⟨*cli*⟩+≡
```
awk '$4 ~ /-/' mgGenes.txt | head
```

```
MG_011  12701   13564   -
MG_012  13569   14432   -
MG_013  14395   15216   -       folD
...
```

1.123 We apply `diff` to `minus1.txt` and `minus2.txt`.

⟨*cli*⟩+≡
```
diff minus*
```

This gives two lines of output.

```
46a47
> MG_261        315701  318325  +       polC-2
```

The first line tells us that line 46 in the first file was added (`a`) to make line 47 in the second file. That line concerns gene `polC-2`, which is on the plus strand, but has a name with a hyphen (minus).

1.124 We copy the file and find that `awk` and `wc` both tell us there are 8289 lines in `mgGenome.fasta`.

⟨*cli*⟩+≡
```
cp $BEB/data/mgGenome.fasta .
awk 'END{print NR}' mgGenome.fasta
wc -l mgGenome.fasta
```

1.125 The last field in a line is called `$NF`. It is either a gene name or the gene strand, if there's no name.

⟨*cli*⟩+≡
```
awk '{print $NF}' mgGenes.txt | head
```

```
dnaN
+
gyrB
...
```

1.126 We count the lines with five fields and divide by the total number of lines times 100 to find that only 42%, or less than half, of the genes have names.

⟨*cli*⟩+≡
```
awk 'NF == 5{c = c + 1}END{print c / NR * 100}' mgGenes.txt
```

```
41.7143
```

1.127 We copy the proteome, count the headers and print the result to find that the genome of *M. genitalium* encodes 476 proteins.

⟨*cli*⟩+≡
```
cp $BEB/data/mgProteome.fasta .
awk '/^>/{c++}END{print c}' mgProteome.fasta
```

1.128 We sum the lengths of the data lines to find the familiar 580,076 bp.

⟨*cli*⟩+≡
```
awk '!/^>/{s+=length($1)}END{print s}' mgGenome.fasta
```

```
580076
```

1.129 We filter for the data lines to avoid including part of the header.

⟨*cli*⟩+≡
```
head -n 3 mgGenome.fasta | awk '!/^>/{t = t $1}END{print t}'
```

1.130 Splitting at `a` gives an empty string followed by `xb`.

⟨*cli*⟩+≡
```
printf "axb\n" |
    awk '{n=split($1,a,"a");for(i=1;i<=n;i++)print a[i]}'
```

```
xb
```

Similarly, splitting at `b` gives `ax` followed by an empty string.

⟨*cli*⟩+≡
```
printf "axb\n" |
    awk '{n=split($1,a,"b");for(i=1;i<=n;i++)print a[i]}'
```

```
ax
```

1.131 We split our toy string as before and pipe the result through a filter that stops empty lines.

⟨*cli*⟩+≡
```
  printf "axb\n" |
      awk '{n=split($1,a,"b");for(i=1;i<=n;i++)print a[i]}' |
      awk '!/^$/'
```

```
ax
```

1.132 An empty string as delimiter splits the target string into all characters.

⟨*cli*⟩+≡
```
  printf "axb\n" |
      awk '{n=split($1,a,"");for(i=1;i<=n;i++)print a[i]}'
```

```
a
x
b
```

1.133 We get the header, remove the first character, split at `|`, and remove the empty line.

⟨*cli*⟩+≡
```
  grep '>' mgGenome.fasta | tr -d '>' |
      awk '{n=split($1,a,"|");for(i=1;i<=n;i++)print a[i]}' |
      awk '!/^$/'
```

```
gi
84626123
gb
L43967.2
```

1.134 We add a `for` loop with `in` to the end of our script to list each nucleotide and its count. Our Awk program is constructed in two steps to make it fit the page; you can also try this, it works. But on the screen we'd normally just write the Awk code in one string—or use an editor.

⟨*cli*⟩+≡
```
  a="{counts[\$1]++}"
  a="$a END{for(c in counts) print c, counts[c]}"
  printf "A\nA\nA\nC\nC\nT\n" | awk "$a"
```

```
A 3
C 2
T 1
```

1.135 We split each line and count the residues it contains. We also sum the total residue count.

⟨*Count residues, Prog. 1.3*⟩≡
```
n = split($1, a, "")
for (i = 1; i <= n; i++)
  counts[a[i]]++
t += n
```

1.136 We've already done the sums, so we just print the result.

⟨*Print total residue count, Prog. 1.3*⟩≡
```
print "Total:", t
```

1.137 We print a table header and then iterate over the residues to print the count and frequency of each one. A residue is a string (`%s`), a count a decimal integer (`%d`), and a frequency a floating point number (`%f`).

⟨*Print residue counts, Prog. 1.3*⟩≡
```
printf "Res.\tCount\tFreq.\n"
for (nuc in counts)
  printf "%s\t%d\t%f\n", nuc, counts[nuc], counts[nuc]/t
```

1.138 We run `cres.awk` and find that the nucleotide frequencies are far removed from one quarter; the genome of *M. genitalium* is very A/T rich.

⟨*cli*⟩+≡
```
awk -f cres.awk mgGenome.fasta
```

```
Total: 580076
Res.    Count   Freq.
A       200544  0.345720
C       91515   0.157764
G       92306   0.159127
T       195711  0.337389
```

1.139 We run `cres.awk` on the proteome, cut off the first two lines of the output, and sort by amino acid frequencies. This tells us cysteine is the least frequent amino acid with roughly 1%, leucine the most frequent with roughly 11%. Both frequencies are quite far from 5%, the frequency we'd get if the twenty amino acids were uniformly distributed.

⟨*cli*⟩+≡
```
awk -f cres.awk mgProteome.fasta | tail -n +3 | sort -k 3 -n
```

```
C       1446    0.008238
...
L       18730   0.106704
```

Chapter 2
Optimal Alignment

2.1 Keeping Score

2.1 We change into the chapter directory and make the directory for this chapter, `2`. We change into that directory, make the directory for this section, `1`, and change into that. Don't forget to make your life easier by using TAB to complete the names of the directories when you change into them.

⟨*cli*⟩≡
```
cd $BEB/ch/
mkdir 2
cd 2/
mkdir 1
cd 1/
```

2.2 We print `ACCGT` into `s1.fasta` and `ACCCT` into `s2.fasta` before we align the two sequences.

⟨*cli*⟩+≡
```
printf ">s1\nACCGT\n" > s1.fasta
printf ">s2\nACCCT\n" > s2.fasta
al s1.fasta s2.fasta
```

The alignment score is 1, which makes sense as match is 1 and mismatch -3, so we have $4 - 3 = 1$.

```
Query   s1 (5 residues)
Subject s2 (5 residues)
Score   1
Error   1 (0 gaps, 1 mismatch)

Query   1 ACCGT 5
            ||| |
Subject 1 ACCCT 5
```


B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_9

2.3 We print `ATCGTA` into `s3.fasta` and `ACCT` into `s4.fasta`. Then we align the two sequences.

⟨*cli*⟩+≡
```
printf ">s3\nATCGTA\n" > s3.fasta
printf ">s4\nACCT\n"   > s4.fasta
al s3.fasta s4.fasta
```

Our result is not the true alignment in Fig. 2.1B, as that has only two matches, while the one picked by `al` has three. So the historically wrong alignment can have the greater score.

```
Query   s3 (6 residues)
Subject s4 (4 residues)
Score   -10
Errors  3 (2 gaps, 1 mismatch)


Query   1 ATCGTA 6
            | | |
Subject 1 ACC-T- 4
```

2.4 We saw that the alignment

```
Query   1 ATCGTA 6
            | | |
Subject 1 ACC-T- 4
```

has score -10. With three matches (1), one mismatch (-3), $g_o = -5$, and $g_e = -2$, this means `al` scores gaps as

$$g(l) = g_o + (l-1)g_e$$

2.5 Our version of `gapScore.awk` is

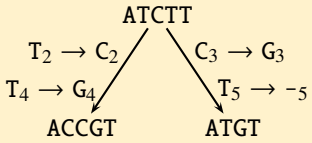
Prog. 2.8 (`gapScore.awk`)

⟨*gapScore.awk*⟩≡
```
BEGIN{
  go = -5
  ge = -2
  print 0, 0
  for (l = 1; l <= 5; l++)
    print l, go + (l-1) * ge
}
```

We run the program and plot the result.

⟨*cli*⟩+≡
```
awk -f gapScore.awk | plotLine -x l -y "g(l)"
```

2.6 Our solution contains three mutations and one deletion. It is one solution out of many:



2.7 We spontaneously thought these two sequences should be aligned as

```
ACAGTTC
--AGTTC
```

However, `al` gives us

```
ACAGTTC
A--GTTC
```

which has the same score. Alignments with the same score, or co-optimal alignments, cannot be distinguished with `al`. It just picks one of them.

2.8 This time we get the alignment we originally favored,

```
ACAGTTC
--AGTTC
```

However, its score of 5 reflects only the matches, the two flanking gaps are ignored.

2.9 We copy `hbb1.fasta` and `hbb2.fasta`, and print their residue counts together with their header lines.

⟨*cli*⟩+≡

```
cp $BEB/data/hbb*.fasta .
cres -s hbb*.fasta | grep '^>'
```

We see that `hbb1.fasta` contains the 628 nucleotides of the mRNA encoding the β-globin subunit of human hemoglobin, while `hbb2.fasta` contains the 748 nucleotides of the mRNA encoding the β-globin of *Pan troglodytes*, the chimp. The chimp sequence was *predicted* from the genome sequence rather than observed directly.

```
>NM_000518.5 Homo sapiens...(HBB), mRNA: 628
>XM_508242.4 PREDICTED: Pan troglodytes...(HBB), mRNA: 748
```

2.10 We align the two sequences to find they differ by a large indel at the 5' end and a single mutation in the rest of the sequence.

⟨*cli*⟩+≡

```
al hbb1.fasta hbb2.fasta
```

2.11 We used a line length of 20 and paged through the result with `less` to find that the mutation is at position 59 in the human sequence and at position 179 in the chimp sequence.

⟨*cli*⟩+≡
```
al -L 20 hbb1.fasta hbb2.fasta | less
```

2.12 With reading frame 3 we get the longest ORF.

⟨*cli*⟩+≡
```
translate -f 3 hbb1.fasta
```

```
>NM_000518.5 Homo sapiens...(HBB), mRNA - translated
ICF*HNCVH*QPQTDTMVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFE
SFGDLSTPDAVMGNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENF
RLLGNVLVCVLAHHFGKEFTPPVQAAYQKVVAGVANALAHKYH*ARFLAVQFLLKVPLFP
KSNY*TGGYYEGP*ASGFCLIKNIYFHC
```

2.13 The first `M` is the start, it is at position 17. The next downstream asterisk (*) marks the end of the protein, it is at position 164. So, β-globin covers interval 17–164.

⟨*cli*⟩+≡
```
translate -f 3 hbb1.fasta | keyMat M
translate -f 3 hbb1.fasta | keyMat '*'
```

2.14 The mutation is at nucleotide position 59. The protein starts in frame 3 at amino acid 17, so the starting nucleotide of the protein is $16 \times 3 + 2 = 51$. This is upstream of the mutation, which is hence inside the CDS.

2.15 We cut out the protein sequence without the stop symbol and save it to `hbb1p.fasta`.

⟨*cli*⟩+≡
```
translate -f 3 hbb1.fasta | cutSeq -r 17-163 > hbb1p.fasta
```

2.16 The third reading frame gives the longest ORF.

⟨*cli*⟩+≡
```
translate -f 3 hbb2.fasta
```

```
>XM_508242.4 PREDICTED: Pan troglodytes...(HBB), mRNA - tr...
IT*TSPCGATP*GWPIYSQEQGGQEPGLGIKVRAEPSIAYICF*HNCVH*QPQTDTMVHL
TPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKVKAH
GKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFGKEFT
PPVQAAYQKVVAGVANALAHKYH*ARFLAVQFLLKVPLFPKSNY*TGGYYEGP*ASGFCL
IKNIYFHC
```

It stretches from the first `M` in position 57 to the first downstream stop in position 204.

⟨*cli*⟩+≡
```
translate -f 3 hbb2.fasta | keyMat M
translate -f 3 hbb2.fasta | keyMat '*'
```

So the chimp β-globin lies between amino acids 57 and 203.

⟨*cli*⟩+≡
```
translate -f 3 hbb2.fasta | cutSeq -r 57-203 > hbb2p.fasta
```

2.17 PAM70 is symmetrical, because the score of a pair of amino acids is independent of their order; the pair `WR`, for example, has the same score as the pair `RW`.

2.18 We copy `pam70.txt` to our current directory and count its 25 residues, which means the score matrix has $25^2 = 625$ entries.

⟨*cli*⟩+≡
```
cp $BEB/data/pam70.txt .
head -n 2 pam70.txt | awk '!/^#/{print NF, NF*NF}'
```

The five extra residues are listed to the right of the canonical 20. They consist of four ambiguity codes and the stop codon.

- `B` for aspartate or asparagine
- `J` for leucine or isoleucine
- `Z` for glutamate or glutamine
- `X` any amino acid
- * translation of stop codons

2.19 The scores for the 20 canonical amino acids are contained in lines 3–22 of `pam70.txt`. In our script we print the entries on the main diagonal of these lines, and sort them. We find the smallest match score is 5, the largest 13. You can find these values in Fig. 2.3, too.

⟨*cli*⟩+≡
```
awk 'NR>2 && NR<23 {print $(NR-1)}' pam70.txt | sort -n
```

2.20 We store the sorted mismatch scores in the file `mm.dat` and find that -11 is the smallest and 4 the largest.

⟨*cli*⟩+≡
```
awk 'NR>2 && NR<23 {for(i=2;i<NR-1;i++)print $i}' pam70.txt |
    sort -n > mm.dat
head -n 1 mm.dat
tail -n 1 mm.dat
```

So we plot the distribution of mismatch values as a histogram with $4 - -11 = 15$ bins to get Fig. 2.24.

⟨*cli*⟩+≡
```
histogram -b 15 mm.dat | plotLine -x Score -y Count
```

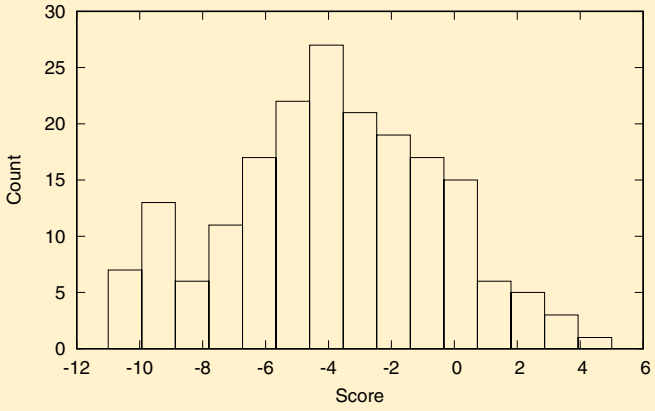


Fig. 2.24 The distribution of mismatch scores in PAM70

2.21 The `-m` switch of `al` takes a substitution matrix as argument. The alignment shows that the chimp and human β-globin sequences are identical. Thus the two codons generated by the mutation encode the same amino acid, the mutation is *synonymous*.

⟨*cli*⟩+≡

```
al -m pam70.txt hbb1p.fasta hbb2p.fasta
```

2.22 The smallest number of codons per amino acid is also the smallest possible, one; methionine (`M`) and tryptophane (`W`) are encoded by single codons. The largest number of codons per amino acid is six; leucine (`L`), serine (`S`), and arginine (`R`) are each encoded by six codons.

2.23 Leucine (`L`) is encoded by `TT[AG]` and `CT[TCAG]`. So the two codons `TT[AG]` can mutate at their first position to `CT[AG]` without amino acid change. There are no synonymous mutations at the second codon position.

2.24 Phenylalanine is encoded by `TT[TC]`. It can mutate in one step into, for example, leucine (`L`), as its six codons differ either at the first or the third position. Phenylalanine is two mutations removed from histidine (`H`) encoded by `CA[TC]`; and it is three mutations removed from glutamine (`Q`) encoded by `CA[AG]`.

2.25 The least polar amino acid is leucine, the most polar aspartate.

2.26 We copy the polarity data. To calculate the average polarity from it, we exclude the header lines, sum and count the polarities, and divide the sum by the count. The average polarity of amino acids is 8.325.

⟨*cli*⟩+≡

```
cp $BEB/data/polarity.dat .
```

```
awk '!/^#/{s+=$2;c++}END{print s/c}' polarity.dat
```

2.27 The script produces the plot of the amino acids along an axis of polarity shown in Fig. 2.25. The input to the plot consists of three columns of data, x, y, and group—in this case the amino acid names. The y range is scaled such that the legend fits (`-Y`). Since this is a one-dimensional plot, the y axis is meaningless, so it is unset (`-u`). We plot only points (`-P`) and label the x axis "Polarity" (`-x`). The colors in your plot will be the default eight colors that `gnuplot` rotates through. In Fig. 2.25 we replaced them with the polarity colors in Fig. 2.5.

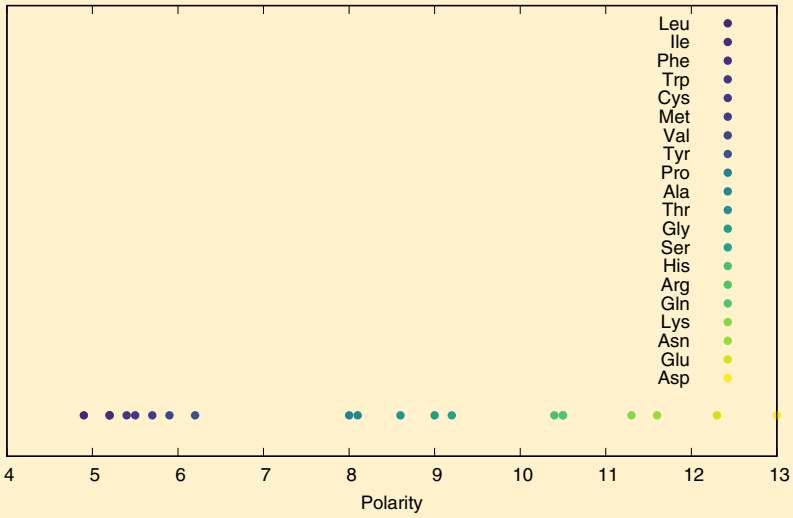


Fig. 2.25 Amino acid polarities

2.28 This is a bit hard to tell because one-step mutations can lead to highly diverse jumps in the codon table. A mutation in the first position corresponds to vertical jumps by four, eight, or twelve cells. A mutation in the second codon position corresponds to horizontal jumps ranging from one to three cells. A mutation in the third position corresponds to vertical jumps also ranging from one to three cells. In other words, the code table does not depict amino acids in mutational space. This makes it difficult to judge whether similar amino acids are in fact mutational neighbors.

2.29 We run `geco` and extract the d value for the natural code, which is 6.138.

⟨*cli*⟩+≡

```
geco polarity.dat | grep d
```

2.30 We write a loop to compute

$$20! = 2432902008176640000 \approx 2.4 \times 10^{18}.$$

The approximation is found by calculating that $log(20!)/log(10) \approx 18.4$, which means 20! has 19 digits.

⟨*cli*⟩+≡
```
awk 'BEGIN{p=1;for(i=2;i<=20;i++)p*=i;print p, p/10^18}'
```

To keep 20 books in order, let alone a whole library, is no mean feat...

2.31 We extract the d values from the simulation and plot them with the arrow and an expanded x range to get Fig. 2.7.

⟨*cli*⟩+≡
```
geco -n 10000 polarity.dat | grep ^d | awk '{print $2}' |
    histogram | plotLine -x d -y Count \
                -g "set arrow from 6.138,200 to 6.138,0" \
                -X 5:17
```

The d value of the natural code, 6.138, is unusually small.

2.32 We ran the simulation with one million iterations to find that the deviation of the natural code from random with respect to polarity is highly significant, $P = 2.5 \times 10^{-5}$.

⟨*cli*⟩+≡
```
geco -n 1000000 polarity.dat |
    grep ^d |
    awk '$2<=6.138{c++}END{print c/NR}'
```

2.33 We have summarized our results in Table 2.2. The genetic code is optimized with respect to polarity and—to a lesser extent—hydropathy. This is not surprising as polarity and hydropathy are closely related, polar molecules are hydrophilic, non-polar molecules hydrophobic. But there is no optimization with respect to volume or charge.

Table 2.2 Testing code optimization with respect to polarity, hydropathy, volume, and charge

Attribute	d	P
Polarity	6.138	2.5×10^{-5}
Hydropathy	9.394	7.9×10^{-3}
Volume	2266	0.11
Charge	4.728	0.56

2.34 We get the two proteins, align them with the PAM70 matrix we've already got handy and look at the first four lines of output. This tells us the %-mismatch is $36/3224 \times 100 \approx 1\%$. In other words, we can think of 1 PAM as the divergence time between human and chimp, roughly 5 million years.

⟨*cli*⟩+≡
```
getSeq P49792 $BEB/data/uniprot_sprot.fasta > human.fasta
getSeq H2QII6 $BEB/data/uniprot_sprot.fasta > chimp.fasta
al -m pam70.txt human.fasta chimp.fasta | head -n 4
```

2.35 We copy the file with the mutation probabilities.

⟨*cli*⟩+≡
```
cp $BEB/data/pam1.txt .
```

It tells us the probability that alanine has mutated into serine after 1 PAM is 0.28%, the probability that serine has mutated into alanine during the same time interval is 0.35%. These probabilities are quite small, as only roughly five million years have elapsed.

2.36 The data occupies the last 20 lines of the file, which we extract with `tail`. Then we compute the average match probability of the entries on the main diagonal, 0.99023. This corresponds to $(1 - 0.99023) \times 100 \approx 1\%$, the unit of time we are considering.

⟨*cli*⟩+≡
```
tail -n 20 pam1.txt | awk '{s+=$(NR+1)}END{print s/NR}'
```

2.37 We calculate, for example, $n = 2$, to find roughly 2% mismatch.

⟨*cli*⟩+≡
```
pam -n 2 pam1.txt | tail -n 20 |
    awk '{s+=$(NR+1)}END{print (1-s/NR)*100}'
```

```
1.941
```

As we increase n, we find that the percent mismatch diverges more and more from the percent accepted mutations (Table 2.3). This makes sense: The percent mismatch cannot exceed 100%, while the number of mutations has no upper bound.

Table 2.3 The percent mismatch as a function of the percent accepted mutations, PAMn

PAMn	%-Mismatch
1	1.0
2	2.0
5	4.8
10	9.2
20	17.2
50	36.1
100	55.7

2.38 We write the script `pm.sh` and just sample every tenth point along the x axis.

Prog. 2.9 (pm.sh)

⟨*pm.sh*⟩≡
```
for a in $(seq 1 10 $1)
do
    printf "%s\t" $a
    pam -n $a pam1.txt |
          tail -n 20 |
          awk '{s+=$(NR+1)}END{print (1-s/NR)*100}'
done
```

We run the script for $n = 1, ..., 1000$ and pipe the result through `plotLine`.

⟨*cli*⟩+≡
```
bash pm.sh 1000 | plotLine -x PAMn -y %-Mismatch
```

The percent mismatch in Fig. 2.8 seems to approach a limit of $(1-1/20)\times 100 = 95\%$, which again makes sense, given that there are 20 amino acids.

2.39 We copy the frequency file, cut the header from the frequency table, and sort its entries. We find that tryptophane (`W`) is the least frequent amino acid, glycine (`G`) the most frequent.

⟨*cli*⟩+≡
```
cp $BEB/data/aa.txt .
tail -n 20 aa.txt | sort -k 2 -n aa.txt
```

```
W 0.010
...
G 0.089
```

2.40 We write the entries on the main diagonal to the file `matchProb.txt`. We also extract the frequencies and write them to the file `freq.txt`. Then we paste the two data columns together, divide match by background probabilities, and sort numerically.

⟨*cli*⟩+≡
```
pam -n 70 pam1.txt | tail -n +2 |
    awk '{print $1,$(NR+1)}' > matchProb.txt
tail -n 20 aa.txt > freq.txt
paste matchProb.txt freq.txt | awk '{print $1, $2/$4}' |
    sort -k 2 -n
```

We find that tryptophane (`W`) is the most conserved amino acid by quite some margin. The reason for this might be that tryptophane is the only amino acid with two rings (Fig. 2.5). Any mutation to another amino acid might therefore disrupt the protein structure to an unusual degree and be selected against.

```
A 4.9069
S 5.17
...
```

```
M 27.9667
W 84.57
```

2.41 We apply the division by background frequency to all 400 entries in the probability matrix using `pam`. Then we extract the 20 rows of the probability matrix proper, print its diagonal elements, and sort them.

⟨*cli*⟩+≡

```
pam -n 70 pam1.txt | pam -a aa.txt | tail -n 20 |
    awk '{print $1, $(NR+1)}' | sort -k 2 -n
```

The results are identical to what we just got with slightly less automation.

```
A 4.9069
S 5.1700
...
M 27.9667
W 84.5700
```

2.42 We calculate the log base two of the odds ratio for tryptophane and multiply by two to get the half bits.

⟨*cli*⟩+≡

```
echo 'l(84.57)/l(2)*2' | bc -l
```

```
12.804148147033706534 26
```

Rounding this to the nearest integer gives a match score for tryptophane of 13.

2.43 We calculate PAM140 and find that the match score of tryptophane is now 12.

⟨*cli*⟩+≡

```
pam -n 140 pam1.txt | pam -a aa.txt | pam
```

2.2 Construction

2.44 We change into the directory for our current chapter, make a directory for the new section, and change into it.

⟨*cli*⟩≡

```
cd $BEB/ch/2/
mkdir 2
cd 2/
```

2.45 There are three possible alignments of two sequences of length one.

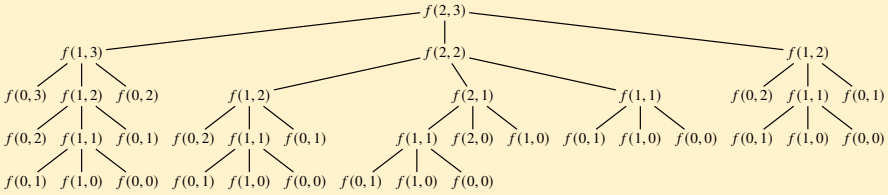
```
A   -   A
-   T   T
```

In case you were tempted to also align a gap with a gap, please don't, gaps differ fundamentally from residues—gaps aren't part of the data—and cannot be aligned with each other.

2.46 We apply equations (2.1) and (2.2) to find there are five possible alignments between two sequences of lengths 1 and 2.

$$\begin{aligned} f(2,1) &= f(1,1) + f(2,0) + f(1,0) \\ &= 3 + 1 + 1 \\ &= 5 \end{aligned}$$

2.47 We complete the recursion tree.



By counting its leaves, we find there are 25 possible alignments between sequences of lengths 2 and 3.

2.48 The number of leaves on that tree is 63, so there are 63 possible alignments of two sequences length 3.

2.49 On our computer, length 4 was still ok, length 5 barely visible, and length 6 took longer than we cared to wait, so we aborted with `C-c`.

2.50 We ran the iteration for 15 steps and saved the results in `numAl.out`. Then we extracted the data from `numAl.out` for plotting Fig. 2.26A.

⟨*cli*⟩+≡

```
for a in $(seq 15); do numAl -t $a $a; done > numAl.out
awk '{print NR, $(NF-1)}' numAl.out | tr -d '(' |
    plotLine -x Length -y "Time (s)"
```

Fig. 2.26A looks like an exponential function. So we plot it again with a logarithmic y axis to get Fig. 2.26B. Its slope still increases along the x axis, so we are dealing with more than exponential growth, we can call it hyperexponential growth.

⟨*cli*⟩+≡

```
awk '{print NR, $(NF-1)}' numAl.out | tr -d '(' |
    plotLine -x Length -y "Time (s)" -l y \
                -g "set format y '%.0e'"
```

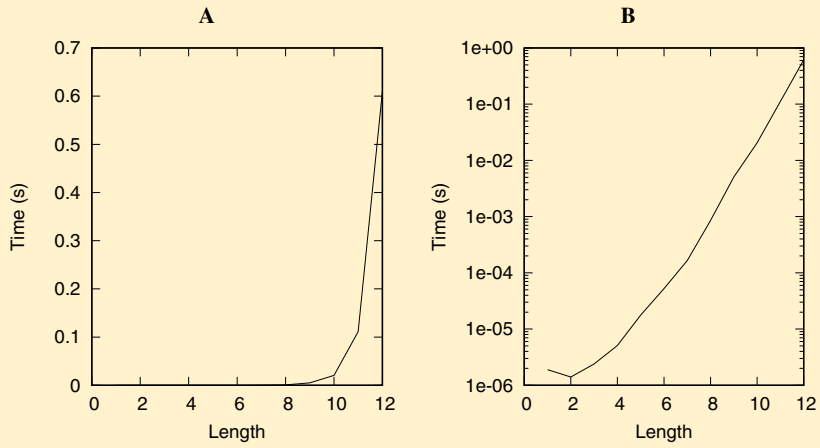


Fig. 2.26 Linear (**A**) and logarithmic (**B**) plot of the run time of top down computation of the number of possible alignments as a function of sequence length

2.51 On our machine, we measured 111.34 s for 4.46×10^{10} alignments, so roughly 2.5×10^{-9} s, or 2.5 nanoseconds, per alignment. This is best seen when we print the output of `bc` in engineering format.

⟨*cli*⟩+≡
```
echo '111.34 / 4.46 / 10^10' | bc -l |
    awk '{printf "%e\n", $1}'
```

```
2.496413e-09
```

2.52 There are roughly 2×10^{75} possible alignments between two sequences length 100.

⟨*cli*⟩+≡
```
numAl 100 100
```

```
f(100, 100) = 2.053716830872416e+75 (0.0001451 s)
```

2.53 Since it takes roughly 2.5 nanoseconds per alignment, it would take 5×10^{66} s, or

$$5 \times 10^{64}/3600/24/365.25 \approx 1.6 \times 10^{57} \text{ years.}$$

⟨*cli*⟩+≡
```
echo '5 * 10^64 / 3600 / 24 / 365.25' | bc -l |
    awk '{printf "%e\n", $1}'
```

```
1.584404e+57
```

This dwarfs the age of the universe of roughly 10^{10} years. A good algorithm can make the difference between immediately and never.

2.54 We fill in the table.

	0	1	2	3
0	1	1	1	1
1	1	3	5	7
2	1	5	13	25
3	1	7	25	63

The answer is in the bottom right hand corner, 63. We already knew this from counting the leaves in the recursion tree in Fig. 2.9, but it's much easier this way.

2.55 We run `numAl` to find there are 1,462,563 possible alignments between two sequences of length 9, roughly 1.5 million (Fig. 2.27).

⟨*cli*⟩+≡
```
numAl -p 9 9
```

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	3	5	7	9	11	13	15	17	19
2	1	5	13	25	41	61	85	113	145	181
3	1	7	25	63	129	231	377	575	833	1159
4	1	9	41	129	321	681	1289	2241	3649	5641
5	1	11	61	231	681	1683	3653	7183	13073	22363
6	1	13	85	377	1289	3653	8989	19825	40081	75517
7	1	15	113	575	2241	7183	19825	48639	108545	224143
8	1	17	145	833	3649	13073	40081	108545	265729	598417
9	1	19	181	1159	5641	22363	75517	224143	598417	1462563

Fig. 2.27 The programming matrix for computing the number of possible alignments between two sequences of length 9

2.56 A bit of trial and error shows that the longest pair of sequences for which `numAl` gives a result has length 404.

⟨*cli*⟩+≡
```
numAl 404 404
```

For a pair of sequences length 405 the number of possible alignments is given as infinity, which is, of course, a bit of an overstatement.

2.57 We typed the plot in Fig. 2.28 into our editor. Notice the two forward off-diagonals for the repeat `GATATA` and the ascending line for reading `ATATAGATATA` backward.

2.58 We looked up the definition of `split` in the `man` page for `awk`. Then we split the query and subject into the arrays `qa` and `sa` and saved their lengths in `m` and `n`.

⟨*Split query and subject, Prog. 2.1*⟩≡
```
m = split(q, qa, "")
n = split(s, sa, "")
```

```
  G A T A T A G A T A T A
G * *
A   *   *   *   *   *   *
T     *   *       *   *
A   *   *   *   *   *   *
T     *   *       *   *
A   *   *   *   *   *   *
G * *
A   *   *   *   *   *   *
T     *   *       *   *
A   *   *   *   *   *   *
T     *   *       *   *
A   *   *   *   *   *   *
```

Fig. 2.28 Dot plot comparing `GATATAGATATA` to itself

2.59 We begin with printing a single blank. Then we iterate over the subject array and for each nucleotide print a blank followed by the nucleotide. We close the line with a newline character.

⟨*Print header of dot matrix, Prog. 2.1*⟩≡

```
printf " "
for (i = 1; i <= n; i++)
  printf " %c", sa[i]
printf "\n"
```

2.60 For the rest of the dot plot, we start every row with a nucleotide, followed by a blank and a dot for match or a second blank for mismatch.

⟨*Print rest of dot matrix, Prog. 2.1*⟩≡

```
for (i = 1; i <= m; i++) {
  printf "%c", qa[i]
  for (j = 1; j <= n; j++) {
    c = " "
    if (qa[i] == sa[j])
        c = "*"
    printf " %c", c
  }
  printf "\n"
}
```

2.61 We run our pipeline to print the two sequences, find their repeats, convert the repeats to segments and plot the segments to get Fig. 2.29 containing all matches at least two bases long. Segments of length 1 are ignored by `plotSeg`.

⟨*cli*⟩+≡

```
printf ">s1\nATATTACTAT\n>s2\nATATTACTAT\n" |
    repeater -m 1 -p | rep2plot | plotSeg
```

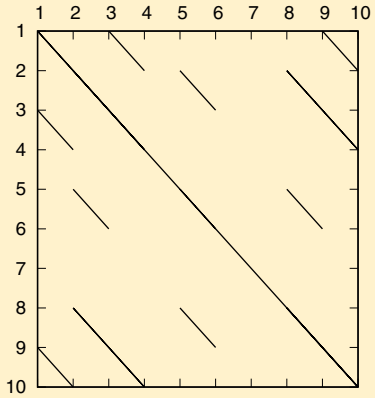


Fig. 2.29 Match plot of ATATTACTAT with itself; compare to the corresponding dot plot in Fig. 2.11

2.62 We copy the two FASTA files to our current working directory. As we increase the minimum match length, the smallest that gives a clean plot is 10. In Fig. 2.30 we draw an arrow to mark the tiny blank corresponding to the synonymous mutation we found earlier at position 59 in the human sequence. Notice the two backslashes in the `plotSeg` command, which continue the line across the carriage returns.

⟨*cli*⟩+≡
```
cp $BEB/data/hbb1.fasta .
cp $BEB/data/hbb2.fasta .
cat hbb*.fasta | repeater -m 10 -p | rep2plot |
    plotSeg -x "Human beta-globin" \
              -y "Chimp beta-globin" \
              -g "set arrow from 59,300 to 59,200"
```

2.63 Print two sequences of different lengths to find out that the first sequence is written along the x axis, the second along the y axis.

2.64 The jump in the matches highlights a region in *D. melanogaster* that isn't present in *D. guanche*. So the transposon inserted into the *Adh* region of *D. melanogaster*.

2.65 We copy `d[mg]Adh*.fasta`. Then we extend our pipeline of `repeater`, `rep2plot`, and `plotSeg` by converting from bp to kb to get Fig. 2.13.

⟨*cli*⟩+≡
```
cp $BEB/data/d[mg]Adh*.fasta .
cat dmAdhAdhdup.fasta dgAdhAdhdup.fasta |
    repeater -m 12 -p | rep2plot |
    awk '{f=1000; print $1/f, $2/f, $3/f, $4/f}' |
    plotSeg -x "D. melanogaster (kb)" -y "D. guanche (kb)"
```

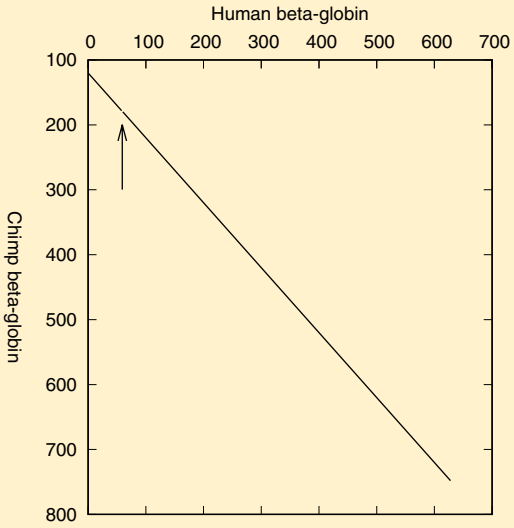


Fig. 2.30 Match plot of human and chimp β-globin; the arrow points to the single mismatch

2.66 We couldn't see any difference by eye, so we saved the forward matches to the file `adhF.txt` and the forward and reverse matches to file `adhR.txt`. Then we compared `adhF.txt` and `adhR.txt` with `diff`.

⟨*cli*⟩+≡

```
cat dmAdhAdhdup.fasta dgAdhAdhdup.fasta |
    repeater -m 12 -p | rep2plot > adhF.txt
cat dmAdhAdhdup.fasta dgAdhAdhdup.fasta |
    repeater -m 12 -p -r | rep2plot > adhR.txt
diff adhF.txt adhR.txt
```

The difference between them is a single line, so inclusion of the reverse strand results in one extra match.

```
0a1
> 99    -1836  88    -1847
```

2.67 We copy the Genbank files, `grep` for CDS, and find the entries in Table 2.1.

⟨*cli*⟩+≡

```
cp $BEB/data/*.gb .
grep CDS dm*.gb dg*.gb
```

```
dm*.gb: CDS join(2021..2119,2185..2589,2660..2926)
dm*.gb: CDS join(3226..3321,3748..4152,4204..4521)
dg*.gb: CDS join(1984..2076,2145..2549,2613..2879)
dg*.gb: CDS join(3221..3316,3540..3944,4007..4345)
```

2.68 We cut out the four CDSs using the join option of `cutSeq` and translate them to find that the stop codon is part of a CDS.

⟨*cli*⟩+≡
```
cutSeq -r 2021-2119,2185-2589,2660-2926 \
         -j dmAdhAdhdup.fasta > dmAdhCds.fasta
cutSeq -r 3226-3321,3748-4152,4204-4521 \
         -j dmAdhAdhdup.fasta > dmDupCds.fasta
cutSeq -r 1984-2076,2145-2549,2613-2879 \
         -j dgAdhAdhdup.fasta > dgAdhCds.fasta
cutSeq -r 3221-3316,3540-3944,4007-4345 \
         -j dgAdhAdhdup.fasta > dgDupCds.fasta
translate *Cds.fasta
```

2.69 With `exex.sh` we `grep` for `CDS` and carry out three translations. The first removes all characters, parentheses, and blanks. The second converts commas into newlines, and the third converts runs of dots into single blanks.

2.70 We run `exex.sh` and print the exons as horizontal lines with y coordinates corresponding to the end of the vertical *D. guanche* sequence, which is 4433 bp long. We redirect the result into `annot.txt`.

⟨*cli*⟩+≡
```
bash exex.sh < dmAdhAdhdup.gb |
    awk '{print $1, 4433, $2, 4433}' > annot.txt
```

2.71 This time the coordinates extracted from the Genbank file are y coordinates and the x coordinate is zero. We append the new lines to `annot.txt`.

⟨*cli*⟩+≡
```
bash exex.sh < dgAdhAdhdup.gb |
    awk '{print 0, $1, 0, $2}' >> annot.txt
```

2.72 We take the start of the first exon, the end of the third, the start of the fourth, and the end of the sixth.

⟨*Extract gene positions, Prog. 2.3*⟩≡
```
p[1] = s[1]
p[2] = e[3]
p[3] = s[4]
p[4] = e[6]
```

2.73 For the vertical lines the positions are interpreted as x coordinates.

⟨*Draw vertical lines, Prog. 2.3*⟩≡
```
for (i = 1; i <= 4; i++)
  print p[i], y1, p[i], y2
```

2.74 For the horizontal lines the positions are interpreted as y coordinates.

⟨*Draw horizontal lines, Prog. 2.3*⟩≡
```
for (i = 1; i <= 4; i++)
```

```
    print x1, p[i], x2, p[i]
```

2.75 For the vertical lines we take the *D. melanogaster* exons as input and supply the y coordinates. These extend from the end of the *D. guanche* sequence, 4433, to the start of its first exon, 1984. We print the vertical lines and append them to `annot.txt`.

⟨*cli*⟩+≡
```
bash exex.sh < dmAdhAdhdup.gb |
    awk -v y1=4433 -v y2=1984 -f lines.awk >> annot.txt
```

2.76 For the horizontal lines we take the *D. guanche* exons as input and supply the x coordinates, which extend from zero to the end of the last *D. melanogaster* exon, 4521. We draw the horizontal lines and append them to `annot.txt`.

⟨*cli*⟩+≡
```
bash exex.sh < dgAdhAdhdup.gb |
    awk -v x1=0 -v x2=4521 -f lines.awk >> annot.txt
```

2.77 We transform our two data files, `adhR.txt` and `annot.txt`, from bp to kb and pipe them through `plotSeg`. In `plotSeg`, we label the axes and adjust their ranges to get Fig. 2.14. Since the y range is reversed, the y axis label is written top to bottom rather than bottom to top as in standard graphs.

⟨*cli*⟩+≡
```
cat adhR.txt annot.txt |
    awk '{f=1000;print $1/f, $2/f, $3/f, $4/f}' |
    plotSeg -x "D. melanogaster" -y "D. guanche" \
              -X "-0.2:4.7" -Y "4.7:0"
```

2.78 As can be seen in Fig. 2.14, the transposon inserted into an intron of *Adh-dup* in *D. melanogaster*.

2.79 Only the orthologs show up in Fig. 2.13. The gene duplication leading to the paralogs is so old that mutation has erased any homology detectable by our match plot.

2.80 Our script `ranAdh.sh` reads the number of iterations and the minimum repeat length from the command line.

Prog. 2.10 (`ranAdh.sh`)

⟨*ranAdh.sh*⟩≡
```
for a in $(seq $1)
do
    randomizeSeq dgAdhAdhdup.fasta > r.fasta
    cat r.fasta dmAdhAdhdup.fasta |
          repeater -p -r -m $2 |
          rep2plot |
          wc -l
done
```

We ran the script with 100 iterations and repeat length 12, and counted the random matches. There were 2.4 on average

⟨*cli*⟩+≡
```
bash ranAdh.sh 100 12 | awk '{s+=$1;c++}END{print s/c}'
```

However, when you look at random plots, the matches are usually not on the main diagonal. So there can be no doubt that the ensemble of matches on the main diagonal in Fig. 2.13 is due to homology. But perhaps we shouldn't place much weight on any individual match of length 12 between our two *Adh* sequences.

2.81 We apply the gap scoring scheme to fill in the first column of the alignment matrix.

	-	A	C	G
-	0			
A	-1			
G	-2			

2.82 We extend the preceding cells by gaps and leave the corresponding arrows pointing backwards.

	-	A	C	G
-	0	← −1	← −2	← −3
A	↑ −1			
G	↑ −2			

2.83 We fill in the remaining six cells and note the back pointers.

	-	A	C	G
-	0	← −1	← −2	← −3
A	↑ −1	↖ 1	← 0	← −1
G	↑ −2	↑ 0	↑ −1	↖ 1

2.84 We store the sequences in files and align them with score printing. The score scheme is $g_o = g_e = -1$ (`-p`, `-e`), mismatch -1 (`-i`), and match is left to its default 1.

⟨*cli*⟩+≡
```
printf ">s1\nAG\n"  > q.fasta
printf ">s2\nACG\n" > s.fasta
al -P s -p -1 -e -1 -i -1 q.fasta s.fasta
```

```
   -   A   C   G
-  0 <-1 <-2 <-3
A ^-1  \1  <0 <-1
G ^-2  ^0 ^-1  \1
```

2.85 We follow the alignment path to find

```
q A-G
s ACG
```

We can check the result with `al`.

⟨*cli*⟩+≡
```
al -p -1 -e -1 -i -1 q.fasta s.fasta
```

2.86 We start the trace-back at the bottom right corner and move diagonally, which means we get C/C. This is repeated twice to give TCC/TCC. Then we move horizontally twice to get --TCC/GGTCC. Five more diagonal moves give the final alignment

```
q TACAG--TCC
s TTCAGGGTCC
```

Its score is $7 - 1 - 2 = 4$, as stated in the bottom right hand corner of the matrix.

2.87 We transform the coordinates of the trace-back to kb, convert them to segments, and pipe them through `plotSeg`. Conversion to segments is achieved by printing every other line with a terminal blank instead of a carriage return.

⟨*cli*⟩+≡
```
al -P t dgAdhAdhdup.fasta dmAdhAdhdup.fasta |
    awk '{f=1000;print $1/f, $2/f}' |
    awk 'NR%2==1{printf "%s ", $0} NR%2==0' |
    plotSeg -x "D. melanogaster (kb)" -y "D. guanche (kb)"
```

2.88 Trace-back along the first row or column can only add gaps to the alignment. Gaps can only decrease the score. To avoid this, local alignment matrices have their first rows and columns filled with stop signals, zeros.

2.89 The optimal local alignment is a truncated version of the global alignment. Apart from the alignment itself, we also note its coordinates.

```
q 3 CAG--TCC  8
s 3 CAGGGTCC 10
```

2.90 The second best non-redundant alignment is

```
q 6 TC 7
s 2 TC 3
```

The query segment was already part of the best alignment, but not the subject segment, which makes this a new alignment. This is the situation sketched in Fig. 2.19, where a fragment in the top sequence is homologous to two fragments in the bottom sequence.

2.91 We align the two sequences using the local algorithm.

⟨*cli*⟩+≡
```
al -l dmAdhAdhdup.fasta dgAdhAdhdup.fasta
```

The result covers the second exon of *Adh*, which is best seen when we tabulate our results.

Organism	Alignment *Adh* exon 2
D. melanogaster	2182–2594 2185–2589
D. guanche	2142–2554 2145–2549

It seems as if conservation stretches a tiny bit into both flanking introns.

2.92 We calculate the two best alignments.

⟨*cli*⟩+≡
```
al -l -n 2 dmAdhAdhdup.fasta dgAdhAdhdup.fasta
```

This time, the second exon of *Adh-dup* is picked, but only the 3' intron-exon junction is fully conserved.

Organism	Alignment *Adh-dup* exon 2
D. melanogaster	3829–4158 3748–4152
D. guanche	3621–3950 3540–3944

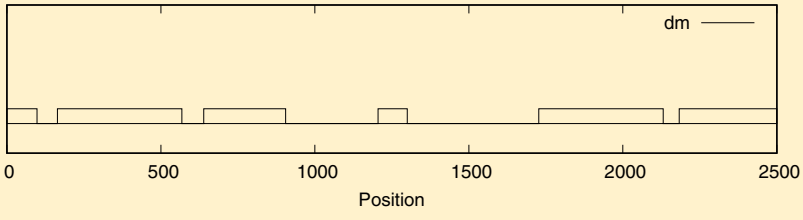
2.3 Application

2.93 We change into the directory for Chapter 2, make the directory for Section 3, and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/2/
mkdir 3
cd 3/
```

2.94 We copy the script `exex.sh` and the *Adh* Genbank files to our current directory. Then we extract the exon coordinates for *D. melanogaster*, remove their offset to make them start at zero, place them on the plus strand, and draw them as genes. The "genes" are marked `dm`, saved in `cds.dat`, and plotted with nice dimensions.

⟨*cli*⟩+≡
```
cp ../2/*.gb ../2/exex.sh .
bash exex.sh < dmAdhAdhdup.gb |
    awk '{o=2021; print $1-o, $2-o, "+"}' | drawGenes |
    awk '{print $0, "dm"}' > cds.dat
plotLine -Y -2:8 -u y -x Position cds.dat
```



2.95 We repeat the exon extraction for *D. guanche*, only this time we draw the exons on the negative strand and append the results marked `dg` to `cds.dat`. By applying `plotLine` to `cds.dat` we reproduce Fig. 2.21.

⟨*cli*⟩+≡
```
bash exex.sh < dgAdhAdhdup.gb |
    awk '{o=1984; print $1-o, $2-o, "-"}' | drawGenes |
    awk '{print $0, "dg"}' >> cds.dat
plotLine -Y -2:8 -u y -x Position cds.dat
```

2.96 We copy the FASTA files into our working directory and cut the sequences along the coordinates given in Table 2.1.

⟨*cli*⟩+≡
```
cp $BEB/data/d[gm]Adh*.fasta .
cutSeq -r 2185-2589 dmAdhAdhdup.fasta > dmAdhE2.fasta
cutSeq -r 3748-4152 dmAdhAdhdup.fasta > dmDupE2.fasta
cutSeq -r 2145-2549 dgAdhAdhdup.fasta > dgAdhE2.fasta
cutSeq -r 3540-3944 dgAdhAdhdup.fasta > dgDupE2.fasta
```

2.97 We apply `cres` to the four sequences separately (`-s`) to find they are all 405 bp long.

⟨*cli*⟩+≡
```
cres -s *E2.*
```

2.98 We print the sequences compared, followed by the score of their alignment.

⟨*Calculate score, Prog. 2.4*⟩≡
```
printf "%s %s " ${i} ${j}
al d${i}*E2.* d${j}*E2.* |
    grep Sc
```

2.99 We run `scores.sh`.

⟨*cli*⟩+≡
```
bash scores.sh
```

So we can fill in our table.

	Adh_{dm}	$Adh\text{-}dup_{\text{dm}}$	Adh_{dg}	$Adh\text{-}dup_{\text{dg}}$
Adh_{dm}	—	−344	209	−305
$Adh\text{-}dup_{\text{dm}}$		—	−361	81
Adh_{dg}			—	−342
$Adh\text{-}dup_{\text{dg}}$				—

2.100 Paralogs have negative scores, orthologs positive.

2.101 We run 1000 iterations and save the scores to `ral.out`. Then we generate the histogram and plot it. We stretch the x range slightly to insert more space between the tic labels.

⟨*cli*⟩+≡
```
bash ral.sh 1000 dmAdhE2.fasta dmDupE2.fasta |
    awk '{print $2}' > ral.out
histogram ral.out |
    plotLine -x Score -y Count -X -530:-430
```

2.102 We can directly count the number of random scores greater or equal to -344.

⟨*cli*⟩+≡
```
bash ral.sh 1000 dmAdhE2.fasta dmDupE2.fasta |
    awk '$2>=-344{c++}END{print c/NR}'
```

Even in 10^4 iterations the count was zero. This is not surprising when we look at the distribution of random scores in Fig. 2.22, where the observed value of -344 is very far to its right. We can illustrate this by marking -344 on the x axis as shown in Fig. 2.31.

⟨*cli*⟩+≡
```
histogram ral.out |
    plotLine -x Score -y Count -X -540:-340 \
                -g "set arrow from -344,20 to -344,1"
```

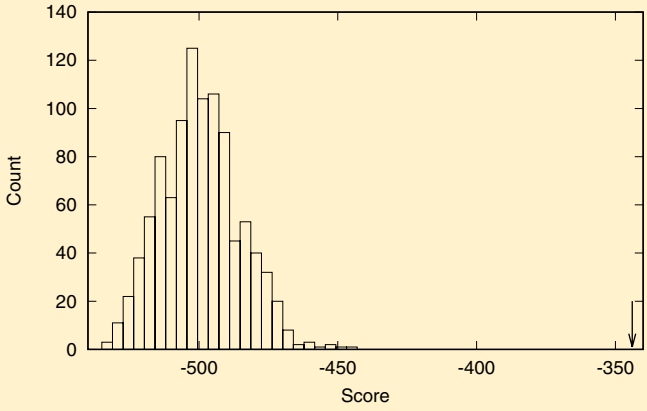


Fig. 2.31 The null distribution of alignment scores for exon 2 from Adh_{dm} and $Adh\text{-}dup_{\mathrm{dm}}$; the arrow points to the observed score

2.103 All our comparisons had $P < 10^{-3}$ and looked highly significant. Clearly, alignment is much more sensitive than exact matching when it comes to detecting homology.

2.104 We run `al` and read 49 mismatches from the "Errors" line of its output. Then we calculate $49/405 \approx 0.12$ to get the mismatches per site.

⟨*cli*⟩+≡
```
al dmAdhE2.fasta dgAdhE2.fasta | grep '^E'
echo '49/405' | bc -l
```

2.105 We align the two sequences, get the error line, and parse the number of mismatches from its fifth field. Then we divide the number of mismatches by the sequence length.

⟨*Calculate mismatches per site, Prog. 2.6*⟩≡
```
al ${i}E2.fasta ${j}E2.fasta |
    awk '/^E/{printf " %.4f", $5/405}'
```

2.106 We save the result of the mismatch computation to `mism.out` and apply `nj` to it. The resulting tree is shown in parentheses notation. Not much of a tree, so far.

⟨*cli*⟩+≡
```
bash mism.sh > mism.out
nj mism.out
```

```
(dmDup:0.0982,dgDup:0.102,(dmAdh:0.0623,dgAdh:0.0587):0.155);
```

2.107 We calculate the tree and get the plot in Fig. 2.32. This is an *unrooted tree* of our sequences.

⟨*cli*⟩+≡
```
nj mism.out | plotTree
```

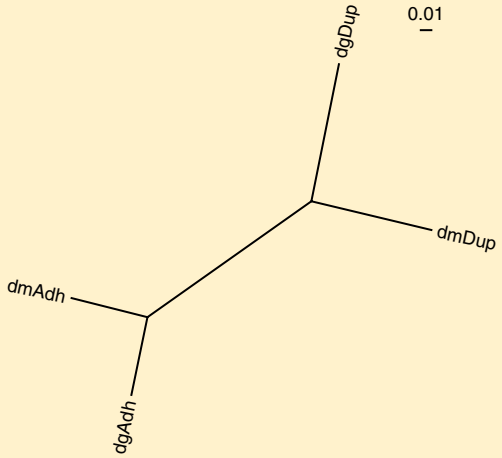


Fig. 2.32 Unrooted mismatch tree of *Drosophila Adh* exon 2

2.108 We insert `midRoot` into our pipeline.

⟨*cli*⟩+≡
```
nj mism.out | midRoot | plotTree
```

2.109 The number of mismatches per site varies, we got, 0.37, 0.36, 0.36, 0.38, and 0.36 on our first five trials, all much smaller than 0.5.

2.110 In our case we have

mismatches	mutations
0.36	0.50
0.37	0.51
0.38	0.53

⟨*cli*⟩+≡
```
for a in 0.36 0.37 0.38
do
    echo "-3/4*l(1-4/3*${a})" | bc -l
done
```

All these mutation estimates are close to 0.5. So even on sequences as short as 405 bp, the Jukes-Cantor equation works quite well.

2.111 We iterate over the mismatch values, convert them to mutations, and print them.

⟨*Calculate row of mutations, Prog. 2.7*⟩≡
```
for (i = 2; i <= NF; i++) {
  k = -3/4 * log(1 - 4/3 * $i)
  printf " %.4f", k
}
```

2.112 We convert the mismatches to mutations and draw the tree.

⟨*cli*⟩+≡
```
awk -f mut.awk mism.out |
    nj |
    midRoot |
    plotTree
```

The mismatch and mutation trees are similar but not identical.

2.113 We recall the computation of the number of mutations, or substitutions, per site.

⟨*cli*⟩+≡
```
awk -f mut.awk mism.out
```

```
4
dmAdh 0.0000 0.4190 0.1320 0.4019
dmDup 0.4190 0.0000 0.3935 0.2326
dgAdh 0.1320 0.3935 0.0000 0.4190
dgDup 0.4103 0.2326 0.4190 0.0000
```

The average number of substitutions for speciation is $(0.1320 + 0.2326)/2 = 0.1823$ and for duplication $(0.4190 + 0.4019 + 0.3935 + 0.4190)/4 \approx 0.4084$. So the duplication time is roughly $0.4084/0.1823 \times 32 \approx 72$ million years.

Chapter 3
Exact Matching

3.1 Keyword Trees

3.1 We change into the directory holding the chapters, make directory 3 for the new chapter, change into it, make directory 1, and change into that.

⟨*cli*⟩≡
```
cd $BEB/ch/
mkdir 3
cd 3/
mkdir 1
cd 1/
```

3.2 If we are not dealing with a header line, we are dealing with a data line. In that case we concatenate it to form the text, `t`.

⟨*Deal with data, Prog. 3.1*⟩≡
```
!/^>/ {
  t = t $0
}
```

3.3 We match the initial >. If we are not in the first line, we are not dealing with the first header, so we print the previous sequence and reset it.

⟨*Deal with headers, Prog. 3.1*⟩≡
```
/^>/ {
  if (NR > 1) {
    print t
    t = ""
  }
  print
}
```


B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_10

3.4 The `END` pattern just triggers printing of the sequence, `t`, constructed up to that point.

⟨*Deal with last sequence, Prog. 3.1*⟩≡
```
END {
  print t
}
```

3.5 As in `readFasta.awk`, dealing with the data means concatenating the current sequence.

⟨*Deal with data, Prog. 3.2*⟩≡
```
!/^>/ {
  t = t $0
}
```

3.6 We match the header. If it's not the first one, we call `naive` and reset the text, `t`. Then we print the header.

⟨*Deal with headers, Prog 3.2*⟩≡
```
/^>/ {
  if (NR > 1) {
    naive(p, t)
    t = ""
  }
  print
}
```

3.7 We call `naive` on the last sequence.

⟨*Deal with last sequence, Prog. 3.2*⟩≡
```
END{
  naive(p, t)
}
```

3.8 We loop over the text and over the pattern while matching their characters. If we reach the end of the pattern, we print its starting position in `t`.

⟨*Find matches, Prog. 3.2*⟩≡
```
for (i = 1; i <= n-m+1; i++) {
  for (j = 1; j <= m; j++)
    if (pa[j] != ta[i+j-1])
        break
  if (j > m)
    print i
}
```

We look for `AT` in `s.fasta` and find two matches.

⟨*cli*⟩+≡
```
awk -f naive.awk -v p=AT s.fasta
```

```
>s1
2
>s2
9
```

3.9 We generate the test sequence and search it to find the expected matches at positions 2, 6, and 8.

⟨*cli*⟩+≡

```
printf ">s3\nCACAGACACAT\n" > s3.fasta
awk -f naive.awk -v p=ACA s3.fasta
```

```
>s3
2
6
8
```

3.10 We copy `mgGenome.fasta` to our working directory and search it for `ACGTCG`. We remove the header line from the output, and find the pattern occurs once. Then we search for its slightly shorter sibling, `ACGTC`, which occurs 51 times.

⟨*cli*⟩+≡

```
cp $BEB/data/mgGenome.fasta .
awk -f naive.awk -v p=ACGTCG mgGenome.fasta | grep -v '^>' |
    wc -l
awk -f naive.awk -v p=ACGTC  mgGenome.fasta | grep -v '^>' |
    wc -l
```

3.11 We reverse complement the genome sequence and find that it contains three copies of `ACGTCG`. So `ACGTCG` is not unique in *M. genitalium*.

⟨*cli*⟩+≡

```
revComp mgGenome.fasta | awk -f naive.awk -v p=ACGTCG
```

```
>gi|84626123|gb|L43967.2| Mycoplasma genitalium G37...
88125
208713
542138
```

3.12 The real time is 37.8 s on our machine.

3.13 It took 3.57 s on our machine, roughly a tenth of the search for 20 `A`s. That's because in the previous run we had $10^6 \times 21 = 2.1 \times 10^7$ character comparisons, while now there was a mismatch in the first position, so there were roughly 20 times fewer comparisons. We don't know why this translates to 10% rather than 5% run time, but that just goes to show the importance of run time measurements.

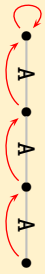
3.14 We time the search for the long pattern and the lone `G`.

⟨*cli*⟩+≡

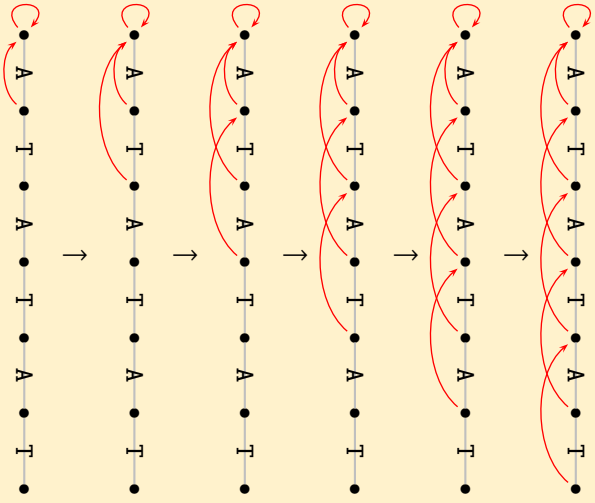
```
time naiveMatcher AAAAAAAAAAAAAAAAAAAAG ran.fasta
time naiveMatcher G ran.fasta
```

Searching for the long pattern takes 0.21 s, roughly a 180-fold increase in speed compared to the Awk program. Just searching for the `G` took 0.04 s, which again seems less than the twenty-fold speed-up expected from the number of matches. However, matching isn't the only thing going on in `naiveMatcher`, it also reads the data, allocates memory etc., aspects not taken into account by our idea that run time is proportional to the number of comparisons.

3.15 The failure links for `AAA` jump up one node each.



3.16 We initialize the links of the root and the first node down. Then we work our way down the "tree" and for every node follow the parent's failure links until we either find a match or reach the root.



3.17 Fig. 3.27 shows the keyword tree with the failure links included.

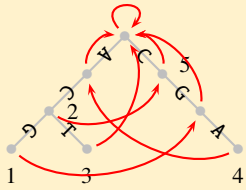


Fig. 3.27 Keyword tree with failure links for the five patterns P_1 = ACG, P_2 = AC, P_3 = ACT, P_4 = CGA, and P_5 = C

3.18 In our matching, we first find P_2 = AC and P_1 = ACG next. Then we follow the mismatch link twice to find P_5 = C. However, we have missed the first occurrence of C. So, having constructed the tree and its failure links, there is a third and last step in the construction of keyword trees. In that step we visit every node and collect the matches on the path of failure links starting at the current focal node. These matches are stored in an output set attached to the focal node.

3.19 We pipe the tree through `plotTree`.

⟨*cli*⟩+≡
```
drawKt -t ACG AC ACT CGA C | plotTree
```

3.20 We draw the keyword tree and also ask for a LaTeX wrapper. Then we follow the instructions given by `drawKt` and typeset the document, convert the resulting device-independent file to postscript, and convert the postscript to pdf. We can view the pdf with `evince` or some other pdf viewer. `evince` opens a new window to display the pdf. As usual, the ampersand (&) leaves the command line responsive while the new window is active.

⟨*cli*⟩+≡
```
drawKt -w ktWrapper.tex ACG AC ACT CGA C > kt.tex
latex ktWrapper
dvips ktWrapper
ps2pdf ktWrapper.ps
evince ktWrapper.pdf &
```

3.21 There is now a title line, an author line, and the current date. LaTeX is a typesetting system used, for example, to typeset this book. It is best described by its authors [31, 28].

3.22 We run `naiveMatcher` and `keyMat` on the short and long pattern. `keyMat` is approximately twice as fast as `naiveMatcher` on the short pattern. Doubling the pattern length doubles the run time of `naiveMatcher` but leaves that of `keyMat` unchanged.

⟨*cli*⟩+≡
```
time naiveMatcher AAAAAAAAAAAAAAAAAAAG ran.fasta
time keyMat       AAAAAAAAAAAAAAAAAAAG ran.fasta
time naiveMatcher AAAAAAAAAAAAAAAA...G ran.fasta
```

```
time keyMat       AAAAAAAAAAAAAAAAA...G ran.fasta
```

3.23 We copy the genomes of *E. coli* K12 and O157H7, cut out the query from K12, and search for it in O157H7 with a keyword tree and with naïve matching.

⟨*cli*⟩+≡

```
cp $BEB/data/ecoliK12.fasta .
cp $BEB/data/ecoliO157H7.fasta .
cutSeq -r 1-1000 ecoliK12.fasta > q.fasta
time sblast q.fasta ecoliO157H7.fasta
time sblast -n q.fasta ecoliO157H7.fasta
```

The search with keyword tree takes 0.3 s, the naïve search 60 s, that is, 200 times longer.

3.2 Suffix Trees

3.24 We change into the directory for this chapter, make a directory for the current section, and change into that. Did you tab to auto complete the path and simplify life on the command line?

⟨*cli*⟩≡

```
cd $BEB/ch/3/
mkdir 2
cd 2/
```

3.25 There are eight characters in `CATGGCAT`, so it has eight suffixes.

```
1 CATGGCAT
2  ATGGCAT
3   TGGCAT
4    GGCAT
5     GCAT
6      CAT
7       AT
8        T
```

3.26 Walk into the suffix tree to find `AT` above leaves 2 and 7; so `AT` starts in t at positions 2 and 7. `X` is nowhere, which we discover right at the root of the tree.

3.27 We add `AT$`, `T$`, and `$` to get Fig. 3.6. The branch for the sentinel, `$`, is not really necessary, as by definition it can never be searched for, but for completeness sake we also include it when drawing suffix trees.

3.28 We start suffix tree construction with the first suffix on an edge connecting the root and the first leaf and then add all other suffixes to the tree as shown in Fig. 3.7. The result is Fig. 3.28.

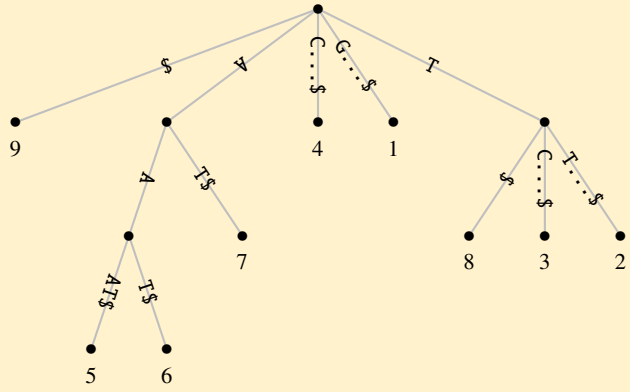


Fig. 3.28 Suffix tree for `GTTCAAAT`

3.29 There are four frontier nodes and their path labels are `AT`, `CAT`, `G`, and `T`. So the longest repeat in `CATGGCAT` is `CAT`.

3.30 We copy the genome of *M. genitalium* to our working directory. Then we run `repeater` with `-r` on its forward and reverse strand, to find the same longest repeat of 243 bp on both strands.

⟨*cli*⟩+≡
```
cp $BEB/data/mgGenome.fasta .
repeater -r mgGenome.fasta
```

3.31 We calculate the nucleotide frequencies of *M. genitalium*.

⟨*cli*⟩+≡
```
cres mgGenome.fasta
```

The frequency of `A` is 0.346, so the probability of drawing `AA` is $0.346^2 \approx 0.12$, which is quite different from the 1/4 of the equiprobable case.

3.32 We again calculate the nucleotide frequencies of *M. genitalium*, pick them from the output, and sum their squares.

⟨*cli*⟩+≡
```
cres mgGenome.fasta |
    awk '/^[ACGT]/{s+=$3*$3}END{print s}'
```

The probability of randomly picking a pair of identical nucleotides from the genome of *M. genitalium* is roughly 0.284, which isn't all that different from the 1/4 we get when assuming equal nucleotide frequencies.

3.33 We need to solve

$$1 = P_{\mathrm{m}}^{l} \times L^2$$

for l. By rearranging and taking logarithms, we get

$$l = \frac{\log(1/L^2)}{\log(P_{\mathrm{m}})}.$$

By substituting $P_m = 0.284$ and $L = 580,076$, we find that the expected longest repeat in *M. genitalium* has length $l \approx 21.1$. This is much shorter than the observed longest repeat of length 243.

⟨*cli*⟩+≡
```
echo 'l(1/580076^2)/l(0.284)' | bc -l
```

```
21.08534205633167458539
```

3.34 We pipe the randomized genome through `repeater` and make sure we count results only once by cutting them off after the first line.

⟨*cli*⟩+≡
```
for a in $(seq 100)
do
    randomizeSeq mgGenome.fasta |
          repeater |
          tail -n +2 |
          head -n 1
done | awk '{s+=$1}END{print s/NR}'
```

The average we found was 20.1, which isn't identical to 21.1, but close. One reason for the discrepancy might be that in our model all starting points in the matrix are independent of each other, which is a simplification. In any case, random longest matches are much shorter than the observed match of 243.

3.35 Without sentinel, the branch leading to leaf 9 is gone, as the sentinel is not part of the text any more. Moreover, all the additional three branches occupied by just a sentinel in Fig. 3.6 are now empty in Fig. 3.29.

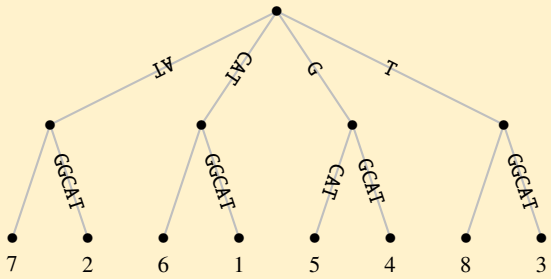


Fig. 3.29 The suffix tree for `CATGGCAT` without sentinel

3.36 When we apply naïve suffix tree construction to a sequence like `AAAA`, it displays its worst case: for every suffix we have to walk from beginning to end. This leads to a run time that is quadratic in input length, which doesn't scale well. Now, our scenario of a sequence consisting only of `A` isn't realistic. But genomes do contain long exact repeats, which have the same effect as our mononucleotide sequence. In general, we should avoid construction algorithms that might run in time quadratic in the length of their input.

3.37 We get the day of the week, the day of the month, the month, and the year. This is followed by the time.

⟨*cli*⟩+≡
```
date
```

```
Tue 14 Jun 2022 01:27:02 PM CEST
```

3.38 January 1st 1970 was 1,655,206,301, or 1.7×10^9, s ago. That's also roughly the number of heart beats for a person born around that time.

3.39 `ls` took 2,815,900 ns on our computer, roughly 3 ms.

⟨*cli*⟩+≡
```
echo '198105600-195289700' | bc
```

3.40 We repeat the code for mononucleotides without the conversion to all `A`. This time we tag with `rand`.

⟨*Measure time for random sequence, Prog. 3.3*⟩≡
```
st=$(date +%s.%N)
repeater r.fasta > /dev/null
en=$(date +%s.%N)
rt=$(echo "$en - $st" | bc -l)
echo $a $rt "rand"
```

3.41 We run `repeater.sh`, save its output and plot it with `plotLine` using lines and points. The quotes around the axis labels are required to turn the multiple strings separated by blanks into single strings. The `-L` option gives lines and points, `-P` would be only points. If in doubt, get help with `-h`. We also move the key center top so it doesn't intersect the graph.

⟨*cli*⟩+≡
```
bash repeater.sh > repeater.dat
plotLine -x "Sequence length (Mb)" \
           -y "Time (s)" -L \
           -g "set key top center" repeater.dat
```

3.42 If we extend the path label of a frontier node by one nucleotide, it becomes unique. So we have `ATG`, `CATG`, `GC`, `GG`, and `TG`. Of these six unique substrings, `GC`, `GG`, and `TG` are the shortest.

3.43 We run `shustring` on both strands of the *M. genitalium* genome and find two shortest unique substrings of length six.

⟨*cli*⟩+≡
```
shustring -r mgGenome.fasta
```

```
>gi|84626123|gb|L43967.2| Mycoplasma genitalium...
#  Count  Position  Length  Shustring
   1      174222    6       GACGGC
   2      567107    6       GCCGGG
```

3.44 We run `shustring.sh` with 100 iterations and count the occurrences of shustrings length 6 or less. We don't find any.

⟨*cli*⟩+≡
```
bash shustring.sh 100 | awk '$3<=6'
```

3.45 We run `shustring` on the genome sequence, extract the genome positions in units of 100 kb and the shustring lengths, and plot them.

⟨*cli*⟩+≡
```
shustring -r -l mgGenome.fasta | tail -n +3 |
    awk '{print $1/100000, $2}' |
    plotLine -x "Position (100 kb)" -y Length
```

3.3 Suffix Arrays

3.46 We change into the directory for this chapter, make the directory for this section, and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/3/
mkdir 3
cd 3/
```

3.47 We apply `suf.awk` to `CATGGCAT` to get a triangle of suffixes.

⟨*cli*⟩+≡
```
printf ">s\nCATGGCAT\n" | awk -f suf.awk
```

```
1       CATGGCAT
2       ATGGCAT
3       TGGCAT
4       GGCAT
5       GCAT
6       CAT
7       AT
8       T
```

3.48 As we read the entries in the suffix array, sa, from top to bottom, we get the same sequence of suffices as when we read the leaves of the suffix tree from left to right.

3.49 We apply `suf.awk` to `CATGGCAT$`, sort the suffixes, and print them with a new index.

⟨*cli*⟩+≡
```
printf ">s\nCATGGCAT$\n" | awk -f suf.awk | sort -k 2 |
    cat -n
```

3.50 The root corresponds to interval sa[1...6], the node with path label C to interval sa[4...6], and the node with path label CC to interval sa[5...6].

3.51 Fig. 3.30A shows all common prefixes, which are also color-coded in the column of suffixes. These colors are repeated in the suffix tree in Fig. 3.30B. All three edges leading to an inner node of the suffix tree are labeled by a common prefix, because a suffix tree essentially summarizes the common prefixes of suffixes.

A

index	sa	suf	cp
1	6	$	nd
2	5	A$	-
3	1	ACCCA$	A
4	4	CA$	-
5	3	CCA$	C
6	2	CCCA$	CC

B

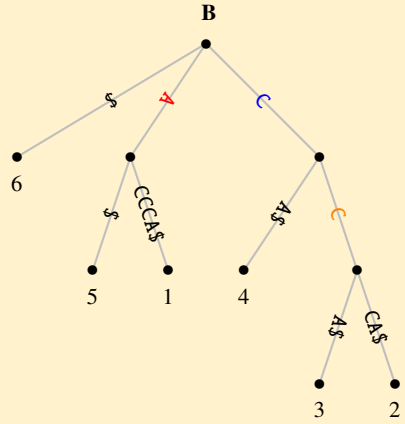


Fig. 3.30 Complete table of common prefixes (**A**) and their occurrence in the suffix tree (**B**)

3.52 Table 3.5 shows the suffix array, sa, with the complete lcp array.

Table 3.5 Complete table of longest common prefix lengths, the lcp array

index	sa	suf	cp	lcp
1	6	$	nd	-1
2	5	A$	-	0
3	1	ACCCA$	A	1
4	4	CA$	-	0
5	3	CCA$	C	1
6	2	CCCA$	CC	2

3.53 The root has no path label, so its string depth is zero, then there are two internal nodes with string depth 1 and one with string depth 2.

3.54 The distinct entries in the lcp array, 0, 1, and 2, correspond to the string depth of the suffix tree in Fig. 3.11B.

3.55 Fig. 3.31 shows the construction of the remaining lcp intervals.

3.56 We've got the suffix array and the suffix tree in Fig. 3.10, so we can draw the lcp interval tree in Fig. 3.32 from them.

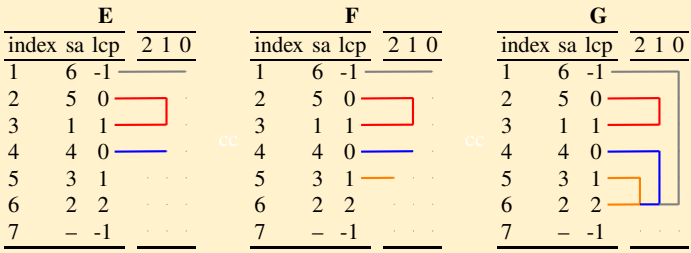


Fig. 3.31 Completing the construction of lcp intervals begun in Fig. 3.13

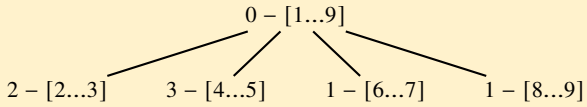


Fig. 3.32 Lcp interval tree of `CATGGCAT$`

3.57 We pipe `CATGGCAT$` through `drawSt` with the interval option (`-i`) and redirect the output to `st.tex`. We also write a LaTeX wrapper file, which we typeset and view with `evince`.

⟨*cli*⟩+≡

```
printf ">s\nCATGGCAT$\n" | drawSt -i -w wrapSt.tex > st.tex
latex wrapSt
dvips wrapSt
ps2pdf wrapSt.ps
evince wrapSt.pdf &
```

3.58 We print the suffixes, sort them, and add a new index column.

⟨*cli*⟩+≡

```
printf ">s\nGTAAACTATT$\n" | awk -f suf.awk | sort -k 2 |
    cat -n
```

```
1  11 $
2  3  AAACTATT$
3  4  AACTATT$
4  5  ACTATT$
5  8  ATT$
6  6  CTATT$
7  1  GTAAACTATT$
8  10 T$
9  2  TAAACTATT$
10 7  TATT$
11 9  TT$
```

3.59 We compare every suffix with its left-hand neighbor and write down the length of their common prefix. We do this by hand for now, but will soon automate it.

```
1  11 $             -1
2  3  AAACTATT$      0
3  4  AACTATT$       2
4  5  ACTATT$        1
5  8  ATT$           1
6  6  CTATT$         0
7  1  GTAAACTATT$    0
8  10 T$            0
9  2  TAAACTATT$     1
10 7  TATT$          2
11 9  TT$            1
12                  -1
```

3.60 To automatically draw the suffix tree, we generate the LaTeX input file, typeset its wrapper, and view it.

⟨*cli*⟩+≡

```
printf ">s\nGTAAACTATT$\n" | drawSt > st.tex
latex wrapSt.tex
dvips wrapSt
ps2pdf wrapSt.ps
evince wrapSt.pdf &
```

3.61 Rather than following the rules about the relationship between lcp[i] and lcp[$i + 1$], we just looked at the suffix tree to find the clusters in Table 3.6. In case you were tempted to also include, for example, the interval $1 - [2...4]$, recall that an interval contains *all* the leaves of its subtree.

Table 3.6 Lcp intervals in the enhanced suffix array of `GTAAACTATT$`

i	suf	sa	lcp
1	`$`	11	-1
2	`AAACTATT$`	3	0
3	`AACTATT$`	4	2
4	`ACTATT$`	5	1
5	`ATT$`	8	1
6	`CTATT$`	6	0
7	`GTAAACTATT$`	1	0
8	`T$`	10	0
9	`TAAACTATT$`	2	1
10	`TATT$`	7	2
11	`TT$`	9	1
12	—	—	-1

3.62 The topology of the lcp interval tree is the suffix tree in Fig. 3.15 stripped of its leaves. The node content can be read from the enhanced suffix array that we've annotated with lcp intervals in Table 3.6. To draw this with `drawSt`, we use its interval option (`-i`).

⟨*cli*⟩+≡
```
printf ">s\nGTAAACTATT$\n" | drawSt -i > st.tex
latex wrapSt.tex
dvips wrapSt
ps2pdf wrapSt.ps
evince wrapSt.pdf &
```

3.63 Tables 3.7A–F show the enhanced suffix array of `ACCCA$` and the construction of its inverse. We start in Table 3.7A at $i = 1$ and sa[1] = 6, which means that isa[6] = 1. In Table 3.7B we move to $i = 2$ and sa[2] = 5, so isa[5] = 2. Table 3.7C takes us to $i = 3$, and so on.

Table 3.7 Construction of the inverse suffix array of `ACCCA$`

A

isa	*i*	sa	suf	cp	lcp
	1	**6**	$	-	-1
	2	5	A$	-	0
	3	1	ACCCA$	A	1
	4	4	CA$	-	0
	5	3	CCA$	C	1
1	**6**	2	CCCA$	CC	2

B

isa	*i*	sa	suf	cp	lcp
	1	6	$	-	-1
	2	**5**	A$	-	0
	3	1	ACCCA$	A	1
	4	4	CA$	-	0
2	**5**	3	CCA$	C	1
1	6	2	CCCA$	CC	2

C

isa	*i*	sa	suf	cp	lcp
3	**1**	6	$	-	-1
	2	5	A$	-	0
	3	**1**	ACCCA$	A	1
	4	4	CA$	-	0
2	5	3	CCA$	C	1
1	6	2	CCCA$	CC	2

D

isa	*i*	sa	suf	cp	lcp
3	1	6	$	-	-1
	2	5	A$	-	0
	3	1	ACCCA$	A	1
4	**4**	**4**	CA$	-	0
2	5	3	CCA$	C	1
1	6	2	CCCA$	CC	2

E

isa	*i*	sa	suf	cp	lcp
3	1	6	$	-	-1
	2	5	A$	-	0
5	**3**	1	ACCCA$	A	1
4	4	4	CA$	-	0
2	**5**	**3**	CCA$	C	1
1	6	2	CCCA$	CC	2

F

isa	*i*	sa	suf	cp	lcp
3	1	6	$	-	-1
6	**2**	5	A$	-	0
5	3	1	ACCCA$	A	1
4	4	4	CA$	-	0
2	5	3	CCA$	C	1
1	**6**	**2**	CCCA$	CC	2

3.64 We transcribe the definition of the inverse suffix array into Awk.

⟨*Construct* isa*, Prog. 3.6*⟩≡
```
isa[sa[n]] = n
```

Then we run `isa.awk` on the suffix array of `ACCCA$`.

⟨*cli*⟩+≡
```
printf ">s\nACCCA$\n" | awk -f suf.awk | sort -k 2 |
    awk -f isa.awk
```

In the result, sa[1] = 6, so isa[6] = 1, as expected.

```
# i     sa      isa     suf
1       6       3       $
2       5       6       A$
3       1       5       ACCCA$
4       4       4       CA$
5       3       2       CCA$
6       2       1       CCCA$
```

3.65 The first column of the input contains the suffix array, the second the suffixes. As index we use the number of records, NR.

⟨*Store* sa *and* suf*, Prog. 3.7*⟩≡
```
n = NR
sa[n] = $1
suf[n] = $2
```

3.66 The input sequence is suffix number 1, which we save to variable t.

⟨*Find input sequence, t, Prog. 3.7*⟩≡
```
if ($1 == 1)
  t = $2
```

3.67 As we did before, we just follow the definition $\text{isa}[sa[i]] = i$.

⟨*Compute* isa*, Prog. 3.7*⟩≡
```
for (i = 1; i <= n; i++)
  isa[sa[i]] = i
```

3.68 We apply the Awk function split with an empty string as delimiter and generate the character array ta.

⟨*Split t into character array* ta*, Prog. 3.7*⟩≡
```
split(t, ta, "")
```

3.69 We set the first lcp value to -1 as stated in line 5 of Algorithm 1. Awk automatically initializes the ℓ to zero.

⟨*Initialize* lcp *computation, Prog. 3.7*⟩≡
```
lcp[1] = -1
```

3.70 From the isa we get the position of $t[i...]$ in the suffix array, j. The desired partner of $t[i...]$ is $t[k...]$, where $k = \text{sa}[j-1]$. But $\text{sa}[j-1]$ only makes sense if j is not the first position in sa; otherwise, we skip the rest of the loop.

⟨*Find suffix most similar to* $t[i...]$*,* $t[k...]$*, Prog. 3.7*⟩≡
```
j = isa[i]
if (j == 1)
  continue
k = sa[j-1]
```

3.71 Lines 10–14 of Algorithm 1 show the details of how the lcp value is calculated. We transcribe them into Awk.

⟨*Calculate lcp value, Prog. 3.7*⟩≡
```
while (ta[i+l] == ta[k+l])
  l++
lcp[j] = l
if (l > 0)
  l--
```

3.72 We print the index, the suffix array, the lengths of the longest common prefixes, and the suffixes.

⟨*Print enhanced suffix array, Prog. 3.7*⟩≡
```
    printf "# i\tsa\tlcp\tsuf\n"
    for (i = 1; i <= n; i++)
        printf "%d\t%d\t%d\t%s\n", i, sa[i], lcp[i], suf[i]
```

3.73 We pipe the sorted output of `suf.awk` through `esa.awk` to get the enhanced suffix array.

⟨*cli*⟩+≡
```
printf ">s\nACCCA$\n" | awk -f suf.awk | sort -k 2 |
    awk -f esa.awk
```

```
# i     sa      lcp     suf
1       6       -1      $
2       5       0       A$
3       1       1       ACCCA$
4       4       0       CA$
5       3       1       CCA$
6       2       2       CCCA$
```

3.74 The script `lrep.sh` extracts suffixes from its input sequence and sorts them. The sorted suffixes are the input to `esa.awk`. From its output, `lrep.sh` removes the header before sorting it in reverse by lcp value. The longest repeat is in the first row of the output.

3.75 We copy the *Adh* region of *D. guanche* to our working directory and run `lrep.sh` on it.

⟨*cli*⟩+≡
```
cp $BEB/data/dgAdhAdhdup.fasta .
bash lrep.sh dgAdhAdhdup.fasta
```

The longest repeat has length 12 and one copy of it occurs at position 988.

```
3325    988     12      TACATTACATTA...
```

3.76 We randomize the *D. guanche Adh* region 100 times and count the frequency with which we find a longest repeat of length 12 or greater.

⟨*cli*⟩+≡
```
for a in $(seq 100)
do
    randomizeSeq dgAdhAdhdup.fasta | bash lrep.sh | cut -f 3
done | awk '$1>=12{c++}END{print c/NR}'
```

Our result happens to be 0.46, which would be an unacceptably large error probability when rejecting our null hypothesis, so we don't. In other words, the longest repeat we found in the *Adh* region of *D. guanche* may well be due to chance.

3.77 We use `keyMat` to search for `TACATTACATTA` and find that, as expected, it occurs at position 988, and also at position 983.

⟨*cli*⟩+≡
```
keyMat TACATTACATTA dgAdhAdhdup.fasta
```

3.78 We run `repeater` with the positions option (`-p`) on the *D. guanche Adh* region and find that, as expected, its longest repeat is `TACATTACATTA`, which occurs at positions 983 and 988.

⟨*cli*⟩+≡
```
repeater -p dgAdhAdhdup.fasta
```

3.79 We copy the *D. melanogaster Adh* region to our working directory and apply `lrep.sh` to it.

⟨*cli*⟩+≡
```
cp $BEB/data/dmAdhAdhdup.fasta .
bash lrep.sh dmAdhAdhdup.fasta
```

The longest repeat has length 16, and one of its copies is found at position 3908.

```
3890    3908    16      TCGATGTCCTGATCAA...
```

3.80 We run `repeater` with the positions option to find that the longest repeat also occurs at position 2345.

⟨*cli*⟩+≡
```
repeater -p dmAdhAdhdup.fasta
```

3.81 The longest repeat for the concatenated *Adh* sequences must be at least as long as the longer of the two repeats seen in the individual sequences, that is, it should be at least 16 bp long.

3.82 We concatenate the two *Adh* sequences and pipe them through `lrep.sh`.

⟨*cli*⟩+≡
```
cat d[gm]AdhAdhdup.fasta | bash lrep.sh
```

```
1591 3287 37 AGCAAGGTTCTCATGACCAAGAATATAGCGGTGAGTG...
```

As expected, the longest repeat is at least 16 bp long. In fact, the actual longest repeat of 37 bp is much longer than this lower bound. The reason for this is that the two alcohol dehydrogenase sequences were taken from two *Drosophila* species; the relatedness between these species means there is much more sequence similarity between than within the sequences.

3.83 The script `plotLcp.sh` calculates the enhanced suffix array from the input sequence. It then cuts off two lines, the header and the first line of the table. The first line of the table is cut off, because we saw earlier that its lcp entry is by definition -1. The remaining lcp values are extracted from the table and sorted by suffix position.

3.84 We plot the lcp values for the two *Adh* regions using `plotLcp.sh` on the *Adh* region of *D. guanche* and plot the resulting lcp values with an adjusted y range.

⟨*cli*⟩+≡

```
bash plotLcp.sh dgAdhAdhdup.fasta |
    plotLine -Y 0:40 -x Position -y lcp
```

3.85 We repeat the run of `plotLcp.sh` on the *melanogaster* sequence and plot the output.

⟨*cli*⟩+≡

```
bash plotLcp.sh dmAdhAdhdup.fasta |
    plotLine -Y 0:40 -x Position -y lcp
```

3.86 We run `plotLcp.sh` on the two *Adh* regions and adjust both the y range and the x range. The range of the x axis is the length of the *D. guanche Adh*, 4433 bp.

⟨*cli*⟩+≡

```
bash plotLcp.sh d[gm]AdhAdhdup.fasta |
    plotLine -Y 0:40 -X 0:4433 -x Position -y lcp
```

3.4 Text Compression

3.87 We change into the chapter's directory, make a new subdirectory, and change into that.

⟨*cli*⟩≡

```
cd $BEB/ch/3/
mkdir 4
cd 4/
```

3.88 We shift `thatisthequestion$` five times to the left to get

```
thatisthequestion$
hatisthequestion$t
atisthequestion$th
tisthequestion$tha
isthequestion$that
```

3.89 The trick is to use the modulo operator, `%`, to wrap around the positions in the string.

⟨*Print string rotation, Prog. 3.10*⟩≡
```
for (j = 0; j < n; j++) {
  p = (i-1+j) % n
  printf("%c", ta[p+1])
}
printf("\n")
```

3.90 We run `rotate.awk` on our text and pipe the result through `sort`.

⟨*cli*⟩+≡
```
printf ">s\ntobeornottobe$\n" | awk -f rotate.awk | sort
```

3.91 We read the last column of Fig. 3.21B from top to bottom to get the Burrows-Wheeler transform of `tobeornottobe$`,

```
eoobbrttenot$o
```

3.92 We copy `suf.awk` and use it to compute the suffix array of `tobeornottobe$`. Then we sort the suffixes to get Table 3.8A.

⟨*cli*⟩+≡
```
cp ../3/suf.awk .
printf ">s\ntobeornottobe$\n" | awk -f suf.awk | sort -k 2
```

Now we construct the Burrows-Wheeler transform from the suffix array by applying equation (3.1). So we go through the entries in the suffix array from the top and concatenate the characters $t[14-1]$, $t[12-1]$, $t[3-1]$, and so on from Table 3.8B. The only exception is that we define $t[0]$ as the sentinel. This gives us the transform,

```
eoobbrttenot$o
```

3.93 We copy `hamlet.fasta`, run `cres` on it, cut off the header of its output and sort the character counts to find that ampersand is the least frequent, followed by `J`. Underscore is the most frequent character because it is used instead of blanks. The most frequent proper character is `e`, followed by `t`.

⟨*cli*⟩+≡
```
cp $BEB/data/hamlet.fasta .
cres hamlet.fasta | tail -n +3 | sort -k 2 -n
```

Table 3.8 The suffix array (**A**) of `tobeornottobe$` (**B**)

A	B
14 $	
12 be$	
3 beornottobe$	
13 e$	
4 eornottobe$	
7 nottobe$	
11 obe$	
2 obeornottobe$	
5 ornottobe$	
8 ottobe$	
6 rnottobe$	
10 tobe$	
1 tobeornottobe$	
9 ttobe$	

```
&       5     2.83e-05
J       9     5.09e-05
(       16    9.06e-05
...
t       10986 0.0622
e       14484 0.082
_       32239 0.182
```

3.94 We transform *Hamlet* and see long runs of characters as we browse through it.

⟨*cli*⟩+≡

```
bwt hamlet.fasta | less
```

```
...
rrstgtptrswtsIIII____________________LLLLLLLLLLLLLLLLLLLLL
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLNNNNNNNNNN
NNNNNNNRHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
...
```

3.95 The first run starts at position $s = 1$. The first position where a second run might start is $i = 2$. If the current text position differs from the previous one, we've reached the start of a new run. In that case we compute the length of the run by subtracting its start from the current position. We store the run length and update the start of the next run to the current position. Having looped over the text, we store the length of the last run.

⟨*Count run lengths, Prog. 3.11*⟩≡

```
s = 1
for (i = 2; i <= n; i++) {
  if (ta[i-1] != ta[i]){
    l = i - s
```

```
      cl[l]++
      s = i
    }
  }
  l = i - s
  cl[l]++
```

3.96 We iterate over all lengths and print them with their counts.

⟨*Print run lengths and counts, Prog. 3.11*⟩≡
```
  for (l in cl)
    print l, cl[l]
```

3.97 We run `runs.awk` on `hamlet.fasta` and find that the longest run has length 3 and occurs 7 times.

⟨*cli*⟩+≡
```
  awk -f runs.awk hamlet.fasta
```

```
1 170253
2 3204
3 7
```

3.98 We apply `runs.awk` to the transformed `hamlet.fasta` and look at the last entry in the sorted output to find that the longest run now has length 567 and occurs once.

⟨*cli*⟩+≡
```
  bwt hamlet.fasta | awk -f runs.awk | sort -n | tail
```

```
...
410 1
503 1
567 1
```

3.99 In natural languages like English there are strong preferences for certain combinations of phonemes, and hence characters. Since the Burrows-Wheeler transform sorts on suffixes and then returns the prefixing character of each suffix, it exposes the context preference of the characters in a text.

3.100 We generate a list of run lengths labeled `ori` and another labeled `bwt`. We sort both lists and store them in `runs.dat`. Then we plot `runs.dat` with `plotLines`. We log-transform the y axis, expand the x range a bit to the left to make the *ori* curve visible, expand the y range a bit to the bottom to make the tail of single runs visible, and print the numbers along the y axis in "engineering" format.

⟨*cli*⟩+≡
```
  awk -f runs.awk hamlet.fasta | awk '{print $1, $2, "ori"}' |
      sort -n > runs.dat
  bwt hamlet.fasta | awk -f runs.awk |
```

```
    awk '{print $1, $2, "bwt"}' | sort -n >> runs.dat
plotLine -x Length -y Count -l y -X -20:600 -Y 0.5: \
           -g "set format y '%.0e'" runs.dat
```

3.101 We randomize `hamlet.fasta` and run it through `runs.awk`. Then we also apply the Burrows-Wheeler transform to the randomization and again print the run lengths.

⟨*cli*⟩+≡

```
randomizeSeq hamlet.fasta | awk -f runs.awk
randomizeSeq hamlet.fasta | bwt | awk -f runs.awk
```

The two distributions are almost identical, for example,

```
1 155446
2 8791
3 972
4 147
5 19
6 7
7 2
```

In other words, in random sequences the Burrows-Wheeler transform has no effect on the distribution of run lengths.

3.102 We prepare our auxiliary table with the S and E column in Table 3.9 and find that the message is `thatisthequestion$`. We can check this with `bwt` in decoding mode.

⟨*cli*⟩+≡

```
printf ">s\nnhhuttttoieie\$ssaq\n" | bwt -d
```

Table 3.9 Decoding the Burrows-Wheeler transform `nhhuttttoieie$ssaq`

S	E	S	E	S	E
$_1$	n$_1$	i$_1$	t$_3$	s$_2$	e$_2$
a$_1$	h$_1$	i$_2$	t$_4$	t$_1$	$_1$
e$_1$	h$_2$	n$_1$	o$_1$	t$_2$	s$_1$
e$_2$	u$_1$	o$_1$	i$_1$	t$_3$	s$_2$
h$_1$	t$_1$	q$_1$	e$_1$	t$_4$	a$_1$
h$_2$	t$_2$	s$_1$	i$_2$	u$_1$	q$_1$

3.103 The alphabet can be in any order, we use the order in which the characters appear in the input:

Character	Number	Character	Number
t	0	e	3
o	1	r	4
b	2	n	5

3.104 This time our alphabet just contains two entries, `T` and `A`:

Character	Number
T	0
A	1

The encoded sequence is

```
0 1 1 1 0 0 1 0 0 0
```

3.105 We print the string in question and pipe it through `mtf` alone or through `bwt` and `mtf`.

⟨*cli*⟩+≡
```
printf ">s\nthatisthequestion\n" | mtf
printf ">s\nthatisthequestion\n" | bwt | mtf
```

The results contain the alphabet in double quotes at the end of the header. The untransformed "sequence" contains one zero after move to front, the transformed sequence six.

```
>s - mtf "thaisequon"
0 1 2 2 3 4 2 4 5 6 7 2 5 5 6 8 9
>s - bwt - mtf "nhutoie$saq"
0 1 0 2 3 0 0 0 4 5 6 1 1 7 8 0 9 10
```

3.106 We run `mtf` on `hamlet.fasta`, cut off the header, convert the blanks to newlines, sort the codes, count the unique codes, and reverse-sort them by count.

⟨*cli*⟩+≡
```
mtf hamlet.fasta | tail -n +2 | tr ' ' '\n' | sort |
    uniq -c | sort -n -r | head
```

We find that `3` is the most frequent code with `4` not far behind.

```
13453 3
13338 4
10617 5
10357 2
...
```

3.107 We apply `bwt` to `hamlet.fasta`, run the result through `mtf`, and sort the code counts.

⟨*cli*⟩+≡
```
bwt hamlet.fasta | mtf | tail -n +2 | tr ' ' '\n' | sort |
    uniq -c | sort -n -r | head
```

We find that `0` is now by far the most frequent code.

```
88093 0
22769 1
13090 2
```

```
 9157 3
 7046 4
...
```

3.108 We run `hamlet.fasta` through either `mtf` only, or through `bwt` followed by `mtf`, and format the codes with `fc.sh`. We transform the two output columns of `fc.sh` to three columns, consisting of code, count, and a marker, either `mtf` or `bwt/mtf`. We store the data in `mtf.dat`, which we plot with `plotLines` after dividing the counts by 1000.

⟨*cli*⟩+≡
```
mtf hamlet.fasta | bash fc.sh |
    awk '{print $2, $1, "mtf"}' > mtf.dat
bwt hamlet.fasta | mtf | bash fc.sh |
    awk '{print $2, $1, "bwt/mtf"}' >> mtf.dat
awk '{print $1, $2/1000, $3}' mtf.dat |
    plotLine -x Code -y "Count (x 1000)"
```

In the script `fc.sh` we remove the header, convert the rows of codes to a single column, count the unique codes, and sort them by code.

Prog. 3.12 (`fc.sh`)

⟨*fc.sh*⟩≡
```
    tail -n +2 |
    tr ' ' '\n' |
    sort |
    uniq -c |
    sort -n -k 2
```

3.109 The decoded sequence is `CCCGGCCGGC`, which we can check using `mtf` in decoding mode by supplying the alphabet in double quotes at the end of the header. The input to `mtf` consists of numbers delimited by blanks rather than a sequence of characters without delimiters as we are used to in FASTA files.

⟨*cli*⟩+≡
```
printf ">s \"GC\"\n0 0 0 1 0 1 0 1 0 1\n" | mtf -d
```

3.110 The genome of *M. genitalium* contains 580,076 nucleotides, so it occupies $580,076 \times 8 = 4,640,608$ bits.

3.111 Each code is two bits long, so the *M. genitalium* genome now occupies $580,076 \times 2 = 1,160,152$ bits.

3.112 We look up `11` to find `T`, `01` to find `C`, and so on, to finally get `TCTGAATGGT`. Decoding fixed length codes like Table 3.3 or the genetic code is easy.

3.113 We copy the file `mgGenome.fasta` to our current directory. Since the nucleotides differ in their space requirements, we count them with `cres`.

⟨*cli*⟩+≡
```
cp $BEB/data/mgGenome.fasta .
cres mgGenome.fasta
```

```
A       200544 0.346
C       91515  0.158
G       92306  0.159
T       195711 0.337
```

So we compute a space requirement of

$$200544 + (91515 + 92306) \times 3 + 195711 \times 2 = 1,143,429$$

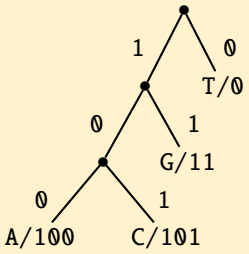
bits. This is $1,143,429/1,160,152 \times 100 \approx 98.6\%$ of the two bit encoding. Not much of a saving, but abandoning bytes helped a lot.

3.114 The first 1 is not a code yet, so we read the next bit to get 11. That's T. Then we read 0, that's A, and so on until we get

TAAGTTATTA

3.115 The first two bits give T, the next 0 might be an A or the prefix of C, we extend to 00, which might be an AA or the prefix of C. This ambiguity is resolved in the next step, the 1 decides we've read AA. We extend the 1 to 11, which gives us a C, but then we are stuck—000 is either C or AAA and there is no way to distinguish between these two possibilities. We cannot decode this string of bits any further.

3.116 We can switch the labels of the branches joined at the root to get an alternative prefix code.



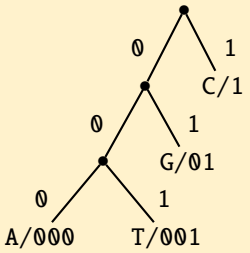
So there is nothing special about labeling the left branch zero. All we have to do is give the two child branches distinct labels.

3.117 We again look up the counts for A (200544), C (91515), G (92306), and T (195711). Then we calculate a space requirement of

$$(200544 + 91515) \times 3 + 92306 \times 2 + 195711 = 1,256,500$$

bits. This is slightly more than the 1,143,429 bits required under the prefix code in Table 3.4A and also more than the 1,160,152 bits for the code with a fixed length of 2.

3.118 We can relabel the leaves in Fig. 3.24D such that the most frequent nucleotides get the longest codes,



This code would require

$$(200544 + 195711) \times 3 + 92306 \times 2 + 91515 = 1,464,892$$

bits.—Not exactly the direction we'd like to optimize a code in.

3.119 We copy the sequence to our current directory and count its nucleotides with `cres`

⟨*cli*⟩+≡

```
cp $BEB/data/dmAdhAdhdup.fasta .
cres dmAdhAdhdup.fasta
```

```
A       1417  0.298
C       1007  0.212
G       989   0.208
T       1348  0.283
```

So we first join `C` and `G` into a node of weight 0.42. The lightest node we can form next is by joining `A` and `T` into a node of weight 0.581. Finally, we join these two clusters in the root to get Fig. 3.33A, which leads to the Huffman codes in Fig. 3.33B.

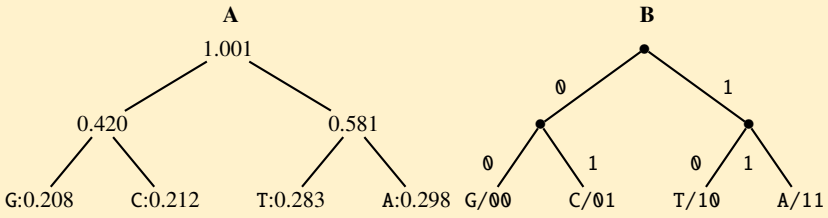


Fig. 3.33 Binary tree (**A**) to construct Huffman codes (**B**)

3.120 We run `hut` on `mgGenome.fasta` and plot the tree with `plotTree`.

⟨*cli*⟩+≡

```
hut mgGenome.fasta | plotTree
```

3.121 We copy the proteome and concatenate all sequences into a single one. Then

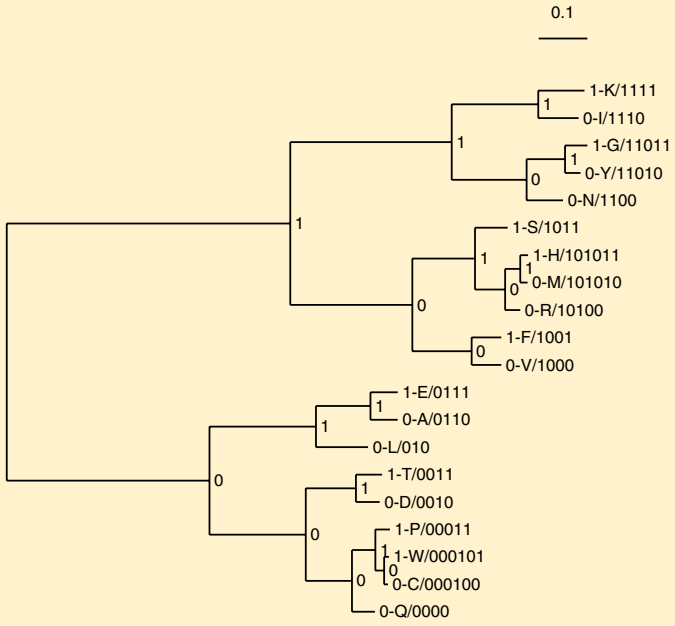


Fig. 3.34 Huffman tree for the proteome of *M. genitalium*

we construct and plot the Huffman tree shown in Fig. 3.34.

⟨*cli*⟩+≡

```
cp $BEB/data/mgProteome.fasta .
echo ">mgProteome" > mgp.fasta
grep -v '^>' mgProteome.fasta >> mgp.fasta
hut mgp.fasta | plotTree
```

3.122 In Huffman trees the most frequent residues have the shortest codes. The shortest code in the Huffman tree of the *M. genitalium* proteome in Fig. 3.34 has length 3. There is only one such code in the tree, that for leucine, L. So leucine is the most frequent amino acid in the proteome of *M. genitalium*. We confirm this by directly counting the amino acids and sorting their counts.

⟨*cli*⟩+≡

```
cres mgProteome.fasta | tail -n +3 | sort -k 2 -n -r
```

```
L       18730 0.107
K       16671 0.095
I       14520 0.0827
...
```

3.123 We calculate the Huffman tree with `hut` and store it in `mgGenome.nwk`. Then we use that tree to encode the *M. genitalium* genome with `huff`. According to `cres`, the result contains 1,143,429 bits, which agrees with our previous calculation.

⟨*cli*⟩+≡
```
hut mgGenome.fasta > mgGenome.nwk
huff mgGenome.nwk mgGenome.fasta | cres
```

```
Total: 1143429
Residue Count  Fraction
0       475880 0.416
1       667549 0.584
```

3.124 We count the 176,682 characters in *Hamlet* with `cres` and the 838,648 bits in its encoding with `hut`, so the compression ratio is roughly 1.69.

⟨*cli*⟩+≡
```
cres hamlet.fasta | grep Total
hut -b hamlet.fasta
echo '176682 * 8 / 838648' | bc -l
```

3.125 We decode the sequence `ACTTTTAACA` and check this by decoding the bits with `huff`.

⟨*cli*⟩+≡
```
printf ">s\n0100111111111001000\n" | huff -d mgGenome.nwk
```

3.126 We construct the pipeline to find that the final number of bits is 511,512. The compression ratio is therefore 2.76.

⟨*cli*⟩+≡
```
bwt hamlet.fasta | mtf | num2char | hut -b
echo '176682 * 8 / 511512' | bc -l
```

3.127 With `wc` we find that the number of bytes in `hamlet.txt` is 176,682, after `gzip` it is 70,426, and we unzip `hamlet.txt.gz` to return to the original file. So the compression ratio of `gzip` on *Hamlet* is 2.51, a bit less than our ratio of 2.76. When we're done, we reverse the action of `gzip` with `gunzip`.

⟨*cli*⟩+≡
```
wc -c hamlet.txt
gzip hamlet.txt
wc -c hamlet.txt.gz
echo '176682 / 70426' | bc -l
gunzip hamlet.txt.gz
```

3.128 We bzip `hamlet.txt`. The number of bytes is now 55,538, and unbzip it again to return to the original `hamlet.txt`. So the compression ratio of `bzip2` is 3.18, quite a bit better than our 2.76. Explicit encoding of repeats seems to pay.

⟨*cli*⟩+≡
```
bzip2 hamlet.txt
wc -c hamlet.txt.bz2
bunzip2 hamlet.txt.bz2
echo '176682 / 55538' | bc -l
```

3.129 We begin by removing the header and the newlines from the genome of *M. genitalium*. Then we find that `gzip` has a compression ratio of 3.56, `bzip2` of 3.82, and our approach of 3.69. In other words, our approach, which we've taken from the original Burrows-Wheeler paper [6], again gives us an efficiency in between two established implementations. Not bad for a first attempt.

⟨*cli*⟩+≡
```
tail -n +2 mgGenome.fasta | tr -d '\n' > mgGenome.txt
gzip mgGenome.txt
wc -c mgGenome.txt.gz
gunzip mgGenome.txt.gz
echo '580076 / 163167' | bc -l
bzip2 mgGenome.txt
wc -c mgGenome.txt.bz2
bunzip2 mgGenome.txt.bz2
echo '580076 / 151811' | bc -l
bwt mgGenome.fasta | mtf | num2char | hut -b
echo '580076 * 8 / 1256825' | bc -l
```

Chapter 4
Fast Alignment

4.1 Global

4.1 We change into the directory for chapters, make the directory for this chapter, change into that, make the directory for this section, and change into that.

⟨*cli*⟩≡
```
cd $BEB/ch/
mkdir 4
cd 4/
mkdir 1
cd 1/
```

4.2 We sandwich `ls` by time measurements with nanosecond precision and compute their difference to find that `ls` takes something like 5 milliseconds.

⟨*cli*⟩+≡
```
start=$(date +%s.%N); ls; end=$(date +%s.%N)
dur=$(echo "$end-$start" | bc)
echo $dur
```

4.3 We iterate over the sequence lengths. For each length we construct a random sequence, record the start time, and calculate the alignment, which we throw away by writing to the null device. Then we record the end time and the duration as the difference between start and end. At the end we echo the duration as a function of the sequence length.

Prog. 4.12 (`rtAl.sh`)

⟨*rtAl.sh*⟩≡
```
for a in 1 2 5 10 20 50 100
do
    ranseq -l ${a}00 > r.fasta
    start=$(date +%s.%N)
```


B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_11

```
    al r.fasta r.fasta > /dev/null
    end=$(date +%s.%N)
    dur=$(echo "$end - $start" | bc)
    echo $a $dur
done
```

Now we run `rtAl.sh`, store its output, and plot it.

⟨*cli*⟩+≡
```
bash rtAl.sh > rtAl.dat
plotLine  -x "Length (100 bp)" -y "Time (s)" -L rtAl.dat
```

4.4 We copy the two *Adh* sequences and run `mummer` on them. Then we pipe the result through `mum2plot` and plot it with `plotSeg`.

⟨*cli*⟩+≡
```
cp $BEB/data/d[mg]Adh*.fasta .
mummer dmAdhAdhdup.fasta dgAdhAdhdup.fasta | mum2plot |
    plotSeg -x "D. melanogaster" -y "D. guanche"
```

4.5 The first MUM starts at position 679 in the *melanogaster* sequence and ends at position $679 + 22 - 1 = 700$. We cut it with `cutSeq` and find it with `keyMat` at position 549 in the *guanche* sequence.

⟨*cli*⟩+≡
```
cutSeq -r 679-700 dmAdhAdhdup.fasta > mum.fasta
keyMat -p mum.fasta dgAdhAdhdup.fasta
```

4.6 We extract the MUM lengths from the `mummer` output and sort them to find that the shortest MUM has length 21, the longest 37. The 37 might look familiar, we found it already as the length of the longest common prefix—the longest common prefix is always a MUM.

⟨*cli*⟩+≡
```
mummer dmAdhAdhdup.fasta dgAdhAdhdup.fasta | grep -v '^>' |
    awk '{print $3}' | sort -n | head -n 1
mummer dmAdhAdhdup.fasta dgAdhAdhdup.fasta | grep -v '^>' |
    awk '{print $3}' | sort -n | tail -n 1
```

4.7 This redirection only removes the last two lines of output. That's because `mummer` writes its messages to the standard error stream and its output to the standard output stream. A plain redirect, >, only redirects the data on the standard output stream.

4.8 The messages are gone, all we get are two lines of output for the 10 kb match.

```
> Rand1
        1          1      10000
```

4.9 This time we get no output at all.

4.10 We write the script `rtMummer.sh` on the same pattern as `rtAl.sh`. However, this time we iterate over steps of 10 kb instead of 100 bp. We redirect all output to the null device and print the run time as a function of sequence length.

Prog. 4.13 (`rtMummer.sh`)

⟨*rtMummer.sh*⟩≡
```
for a in 1 2 5 10 20 50 100 200 500
do
    ranseq -l ${a}0000 > r1.fasta
    start=$(date +%s.%N)
    mummer r1.fasta r1.fasta &> /dev/null
    end=$(date +%s.%N)
    dur=$(echo "$end - $start" | bc)
    echo $a $dur
done
```

Then we run the script, store its results, and plot them.

⟨*cli*⟩+≡
```
bash rtMummer.sh > rtMummer.dat
plotLine -L -x "Length (10 kb)" -y "Time (s)" rtMummer.dat
```

4.11 We write the script `rtMummer2.sh` on the pattern of `rtMummer.sh`. We generate a 500 kb sequence, which we mutate with rates ranging from 0 to 0.5. Then we measure the run time and report it as a function of mutation rate.

Prog. 4.14 (`rtMummer2.sh`)

⟨*rtMummer2.sh*⟩≡
```
ranseq -l 500000 > r1.fasta
for a in 0 0.01 0.02 0.05 0.1 0.2 0.5
do
    mutator -m $a r1.fasta > r2.fasta
    start=$(date +%s.%N)
    mummer r1.fasta r2.fasta &> /dev/null
    end=$(date +%s.%N)
    dur=$(echo "$end - $start" | bc)
    echo $a $dur
done
```

We run the script, save the results, and plot them.

⟨*cli*⟩+≡
```
bash rtMummer2.sh > rtMummer2.dat
plotLine -L -x "Mutation Rate" -y "Time (s)" rtMummer2.dat
```

4.12 We got 4832 SNPs, which is reassuringly close to the expected 5000.

4.13 We pipe the two sequences through `dnaDist` and find 4832 mismatches, exactly the number counted by `nucmer`.

⟨*cli*⟩+≡
```
cat r1.fasta r2.fasta | dnaDist -r
```

4.14 We write the script `mutate.sh` on the pattern of `rtMummer2.sh`. We again iterate over mutation rates and print the expected and observed number of mutations as a function of the mutation rate.

Prog. 4.15 (`mutate.sh`)

⟨*mutate.sh*⟩≡
```
for a in 0 0.01 0.02 0.05 0.1 0.2 0.5
do
    mutator -m $a r1.fasta > r2.fasta
    nucmer r1.fasta r2.fasta &> /dev/null
    e=$(cat r[12].fasta |
              dnaDist -r |
              tail -n +3 |
              awk '{print $2}')
    o=$(show-snps out.delta | tail -n +6 | wc -l)
    echo $a $o "obs"
    echo $a $e "exp"
done
```

Then we run `mutate.sh`, save the result, and plot the number of SNPs divided by 1000 as a function of the muation rate. Fig. 4.35 shows that the observed number of SNPs is exact until it breaks down for high mutation rates.

⟨*cli*⟩+≡
```
bash mutate.sh > mutate.dat
awk '{print $1, $2/1000, $3}' mutate.dat |
plotLine -L -x "Mutation rate" -y "SNPs (x 1000)"
```

4.15 We write the script `numMum.sh` on the pattern of `mutate.sh`. We iterate over the mutation rates and for each rate print the number of MUMs as a function of mutation rate.

Prog. 4.16 (`numMum.sh`)

⟨*numMum.sh*⟩≡
```
for a in 0 0.01 0.02 0.05 0.1 0.2 0.5
do
    mutator -m $a r1.fasta > r2.fasta
    n=$(mummer r1.fasta r2.fasta |
              tail -n +2 |
              wc -l)
    echo $a $n
done
```

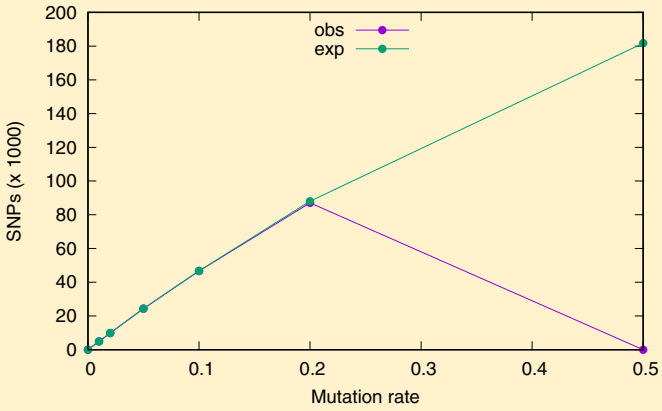


Fig. 4.35 The expected and observed number of SNPs as a function of the mutation rate

Then we run `numMum.sh`, save its output, and plot the number of MUMs divided by 1000 as a function of the mutation rate with `plotLine`.

⟨*cli*⟩+≡
```
bash numMum.sh > numMum.dat
awk '{print $1, $2/1000}' numMum.dat |
    plotLine -L -x "Mutation Rate" -y "MUMs (x 1000)"
```

4.16 We copy the sequence files, run `mummer` on both the forward and the reverse strand (`-b`), and report the matches on the reverse strand relative to the query (`-c`). Then we transform the `mummer` output to plot segments and visualize them with `plotSeg`.

⟨*cli*⟩+≡
```
cp $BEB/data/ecoli*.fasta .
mummer -b -c ecoliK12.fasta ecoliO157H7.fasta | mum2plot |
    plotSeg -x "K12" -y "O157H7"
```

4.2 Local

4.17 We change into the directory for this chapter, make the directory for this section, and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/4/
mkdir 2
cd 2/
```

4.18 We copy the two sequences, align them locally, and find that the optimal local alignment has coordinates 2182–2594 in *D. melanogaster* and 2142–2554 in *D. guanche* with a score of 217.

⟨*cli*⟩+≡
```
cp $BEB/data/d*Adh*.fasta .
al -l dmAdhAdhdup.fasta dgAdhAdhdup.fasta
```

4.19 We run `sblast` to find three alignments, the best of which coincides with the best alignment returned by `al`. That's reassuring.

⟨*cli*⟩+≡
```
sblast dmAdhAdhdup.fasta dgAdhAdhdup.fasta
```

```
#qa    sa        qs    qe    ss    se    score
DMADH  DGADHDUP  2182  2594  2142  2554  217.0
DMADH  DGADHDUP  3829  4028  3621  3820  80.0
DMADH  DGADHDUP  3225  3328  3220  3323  68.0
```

The columns in the output of `sblast` are query accession (`qa`), subject accession (`sa`), query start (`qs`), query end (`qe`), subject start (`ss`), subject end (`se`), and alignment score (`score`).

4.20 All query words have the same length, 11, and there are 4751 of them. This is consistent with the length of the *D. melanogaster Adh*, 4761, since $4751 + 11 - 1 = 4761$.

⟨*cli*⟩+≡
```
sblast -l dmAdhAdhdup.fasta dgAdhAdhdup.fasta | tail -n +2 |
    awk '{print length($3)}' | sort | uniq
sblast -l dmAdhAdhdup.fasta dgAdhAdhdup.fasta | tail -n 1
cres dmAdhAdhdup.fasta
```

4.21 With `sort` and `uniq` we find that there are 17 words in `dmAdhAdhdup.fasta` that are not unique.

⟨*cli*⟩+≡
```
sblast -l dmAdhAdhdup.fasta dgAdhAdhdup.fasta | tail -n +2 |
    awk '{print $3}' | sort | uniq -c | sort -n -r |
    head -n 18
```

```
2 TTTTTTTTTTT
2 TTTTTTTTAA
2 TGTTTTTTTTT
2 TGTCCTGATCA
2 TCGATGTCCTG
...
```

4.22 We convert the words in *melanogaster Adh* into a multiple FASTA file. The words are printed in three columns, name, number, sequence. We concatenate the name and the number into the FASTA header, which is followed by the sequence.

⟨*cli*⟩+≡
```
sblast -l dmAdhAdhdup.fasta dgAdhAdhdup.fasta | tail -n +2 |
    awk '{printf ">%s\n%s\n", $1 $2, $3}' > dmWords.fasta
```

Now we search for the *melanogaster* words on the forward and reverse strands of *guanche*. We remove the header lines and count the remaining lines to find 274 matches.

⟨*cli*⟩+≡
```
keyMat -r -p dmWords.fasta dgAdhAdhdup.fasta | grep -v '^#' |
    wc -l
```

4.23 We run `al` to find that in the second alignment the query segment returned by `sblast` ends at 4028 rather than 4158. In other words, the second alignment returned by `sblast` is $4158 - 4028 + 1 = 131$ nucleotides too short.

⟨*cli*⟩+≡
```
al -l -n 3 dmAdhAdhdup.fasta dgAdhAdhdup.fasta
```

4.24 With a bit of trial and error we find that with 52 idle extensions `sblast` gives the same result as `al`.

⟨*cli*⟩+≡
```
sblast -s 52 dmAdhAdhdup.fasta dgAdhAdhdup.fasta
```

```
#qa    sa        qs    qe    ss    se    score
DMADH  DGADHDUP  2182  2594  2142  2554  217.0
DMADH  DGADHDUP  3829  4158  3621  3950  102.0
DMADH  DGADHDUP  3225  3328  3220  3323  68.0
```

4.25 `sblast` takes approximately 30 milliseconds, `al` 2780 milliseconds. So on this small example `sblast` is $2780/30 \approx 90$ times faster than `al`.

⟨*cli*⟩+≡
```
time al dmAdhAdhdup.fasta dgAdhAdhdup.fasta > /dev/null
time sblast -s 52 dmAdhAdhdup.fasta dgAdhAdhdup.fasta \
    > /dev/null
echo "2780/30" | bc -l
```

4.26 We copy the files to our current directory and use `cres` to find that the genome sequence of *D. melanogaster* is roughly 138 Mb long.

⟨*cli*⟩+≡
```
cp $BEB/data/dmChr*.fasta .
cres dmChr*.fasta
```

4.27 We repeat our search with `keyMat`, only this time we look through the entire genome to find a staggering 1,257,060 matches on the forward and reverse strands.

⟨*cli*⟩+≡
```
keyMat -r -p dmWords.fasta dmChr*.fasta | grep -v '^#' |
    wc -l
```

4.28 We run `sblast` and find that the *Adh* region is located at 14,614,315–14,619,393 in accession NT_033779.

⟨*cli*⟩+≡
```
sblast dmAdhAdhdup.fasta dmChr*.fasta
```

```
#qa    sa           qs    qe    ss        se        score
DMADH  NT_033779.5  3005  4589  14617327  14618911  1549.0
DMADH  NT_033779.5  1720  2951  14616038  14617269  1184.0
DMADH  NT_033779.5  1     921   14614315  14615235  909.0
DMADH  NT_033779.5  916   1690  14615229  14616003  731.0
DMADH  NT_033779.5  4648  4743  14619298  14619393  96.0
```

This accession corresponds to the left arm of chromosome 2.

⟨*cli*⟩+≡
```
head -n 1 dmChr*.fasta | grep NT_033779
```

```
>NT_033779.5 Drosophila melanogaster chromosome 2L
```

4.29 On our computer, `sblast` takes approximately 12 s.

⟨*cli*⟩+≡
```
time sblast dmAdhAdhdup.fasta dmChr*.fasta
```

4.30 When we aligned the *Adh* regions from *D. melanogaster* and *D. guanche*, we learned that the minimum number of idle extension steps should be 52 rather than the default 30. Using this, we find a hit in region 14,616,500–14,618,480 of chromosome 2L, which extends from 2142 to 3950 in the query.

⟨*cli*⟩+≡
```
sblast -s 52 dgAdhAdhdup.fasta dmChr*.fasta
```

```
#qa       sa           qs    qe    ss        se        score
DGADHDUP  NT_033779.5  2142  2554  14616500  14616912  217.0
DGADHDUP  NT_033779.5  3621  3950  14618151  14618480  106.0
DGADHDUP  NT_033779.5  3220  3323  14617547  14617650  64.0
DGADHDUP  NT_033779.5  2766  2879  14617131  14617244  50.0
```

This region overlaps the coding regions of *Adh* and *Adh-dup*, which extend from 2021 to 4521.

⟨*cli*⟩+≡
```
grep CDS $BEB/data/dmAdhAdhdup.gb
```

```
CDS join(2021..2119,2185..2589,2660..2926)
CDS join(3226..3321,3748..4152,4204..4521)
```

4.31 We find the single region, 3195–3328, in the *melanogaster* sequence.

⟨*cli*⟩+≡
```
blastn -query dmAdhAdhdup.fasta -subject dgAdhAdhdup.fasta
```

These 132 homologous nucleotides are far fewer than the 411 nucleotides in the top alignment found by `al` and don't even overlap it.

⟨*cli*⟩+≡
```
al -l dmAdhAdhdup.fasta dgAdhAdhdup.fasta
```

4.32 The best three results of `blastn` cover the region 1917–4517, compared to 2182–4158 for the three regions returned by `sblast` with `-s 52`. Now the `sblast` result is contained in the larger `blastn` result.

4.33 We look at the help page and find that `-outfmt 6` gives tabular output, while `-outfmt 7` gives tabular output with comments.

⟨*cli*⟩+≡
```
blastn -task blastn -query dmAdhAdhdup.fasta \
          -subject dgAdhAdhdup.fasta -outfmt 6 |
     head -n 3
```

```
DM DG 72.107 1452 296 29 2036 3420 1993 3402 0.0       761
DM DG 76.552 1015 212 11 1917 2926 1886 2879 0.0       732
DM DG 75.888 788  175 5  3743 4517 3535 4320 5.38e-161 556
```

From the comments in output format 7 we learn what the 12 columns mean:

1. query accession	5. mismatches	9. subject start
2. subject accession	6. gaps	10. subject end
3. % identity	7. query start	11. E-value
4. alignment length	8. query end	12. bit score

4.34 We begin by redirecting the chromosome files into a subject file. Then we run `blastn` and find two fragments, which cover the region 14,614,315–14,619,405 on chromosome 2L. This is a mere 13 nucleotides longer than the region identified by `sblast` in five fragments, 14,614,315–14,619,393. So the original, ungapped Blast algorithm depicted in Fig. 4.7 and implemented in `sblast` is already quite useful.

⟨*cli*⟩+≡
```
cat dmChr*.fasta > subject.fasta
blastn -query dmAdhAdhdup.fasta -subject subject.fasta \
          -outfmt 6
```

4.35 With the default megablast mode, `blastn` finds a single alignment of 134 nucleotides. This is much less than the 961 nucleotides aligned by `sblast`. However,

if we set the task to `blastn`, the top 3 alignments have a combined length of 1917, so the difference between the modes can be drastic.

⟨*cli*⟩+≡
```
blastn -query dgAdhAdhdup.fasta -subject subject.fasta \
         -outfmt 6
blastn -task blastn -query dgAdhAdhdup.fasta \
         -subject subject.fasta -outfmt 6 | head -n 3 |
    awk '{s+=$4}END{print s}'
```

4.36 `blastn` takes approximately 2.5 s, `sblast` took 12 s, so `blastn` is roughly 5 times faster than `sblast`.

⟨*cli*⟩+≡
```
time blastn -query dmAdhAdhdup.fasta -subject subject.fasta \
     -outfmt 6
```

4.37 The output of `makeblastdb` says that it took approximately 1.5 s on our computer.

4.38 We count the number of bytes in `subject.fasta` and the number of bytes in the files generated by `makeblastdb`, `dm.*`. Dividing one by the other gives a compression ratio of roughly 4.

⟨*cli*⟩+≡
```
wc -c subject.fasta
wc -c dm.*
echo '139287503/34432775' | bc -l
```

4.39 The database search took 60 milliseconds compared to 2.5 seconds on the native subject. That's a 40-fold speed-up, eight times more than the difference between `blastn` and `sblast` with uncompressed data.

⟨*cli*⟩+≡
```
time blastn -query dmAdhAdhdup.fasta -db dm -outfmt 6
echo '2.5/0.06' | bc -l
```

4.40 The more sensitive run took 460 milliseconds on our machine, 8 times more than the default megablast run.

⟨*cli*⟩+≡
```
time blastn -task blastn -query dmAdhAdhdup.fasta -db dm \
     -outfmt 6
echo '460/60' | bc -l
```

4.41 On our machine the run time with eight threads was 210 milliseconds, that is, less than half of the single-threaded run time. However, it wasn't one eighths, as might have been expected. Notice also the difference between the *real* time and the *user* time when running multiple threads.

⟨*cli*⟩+≡
```
time  blastn -num_threads 8 -task blastn \
        -query dmAdhAdhdup.fasta -db dm \
        -outfmt 6 > /dev/null
```

4.42 The database information says that the longest sequence has 32,079,331 bases, or 32 Mb.

4.43 We run the database entries through `cres` and get the same 137,567,484 nucleotides as for `subject.fasta`.

⟨*cli*⟩+≡
```
blastdbcmd -entry all -db dm | cres
cres subject.fasta
```

4.44 From `blastdbcmd` we get that chromosome 2L is 23,513,712 bases long. We retrieve chromosome 2L by its accession, NT_033779, and find that its residue count is also 23,513,712.

⟨*cli*⟩+≡
```
blastdbcmd -db dm -entry NT_033779 | cres
```

4.45 With mutation rate 0.03 we always find a hit, with mutation rate 0.3 we often don't.

4.46 We run `megablast.sh` with a mutation rate of 0.3 and get 33 successful runs. This number varies slightly from run to run.

⟨*cli*⟩+≡
```
bash megablast.sh 0.3
```

4.47 We run `blastn.sh` with 100% success.

⟨*cli*⟩+≡
```
bash blastn.sh 0.3
```

4.48 We write the program `sens.sh` to compare the sensitivity of `megablast` and `blastn`. In `sens.sh` we iterate over the mutation rates $0, 0.1, \ldots, 0.7$. For each mutation rate we run `megablast.sh` and `blastn.sh` and print the results with a category "m" for `megablast` and a category "n" for `blastn`.

Prog. 4.17 (`sens.sh`)

⟨*sens.sh*⟩≡
```
for a in $(seq 0 0.1 0.7)
do
    m=$(bash megablast.sh $a)
    n=$(bash blastn.sh $a)
    echo $a $m "m"
    echo $a $n "n"
done
```

We run `sens.sh`, store the results, and plot them with `plotLine`.

⟨*cli*⟩+≡
```
bash sens.sh > sens.dat
plotLine -L -x "Mutation Rate" -y "%-Success" sens.dat
```

4.49 The match score is 2, the mismatch score -3. The alignment consists of 27 matches and 3 mismatches. This means the raw score should be $S = 2{\times}27 - 3{\times}3 = 45$, which is also what's printed.

4.50 We calculate $S' = (0.625 \times 45 - \log(0.41))/\log(2)) \approx 41.9$, as expected.

⟨*cli*⟩+≡
```
echo '(0.625 * 45 - l(0.41)) / l(2)' | bc -l
```

4.51 We calculate $E = 100 \times 137,567,484 \times 2^{-41.9} \approx 0.003$. That's not exactly 0.005, and we are not sure why. To calculate E, we need a power function, but `bc` doesn't have one. Instead, we can apply the fact that

$$a^b = e^{b \log(a)}.$$

⟨*cli*⟩+≡
```
echo '100 * 137567484 * e(-41.9*l(2))' | bc -l
```

Alternatively, we can use Awk.

⟨*cli*⟩+≡
```
awk 'BEGIN{print 100 * 137567484 * 2^-41.9}'
```

4.52 We write the program `eval.awk`, where we iterate over a range of E-values. For each E-value we print two things, the P-value as a function of the E-value, which we mark as "true", and the E-value as a function of itself, which we mark as "E=P".

Prog. 4.18 (`eval.awk`)

⟨*eval.awk*⟩≡
```
BEGIN {
  x=0.01
  for (e = x; e <= 0.5; e += x) {
    print e, 1-exp(-e),"true"
    print e,e,"E=P"
  }
}
```

We run `eval.awk` and plot the result with `plotLine`, where we center the key.

⟨*cli*⟩+≡
```
awk -f eval.awk |
    plotLine -x E -y P -g "set key top center"
```

4.53 We run `simStats.sh` with 1000 iterations and count the lines it returns, 13 in our case.

⟨*cli*⟩+≡
```
bash simStats.sh 1000 | wc -l
```

So $P \approx 0.01$, which is a bit larger than the E-value of 0.005, which is virtually identical to the P-value. In this case the theoretical E-value is slightly too small, or over-optimistic as to the alignment's significance.

4.54 We get the same result regardless of which sequence we designate *query*. This is reassuring, though not guaranteed, as the Blast algorithm sketched in Fig. 4.7 treats query and subject differently.

⟨*cli*⟩+≡
```
sblast -s 52 dmAdhAdhdup.fasta dgAdhAdhdup.fasta
sblast -s 52 dgAdhAdhdup.fasta dmAdhAdhdup.fasta
```

4.55 The exception is the pair 721 → 2239, where the comparison is only significant if 721 is query, not if 2239 is query.

4.56 We now have numbers instead of characters as node names, but this makes no difference to the plot notation.

Prog. 4.19 (`g2.dot`)

⟨*g2.dot*⟩≡
```
graph G {
    1 -- 2
    2 -- 3
    2 -- 5
    4 -- 5
}
```

We apply `neato` to `g2.dot` and get Fig. 4.12.

⟨*cli*⟩+≡
```
neato -T x11 g2.dot
```

4.57 The program `circo` draws connected nodes on the circumference of a circle to give Fig. 4.36.

⟨*cli*⟩+≡
```
circo -T x11 yeast.dot
```

4.58 We set the edge probability to 1 and plot the resulting graph with `neato` to get Fig. 4.37A and with `circo` to get Fig. 4.37B.

⟨*cli*⟩+≡
```
ranDot -p 1 | neato -T x11
ranDot -p 1 | circo -T x11
```

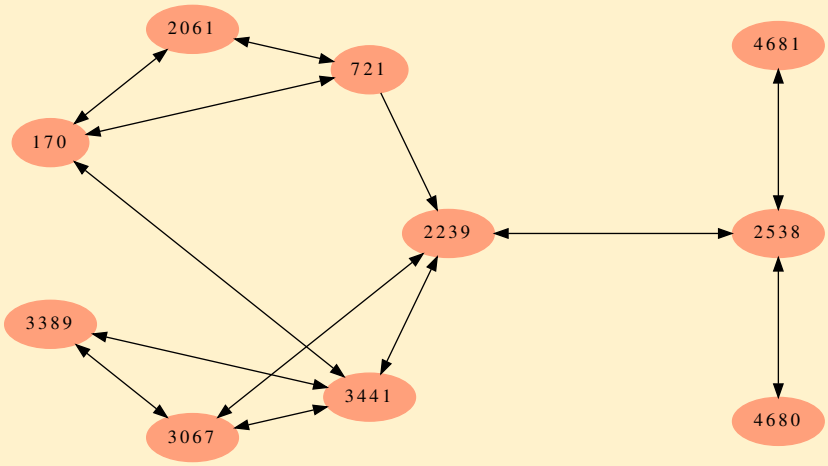


Fig. 4.36 The yeast protein family of Fig. 4.11 laid out with `circo` instead of `neato`

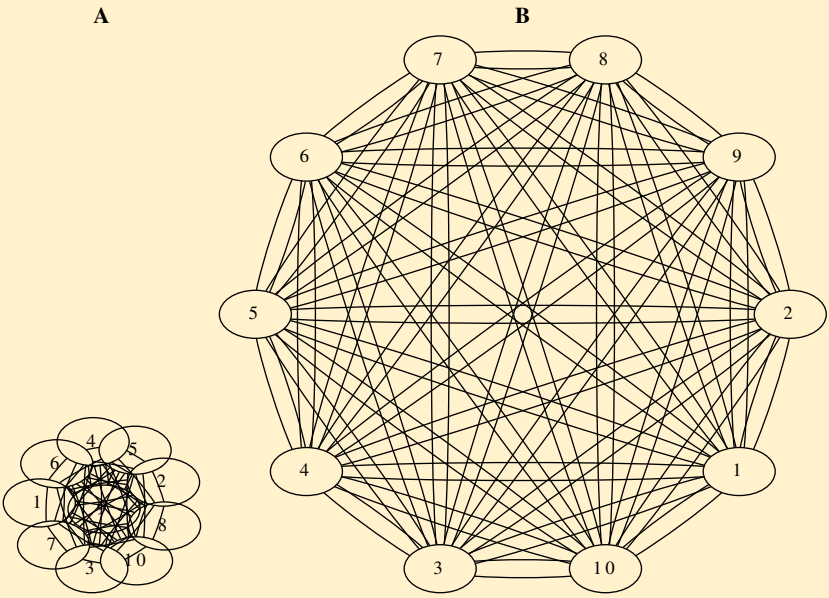


Fig. 4.37 Ten nodes with all possible edges drawn with `neato` (**A**) and `circo` (**B**)

4.59 There are 10 proteins and

$$\binom{10}{2} = \frac{10 \times 9}{2} = 45$$

protein pairs, each characterized by up to two edges. So the number of possible edges is 90. Of these, 25, that is $25/90 \approx 28\%$, actually exist.

4.60 Yes, they are all homologous to each other, because regardless of where in Fig. 4.11 we start, we can reach any of the other proteins.

4.61 We copy the proteome and count its 476 entries.

⟨*cli*⟩+≡

```
cp $BEB/data/mgProteome.fasta .
grep '^>' mgProteome.fasta | wc -l
```

4.62 The number of pairwise comparisons between the 476 *M. genitalium* proteins is 113,050.

⟨*cli*⟩+≡

```
echo '476 * 475 / 2' | bc
```

4.63 The `-F` option sets the field separator from default white space to underscore. Then the program either deals with a header line or with a data line. For a header line we print >, followed by everything after the underscore. A data line is printed as is.

4.64 The file `mgp.bl` has 836 entries, which is also the number of Blast hits.

⟨*cli*⟩+≡

```
wc -l mgp.bl
```

4.65 We cut the query column from the Blast result, sort it, count the unique entries, sort their counts, and look at the top proteins. Proteins 410, 180, and 179 each have 17 hits. Since one hit is bound to be to the query itself, they each have 16 homologs in the proteome.

⟨*cli*⟩+≡

```
cut -f 1 mgp.bl | sort | uniq -c | sort -n -r | head
```

```
17 410
17 180
17 179
16 526
...
```

4.66 We get the header lines of proteins 410, 180, and 179 to find that all three are ABC transporters. 410 transports phosphate, the other two metal ions.

⟨*cli*⟩+≡

```
grep '>410' mgProteome2.fasta
```

```
grep '>180' mgProteome2.fasta
grep '>179' mgProteome2.fasta
```

4.67 We count 101 proteins that appear at least twice in the query list.

⟨*cli*⟩+≡
```
cut -f 1 mgp.bl |
    sort |
    uniq -c |
    awk '$1 > 1' |
    wc -l
```

4.68 We apply `blast2dot` to the Blast result and pipe the graph through `circo`.

⟨*cli*⟩+≡
```
blast2dot mgp.bl | circo -T x11
```

4.69 The circle contains the 17 entries 410, 421, 303, 119, 014, 304, 015, 042, 065, 290, 187, 179, 467, 080, 526, 079, and 180. We write the program `entr.sh` to iterate over them.

Prog. 4.20 (`entr.sh`)

⟨*entr.sh*⟩≡
```
for a in 410 421 303 119 014 304 015 042 065 \
              290 187 179 467 080 526 079 180
do
    grep ">$a" mgProteome2.fasta
done
```

We run `entr.sh` to find that all members of the great circle are ABC transporters, with the apparent exception of 042, which is annotated as "spermidine". However, a web search revealed that this is in fact an ABC transporter of spermidine. So, without exception, the 17 members of the great circle in Fig. 4.13 are ABC transporters.

⟨*cli*⟩+≡
```
bash entr.sh
```

4.70 We copy `prosite.doc`.

⟨*cli*⟩+≡
```
cp $BEB/data/prosite.doc .
```

We open the file in `less` and search for "ABC transporter".

```
{PDOC00185}
{PS00211; ABC_TRANSPORTER_1}
{PS50893; ABC_TRANSPORTER_2}
{BEGIN}
 ********************************************************************
 * ATP-binding cassette, ABC transporter-type, signature and profile *
 ********************************************************************

ABC  transporters  belong  to the ATP-Binding Cassette (ABC) superfamily which
uses  the  hydrolysis  of  ATP  to  energize  diverse  biological systems. ABC
transporters  are  minimally  constituted  of  two conserved regions: a highly
```

```
conserved ATP binding cassette (ABC) and a less conserved transmembrane domain
(TMD). These regions can be found on the same protein or on two different
ones. Most ABC transporters function as a dimer and therefore are constituted
of four domains, two ABC modules and two TMDs [1].
...
{END}
```

The entry says next that "The major function of ABC import systems is to provide essential nutrients to bacteria." Perhaps this explains also why it is the largest gene family in *M. genitalium*—all nutrients that require active transport into the bacterial cell need to be taken care of.

4.71 We include the singletons in `blast2dot` and color the nodes using colors we pick from the list quoted in the help section of `blast2dot`,

`www.graphviz.org/doc/info/colors.html`

Then we pipe the graph through `circo`.

⟨*cli*⟩+≡

```
blast2dot -s -C lightgray -c lightsalmon mgp.bl |
    circo -T x11
```

4.3 Glocal

4.72 We change into the directory for Chapter 4, make the directory for Section 3, and change into that.

⟨*cli*⟩≡

```
cd $BEB/ch/4/
mkdir 3
cd 3/
```

4.73 We use `keyMat` and the local mode of `al` to look for our 100 bp fragment in the *melanogaster Adh*. `keyMat` just gives us the expected starting position, 2301, `al` the full local alignment with zero errors.

⟨*cli*⟩+≡

```
keyMat -p dmFrag.fasta dmAdhAdhdup.fasta
al -l dmFrag.fasta dmAdhAdhdup.fasta
```

4.74 We run `keyMat` on `dmFrag2.fasta` and get no result, then we run `al` in local mode and get an alignment with a single mismatch, all as expected.

⟨*cli*⟩+≡

```
keyMat -p dmFrag2.fasta dmAdhAdhdup.fasta
al -l dmFrag2.fasta dmAdhAdhdup.fasta
```

4.75 If we divide the fragment into two parts of equal length, the mismatch can only be located in one of them, and we have minimized the number of matches to look at.

4.76 By default, k is set to 1 in `kerror`. So we run it with `-l` to print the fragment list. We find two fragments of length 50, as expected.

⟨*cli*⟩+≡
```
kerror -l dmFrag2.fasta dmAdhAdhdup.fasta
```

```
#Id Start Fragment
1   1     GACCACCAAGCTGCTGAAGACCATATTCGCCCAGCTGAAGACCGTCGATG
2   51    TCCTGATCAACGGAGCTGGTATCCTGGACGATCACCAGATCGAGCGCACC
```

In contrast, with `sblast` we get all possible words of length 11, of which there are $100 - 11 + 1 = 90$.

⟨*cli*⟩+≡
```
sblast -l dmFrag2.fasta dmAdhAdhdup.fasta | head -n 4
```

```
#qa    n   word
DMADH  1   GACCACCAAGC
DMADH  2   ACCACCAAGCT
DMADH  3   CCACCAAGCTG
```

4.77 We copy the genome to our working directory. Then we execute a timed `kerror` run, which takes 1.6 s, and gives one hit in the *Adh* locus on chromosome 2L, as expected.

⟨*cli*⟩+≡
```
cp $BEB/data/dmChr*.fasta .
time kerror dmFrag2.fasta dmChr*.fasta
```

4.78 We already have a run time for `kerror`, 1.6 s for the forward strand, so we time the `sblast` run, which takes 4.4 s for the forward and reverse strands. This is significantly more than twice the time required by `kerror`, so `kerror` is faster than `sblast` in this instance.

⟨*cli*⟩+≡
```
time sblast dmFrag2.fasta dmChr*.fasta
```

4.79 We convert the two fragments used by `kerror` into a FASTA file and submit that to a search with `keyMat` across the forward strand of the *melanogaster* genome. This returns a single match.

⟨*cli*⟩+≡
```
kerror -l dmFrag2.fasta dmAdhAdhdup.fasta |
    awk '!/^#/{printf ">%s\n%s\n", $1, $3}' > kerrorF.fasta
keyMat -p kerrorF.fasta dmChr*.fasta
```

4.80 Again, we convert the words listed by `sblast` into a FASTA file, which we submit to `keyMat`. This uncovers 3220 matches. In other words, `sblast` checks 3220 times more fragments than `kerror` on the same input sequence. This slows `sblast` down compared to `kerror`.

⟨*cli*⟩+≡
```
sblast -l dmFrag2.fasta dmAdhAdhdup.fasta |
    awk '!/^#/{printf">%s\n%s\n", $2, $3}' > sblastF.fasta
keyMat -p sblastF.fasta dmChr*.fasta | grep -v '^#' | wc -l
```

4.81 We run `al` to compare the original fragment as query with the new one as subject and observe the expected gap in the subject.

⟨*cli*⟩+≡
```
al dmFrag.fasta dmFrag3.fasta
```

4.82 `kerror` finds the gapped alignment as expected. For `sblast` we have to adjust the score threshold from 50 to 49 in order to find the expected two alignments. So `sblast` breaks alignments at gaps.

⟨*cli*⟩+≡
```
kerror dmFrag3.fasta dmChr*.fasta
sblast -t 49 dmFrag3.fasta dmChr*.fasta
```

4.83 We use `al` to compare the original fragment as query to the new one and observe the single nucleotide inserted in the subject.

⟨*cli*⟩+≡
```
al dmFrag.fasta dmFrag4.fasta
```

4.84 `kerror` again finds the gapped alignment as expected. Then we run `sblast` in default mode to find the two expected fragments length 51 and 50; as with the deletion, `sblast` breaks the "true" glocal alignment at the gap.

⟨*cli*⟩+≡
```
kerror dmFrag4.fasta dmChr*.fasta
sblast dmFrag4.fasta dmChr*.fasta
```

4.85 We wrote the script `kerror.sh` to drive `kerror` with $k = 2^0, 2^1, 2^2, \ldots$. Now, the fragment length is the sequence length divided by $k + 1$. The *melanogaster Adh* region is 4761 bp long, so $k = 2^9 = 512$ would result in fragments of length 9 or 10. That seemed a bit short, so we stopped at $k = 2^8 = 256$.

Prog. 4.21 (`kerror.sh`)

⟨*kerror.sh*⟩≡
```
for a in $(seq 0 8)
do
    ((e=2**$a))
    echo $e
    kerror -k $e dmAdhAdhdup.fasta dmChr*.fasta |
          grep Errors
```

```
done
```

We run `kerror.sh` and observe first of all how `kerror` slows down as we increase k. We also find that for $k = 256$ the minimum number of errors is 148. In addition, `kerror` returns a second alignment with 228 errors. We would pick that with the fewer errors.

⟨*cli*⟩+≡
```
bash kerror.sh
```

4.86 From the `al` result we find that we'd have to run `kerror` with $k = 1987$ or greater. But that implies fragment lengths of 2 or 3, which would result in an overwhelming number of matches for checking. So searching for the *guanche Adh* region in the *melanogaster* genome wouldn't be a good idea. So the simple Blast algorithm is more sensitive than k-error.

4.87 We count the template nucleotides, 4761, and the read nucleotides, 4800, which gives a coverage of $4800/4761 \approx 1$. This fits with the expected coverage of 1.

⟨*cli*⟩+≡
```
cres dmAdhAdhdup.fasta
cres reads.fasta
```

4.88 If the user didn't set `n`, we print a usage message and exit.

⟨*Get template length, n, Prog. 4.6*⟩≡
```
if (n == 0) {
  print "Usage: awk -f cov.awk -v n=<n>"
  exit
}
```

4.89 We iterate over the n template positions and set the coverage to zero.

⟨*Initialize array of coverages, `cov`, Prog. 4.6*⟩≡
```
  for (i = 1; i <= n; i++)
    cov[i] = 0
```

4.90 We assign the value of the start position to the temporary variable, assign the end to the start, and assign the temporary value to the end. This roundabout way of switching the values of two variables via a temporary variable is a standard idiom in most programming languages.

⟨*Switch start and end, Prog. 4.6*⟩≡
```
tmp = s
s = e
e = tmp
```

4.91 We print the coverage for each position with a `for` loop over the template positions inside an `END` block.

⟨*Print coverage per position, Prog. 4.6*⟩≡
```
END {
```

```
    for (i = 1; i <= n; i++)
      print i, cov[i]
  }
```

4.92 We run `cov.awk` on `adh.bl`, mark the result with "c=1", and store it in `cov.dat`. Then we repeat the cycle of simulation, coverage calculation, and data storage for coverage 15. The final plot is generated with `plotLine`.

⟨*cli*⟩+≡

```
  awk -f cov.awk -v n=4761 adh.bl | \
      awk '{print $0, "c=1"}' > cov.dat
  sequencer -c 15 dmAdhAdhdup.fasta > reads.fasta
  blastn -query reads.fasta -subject dmAdhAdhdup.fasta \
           -outfmt 6 > adh.bl
  awk -f cov.awk -v n=4761 adh.bl | \
      awk '{print $0, "c=15"}' >> cov.dat
  plotLine -x Position -y Coverage cov.dat
```

4.93 We pick the rows marked "c=1" from `cov.dat`, count the coverages, and print them as a sorted list.

⟨*cli*⟩+≡

```
  awk '$3=="c=1"' cov.dat |
      awk '{c[$2]++}END{for(a in c)print a, c[a]}' | sort -n
```

We find that the per site coverage in our simulation ranges from 0 to 7.

```
0 1899
1 1684
2 712
3 357
4 53
5 31
6 21
7 4
```

4.94 We mark the results we just produced "c=1" and save them in the file `countCov.dat`. Then we repeat the calculation for coverage 15 and append the marked results to `countCov.dat`. In a third step we plot the two distributions with `plotLine`.

⟨*cli*⟩+≡

```
  awk '$3=="c=1"' cov.dat |
      awk '{c[$2]++}END{for(a in c)print a, c[a]}' | sort -n |
      awk '{print $0, "c=1"}' > countCov.dat
  awk '$3=="c=15"' cov.dat |
      awk '{c[$2]++}END{for(a in c)print a, c[a]}' | sort -n |
      awk '{print $0, "c=15"}' >> countCov.dat
  plotLine -x Coverage -y "Number of Sites" countCov.dat
```

4.95 In our experiment there were 4 unsequenced sites left with coverage 15; we looked this up in `countCov.dat`. Down from 1899 with coverage 1, but still not zero.

⟨*cli*⟩+≡
```
less countCov.dat
```

4.96 We run `simCov.sh` and find that coverages 49, 61, 96, and 98 result in all nucleotides being sequenced at least once. The message here is, a few nucleotides are bound to escape sequencing even in an idealized sequencing experiment with very high coverage.

⟨*cli*⟩+≡
```
bash simCov.sh | grep all
```

4.97 The message lines include

```
[bwa_index] Pack FASTA... 0.00 sec
[bwa_index] Construct BWT for the packed sequence...
[bwa_index] 0.00 seconds elapse.
[bwa_index] Update BWT... 0.00 sec
[bwa_index] Pack forward-only FASTA... 0.00 sec
[bwa_index] Construct SA from BWT and Occ... 0.00 sec
```

We recognize FASTA, of course, but perhaps also BWT, which stands for "Burrows-Wheeler transform", from Section 3.4, and SA, which stands for "suffix array", from Section 3.3.

4.98 There are at least two ways to find out, we could look up semicolon in the ASCII table in section 7 of the manual, or we could use `od` to give us the specific decimal value we are looking for, $Q = 59$.

⟨*cli*⟩+≡
```
man 7 ascii
printf ";" | od -t u1
```

4.99 Table 4.2 lists each of the four new characters plus semicolon for completeness sake. For each character we get its raw score, its quality score, and the error probability.

Table 4.2 Characters, their raw score, Q, Phred score, Q', and the implied error probability for the quality characters in the example read in Fig. 4.18

Character	Q	Q'	P_e
;	59	26	0.0025
:	58	25	0.0032
9	57	24	0.0040
8	56	23	0.0050
7	55	22	0.0063

⟨*cli*⟩+≡
```
printf ":987" | od -t u1
echo 'e(-2.5 * l(10))' | bc -l
echo 'e(-2.4 * l(10))' | bc -l
echo 'e(-2.3 * l(10))' | bc -l
echo 'e(-2.2 * l(10))' | bc -l
```

4.100 We get two columns, the first contains the sequence header, the second the sequence data.

⟨*cli*⟩+≡
```
fasta2tab reads.fasta | head -n 3
```

```
Read1   AGCAAATTTTGAATATAGGGC...
Read2   ACTCGAGTTGGTCAAATTGCA...
Read3   CCTATATTCAAAATTTGCTCA...
```

4.101 We print the header, the sequence, and again the header. Then we print a semicolon for each nucleotide in the read.

Prog. 4.22 (`tab2fastq.awk`)

⟨*tab2fastq.awk*⟩≡
```
{
  printf("@%s\n%s\n+%s\n", $1, $2, $1)
  for (i = 1; i <= length($2); i++)
    printf(";")
  printf("\n")
}
```

4.102 We convert the FASTA file to tabular format, convert that to FASTQ, and redirect the result to `reads.fastq`.

⟨*cli*⟩+≡
```
fasta2tab reads.fasta |
    awk -f tab2fastq.awk > reads.fastq
```

4.103 Your header should look similar to ours, the version of `bwa` might have changed, though.

4.104 We filter for the first read mapped to the reverse strand, which happens to be Read4 mapped to position 1240 on the template.

⟨*cli*⟩+≡
```
awk '$2==16' adh.sam | head -n 1
```

```
Read4 16 DMADH 1240 60 100M * 0 0 ACCGAT... *
```

We can search for this read and find it at position 1240 on the forward strand.

⟨*cli*⟩+≡
```
keyMat ACCGAT dmAdhAdhdup.fasta
```

We can also retrieve the read and find that it is the reverse complement of what is shown in the SAM file.

⟨*cli*⟩+≡
```
getSeq "Read4$" reads.fasta
```

```
>Read4
...ATCGGT
```

In other words, in a SAM file every read is listed in its forward orientation.

4.105 Our SAM file contains 192,548 bytes, our BAM file 21,192 bytes, which amounts to the large compression ratio of approximately 9. Your values may well differ.

⟨*cli*⟩+≡
```
wc -c adh.sam
wc -c adh.bam
echo '192548 / 21192' | bc -l
```

4.106 We save the decoded BAM file to a temporary SAM file and compare it with the original to find we've lost the header.

⟨*cli*⟩+≡
```
samtools view adh.bam > t.sam
diff t.sam adh.sam
```

This is the intended default behavior, but we can also retain the header with `samtools view -h`.

4.107 We look at the sorted file to find it is sorted with respect to the read position in column 4.

⟨*cli*⟩+≡
```
samtools view adhS.bam | cut -f 4 | head
```

4.108 We already measured that the unsorted file contains 21,192 bytes. The sorted file contains 13,812 bytes, which amounts to an additional compression ratio of roughly 1.5.

⟨*cli*⟩+≡
```
wc -c adhS.bam
echo '21192 / 13812' | bc -l
```

4.109 The index is called `adhS.bam.bai`.

4.110 From the help menu we learn that `b` colors the bases. Our bases are all yellow, which, again according to the help menu, indicates a quality of 20–29. This makes sense as the semicolons we put as quality scores in the FASTQ file correspond to a quality of 26.

4.111 We press `g` and enter

```
DMADH:2000
```

The desired position is now the leftmost position on the screen.

4.112 We generate the reads, count the 352,705,700 nucleotides they contain, roughly 353 million, and convert them to FASTQ. This last step takes a slightly tedious 80 s on our machine; printing 353 million semicolons takes its toll.

⟨*cli*⟩+≡
```
sequencer -c 15 dmChr2L.fasta > reads2L.fasta
cres reads2L.fasta
fasta2tab reads2L.fasta |
    awk -f tab2fastq.awk > reads2L.fastq
```

4.113 We run `makeblastdb` on `dmChr2L.fasta` and call the database `dmChr2L`.

⟨*cli*⟩+≡
```
makeblastdb -in dmChr2L.fasta -out dmChr2L -dbtype nucl
```

4.114 As we've done before, we use `date`—or `gdate` on macOS—to time a command, here `blastn`. By subtracting the start from the end we get the run time. We write the output of `blastn` to the null device, the bin of the command line.

⟨*Time Blast run, Prog. 4.8*⟩≡
```
start=$(date +%s.%N)
blastn -query query.fasta -db dmChr2L > /dev/null
end=$(date +%s.%N)
rt=$(echo "$end - $start" | bc)
```

4.115 We run `timeBlast.sh`, mark its output lines with `blast`, and redirect them to `time.dat`

⟨*cli*⟩+≡
```
bash timeBlast.sh | awk '{print $0, "blast"}' > time.dat
```

4.116 In `timeBlast.sh` we insert `-num_threads 8` in the blast command and run the script again. The run times stay the same, which we find disappointing, but there we are.

4.117 Fig. 4.19 shows that the run time is linear in the number of reads. It took 24 s to map 10^5 reads. There are 3.5×10^6 reads, so it would take $35 \times 24 = 840$ s to map all reads.

4.118 We run the `index` function of `bwa` on chromosome 2L and call the index `dmChr2L`.

⟨*cli*⟩+≡
```
bwa index -p dmChr2L dmChr2L.fasta
```

4.119 We again use `date` to measure time.

⟨*Time bwa run, Prog. 4.9*⟩≡
```
start=$(date +%s.%N)
bwa mem -t 8 -v 1 dmChr2L reads.fastq > /dev/null
end=$(date +%s.%N)
```

```
rt=$(echo "$end - $start" | bc)
```

4.120 It takes `bwa` 1 s to map 10^5 reads, so for all 3.5×10^6 reads it would take 35 s. That's $840/35 = 24$ times faster than Blast with the added advantage that `bwa` can be further sped up by adding more threads, hardware permitting.

⟨*cli*⟩+≡
```
bash timeBwa.sh
```

4.121 We run `timeBwa.sh`, mark its lines `bwa`, and append them to `time.dat`. Then we plot the times with `plotLine`.

⟨*cli*⟩+≡
```
bash timeBwa.sh | awk '{print $0, "bwa"}' >> time.dat
plotLine -L -x "Reads (x1000)" -y "Time (s)" time.dat
```

4.4 Assembly

4.122 We change into the directory for this chapter, make directory `4` for this section, and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/4/
mkdir 4
cd 4/
```

4.123 The overlap between a and b consists of the suffix `TAC` in a and the same prefix in b. It can be visualized in an overlap alignment.

```
GATAC--
--TACAG
```

4.124 We print the sequences to files and apply `al` in overlap mode to get the expected overlap alignment.

⟨*cli*⟩+≡
```
printf ">a\nGATAC\n" > a.fasta
printf ">b\nTACAG\n" > b.fasta
al -o a.fasta b.fasta
```

```
Query   1 GATAC-- 5
            |||
Subject 1 --TACAG 5
```

4.125 We calculate the global and local alignments.

⟨*cli*⟩+≡
```
al a.fasta b.fasta
al -l a.fasta b.fasta
```

We find that the global alignment isn't very convincing.

```
Query   1 GATAC 5
            | |
Subject 1 TACAG 5
```

The local alignment, on the other hand, is just the overlap.

```
Query   3 TAC 5
          |||
Subject 1 TAC 3
```

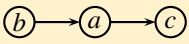
4.126 When merging a and b we get `GATACAG`.

4.127 We run `sass` with a score threshold of 3 to get the expected contig.

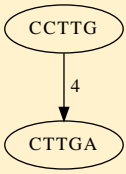
⟨*cli*⟩+≡

```
sass -t 3 a.fasta b.fasta
```

4.128 We write the matches suffix to prefix and get



4.129 There's only one overlap between r_1 and r_2, `CTTG`, which has length 4.



4.130 We print the two reads to two files and visualize them with `olga`. We could also have printed the reads to a single file or piped them directly through `olga`.

⟨*cli*⟩+≡

```
printf ">r1\nCCTTG\n" > r1.fasta
printf ">r2\nCTTGA\n" > r2.fasta
olga r1.fasta r2.fasta | dot -T x11
```

4.131 We copy `pentamers.fasta`, count its 12 entries, and find that r_1 and r_2 are among them.

⟨*cli*⟩+≡

```
cp $BEB/data/pentamers.fasta .
grep '^>' pentamers.fasta | wc -l
grep CTTG pentamers.fasta
```

4.132 We draw the overlap graph, where the leftmost node is r_1 and the right-hand neighbor it points to is r_2.

⟨*cli*⟩+≡

```
olga pentamers.fasta | dot -T x11
```

4.133 We restrict `olga` to overlaps of miminum length 3.

⟨*cli*⟩+≡
```
olga -k 3 pentamers.fasta | dot -T x11
```

4.134 We begin with `CCTTG`, append the `A` from `CTTGA`, then the `AT` from `TGAAT`, and so on, to finally get

`CCTTGAATTTCTAGTTCCTC`

We check our answer by running `sass`.

⟨*cli*⟩+≡
```
sass -t 3 pentamers.fasta
```

4.135 We run `sass` with minimum overlap 3 and get two contigs.

⟨*cli*⟩+≡
```
sass -t 3 reads.fasta
```

```
>Contig_1
TCTAGTTCCTC
>Contig_2
TTGAATTTC
```

Contig 2 overlaps contig 1 by two nucleotides. So when we assemble with an overlap of 2, we get a single contig two nucleotides shorter at the 5' end than the template, `CCTTGA`...

⟨*cli*⟩+≡
```
sass -t 2 reads.fasta
```

```
>Contig_1
TTGAATTTCTAGTTCCTC
```

4.136 To write `reduce.sh`, we remove the header lines, sort the reads (which we assume occupy only a single line), pick the unique reads, and convert them to FASTA output.

Prog. 4.23 (`reduce.sh`)

⟨*reduce.sh*⟩≡
```
grep -v '^>' |
    sort |
    uniq |
    awk '{printf ">%d\n%s\n", NR, $1}'
```

We plot the reduced set of reads to get Fig. 4.25B.

⟨*cli*⟩+≡
```
bash reduce.sh < reads.fasta | olga -k 3 | dot -T x11
```

4.137 We run `olga` on the reduced set of reads with minimum overlap 2 to get Fig. 4.38, where the two subgraphs of Fig. 4.25B are joined together.

⟨*cli*⟩+≡

```
bash reduce.sh < reads.fasta | olga -k 2 | dot -T x11
```

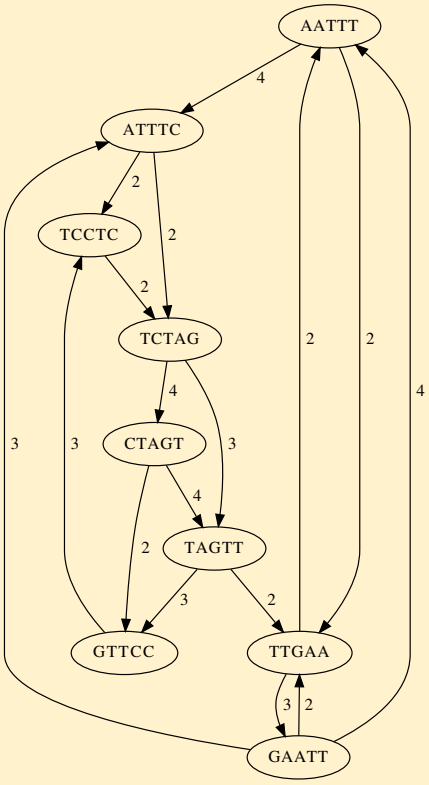


Fig. 4.38 Overlap graph for the same set of reads as Fig. 4.25B, except now the minimum overlap is 2 rather than 3, which leads to the merger of the two subgraphs

4.138 We calculate the assembly to find a single contig, but its last five nucleotides don't match.

⟨*cli*⟩+≡

```
sass -t 3 reads.fasta > cont.fasta
al cont.fasta template.fasta
```

```
Query   1 ------ATTTCTAGTTCCTTGAAT 18
                 ||||||||||||
Subject 1 CCTTGAATTTCTAGTTCCTC---- 20
```

4.139 We are given some feedback about the progress of `velveth`, in particular, that it constructs a "splay table", a table where each column is stored in a separate file, and destroys it again. This is presumably the hash table from which `velveth` extracts the read overlaps based on exact matches of the 21-mers.

4.140 The directory `assem` contains three files, `Log`, `Roadmaps`, and `Sequences`. The "roadmaps" are used for assembly, the sequences are our original reads.

⟨*cli*⟩+≡
```
ls assem/
```

4.141 With a hash length of 32, we get a polite message saying that `velveth` can only handle k-mers up to length 31.

⟨*cli*⟩+≡
```
velveth assem/ 32 -short -fasta reads.fasta
```

4.142 We count the contigs by counting their header lines, 1332 in our case, yours is bound to be different but should be similar.

⟨*cli*⟩+≡
```
grep '^>' assem/contigs.fa |
    wc -l
```

4.143 We repeat the sequencing with coverage 2 and run the assembly with an expected coverage of 2. This time we get 1870 contigs, that is, more than the 1332 before. Our investment in sequencing doesn't seem to have paid off so far.

⟨*cli*⟩+≡
```
sequencer -e 0 -c 2 rg.fasta > reads.fasta
velveth assem/ 21 -short -fasta reads.fasta
velvetg assem/ -exp_cov 2
grep '^>' assem/contigs.fa | wc -l
```

4.144 The arguments for `sequencer`, `velveth`, and `velvetg` are not mandatory, so we put them in square brackets. Then we print the complete message.

⟨*Print usage message, Prog. 4.10*⟩+≡
```
m = m " [-v s=<sequencer_args"
m = m " -v t=<velveth_args>"
m = m " -v g=<velvetg_args>]"
print m
```

4.145 We construct the command for `velveth` and redirect its output to the null device. We also report that the program is running.

⟨*Hash reads, Prog. 4.10*⟩≡
```
cmd = "velveth assem/ 21 %s reads.fasta "
cmd = cmd "> /dev/null"
cmd = sprintf(cmd, h)
printf("# running velveth...")
system(cmd)
```

```
print "done"
```

4.146 We work on the same pattern as with `velveth`: We construct the command for `velvetg` with output redirection to the null device and report it is running.

⟨*Assemble reads, Prog. 4.10*⟩≡
```
cmd = "velvetg assem/ %s "
cmd = cmd "> /dev/null"
cmd = sprintf(cmd, g)
printf("# running velvetg...")
system(cmd)
print "done"
```

4.147 We run `simShot.awk` with coverage 5 and no sequencing error to get 534 contigs. That's better than the 1870 contigs we got with coverage 2, but still nowhere close to the single contig we're hoping for.

⟨*cli*⟩+≡
```
awk -f simShot.awk -v c=5 -v t=rg.fasta -v s="-e 0" \
    -v h="-short -fasta" -v g="-exp_cov 5"
```

4.148 We write a loop to drive the shotgun simulation and save the results to `simShot.dat`, which we plot with `plotLine`.

⟨*cli*⟩+≡
```
for a in $(seq 20)
do
    awk -f simShot.awk -v c=${a} -v t=rg.fasta -v s="-e 0" \
          -v h="-short -fasta" -v g="-exp_cov ${a}"
done > simShot.dat
plotLine -x Coverage -y Contigs simShot.dat
```

4.149 In our simulations, coverage 16 and 17 gave us a single contig each, but coverage 18 gave us 2 contigs, back to 1 for coverages 19 and 20. So this varies, but it seems coverage 20 is pretty safe.

⟨*cli*⟩+≡
```
grep -v '^#' simShot.dat
```

```
1       1379
2       1852
...
15      3
16      1
17      1
18      2
19      1
20      1
```

4.150 We need the probability that a nucleotide is not sequenced times the length of the template:

$$580076 \times e^{-10} \approx 26.$$

That is, the expected combined length of all gaps in the assembly is 26.

⟨*cli*⟩+≡
```
echo '580076*e(-10)' | bc -l
```

4.151 With coverage 15 we expect 0.2 unsequenced nucleotides, with coverage 20 that's down to 0.1%.

⟨*cli*⟩+≡
```
echo '580076*e(-15)' | bc -l
echo '580076*e(-20)' | bc -l
```

4.152 By solving

$$Le^{-c} = 1$$

for c we find

$$c = -\ln\left(\frac{1}{L}\right).$$

In our case the theoretical coverage $c \approx 13.3$.

⟨*cli*⟩+≡
```
echo '-l(1/580076)' | bc -l
```

4.153 Again we write for the desired coverage

$$c = -\ln\left(\frac{0}{L}\right),$$

but since $\ln(0) = -\infty$, $c = \infty$ in this case. In other words, a combined gap length of 0 cannot be achieved. However, in real-world shotgun sequencing things like sequencing errors and repeats create much more problems than the unattainability of certainty with a stochastic process like sequencing.

4.154 We run `simShot.awk` without the sequencing error argument ten times and find between 11 and 56 contigs.

⟨*cli*⟩+≡
```
c=20
for a in $(seq 10)
do
    awk -f simShot.awk -v c=$c -v t=rg.fasta \
          -v h="-short -fasta" -v g="-exp_cov $c"
done
```

4.155 We ran the simulation ten times with coverage 30 and got between 17 and 58 contigs. Greater coverage doesn't seem to help in this case.

⟨*cli*⟩+≡
```
c=30
awk -f simShot.awk -v c=$c -v t=rg.fasta \
    -v h="-short -fasta" -v g="-exp_cov $c"
```

4.156 We calculate the shustring lengths of the randomized sequence, mark them *ran*, and append them to the data file `sl.dat`. Then we plot `sl.dat` with position measured in units of 100 kb with `plotLine`.

⟨*cli*⟩+≡
```
shustring -l rg.fasta | tail -n +3 |
    awk 'NR%100==0{print $1, $2, "ran"}' >> sl.dat
awk '{print $1/100000, $2, $3}' sl.dat |
plotLine -x "Position (100 kb)" -y Shustring-Length
```

4.157 We set the coverage to 20, the sequencing error to zero and find 219 contigs. That is a lot more than the single contig we got when sequencing the randomized version of the genome.

⟨*cli*⟩+≡
```
c=20
awk -f simShot.awk -v c=$c -v s="-e 0" \
    -v t=mgGenome.fasta -v h="-short -fasta" \
    -v g="-exp_cov $c"
```

4.158 We got 187 contigs. Not a great improvement compared to our single-end result, but there we are. Expect to see paired-end reads in real shotgun sequencing.

4.159 We've seen these dot plots before; the lines parallel to the main diagonal are matches on the forward strand, the others matches on the reverse strand. Together they cover almost the entire template. The plot of your contigs is bound to look different. Fig. 4.39 shows another example.

4.160 For the median, we sort the contig lengths and look up the sixth element, 52. For the mean, we calculate the average, 47.6.

⟨*cli*⟩+≡
```
n="31\n66\n74\n6\n5\n79\n83\n52\n10\n90\n28\n"
printf $n | sort -n | tail -n +6 | head -n 1
printf $n | awk '{s+=$1}END{print s/NR}'
```

4.161 We calculate the total contig length of 524. Then we sort the contigs, calculate their cumulative length, and report the contig length where the cumulative length is at least 262, which is 74.

⟨*cli*⟩+≡
```
printf $n | awk '{s+=$1}END{print s}'
printf $n | sort -n | awk '{s+=$1;if(s>=262){print $1;exit}}'
```

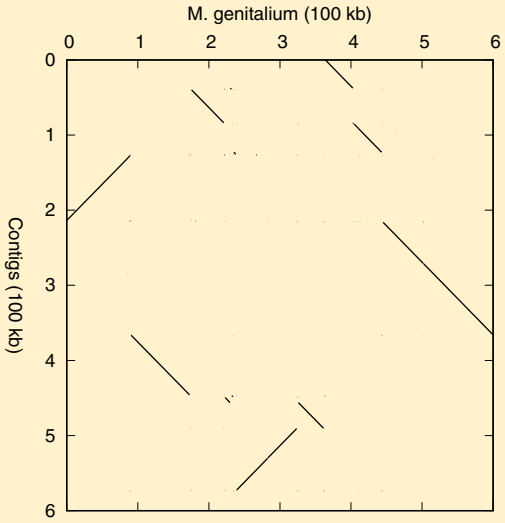


Fig. 4.39 Dot plot of the contigs generated in a simulated shotgun sequencing of *M. genitalium*

4.162 We calculate the total contig length, 575,125 in our case. Then we sort the contig lengths and find an N_{50} of 82,061.

⟨*cli*⟩+≡
```
cres -s assem/contigs.fa | awk '/^>/{print $2}' |
    awk '{s+=$1}END{print s}'
cres -s assem/contigs.fa | awk '/^>/{print $2}' | sort -n |
    awk '{s+=$1;if(s>=575125/2){print $1; exit}}'
```

4.5 Multiple Sequences

4.163 We change into the directory for this chapter, 4, make the directory for this section, 5, and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/4/
mkdir 5
cd 5/
```

4.164 If $|S_i|$ is the length of S_i, the two-dimensional alignment matrix has $|S_1| \times |S_2|$ entries. If we want to be exact, we recall that each sequence is prefixed by a gap, so the number of cells in a two-dimensional alignment matrix is $(|S_1|+1) \times (|S_2|+1)$.

4.165 The alignment matrix for aligning n sequences length ℓ contains $(\ell+1)^n$ cells.

4.166 The space and time requirements for optimal alignment are already a burden for pairs of sequences and become prohibitive for multiple sequences. For this reason, optimal multiple sequence alignment is hardly ever done by computer—aligning by hand would be more realistic.

4.167 We copy `sample.fasta` to our working directory and split it by looping over the sequence names and calling `getSeq` on each one.

⟨*cli*⟩+≡
```
cp $BEB/data/sample.fasta .
for a in a b c d
do
    getSeq $a sample.fasta > ${a}.fasta
done
```

4.168 We run `al` on `a.fasta` and `b.fasta` to get an alignment with two gaps in a. That's our multiple sequence alignment at this stage.

⟨*cli*⟩+≡
```
al a.fasta b.fasta
```

4.169 The sequence data is contained in subsequent target lines.

⟨*Print sequence data, Prog. 4.11*⟩≡
```
$1 == target && NR > 3 {
  print $3
}
```

4.170 We run `al` twice to extract first the query, then the subject. We save both in `anc.fasta` and view them in tabular format, as this makes it easier to compare the sequences than FASTA.

⟨*cli*⟩+≡
```
al a.fasta b.fasta |
    awk -f al2fasta.awk -v target=Query > anc.fasta
al a.fasta b.fasta |
    awk -f al2fasta.awk -v target=Subject >> anc.fasta
fasta2tab anc.fasta
```

```
a GAGCTCA-ACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCGC--GTC
b GAGGTCAGACCGGTTGGTAGGCGTTCATTATTGGGAGGCAGGCGTATGTC
```

4.171 We align `r.fasta` to `c.fasta` and find that there are no new gaps in the reference, but one gap in c. So we can just add c to our alignment. Then we make sure everything looks ok by printing the alignment in tabular format.

⟨*cli*⟩+≡
```
al r.fasta c.fasta |
    awk -f al2fasta.awk -v target=Subject >> anc.fasta
fasta2tab anc.fasta
```

```
a GAGCTCA-ACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCGC--GTC
b GAGGTCAGACCGGTTGGTAGGCGTTCATTATTGGGAGGCAGGCGTATGTC
c GAGCTCAGACGGGTTGGT--GCGTTCATTAATGGGAGGCTGGCGTACGTC
```

4.172 We align `r.fasta` to `d.fasta` and find that again there are no new gaps in the reference. So we just extract d and add it to our alignment. Using `fasta2tab`, we find that the reference is unchanged, and we've constructed our final alignment.

⟨*cli*⟩+≡
```
al r.fasta d.fasta |
    awk -f al2fasta.awk -v target=Subject >> anc.fasta
fasta2tab anc.fasta
```

```
a GAGCTCA-ACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCGC--GTC
b GAGGTCAGACCGGTTGGTAGGCGTTCATTATTGGGAGGCAGGCGTATGTC
c GAGCTCAGACGGGTTGGT--GCGTTCATTAATGGGAGGCTGGCGTACGTC
d GAGCTCA-ACCGGTTGGTAGGCGTTCATTATTAGGACGGAGACGC--GTC
```

4.173 The sum-of-pairs score of the first column is $1-3-3=-5$, the sum-of-pairs score of the second column is $0-2-2=-4$, so the sum-of-pairs score of the whole alignment is $-5-4=-9$, which we test with `sops`.

⟨*cli*⟩+≡
```
printf ">s1\nA-\n>s2\nA-\n>s3\nTT\n" | sops
```

4.174 We apply `sops` to `anc.fasta` to find that its sum-of-pairs score is 111.

⟨*cli*⟩+≡
```
sops anc.fasta
```

4.175 We align c with a, store it in `anc2.fasta`, and notice that c is gapped.

⟨*cli*⟩+≡
```
al c.fasta a.fasta |
    awk -f al2fasta.awk -v target=Query > anc2.fasta
al c.fasta a.fasta |
    awk -f al2fasta.awk -v target=Subject >> anc2.fasta
fasta2tab anc2.fasta
```

So we replace the gaps in the aligned c by `N` and store it in `r.fasta`.

⟨*cli*⟩+≡
```
al c.fasta a.fasta | tail -n +7 | grep Q | tr '-' 'N' |
    awk '{printf ">r\n%s\n", $3}' > r.fasta
```

Now we add sequences b and d to the alignment to get our final alignment, which looks good in tabular format.

⟨*cli*⟩+≡
```
al r.fasta b.fasta |
    awk -f al2fasta.awk -v target=Subject >> anc2.fasta
al r.fasta d.fasta |
```

```
    awk -f al2fasta.awk -v target=Subject >> anc2.fasta
  fasta2tab anc2.fasta
```

```
c GAGCTCAGACGGGTTGGT--GCGTTCATTAATGGGAGGCTGGCGTACGTC
a GAGCTCA-ACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCG--CGTC
b GAGGTCAGACCGGTTGGTAGGCGTTCATTATTGGGAGGCAGGCGTATGTC
d GAGCTCA-ACCGGTTGGTAGGCGTTCATTATTAGGACGGAGACG--CGTC
```

4.176 We run `sops` on `anc2.fasta` and find that the sum-of-pairs score is now 119, up from previously 111.

⟨*cli*⟩+≡

```
  sops anc2.fasta
```

4.177 The difference is in the gapped region on the right hand side of the alignment. When anchored on a, this is

```
a GC--GTC
b GTATGTC
c GTACGTC
d GC--GTC
```

When anchored on c, it is

```
c GTACGTC
a G--CGTC
b GTATGTC
d G--CGTC
```

This gives an extra match between `C` in d and c and hence the higher score.

4.178 Sequence a on top is the anchor, any match to it is a dot. Sequence d is shown with the gap we had also found. Sequence b is truncated at its 3' end, because Blast calculates local alignments. Sequence c is missing altogether, its central gap appears to make the two flanking alignments non-significant.

4.179 We run anchor Blast with output format 4 to find that all nucleotides are printed rather than just the mismatches.

⟨*cli*⟩+≡

```
  blastn -task blastn -query a.fasta \
           -subject subject.fasta -outfmt 4
```

```
a GAGCTCA-ACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCGCGTC
d GAGCTCA-ACCGGTTGGTAGGCGTTCATTATTAGGACGGAGACGCGTC
b GAGGTCAGACCGGTTGGTAGGCGTTCATTATTGGGAGGCAGGCG
```

4.180 We run anchor Blast with output format 1 and see that the insertion in b is written as a subscript rather than accommodated by a gap in a and d.

⟨*cli*⟩+≡

```
  blastn -task blastn -query a.fasta \
           -subject subject.fasta -outfmt 1
```

```
a GAGCTCAACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCGCGTC
d .................A.................C....A......
b ...G.............A.............G.....C.....
         \
         |
         G
```

4.181 We run `al` on `sample.fasta` as query and subject and extract the number of errors. Then we can fill in the distance matrix line by line. The individual distances range from 3 to 12.

⟨*cli*⟩+≡
```
al sample.fasta sample.fasta | grep Errors
```

	a	*b*	*c*	*d*
a	0	8	10	3
b	8	0	7	9
c	10	7	0	12
d	3	9	12	0

4.182 We write the distance matrix, convert it to a tree with `upgma`, and plot it with `plotTree`.

⟨*cli*⟩+≡
```
echo 4 > sample.dist
echo a 0 7 10 3 >> sample.dist
echo b 8 0 7 9 >> sample.dist
echo c 10 7 0 12 >> sample.dist
echo d 3 9 12 0 >> sample.dist
upgma sample.dist | plotTree
```

4.183 We run `al` on `a.fasta` and `d.fasta`, save their alignment in `prog.fasta`, and see that it is gap-free.

⟨*cli*⟩+≡
```
al a.fasta d.fasta |
    awk -f al2fasta.awk -v target=Query > prog.fasta
al a.fasta d.fasta |
    awk -f al2fasta.awk -v target=Subject >> prog.fasta
fasta2tab prog.fasta
```

```
a GAGCTCAACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCGCGTC
d GAGCTCAACCGGTTGGTAGGCGTTCATTATTAGGACGGAGACGCGTC
```

4.184 We add the alignment of `b.fasta` and `c.fasta` to `prog.fasta` and see that the two pairwise alignments are out of whack.

⟨*cli*⟩+≡
```
al b.fasta c.fasta |
```

```
      awk -f al2fasta.awk -v target=Query >> prog.fasta
  al b.fasta c.fasta |
      awk -f al2fasta.awk -v target=Subject >> prog.fasta
  fasta2tab prog.fasta
```

```
a GAGCTCAACCGGTTGGTTGGCGTTCATTATTAGGAGGGAGGCGCGTC
d GAGCTCAACCGGTTGGTAGGCGTTCATTATTAGGACGGAGACGCGTC
b GAGGTCAGACCGGTTGGTAGGCGTTCATTATTGGGAGGCAGGCGTATGTC
c GAGCTCAGACGGGTTGGT--GCGTTCATTAATGGGAGGCTGGCGTACGTC
```

4.185 We save the stream-edited `prog.fasta` to a temporary file, move that back to `prog.fasta`, and check this worked.

⟨*cli*⟩+≡

```
  sed 's/CAAC/CA-AC/' prog.fasta > tmp.fasta
  mv tmp.fasta prog.fasta
  fasta2tab prog.fasta
```

4.186 We substitute `CGCG` by `CG--CG` and make sure this worked. The gap position is unambiguous as the two alternative arrangements, `r--` or `-r-`, would give a smaller score. Then we replace the old version of `prog.fasta` with the new one and check again. Now all sequences have the same length and we are done.

⟨*cli*⟩+≡

```
  sed 's/CGCG/CG--CG/' prog.fasta | fasta2tab
  sed 's/CGCG/CG--CG/' prog.fasta > tmp.fasta
  mv tmp.fasta prog.fasta
  fasta2tab prog.fasta
```

4.187 We apply `sops` to `prog.fasta` and find that its sum-of-pairs score is 119, the best possible.

⟨*cli*⟩+≡

```
  sops prog.fasta
```

4.188 We pipe the `mafft` output through `sops` to find that the sum-of-pairs score is 71, much worse than the scores of the two alignments we constructed semi-manually, 111 and 119. When we look at the alignment, we can see that the terminal gaps in sequences *a* and *d* don't make much sense.

⟨*cli*⟩+≡

```
  mafft sample.fasta | sops
  mafft sample.fasta | fasta2tab
```

```
a gagctca-accggttggttggcgttcattattaggagggaggcgcgtc--
b gaggtcagaccggttggtaggcgttcattattgggaggcaggcgtatgtc
c gagctcagacgggttggt--gcgttcattaatgggaggctggcgtacgtc
d gagctca-accggttggtaggcgttcattattaggacggagacgcgtc--
```

4.189 We calculate the alignment with the `--auto` option and pipe it through `sops` to get a score of 87—better than 71, but still worse than 111 or 119. Again, inspection of the alignment shows that the second set of gaps in sequences *a* and *d* has drifted too far downstream to capture the maximum amount of homology.

⟨*cli*⟩+≡
```
mafft --auto sample.fasta | sops
mafft --auto sample.fasta | fasta2tab
```

```
a gagctca-accggttggttggcgttcattattaggagggaggcgcgt--c
b gaggtcagaccggttggtaggcgttcattattgggaggcaggcgtatgtc
c gagctcagacgggttggt--gcgttcattaatgggaggctggcgtacgtc
d gagctca-accggttggtaggcgttcattattaggacggagacgcgt--c
```

4.190 We pipe the alignment through `sops` to find the score of 119 we had hoped for. We can also see directly that the downstream gaps in sequences *a* and *d* are now in the correct place. It took us a few attempts to get there, so a given multiple sequence alignment is not necessarily optimal with respect to the tool used.

⟨*cli*⟩+≡
```
mafft --parttree sample.fasta | sops
mafft --parttree sample.fasta | fasta2tab
```

```
a gagctca-accggttggttggcgttcattattaggagggaggcg--cgtc
b gaggtcagaccggttggtaggcgttcattattgggaggcaggcgtatgtc
c gagctcagacgggttggt--gcgttcattaatgggaggctggcgtacgtc
d gagctca-accggttggtaggcgttcattattaggacggagacg--cgtc
```

4.191 We copy `primates.fasta` to our working directory. Then we write a loop for appending the sequences to `simians.fasta`. To make sure we are not appending to existing data, we start by emptying `simians.fasta`.

⟨*cli*⟩+≡
```
cp $BEB/data/primates.fasta .
printf "" > simians.fasta
for a in P_trog P_pan H_sap G_gor P_pyg H_lar
do
    getSeq ${a} primates.fasta >> simians.fasta
done
```

4.192 We pipe the alignment through `sops` and find that the sum-of-pairs score is 140,228.

⟨*cli*⟩+≡
```
mafft --auto simians.fasta | sops
```

4.193 We run `mafft` with `--parttree`, which increases the run time from one second to over a minute. In spite of this heavy investment in run time, the new sum-of-pairs score, 140,084, is actually worse than the 140,228 we got before.

⟨*cli*⟩+≡

```
mafft --parttree simians.fasta | sops
```

4.194 `a` is `^[0-9]_`, which is any number beginning at a line followed by an underscore; `b` is nothing. So the substitution amounts to deleting any leading digit followed by an underscore.

4.195 In most phylogenies of the simians, chimps are most closely related to humans. However, in the guide tree chimps are most closely related to gorillas. This just goes to show that guide trees are rather approximate phylogenies.

Chapter 5
Evolution Between Species: Phylogeny

5.1 Trees of Life

5.1 We change into the directory for chapters, make a directory for this chapter, `5`, change into it, make a directory for this section, `1`, and change into that.

⟨*cli*⟩≡
```
cd $BEB/ch/
mkdir 5
cd 5/
mkdir 1
cd 1/
```

5.2 In Newick only internal nodes are delimited by parentheses, subtrees are separated by commas, and the tree is terminated by a semicolon.

5.3 Without internal node labels, our tree becomes

```
(B,(D,E));
```

5.4 We print the Newick string of our tree and pipe it through `plotTree`.

⟨*cli*⟩+≡
```
printf "(B:1,(D:1,E:1));\n" | plotTree
```

5.5 Again, we pipe the Newick string through `pipePlot`. Notice that the branch length from gorilla to the root is 1.1, which is the branch length of human or chimp (1.0) plus the distance from their most recent common ancestor to the root (0.1).

⟨*cli*⟩+≡
```
printf "((human:1,chimp:1):0.1,gorilla:1.1);\n" | plotTree
```

5.6 We add "ancestor" and "root" to the tree and plot it.

⟨*cli*⟩+≡
```
printf "((human:1,chimp:1)ancestor:0.1,gorilla:1.1)root;\n" |
    plotTree
```


B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_12

5.7 We print a single node with three children and plot it. Notice that the gorilla branch now has the combined length of the two branches it was generated from, 1.2.

⟨*cli*⟩+≡
```
printf "(human:1,chimp:1,gorilla:1.2);\n" | plotTree
```

5.8 Fig. 5.5 is an ordinary tree with a root and three leaves. The "unrooted" aspect is expressed through its radial layout, which means there is no preferred direction of time, the position of the true ancestor is uncertain. In other words, trees with bifurcating roots are called rooted and drawn like Fig. 5.4, trees with trifurcating roots are called unrooted and drawn in radial layout like Fig. 5.5.

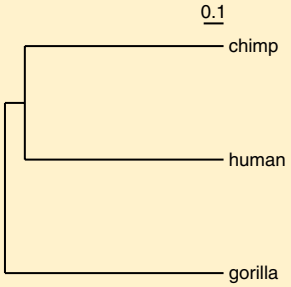


Fig. 5.19 Midpoint-rooted phylogeny of human, chimp, and gorilla

5.9 We apply `midRoot` to our unrooted tree and get Fig. 5.19. This is the same tree we started out with, Fig. 5.3, since branch rotation does not change the evolutionary history. In other words, `(A,B);` tells the same evolutionary story as `(B,A);`.

⟨*cli*⟩+≡
```
printf "(human:1,chimp:1,gorilla:1.2);\n" | midRoot |
    plotTree
```

5.10 A branch without length is drawn by `plotTree` as a branch with length 1.

5.11 We leave out the branch lengths and just concentrate on getting the labels right.

⟨*cli*⟩+≡
```
printf "(((1,3)2,5)4,(7,9)8)6;\n" | plotTree
```

5.12 The preorder traversal of the tree in Fig. 5.7 visits nodes 6, 4, 2, 1, 3, 5, 8, 7, and 9. We check our result with `travTree`.

⟨*cli*⟩+≡
```
printf "(((1,3)2,5)4,(7,9)8)6;\n" | travTree
```

```
#Label  Parent  Dist.  Type
6       none    0      root
4       6       0      internal
```

```
2       4       0       internal
1       2       0       leaf
3       2       0       leaf
5       4       0       leaf
8       6       0       internal
7       8       0       leaf
9       8       0       leaf
```

5.13 The inorder traversal visits 1, 3, 2, 5, 4, 7, 9, 8, and 6, which we check with `travTree`.

⟨*cli*⟩+≡

```
printf "(((1,3)2,5)4,(7,9)8)6;\n" | travTree -i
```

```
#Label  Parent  Dist.  Type
1       2       0      leaf
3       2       0      leaf
2       4       0      internal
5       4       0      leaf
4       6       0      internal
7       8       0      leaf
9       8       0      leaf
8       6       0      internal
6       none    0      root
```

5.14 The postorder traversal visits 3, 1, 5, 2, 9, 7, 8, 4, and 6, which we check with `travTree`.

⟨*cli*⟩+≡

```
printf "(((1,3)2,5)4,(7,9)8)6;\n" | travTree -o
```

```
#Label  Parent  Dist.  Type
3       2       0      leaf
1       2       0      leaf
5       4       0      leaf
2       4       0      internal
9       8       0      leaf
7       8       0      leaf
8       6       0      internal
4       6       0      internal
6       none    0      root
```

5.15 With `-a`, the branch lengths correspond to absolute times. As a result, the terminal branches all end at the same imaginary zero time. Without `-a`, the branch

lengths are proportional to the number of mutations along the branches. In that case the branches fluctuate around the imaginary zero, as shown in Fig. 5.20.

⟨*cli*⟩+≡

```
genTree | plotTree
```

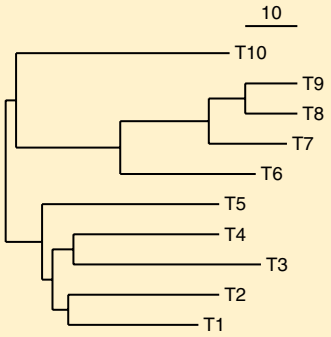


Fig. 5.20 Random phylogeny of ten taxa with branch lengths proportional to the number of mutations

5.16 With `-t` 100, the branches fluctuate widely around the zero line, as shown in Fig. 5.21A. The fluctuations are much smaller with `-t` 10,000, as shown in Fig. 5.21B.

⟨*cli*⟩+≡

```
genTree -t 100 | plotTree
genTree -t 10000 | plotTree
```

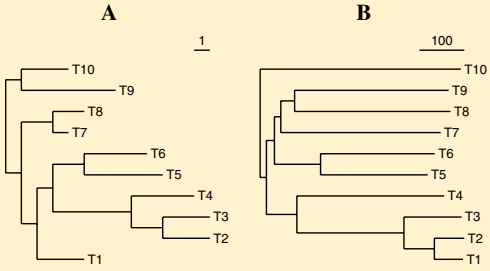


Fig. 5.21 Random phylogeny with an expected 100 mutations (**A**) and 10,000 mutations (**B**)

5.17 Given a rooted tree of four taxa, there are seven edges to which we can add the new root. So there are $3 \times 5 \times 7 = 105$ trees with five taxa.

5.18 We directly transcribe the equation we've been given.

⟨*Calculate number of trees, Pr. 5.1*⟩≡
```
nt = 1
for (i = 3; i <= n; i++)
  nt *= 2 * i - 3
```

5.19 We print the number of taxa, `n`, and the number of trees, `nt`.

⟨*Print number of trees, Pr. 5.1*⟩≡
```
print n, nt
```

5.20 There are 34,459,425, or 34 million, distinct trees with ten taxa. No wonder we never saw the same one again.

⟨*cli*⟩+≡
```
awk -f numTrees.awk -v n=10
```

5.21 We wrap `numTree.awk` in a loop and plot the result with the y axis log-scaled.

⟨*cli*⟩+≡
```
for a in $(seq 2 100)
do
    awk -f numTrees.awk -v n=$a
done | plotLine -l y -x Taxa -y Trees
```

Like with the number of possible alignments, we are looking at hyperexponential growth.

5.2 Rooted Trees

5.22 We change into the chapter directory, `5`, make the directory for this section, `2`, and change into it.

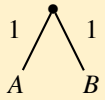
⟨*cli*⟩≡
```
cd $BEB/ch/5/
mkdir 2
cd 2/
```

5.23 Here is the complete distance matrix:

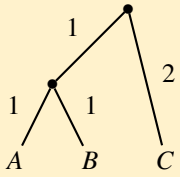
	A	*B*	*C*	*D*
A	-			
B	1	-		
C	3	3	-	
D	3	3	2	-

5.24 Distances are symmetric in the sense that the distance between London and Hamburg is equal to the distance between Hamburg and London. As a result, distance matrices are symmetrical and one triangle implies the other.

5.25 Taxa A and B have the smallest distance, 2, so the corresponding tree is



5.26 We pick the pair $(A, B), C$ to get the tree

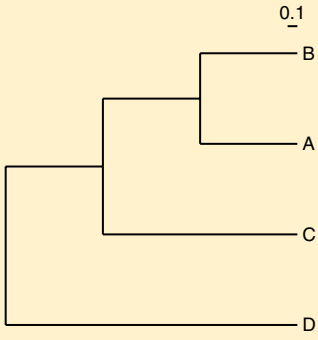


5.27 We put the distances into the file `ex1.dist`, either on the command line with `printf` or with an editor. Then we run `upgma` and `plotTree`.

⟨*cli*⟩+≡

```
printf "4\n"          > ex1.dist
printf "A 0 2 4 6\n" >> ex1.dist
printf "B 2 0 4 6\n" >> ex1.dist
printf "C 4 4 0 6\n" >> ex1.dist
printf "D 6 6 6 0\n" >> ex1.dist
upgma ex1.dist | plotTree
```

The tree we get looks a bit different from Fig. 5.10B, but it's meaning is the same.



5.28 We run `upgma` on our example distances with matrix printing switched on. There are three intermediate matrices followed by the final tree.

⟨*cli*⟩+≡

```
upgma -m ex1.dist
```

```
4
A  0  2  4  6
B  2  0  4  6
C  4  4  0  6
D  6  6  6  0
3
C      0  6  4
D      6  0  6
(A,B)  4  6  0
2
D          0  6
(C,(A,B))  6  0
(D:3,(C:2,(A:1,B:1):1):1);
```

5.29 Fig. 5.22 shows the three steps required to calculate the tree.

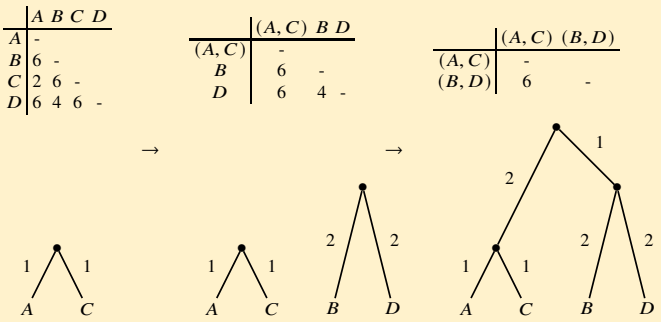


Fig. 5.22 Tracing the Upgma algorithm

To check our results, we generate the distance matrix and apply `upgma` to it.

⟨*cli*⟩+≡

```
printf "4\n"          > ex2.dist
printf "A 0 6 2 6\n" >> ex2.dist
printf "B 6 0 6 4\n" >> ex2.dist
printf "C 2 6 0 6\n" >> ex2.dist
printf "D 6 4 6 0\n" >> ex2.dist
upgma -m ex2.dist
```

```
4
A  0  6  2  6
B  6  0  6  4
C  2  6  0  6
D  6  4  6  0
```

```
3
B       0  4  6
D       4  0  6
(A,C)   6  6  0
2
(A,C)   0  6
(B,D)   6  0
((A:1,C:1):2,(B:2,D:2):1);
```

5.30 A Upgma tree not only implies a molecular clock, it also implies that the stochastic clock, the number of mutations per unit time, behaves like a conventional clock, where the second hand moves exactly once per second, rather than on average.

5.31 The distance matrix implied by the tree is

	A	*B*	*C*	*D*
A	0	6	6	8
B	6	0	4	8
C	6	4	0	8
D	8	8	8	0

which is quite different from the input in Fig. 5.11.

5.32 There are 4 distinct entries, 4, 5, 7, and 10. That's one more than allowed under Upgma.

5.33 Let's switch 6 and 4 in the first column of the distance matrix Fig. 5.10A to get

	A	*B*	*C*	*D*
A	-			
B	2	-		
C	6	4	-	
D	4	6	6	-

Now we consider taxa A, B, C, for which we have $d_{A,B} = 2$, $d_{A,C} = 6$, and $d_{B,C} = 4$. Here the three-point criterion doesn't hold, as the maximum of these three distances, 6, is unique among the three, while the three-point criterion posits that it shouldn't be.

5.34 We copy the sequence data to our current directory and run `cres` in *separate* mode, `-s`. We find that all four sequences are 896 characters long, but note that the *Pongo* sequence contains a gap character, so it consists only of 895 nucleotides.

⟨*cli*⟩+≡

```
cp $BEB/data/hominidae.fasta .
cres -s hominidae.fasta
```

```
>Homo: 896
Residue Count Fraction
A       273   0.305
C       297   0.331
G       96    0.107
T       230   0.257
>Pan: 896
Residue Count Fraction
A       277   0.309
C       291   0.325
G       91    0.102
T       237   0.265
>Gorilla: 896
Residue Count Fraction
A       278   0.31
C       292   0.326
G       95    0.106
T       231   0.258
>Pongo: 896
Residue Count Fraction
-       1     0.00112
A       281   0.314
C       309   0.345
G       93    0.104
T       212   0.237
```

5.35 With default options, we find 219 polymorphic sites.

⟨*cli*⟩+≡

```
pps hominidae.fasta | head -n 1
```

```
>Positions (219)
```

This is reduced to 218 if we exclude the gapped position with `-g`. So by default, `pps` counts gapped sites as polymorphic, though many authors skip gapped positions by default.

⟨*cli*⟩+≡

```
pps -g hominidae.fasta | head -n 1
```

```
>Positions (218)
```

5.36 We count the mismatches and find

	Hom	Pan	Gor	Pon
Hom	0	3	4	7
Pan	3	0	5	6
Gor	4	5	0	9
Pon	7	6	9	0

We check this by getting the raw mismatches with `dnaDist`.

⟨*cli*⟩+≡
```
pps hominidae.fasta | cutSeq -r 1-10 | getSeq -c Pos |
    dnaDist -r
```

5.37 We get a distance matrix with six distinct entries, so they are not ultrametric.

⟨*cli*⟩+≡
```
dnaDist -r hominidae.fasta
```

```
4
Homo    0   80  93  145
Pan     80  0   95  153
Gorilla 93  95  0   150
Pongo   145 153 150 0
```

5.38 We calculate the tree from the raw mismatches. Our closest relative is chimp, but note that the branch from (human, chimp) to ((human, chimp), gorilla) is very short. It almost looks as if there was a trifurcation between what later became human, chimp, and gorilla.

⟨*cli*⟩+≡
```
dnaDist -u hominidae.fasta | upgma | plotTree
```

5.39 The log of zero or less is not defined; so m has to be less than $3/4$ for equation (2.3) to work.

5.40 The probability of drawing two identical nucleotides is the probability of drawing two `A`s, or two `C`s, or two `G`s, or two `T`s, which is $4 \times (1/4)^2 = 1/4$. So the probability of drawing two different nucleotides is $1 - 1/4 = 3/4$, the point at which the number of mutations in equation (2.3) becomes infinite. We can calculate m from pairs of random sequences and find that for long sequences it approaches $3/4$.

⟨*cli*⟩+≡
```
ranseq -l 100000 -n 2 | dnaDist -u
```

```
2
Rand1 0       0.75018
Rand2 0.75018 0
```

5.41 We run `jc.awk` and plot the result with the key moved out of the way.

⟨*cli*⟩+≡
```
awk -f jc.awk | plotLine -x m -y k -g "set key center top"
```

5.42 The default distance is Jukes-Cantor. So we calculate the corrected tree and find that in this case the correction is almost imperceptible.

⟨*cli*⟩+≡
```
dnaDist hominidae.fasta | upgma | plotTree
```

5.43 We copy `primates.fasta`, count the 27 sequences and 446,204 nucleotides it contains, and calculate the average length of a primate mitochondrial genome, 16,526 bp.

⟨*cli*⟩+≡
```
cp $BEB/data/primates.fasta .
grep -c '^>' primates.fasta
cres primates.fasta
echo '446204 / 27' | bc -l
```

5.44 Tree construction from a distance matrix is very quick. So we only measure the run time of `mafft` (58 s on our computer) and ignore the rest. The default plot size of 640×384 pixels gets a bit crowded, so we set the size to 840×840 pixels.

⟨*cli*⟩+≡
```
time mafft primates.fasta > mafft.fasta
dnaDist mafft.fasta | upgma | plotTree -d 840,840
```

5.45 We run `phylonium` on the primate sequences and calculate the tree. The distance computation takes about 0.1 s on our computer.

⟨*cli*⟩+≡
```
time phylonium primates/*.fasta | upgma | plotTree
```

5.46 The trees in Fig. 5.14 do not have the same topology, for example, in Fig. 5.14A the closest relative of humans is the gorilla/chimp clade, while in Fig. 5.14B it is chimp alone.

5.3 Unrooted Trees

5.47 We change into the directory for this chapter, `5`, make the directory for this section, `3`, and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/5/
mkdir 3
cd 3/
```

5.48 We pick three taxa, A, B, and C. Their distances are $d_{AB} = 5$, $d_{AC} = 7$, and $d_{BC} = 4$. Among this triple of distances there isn't a pair of equal distances, so the three point criterion is not fulfilled.

5.49 Under Upgma we'd first form the node (B, C), but on the tree the closest neighbor of B is A, not C.

5.50 Distances d_{AB} and d_{CD} cover only the terminal branches.

5.51 The first pair that covers the full tree is $\{d_{AD}, d_{BC}\}$, the second $\{d_{AC}, d_{BD}\}$.

5.52 By plugging the distances into the equation describing the four point criterion, we get

$$5 + 5 \leq 10 + 4 = 7 + 7,$$

which is true.

5.53 Here is the distance matrix with the row sums entered:

	A	B	C	D	r_i
A	-	5	7	10	22
B		-	4	7	16
C			-	5	16
D				-	22

5.54 We fill in the S_{ij} values to get

	A	B	C	D	r_i
A	-	5	7	10	22
B	-14	-	4	7	16
C	-12	-12	-	5	16
D	-12	-12	-14	-	22

To check this, we generate the distance matrix and run `nj` with matrix printing. The first five lines of the output give the expected augmented distance matrix.

⟨*cli*⟩+≡

```
printf "4\n"            > fourTaxa.dist
printf "A  0 5 7 10\n" >> fourTaxa.dist
printf "B  5 0 4  7\n" >> fourTaxa.dist
printf "C  7 4 0  5\n" >> fourTaxa.dist
printf "D 10 7 5  0\n" >> fourTaxa.dist
nj -m fourTaxa.dist | head -n 5
```

```
4
A  0    5    7    10  22
B  -14  0    4    7   16
C  -12  -12  0    5   16
D  -12  -12  -14  0   22
```

5.55 We calculate the distance matrix with cluster (A, B).

	C	D	(A, B)
C	-	5	3
D		-	6
(A, B)			-

We check this by running `nj` with matrix printing and grabbing the second matrix.

⟨*cli*⟩+≡

```
nj -m fourTaxa.dist | tail -n +6 | head -n 4
```

```
3
C       0    5    3  8
D       -14  0    6  11
(A,B)   -14  -14  0  9
```

The output also contains the S_{ji} values and the row sums, which are not strictly necessary once the algorithm is down to the last three taxa.

5.56 The branch from A to its ancestor has length 4,

$$d_{A(AB)} = (2 \times 5 + 22 - 16)/4 = 4,$$

and the branch from B to its ancestor length 1,

$$d_{B(AB)} = (2 \times 5 + 16 - 22)/4 = 1.$$

5.57 We follow the formulae just given to get

$$\begin{aligned} d_{rC} &= (5 + 3 - 6)/2 = 1 \\ d_{rD} &= (5 + 6 - 3)/2 = 4 \\ d_{r(AB)} &= (3 + 6 - 5)/2 = 2 \end{aligned}$$

5.58 We run `nj` on our distance matrix and plot the tree.

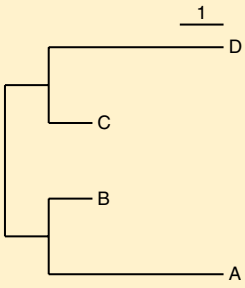
⟨*cli*⟩+≡

```
nj fourTaxa.dist | plotTree
```

5.59 The most distant pair of taxa is (A, B) and the midpoint between them happens to be the position of the root in Fig. 5.15A. We can reproduce this by running `nj` on our distance matrix, piping the resulting tree through `midRoot` and plotting it.

⟨*cli*⟩+≡

```
nj fourTaxa.dist | midRoot | plotTree
```



5.60 We copy the *Hominidae* data, calculate the distances, transform them into the neighbor-joining tree, and plot it.

⟨*cli*⟩+≡
```
cp $BEB/data/hominidae.fasta .
dnaDist hominidae.fasta | nj | plotTree
```

5.61 For additivity we require

$$d_{\text{pongo,homo}} + d_{\text{pan,gorilla}} = d_{\text{pongo,pan}} + d_{\text{homo,gorilla}}$$

But when we plug in the numbers from Fig. 5.17A, we get

$$0.18 + 0.11 \neq 0.19 + 0.11$$

The discrepancy is small, but still. So always beware of the noisiness of real data.

5.62 We run `dnaDist` with the bootstrap option and can clearly see the variations in the distances.

⟨*cli*⟩+≡
```
dnaDist -b 10 hominidae.fasta | head
```

```
4
Homo    0         0.0238115 0.0192173 0.0330852
Pan     0.0238115 0         0.0295941 0.0401165
Gorilla 0.0192173 0.0295941 0         0.0354217
Pongo   0.0330852 0.0401165 0.0354217 0
4
Homo    0         0.0238115 0.028434  0.0330852
Pan     0.0238115 0         0.0249645 0.0354217
Gorilla 0.028434  0.0249645 0         0.0330852
Pongo   0.0330852 0.0354217 0.0330852 0
```

5.63 The answer to this question depends on chance. We got the original pair (Homo, Pan) and also (Homo, Pongo). So the topology did vary.

⟨*cli*⟩+≡
```
dnaDist -b 10 hominidae.fasta | nj | plotTree
```

5.64 We run the bootstrap with 1000 iterations and count the clades to find that in only 47% of trees the (Homo, Pan) clade appeared. In other words, in approximately half of the trees the closest relative of human was *not* chimp. This means we shouldn't rely too much on that particular split.

⟨*cli*⟩+≡
```
dnaDist -b 1000 hominidae.fasta | nj | clac
```

```
#ID Count Taxa Clade
1   466   2    {Homo, Pan}
2   201   2    {Homo, Pongo}
3   136   2    {Gorilla, Pongo}
4   97    2    {Gorilla, Homo}
```

```
5   71    2    {Gorilla, Pan}
6   29    2    {Pan, Pongo}
```

5.65 We calculate the reference tree before we rerun the bootstrap analysis based on it. This time the result is a tree, which we plot.

⟨*cli*⟩+≡

```
dnaDist hominidae.fasta | nj > hominidae.nwk
dnaDist -b 1000 hominidae.fasta |
    nj | clac -r hominidae.nwk | plotTree
```

Chapter 6
Evolution within Populations

6.1 Descent from One or Two Parents

6.1 We change into the chapter directory, create the directory for this chapter, `6`, change into it, create the directory for this section, `1`, and change into that.

⟨*cli*⟩≡
```
cd $BEB/ch/
mkdir 6
cd 6/
mkdir 1
cd 1/
```

6.2 Another two rounds of doubling tells us we have 32 great-great-great-grandparents.

6.3 A naïve calculation tells us we had

$$2^{33} \approx 8.6 \times 10^9$$

ancestors 33 generations back, perhaps 20 times more than the world population at that point and a bit more than the current world population of eight billion people.

⟨*cli*⟩+≡
```
echo "2^33" | bc
echo '2^33 / 4 / 10^8' | bc -l
```

6.4 We run `drag` with the number of generations set to 7 and the population size also set to 7. We trace the ancestry of individual i_4 and render the resulting `dot` code with `neato`.

⟨*cli*⟩+≡
```
drag -g 7 -n 7 -t 4 | neato -T x11
```


B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_13

6.5 The expected number of ancestors in b_i is 2^i. This number quickly outstrips the population size, so the observed number of ancestors falls behind before stalling at 5.

Back	Observed	Expected
b_1	2	2
b_2	3	4
b_3	4	8
b_4	5	16
b_5	5	32
b_6	5	64

6.6 We used 3 as seed for the random number generator so we can trace either just individual i_4 or all individuals on a constant background.

⟨*cli*⟩+≡
```
drag -s 3 -g 7 -n 7 -t 4  | neato -T x11
drag -s 3 -g 7 -n 7 -t -1 | neato -T x11
```

6.7 Blue individuals have left no descendants in the present. Red individuals are ancestors of all extant individuals, they are *universal ancestors*. Green individuals are somewhere in between as they have left some descendants in the present, they are *partial ancestors*.

6.8 As soon as the partial ancestors have vanished, the two rather tangled graphs become identical.

6.9 We can run the simulation for, say, 50 generations, and find that the partial ancestors in green quickly go extinct never to reappear in a sea of red and a smattering of blue.

⟨*cli*⟩+≡
```
drag -g 50 | neato -T x11
```

This is because once an individual has all present day individuals among its descendants, its ancestors also have that property. Green partials are lost irreversibly. Not so the non-ancestors. They can go extinct in simulations with small populations and then reappear again. In any case, the rapid disappearance of partial ancestors means that eventually we all have the same ancestors. So next time someone tells you at a party she is a descendant of X, who lived a long, long time ago, you know there are only two possibilities: Either this is true, then it is true for everyone, or, alas, it is false.

6.10 In all of 10 trials we always got eight generations until the appearance of the first universal ancestor. So this value looks remarkably constant.

⟨*cli*⟩+≡
```
for a in $(seq 10)
do
    drag -a -g 25 -n 1000 | grep univ
done
```

6.11 This time we got 8 times 9 generations and twice 8 generations.

⟨*cli*⟩+≡
```
for a in $(seq 10)
do
    drag -a -g 25 -n 2000 | grep univ
done
```

6.12 We adjust our simulation and find that it takes on average 17.8 generations for partial ancestors to disappear from a population of 1000.

⟨*cli*⟩+≡
```
for a in $(seq 100)
do
    drag -a -g 25 -n 1000
done | awk '/part/{s += $2; c++}END{print s/c}'
```

6.13 We run `drag` with the population size from the input, filter for no partial ancestors, and print the number of generations. This is then summed.

⟨*Sum number of generations, Prog. 6.1*⟩≡
```
n=$(drag -a -n $1 -g 25 | awk '/part/{print $2}')
((sum=$sum+$n))
```

6.14 We iterate over population sizes from 10 to 1000 in approximately doubling steps and save the results in the file `nopanc.dat`. Then we plot the results with `plotLine` and move the key to avoid it intersecting the curves.

⟨*cli*⟩+≡
```
for a in 10 20 50 100 200 500 1000
do
    bash nopanc.sh $a 100
done > nopanc.dat
plotLine -L -l x -x N -y "Generations to no partials" \
           -g "set key top center" nopanc.dat
```

6.15 We trace the genetic ancestry of individual 4 for seven generations in a population of size seven.

⟨*cli*⟩+≡
```
drag -t 4 -G -g 7 -n 7 | neato -T x11
```

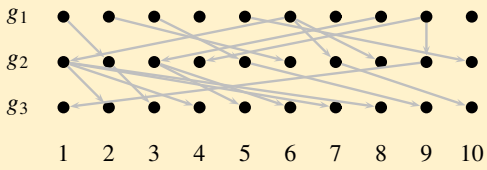
6.16 In repeated runs the most recent common ancestor of the genes is often not reached at all in seven generations, but there are always universal ancestors. So the most recent common ancestor of two genes certainly isn't always located in the first universal ancestor of all individuals.

⟨*cli*⟩+≡
```
for a in $(seq 10)
do
    drag -t 4 -G -g 7 -n 7 | neato -T x11
done
```

6.17 We generate ten random numbers between 1 and 10.

⟨*cli*⟩+≡
```
for a in $(seq 10)
do
    echo "$RANDOM % 10 + 1"
done | bc
```

Our extension of the Wright-Fisher model in Fig. 6.5 looked like this, yours is bound to look different:



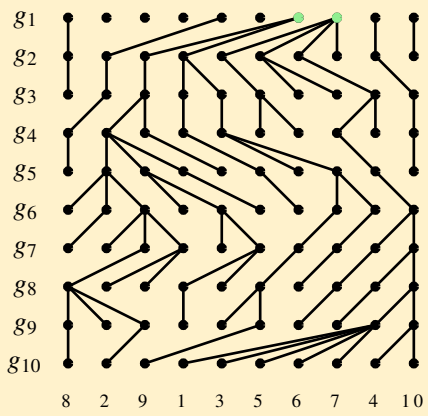
6.18 We run `drawf` in default mode and pipe its result through `neato`.

⟨*cli*⟩+≡
```
drawf | neato -T x11
```

6.19 The trick is again to use the same seed for the random number generator between runs.

⟨*cli*⟩+≡
```
drawf -s 3    | neato -T x11
drawf -s 3 -u | neato -T x11
```

6.20 No it doesn't, there are still two lines active in generation g_1, we've marked them in green.



6.21 We used $N = 30$ and found a red common ancestor in all simulations.—But the times to the most recent common ancestor vary widely, so this is still no guarantee.

⟨*cli*⟩+≡
```
for a in $(seq 10)
do
    drawf -g 30 -u -m | neato -T x11
done
```

6.22 The probability of two specific genes picking a common ancestor is $1/N \times 1/N = 1/N^2$. Since there are N opportunities for picking the same ancestor (each gene has one ancestor), the probability of any two genes picking the same ancestor is $N \times 1/N^2 = 1/N$.

6.23 If the seed or the population size haven't been set, we ask for them and exit.

⟨*Set usage, Prog. 6.2*⟩≡
```
if (!seed || !N) {
  print "Usage: awk -f panc.awk -v N=<N> -v seed=<seed>"
  exit
}
```

6.24 We print the keys in `anc`.

⟨*Print ancestors, Prog. 6.2*⟩≡
```
for (a in anc)
  printf " %d", a
printf "\n"
```

6.25 Here is the result of one run of the loop, in ten trials there is no instance of ten distinct ancestors. In fact, we ran the loop a couple of times and never saw ten distinct ancestors.

```
1 2 3 4 6 5
1 2 3 4 5 7 8 9 10 9
1 3 4 5 6 7 8 9 10 9
3 5 6 7 9 10 6
3 4 5 6 7 8 10 7
1 2 4 5 7 8 9 10 8
3 4 5 6 7 8 10 7
4 5 6 7 8 9 6
1 2 3 4 5 6 8 9 8
2 4 6 8 9 5
```

6.26 We run `pn.awk` to find that for $P_0(10) \approx 4 \times 10^{-4}$.

⟨*cli*⟩+≡
```
awk -f pn.awk -v N=10
```

6.27 We run `panc.awk` 20,000 times and calculated the frequency of finding ten ancestors. It is 5.5×10^{-4} in our case, which is reasonably close to the mathematical result of 3.6×10^{-4}.

⟨*cli*⟩+≡
```
for a in $(seq 20000)
do
    awk -f panc.awk -v N=10 -v seed=$RANDOM
done | awk '{if(NF==10)c++}END{print c/NR}'
```

6.28 We use our formula to calculate

$$P_{\mathrm{a}}(1000, 10) = (10 \times 9)/2/1000 = 0.045.$$

⟨*cli*⟩+≡
```
echo '10 * 9 / 2 / 1000' | bc -l
```

6.29 That waiting time is $1/P_{\mathrm{a}}(1000, 10) \approx 22.2$.

⟨*cli*⟩+≡
```
echo '1 / (10 * 9 / 2 / 1000)' | bc -l
```

6.30 If N, n, or the seed are not defined, we print a usage message and exit.

⟨*Set usage, Prog. 6.3*⟩≡
```
if (!N || !n || !seed) {
  printf "Usage: awk -f tmrca.awk -v N=<N> -v n=<n> "
  printf "-v seed=<seed>\n"
  exit
}
```

6.31 We count the keys in the array `lineages`.

⟨*Count lineages, Prog. 6.3*⟩≡
```
new_n = 0
for (lineage in lineages)
  new_n++
```

6.32 If the number of lineages has decreased, we print it as a function of N generations.

⟨*Report change in lineages, Prog. 6.3*⟩≡
```
if (new_n < n)
  print g / N, new_n
```

6.33 We run `tmrca` and pipe the result through `plotLine`.

⟨*cli*⟩+≡
```
awk -f tmrca.awk -v N=10000 -v n=20 -v seed=$RANDOM |
    plotLine -x "N generations" -y "Lineages"
```

6.34 For a sample of 2, the expected time to their ancestor is N generations, for a very large sample $2N$ generations. In other words, the expected time to the most recent common ancestor varies by no more than a factor of two between large and small samples.

6.35 We run the simulation 1000 times, sum the time to the most recent common ancestor, and divide by 1000. Our result, 1.78, is pleasingly close to the expectation of 1.8.

⟨*cli*⟩+≡
```
for a in $(seq 1000)
do
    awk -f tmrca.awk -v N=10000 -v n=10 -v seed=$RANDOM
done | awk '$2==1{s+=$1;c++}END{print s/c}'
```

6.2 The Coalescent

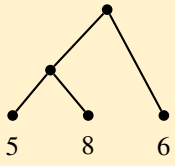
6.36 We change into the directory for this chapter, `6`, make the directory for this section, `2`, and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/6/
mkdir 2
cd 2/
```

6.37 We run `drawf` with default parameters and plot the result with `neato`.

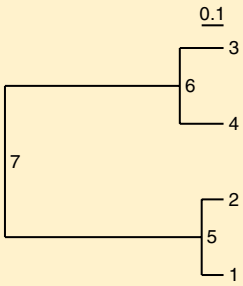
⟨*cli*⟩+≡
```
drawf | neato -T x11
```

6.38 Genes 5 and 8 coalesce first in generation g_8 and are joined by gene 6 in generation g_6.



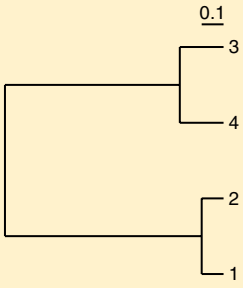
6.39 First, we draw an example coalescent with all nodes labeled. It has four leaves, so it describes the genealogy of a sample of $n = 4$ genes.

⟨*cli*⟩+≡
```
printf "((1:.1,2:.1)5:.9,(4:.2,3:.2)6:.8)7;\n" | plotTree
```



By convention, only leaves are labeled in a coalescent, so we leave out the labels of the internal nodes.

⟨*cli*⟩+≡
```
printf "((1:.1,2:.1):.9,(4:.2,3:.2):.8);\n" | plotTree
```



6.40 T_1 would begin with the most recent common ancestor, the root of the coalescent, and go on forever. It is not shown, because the coalescent stops at the most recent common ancestor.

6.41 If the user didn't set i or the seed, we print a usage message and exit.

⟨*Set usage, Prog. 6.4*⟩≡
```
if (!i || !seed) {
  print "Usage: awk -f ti.awk -v i=<i> -v seed=<seed>"
  exit
}
```

6.42 We run `ti.awk` and get $T_4 = 0.10, T_3 = 0.03, T_2 = 0.29$.

⟨*cli*⟩+≡
```
for i in 4 3 2
do
    ti=$(awk -f ti.awk -v i=${i} -v seed=$RANDOM)
    echo T_${i} $ti
done
```

So the coalescence times are 0.1 for node 5, 0.1 + 0.03 = 0.13 for node 6, and 0.13 + 0.29 = 0.42 for node 7.

Index	1	2	3	4	5	6	7
Node	1	2	3	4	5	6	7
Time	0.00	0.00	0.00	0.00	0.10	0.13	0.42

6.43 If the user omitted the sample size or the seed, we print a usage message and exit.

⟨*Set usage, Prog. 6.5*⟩≡
```
if (!n || !seed) {
  print "Usage: awk -f coat.awk -v n=<n> -v seed=<seed>"
  exit
}
```

6.44 We calculate the coalescence time like we did in `ti.awk`, but here we sum the times and print the cumulative times as a function of the number of active lineages.

⟨*Calculate coalescence time, Prog. 6.5*⟩≡
```
la = 2 / i / (i-1)
t += -la * log(1 - rand())
print i, t
```

6.45 We compute the average from our simulation, 1.599.

⟨*cli*⟩+≡
```
for a in $(seq 1000)
do
    awk -f coat.awk -v n=5 -v seed=$RANDOM
done | awk '$1==2{s += $2; c++}END{print s/c}'
```

This is close to the expectation from equation (6.2) of 1.6.

⟨*cli*⟩+≡
```
echo '2 * (1 - 1/5)' | bc -l
```

6.46 We take a deck of cards, pick our favorite suit, hearts, and take the first five cards, Ace, 2, 3, 4, and 5. We shuffle the five cards and lay them out. Voilá, we have randomized the order of 1, 2, ..., 5.

6.47 Whenever we pick an empty cell, we have to try again.

6.48 We pick the entry at position 1 and swap it with the entry at position 5. Then we pick the entry at position 3 and swap it with the entry at position 4, and so on. In the last step the entry at position 2 is swapped with itself, so nothing changes and we get our final permutation, 4, 2, 5, 3, and 1.

$r = 1, n = 5$	$r = 3, n = 4$	$r = 1, n = 3$	$r = 2, n = 2$	Result
1 2 3 4 5	1 2 3 4 5	1 2 3 4 5	1 2 3 4 5	1 2 3 4 5
1 2 3 4 **5**	5 2 **3 4** 1	**5** 2 **4** 3 1	4 **2** 5 3 1	4 2 5 3 1

6.49 We note that child1 of node 5 is 1.

Index	1	2	3	4	5	6	7
Node	1	2	3	4	5	6	7
Child1					1		
Child2							
Time	0	0	0	0			

6.50 We replace node 1 by node 4.

Index	1	2	3	4	5	6	7
Node	1	2	3	4	5	6	7
	4						
Child1					1		
Child2							
Time	0	0	0	0			

6.51 We enter node 2 as the second child of node 5 and replace node 2 by node 5.

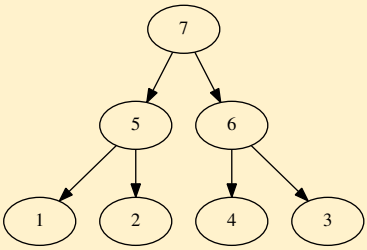
Index	1	2	3	4	5	6	7
Node	1	2	3	4	5	6	7
	4	5					
Child1					1		
Child2					2		
Time	0	0	0	0			

6.52 We continue the construction with parent 6. Its first child is the node at position 1, which is 4. Then node 3 is placed at position 1, and is thus drawn as the second child of 6. Finally, the node at position 1 is replaced by node 6:

Index	1	2	3	4	5	6	7
Node	1	2	3	4	5	6	7
	4	5					
	3						
	6						
Child1					1	4	
Child2					2	3	
Time	0	0	0	0			

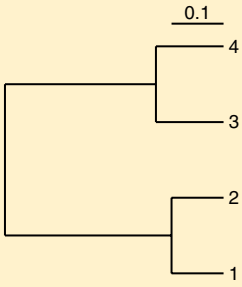
This leaves only two children for 7, 5 and 6; so the final topology is

⟨*cli*⟩+≡
```
printf "digraph g{7->5;7->6;5->1;5->2;6->4;6->3}\n" |
    dot -T x11
```



6.53 Our node times were 0.1, 0.13, and 0.42, which gives us the branch lengths.

⟨*cli*⟩+≡
```
printf "((1:0.1,2:0.1):0.32,(3:0.13,4:0.13):0.29);\n" |
    plotTree
```



6.54 If the user hasn't set a sample size or a seed, we print a usage message and exit.

⟨*Set usage, Prog. 6.6*⟩≡
```
if (!n || !seed) {
  print "Usage: awk -f pick.awk -v n=<n> -v seed=<seed>"
  exit
}
```

6.55 We first run `pick.awk`.

⟨*cli*⟩+≡
```
awk -f pick.awk -v seed=$RANDOM -v n=4
```

```
# Pa    C1      C2
5       1       2
6       1       2
7       1       1
```

Then we run `coat.awk`.

⟨*cli*⟩+≡
```
awk -f coat.awk -v n=4 -v seed=$RANDOM
```

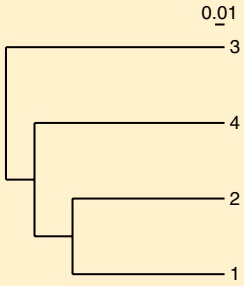
```
T_4 0.156085
T_3 0.204576
T_2 0.234759
```

So we can fill in our table:

Index	1	2	3	4	5	6	7
Node	1	2	3	4	5	6	7
	4	5					
	3	6					
	6						
Child1					1	4	3
Child2					2	5	6
Time	0.00	0.00	0.00	0.00	0.16	0.20	0.23

From this we plot the tree.

⟨*cli*⟩+≡
```
printf "(((1:0.16,2:0.16):0.04,4:0.2):0.03,3:0.23);\n" |
    plotTree
```



6.56 We calculate $\lambda = 0.5 \times 10/2 = 2.5$ and run `rpos` with this mean.

⟨*cli*⟩+≡
```
rpois -m 2.5
```

6.57 We loop over the six branch lengths to find a total of four mutations.

⟨*cli*⟩+≡
```
for a in 0.16 0.16 0.04 0.2 0.03 0.23
do
    printf "%s " $a
    l=$(echo "$a / 2 * 10" | bc -l)
    rpois -m $l
done
```

```
0.16 0
0.16 1
0.04 0
0.20 0
0.03 0
0.23 3
```

6.58 Our tree had four segregating sites, which is far fewer than the expected 18.3.

⟨*cli*⟩+≡
```
watterson -n 4 -t 10
```

6.59 We run the simulations, grab the trees, and plot them. We know from our coalescent construction that the branching order and the branch lengths are not affected by θ, only the mutations on the branches. Just to remind ourselves that that's the case, we play with $\theta = 10$ and $\theta = 99$.

⟨*cli*⟩+≡
```
ms 4 10 -t 10 -T | grep '^(' | plotTree
ms 4 10 -t 99 -T | grep '^(' | plotTree
```

6.60 In a coalescent, the scale bar measures branch lengths in units of N generations, where N is the total number of copies of that particular gene. The scale bar in Fig. 6.14 is $N/100$ generations long.

6.61 We generate 10^4 replicates with `ms`, extract the number of segregating sites, and plot their histogram.

⟨*cli*⟩+≡
```
ms 4 10000 -t 10 |
    awk '/^s/{print $2}' |
    histogram -f |
    plotLine -x S -y Frequency
```

6.62 We simulate 10^4 samples and compute the frequency of samples with at least 50 segregating sites. That frequency is the error probability when rejecting the null hypothesis that the observation arose under the Wright-Fisher model. It turns out that $P \approx 0.025$, so we reject the null hypothesis and start looking for an alternative explanation.

⟨*cli*⟩+≡
```
ms 4 100000 -t 10 |
    awk '/^s/{if($2>=50)s++;c++}END{print "P=",s/c}'
```

6.63 We get a HTML file listing directories and files.

6.64 At the time of writing we counted 10 releases.

⟨*cli*⟩+≡
```
curl -s $url | grep REL | wc -l
```

6.65 We run `curl` on the current URL and get a HTML-formatted list of files.

⟨*cli*⟩+≡
```
curl -s $url
```

6.66 From the previous `curl` listing we find that our target file contains a whopping 21 G, that is gigabytes. So it would be nice if we could query this large file without downloading it.

6.67 The `tabix man` page tells us that `-l` lists the chromosome names. These are the autosomes 1–19, the sex chromosomes X and Y, and the mitochondrial genome MT.

⟨*cli*⟩+≡
```
man tabix
tabix -l $url
```

```
1
2
...
19
X
Y
MT
```

6.68 We find that the index file contains 2.5 MB, over eight thousand times less than the parent file.

⟨*cli*⟩+≡
```
ls -l *.tbi
echo '21000/2.5' | bc -l
```

6.69 We grab the lines with contig lengths, format them as a table, and sort that table by chromosome length. We find that, indeed, chromosome 19 is the shortest mouse chromosome with 61.4 Mb.

⟨*cli*⟩+≡
```
tabix -H $url | grep length | tr ',=' '\t' | tr -d '>' |
    sort -k 5 -n
```

```
##contig <ID MT length  16299
##contig <ID 19 length  61431566
##contig <ID 18 length  90702639
##contig <ID Y  length  91744698
##contig <ID 17 length  94987271
##contig <ID 16 length  98207768
...
```

6.70 We might want to access the header line repeatedly, so we save it to file. Then we list the first nine column headers.

⟨*cli*⟩+≡
```
tabix -H $url | tail -n 1 > h.txt
awk '{for(i=1;i<=9;i++)print i,$i}' h.txt
```

```
1 #CHROM   4 REF    7 FILTER
2 POS      5 ALT    8 INFO
3 ID       6 QUAL   9 FORMAT
```

The important columns for us are 1 (chromosome), 2 (position), 4 (reference base), and 5 (alternative alleles). If you're interested in more details on the VCF format, take a look at the VCF specification at

`https://samtools.github.io/hts-specs`

6.71 There are 45 columns in total, so the sample size is $45 - 9 = 36$.

⟨*cli*⟩+≡
```
awk '{print NF}' h.txt
```

6.72 The first SNP is a `T/G` polymorphism at position 3,000,287, the second SNP is a `G/A` polymorphism at position 3,000,430.

6.73 We count the number of lines in the region and find that it contains 65 SNPs.

⟨*cli*⟩+≡
```
tabix $url 19:50,900,001-50,901,000 | wc -l
```

6.74 Mice are diploid and 36 of them were genotyped, so we might be tempted to say $n = 2 \times 36 = 72$. However, the mice in the sample are inbred strains, so we treat them as effectively haploid and set $n = 36$.

6.75 The SNP count returns 1,819,897 SNPs, 1.8 million.

6.76 According to Watterson's equation,

$$\theta = \frac{S}{\sum_{i=1}^{n-1} 1/i}.$$

S is the number of SNPs, 1,819,897, divided by the length of chromosome 19, 61,431,566. To get the harmonic number, $\sum_{i=1}^{n-1} 1/i$, we realize that according to Watterson's equation this is equal to S for $\theta = 1$. So we run `watterson` with $\theta = 1$ and $n = 36$, to get $\sum_{i=1}^{n-1} 1/i \approx 4.15$.

⟨*cli*⟩+≡
```
watterson -t 1 -n 36
```

Using these numbers, we estimate $\theta = 0.007$ per nucleotide.

⟨*cli*⟩+≡
```
echo '1819897/61431566/4.1467814' | bc -l
```

6.77 Our θ per site becomes 7 for the 1 kb region, and with $n = 36$ we calculate that approximately 29 SNPs would be expected.

⟨*cli*⟩+≡
```
watterson -t 7 -n 36
```

6.78 We simulate 10^4 samples of 36 haplotypes with $\theta = 7$ and calculate the frequency with which we find 65 or more segregating sites. That frequency is 0.005, so the observed number of SNPs is incompatible with the simple Wright-Fisher model we used in our simulation.

⟨*cli*⟩+≡
```
ms 36 10000 -t 7 |
    awk '/^s/{if($2>=65)s++;c++}END{print s/c}'
```

Chapter 7
Interrogating and Storing Data

7.1 Statistics

7.1 We change into the directory for chapters, make the directory for this particular chapter, 7, change into that, make the directory for this section, 1, and change into that, too.

⟨*cli*⟩≡
```
cd $BEB/ch/
mkdir 7
cd 7/
mkdir 1
cd 1/
```

7.2 We are told in the abstract that eight patients were included.

7.3 We browse the file with `less` and find that it contains a header section delimited by three characters, hat, exclamation mark, and hash. This is followed by the data table consisting of 18 columns. The first column is the name of the expression probe, the second the name of the genetic locus. This is followed by 16 columns of expression values for that probe.

7.4 The 18 columns are described in the section `^DATASET`. Columns 3–10 are the expression measurements from day 1, columns 11–18 the expression measurements from day 60.

7.5 We repeat the command for constructing `d1.txt`, except this time we print data columns 11–18.

⟨*cli*⟩+≡
```
awk '!/^[!^#]/' GDS4374.soft |
    tail -n +2 |
    awk '{printf "%s$%s", $1, $2;
    for(i=11;i<=18;i++)printf "\t%s",
```


B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2_14

```
    $i; printf "\n"}' > d60.txt
```

7.6 The first ten lines of `d1.txt` contain a mix of genome locations, `chr*`, and genes with names like *OR4F17*, which stands for "olfactory receptor family 4 subfamily F member 17".

⟨*cli*⟩+≡

```
head d1.txt
```

```
7896736$chr1:53049-54936 5.39918 5.61156...
7896738$chr1:63015-63887 3.00959 4.17693...
7896740$OR4F17 4.20353 4.95256...
7896742$LOC100134822 6.7929 7.1427...
7896744$OR4F29 6.25756 6.58153...
7896746$chr1:564951-565019 8.24911 8.7578...
7896748$chr1:566062-566129 9.42788 9.9854...
7896750$chr1:568069-568136 6.29291 5.08721...
7896752$M37726 8.16604 8.28596...
7896754$LOC100287934 7.45301 6.97922...
```

7.7 The last ten lines of `d1.txt` contain only control probes.

⟨*cli*⟩+≡

```
tail d1.txt
```

```
...
7896730$control 6.59805 7.52338...
```

7.8 We count the lines in `d1.txt` or `d60.txt` to find that 33,297 probes were assayed.

⟨*cli*⟩+≡

```
wc -l d1.txt
```

7.9 We extract the gene names from the probe identifiers in the first column, remove lines containing `chr` and `control`, and count the remaining unique entries to find that 20,414 distinct genes were assayed.

⟨*cli*⟩+≡

```
cut -f 1 d1.txt | tr '$' '\t' | cut -f 2 | grep -v chr |
    grep -v con | sort | uniq | wc -l
```

7.10 We repeat the previous command, except that this time we use `uniq -c` to count the distinct genes. Then we reverse-sort the gene counts and print the first 10. Their probe counts vary from 26 for *LINC00965* to 12 for *DUX4*.

⟨*cli*⟩+≡

```
cut -f 1 d1.txt | tr '$' '\t' | cut -f 2 | grep -v chr |
    grep -v con | sort | uniq -c | sort -n -r | head
```

```
26 LINC00965      15 USP17L15
17 SNORD115-24    15 NBPF25P
17 IGKC           13 PKD1P6-NPIPP1
16 SPDYE16        12 GOLGA8R
16 LOC100134822   12 DUX4
```

7.11 We use `grep` to find the rows of *ACKR2* expression values. There happen to be only two, one for each day monitored.

⟨*cli*⟩+≡
```
grep -i ACKR2 d*.txt
```

```
d1.txt:8079117_ACKR2    5.72763 6.04518 5.67683 5.69512 ...
d60.txt:8079117_ACKR2   6.59111 6.40809 6.11825 6.2248  ...
```

7.12 We run `expr.sh` on *ACKR2* and plot its results with `plotLine`, without the y axis.

⟨*cli*⟩+≡
```
bash expr.sh ACKR2 | plotLine -P -x "Expression level" -u y
```

7.13 We compute the average expression values of *ACKR2* with Awk. It is 5.8 on day 1 and 6.3 on day 60.

⟨*cli*⟩+≡
```
grep ACKR2 d*.txt |
    awk '{s=0;for(i=2;i<=NF;i++)s+=$i;print $1,s/(NF-1)}'
```

```
d1.txt:8079117_ACKR2 5.82425
d60.txt:8079117_ACKR2 6.30631
```

7.14 We calculate the fold change as $f = 2^{|a_1 - a_2|}$ and find a $f = 1.4$. So acute amebic colitis leads to a 1.4-fold decrease in the expression of *ACKR2*.

⟨*cli*⟩+≡
```
awk 'BEGIN{d=6.30631-5.82425; print 2^d}'
```

```
1.39674
```

7.15 `testMeans` works on pairs of files, so we extract the expression values for *ACKR2* into files `ackr2_1.txt` and `ackr2_60.txt` and then apply `testMeans` to them. We find $P = 10^{-4}$, which indicates a highly significant difference if our acceptance threshold is the customary $\alpha = 0.05$.

⟨*cli*⟩+≡
```
grep ACKR2 d1.txt > ackr2_1.txt
grep ACKR2 d60.txt > ackr2_60.txt
testMeans ackr2_1.txt ackr2_60.txt
```

```
# ID           m1    m2    t      P
8079117_ACKR2  5.82  6.31  -5.33  0.000106
```

We are shown the two means, the t statistic for Student's t test, and the P-value.

7.16 The program `var` tells us $a_1 = 5.8243, a_2 = 6.3063, v_1 = 0.0213$, and $v_2 = 0.0441$. So $t = -5.33$, as printed by `testMeans`.

⟨*cli*⟩+≡

```
tr '\t' '\n' < ackr2_1.txt  | tail -n +2 | var
tr '\t' '\n' < ackr2_60.txt | tail -n +2 | var
echo '(5.8243-6.3063)/sqrt(7*(0.0213+0.0441)/14)/sqrt(2/8)' |
    bc -l
```

7.17 We got the averages 6.133 and 5.997, so the difference is 0.136, which is much smaller than the original difference of roughly 0.48.

⟨*cli*⟩+≡

```
echo '5.8243-6.3063' | bc -l
```

7.18 Like gambling at the Monte Carlo Casino in Monaco, a Monte Carlo method in statistics is based on chance.

7.19 We run `testMeans` with 10^7 replicates and find $P \approx 3 \times 10^{-5}$, which is even smaller than the $P \approx 10^{-4}$ found with the t test.

⟨*cli*⟩+≡

```
testMeans -m 10000000 ackr2_*.txt
```

```
# ID           m1    m2    t      P
8079117$ACKR2  5.82  6.31  -5.33  3.18e-05
```

7.20 With 100 iterations we got $P < 10^{-2}$. One implication of the Monte Carlo method with n iterations is that we cannot exactly quantify P-values less than $1/n$.

7.21 A "significant" difference might not be all that surprising if we repeat an experiment 33 thousand times.

7.22 We remove the header line and extract four pseudo-positive results from our data, which is close to the expectation of $100 \times \alpha = 5$.

⟨*cli*⟩+≡

```
testMeans exp1.txt exp2.txt | awk '!/^#/ && $5<=0.05'
```

7.23 We run `simNorm` twice to create new versions of `exp1.txt` and `exp2.txt`. Then we apply `testMeans` and find 485 false positives, which again is close to the expectation of $10^4 \times 0.05 = 500$.

⟨*cli*⟩+≡

```
simNorm -i 10000 -m 12 > exp1.txt
simNorm -i 10000 -m 12 > exp2.txt
testMeans exp1.txt exp2.txt | awk '!/^#/ && $5<=0.05' | wc -l
```

7.24 With 99% probability, that is almost certainly, will we have at least one false positive in 100 tests without true difference when $\alpha = 0.05$.

⟨*cli*⟩+≡
```
echo '1 - (1 - 0.05)^100' | bc -l
```

7.25 In the script `fp.sh` we iterate over the α-values. For each α-value we iterate over 10^4 tests and print the results marked by α.

Prog. 7.7 (`fp.sh`)

⟨*fp.sh*⟩≡
```
for a in 0.05 0.01 0.001 0.0001
do
    awk -v a=${a} \
    'BEGIN{for(i=1;i<=10000;i++)print i, 1-(1-a)^i, a}'
done
```

Then we run `fp.sh` and generate a log/log plot with `plotLine`. We move the key out of the way of the graph.

⟨*cli*⟩+≡
```
bash fp.sh |
    plotLine -l xy -x "Number of tests" -y "f_p" \
                -g "set key center right"
```

7.26 We use the same computation of the false positive rate as before, except this time we divide α by 10^4. With this correction, we find no false positives. We know that all samples were drawn from the same population, so this is the correct result.

⟨*cli*⟩+≡
```
testMeans exp1.txt exp2.txt | awk '!/^#/ && $5<=0.05/10000'
```

7.27 We count 5,466 false negative tests, so $\beta > 0.5$. The small difference in means and the small sample size makes more than half the results false negative.

⟨*cli*⟩+≡
```
testMeans exp1.txt exp2.txt | awk '!/^#/ && $5>0.05' | wc -l
```

7.28 With $n = 16$ we find $\beta = 0.23$.

⟨*cli*⟩+≡
```
simNorm -n 16 -i 10000 -m 12 > exp1.txt
simNorm -n 16 -i 10000 -m 11 > exp2.txt
testMeans exp1.txt exp2.txt | awk '!/^#/ && $5>0.05' | wc -l
```

With $n = 32$ we find $\beta = 0.02$, which might be a level of under-reporting we are prepared to live with.

⟨*cli*⟩+≡
```
simNorm -n 32 -i 10000 -m 12 > exp1.txt
simNorm -n 32 -i 10000 -m 11 > exp2.txt
testMeans exp1.txt exp2.txt | awk '!/^#/ && $5>0.05' | wc -l
```

7.29 We run the analysis with the corrected α and find a disconcertingly large false negative rate of 0.82. This means, the Bonferroni correction leads to a large type II error and thereby obscures a lot of true differences between the two sets of experiments.

⟨*cli*⟩+≡
```
testMeans exp1.txt exp2.txt | awk '!/^#/ && $5>0.05/10000' |
    wc -l
```

7.30 We count the non-significant values among the sorted P-values to find the false negative rate, β, which is 0.02 in our case. This is close to the β we saw before, only this time with correction for multiple testing.

⟨*cli*⟩+≡
```
testMeans exp1.txt exp2.txt | tail -n +2 | sort -g -k 5 |
    awk '$5>NR*0.05/10000' | wc -l
```

7.31 As expected, the type I error is approximately α, in our case it happens to be 0.0501.

⟨*cli*⟩+≡
```
testMeans exp1.txt exp2.txt | awk '!/^#/ && $5<=0.05' | wc -l
```

7.32 In our simulation the type I error is removed entirely. In other words, the Benjamini-Hochberg correction removes the type I error as thoroughly as the Bonferroni correction, but without the concomitant increase in type II error.

⟨*cli*⟩+≡
```
testMeans exp1.txt exp2.txt | tail -n +2 | sort -g -k 5 |
    awk '$5<=NR*0.05/10000' | wc -l
```

7.33 Our type I error is still zero.

⟨*cli*⟩+≡
```
testMeans exp1.txt exp2.txt | tail -n +2 | sort -g -k 5 |
    awk '$5<=NR*0.1/10000' | wc -l
```

7.34 We run `testMeans` on the experimental data and analyze the sorted P-value keeping in mind that a total of 33,297 tests is being carried out. We filter out the control probes and end up with 25 significant loci, among them *ACKR2*.

⟨*cli*⟩+≡
```
testMeans d*.txt | tail -n +2 | sort -k 5 -g |
    awk 'BEGIN{m=33297;d=0.1}{if($5<=NR*d/m)print}' |
    tr '$' '\t' | awk '{print $2}' | sort | uniq |
    grep -v control | wc -l
```

7.2 Relational Databases

7.35 We change into the directory for this chapter, `7`, make the directory for this section, `2`, and change into that.

⟨*cli*⟩≡
```
cd $BEB/ch/7/
mkdir 2
cd 2/
```

7.36 We copy the two files from the neighboring directory.

⟨*cli*⟩+≡
```
cp ../1/d*.txt .
```

7.37 We declare the remaining seven measurements. Mind the comma after the last declaration.

⟨*Attributes, Prog. 7.2*⟩+≡
```
M2 float,
M3 float,
M4 float,
M5 float,
M6 float,
M7 float,
M8 float,
```

7.38 We can reuse the attributes and add the keys.

⟨*Create d60, Prog. 7.2*⟩≡
```
create table d60 (
          ⟨Attributes, Prog. 7.2⟩
          primary key(Probe),
          foreign key(Probe) references d1(Probe)
);
```

7.39 We get the names of the two tables we just created, `d1` and `d60`.

7.40 The table schema are the structures of the tables in a database. These structures are reproduced by showing the SQL commands we used for creating the tables.

7.41 We replace the dollar by a tab, write the result to a temporary file, and move the temporary file to its old name.

⟨*cli*⟩+≡
```
tr '$' '\t' < d1.txt > tmp
mv tmp d1.txt
tr '$' '\t' < d60.txt > tmp
mv tmp d60.txt
```

7.42 We work on the same pattern we just saw, replace tabs by pipes, save to a temporary file, and move the temporary file to the old name.

⟨*cli*⟩+≡
```
tr '\t' '|' < d1.txt > tmp
mv tmp d1.txt
tr '\t' '|' < d60.txt > tmp
mv tmp d60.txt
```

7.43 We get all attributes for the first ten lines of table `d1`.

```
7896736|chr1:53049-54936|5.39918|5.61156|5.29747|5.36575|...
7896738|chr1:63015-63887|3.00959|4.17693|3.14918|2.88891|...
7896740|OR4F17|4.20353|4.95256|3.69622|3.83216|...
7896742|LOC100134822|6.7929|7.1427|6.52695|7.12517|...
...
```

7.44 We repeat the counting for table `d60` and find that, as expected, it also has 33,297 entries.

⟨*cli*⟩+≡
```
sqlite3 colitis.db "select count(*) from d60"
```

7.45 We edit the *from* part to refer to table `d60` and repeat the query.

⟨*cli*⟩+≡
```
f60="from d60"
q="$s $f60 $w"
sqlite3 colitis.db "$q"
```

```
8079117|ACKR2|6.59111|6.40809|6.11825|6.2248|...
```

7.46 We edit the query to select from table `60`, run it again, and find as before that the average expression of *ACKR2* on day 60 is 6.31.

⟨*cli*⟩+≡
```
q="$s $f60 $w"
sqlite3 colitis.db "$q"
```

7.47 We construct the histogram for day 1 and mark it *d1*. Then we construct the histogram for day 60, mark it *d60*, and append it to the data for day 1. In the last step we plot the two histograms with `plotLine`.

⟨*cli*⟩+≡
```
q="$s $f1"
sqlite3 colitis.db "$q" | histogram |
    awk '{print $0, "d1"}' > hist.dat
q="$s $f60"
sqlite3 colitis.db "$q" | histogram |
    awk '{print $0, "d60"}' >> hist.dat
plotLine -x Expression -y Count hist.dat
```

7.48 We just change the table we select from and find that on day 60 the ribosomal precursor is also the most highly expressed gene.

⟨*cli*⟩+≡
```
q="$s $f60 $w"
sqlite3 colitis.db "$q"
```

```
RNA45S5|13.2882375
```

7.49 On day 1 the long intergenic non-protein coding RNA 910, *LINC00910*, had the lowest expression. On day 60 it was the small nuclear RNA *SNORD115-4*.

⟨*cli*⟩+≡
```
s="select sym, min((m1+m2+m3+m4+m5+m6+m7+m8)/8)"
q="$s $f1 $w"
sqlite3 colitis.db "$q"
q="$s $f60 $w"
sqlite3 colitis.db "$q"
```

7.50 Many of the fold changes are negative. There is no easy way to fix this inside SQL as it lacks if clauses. Also, remember that we've just calculated the exponents, x, of a fold change $f = 2^x$. Again, this cannot be expressed easily in SQL.

```
AQP8|-1.7650625        ABCG2|-1.31636
SLC26A2|-1.5081625     TMIGD1|-1.18503125
CLDN8|-1.4194175       CKB|-1.1636475
SLC30A10|-1.34726375   CD177|-1.15492875
TRPM6|-1.3328225       HMGCS2|-1.1079
```

7.51 We run `fc.awk` with the variable `h` set to a value that is neither zero nor the empty string.

⟨*cli*⟩+≡
```
awk -f fc.awk -v h=1
```

7.52 We print a table of symbol, fold change, and type. The simplicity of the table makes it suitable for further downstream analysis.

⟨*Print fold change, Prog. 7.4*⟩≡
```
printf "%s\t%f\t%s\n", sym, 2^ex, type
```

7.53 We filter the output from `fc.awk` for *increase* and look for the greatest to find that *REG1B* and *REG1A* have the largest fold changes.

⟨*cli*⟩+≡
```
sqlite3 colitis.db < fc.sql | tr '|' '\t' | awk -f fc.awk |
    awk '$3 ~ /^i/' | sort -k 2 -n -r | head
```

```
REG1B  10.505706 incr   DMBT1   3.686329 incr
REG1A  7.136121  incr   CHI3L1  3.499157 incr
MMP3   5.495597  incr   AQP9    3.212249 incr
S100A8 4.393582  incr   S100A9  3.009985 incr
MMP1   3.980694  incr   SLC6A14 2.818331 incr
```

7.54 *REG1B* and in particular *REG1A* are associated with islet cell regeneration. The authors of the colitis study suggested that these two genes are also involved in the regeneration of the intestinal mucosa, which is ulcerated in acute amebic colitis [36].

7.55 We repeat our `esearch` query, only this time we filter for *decrease*.

⟨*cli*⟩+≡
```
sqlite3 colitis.db < fc.sql | tr '|' '\t' | awk -f fc.awk |
    awk '$3 ~ /^d/' | sort -k 2 -n -r | head
```

```
AQP8     3.398887 decr   ABCG2  2.490370 decr
SLC26A2  2.844475 decr   TMIGD1 2.273683 decr
CLDN8    2.674775 decr   CKB    2.240231 decr
SLC30A10 2.544291 decr   CD177  2.226733 decr
TRPM6    2.518950 decr   HMGCS2 2.155317 decr
```

7.56 We repeat our query and replace *REG1B* by *AQP8*. We find that *AQP8* is expressed in the colon, where it encodes a water channel. Given that colitis causes diarrhea, which is associated with dehydration, changes in water channel expression in the colon seems like a plausible response.

⟨*cli*⟩+≡
```
q="AQP8 [GENE] AND Homo sapiens [ORGN]"
esearch -db gene -query "$q" | efetch -format docsum |
    xtract -pattern DocumentSummary -element Summary
```

7.57 We append the missing part of the query.

⟨*Construct query, Prog. 7.5*⟩+≡
```
"(d60.m1+d60.m2+d60.m3+d60.m4+" +
"d60.m5+d60.m6+d60.m7+d60.m8)/8 " +
"from d1 join d60 " +
"where d1.probe=d60.probe " +
"and d1.sym not like 'chr%' " +
"and d1.sym not like 'control'"
```

7.58 We work on the same pattern as the previous error check and abort with message if things went awry.

⟨*Run query, Prog. 7.5*⟩+≡
```
if err != nil {
        log.Fatalf("couldn't run %q", q)
}
```

7.59 We've previously had three columns, the gene symbol, the fold change, and the type of change. That's what we should aim for here, too.

7.60 The first column is the gene symbol, the second the exponent of the fold change.

7.61 If there is an error, we bail with a friendly message.

⟨*Print a row, Prog. 7.5*⟩+≡
```
if err != nil {
          log.Fatal("can't scan this row")
}
```

7.62 We just add `math` in quotes to our list of imports.

⟨*Imports, Prog. 7.5*⟩+≡
```
"math"
```

7.63 We add `fmt` in quotes to the list of imports.

⟨*Imports, Prog. 7.5*⟩+≡
```
"fmt"
```

7.64 Like in the `printf` function of Awk and the shell, these are printing verbs, `%s` is replaced by a string, `%.6f` by a floating point number with six significant digits.

7.65 We are told the module `alpha.beth` has been created and that we should run `go mod tidy` next.

```
go: creating new go.mod: module alpha.beth
go: to add module requirements and sums:
          go mod tidy
```

7.66 We are told that Go is finding the `go-sqlite3` package and that it found a particular version of it. In our case version 1.14.13.

```
go: finding module for package .../go-sqlite3
go: found .../go-sqlite3 in .../go-sqlite3 v1.14.13
```

7.67 We get a nicely formatted table of fold changes.

```
#Sym                  FC       Type
OR4F17                1.139907 incr
LOC100134822          1.106722 incr
...
```

7.68 We cut off the table header, sort by fold change and look at the genes with the smallest fold changes, which have remained remarkably constant.

⟨*cli*⟩+≡
```
./fc | tail -n +2 | sort -n -k 2 | head
```

```
SPACA6  1.000005 decr   LHX6     1.000026 incr
OR2T5   1.000009 incr   MIR105-2 1.000033 decr
SNORA48 1.000010 incr   KLK4     1.000038 decr
MAML1   1.000017 incr   APBA3    1.000043 decr
GRIK2   1.000023 decr   SNORA14A 1.000049 decr
```

7.69 At the time of writing, Ensembl consisted of a staggering 17,119 databases.

⟨*cli*⟩+≡
```
tail -n +2 dbs.txt | wc -l
```

7.70 We filter for the 431 databases on human.

⟨*cli*⟩+≡
```
grep homo_sapiens dbs.txt | wc -l
```

7.71 We count the 59 core databases for human.

⟨*cli*⟩+≡
```
grep homo_sapiens_core_ dbs.txt | wc -l
```

7.72 We sort the version numbers to find that at the time of writing the highest is 106.

⟨*cli*⟩+≡
```
grep homo_sapiens_core_ dbs.txt | tr '_' '\t' |
    sort -k 4 -n | tail -n 1
```

7.73 There are 77 tables in `homo_sapiens_core_106_38`.

⟨*cli*⟩+≡
```
mysql $c -e "show tables" | tail -n +2 | wc -l
```

7.74 We get a table with four rows, one for each attribute. An attribute is called *field* in Mysql. Each field has a type and cannot be null. The field `seq_region_id` is the primary key. The default value of each field is null, and `seq_region_id` is automatically incremented whenever a new row is added to `seq_region`.

Field	Type	Null	Key	Default	Extra
seq_region_id	int(10) unsigned	NO	PRI	NULL	auto_increment
name	varchar(255)	NO	MUL	NULL	
coord_system_id	int(10) unsigned	NO	MUL	NULL	
length	int(10) unsigned	NO		NULL	

7.75 There are 268,933, roughly 270 thousand, rows in `seq_region`, so listing all of them wouldn't be very productive.

```
+----------+
| count(*) |
+----------+
|   268933 |
+----------+
```

7.76 There are nine distinct coordinate system IDs among the 270 thousand entries.

```
+---------------------------------+
| count(distinct(coord_system_id)) |
+---------------------------------+
|                               9 |
+---------------------------------+
```

7.77 We list its six attributes `coord_system_id`, `species_id`, `name`, `version`, `rank`, and `attribute`.

⟨*cli*⟩+≡
```
mysql $c -e "describe coord_system"
```

7.78 We count nine entries in the table. We might have guessed this from the fact that there are nine distinct coordinate system IDs in `seq_region`.

⟨*cli*⟩+≡
```
mysql $c -e "select count(*) from coord_system"
```

```
+----------+
| count(*) |
+----------+
|        9 |
+----------+
```

7.79 We list all of `coord_system` to find that the chromosomes from GRCh38 have ID 4.

⟨*cli*⟩+≡
```
mysql $c -e "select * from coord_system"
```

7.80 It is true, we get the expected 24 chromosomes.

⟨*cli*⟩+≡
```
mysql $c -e "$q" | tail -n +2 | wc -l
```

7.81 We store the chromosome lengths in the file `chrLen.txt` so we don't have to repeatedly query Ensembl while experimenting with the plotting. Then we format the chromosome lengths and plot them with `plotLine`.

⟨*cli*⟩+≡
```
mysql $c -e "$q" | tail -n +2 > chrLen.dat
awk -f fcl.awk chrLen.dat |
    plotLine -x Chromosome -y Length -P
```

7.82 We edit the first line of our current query and find that the human genome comprises 3,088,269,832 bp, 3.1 Gb.

⟨*cli*⟩+≡
```
q="select sum(length)"
q="$q from seq_region"
```

```
  q="$q where coord_system_id = 4"
  q="$q and name not like 'CHR%'"
  q="$q and name not like 'MT'"
  mysql $c -e "$q"
```

```
+-------------+
| sum(length) |
+-------------+
|  3088269832 |
+-------------+
```

7.83 There are three chromosomes that have more DNA than their name would suggest. The closest call is for chromosome 10 with 134 Mb and chromosome 11 with 135 Mb. Among the short chromosomes we have two odd pairs. Chromosome 19 has 59 Mb, chromosome 20 has 64 Mb; and chromosome 21 has 47 Mb, chromosome 22 has 51 Mb.

7.84 We describe table `gene` to find 16 attributes.

⟨*cli*⟩+≡

```
  mysql $c -e "describe gene"
```

7.85 We count the 69,340 entries in `gene`, so there are roughly seventy thousand genes known in human.

⟨*cli*⟩+≡

```
  mysql $c -e "select count(*) from gene"
```

```
+----------+
| count(*) |
+----------+
|    69340 |
+----------+
```

7.86 We cut the table header from the gene lengths, divide them by 100,000, and save them in the file `len.dat`. Then we plot them with `histogram` and `plotLine`. The y axis has a log scale, which is only possible if there are no zeros among the y values. So wherever there's a zero y, we set it to 1.

⟨*cli*⟩+≡

```
  q="select seq_region_end - seq_region_start + 1"
  q="$q from gene"
  mysql $c -e "$q" | tail -n +2 |
      awk '{print $1/100000}' > geneLen.dat
  histogram geneLen.dat |
      awk '$2!=0{print}$2==0{print $1, 1}' |
      plotLine -x "Gene length (x 100 kb)" -y Count -l y
```

7.87 The combined gene length is roughly 2.2 Gb, which means that at most $2.2/3.1 \approx 71\%$ of the human genome are covered with genes. We say "at most", as the entries in `gene` might overlap.

⟨*cli*⟩+≡
```
q="select sum(seq_region_end-seq_region_start+1)"
q="$q from gene"
mysql $c -e "$q"
```

```
+----------------------------------------+
| sum(seq_region_end-seq_region_start+1) |
+----------------------------------------+
|                             2194529663 |
+----------------------------------------+
```

7.88 We describe `exon` to find 13 attributes.

⟨*cli*⟩+≡
```
mysql $c -e "describe exon"
```

7.89 We count the 43,460,686 nucleotides, 43 Mb, in constitutive exons, which cover only 1.4% of the human genome. It seems that vast regions of our genome contribute little to the functioning of our cells.

```
+----------------------------------------+
| sum(seq_region_end-seq_region_start+1) |
+----------------------------------------+
|                                43460686 |
+----------------------------------------+
```

⟨*cli*⟩+≡
```
echo '43/3088' | bc -l
```

7.90 We look up the entries for *ACKR2* and are reminded that it is an "atypical chemokine receptor 2".

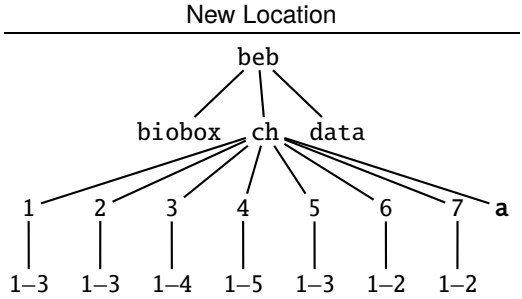
⟨*cli*⟩+≡
```
q="select *"
q="$q from xref"
q="$q where display_label like 'ACKR2'"
mysql $c -e "$q"
```

7.91 We find that ENSG00000144648 is the stable ID of *ACKR2*. It's clear why we like using mnemonic names like *ACKR2* rather than stable IDs...

```
+---------------+-----------------+
| display_label | stable_id       |
+---------------+-----------------+
| ACKR2         | ENSG00000144648 |
+---------------+-----------------+
```

Appendix A
Unix Guide

This is a summary of some of our favorite Unix commands. It is meant to be read next to a computer running the Unix operating system, so that readers can experiment. For further reading we recommend the system's online documentation. In addition, we have learned our Unix craft from books, for example [1].



We create the directory `a` for *appendix* and change into it.

⟨*cli*⟩≡
```
cd $BEB/ch/
mkdir a
cd a/
```

File Editing

Of the various text editors available on Unix systems, we recommend `emacs`, as it comes with a standard graphical user interface for casual use. At the same time, it is a powerful tool for professionals. We start it and simultaneously create the file `new.txt`. If the file `new.txt` had already existed, we'd have opened it.

⟨*cli*⟩+≡
```
emacs new.txt &
```


B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2

This opens a window with standard menus running `emacs`. We can now edit `new.txt` using keyboard input and mouse moves. In addition, `emacs` comes with a rich set of keyboard shortcuts, called "key bindings", making its use much more efficient. Table A.1 lists the key bindings we use regularly in our own work. The table also illustrates the principle of creating different versions of commands through alternate use of the control key, `C`, and the meta key, `M`. Commands referring to sentences only work if a full stop is followed by at least two blanks. Two of these key combinations are special: `C-x` and `M-x`. `C-x` is a prefix for other key combinations, and we have listed the ones we find most useful in Table A.2. `M-x` is also a prefix for further commands, but these are called by extended names like `calendar` rather than one or two characters. Again, Table A.2 lists our favorites. The full list of key bindings can be accessed by `C-h b`. The best place to start using the key bindings is the "Emacs Tutorial" opened by `C-h t`.

Table A.1 Paired `emacs` key bindings (shortcuts)

Key	Binding	Key	Binding
`C-a`	move to beginning of line	`M-a`	move to beginning of sentence
`C-b`	move backward one character	`M-b`	move backward one word
`C-d`	delete character	`M-d`	delete word to the right
—	—	`M-BACKSP`	delete word to the left
`C-e`	move to end of line	`M-e`	move to end of sentence
`C-f`	forward one character	`M-f`	forward one word
`C-g`	keyboard quit	—	—
`C-h`	help	`M-h`	mark paragraph
`C-k`	delete line	`M-k`	delete sentence
`C-l`	center buffer on current line	`M-l`	lower-case word
`C-n`	next line	—	—
`C-p`	previous line	—	—
—	—	`M-q`	layout paragraph
`C-r`	search backward	`M-r`	move to top/bottom of window
`C-s`	search forward	—	—
`C-t`	transpose characters	`M-t`	transpose words
—	—	`M-u`	upper-case word
`C-v`	scroll up	`M-v`	scroll down
`C-w`	delete selection	`M-w`	copy selection
`C-x`	command prefix (Table A.2)	`M-x`	execute extended command (Table A.2)
`C-y`	paste	—	—
`C-z`	suspend frame	`M-z`	delete to character
`C-_`	undo	—	—
`C-SPC`	set mark	—	—
`C-+`	increase font size	—	—
`C--`	decrease font size	—	—
—	—	`M-<`	move to beginning of buffer
—	—	`M->`	move to end of buffer

Table A.2 A selection of frequently used composite commands in `emacs`

Key	Binding
`C-M-\`	indent region
`C-x C-c`	quit
`C-x C-f`	find file
`C-x C-s`	save buffer
`C-x C-w`	write file
`C-x b`	switch buffer
`C-x k`	kill buffer
`C-x o`	switch to other buffer
`M-x calendar`	start calendar
`M-x count-words`	count lines, words, and characters
`M-x g`	go to line
`M-x help`	start help menu
`M-x rename-buffer`	rename the current buffer
`M-x shell`	run shell in `emacs` buffer

Working with Files

The myriad things we need to do with text files on a regular basis include viewing them, measuring their size, and finding patterns in them. Table A.3 lists some of the commands to carry out these tasks. The second command in that table, `cp`, copies one or more files.

⟨*cli*⟩+≡
```
cp $BEB/data/aa.txt $BEB/data/polarity.dat .
```

The first command in Table A.3, `cat`, writes the contents of a file to the screen.

⟨*cli*⟩+≡
```
cat polarity.dat
```

Unix commands tend to come with options, for example, `cat` can number the lines in a file.

⟨*cli*⟩+≡
```
cat -n polarity.dat
```

Commands and their options are documented in the manual pages, which are accessed using the program `man`; for example,

⟨*cli*⟩+≡
```
man ls
```

This invokes the text viewer `less`, which responds to some of the same key bindings as `emacs`, e. g. `C-v` to scroll down and `M-v` to scroll up. Inside `man`, `h` invokes help and `q` quits.

Table A.3 Commands for working with files

Command	Explanation
`cat`	print (conCATenate) to screen
`-n`	print numbered lines
`cp` *file1 file2*	copy *file1* to *file2*, overwrite old *file2* if it exists
`cp` *file1 file2 toDir*	copy *file1* and *file2* to directory *toDir*
`cut -f n`	cut the *n*th field
`diff` *fromFile toFile*	find differences between *fromFile* and *toFile*
`grep` *pattern*	print lines matching *pattern*
`-v`	print lines not matching
`head` *filename*	print first 10 lines of file
`-n n`	print first *n* lines
`less`	pager for viewing text
`ls`	list names of all files in current directory
`-l`	long listing for more information
`mv` *file1 file2*	move *file1* to *file2*, overwrite old *file2* if it exists
`rm` *filenames*	remove named files, irrevocably
`-r`	remove recursively directories and their contents
`rmdir` *directory*	remove named directory
`sort`	sort files alphabetically by line
`-n`	sort numerically
`-k n`	sort by column *n*
`-r`	reverse sort
`-R`	randomize
`tail`	print last 10 lines of file
`-n`	print last *n* lines
`+n n`	start printing file at line *n*
`uniq` *filenames*	filter out repeated lines in sorted input
`-c`	count repeated lines
`wc`	count lines, words, and characters
`-l`	count lines

Entering Commands Interactively

Any command entered at a command prompt is interpreted by a program called the "shell", which runs inside a terminal window. There are different kinds of shells and we can echo ours by printing the value of the variable `SHELL`.

⟨*cli*⟩+≡

```
echo $SHELL
```

The following description applies to the `bash`. Its most popular alternative, the `zsh` is very similar. If the `bash` isn't already running, it can be started by entering `bash`.

The command line auto completes prefixes of commands and file names in response to pressing TAB once if the prefix is unique. Otherwise, by pressing TAB a second time, the list of possible completions is presented. The most effective way of interacting with the command line is to let this auto completion feature do as much work as possible by mixing typing and tabbing. This may seem tricky at first, but after a while it becomes second nature.

Like `man`, the shell is responsive to the same basic key bindings as `emacs`, which is an added benefit from learning them.

Combining Commands: Pipes

Unix commands such as those listed in Table A.3 can be combined into programs by using the output of one command as the input of another. To do this, individual commands are combined via *pipes* denoted by a vertical line, `|`. For example, we count the files in the current directory by making the output of `ls` the input of `wc`.

⟨*cli*⟩+≡
```
ls | wc -l
```

The shell can also expand file names, which allows us to count the files that end in `.txt`.

⟨*cli*⟩+≡
```
ls *.txt | wc -l
```

Redirecting Output

By default, the result of a command is printed to the standard output stream called `stdout`. This usually corresponds to the screen. We can redirect (>) the output from the screen to a file.

⟨*cli*⟩+≡
```
ls > tmp
```

Now there's a new file, `tmp`, containing the list of files.

⟨*cli*⟩+≡
```
ls
cat tmp
```

Redirection deletes the original contents of the target file. Its variant >> appends to whatever is already in the file.

⟨*cli*⟩+≡
```
ls >> tmp
cat tmp
```

The redirection can also go the other way:

⟨*cli*⟩+≡
```
cat < tmp
```

Shell Scripts

Any command entered on the command line can be submitted to the system from a text file called a "shell script". We generate the shell script `ls.sh` for listing files.

⟨*cli*⟩+≡
```
echo ls > ls.sh
cat ls.sh
```

This can be executed by passing it to the `bash`.

⟨*cli*⟩+≡
```
bash ls.sh
```

The program `bash` reads `ls.sh` and follows the instructions it contains. Reading and writing are the two operations permitted to user `beth` on `ls.sh`. We find out about these file permissions with the long form of `ls`.

⟨*cli*⟩+≡
```
ls -l ls.sh
```

```
-rw-r--r-- 1 beth beth 3 Jun 24 10:39 ls.sh
```

The string `-rw-r--r--` displays ten mode bits describing the file permissions. The first mode bit is either `d` or dash for directory or file. The next three bits describe the permissions of the user, `beth`. The first user bit is either `r` or dash for read or not. The second user bit is either `w` or dash for write or not. The third user bit is either `x` or dash for execute or not. The next three mode bits describe the permissions of members of the group `beth`. The last three mode bits describe the permissions of all users.

We switch on the execution mode bit of `ls.sh` for all users.

⟨*cli*⟩+≡
```
chmod +x ls.sh
ls -l ls.sh
```

```
-rwxr-xr-x 1 beth beth 3 Jun 24 10:39 ls.sh
```

Now we, and all other users, can execute `ls.sh`.

⟨*cli*⟩+≡
```
./ls.sh
```

Shell scripts can contain do loops and conditional statements. Say, a set of sequence files with names of the form `fileName.txt` need changing to `fileName.fasta`. The script `files.sh` generates 10 example files.

⟨*files.sh*⟩≡
```
for f in 1 2 3 4 5 6 7 8 9 10
do
    echo '>s'${f} > s${f}.txt
    echo 'ACCGT' >> s${f}.txt
done
```

The variable `f` takes the values of the list to the right of `in`. The value of `f` is referenced by writing it in curly brackets and prefixing it with `$`. The command `echo` prints its argument. These files are in FASTA format, a header line starting with > followed by one or more lines of sequence data.

Instead of explicitly specifying a sequence of numbers, `seq` can be used:

```
for f in $(seq 10)
```

To change the file extensions from `.txt` to `.fasta`, we construct the script `rename.sh` by looping over all arguments on the command line, which are contained in the variable `$@`.

⟨*rename.sh*⟩≡
```
for f in $@
do
    mv ${f} ${f%txt}fasta
done
```

So we can capture all files with extension `.txt` and replace their extension by `.fasta`.

⟨*cli*⟩+≡
```
bash rename.sh *.txt
```

Directories

Directories may contain files and other directories. The Unix file system is thus hierarchical and can be depicted as a tree of directories. Fig. A.1 shows a portion of this tree. There are four special directory names:

1. `/` The root directory is the most basic directory situated at the root of the file system.
2. `~` The home directory is accessed after logging in. It is the only directory in which a user can create files and directories.
3. `.` The working directory is the user's current location.
4. `..` The parent directory is up one level from the working directory.

The full name of a directory such as

```
/home/beth
```

is known as its *path*. A forward slash (`/`) can therefore either refer to the root directory or function as a delimiter of directory names. Table A.4 summarizes the essential commands for working with directories.

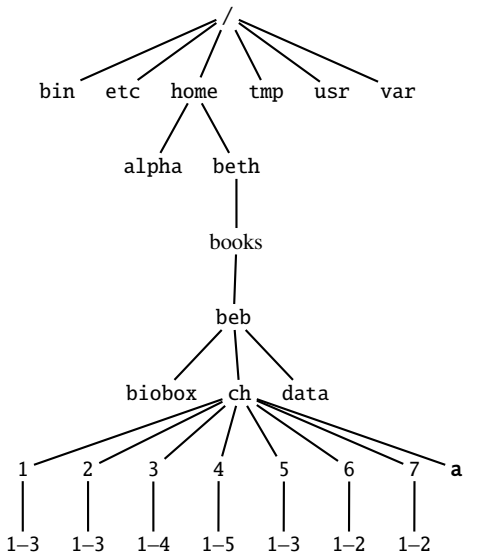


Fig. A.1 Part of a directory tree

Table A.4 Commands for working with directories

Command	Explanation
`cd ..`	move up one level in the file system
`cd`	return to home directory
`cd` *directoryname*	change to the named directory
`mkdir` *directoryname*	make the named directory
`pwd`	print working directory
`rmdir` *directoryname*	remove the named directory

Filters

Programs for filtering textual data are the bread and butter of bioinformatics. The following sections introduce four of the most popular filters: `grep`, `tr`, `sed`, and `awk`. Three of these, `grep`, `sed`, and `awk`, make use of a common notation for specifying patterns in strings. Such patterns are called "regular expressions", and we'll see a few examples.

```
a 10
b 11
c 12
d 15
e 11
```

Fig. A.2 Sample data contained in the file `data.dat`

grep

We copy the file of example data to our current directory and look at its contents, which are also shown in Fig. A.2.

⟨*cli*⟩+≡
```
cp $BEB/data/data.dat .
cat data.dat
```

We extract the lines matching `11`.

⟨*cli*⟩+≡
```
grep 11 data.dat
```

```
b 11
e 11
```

With `-v` we get the lines *not* matching.

⟨*cli*⟩+≡
```
grep -v 11 data.dat
```

```
a 10
c 12
d 15
```

We can also search for more complex patterns, for example, `[25]` matches `2` or `5`.

⟨*cli*⟩+≡
```
grep '[25]' data.dat
```

```
c 12
d 15
```

tr

The program `tr` is used to translate or delete characters. Unlike many Unix tools, `tr` does not take file names as arguments, it only reads from `stdin`. We delete the line breaks.

⟨*cli*⟩+≡
```
tr -d '\n' < data.dat
```

```
a 10b 11c 12d 15e 11
```

We translate blanks to newlines.

⟨*cli*⟩+≡
```
tr ' ' '\n' < data.dat
```

```
a
10
...
```

Or we convert the line labels in `data.dat` to upper case using character ranges.

⟨*cli*⟩+≡
```
tr a-z A-Z < data.dat
```

```
A 10
B 11
```

Number ranges are written similarly, which allows us to convert numbers into characters.

⟨*cli*⟩+≡
```
tr 0-9 a-z < data.dat
```

```
a ba
b bb
c bc
d bf
e bb
```

A biologically more relevant translation is to encode `A` or `G` as purine (`R`) and `C` or `T` as pyrimidine (`Y`).

⟨*cli*⟩+≡
```
echo ACGT | tr AGCT RRYY
```

```
RYRY
```

sed

We've already used `emacs`, which is an interactive editor. In contrast, `sed` is a non-interactive, "stream" editor. Perhaps the simplest operation with `sed` is to delete a single line, say the second line.

⟨*cli*⟩+≡
```
sed '2d' data.dat
```

```
a 10
c 12
d 15
e 11
```

Instead of deleting the second line, we print it.

⟨*cli*⟩+≡
```
sed '2p' data.dat
```

```
a 10
b 11
b 11
c 12
d 15
e 11
```

By default, `sed` applies its pattern to every line it encounters and prints all other lines unchanged. That's why we get "`b 11`" twice. But we can restrict the output to the matching line with `-n`.

⟨*cli*⟩+≡

```
sed -n '2p' data.dat
```

```
b 11
```

We can also print line ranges.

⟨*cli*⟩+≡

```
sed -n '2,4p' data.dat
```

```
b 11
c 12
d 15
```

So we can replace "`head -n`" with "`sed -n`".

⟨*cli*⟩+≡

```
head -n 2 data.dat
sed -n '1,2p' data.dat
```

```
a 10
b 11
```

Like directories in paths, regular expressions are delineated by forward slashes. We can thus print lines that match `b`.

⟨*cli*⟩+≡

```
sed '/b/p' data.dat
```

```
a 10
b 11
b 11
...
```

As before, we can restrict the output to matching lines, which gives us an alternative to `grep`.

⟨*cli*⟩+≡

```
sed -n '/b/p' data.dat
grep b data.dat
```

```
b 11
```

Similarly, we can express the complement of matching in `sed` and `grep`.

⟨*cli*⟩+≡

```
sed '/b/d' data.dat
grep -v b data.dat
```

```
a 10
c 12
d 15
e 11
```

Apart from the print and delete operations, which in practice are often dealt with using `grep`, `sed` can carry out substitutions, which `grep` can't. We substitute the first occurrence of `1` in each line by `one`.

⟨*cli*⟩+≡

```
sed 's/1/one/' data.dat
```

```
a one0
b one1
...
```

The *global* version of this command replaces all occurrences of `1` by `one`.

⟨*cli*⟩+≡

```
sed 's/1/one/g' data.dat
```

```
a one0
b oneone
...
```

The regular expression first encountered with `grep`, `[25]`, which specifies `2` *or* `5`, can also be used in a substitution.

⟨*cli*⟩+≡

```
sed 's/[25]/x/g' data.dat
```

```
a 10
b 11
c 1x
d 1x
e 11
```

Awk

Awk mixes a programming language with the line- and pattern-based paradigm of `grep` and `sed`. The following exposition is adapted from the original description of the language by its authors [2, Appendix A]. An Awk program is executed as

```
awk 'program' <file>
```

or

```
awk -f program.awk <file>
```

Each program consists of patterns and associated actions.

```
pattern {action}
```

Table A.5 Patterns in Awk

Pattern	Meaning
`BEGIN`	true before any input lines are processed
`END`	true after all input lines have been processed
expression	any expression in the Awk language
pattern, pattern	pattern range; true if in range
/regular expression/	true if matched

The pattern is evaluated for each input line and if true, the statements in the action block are executed. Table A.5 lists the most common patterns. Expressions and regular expressions can be combined with the logical operators `&&` (and), `||` (or), and `!` (not). Actions are specified through sequences of statements, some of which are listed in Table A.6.

Table A.6 Awk actions

Action	Meaning
`delete` *array element*	delete specific entry from an array
`exit`	terminate program
`if` (*expression*) *statement* [`else` *statement*]	conditional execution
input-output statement	see Table A.7
`for` (*expression*; *expression*; *expression*) *statement*	repeat a fixed number of times
`for` (*variable* `in` *variable*) *statement*	iterate over the keys of a hash
{*statement*}	statements are grouped by curly brackets
`while` (*expression*) *statement*	execute while true

Statements are separated by newlines or semicolons, lines starting with `#` are comments. The most common input and output functions of Awk are listed in Table A.7.

One of these, `printf`, produces formatted output. Formatting is done via the format conversion commands listed in Table A.8. Apart from `printf`, which prints to the screen (`stdout`), these format conversion commands are also recognized by `sprintf`, which prints to a string.

Awk has a number of built-in variables (Table A.9). Of these, `NF`, the number of fields, is particularly useful, as it allows traversal of all fields in a line, as in

Table A.7 Input and output in Awk

Action	Meaning
`close(`*fileOrPipe*`)`	close file or pipe
command `| getline`	pipe into `getline`
`print`	print current line
`printf` *fmt*, *expr-list*	print formatted output
`system(`*cmd*`)`	send *cmd* to the shell for execution

Table A.8 Formatting for `printf` and `sprintf` in Awk

Command	Meaning
`%c`	character
`%d`	decimal number
`%e`	engineering convention, `[-]d.dddddde[-+]dd`
`%f`	floating point number, `[-]d.dddddd`
`%g`	general: `%d`, `%f`, or `%e`, whichever is shorter
`%s`	string

```
for (i = 1; i <= NF; i++)
   print $i
```

Table A.9 Awk built-in variables

Variable	Meaning
`ARGC`	number of arguments on command line
`ARGV`	array of arguments on command line, `ARGV[0...ARGC-1]`
`FILENAME`	name of current input file
`FS`	field separator
`NF`	number of fields in current line
`NR`	number of records (= lines)

Awk is designed for manipulating strings and Table A.10 lists its built-in string functions, including `sprintf`.

Table A.10 Awk string manipulation functions; *s* & *t*: strings, *r*: regex, *i* & *n*: integers

Function	Meaning
`gsub(`r, s, t`)`	globally substitute r by s in t
`index(`s, t`)`	return first starting position of t in s or 0 for no match
`length(`s`)`	return length of s
`match(`s, r`)`	return first starting position of r in s or 0 for no match
`split(`s, a, f`)`	split s on f into fields saved in a, return number of fields
`sprintf(`*fmt*, *expr-list*`)`	return *expr-list* as a string formatted according to *fmt*
`sub(`r, s, t`)`	like `gsub`, except that only first occurrence of r in t is replaced by s
`substr(`s, i, n`)`	return n-char substring starting at i

Awk also provides a selection of arithmetic functions such as `log` and `exp`, which are listed in Table A.11.

Table A.11 The arithmetic functions of Awk

Function	Meaning
`cos(`x`)`	$\cos(x)$
`exp(`x`)`	e^x
`int(`x`)`	truncate x to integer
`log(`x`)`	natural logarithm
`rand()`	random number, r, $0 \leq r < 1$
`sin(`x`)`	$\sin(x)$
`sqrt(`x`)`	$\sqrt{x}$
`srand(`x`)`	seed the random number generator with x; if x is omitted, x = current second

Expressions are combined using the operators in Table A.12. Perhaps the most obscure—but in practice very useful—operator is string concatenation, which is implied when writing strings next to each other. We print `bioinformatics` by concatenating `bio` and `informatics`.

⟨*cli*⟩+≡
```
awk 'BEGIN{a="bio"; b="informatics"; c=a b; print c}'
```

Table A.12 Awk operators

Operators	Meaning
`= += -= *= /= %= ^=`	assignment
`?:`	conditional expression
`||`	OR
`&&`	AND
`in`	key in hash
`~ !~`	regular expression match and its negation
`< <= > >= != ==`	comparisons
`s1 s2`	concatenate strings `s1` and `s2`
`+ -`	addition, subtraction
`* / %`	multiplication, division, modulo
`!`	NOT
`^`	to the power of
`++ --`	increment, decrement, can be used in prefix and postfix notation
`$`	field (column)

We print the second column of our example data, which we format as decimal numbers, `%d`.

⟨*cli*⟩+≡
```
awk '{printf "%d\n", $2}' data.dat
```

```
10
11
...
```

Awk is designed for computation on data, which is difficult or impossible in `grep` or `sed`. We sum the entries in the second column.

⟨*cli*⟩+≡
```
awk '{s += $2}END{printf "sum: %d\n", s}' data.dat
```

```
sum: 59
```

We calculate the average of these five numbers.

⟨*cli*⟩+≡
```
awk '{c++; s += $2}END{printf "avg: %g\n", s/c}' data.dat
```

```
avg: 11.8
```

Awk arrays behave like hash tables with strings or numbers as keys. We count the occurrences of the numbers in the second column of the input data file. One of these occurs twice, `11`.

⟨*cli*⟩+≡
```
awk '{s[$2]++}END{for(a in s)print a, s[a]}' data.dat
```

```
10 1
11 2
12 1
15 1
```

As a final example, we print all lines that match either `2` or `5`. To express "print the whole line", we can write

```
print $0
```

or

```
print
```

But we can also just rely on the default action, which is printing the line. So we have a choice between three increasingly terse versions of the same command.

⟨*cli*⟩+≡
```
awk '/[25]/{print $0}' data.dat
awk '/[25]/{print}' data.dat
awk '/[25]/' data.dat
```

```
c 12
d 15
```

Like the rest of Unix, `awk` is described in its `man` pages. In addition, we recommend the book on Awk by the authors of Awk, which to us is a model of clarity and usefulness [2].

Regular Expressions

Regular expressions denote sets of strings. In our previous example, `[25]`, the set contains `2` and `5`. Another example for a set of strings is the dot (.). It contains all strings of length one. As a rule, everything is text in Unix, and as a consequence, regular expressions are used in many Unix programs, not just the three examples we saw above, `grep`, `sed`, and Awk, but also in `emacs` and the shell. Knowing about regular expressions is thus very useful when using Unix.

Character Classes

Character classes are written in square brackets. For example, `[ab]` matches either `a` or `b`. The complement of a character class is `[^ab]`, which matches anything but `a` or `b`. Some character classes are used so frequently, there is a standardized notation for them. Of the five character classes listed in Table A.13, we try `digit`.

⟨*cli*⟩+≡

```
sed 's/[[:digit:]]/x/g' data.dat
```

```
a xx
...
```

Table A.13 Five popular character classes

Class	Meaning	Code
`[:alpha:]`	Letters	`[A-Za-z]`
`[:cntrl:]`	Control characters	—
`[:digit:]`	Digits	`[0-9]`
`[:space:]`	Whitespace characters	`[ \t\n\r\f\v]`
`[:print:]`	Printing characters	`[^[:cntrl:]]`

Table A.14 Anchors in regular expressions

Expression	Explanation
\ *b*	Word boundary
`^`	Beginning of line (except when used inside a character class)
`$`	End of line

Anchors

Anchors allow a position within a string to be referenced, and Table A.14 lists the three most important ones, word boundary (\b), beginning of line (^), and end of line ($). We substitute the 1 at the end of a line by x.

⟨*cli*⟩+≡
```
sed 's/1$/x/' data.dat
```

```
a 10
b 1x
c 12
d 15
e 1x
```

Table A.15 Quantifiers in regular expressions

Number of matches (x)	Expression
$m \leq x$	{m,}
$m \leq x \leq n$	{m,n}
$x \geq 0$	*
$x \geq 1$	+

Quantifiers

There are four different types of quantifiers in regular expressions (Table A.15). They are *greedy*, which means they maximize the number of matches. For example, the expression .* would match the entire line of text rather than stopping at the beginning of the line upon encountering the first match. Quantifiers are part of the *extended* repertoire of regular expressions, which are the default in Awk, but have to be activated in grep and sed with -E. So we print lines containing two consecutive 1s with grep, sed, and awk.

⟨*cli*⟩+≡
```
grep -E '1{2}' data.dat
sed -E -n '/1{2}/p' data.dat
awk '/1{2}/' data.dat
```

```
b 11
e 11
```

Backreferences

Assigning values to variables is one of the most important operations in traditional programming languages. In regular expressions, backreferences provide an analogous mechanism. For example, to substitute the first pair of identical digits by just a single occurrence of that digit, we use extended `sed` with two back references `\1`, which refer to the first token matched. Instead of the second `\1`, we could just as well have written `\2`, which refers to the second token matched, as they are identical.

⟨*cli*⟩+≡

```
sed -E 's/([0-9])(\1)/\1/' data.dat
```

```
a 10
b 1
c 12
d 15
e 1
```

Appendix B
Programs

Many of the programs used in this book are part of every Unix installation. Those that aren't, fall in three categories: programs we write ourselves (Table B.1), programs from our collection of bioinformatics tools (biobox, Table B.2), and third-party tools (Tables B.3 and B.4).

B.1 Own

Table B.1 shows the 65 programs we write ourselves and their numbers.

Table B.1 Own programs

`al2fasta.awk`, 4.11	`fc.sql`, 7.3	`nopanc.sh`, 6.1	`runs.awk`, 3.11
`blastn.sh`, 4.2	`fp.sh`, 7.7	`numMum.sh`, 4.16	`scores.sh`, 2.4
`coat.awk`, 6.5	`g1.dot`, 4.4	`numTrees.awk`, 5.1	`sens.sh`, 4.17
`colitis.sql`, 7.2	`g2.dot`, 4.19	`panc.awk`, 6.2	`shustring.sh`, 3.4
`cov.awk`, 4.6	`gapScore.awk`, 2.8	`pick.awk`, 6.6	`simCov.sh`, 4.7
`cres.awk`, 1.3	`ict.sh`, 1.4	`plotLcp.sh`, 3.9	`simShot.awk`, 4.10
`ct.sh`, 1.1	`isa.awk`, 3.6	`pm.sh`, 2.9	`simStats.sh`, 4.3
`dot.awk`, 2.1	`jc.awk`, 5.2	`ral.sh`, 2.5	`ss.sh`, 1.5
`entr.sh`, 4.20	`kerror.sh`, 4.21	`ranAdh.sh`, 2.10	`suf.awk`, 3.5
`esa.awk`, 3.7	`lines.awk`, 2.3	`readFasta.awk`, 3.1	`tab2fastq.awk`, 4.22
`eval.awk`, 4.18	`lrep.sh`, 3.8	`reduce.sh`, 4.23	`ti.awk`, 6.4
`exex.sh`, 2.2	`megablast.sh`, 4.1	`repeater.sh`, 3.3	`timeBlast.sh`, 4.8
`expr.sh`, 7.1	`mism.sh`, 2.6	`rotate.awk`, 3.10	`timeBwa.sh`, 4.9
`fc.awk`, 7.4	`mutate.sh`, 4.15	`rtAl.sh`, 4.12	`tmrca.awk`, 6.3
`fc.go`, 7.5	`mut.awk`, 2.7	`rtMummer2.sh`, 4.14	`varLen.awk`, 1.2
`fcl.awk`, 7.6	`naive.awk`, 3.2	`rtMummer.sh`, 4.13	`yeast.dot`, 4.5
`fc.sh`, 3.12			

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2

B.2 Biobox

In addition to the programs we write ourselves, we use 54 programs contained in a set of computational tools for biology, our "biobox" (Table B.2), which is explained in Section 1.2 and hosted at

sn.pub/dy6S42

Table B.2 Programs from the biobox used in this book

al	drawSt	mtf	plotTree	sequencer
blast2dot	fasta2tab	mum2plot	pps	shustring
bwt	geco	mutator	randomizeSeq	simNorm
clac	genTree	naiveMatcher	ranDot	sops
cres	getSeq	nj	ranseq	testMeans
cutSeq	huff	num2char	rep2plot	translate
dnaDist	hut	numAl	repeater	travTree
drag	histogram	olga	revComp	upgma
drawf	kerror	pam	rpois	var
drawGenes	keyMat	plotLine	sass	watterson
drawKt	midRoot	plotSeg	sblast	

B.3 Third-Party

In addition to our own programs and the biobox, we use a number of third-party tools. Almost all of these third-party tools can be installed using a package manager, either `apt` on Ubuntu and similar systems (Table B.3) or Homebrew on macOS (Table B.4). The one exception to this rule is `ms`[1]. This needs to be downloaded and compiled according to the instructions on its web site. Once you've compiled it, copy `ms` into `$BEB/biobox/bin` to make it available on your system.

[1] http://home.uchicago.edu/rhudson1/source/mksamples.html

Table B.3 Third-party programs used in this book and where to get them from when using the `apt` package manager on systems like Ubuntu

Program	Package
`blastn`, etc.	`ncbi-blast+`
`bwa` & samtools	`samtools`
`curl`	`curl`
`dot`, etc.	`graphviz`
`emacs`	`emacs`
`esearch`, etc.	`ncbi-entrez-direct`
`evince`	`evince`
`mafft`	`mafft`
`ms`	none (but see above)
`mummer`, etc.	`mummer`
`mysql`	`mysql-client`
`phylonium`	`phylonium`
`sqlite3`	`sqlite3`
`velveth`, etc.	`velvet`

Table B.4 Third-party programs used in this book and where to get them from when using the Homebrew package manager on systems like macOS

Program	Package
`blastn`, etc.	`blast`
`bwa`	`bwa`
`dot`, etc.	`graphviz`
`emacs`	`emacs`
`esearch`, etc.	`brewsci/science/edirect`
`evince`	`evince`
`gdate`	`coreutils`
`mafft`	`mafft`
`ms`	none (but see above)
`mummer`, etc.	`mummer`
`mysql`	`mysql`
`phylonium`	`phylonium`
`samtools`	`samtools`
`sqlite3`	`sqlite3`
`velveth`, etc.	`velvet`
`wget`	`wget`

References

1. Abrahams, P.W., Larson, B.R.: UNIX for the Impatient, 2nd Edition. Addison-Wesley (1996)
2. Aho, A.V., Kernighan, B.W., Weinberger, P.J.: The AWK Programming Language. Addison-Wesley, Reading, MA (1988)
3. Aken, B.L., Achuthan, P., Akanni, W., Amode, M.R., Bernsdorff, F., Bhai, J., Billis, K., Carvalho-Silva, D., Cummins, C., Clapham, P., Gil, L., Girón, C.G., Gordon, L., Hourlier, T., Hunt, S.E., Janacek, S.H., Juettemann, T., Keenan, S., Laird, M.R., Lavidas, I., Maurel, T., McLaren, W., Moore, B., Murphy, D.N., Nag, R., Newman, V., Nuhn, M., Ong, C.K., Parker, A., Patricio, M., Riat, H.S., Sheppard, D., Sparrow, H., Taylor, K., Thormann, A., Vullo, A., Walts, B., Wilder, S.P., Zadissa, A., Kostadima, M., Martin, F.J., Muffato, M., Perry, E., Ruffier, M., Staines, D.M., Trevanion, S.J., Cunningham, F., Yates, A., Zerbino, D.R., Flicek, P.: Ensembl 2017. Nucleic Acids Research **45**(D1), D635 (2017)
4. Benjamini, Y., Hochberg, Y.: Controlling the false discovery rate: a practical and powerful approach to multiple testing. Journal of the Royal Statistical Society, Series B **57**, 289–300 (1995)
5. Börsch-Haubold, A.G., Montero, I., Konrad, K., Haubold, B.: Genome-wide quantitative analysis of histone H3 lysine 4 trimethylation in wild house mouse liver: Environmental change causes epigenetic plasticity. PlosOne **9**, e97568 (2014)
6. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California (1994)
7. Codd, E.F.: A relational model for large shared data banks. Communications of the ACM **13**, 377–387 (1970)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge, MA (2001)
9. Dayhoff, M.O., Schwartz, R.M., Orcutt, B.C.: A model of evolutionary change in proteins. In: M.O. Dayhoff (ed.) Atlas of Protein Sequence and Structure, vol. 5/suppl.3, pp. 345–352. National Biomedical Research Foundation, Washington DC (1978)
10. Dhandayuthapani, S., Rasmussen, W.G., Baseman, J.B.: Disruption of gene mg218 of mycoplasma genitalium through homologous recombination leads to an adherence-deficient phenotype. Proceedings of the National Academy of Sciences **96**(9), 5227–5232 (1999)
11. Donovan, A.A.A., Kernighan, B.W.: The Go Programming Language. Addison-Wesley, New York (2016)
12. Efron, B.: Bootstrap methods: another look at the Jackknife. The Annals of Statistics **7**, 1–26 (1979)
13. Fairley, S., Lowy-Gallego, E., Perry, E., Flicek, P.: The International Genome Sample Resource (IGSR) collection of open human genomic variation resources. Nucleic Acids Research **48**, D941–D947 (2020)
14. Felsenstein, J.: Inferring Phylogenies. Sinauer, Sunderland (2004)

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2

15. Fraser, C.M., Gocayne, J.D., White, O., Adams, M.D., Clayton, R.A., Fleischmann, R.D., Bult, C.J., Kerlavage, A.R., Sutton, G.G., Kelley, J.M., Fritchman, J.L., Weidman, J.F., Small, K.V., Sandusky, M., Fuhrmann, J.L., Nguyen, D.T., Utterback, T., Saudek, D.M., Phillips, C.A., Merrick, J.M., Tomb, J., Dougherty, B.A., Bott, K.F., Hu, P.C., Lucier, T.S., Peterson, S.N., Smith, H.O., Venter, J.C.: The minimal gene complement of Mycoplasma genitalium. Science **270**, 397–403 (1995)
16. Gao, J., Watabe, H., Taotsuka, T., Pang, J., Y., Z.: Molecular phylogeny of the *Drosophila obscura* species group, with emphasis on the old world species. BMC Molecular Evolutionary Biology **7**, 87 (2007)
17. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book, 2nd edn. Pearson (2008)
18. Grantham, R.: Amino acid difference formula to help explain protein evolution. Science **185**, 862–864 (1974)
19. Haig, D., Hurst, L.D.: A quantitative measure of error minimization in the genetic code. Journal of Molecular Evolution **33**, 412–417 (1991)
20. Hudson, R.R.: Generating samples under a Wright-Fisher neutral model of genetic variation. Bioinformatics **18**, 337–338 (2002)
21. Huffman, D.: A method for the construction of minimum-redundancy codes. Proceedings of the I.R.E **40**, 1098–1101 (1952)
22. Jukes, T.H., Cantor, C.R.: Evolution of protein molecules. In: H.N. Munro (ed.) Mammalian Protein Metabolism, vol. 3, pp. 21–132. Academic Press, New York (1969)
23. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. LNCS **2089**, 181–192 (2001)
24. Katoh, K., Misawa, K., Kuma, K., Miyata, T.: MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. Nucleic Acids Research **30**, 3059–3066 (2002)
25. Katoh, K., Standley, D.M.: MAFFT multiple sequence alignment software version 7: improvements in performance and usability. Molecular Biology and Evolution **30**, 772–780 (2013)
26. Klötzl, F., Haubold, B.: `Phylonium`: fast estimation of evolutionary distances from large samples of similar genomes. Bioinformatics **36**, 2040–46 (2020)
27. Knuth, D.E.: Literate Programming. The Center for the Study of Language and Information Publications (1992)
28. Knuth, D.E.: The TEXbook. Addison-Wesley, Reading, Massachusetts (1994)
29. Knuth, D.E.: The Art of Computer Programming: Fundamental Algorithms, vol. 1. Addision Wesley, Boston (1997)
30. Korf, I., Yandell, M., Bedell, J.: BLAST. Basic Local Alignment Search Tool. O'Reilly (2003)
31. Lamport, L.: A Document Preparation System: LATEX, 2nd edn. Addison-Wesley, Boston (1994)
32. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology **10**, R25 (2009)
33. Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., 1000 Genome Project Data Processing Subgroup: The sequence alignment/map format and SAMtools. Bioinformatics **25**, 2078–2079 (2009)
34. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing **22**, 935–948 (1993)
35. Ohlebusch, E.: Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction. Enno Ohlebusch, Ulm (2013)
36. Peterson, K.M., Guo, X., Elkahloun, A.G., Mondal, D., Bardhan, P.K., Sugawara, A., Duggal, P., Haque, R., Petri Jr., W.A.: The expression of REG 1A and REG 1B is increased during acute amebic colitis. Parasitology International **60**, 296–300 (2011)
37. Rohde, D.L.T., Olson, S., Chang, J.T.: Modelling the recent common ancestry of all living humans. Nature **431**, 562–566 (2004)
38. Sanger, F., Coulson, A.R., Hong, G.F., Hill, D.F., Petersen, G.B.: Nucleotide sequence of bacteriophage λ DNA. Journal of Molecular Biology **162**, 729–773 (1982)

39. Student: The probable error of a mean. Biometrica **6**, 1–25 (1908)
40. Wakeley, J.: Coalescent Theory: An Introduction. Roberts & Company, Colorado (2009)
41. Waterman, M.S.: Introduction to Computational Biology; Maps, Sequences and Genomes. Chapman & Hall/CRC, London (1995)
42. Watterson, G.A.: On the number of segregating sites in genetical models without recombination. Theoretical Population Biology **7**, 256–276 (1975)
43. Zerbino, D., Birney, E.: Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res **18**(5), 821–829 (2008)

Index

ABC transporter, 109
ACKR2, 185, 186, 190, 193, 204
additive distance, 155–157
alcohol dehydrogenase (*Adh*), 42, 43, 45, 51, 80
  evolution, 54, 56
  in genome, 100
  lcp values, 80, 81
  longest repeat, 79, 80
  MUMs in, 95
  transposon, 43, 47
  tree, 55, 56
alignment, 31
  and evolution, 29–33
  application, 51–56
  construction, 39–50
  detecting homology, 51–54
  gap extension, 30, 48
  gap opening, 30, 48, 55
  gap score, 29
  global, 48–50, 93–97
  glocal (global/local), 93, 110–121
  k-error, 112–114
  local, 48–50, 98–110
  number of alignments, 39–41
  optimal, 29–56, 94, 133
  overlap, 30, 32
  random, 53
  score, 29–39, 49, 53
  sensitivity compared to match plot, 54
  trace-back, 48–50, 133
alignment matrix, 48–50, 112, 133
allele, 30
alphabet, 86
amebic colitis, 184–186, 190, 196
amino acids
  background frequencies, 37, 38
  charge, 37
  hydropathy, 37
  match probability, 38
  polarity, 33, 34
  side chains, 38
  volume, 37
ancestors
  non-, 164
  number of, 162, 163
  partial, 162, 164
  universal, 162, 164, 165
anchor alignment, 134–136
`apt`, 18, 398
AQP8, 197
ASCII code, 13
assembly, 122–132
assembly quality, 132
`awk`, 17, 22–27, 384, 388–392
  action, 21, 24, 389
  array, 22, 23, 392
  `BEGIN` block, 21, 23, 115, 389
  `break`, 60
  character array, 78
  comment, 23, 389
  concatenation, 25, 391
  `continue`, 78
  default action, 24, 392
  `END` block, 21, 23, 389
  `for` loop, 23, 26, 60, 389
  initialize variables, 115
  `length`, 25, 390
  number of fields (`NF`), 25, 389, 390
  number of records (`NR`), 24, 390
  pattern, 21, 24, 389
  `print`, 22, 27, 390
  `printf`, 27, 389, 390
  `split`, 25, 26, 43, 390

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*,
https://doi.org/10.1007/978-3-031-20414-2

split FASTA file, 154
usage message, 128
while, 79, 389

BAM file, 114, 119
bash, 4, 5, 9, 69
ampersand (&), 18, 19
append (>>), 4, 5, 14, 381
arguments ($@), 80
auto completion, 4, 6, 7
file permissions, 382
for loop, 18, 20, 38
history, 8
if block, 52
PATH, 11, 14
pipe, 4, 8, 11, 381
quotes, 14
redirect (>), 12, 14, 381
scripts, 18–21, 382–383
SHELL, 5
split words, 201
time, 60, 69
undo, 7
bc, 4, 10
Benjamini, Y., 189
Benjamini-Hochberg correction, 187, 189, 190
β-globin, 32, 43
binary file, 15
binary tree, 90
biobox, 14
al, 30–32, 37, 49, 51, 93, 94, 96, 99, 100, 104, 112, 114
blast2dot, 109, 110
bwa, 88
bwt, 82, 84, 86, 91
clac, 155, 159
cres, 15, 91, 151
cutSeq, 11, 16, 32, 45, 52
dnaDist, 94, 97, 152, 159
drag, 162–164
drawf, 165, 167, 172
drawGenes, 11, 17, 52
drawKt, 58, 63, 64
drawSt, 65, 67, 75, 76
fasta2tab, 118
geco, 33, 34, 37
genTree, 142, 146
getSeq, 37, 134
histogram, 18, 20, 32
huff, 88, 91
hut, 88, 91, 92
kerror, 112–114
keyMat, 30, 32, 64, 100, 112
midRoot, 54, 144
mtf, 86, 87, 91, 92
mum2plot, 94
mutator, 54, 55, 96, 102
naiveMatcher, 61, 64
nj, 54, 157–159
num2char, 92
numAl, 39, 41
olga, 122, 123
pam, 37–39
plotLine, 11, 17, 20, 21, 32, 51
plotSeg, 42, 43, 49, 94
plotTree, 54, 63, 138, 142, 146
pps, 148, 151
randomizeSeq, 67, 79, 85
ranDot, 106, 108
ranseq, 152
rep2plot, 42, 43
repeater, 42, 43, 45, 66–69, 79, 94
revComp, 58, 60
sass, 125, 126
sblast, 64, 98–101, 104, 106, 112–114
sequencer, 114, 124, 128, 130
shustring, 65, 69, 70
simNorm, 187
sops, 134, 135
testMeans, 184, 186–188
translate, 11, 16, 32
travTree, 142, 145
upgma, 136, 138, 150
var, 186
watterson, 172, 177, 179, 181
bit, 4, 10, 88, 89
Blast, 64, 93, 98–110, 114
algorithm, 99
all-against-all, 109
anchor mode, 136
bit score, 104
database, 101, 102
E-value, 103–105
extension step, 99, 100
graph of hits, 106
high-scoring pair, 99, 103
megablast mode, 101
mutation rate, 103
NCBI, 101–103
P-value, 104, 106
reciprocal hit, 106
sensitivity, 103
sequence divergence, 102
simple, 99–100
statistics, 98, 104–110
tabular output, 101
unidirectional hit, 106
word list, 99, 100, 112

blastdbcmd, 102
blastn, 101–106, 114, 120
blastp, 106, 109
Bonferroni correction, 187, 189
Bonferroni, C. E., 189
bootstrap, 158, 159
Bourne-again shell, 5
bowtie, 88, 92
bunzip2, 88, 92
Burrows-Wheeler transform, 82, 84, 87
  decode, 86
  encode, 84
  suffix array, 84
bwa, 92, 117, 118, 120, 121
byte, 12, 14, 15, 88
bzip2, 88, 92

cat, 11, 15, 16, 21
cd, 4, 9
chromosome, 202
circo, 106, 108, 109
coalescence
  probability of, 174
  time to, 174
  times, 173, 175
coalescent, 161, 172–181
  construction, 175, 176
  intervals, 173
  mutations, 177
  pick children, 177
  population mutation rate, 179, 181
  population size, 174
  sample size, 173, 174, 181
  scale bar, 179
  time to the most recent common ancestor, 175
Codd, E. F., 190
code chunk, 4
coding sequence (CDS), 30, 32, 45
codons, 9
coin tossing, 18, 20
command history, 8
command line, 4
  compared to graphical user interface, 3
  compared to language, 3
  editing, 7
  navigating, 7
common ancestor, 141
compiled *vs.* scripting language, 61
compression, 15, 81–92, 101, 119
contig, 123, 125, 127, 128
control *vs.* experiment, 188
control key, 7
coverage, 114, 116, 127–129
cp, 5
cres, 11
curl, 172, 180
cursor, 4, 7
cut, 11, 16

D-loop, 151
Darwin, C., 141
data stream, 94
date, 65, 69, 94
Dayhoff, M. O., 37
descent
  one parent, 165–171
  two parents, 162–165
diff, 21, 24, 45
directory, 11, 383–384
  ./bin, 14
  /bin, 11
  BEB, 4, 5, 14
  beb, 4
  ch, 6
  data, 16, 21
  delete, 7
  home, 4, 11, 383
  make a, 4
  path, 4, 383
  root, 11, 383
  tree, 11, 384
distance matrix, 54, 137, 148, 149, 157, 159
divergence time, 37
dot, 39, 41, 123
dot notation, 106, 108
dot plot, 42–48, 66, 96
Drosophila guanche, 42, 43, 52
Drosophila melanogaster, 42, 43
dvips, 58
dynamic programming matrix, *see* alignment matrix

echo, 4, 10, 12, 14, 27
edirect software, 184
editor, *see* text editor
efetch, 184, 197
emacs, 18–20
empty lines, 26
empty string, 26
enhanced suffix array, 71, 73, 75–77, 79
Ensembl database collection, 190, 191, 200–204
  coordinate system, 201, 202
  core, 201
  human, 201
entity relation (ER) model, 191
  attribute, 191, 192

foreign key, 191, 192
primary key, 191, 192
Entrez database collection, 184, 197, 200
Escherichia coli, 64
genome alignment, 97
`esearch`, 184, 197
`evince`, 58
exact matching, 57–93, 96, 112
failure link, 61
graph, 61
naïve, 58–60
pattern, 57
preprocessing, 57
run time, 60
set, 58, 63
exon, 45, 47, 204
exponential distribution, 172, 174
`export`, 4
expression data, 184, 185, 190, 193, 194
expression probe, 185

false discovery rate, 189
fast alignment, 93–140
FASTA file, 15
dealing with data, 59
dealing with header, 59, 60
dealing with last sequence, 59
FASTA format, 11, 12
FASTQ file, 121
FASTQ format, 117, 118
Fisher, R. A., 166
fixed length code, 88
`fold`, 11, 17
fold change, 186, 195–197
four point criterion, 155, 157

gapped alignment, 113
`gdate`, 69, 94
Genbank file, 45
gene duplication, 42, 53
genetic code
mutation, 33–37
number of possible codes, 35
random, 33, 34, 37
Geo database, 184
`git`, 11, 14
`gnuplot`, 21, 41
Go, 197–200
comment, 197
compile, 200
main function, 198
module, 200
package, 198
pointer, 199
printing, 200
string concatenation, 198
tab writer, 199
Gosset, W. S., 183
great apes, 151–153
`grep`, 11, 16, 20, 384–385
`tr`, 385–386
guide tree, 136–139
`gunzip`, 11, 15, 88, 92
`gzip`, 11, 15, 88, 92

Hamlet, 84, 87, 91
hashing, 126, 127
`head`, 4, 16, 120
histone, 203
`history`, 4
Hochberg, Y., 189
Homebrew, 11, 15, 19, 398
Hominidae, 158, 159
homology, 30, 47
Huffman code, 88, 90
Huffman tree, 91
Huffman, D., 90
human genome, 202–204

indel, 30, 180
inexact matching, 112
insertion, 31
intestinal mucosa, inflammation of, 185
intron, 47
inverse suffix array, 71, 76
inversion, 97

Jukes-Cantor distance, 153
Jukes-Cantor equation, 55

keyword tree, 57–64
Knuth, D. E., 23

LaTeX, 64
`latex`, 58
lcp array, 73, 76–78, 80
lcp interval, 74, 76
lcp interval tree, 71, 75
lcp value, 75
`less`, 11, 16, 84
lines of descent, 163, 165
literate programming, 21, 23
longest common prefix, 71
longest repeat, 79
`ls`, 4, 11, 69, 94

M. genitalium, *see Mycoplasma genitalium*
macOS, 19, 68, 69

mafft, 136, 138
make, 11, 14
makeblastdb, 101
man, 4, 9, 11, 18, 27
match length, 67
match plot, 42–49
  sensitivity compared to alignment, 54
match probability, 66
maximal unique match (MUM), 94, 97
mean, 132
median, 132
merging reads, 122–126
meta key, 7
mismatch matrix, 152
mismatches per site, 152
mitochondrial genome, 153
mkdir, 4, 6
modulo (%), *see* remainder
molecular clock, 142, 146, 154
molecular distance, 152
Monte Carlo method, 184, 187
mouse chromosomes, 180
mouse mutation data, 179
move to front, 86–87
mRNA, 32, 43
ms, 172, 178
multi-dimensional matrix, 133
multiple sequence alignment, 132–140
mummer, 94–97, 131
mutation, 31, 96, 112
  compared to mismatch, 54, 55, 152
mutational space, 33
mv, 7
Mycoplasma genitalium, 2, 15
  gene lengths, 22, 23
  genes, 2, 16, 17, 25
  genome, 15, 27, 60
  genome length, 25
  genome size in bits, 88
  longest repeat, 66, 67
  nucleotide frequencies, 27
  protein families, 109
  proteome, 25, 91, 108, 109
  shortest unique substring, 69, 70, 130
  strand, 16, 21
Mysql, 190, 191

N_{50}, 126, 132
nanosecond precision, 69
neato, 106–108
neighbor-joining, 155, 157, 158
Newick format, 63
normal distribution, 186, 187
nucleosome, 203
nucleotides, 2
nucmer, 94, 97
null device (/dev/null), 69
null distribution, 51
null hypothesis, 37, 79

octal number, 11, 12
od, 11–13
odds ratio, 37, 38
open reading frame (ORF), 30, 32
Oracle, 190
ortholog, 42, 47, 53
overlap graph, 122, 123, 125
overlapping reads, 122–126

P-value, 103
package manager, 19
paired-end sequencing, 131
PAM matrix, 37–39
PAM70, 32
paralog, 42, 47, 53
paste, 37, 38
percent accepted mutations, 37
Phred score, 114, 117
Phylip, 138
Phylip format, 150
phylogeny, *see* tree
phylonium, 148, 153
pipe, *see* bash
Poisson distribution, 172, 177
polymorphic site, 151
population genetics, 161
population mutation rate, 172
population size, 164, 165
Postgresql, 190
prefix code, 88, 90
preprocessing, 64
primates, 153
printf, 11, 12
programs, overview, 397–398
progressive alignment, 136–140
Prosite database, 109
protable network graphics, png, 107
protein families, 106
ps2pdf, 58
pseudo-sample, 159
Pubmed database, 184
pwd, 4

quality score, 117, 130

random match, 43, 47
random number, 18, 166
read mapping, 114–121

recursion, 39, 40
  bottom up solution, 39, 41
  programming matrix, 39, 41
  run time, 41
  top down solution, 39–41
  tree, 40, 41
REG1B, 197
regular expression, 21, 24, 26, 393–395
relational data model, 190
relational database, 190–204
  client-server, 190
remainder (%), 18
reverse complement, 60
`rm`, 4, 7
`rmdir`, 4, 6, 7
`rpois`, 177
rRNA, 195
run time
  `blastn`, 120
  `bwa`, 120
  keyword tree, 64
  measure, 69
  `mummer`, 96
  optimal alignment, 94
runs of characters, 82, 84–86

SAM file, 114, 118, 119
sample with replacement, 159
sample without replacement, 175
`samtools`, 114, 119
scripting language, 61
`sed`, 136, 140, 384, 386–388
segregating site, 172, 177
`seq`, 18, 38
sequencing error, 118, 130
sexual organism, 162
shell, 3, 4, 9, *see* `bash`
shotgun sequencing, 93, 122, 127, 129, 131
shuffle array, 175
simians, 139
single nucleotide polymorphism (SNP), 96, 97, 180
`sort`, 18, 21, 27, 84
speciation, 53
species, 161
SQL, 190, 191, 193, 200
  attribute declaration, 192
  embedding, 197
  join tables, 195, 204
  limitations, 197
  `max`, 194
  `min`, 195
  query, 193
  `sum`, 202
Sqlite, 191
`sqlite3`, 191, 193, 198
standard error, 96
standard input, 96
standard output, 96
standard output stream, 199
statistical significance, 11, 21, 53, 54, 181, 186
statistics, 183–190
  multiple experiments, 187–190
  single experiments, 184–187
stop codon, 32
string rotation, 82, 83
Student's t test, 183, 184, 186
suffix, 65, 71, 84
suffix array, 70–81
  as alternative to suffix tree, 70
  common prefix, 72
  intervals, 72
suffix tree, 64–70, 75, 76
  as index, 64
  construction, 65
  drawing, 67
  internal node, 65
  leaf, 65
  longest repeat, 65
  path label, 65, 69, 73
  root, 65
  run time, 67, 68
  search, 65
  sentinel, 65, 67
  shortest unique substring, 69
  string depth, 71, 73
sum-of-pairs score, 134, 135, 138
synonymous mutation, 30, 33

`tabix`, 180, 181
`tail`, 4, 5, 8, 16, 27, 37
tape archive, 15
`tar`, 11, 15
text editor, 19, 42
text file, 15
The Origin of Species, 141
thread, parallel computing, 102, 121
three point criterion, 148, 151, 156
`touch`, 4, 6
`tr`, 11, 16, 45, 384
tree, 141–159
  bifurcating, 147
  bootstrap, 155
  branch lengths, 142
  directory, 383
  from distances, 148–159
  midpoint rooting, 144, 155, 158
  Newick format, 142

number of possible trees, 146
overdetermined, 156
parenthesis notation, 142
random, 175
recursive structure, 145
root, 141
rooted, 54, 147–154, 158
scale bar, 145
topology, 142, 154
traversal, 142, 145, 146
unrooted, 54, 144, 154–159
type I error, 189
type II error, 189

Ubuntu, 68, 398, 399
ultrametric distance, 148, 151, 156
ungapped alignment, 113
universal ancestor, *see* ancestors
Unix, 3, 19
graphics, `x11`, 107
POSIX standard, 3
Unix epoch, 69
untranslated region (UTR), 32
Unweighted Pair-Group Method using an Arithmetic average, Upgma, 150, 156

variable length code, 88–90
variance, 21–23
variant call format (VCF), 172, 180
`velvetg`, 126, 128, 131
`velveth`, 126, 128, 131

waiting time, 165, 170
Watterson's equation, 172, 177, 181
`wc`, 11, 20
`wget`, 15, 19
`which`, 11
wildcard (∗), 4
Windows, 68
Wright, S., 166
Wright-Fisher model, 165, 166, 172, 173
common ancestor, 167, 168
lines of descent, 172, 173
most recent common ancestor, 165, 169, 171, 172
number of lineages, 170, 171
pick ancestor, 167, 169
population size, 168, 169
sample size, 169
simulate, 170
test, 179
Wright-Fisher population, 167

XML, 197
`xtract`, 191, 197

z shell, 5
`zsh`, 4, 5, 9, 166, 201