biobox: Tools for Molecular Biology

github.com/evolbioinf/biobox

Bernhard Haubold

August 21, 2023

Contents

Chapter 1

Introduction

The biobox is a collection of bioinformatics tools centered on the analysis of FASTA-formatted sequence data. Many of the programs in the biobox are used to illustrate bioinformatics ideas in the forthcoming second edition of *Bioinformatics for Evolutionary Biologists*, which I'm writing together with Angelika Börsch-Haubold. For example, the program `al` illustrates optimal alignment in its global, local, and overlap incarnations. Beyond calculating the actual alignments, `al` can also print the underlying alignment matrix. Another example is `sblast`, which illustrates a simple, ungapped, version of Blast. `sblast` not only calculates the alignments, it can also print the word list that seeds.

The biobox contains program on seven topics: alignment (Table 1.1), compression (Table 1.2), exact matching (Table 1.3), graphics (Table 1.4), sequence manipulation (Table 1.5), trees (Table 1.6), and statistics (Table 1.7). In addition, there is a catch-all group of miscellaneous programs (Table 1.8).

In the remainder of this document the programs are listed in alphabetical order.

Table 1.1: Biobox programs for alignment

Program	Function
`al`	optimal alignment
`kerror`	k-error alignment
`numAl`	number of possible alignments
`pam`	amino acid substitution matrices
`sass`	simple assembler
`sblast`	simple Blast
`sops`	sum-of-pairs score

Table 1.2: Biobox programs for compression

Program	Function
`bwt`	Burrows-Wheeler transform
`huff`	Huffman encoding
`hut`	Huffman tree
`mtf`	move to front

Table 1.3: Biobox programs for exact matching

Program	Function
`keyMat`	match with keyword tree
`maf`	match factors
`naiveMatcher`	naive exact matching
`olga`	overlap graph
`repeater`	maximal exact repeats
`shustring`	shortest unique substrings

Table 1.4: Biobox programs for graphics

Program	Function
`blast2dot`	convert Blast output to dot graph
`drag`	draw genealogy of diploid individuals
`drawKt`	draw keyword tree
`drawSt`	draw suffix tree
`drawGenes`	draw genes as simple boxes
`drawf`	draw Wright-Fisher population
`histogram`	draw histogram
`mum2plot`	convert MUMmer output to `plotSeg` input
`plotLine`	plot x/y data
`plotSeg`	segment (dot) plots
`plotTree`	plot phylogenetic trees
`ranDot`	random graph in dot notation
`rep2plot`	convert `repeater` output to `plotSeg` input

Table 1.5: Biobox programs for sequence manipulation

Program	Function
`cres`	count residues
`cutSeq`	cut regions from sequence
`fasta2tab`	convert FASTA data to table
`getSeq`	get sequence from FASTA file
`mutator`	mutate sequence
`pps`	print polymorphic sites
`randomizeSeq`	randomize sequence
`ranseq`	generate random sequence
`revComp`	reverse-complement DNA sequence
`sequencer`	simulate shotgun sequencing
`translate`	translate DNA sequence
`wrapSeq`	wrap data lines

Table 1.6: Biobox programs for tree manipulation

Program	Function
`clac`	clade counter
`dnaDist`	calculate DNA distances
`genTree`	generate random trees
`midRoot`	midpoint-root tree
`nj`	neighbor-joining
`travTree`	traverse tree
`upgma`	Upgma

Table 1.7: Biobox programs for statistics

Program	Function
`rpois`	Poisson-distributed random variables
`simNorm`	simulate normally distributed data
`simOrf`	simulate lengths of random open reading frames
`testMeans`	test the difference between means
`var`	calculate variance

Table 1.8: Biobox programs with miscellaneous functions

Program	Function
`geco`	random genetic code
`num2char`	convert numbers to characters
`watterson`	Watterson's equation

Chapter 2

Program `al`: Align Sequences

Introduction

The program `al` aligns two sequences, a query, q, and a subject, s. It computes a global alignment by default (Figure 2.1A), but the user can request a global alignment (Figure 2.1B) or an overlap (Figure 2.1C) alignment. `al` can align DNA sequences using match/mismatch scores, or protein sequences using substitution matrices. It uses an affine gap score, where a gap of length l has score

$$g(l) = g_{\mathrm{o}} + g_{\mathrm{e}}(l - 1),$$

and g_{o} is the gap opening score, g_{e} the gap extension score. The program is based on the package `github.com/evolbioinf/pal`, where the algorithms are described in detail.

Implementation

The program outline contains hooks for imports, variables, functions, and the logic of the main function.

7 ⟨*al.go* 7⟩≡

```
package main

import (
        ⟨Imports, Ch. 2 8b⟩
)
⟨Variables, Ch. 2 8e⟩
```

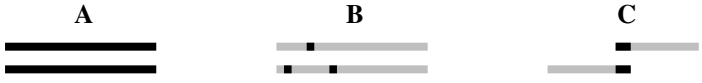


Figure 2.1: The three types of alignment, global (**A**), local (**B**), and overlap (**C**). Homology in black.

⟨*Functions, Ch. 2* 11c⟩
```
func main() {
        ⟨Main function, Ch. 2 8a⟩
}
```

In the main function we prepare the `log` package, set the usage, parse the options, and compute one or more alignments.

8a ⟨*Main function, Ch. 2* 8a⟩≡ (7)
```
util.PrepLog("al")
⟨Set usage, Ch. 2 8c⟩
⟨Parse options, Ch. 2 9c⟩
⟨Compute alignments, Ch. 2 11a⟩
```

We import `util`.

8b ⟨*Imports, Ch. 2* 8b⟩≡ (7) 8d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the usage proper, an explanation of the program's purpose, and an example command.

8c ⟨*Set usage, Ch. 2* 8c⟩≡ (8a)
```
u := "al [-h] [options] query.fasta [subject files]"
p := "Align two sequences."
e := "al query.fasta subject.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

8d ⟨*Imports, Ch. 2* 8b⟩+≡ (7) ◁8b 8f▷
```
"github.com/evolbioinf/clio"
```

Apart from the standard *version* option, we declarealgorithm options and output options.

8e ⟨*Variables, Ch. 2* 8e⟩≡ (7)
```
var optV = flag.Bool("v", false, "version")
⟨Algorithm options, Ch. 2 8g⟩
⟨Output options, Ch. 2 9a⟩
```

We import `flag`.

8f ⟨*Imports, Ch. 2* 8b⟩+≡ (7) ◁8d 9b▷
```
"flag"
```

With the algorithm options we pick the alignment type, set the scoring of scoring of pairs of residues and gaps, and choose the number of local alignments returned.

8g ⟨*Algorithm options, Ch. 2* 8g⟩≡ (8e)
```
var optL = flag.Bool("l", false, "local (default global)")
var optO = flag.Bool("o", false, "overlap (default global)")
var optI = flag.Float64("i", -3, "mismatch")
var optA = flag.Float64("a",  1, "match")
var optM = flag.String("m", "", "file containing score matrix")
var optP = flag.Float64("p", -5, "gap opening")
var optE = flag.Float64("e", -2, "gap extension")
var optN = flag.Int("n", 1, "number of local alignments")
```

With the output options we set the line length in the printout and can also opt to have the dynamic programming matrix printed.

9a ⟨*Output options, Ch. 2* 9a⟩≡ (8e)
```
var optLL = flag.Int("L", fasta.DefaultLineLength, "line length")
var optPP = flag.String("P", "", "print programming matrix (d|v|h|s|t)")
```

We import `fasta`.

9b ⟨*Imports, Ch. 2* 8b⟩+≡ (7) ◁8f 9e▷
```
"github.com/evolbioinf/fasta"
```

When parsing the options, we check for version printing and matrix printing. Then get the files for the query, the subject, and the score matrix.

9c ⟨*Parse options, Ch. 2* 9c⟩≡ (8a)
```
flag.Parse()
if *optV {
        util.PrintInfo("al")
}
⟨Check matrix printing, Ch. 2 9d⟩
⟨Get query and subject files, Ch. 2 10b⟩
⟨Get score matrix, Ch. 2 10c⟩
```

The four matrix elements are called `d` for *diagonal*, `v` for *vertical*, `h` for `horizontal`, and `s` for *score*. We can think of them as arranged in a square

d	v
h	s

Our program should check that only one of these four options has been entered, plus `t` for trace back. However, in package `pal`, these elements are called like this:

g	e
f	v

So we also translate to the nomenclature of `pal`.

9d ⟨*Check matrix printing, Ch. 2* 9d⟩≡ (9c)
```
m := "-P should be d, v, h, s for the cell element " +
        "or t for the traceback"
if *optPP != "" {
        if *optPP != "d" && *optPP != "v" &&
                (*optPP) != "h" && *optPP != "s" &&
                (*optPP) != "t" {
                fmt.Println(m)
                os.Exit(-1)
        }
}
⟨Translate cell nomenclature, Ch. 2 10a⟩
```

We import `fmt` and `os`.

9e ⟨*Imports, Ch. 2* 8b⟩+≡ (7) ◁9b 10d▷
```
"fmt"
"os"
```

As shown in the squares above, we translate

d to g
v to e
h to f
s to v

10a ⟨*Translate cell nomenclature, Ch. 2* 10a⟩≡ (9d)

```
if *optPP == "d" {
        (*optPP) = "g"
} else if *optPP == "v" {
        (*optPP) = "e"
} else if *optPP == "h" {
        (*optPP) = "f"
} else if *optPP == "s" {
        (*optPP) = "v"
}
```

When accessing the input files, we make sure that the user has actually given a query file.

10b ⟨*Get query and subject files, Ch. 2* 10b⟩≡ (9c)

```
files := flag.Args()
if len(files) < 1 {
        fmt.Fprintf(os.Stderr, "please give the name " +
                "of a query file\n")
        os.Exit(0)
}
query := files[0]
subject := files[1:]
```

The score matrix is either constructed from the match and mismatch scores, or read from a file.

10c ⟨*Get score matrix, Ch. 2* 10c⟩≡ (9c)

```
var mat *pal.ScoreMatrix
if *optM == "" {
        mat = pal.NewScoreMatrix(*optA, *optI)
} else {
        f, err := os.Open(*optM)
        if err != nil {
                log.Fatalf("couldn't open score matrix %q\n",
                        *optM)
        }
        mat = pal.ReadScoreMatrix(f)
        f.Close()
}
```

We import `pal`.

10d ⟨*Imports, Ch. 2* 8b⟩+≡ (7) ◁9e 11b▷

```
"github.com/evolbioinf/pal"
```

When computing the alignments, we iterate over the query sequences and pass each one to the `scan` function, together with the names of the subject files and the substitution matrix.

11a ⟨*Compute alignments, Ch. 2* 11a⟩≡ (8a)
```
qf, err := os.Open(query)
if err != nil {
        log.Fatalf("couldn't open %q\n", query)
}
sc := fasta.NewScanner(qf)
for sc.ScanSequence() {
        q := sc.Sequence()
        clio.ParseFiles(subject, scan, q, mat)
}
```

We import `log`.

11b ⟨*Imports, Ch. 2* 8b⟩+≡ (7) ◁10d 11d▷
```
"log"
```

In the function `scan`, the arguments just passed are retrieved again and we iterate over the subject sequences.

11c ⟨*Functions, Ch. 2* 11c⟩≡ (7)
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 2 11e⟩
        ⟨Iterate over subject sequences, Ch. 2 11f⟩
}
```

We import `io`.

11d ⟨*Imports, Ch. 2* 8b⟩+≡ (7) ◁11b 12c▷
```
"io"
```

The arguments are retrieved via type assertions, or as global variables.

11e ⟨*Retrieve arguments, Ch. 2* 11e⟩≡ (11c)
```
q := args[0].(*fasta.Sequence)
mat := args[1].(*pal.ScoreMatrix)
isLocal := *optL
isOverlap := *optO
gapO := *optP
gapE := *optE
numAl := *optN
var printMat byte
if *optPP != "" {
        printMat = []byte(*optPP)[0]
}
ll := *optLL
```

Then we iterate across the subject sequences and align each one with the query.

11f ⟨*Iterate over subject sequences, Ch. 2* 11f⟩≡ (11c)
```
sc := fasta.NewScanner(r)
for sc.ScanSequence() {
        s := sc.Sequence()
        ⟨Align query and subject, Ch. 2 12a⟩
}
```

We calculate either a local, an overlap, or a global alignment.

12a ⟨*Align query and subject, Ch. 2* 12a⟩≡ (11f)
```
if isLocal {
        ⟨Calculate local alignment, Ch. 2 12b⟩
} else if isOverlap {
        ⟨Calculate overlap alignment, Ch. 2 12d⟩
} else {
        ⟨Calculate global alignment, Ch. 2 13a⟩
}
```

We initialize a local alignment and set its line length. Then we align the requested number of times and print the matrix or the alignment.

12b ⟨*Calculate local alignment, Ch. 2* 12b⟩≡ (12a)
```
al := pal.NewLocalAlignment(q, s, mat, gapO, gapE)
al.SetLineLength(ll)
for i := 0; i < numAl && al.Align(); i++ {
        if printMat != 0 {
                s := al.PrintMatrix(printMat)
                fmt.Printf(s)
        } else {
                fmt.Println(al)
        }
}
```

We import `fmt`.

12c ⟨*Imports, Ch. 2* 8b⟩+≡ (7) ◁11d
```
"fmt"
```

Similarly, we initialize an overlap alignment, set its line length, carry out the actual alignment, and print it.

12d ⟨*Calculate overlap alignment, Ch. 2* 12d⟩≡ (12a)
```
al := pal.NewOverlapAlignment(q, s, mat, gapO, gapE)
al.SetLineLength(ll)
al.Align()
if printMat != 0 {
        s := al.PrintMatrix(printMat)
        fmt.Printf(s)
} else {
        fmt.Println(al)
}
```

Finally, we initialize the default global alignment, set its line length, carry out the alignment, and print it.

13a ⟨*Calculate global alignment, Ch. 2* 13a⟩≡ (12a)
```
al := pal.NewGlobalAlignment(q, s, mat, gapO, gapE)
al.SetLineLength(ll)
al.Align()
if printMat != 0 {
        s := al.PrintMatrix(printMat)
        fmt.Printf(s)
} else {
        fmt.Println(al)
}
```

The implementation of `al` is finished, time to test it.

Testing

The testing outline contains hooks for imports and the testing logic.

13b ⟨*al_test.go* 13b⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 2 13d⟩
)

func TestAl(t *testing.T) {
        ⟨Testing, Ch. 2 13c⟩
}
```

We construct a set of tests and run them.

13c ⟨*Testing, Ch. 2* 13c⟩≡ (13b)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 2 13e⟩
for i, test := range tests {
        ⟨Run test, Ch. 2 14e⟩
}
```

We import `exec`.

13d ⟨*Testing imports, Ch. 2* 13d⟩≡ (13b) 15▷
```
"os/exec"
```

In our tests, We go through the alignment types, beginning with global. First two short peptides are aligned

13e ⟨*Construct tests, Ch. 2* 13e⟩≡ (13c) 14a▷
```
test := exec.Command("./al", "-m", "BLOSUM62", "s1.fasta",
        "s2.fasta")
tests = append(tests, test)
```

In the next test, the alcohol dehydrogenase loci of two *Drosophila* species, *D. melanogaster* and *D. guanche*, are aligned. The two sequences are 4.8 kb and 4.4 kb long, which results in a fairly substantial computation, but `al` is reasonably quick.

14a ⟨*Construct tests, Ch. 2* 13e⟩+≡ (13c) ◁13e 14b▷

```
test = exec.Command("./al", "dmAdhAdhdup.fasta",
          "dgAdhAdhdup.fasta")
tests = append(tests, test)
```

We align a pair of artificial overlapping sequences using overlap alignment, `o1.fasta` and `o2.fasta`.

14b ⟨*Construct tests, Ch. 2* 13e⟩+≡ (13c) ◁14a 14c▷

```
test = exec.Command("./al", "-o", "o1.fasta", "o2.fasta")
tests = append(tests, test)
```

Next, we compute local alignments. First, just the best, then the top three.

14c ⟨*Construct tests, Ch. 2* 13e⟩+≡ (13c) ◁14b 14d▷

```
test = exec.Command("./al", "-l", "dmAdhAdhdup.fasta",
          "dgAdhAdhdup.fasta")
tests = append(tests, test)
test = exec.Command("./al", "-l", "-n", "3", "dmAdhAdhdup.fasta",
          "dgAdhAdhdup.fasta")
tests = append(tests, test)
```

Our last testing topic is printing the matrix, in all five variants. We use two short DNA sequences for this stored in `s3.fasta` and `s4.fasta`.

14d ⟨*Construct tests, Ch. 2* 13e⟩+≡ (13c) ◁14c

```
test = exec.Command("./al", "-P", "s", "s3.fasta", "s4.fasta")
tests = append(tests, test)
test = exec.Command("./al", "-P", "v", "s3.fasta", "s4.fasta")
tests = append(tests, test)
test = exec.Command("./al", "-P", "h", "s3.fasta", "s4.fasta")
tests = append(tests, test)
test = exec.Command("./al", "-P", "d", "s3.fasta", "s4.fasta")
tests = append(tests, test)
test = exec.Command("./al", "-P", "t", "s3.fasta", "s4.fasta")
tests = append(tests, test)
```

A test is run by storing the result we get and comparing it to the result we want, stored in files `r1.txt`, `r2.txt`, and so on.

14e ⟨*Run test, Ch. 2* 14e⟩≡ (13c)

```
get, err := test.Output()
if err != nil { t.Errorf("can't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("can't open %q", f) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

15 ⟨*Testing imports, Ch. 2* 13d⟩+≡ (13b) ◁13d

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 3

Program `blast2dot`: Convert BLAST Output to dot Code for Plotting

Table 3.1: Example BLAST results.

Query	Subject
p_1	p_1
p_1	p_2
p_2	p_1
p_2	p_2
p_3	p_1
p_3	p_3
p_4	p_4

A

```
graph G {
node [style=filled, color=lightsalmon]
          p1 -- p2 [dir=both]
          p3 -- p1 [dir=forward]
node [style=filled, color=lightgray]
          p4
}
```

B

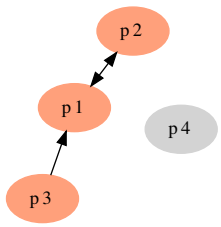


Figure 3.1: The homology relationships of Table 3.1 written in dot notation (**A**) and rendered with `neato` (**B**).

Introduction

The output of BLAST consist of lines of matches between a query and a subject. For example, let's say we have four protein sequences, $\{p_1, p_2, p_3, p_4\}$, and we carry out an all against all BLAST run. Table 3.1 shows the results of this run. We can see that each protein is, of course, homologus to itself. In addition, p_1, p_2, and p_3 also belong to a protein family, while p_4 doesn't, we call it a singleton. Within the protein family, p_1 and p_2 are connected by a reciprocal hit, $p_1 \leftrightarrow p_2$, while for p_1 and p_3 there is only $p_3 \rightarrow p_1$ and no $p_1 \rightarrow p_3$.

The way we have just talked about the relationships between our proteins is the language of graphs. Graphs are commonly written in the dot notation, and Figure 3.1A shows the relationship among our four example proteins in this notation. When rendered with the program `neato`, which is part of the free GraphViz package, we get Figure 3.1B, an explicit graph of the homology relationships among our four proteins. The program `blast2dot` reads BLAST output and writes the homology relationships it contains as `neato` input.

Implementation

The outline of `blast2dot` contains hooks for imports, functions, and the logic of the main function.

17 ⟨*blast2dot.go* 17⟩≡

```
package main

import (
        ⟨Imports, Ch. 3 18b⟩
)

⟨Functions, Ch. 3 19c⟩
func main() {
        ⟨Main function, Ch. 3 18a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare and parse the options, and parse the input files.

18a ⟨*Main function, Ch. 3* 18a⟩≡ (17)
```
util.PrepLog("blast2dot")
⟨Set usage, Ch. 3 18c⟩
⟨Declare options, Ch. 3 18e⟩
⟨Parse options, Ch. 3 19a⟩
⟨Parse input files, Ch. 3 19b⟩
```

We import `util`.

18b ⟨*Imports, Ch. 3* 18b⟩≡ (17) 18d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the program's purpose, and an example command.

18c ⟨*Set usage, Ch. 3* 18c⟩≡ (18a)
```
u := "blast2dot [-h] [option]... [file]..."
p := "Convert BLAST output to dot code " +
        "for plotting with GraphViz programs " +
        "like dot, neato, or circo."
e := "blast2dot -C lightgray -c lightsalmon foo.bl | neato -T x11"
clio.Usage(u, p, e)
```

We import `clio`.

18d ⟨*Imports, Ch. 3* 18b⟩+≡ (17) ◁18b 18f▷
```
"github.com/evolbioinf/clio"
```

We may be interested only in the gene families. So by default we don't include singletons in the output, but the user can chose to do so (`-s`). (S)he can also set the color of the gene families (`-c`) and the singletons (`-C`), and ask for the version (`-v`). As `-C` is alphabetically less than `-c`, we add the hint where to find color names to `-C`.

18e ⟨*Declare options, Ch. 3* 18e⟩≡ (18a)
```
var optS = flag.Bool("s", false, "include singletons")
var optC = flag.String("c", "", "color of gene families")
var optCC = flag.String("C", "", "color of singletons; color names: " +
        "www.graphviz.org/doc/info/colors.html")
var optV = flag.Bool("v", false, "version")
```

We include `flag`.

18f ⟨*Imports, Ch. 3* 18b⟩+≡ (17) ◁18d 19e▷
```
"flag"
```

We parse the options, and respond to `-v`. Moreover, if the user set a color for the singletons with `-C`, the implication is that singletons should be printed and we set `-s` to true.

19a ⟨*Parse options, Ch. 3* 19a⟩≡ (18a)
```
flag.Parse()
if *optV {
        util.PrintInfo("blast2dot")
}
if *optCC != "" { *optS = true }
```

The remaining tokens on the command line are taken as the names of input files. These are parsed with the function `scan`, which takes as argument the options that determine the appearance of the graph, singletons (`-s`), family color (`-c`), and singleton color (`-C`).

19b ⟨*Parse input files, Ch. 3* 19b⟩≡ (18a)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optS, *optC, *optCC)
```

In `scan` we retrieve the options, read the accessions, and reserve space for them in the variable `accessions`. This is a map between the accession string and an integer we shall later use as an index. We also store the relationships between the accessions in the variable `families`. It is a map of maps to associate a query with all the subjects it is homologous to. The information stored in these two variables is then converted to a match matrix. Based on that, we write the graph.

19c ⟨*Functions, Ch. 3* 19c⟩≡ (17)
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve options, Ch. 3 19d⟩
        accessions := make(map[string]int)
        families := make(map[string]map[string]bool)
        n := 1
        sc := bufio.NewScanner(r)
        ⟨Read accessions, Ch. 3 20a⟩
        ⟨Construct match matrix, Ch. 3 20d⟩
        ⟨Write graph, Ch. 3 21b⟩
}
```

We retrieve the options by type assertion.

19d ⟨*Retrieve options, Ch. 3* 19d⟩≡ (19c)
```
optS := args[0].(bool)
optC := args[1].(string)
optCC := args[2].(string)
```

We import `io` and `bufio`.

19e ⟨*Imports, Ch. 3* 18b⟩+≡ (17) ◁18f 20b▷
```
"io"
"bufio"
```

We read the query and subject accessions. Each new query is assigned a new index number. We also store the match.

20a ⟨*Read accessions, Ch. 3* 20a⟩≡ (19c)
```
for sc.Scan() {
        line := sc.Text()
        fields := strings.Fields(line)
        query := fields[0]
        sbjct := fields[1]
        if accessions[query] == 0 {
                accessions[query] = n
                n++
        }
        ⟨Store match, Ch. 3 20c⟩
}
```

We import `strings`.

20b ⟨*Imports, Ch. 3* 18b⟩+≡ (17) ◁19e 21e▷
```
"strings"
```

If we are not dealing with a hit to self, the subject is stored in its matching family.

20c ⟨*Store match, Ch. 3* 20c⟩≡ (20a)
```
if query != sbjct {
        if families[query] == nil {
                families[query] = make(map[string]bool)
        }
        qm := families[query]
        qm[sbjct] = true
}
```

The match matrix is an $n \times n$ matrix of boolean entries. If $m_{i,j}$ is true, query i matches subject j. We allocate this matrix. The indexes we have in hand range over the interval $(1, n)$. However, we need indexes over the interval $(0, n-1)$, so we adjust them. Then we fill the match matrix.

20d ⟨*Construct match matrix, Ch. 3* 20d⟩≡ (19c)
```
mm := make([][]bool, n)
for i := 0; i < n; i++ {
        mm[i] = make([]bool, n)
}
⟨Adjust indexes, Ch. 3 20e⟩
⟨Fill match matrix, Ch. 3 21a⟩
```

We reduce each accession index by one.

20e ⟨*Adjust indexes, Ch. 3* 20e⟩≡ (20d)
```
for k, v := range accessions {
        accessions[k] = v - 1
}
```

Every cell in the match matrix that corresponds to a BLAST hit is set to true.

21a ⟨*Fill match matrix, Ch. 3* 21a⟩≡ (20d)

```
for q, m := range families {
        i := accessions[q]
        for s, _ := range m {
                j := accessions[s]
                mm[i][j] = true
        }
}
```

To write the graph, we need to go from an index to an accession, so we store the accessions in a string slice that embodies this mapping. Then we write the three parts of the graph, the header, the body, and the footer.

21b ⟨*Write graph, Ch. 3* 21b⟩≡ (19c)

```
⟨Map indexes to accessions, Ch. 3 21c⟩
⟨Write header, Ch. 3 21d⟩
⟨Write body, Ch. 3 21f⟩
⟨Write footer, Ch. 3 22d⟩
```

There are n accessions, so we store them at their correct positions in a slice of that size.

21c ⟨*Map indexes to accessions, Ch. 3* 21c⟩≡ (21b)

```
names := make([]string, n)
for k, v := range accessions {
        names[v] = k
}
```

In the header we explain in a comment that the graph was generated with `blast2dot` and how to render it. Then we open it.

21d ⟨*Write header, Ch. 3* 21d⟩≡ (21b)

```
fmt.Println("# Graph written by blast2dot.")
fmt.Println("# Render: dot|neato|circo foo.dot")
fmt.Println("graph G {")
```

We import `fmt`.

21e ⟨*Imports, Ch. 3* 18b⟩+≡ (17) ◁20b

```
"fmt"
```

The graph body consists of the gene families and, if requested, the singletons.

21f ⟨*Write body, Ch. 3* 21f⟩≡ (21b)

```
⟨Write gene families, Ch. 3 22a⟩
if optS {
        ⟨Write singletons, Ch. 3 22c⟩
}
```

The members of gene families are plotted in nodes that may be tinted. Once the node color is specified, we go through the match matrix and write the query/subject pairs.

22a ⟨*Write gene families, Ch. 3* 22a⟩≡ (21f)
```
if optC != "" {
        fmt.Printf("node [style=filled, color=%s]\n", optC)
}
for i, v := range mm {
        for j, _ := range v {
                if i != j && mm[i][j] {
                        ⟨Write query/subject pair, Ch. 3 22b⟩
                }
        }
}
```

A query/subject par can be reciprocal or one-sided. To avoid duplications, we set the cells we are done with to false.

22b ⟨*Write query/subject pair, Ch. 3* 22b⟩≡ (22a)
```
fmt.Printf("\t%s -- %s[dir=", names[i], names[j])
if mm[j][i] {
        fmt.Printf("both]\n")
        mm[j][i] = false
} else {
        fmt.Printf("forward]\n")
}
mm[i][j] = false
```

The singletons are those accessions that are not part of a gene family. Again, their nodes may or may not be tinted. If their nodes are not tinted, but those of the gene families were, we reset the node style to default.

22c ⟨*Write singletons, Ch. 3* 22c⟩≡ (21f)
```
if optCC != "" {
        fmt.Printf("node [style=filled, color=%s]\n", optCC)
} else if optC != "" {
        fmt.Println("nod [style=\"\", color=\"\"]")
}
for k, _ := range accessions {
        if families[k] == nil {
                fmt.Printf("\t%s\n", k)
        }
}
```

The footer just closes the curly bracket opened in the first line of the graph.

22d ⟨*Write footer, Ch. 3* 22d⟩≡ (21b)
```
fmt.Println("}")
```

We have finished writing `blast2dot`, so let's test it.

Testing

Our testing code has hooks for imports and the testing logic.

23a ⟨*blast2dot_test.go* 23a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 3 23c⟩
)

func TestBlast2dot(t *testing.T) {
        ⟨Testing, Ch. 3 23b⟩
}
```

We construct the tests and run them.

23b ⟨*Testing, Ch. 3* 23b⟩≡ (23a)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 3 23d⟩
for i, test := range tests {
        ⟨Run test, Ch. 3 23e⟩
}
```

We import `exec`.

23c ⟨*Testing imports, Ch. 3* 23c⟩≡ (23a) 24▷
```
"os/exec"
```

We construct two tests, one without singletons, one with. In both cases we color the nodes.

23d ⟨*Construct tests, Ch. 3* 23d⟩≡ (23b)
```
f := "test.bl"
cmd := exec.Command("./blast2dot", "-c", "lightsalmon", f)
tests = append(tests, cmd)
cmd = exec.Command("./blast2dot", "-c", "lightsalmon",
        "-C", "lightgray", f)
tests = append(tests, cmd)
```

We run the tests and compare what we get with what we want, which is stored in files like `r1.dot`.

23e ⟨*Run test, Ch. 3* 23e⟩≡ (23b)
```
get, err := test.Output()
if err != nil { t.Error(err.Error()) }
f = "r" + strconv.Itoa(i + 1) + ".dot"
want, err := ioutil.ReadFile(f)
if err != nil { t.Error(err.Error()) }
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n",
                string(get), string(want))
}
```

We import `strconv`, `ioutil`, and `bytes`.

24 ⟨*Testing imports, Ch. 3* 23c⟩+≡ (23a) ◁23c

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 4

Program `bwt`: Burrows-Wheeler Transform

Table 4.1: Rotation (**A**) and sorted rotation (**B**).

A	**B**
TTAAAATTTA$	$TTAAAATTTA
TAAAATTTA$T	A$TTAAAATTT
AAAATTTA$TT	AAAATTTA$TT
AAATTTA$TTA	AAATTTA$TTA
AATTTA$TTAA	AATTTA$TTAA
ATTTA$TTAAA	ATTTA$TTAAA
TTTA$TTAAAA	TA$TTAAAATT
TTA$TTAAAAT	TAAAATTTA$T
TA$TTAAAATT	TTA$TTAAAAT
A$TTAAAATTT	TTAAAATTTA$
$TTAAAATTTA	TTTA$TTAAAA

Introduction

A simple, but effective method for compressing a string is to summarize runs of identical characters. For example a sequence

TTAAAATTTA

could be compressed as

$$\mathtt{T}_2\mathtt{A}_4\mathtt{T}_3\mathtt{A}$$

Now, the most efficient compression would be to sort the characters to give $\mathtt{A}_5\mathtt{T}_5$, but this is irreversible. Still, sorting is not a bad idea when applied to the rotations of a string. For our example the string rotations are shown in Table 4.1A and their sorted version in Table 4.1B. Notice the sentinel character, `$`, at the end. The last column in the sorted rotation is the transform,

ATTAAATTT$A

It is called the Burrows-Wheeler transform published by Michael Burrows and David Wheeler [5].

In longer texts the transform clusters identical characters into runs, which can then be compressed. The transform can be reversed, or decoded, using the simple linear-time algorithm listed in [1, p. 26].

Given the suffix array, sa of text T, we can look up the transform as $T[sa[i]-1]$, if we realize that the character to the left of the first suffix, $T[sa[i]-1]$ must be the sentinel. This way we can transform a string without rotating it first.

The program `bwt` takes as input a sequence in FASTA format and returns its transform. It can also read a transformed sequence and decode it.

Implementation

Our implementation of `bwt` contains hooks for imports, functions, and the logic of the main function.

26 ⟨*bwt.go* 26⟩≡

```
package main

import (
        ⟨Imports, Ch. 4 27b⟩
)
⟨Functions, Ch. 4 28b⟩
func main() {
        ⟨Main function, Ch. 4 27a⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

27a ⟨*Main function, Ch. 4* 27a⟩≡ (26)
```
util.PrepLog("bwt")
⟨Set usage, Ch. 4 27c⟩
⟨Declare options, Ch. 4 27e⟩
⟨Parse options, Ch. 4 27g⟩
⟨Parse input files, Ch. 4 28a⟩
```

We import `util`.

27b ⟨*Imports, Ch. 4* 27b⟩≡ (26) 27d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `bwt`, and an example command.

27c ⟨*Set usage, Ch. 4* 27c⟩≡ (27a)
```
u := "bwt [-h] [option]... [foo.fasta]..."
p := "Compute the Burrows-Wheeler transform."
e := "bwt foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

27d ⟨*Imports, Ch. 4* 27b⟩+≡ (26) ◁27b 27f▷
```
"github.com/evolbioinf/clio"
```

Apart from the version (`-v`), we declare an option for decoding `-d`.

27e ⟨*Declare options, Ch. 4* 27e⟩≡ (27a)
```
var optV = flag.Bool("v", false, "version")
var optD = flag.Bool("d", false, "decode")
```

We import `flag`.

27f ⟨*Imports, Ch. 4* 27b⟩+≡ (26) ◁27d 28c▷
```
"flag"
```

We parse the options and respond to `-v` as this stops the program.

27g ⟨*Parse options, Ch. 4* 27g⟩≡ (27a)
```
flag.Parse()
if *optV {
        util.PrintInfo("bwt")
}
```

The remaining tokens on the command line are taken as input files. We parse each of them using the function `scan`, which in turn takes the decode option (`-d`) as argument.

28a ⟨*Parse input files, Ch. 4* 28a⟩≡ (27a)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optD)
```

Inside `scan` we retrieve the decode option, iterate over the sequences and transform each one before we print it.

28b ⟨*Functions, Ch. 4* 28b⟩≡ (26)
```
func scan(r io.Reader, args ...interface{}) {
        decode := args[0].(bool)
        var in, out *fasta.Sequence
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                in = sc.Sequence()
                ⟨Transform sequence, Ch. 4 28d⟩
                fmt.Printf("%s\n", out)
        }
}
```

We import `io`, `fasta`, and `fmt`.

28c ⟨*Imports, Ch. 4* 27b⟩+≡ (26) ◁27f 29c▷
```
"io"
"github.com/evolbioinf/fasta"
"fmt"
```

We either decode or encode the sequence.

28d ⟨*Transform sequence, Ch. 4* 28d⟩≡ (28b)
```
if decode {
        ⟨Decode sequence, Ch. 4 28e⟩
} else {
        ⟨Encode sequence, Ch. 4 29d⟩
}
```

For decoding we follow Algorithm 2.1 in [1, p. 26]. It relies on a set of auxiliary arrays, which we construct and then calculate. The original sequence is decoded from them.

28e ⟨*Decode sequence, Ch. 4* 28e⟩≡ (28d)
```
⟨Construct auxiliary arrays, Ch. 4 28f⟩
⟨Calculate auxiliary arrays, Ch. 4 29a⟩
⟨Extract original sequence, Ch. 4 29b⟩
```

There are three auxiliary arrays, a character count, `count` (called K in the algorithm), the first position of each character in the first column of the sorted rotation, `first` (M), and the prior count of the current character, `prior` (C).

28f ⟨*Construct auxiliary arrays, Ch. 4* 28f⟩≡ (28e)
```
var count, first [256]int
var prior []int
transform := in.Data()
prior = make([]int, len(transform))
```

We iterate across the transform and compute the prior count of each character, which also gives us their total counts. Then we locate the first occurrence of each character in the first column of the rotation, where the characters appear in alphabetical order. Hence the position of their first appearance is the cumulative sum of their counts.

29a ⟨*Calculate auxiliary arrays, Ch. 4* 29a⟩≡ (28e)

```
for i, t := range transform {
          prior[i] = count[t]
          count[t]++
}
s := 0
for i, c := range count {
          first[i] = s
          s += c
}
```

We check the sentinel `$` appears exactly once. Then we start the reconstruction of the original string from the sentinel's position in the transform. At the end, we drop the sentinel from the output.

29b ⟨*Extract original sequence, Ch. 4* 29b⟩≡ (28e)

```
o := make([]byte, len(transform))
if c := bytes.Count(transform, []byte("$")); c != 1 {
          m := "sentinel, $, appears %d times rather than once"
          log.Fatalf(m, c)
}
i := bytes.IndexByte(transform, '$')
for j := len(transform) - 1; j > -1; j-- {
          o[j] = transform[i]
          i = prior[i] + first[transform[i]]
}
h := in.Header() + " - decoded"
out = fasta.NewSequence(h, o[:len(o)-1])
```

We import `bytes` and `log`.

29c ⟨*Imports, Ch. 4* 27b⟩+≡ (26) ◁28c 30a▷

```
"bytes"
"log"
```

To encode the sequence from its suffix array, as explained in the Introduction.

29d ⟨*Encode sequence, Ch. 4* 29d⟩≡ (28d)

```
data := in.Data()
sent := byte('$')
data = append(data, sent)
sa := esa.Sa(data)
o := make([]byte, len(data))
for i, s := range sa {
          o[i] = sent
          if s > 0 { o[i] = data[s-1] }
}
h := in.Header() + " - bwt"
out = fasta.NewSequence(h, o)
```

We import `esa`.

30a ⟨*Imports, Ch. 4* 27b⟩+≡ (26) ◁29c
```
"github.com/evolbioinf/esa"
```

We've finished with `bwt`, time to test it.

Testing

The outline of our testing program has hooks for imports and the testing logic.

30b ⟨*bwt_test.go* 30b⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 4 30d⟩
)

func TestBwt(t *testing.T) {
        ⟨Testing, Ch. 4 30c⟩
}
```

We construct a set of tests and run them.

30c ⟨*Testing, Ch. 4* 30c⟩≡ (30b)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 4 30e⟩
for i, test := range tests {
        ⟨Run test, Ch. 4 31a⟩
}
```

We import `exec`.

30d ⟨*Testing imports, Ch. 4* 30d⟩≡ (30b) 31b▷
```
"os/exec"
```

We construct two tests, which run on the test word used in Tabls 4.1 as input contained in `t1.fasta` and as transformed input in `t2.fasta`.

30e ⟨*Construct tests, Ch. 4* 30e⟩≡ (30c)
```
test := exec.Command("./bwt", "t1.fasta")
tests = append(tests, test)
test = exec.Command("./bwt", "-d", "t2.fasta")
tests = append(tests, test)
```

We compare the result we get with the result we want stored in `r1.fasta` and `r2.fasta`.

31a ⟨*Run test, Ch. 4* 31a⟩≡ (30c)

```
get, err := test.Output()
if err != nil { t.Errorf("can't run %s", test) }
f := "r" + strconv.Itoa(i+1) + ".fasta"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("can't read %q", f) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

31b ⟨*Testing imports, Ch. 4* 30d⟩+≡ (30b) ◁30d

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 5

Program `clac`: Clade Counter

Introduction

Bootstrapping phylogenies is a standard method in molecular evolution to quantify the reliability of individual clades. The quantification relies on generating a large number of resampled trees, from which we count the clades. The program `clac` is a clade counter. It reads as input a stream of trees and prints a count of all clades encountered sorted by count. For example, when given the twelve random trees in Figure 5.1, `clac` prints

```
#ID Count Taxa Clade
1   10    2    {T1, T2}
2   8     2    {T4, T5}
3   6     3    {T1, T2, T3}
4   4     3    {T3, T4, T5}
5   4     2    {T3, T4}
6   2     4    {T1, T2, T3, T4}
7   1     3    {T2, T3, T4}
8   1     2    {T2, T3}
```

Alternatively, `clac` can also read a file of one or more reference trees and label their nodes with bootstrap percentages. For example, if Figure 5.1A is the reference and all others the input, the bootstrap tree is Figure 5.2.

Implementation

Our outline of `clac` contains hooks for imports, types, methods, functions, and the logic of the main function.

33a ⟨*clac.go* 33a⟩≡

```
package main

import (
        ⟨Imports, Ch. 5 35a⟩
)
⟨Types, Ch. 5 39a⟩
⟨Methods, Ch. 5 39b⟩
⟨Functions, Ch. 5 37a⟩
func main() {
        ⟨Main function, Ch. 5 33b⟩
}
```

In the main function we prepare the `log` package, set the usage of `clac`, declare its options, parse the options, and parse the input files.

33b ⟨*Main function, Ch. 5* 33b⟩≡ (33a)

```
util.PrepLog("clac")
⟨Set usage, Ch. 5 35b⟩
⟨Declare options, Ch. 5 35d⟩
⟨Parse options, Ch. 5 36a⟩
⟨Parse input files, Ch. 5 36e⟩
```

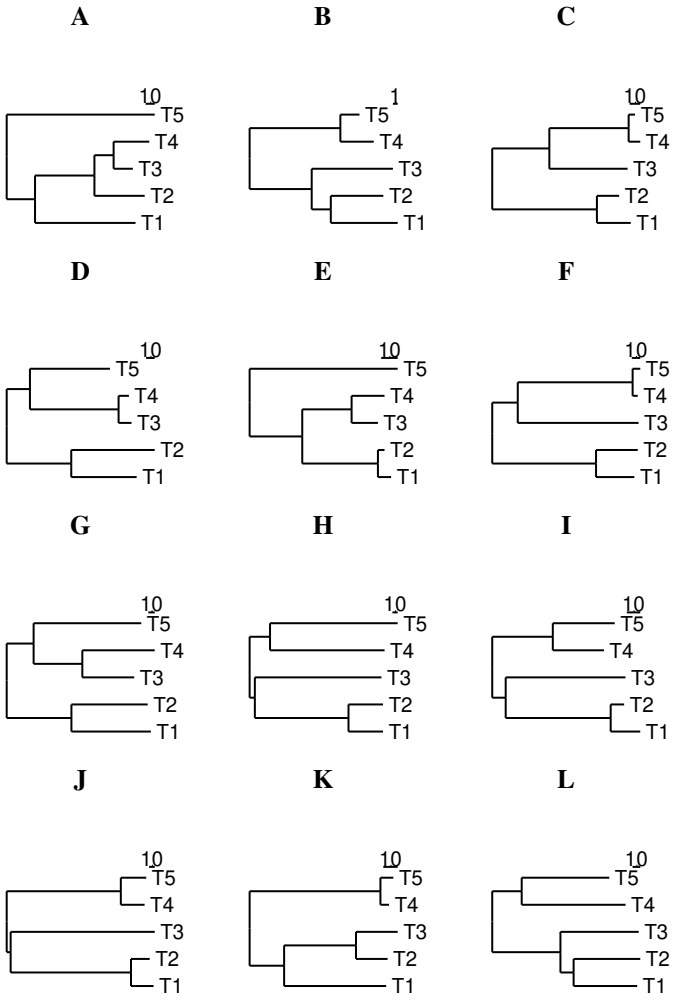


Figure 5.1: A dozen random example trees

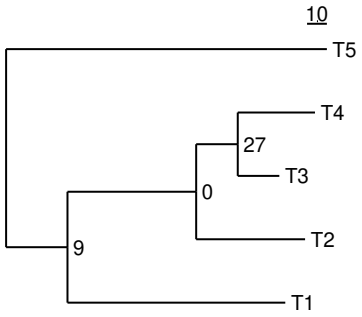


Figure 5.2: Bootstrap values for the tree in Figure 5.1A when compared to the trees in Figure 5.1B–L.

We import `util`.

35a ⟨*Imports, Ch. 5* 35a⟩≡ (33a) 35c ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of `clac`'s purpose, and an example command.

35b ⟨*Set usage, Ch. 5* 35b⟩≡ (33b)
```
u := "clac [-h] [option]... [trees.nwk]..."
p := "Count the clades in phylogenies."
e := "dnaDist -b 1000 foo.fasta | nj | clac"
clio.Usage(u, p, e)
```

We import `clio`.

35c ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁35a 35e ▷
```
"github.com/evolbioinf/clio"
```

We declare two options, the version, and an option to read reference trees from a file.

35d ⟨*Declare options, Ch. 5* 35d⟩≡ (33b)
```
var optV = flag.Bool("v", false, "version")
var optR = flag.String("r", "", "file of reference tree(s)")
```

We import `flag`.

35e ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁35c 36b ▷
```
"flag"
```

We parse the options and first respond to `-v` as this stops the program. If the user supplied a file of reference trees, we read them into the slice of trees we set aside.

36a ⟨*Parse options, Ch. 5* 36a⟩≡ (33b)
```
flag.Parse()
if *optV {
        util.PrintInfo("clac")
}
var refTrees []*nwk.Node
if *optR != "" {
        ⟨Read reference trees, Ch. 5 36c⟩
}
```

We import `nwk`.

36b ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁35e 36d▷
```
"github.com/evolbioinf/nwk"
```

We open the file of reference trees, read the trees, and store them.

36c ⟨*Read reference trees, Ch. 5* 36c⟩≡ (36a)
```
tf, err := os.Open(*optR)
if err != nil {
        log.Fatalf("couldn't open %q", *optR)
}
defer tf.Close()
sc := nwk.NewScanner(tf)
for sc.Scan() {
        refTrees = append(refTrees, sc.Tree())
}
```

We import `log`.

36d ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁36b 37b▷
```
"log"
```

The remaining tokens on the command line are interpreted as input files. We parse them with the function `scan`, which takes as argument the reference trees.

36e ⟨*Parse input files, Ch. 5* 36e⟩≡ (33b)
```
files := flag.Args()
clio.ParseFiles(files, scan, refTrees)
```

Inside `scan`, we retrieve the reference trees and iterate over the trees in the input file. We count the trees and the clades in the trees and print the results.

37a ⟨*Functions, Ch. 5* 37a⟩≡ (33a) 37c▷
```
func scan(r io.Reader, args ...interface{}) {
        refTrees := args[0].([]*nwk.Node)
        sc := nwk.NewScanner(r)
        clades := make(map[string]int)
        nt := 0
        for sc.Scan() {
                root := sc.Tree()
                nt++
                countClades(root, clades)
        }
        ⟨Print results, Ch. 5 37d⟩
}
```

We import `io`.

37b ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁36d 37f▷
```
"io"
```

We count the clades defined by the internal nodes in a recursive tree traversal.

37c ⟨*Functions, Ch. 5* 37a⟩+≡ (33a) ◁37a 38a▷
```
func countClades(v *nwk.Node, c map[string]int) {
        if v == nil { return }
        if v.Parent != nil && v.Child != nil {
                k := v.Key("$")
                c[k]++
        }
        countClades(v.Child, c)
        countClades(v.Sib, c)
}
```

When printing the results, we either print the reference trees annotated with bootstrap-percentages, or all clades and their counts.

37d ⟨*Print results, Ch. 5* 37d⟩≡ (37a)
```
if len(refTrees) > 0 {
        ⟨Print reference trees, Ch. 5 37e⟩
} else {
        ⟨Print clades, Ch. 5 38c⟩
}
```

We annotate each reference tree with percent clade counts and print it.

37e ⟨*Print reference trees, Ch. 5* 37e⟩≡ (37d)
```
for _, root := range refTrees {
        annotateTree(root, clades, nt)
        fmt.Println(root)
}
```

We import `fmt`.

37f ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁37b 38b▷
```
"fmt"
```

We annotate the internal nodes of a reference tree in a recursive tree traversal. Each internal node is labeled with a bootstrap percentages rounded to the nearest integer.

38a ⟨*Functions, Ch. 5* 37a⟩+≡ (33a) ◁37c

```
func annotateTree(v *nwk.Node, clades map[string]int, nc int) {
        if v == nil { return }
        if  v.Parent != nil && v.Child != nil {
                p := float64(clades[v.Key("$")]) /
                        float64(nc) * 100.0
                p = math.Round(p)
                v.Label = strconv.Itoa(int(p))
        }
        annotateTree(v.Child, clades, nc)
        annotateTree(v.Sib, clades, nc)
}
```

We import `math` and `strconv`.

38b ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁37f 38d▷

```
"math"
"strconv"
```

We sort the clades by count and print them in a table that we typeset with a tab writer. The table has four columns: clade-ID, clade count, the number of taxa in the clade, and the clade itself.

38c ⟨*Print clades, Ch. 5* 38c⟩≡ (37d)

```
w := tabwriter.NewWriter(os.Stdout, 1, 1, 1, ' ', 0)
fmt.Fprintf(w, "#ID\tCount\tTaxa\tClade\n")
⟨Sort clades, Ch. 5 38e⟩
⟨Print sorted clades, Ch. 5 39d⟩
w.Flush()
```

We import `tabwriter` and `os`.

38d ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁38b 38f▷

```
"text/tabwriter"
"os"
```

We store the clades in a slice, which we sort.

38e ⟨*Sort clades, Ch. 5* 38e⟩≡ (38c)

```
cs := make([]clade, 0)
var c clade
for k, n := range clades {
        c.k = k
        c.n = n
        cs = append(cs, c)
}
sort.Sort(cladeSlice(cs))
```

We import `sort`.

38f ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁38d 39e▷

```
"sort"
```

We declare the types `clade` and `cladeSlice`.

39a ⟨*Types, Ch. 5* 39a⟩≡ (33a)
```
type clade struct {
        k string
        n int
}
type cladeSlice []clade
```

We implement two of the three methods of the `sort` interface, `Len` and `Swap`.

39b ⟨*Methods, Ch. 5* 39b⟩≡ (33a) 39c ▷
```
func (c cladeSlice) Len() int {
        return len(c)
}
func (c cladeSlice) Swap(i, j int) {
        c[i], c[j] = c[j], c[i]
}
```

The third method of the sort interface, `Less` requires a bit more thought. The primary sort key is the count, but if the counts are equal, we sort alphabetically. This stabilizes our result, which might otherwise vary between runs.

39c ⟨*Methods, Ch. 5* 39b⟩+≡ (33a) ◁ 39b
```
func (c cladeSlice) Less(i, j int) bool {
        if c[i].n != c[j].n {
                return c[i].n < c[j].n
        } else {
                return c[i].k < c[j].k
        }
}
```

Having sorted the clades, we print them.

39d ⟨*Print sorted clades, Ch. 5* 39d⟩≡ (38c)
```
x := 0
for i := len(cs) - 1; i >= 0; i-- {
        x++
        taxa := strings.Split(cs[i].k, "$")
        t := len(taxa)
        fmt.Fprintf(w, "%d\t%d\t%d\t{", x, cs[i].n, t)
        for j, s := range taxa {
                if j > 0 { fmt.Fprintf(w, ", ") }
                fmt.Fprintf(w, s)
        }
        fmt.Fprintf(w, "}\n")
}
```

We import `strings`.

39e ⟨*Imports, Ch. 5* 35a⟩+≡ (33a) ◁ 38f
```
"strings"
```

We're done writing `clac`, time to test it.

Testing

The testing code for `clac` has hooks for imports and the testing logic.

40a ⟨*clac_test.go* 40a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 5 40c⟩
)

func TestClac(t *testing.T) {
        ⟨Testing, Ch. 5 40b⟩
}
```

We construct a number of tests and run them.

40b ⟨*Testing, Ch. 5* 40b⟩≡ (40a)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 5 40d⟩
for i, test := range tests {
        ⟨Run test, Ch. 5 40e⟩
}
```

We import `exec`.

40c ⟨*Testing imports, Ch. 5* 40c⟩≡ (40a) 41 ▷
```
"os/exec"
```

We construct two tests, the first without reference tree, where the input are the twelve trees in Figure 5.1. The second test takes Figure 5.1A as reference and Figures 5.1B–L as data.

40d ⟨*Construct tests, Ch. 5* 40d⟩≡ (40b)
```
test := exec.Command("./clac", "trees.nwk")
tests = append(tests, test)
test = exec.Command("./clac", "-r", "ref.nwk", "rest.nwk")
tests = append(tests, test)
```

We run a test and compare the result we get with the result we want. The results we want are contained in `r1.txt` and `r2.txt`.

40e ⟨*Run test, Ch. 5* 40e⟩≡ (40b)
```
get, err := test.Output()
if err != nil {
        t.Errorf("couldn't run %q", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
        t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

41 ⟨*Testing imports, Ch. 5* 40c⟩+≡ (40a) ◁40c

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 6

Program `cres`: Count Residues

Introduction

Our aim is to count the residues in sequences. What we in fact do, is to count the characters in sequences, without checking whether they are residues or not.

Implementation

The program outline contains hooks for imports, variables, functions, and the meat of the main function.

43a ⟨*cres.go* 43a⟩≡

```
package main

import (
        ⟨Imports, Ch. 6 43c⟩
)

⟨Variables, Ch. 6 43f⟩
⟨Functions, Ch. 6 44f⟩
func main() {
        ⟨Main function, Ch. 6 43b⟩
}
```

In the `main` function we first prepare the `log` package, then set the usage, parse the options set by the user, and finally the input.

43b ⟨*Main function, Ch. 6* 43b⟩≡ (43a)

```
util.PrepLog("cres")
⟨Set usage, Ch. 6 43d⟩
⟨Parse options, Ch. 6 44b⟩
⟨Parse input, Ch. 6 44d⟩
```

We import the package `util`.

43c ⟨*Imports, Ch. 6* 43c⟩≡ (43a) 43e ▷

```
"github.com/evolbioinf/biobox/util"
```

In addition to the usage, we describe the purpose and give an example command.

43d ⟨*Set usage, Ch. 6* 43d⟩≡ (43b)

```
u := "cres [-h] [options] [files]"
p := "Count residues in input."
e := "cres -s *.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

43e ⟨*Imports, Ch. 6* 43c⟩+≡ (43a) ◁ 43c 44a ▷

```
"github.com/evolbioinf/clio"
```

The user can request that each sequence is counted separately, `-s`. There is also the possibility to just print the version and additional information about the program, `-v`.

43f ⟨*Variables, Ch. 6* 43f⟩≡ (43a) 44c ▷

```
var optS = flag.Bool("s", false, "count sequences separately")
var optV = flag.Bool("v", false, "version")
```

This requires the `flag` package.

44a ⟨*Imports, Ch. 6* 43c⟩+≡ (43a) ◁43e 44e▷
```
"flag"
```

After parsing the flags, the program might just print its version.

44b ⟨*Parse options, Ch. 6* 44b⟩≡ (43b)
```
flag.Parse()
if *optV {
        util.PrintInfo("cres")
}
```

The values of `version` and `date` are injected at compile-time. Here, we just declare them.

44c ⟨*Variables, Ch. 6* 43f⟩+≡ (43a) ◁43f
```
var version, date string
```

The input files are parsed using `clio.ParseFiles`, which takes as argument a slice of file names and a function it applies to each file. This function takes as arguments the character counts, and an indicator as to whether or not it is dealing with the first sequence. At the end, `write` prints the counts. Characters are encoded as bytes that are eight bits long. So `counts` is a slice of $2^8 = 256$ long integers.

44d ⟨*Parse input, Ch. 6* 44d⟩≡ (43b)
```
files := flag.Args()
counts := make([]int64, 256)
isFirstSequence := true
clio.ParseFiles(files, scan, counts, *optS, &isFirstSequence)
write(counts, *optS)
```

We import `clio`.

44e ⟨*Imports, Ch. 6* 43c⟩+≡ (43a) ◁44a 45a▷
```
"github.com/evolbioinf/clio"
```

Before scanning a file, the arguments are retrieved. Then the data is scanned linewise. Each line is either a header or consists of characters to be counted. After using `ScanLine`, we flush the scanner.

44f ⟨*Functions, Ch. 6* 44f⟩≡ (43a) 45d▷
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 6 45b⟩
        scanner := fasta.NewScanner(r)
        for scanner.ScanLine() {
                if scanner.IsHeader() {
                        ⟨Deal with header, Ch. 6 45c⟩
                } else {
                        count(counts, scanner.Line())
                }
        }
        count(counts, scanner.Flush())
}
```

Import `io` and `fasta`.

45a ⟨*Imports, Ch. 6* 43c⟩+≡ (43a) ◁44e 45f▷
```
"io"
"github.com/evolbioinf/fasta"
```

As we saw above, `args` contains two arguments: the integer slice of counts, and the pointer to a boolean indicating whether or not we are dealing with the first sequence. We retrieve them by type assertions [7, p. 205].

45b ⟨*Retrieve arguments, Ch. 6* 45b⟩≡ (44f)
```
counts := args[0].([]int64)
separate := args[1].(bool)
isFirstSequence := args[2].(*bool)
```

The response to encountering a header depends on whether the user has requested separate counts for each sequence. If not, we do nothing. If yes, we print and reset counts whenever a header closes a sequence.

45c ⟨*Deal with header, Ch. 6* 45c⟩≡ (44f)
```
if separate {
        if *isFirstSequence {
                *isFirstSequence = false
        } else {
                write(counts, *optS)
                reset(counts)
        }
        fmt.Printf("%s: ", scanner.Line())
}
```

The function `write` prints the total number of characters, and their individual counts and frequencies.

45d ⟨*Functions, Ch. 6* 44f⟩+≡ (43a) ◁44f 46c▷
```
func write(counts []int64, separate bool) {
        ⟨Print total character count, Ch. 6 45e⟩
        ⟨Print individual counts, Ch. 6 46a⟩
}
```

We sum the individual character counts and print them either in "separate" or "total" mode.

45e ⟨*Print total character count, Ch. 6* 45e⟩≡ (45d)
```
var s int64
for _, v := range counts {
        s += v
}
if !separate {
        fmt.Printf("Total: ")
}
fmt.Printf("%d\n", s)
```

We import `fmt`.

45f ⟨*Imports, Ch. 6* 43c⟩+≡ (43a) ◁45a 46b▷
```
"fmt"
```

If any characters were found, we print the individual counts and frequencies in a table formatted using a `tabwriter`.

46a ⟨*Print individual counts, Ch. 6* 46a⟩≡ (45d)
```
w := new(tabwriter.Writer)
w.Init(os.Stdout, 4, 0, 1, ' ', 0)
if s > 0 {
        fmt.Fprintf(w, "Residue\tCount\tFraction\t\n")
}
for i, v := range counts {
        if v > 0 {
                fmt.Fprintf(w, "%c\t%d\t%.3g\t\n", i, v,
                        float64(v)/float64(s))
        }
}
w.Flush()
```

Import the `os` and `tabwriter`.

46b ⟨*Imports, Ch. 6* 43c⟩+≡ (43a) ◁45f
```
"os"
"text/tabwriter"
```

We reset the `counts`.

46c ⟨*Functions, Ch. 6* 44f⟩+≡ (43a) ◁45d 46d▷
```
func reset(counts []int64) {
        for i, _ := range counts {
                counts[i] = 0
        }
}
```

When we at last count the characters, they serve as indexes into the integer slice `counts`.

46d ⟨*Functions, Ch. 6* 44f⟩+≡ (43a) ◁46c
```
func count(counts []int64, data []byte) {
        for _, c := range data {
                counts[c]++
        }
}
```

Testing

We use the standard testing framework.

46e ⟨*cres_test.go* 46e⟩≡
```
package main
import (
        "testing"
        ⟨Testing imports, Ch. 6 47b⟩
)
func TestCres(t *testing.T) {
        ⟨Testing, Ch. 6 47a⟩
}
```

Our test is carried out on the file `test.fasta`, which contains two random sequences, each 100 nucleotides long. We first run `cres` with default options and compare the result with that contained in the file `res1.txt`.

47a ⟨*Testing, Ch. 6* 47a⟩≡ (46e) 47c ▷

```
cmd := exec.Command("./cres", "test.fasta")
o, err := cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
e, err := ioutil.ReadFile("res1.txt")
if err != nil {
          t.Error("couldn't open res1.txt")
}
if !bytes.Equal(o, e) {
          t.Errorf("wanted:\n%s\ngot:\n%s\n", string(e), string(o))
}
```

We import `exec`, `ioutil`, and `bytes`.

47b ⟨*Testing imports, Ch. 6* 47b⟩≡ (46e)

```
"os/exec"
"io/ioutil"
"bytes"
```

There is only one option to test, `-s` for counting sequences separately. This time, we compare the result to that contained in the file `res2.txt`.

47c ⟨*Testing, Ch. 6* 47a⟩+≡ (46e) ◁ 47a

```
cmd = exec.Command("./cres", "-s", "test.fasta")
o, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
e, err = ioutil.ReadFile("res2.txt")
if err != nil {
          t.Error("couldn't open res2.txt")
}
if !bytes.Equal(o, e) {
          t.Errorf("wanted:\n%s\ngot:\n%s\n", string(e), string(o))
}
```

Chapter 7

Program `cutSeq`: Cut Sequence Regions

Introduction

The program `cutSeq` cuts one or more regions from the sequences in the input. The regions' start and end positions are one-based and inclusive. The user can opt to join the regions.

Implementation

The program outline contains hooks for imports, types, variables, functions, and the logic of the main function.

49a ⟨*cutSeq.go* 49a⟩≡
```
package main

import (
        ⟨Imports, Ch. 7 49c⟩
)
⟨Types, Ch. 7 50f⟩
⟨Variables, Ch. 7 50c⟩
⟨Functions, Ch. 7 52e⟩

func main() {
        ⟨Main function, Ch. 7 49b⟩
}
```

In the main function, we prepare the `log` package, set the usage, and parse the options and input files.

49b ⟨*Main function, Ch. 7* 49b⟩≡ (49a)
```
util.PrepLog("cutSeq")
⟨Set usage, Ch. 7 49d⟩
⟨Parse options, Ch. 7 50a⟩
⟨Parse input, Ch. 7 52d⟩
```

We import `util`.

49c ⟨*Imports, Ch. 7* 49c⟩≡ (49a) 49e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts: The usage proper, the program's purpose, and an example command.

49d ⟨*Set usage, Ch. 7* 49d⟩≡ (49b)
```
u := "cutSeq [-h] [options] [files]"
p := "Cut regions from sequence."
e := "cutSeq -r 10-20,25-50 *.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

49e ⟨*Imports, Ch. 7* 49c⟩+≡ (49a) ◁ 49c 50b ▷
```
"github.com/evolbioinf/clio"
```

After parsing the options, we check whether the version is to be printed.

50a ⟨*Parse options, Ch. 7* 50a⟩≡ (49b) 50d▷
```
flag.Parse()
if *optV {
        util.PrintInfo("cutSeq")
}
```

We import `flag`.

50b ⟨*Imports, Ch. 7* 49c⟩+≡ (49a) ◁49e 50e▷
```
"flag"
```

Apart from the version flag, `-v`, we also declare a flag for regions to cut, `-r`, for a file with regions, `-f`, and for joining the regions, `-j`.

50c ⟨*Variables, Ch. 7* 50c⟩≡ (49a)
```
var optV = flag.Bool("v", false, "version")
var optR = flag.String("r", "", "regions")
var optF = flag.String("f", "", "file with regions; " +
        "one white-space delimited start/end pair per line")
var optJ = flag.Bool("j", false, "join regions")
```

We continue parsing the options by reading the regions either from the command line or from a file into a slice.

50d ⟨*Parse options, Ch. 7* 50a⟩+≡ (49b) ◁50a
```
var regions []region
if *optR != "" {
        ⟨Parse regions from command line, Ch. 7 51a⟩
} else if *optF != "" {
        ⟨Parse regions from file, Ch. 7 51d⟩
} else {
        fmt.Fprintf(os.Stderr,
                "Please provide a region to cut " +
                        "either via -r or -f.\n")
        os.Exit(0)
}
```

We import `fmt` and `os`.

50e ⟨*Imports, Ch. 7* 49c⟩+≡ (49a) ◁50b 51b▷
```
"fmt"
"os"
```

An individual region is a structure to store a pair of start and end positions.

50f ⟨*Types, Ch. 7* 50f⟩≡ (49a)
```
type region struct {
        start, end int
}
```

On the command line, regions are separated by commas, and positions by hyphens. We check a region before appending it.

51a ⟨*Parse regions from command line, Ch. 7* 51a⟩≡ (50d)
```
re := strings.Split(*optR, ",")
for _, x := range re {
        y := strings.Split(x, "-")
        r := *new(region)
        r.start, _ = strconv.Atoi(y[0])
        r.end, _ = strconv.Atoi(y[1])
        ⟨Check region, Ch. 7 51c⟩
        regions = append(regions, r)
}
```

We import `strings` and `strconv`.

51b ⟨*Imports, Ch. 7* 49c⟩+≡ (49a) ◁50e 51f▷
```
"strings"
"strconv"
```

A sensible regions should have a positive start position that is no greater than the end. Otherwise, we ignore that region and warn the user.

51c ⟨*Check region, Ch. 7* 51c⟩≡ (51a 52a)
```
if r.start < 1 || r.start > r.end || x[0] == '-' ||
        strings.Index(x, "--") > -1 {
        fmt.Fprintf(os.Stderr, "ignoring (%s)\n", x)
        continue
}
```

When parsing a file, we open it, scan it, and close it again.

51d ⟨*Parse regions from file, Ch. 7* 51d⟩≡ (50d)
```
⟨Open file, Ch. 7 51e⟩
⟨Scan file, Ch. 7 52a⟩
⟨Close file, Ch. 7 52c⟩
```

If we can't open the input file, we abort.

51e ⟨*Open file, Ch. 7* 51e⟩≡ (51d)
```
file, err := os.Open(*optF)
if err != nil {
        log.Fatalf("couldn't open %q\n", *optF)
}
```

We import `log`.

51f ⟨*Imports, Ch. 7* 49c⟩+≡ (49a) ◁51b 52b▷
```
"log"
```

The file of regions is scanned using a `Scanner`. As with the regions read from the command line, we check each region for reasonableness.

52a ⟨*Scan file, Ch. 7* 52a⟩≡ (51d)
```
sc := bufio.NewScanner(file)
for sc.Scan() {
        x := sc.Text()
        f := strings.Fields(x)
        r := *new(region)
        s, _ := strconv.Atoi(f[0])
        e, _ := strconv.Atoi(f[1])
        r.start = s
        r.end = e
        ⟨Check region, Ch. 7 51c⟩
        regions = append(regions, r)
}
```

We import `bufio`.

52b ⟨*Imports, Ch. 7* 49c⟩+≡ (49a) ◁51f 52f▷
```
"bufio"
```

After we are done with the file of regions, we close it.

52c ⟨*Close file, Ch. 7* 52c⟩≡ (51d)
```
file.Close()
```

We now scan each input file using the function `scan`, which takes as arguments the regions and the indicator of whether or not the regions are to be joined.

52d ⟨*Parse input, Ch. 7* 52d⟩≡ (49b)
```
files := flag.Args()
clio.ParseFiles(files, scan, regions, *optJ)
```

In function `scan`, we first retrieve the arguments we just passed, then scan the sequences.

52e ⟨*Functions, Ch. 7* 52e⟩≡ (49a)
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 7 52g⟩
        ⟨Scan sequences, Ch. 7 53a⟩
}
```

We import `io`.

52f ⟨*Imports, Ch. 7* 49c⟩+≡ (49a) ◁52b 53b▷
```
"io"
```

The arguments are retrieved by type assertion.

52g ⟨*Retrieve arguments, Ch. 7* 52g⟩≡ (52e)
```
regions := args[0].([]region)
optJ := args[1].(bool)
```

The sequences are scanned using a dedicated `Scanner`, and the cut regions are either printed separately or joined together.

53a ⟨*Scan sequences, Ch. 7* 53a⟩≡ (52e)
```
sc := fasta.NewScanner(r)
for sc.ScanSequence() {
        seq := sc.Sequence()
        if optJ && len(regions) > 1 {
                ⟨Print joined regions, Ch. 7 53c⟩
        } else {
                ⟨Print separate regions, Ch. 7 54a⟩
        }
}
```

We import `fasta`.

53b ⟨*Imports, Ch. 7* 49c⟩+≡ (49a) ◁52f
```
"github.com/evolbioinf/fasta"
```

When joining, we store the regions in a byte slice and prepare a header that lists their coordinates.

53c ⟨*Print joined regions, Ch. 7* 53c⟩≡ (53a)
```
var d []byte
h := seq.Header() + " join("
⟨Construct joined header and sequence, Ch. 7 53d⟩
h = h + ")"
ns := fasta.NewSequence(h, d)
fmt.Println(ns)
```

The joined header and sequence is constructed in one pass over the slice of regions. But before we apply a region to a sequence, we make sure it doesn't overrun the sequence.

53d ⟨*Construct joined header and sequence, Ch. 7* 53d⟩≡ (53c)
```
for i, r := range regions {
        ⟨Ensure region doesn't overrun sequence, Ch. 7 53e⟩
        s := r.start
        e := r.end
        if i > 0 {
                h = h + ","
        }
        h = h + strconv.Itoa(s) + ".." + strconv.Itoa(e)
        d = append(d, seq.Data()[s-1:e]...)
}
```

If the region overruns the sequence, we curtail it to the sequence end and warn the user.

53e ⟨*Ensure region doesn't overrun sequence, Ch. 7* 53e⟩≡ (53d 54a)
```
sl := len(seq.Data())
if r.end > sl {
        fmt.Fprintf(os.Stderr, "curtailing (%d, %d) to (%d, %d)\n",
                r.start, r.end, r.start, sl)
        r.end = sl
}
```

Printing the cut-out regions separately is simpler, we just iterate over the slice of regions. Again, we ensure there's no overrun.

54a ⟨*Print separate regions, Ch. 7* 54a⟩≡ (53a)
```
for _, r := range regions {
        ⟨Ensure region doesn't overrun sequence, Ch. 7 53e⟩
        s := r.start
        e := r.end
        h := seq.Header() + " "
        h = h + strconv.Itoa(s) + ".." + strconv.Itoa(e)
        ns := fasta.NewSequence(h, seq.Data()[s-1:e])
        fmt.Println(ns)
}
```

We're done with `cutSeq`, the rest is testing.

Testing

The testing outline has hooks for imports and the actual testing function.

54b ⟨*cutSeq_test.go* 54b⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 7 55a⟩
)

func TestCutSeq(t *testing.T) {
        ⟨Testing, Ch. 7 54c⟩
}
```

We first cut positions 10-20 from `test.fasta` and compare what we get with what we want, contained in `res1.fasta`.

54c ⟨*Testing, Ch. 7* 54c⟩≡ (54b) 55b▷
```
cmd := exec.Command("./cutSeq", "-r", "10-20", "test.fasta")
g, err := cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err := ioutil.ReadFile("res1.fasta")
if err != nil {
        t.Errorf("couldn't open res1.fasta")
}
if !bytes.Equal(g, w) {
        t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

We import `exec`, `ioutil`, and `bytes`.

55a ⟨*Testing imports, Ch. 7* 55a⟩≡ (54b)
```
"os/exec"
"io/ioutil"
"bytes"
```

Next, we cut two regions, 10–20 and 25–50.

55b ⟨*Testing, Ch. 7* 54c⟩+≡ (54b) ◁54c 55c▷
```
cmd = exec.Command("./cutSeq", "-r", "10-20,25-50", "test.fasta")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res2.fasta")
if err != nil {
          t.Errorf("couldn't open res2.fasta")
}
if !bytes.Equal(g, w) {
          t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

And join them.

55c ⟨*Testing, Ch. 7* 54c⟩+≡ (54b) ◁55b 55d▷
```
cmd = exec.Command("./cutSeq", "-j", "-r", "10-20,25-50", "test.fasta")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res3.fasta")
if err != nil {
          t.Errorf("couldn't open res3.fasta")
}
if !bytes.Equal(g, w) {
          t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

Now we repeat these three tests with coordinates read from file. First, a single pair of coordinates.

55d ⟨*Testing, Ch. 7* 54c⟩+≡ (54b) ◁55c 56a▷
```
cmd = exec.Command("./cutSeq", "-f", "coord1.txt", "test.fasta")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res1.fasta")
if err != nil {
          t.Errorf("couldn't open res1.fasta")
}
if !bytes.Equal(g, w) {
          t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

Followed by two regions printed separately.

56a ⟨*Testing, Ch. 7* 54c⟩+≡ (54b) ◁55d 56b▷

```
cmd = exec.Command("./cutSeq", "-f", "coord2.txt", "test.fasta")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res2.fasta")
if err != nil {
          t.Errorf("couldn't open res2.fasta")
}
if !bytes.Equal(g, w) {
          t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

And finally, the two regions joined.

56b ⟨*Testing, Ch. 7* 54c⟩+≡ (54b) ◁56a

```
cmd = exec.Command("./cutSeq", "-j", "-f", "coord2.txt", "test.fasta")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res3.fasta")
if err != nil {
          t.Errorf("couldn't open res3.fasta")
}
if !bytes.Equal(g, w) {
          t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

Chapter 8

Program `dnaDist`: Distances between DNA Sequences

Introduction

Given a stet of aligned DNA sequences, `dnaDist` computes their pairwise distances. Perhaps the simplest distance measure is the raw mismatch count, and `dnaDist` implements that, as well as the number of mismatches per base. The mismatches per base are allow computation of two classical distance measures, known by their inventors' names, Jukes-Cantor [20] and Kimura [21].

When computing the Jukes-Cantor distance, all mutations are treated the same, regardless of the base changed. Let π be the number of mismatches per site between a pair of sequences. Since positions can mutate more than once, the number of mismatches may be smaller than the number of mutations, or substitutions, that took place in the past. The Jukes-Cantor distance corrects for this.

$$J = -\frac{3}{4}\log\left(1 - \frac{4\pi}{3}\right). \tag{8.1}$$

As shown in Figure 8.1, $J \approx \pi$ for small values of π. But as π increases, J grows much quicker, until at saturation, when $\pi = 3/4$, J becomes infinity.

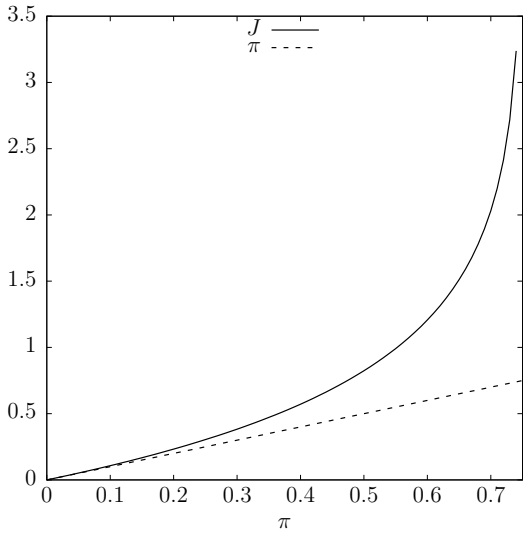


Figure 8.1: The number of substitutions per site, J, as a function of the number of mismatches per site, π.

For the Kimura distance, purines, `A` and `G`, are distinguished from pyrimidines, `C` and `T`. Mutations within the chemical class are called transitions, between the classes transversions. Let α be the number of transitions per site between two sequences, and β the number of transversions. Their sum is the number of mismatches, $\pi = \alpha + \beta$. The Kimura distance is then

$$K = -\frac{1}{2}\log\left((1 - 2\alpha - \beta)\sqrt{1 - 2\beta}\right). \tag{8.2}$$

When calculating pairwise mismatches, we ignore comparisons between pairs of gaps or between a gap and a residue.

Distance matrices are usually summarized as phylogenies, which routinely come with support values attached to internal nodes that quantify the reliability of the group of organisms in the subtree rooted on that node. A popular method for calculating these support values is the bootstrap [9]. The bootstrap is a widely used technique in statistics for simulating repeated sampling [8]. In the case of `dnaDist`, the underlying multiple sequence alignment is resampled. If this consists of an $m \times n$ table of residues, where m is the number of taxa and n the alignment length, a bootstrap sample is generated by randomly drawing n columns. This is done *with replacement*, which means that some columns are drawn repeatedly, others not at all. Given such a bootstrap sample, distances are computed and output, and this bootstrapping is repeated, say 10^4 times.

Implementation

The outline of `dnaDist` contains hooks for imports, types, functions, and the logic of the main function.

59a ⟨*dnaDist.go* 59a⟩≡
```
package main

import (
        ⟨Imports, Ch. 8 59c⟩
)
⟨Types, Ch. 8 63d⟩
⟨Functions, Ch. 8 61d⟩
func main() {
        ⟨Main function, Ch. 8 59b⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options and parse them, and parse the input files.

59b ⟨*Main function, Ch. 8* 59b⟩≡ (59a)
```
util.PrepLog("dnaDist")
⟨Set usage, Ch. 8 59d⟩
⟨Declare options, Ch. 8 60a⟩
⟨Parse options, Ch. 8 60c⟩
⟨Parse input files, Ch. 8 61c⟩
```

We import `util`.

59c ⟨*Imports, Ch. 8* 59c⟩≡ (59a) 59e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of a message, an explanation of the program's purpose, and an example command.

59d ⟨*Set usage, Ch. 8* 59d⟩≡ (59b)
```
u := "dnaDist [-h] [options] [file(s)]"
p := "Calculate distances between DNA sequences."
e := "dnaDist foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

59e ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁ 59c 60b ▷
```
"github.com/evolbioinf/clio"
```

We declare six options: `-v` to print the program version, `-r` to give the raw mismatch count, `-u` for the uncorrected distances, `-k` to compute Kimura instead of Jukes-Cantor distances, and `-b` to specify the number of bootstrap replicates, by default none. Bootstrapping requires random numbers, and their generator can be seeded via `-s` to generate exact repeats.

60a ⟨*Declare options, Ch. 8* 60a⟩≡ (59b)
```
var optV = flag.Bool("v", false, "version")
var optR = flag.Bool("r", false, "raw mismatches")
var optU = flag.Bool("u", false, "uncorrected mismatches")
var optK = flag.Bool("k", false, "Kimura distances (default: Jukes-Cantor)")
var optB = flag.Int("b", 0, "number of bootstrap replicates")
var optS = flag.Int("s", 0, "seed for random number generator " +
          "(default: internal)")
```

We import `flag`.

60b ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁59e 60g▷
```
"flag"
```

When parsing options, `-v` and `-b` require action at this point.

60c ⟨*Parse options, Ch. 8* 60c⟩≡ (59b)
```
flag.Parse()
⟨Respond to -v, Ch. 8 60d⟩
⟨Respond to -b, Ch. 8 60e⟩
```

If the user requested version printing, we call a dedicated function that takes the program name as argument.

60d ⟨*Respond to -v, Ch. 8* 60d⟩≡ (60c)
```
if *optV {
        util.PrintInfo("dnaDist")
}
```

As to the number of bootstrap replicates, two cases require attention: Less than zero, where the user made a mistake, and more than zero, where the user requested bootstrapping.

60e ⟨*Respond to -b, Ch. 8* 60e⟩≡ (60c)
```
⟨Less than zero replicates, Ch. 8 60f⟩
⟨More than zero replicates, Ch. 8 61a⟩
```

If the user requested a negative number of bootstrap replicates, this is set to zero and the user warned.

60f ⟨*Less than zero replicates, Ch. 8* 60f⟩≡ (60e)
```
if *optB < 0 {
        fmt.Fprintf(os.Stderr, "resetting %d bootstrap " +
                "replicates to zero", *optB)
        *optB = 0
}
```

We import `fmt` and `os`.

60g ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁60b 61b▷
```
"fmt"
"os"
```

If the user requested at least one bootstrap replicate, the random number generator is seeded. If no seed is provided, it is generated internally.

61a ⟨*More than zero replicates, Ch. 8* 61a⟩≡ (60e)

```
var ran *rand.Rand
if *optB > 0 {
        if *optS != 0 {
                ran = rand.New(rand.NewSource(int64(*optS)))
        } else {
                t := time.Now().UnixNano()
                ran = rand.New(rand.NewSource(t))
        }
}
```

We import `rand` and `time`.

61b ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁60g 61e▷

```
"math/rand"
"time"
```

We parse the input files using the function `parseFiles`. It takes as input the names of the input files, a function applied to each of these files, and the arguments of that function. These arguments are the values of `-b`, `-r`, `-u`, `-k`, the random number generator, and a matrix for looking up transitions.

61c ⟨*Parse input files, Ch. 8* 61c⟩≡ (59b)

```
files := flag.Args()
ts := util.NewTransitionTab()
clio.ParseFiles(files, scan, *optB, *optR, *optU, *optK, ran, ts)
```

In the function `scan`, we retrieve the options just passed, read the input file, convert it to a multiple sequence alignment, and calculate the pairwise distances from it.

61d ⟨*Functions, Ch. 8* 61d⟩≡ (59a) 64c▷

```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve options, Ch. 8 61f⟩
        ⟨Read input file, Ch. 8 62a⟩
        ⟨Construct multiple sequence alignment, Ch. 8 62d⟩
        ⟨Calculate distances, Ch. 8 63b⟩
}
```

We import `io`.

61e ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁61b 62b▷

```
"io"
```

The values of six options were passed, the number of bootstrap replicates (`-b`), raw distances (`-r`), uncorrected mismatches (`-u`), Kimura distances (`-k`), the random number generator, and the transition table. We retrieve them by type assertion.

61f ⟨*Retrieve options, Ch. 8* 61f⟩≡ (61d)

```
optB := args[0].(int)
optR := args[1].(bool)
optU := args[2].(bool)
optK := args[3].(bool)
ran  := args[4].(*rand.Rand)
ts   := args[5].(util.TransitionTab)
```

When reading the input file, the sequences are stored in a slice of sequences. To make sure the sequences are in fact aligned, we check their lengths.

62a ⟨*Read input file, Ch. 8* 62a⟩≡ (61d)
```
sc := fasta.NewScanner(r)
var sa []*fasta.Sequence
for sc.ScanSequence() {
        sa = append(sa, sc.Sequence())
        ⟨Check sequence lengths, Ch. 8 62c⟩
}
```

We import `fasta`.

62b ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁61e 62f▷
```
"github.com/evolbioinf/fasta"
```

When we find a variation in the sequence length, we are not dealing with an alignment and abort.

62c ⟨*Check sequence lengths, Ch. 8* 62c⟩≡ (62a)
```
if len(sa) > 1 {
        i := len(sa) - 1
        if len(sa[i].Data()) != len(sa[i-1].Data()) {
                fmt.Fprintf(os.Stderr, "this doesn't look " +
                        "like an alignment\n")
                os.Exit(-1)
        }
}
```

A multiple sequence alignment is a two-dimensional table of bytes. When computing distances, all pairwise comparisons between the residues in a column need to be made. This is only necessary for polymorphic columns and we can potentially save a lot of work by ignoring monomorphic columns. Hence we augment the multiple sequence alignment by a table of polymorphic sites.

62d ⟨*Construct multiple sequence alignment, Ch. 8* 62d⟩≡ (61d)
```
⟨Construct byte table, Ch. 8 62e⟩
⟨Construct polymorphism table, Ch. 8 63a⟩
```

We construct an $m \times n$ table of residues, which are set to upper case.

62e ⟨*Construct byte table, Ch. 8* 62e⟩≡ (62d)
```
m := len(sa)
n := len(sa[0].Data())
msa := make([][]byte, m)
for i, s := range sa {
        msa[i] = bytes.ToUpper(s.Data())
}
```

We import `bytes`.

62f ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁62b 65d▷
```
"bytes"
```

The polymorphism table consists of booleans for each column; if *true*, the position is polymorphic.

63a ⟨*Construct polymorphism table, Ch. 8* 63a⟩≡ (62d)
```
pol := make([]bool, n)
gap := byte('-')
for j := 0; j < n; j++ {
        for i := 0; i < m; i++ {
                if msa[i][j] != gap && msa[i][j] != msa[0][j] {
                        pol[j] = true
                        break
                }
        }
}
```

When calculating distances, this is done either with or without bootstrapping. Wit bootstrapping, we sample with replacement columns of the residue table. We do this with an indicator table containing the positions from which the distances are computed. The distance matrix is constructed at the beginning and reused throughout the bootstrap.

63b ⟨*Calculate distances, Ch. 8* 63b⟩≡ (61d)
```
ind := make([]int, n)
⟨Make distance matrix, Ch. 8 63c⟩
if optB > 0 {
        ⟨With bootstrap, Ch. 8 64a⟩
} else {
        ⟨Without bootstrap, Ch. 8 67c⟩
}
```

The distance matrix is an $n \times n$ table of cells.

63c ⟨*Make distance matrix, Ch. 8* 63c⟩≡ (63b)
```
dm := make([][]cell, m)
for i := 0; i < m; i++ {
        dm[i] = make([]cell, m)
}
```

Each cell consists of two integers, the raw counts that go into the computation of α, β, and a float holding the final distance, d.

63d ⟨*Types, Ch. 8* 63d⟩≡ (59a)
```
type cell struct {
        a, b int
        d float64
}
```

With bootstrap, we resample the column indexes, then compute the distance, and finally print them. Since distance computation and printing is the same without bootstrap, these steps are delegated to functions.

64a ⟨*With bootstrap, Ch. 8* 64a⟩≡ (63b)
```
for i := 0; i < optB; i++ {
        ⟨Bootstrap indexes, Ch. 8 64b⟩
        distMat(dm, msa, pol, ind, optR, optU, optK, ts)
        printDist(dm, sa)
        ⟨Reset distance matrix, Ch. 8 67b⟩
}
```

Bootstrapping the indexes consists of drawing n random numbers between zero and $n-1$.

64b ⟨*Bootstrap indexes, Ch. 8* 64b⟩≡ (64a)
```
for j := 0; j < n; j++ {
        ind[j] = ran.Intn(n)
}
```

The function `distMat` fills the distance matrix by first counting the transitions and transversions and then entering the actual distances.

64c ⟨*Functions, Ch. 8* 61d⟩+≡ (59a) ◁61d 66a▷
```
func distMat(dm [][]cell, msa [][]byte, pol []bool,
        ind []int, optR, optU, optK bool,
        ts util.TransitionTab) {
        m := len(msa)
        n := len(msa[0])
        ⟨Count transitions and transversions, Ch. 8 64d⟩
        ⟨Enter distances, Ch. 8 65b⟩
}
```

The count of transitions and transversions is restricted to the polymorphic columns.

64d ⟨*Count transitions and transversions, Ch. 8* 64d⟩≡ (64c)
```
for i := 0; i < n; i++ {
        if pol[ind[i]] {
                ⟨Analyze column, Ch. 8 65a⟩
        }
}
```

In a given column, all pairwise comparisons between residues are made.

65a ⟨*Analyze column, Ch. 8* 65a⟩≡ (64d)
```
for j := 0; j < m-1; j++ {
        c1 := msa[j][i]
        for k := j + 1; k < m; k++ {
                c2 := msa[k][i]
                if c1 != c2 {
                        if ts.IsTransition(c1, c2) {
                                dm[j][k].a++
                        } else {
                                dm[j][k].b++
                        }
                }
        }
}
```

For each distance, we compute α and β, choose its type, and mirror the result in the distance matrix.

65b ⟨*Enter distances, Ch. 8* 65b⟩≡ (64c)
```
for i := 0; i < m-1; i++ {
        for j := i+1; j < m; j++ {
                a := float64(dm[i][j].a) / float64(n)
                b := float64(dm[i][j].b) / float64(n)
                ⟨Choose distance type, Ch. 8 65c⟩
                dm[j][i].d = dm[i][j].d
        }
}
```

Kimura distances are given by equation (8.2), Jukes-Cantor distances by equation (8.1).

65c ⟨*Choose distance type, Ch. 8* 65c⟩≡ (65b)
```
if optR {
        dm[i][j].d = float64(dm[i][j].a + dm[i][j].b)
} else if optU {
        dm[i][j].d = a + b
} else if optK {
        dm[i][j].d = -math.Log((1-2*a-b) * math.Sqrt(1-2*b)) / 2
} else {
        p := a + b
        dm[i][j].d = -0.75 * math.Log(1 - 4./3. * p)
}
```

We import `math`.

65d ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁62f 66d▷
```
"math"
```

We print the distance matrix in the format used by the phylogeny package PHYLIP [10], a de facto standard in the field. A PHYLIP distance matrix consists of the sample size in the first line, followed by the matrix consisting of the taxon name followed by the distances. We use a `tabprinter` to line up columns.

66a ⟨*Functions, Ch. 8* 61d⟩+≡ (59a) ◁64c
```
func printDist(dm [][]cell, sa []*fasta.Sequence) {
        ⟨Print sample size, Ch. 8 66b⟩
        ⟨Construct tabwriter, Ch. 8 66c⟩
        ⟨Print distances, Ch. 8 66e⟩
}
```

The sample size is the length of the distance matrix.

66b ⟨*Print sample size, Ch. 8* 66b⟩≡ (66a)
```
n := len(dm)
fmt.Printf("%d\n", n)
```

The `tabwriter` writes to a byte-buffer. We initialize the writer to a minimal cell width of 1, the width of the tab characters to zero, and add a single blank for padding.

66c ⟨*Construct tabwriter, Ch. 8* 66c⟩≡ (66a)
```
var buf []byte
buffer := bytes.NewBuffer(buf)
w := new(tabwriter.Writer)
w.Init(buffer, 1, 0, 1, ' ', 0)
```

We import `tabwriter`.

66d ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁65d 66f▷
```
"text/tabwriter"
```

The distances are printed in a loop over all entries in the matrix. In distance matrices, the taxon name is separated by blanks from the matrix entries. So we truncate the taxon name at the first blank.

As to the matrix entries themselves, it turns out that a zero result can be *negative* zero, which looks awkward in the printout so we check for it. Then we flush the `tabwriter` and print the buffer.

66e ⟨*Print distances, Ch. 8* 66e⟩≡ (66a)
```
for i := 0; i < n; i++ {
        name := strings.Fields(sa[i].Header())[0]
        fmt.Fprintf(w, "%s\t", name)
        for j := 0; j < n; j++ {
                ⟨Check for negative zeros, Ch. 8 67a⟩
        }
        fmt.Fprintf(w, "\n")
}
w.Flush()
fmt.Printf("%s", buffer)
```

We import `strings`.

66f ⟨*Imports, Ch. 8* 59c⟩+≡ (59a) ◁66d
```
"strings"
```

Negative zeros are discovered using the `Signbit` library function. If we find a negative zero, we print a positive zero instead.

67a ⟨*Check for negative zeros, Ch. 8* 67a⟩≡ (66e)
```
if math.Signbit(dm[i][j].d) {
        fmt.Fprintf(w, "%.6g\t", 0.0)
} else {
        fmt.Fprintf(w, "%.6g\t", dm[i][j].d)
}
```

After printing, the distance matrix are reset.

67b ⟨*Reset distance matrix, Ch. 8* 67b⟩≡ (64a)
```
for i := 0; i < m-1; i++ {
        for j := i + 1; j < m; j++ {
                dm[i][j].a = 0
                dm[i][j].b = 0
        }
}
```

If no bootstrapping is requested, the index array is just filled $0, 1, ..., n-1$.

67c ⟨*Without bootstrap, Ch. 8* 67c⟩≡ (63b)
```
for i, _ := range ind {
        ind[i] = i
}
distMat(dm, msa, pol, ind, optR, optU, optK, ts)
printDist(dm, sa)
```

Our program is finished, let's test.

Testing

The outline of the testing program contains hooks for imports and the code for driving the tests.

67d ⟨*dnaDist_test.go* 67d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 8 68b⟩
)
func TestDnaDist(t *testing.T) {
        ⟨Testing, Ch. 8 68a⟩
}
```

We construct a table of test commands, a table of files with results, and then run the commands.

68a ⟨*Testing, Ch. 8* 68a⟩≡ (67d)
```
tests := make([]*exec.Cmd, 0)
⟨Construct test commands, Ch. 8 68c⟩
⟨Construct result files, Ch. 8 68d⟩
for i, c := range tests {
        ⟨Conduct test, Ch. 8 69a⟩
}
```

We import `exec`.

68b ⟨*Testing imports, Ch. 8* 68b⟩≡ (67d) 68e ▷
```
"os/exec"
```

Testing is done on two data sets, the artificial `test.fa` and a small sample of primate mitochondrial DNA, `pr.fa`.

68c ⟨*Construct test commands, Ch. 8* 68c⟩≡ (68a)
```
c := exec.Command("./dnaDist", "test.fa")
tests = append(tests, c)
c = exec.Command("./dnaDist", "-k", "test.fa")
tests = append(tests, c)
c = exec.Command("./dnaDist", "pr.fa")
tests = append(tests, c)
c = exec.Command("./dnaDist", "-k", "pr.fa")
tests = append(tests, c)
c = exec.Command("./dnaDist", "-b", "5", "-s", "3", "pr.fa")
tests = append(tests, c)
```

The results we want are contained in five files.

68d ⟨*Construct result files, Ch. 8* 68d⟩≡ (68a)
```
results := make([]string, len(tests))
for i, _ := range tests {
        results[i] = "r" + strconv.Itoa(i+1) + ".txt"
}
```

We import `strconv`.

68e ⟨*Testing imports, Ch. 8* 68b⟩+≡ (67d) ◁ 68b 69b ▷
```
"strconv"
```

Now we run the tests and compare what we get with what we want.

69a ⟨*Conduct test, Ch. 8* 69a⟩≡ (68a)
```
get, err := c.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", c)
}
want, err := ioutil.ReadFile(results[i])
if err != nil {
        t.Errorf("couldn't open %q\n", results[i])
}
if !bytes.Equal(get, want) {
        t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `ioutil` and bytes.

69b ⟨*Testing imports, Ch. 8* 68b⟩+≡ (67d) ◁68e
```
"io/ioutil"
"bytes"
```

Chapter 9

Program `drag`: Draw Genealogies

Introduction

Everybody has two parents, who have two parents each, our four grandparents, who have two parents each, our eight great-grandparents, and so on. In other words, the number of ancestors doubles in every generation, which means we quickly share our ancestors with very many people, in case you were ever tempted to brag about them. In fact, in a population of n individuals it only takes approximately $\log_2(n)$ generations into the past until the first universal ancestor appears, individuals everyone living today has in their pedigree [26]. Universal non-ancestors, that is, individuals without any descendants in the present, often appear even earlier.

For example, in Figure 9.1A we have males—the boxes, if you like—and females—the ellipses—in each generation. Males and females mate and leave descendants. The dots inside each individual are their diploid genome. The edges connecting the dots show the lines of descent of the maternal and paternal genomes. Notice that even though we have two sexes in our model, there is no recombination.

The first non-ancestor marked in blue appears one generation in the past, in g_9 or b_1. Its genes are not touched by any of the criss-crossing lines of descent between the genes. One generation further back we have the first universal ancestor in red. By generation b_7, the green partial ancestors have disappeared. On average this happens after $\approx 1.77 \log_2(n)$ generations [26]. Partial ancestors cannot be recreated, so from this point back in time there are only universal ancestors and no-ancestors, with the universal ancestors outnumbering the no-ancestors [26].

Rather than tracing the ancestors of all genes, we can restrict our attention to the genes of only one or a few individuals. For example, in Figure 9.1B only the ancestors of the genes of individual 5 are traced, everything else is the same as in Figure 9.1A. Notice that as soon as the partial ancestors have disappeared, it makes no difference any more whether we are tracing the descent of all genes or only of those of a single individual.

Instead of thinking about individuals, we can concentrate exclusively on the genes of some or all individuals. In Figure 9.1C we did that for individual 5, which is a subgraph of Figure 9.1B with the ancestry of genes without descendants in the present removed.

The program `drag` draws genealogies like those in Figure 9.1. `drag` writes genealogies in the dot notation of the graphviz package. A genealogy can be rendered with `neato`, which is also part of graphviz.

Implementation

The outline of `drag` has hooks for imports, types, and the logic of the main function.

71 ⟨*drag.go* 71⟩≡

```
package main

import (
        ⟨Imports, Ch. 9 73b⟩
)
⟨Types, Ch. 9 75c⟩
func main() {
        ⟨Main function, Ch. 9 73a⟩
}
```

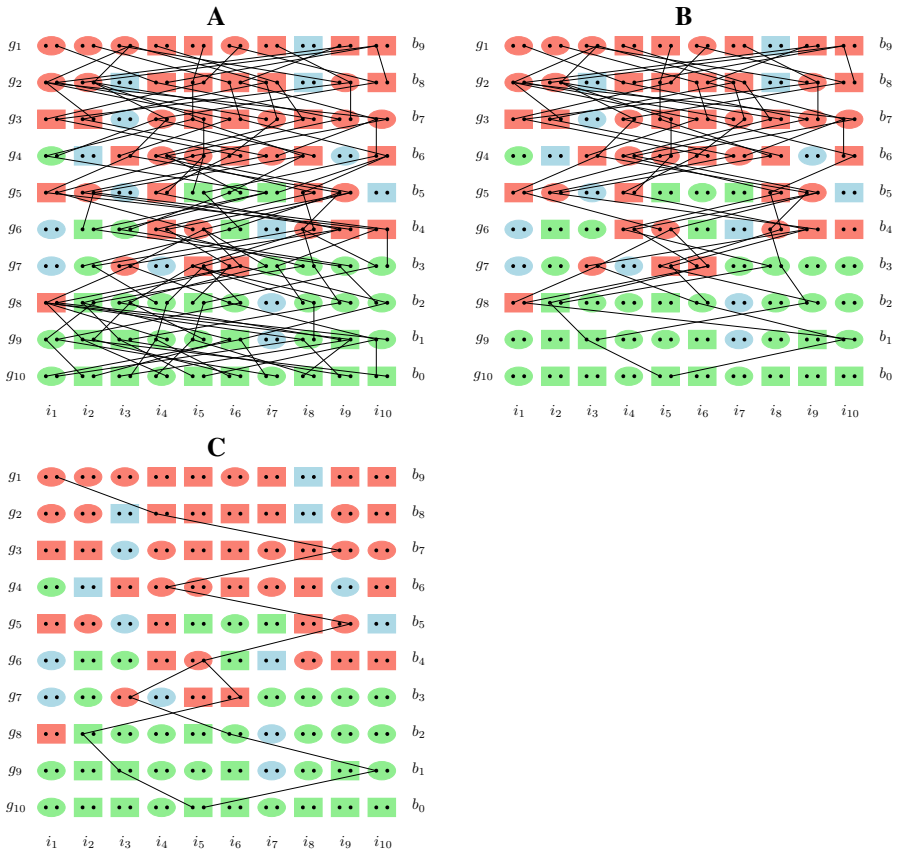


Figure 9.1: Genealogy with genetic ancestry traced for all individuals (**A**), just one individual (**B**), or just the genes of that individual (**C**).

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, construct the genealogy, and print it.

73a ⟨*Main function, Ch. 9* 73a⟩≡ (71)
```
util.PrepLog("drag")
⟨Set usage, Ch. 9 73c⟩
⟨Declare options, Ch. 9 73e⟩
⟨Parse options, Ch. 9 74a⟩
⟨Construct genealogy, Ch. 9 75d⟩
⟨Print genealogy, Ch. 9 80a⟩
```

We import `util`.

73b ⟨*Imports, Ch. 9* 73b⟩≡ (71) 73d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `drag`, and an example command.

73c ⟨*Set usage, Ch. 9* 73c⟩≡ (73a)
```
u := "drag [-h] [option]..."
p := "Draw genealogy of diploid individuals."
e := "drag -t 4,6 | neato -T x11"
clio.Usage(u, p, e)
```

73d ⟨*Imports, Ch. 9* 73b⟩+≡ (71) ◁73b 73f▷
```
"github.com/evolbioinf/clio"
```

Apart from the version, we declare seven options: number of generations (`-g`), number of individuals (`-n`), trace the ancestry of a set of individuals (`-t`), reduce the ancestry to genes (`-G`), set a scaling factor for the plot (`-f`), print just ancestor statistics instead of the graph (`-a`), and set the seed for the random number generator (`-s`).

73e ⟨*Declare options, Ch. 9* 73e⟩≡ (73a)
```
var optV = flag.Bool("v", false, "version")
var optG = flag.Int("g", 10, "number of generations")
var optN = flag.Int("n", 10, "number of individuals")
var optT = flag.String("t", "", "trace genealogy of " +
          "individuals, e.g. 3,4,5; -1 for all")
var optGG = flag.Bool("G", false, "trace genes")
var optF = flag.Float64("f", 1.0, "scale factor for plot")
var optA = flag.Bool("a", false, "ancestor statistics")
var optS = flag.Int64("s", 0, "seed for random number generator")
```

We import `flag`.

73f ⟨*Imports, Ch. 9* 73b⟩+≡ (71) ◁73d 74e▷
```
"flag"
```

We parse the options and first respond to `-v` as this stops the program. Then we extract the individuals to be traced and seed the random number generator.

74a ⟨*Parse options, Ch. 9* 74a⟩≡ (73a)
```
flag.Parse()
if *optV {
        util.PrintInfo("drag")
}
var tr []int
if *optT != "" {
        ⟨Extract individuals to be traced, Ch. 9 74b⟩
}
⟨Seed random number generator, Ch. 9 75a⟩
```

We either trace all individuals or a list of individuals we still need to extract.

74b ⟨*Extract individuals to be traced, Ch. 9* 74b⟩≡ (74a)
```
s := *optT
if s[0] == '-' {
        ⟨Add all individuals, Ch. 9 74c⟩
} else {
        ⟨Extract individuals, Ch. 9 74d⟩
}
```

To add all individuals we store their names, $0, 1..., n-1$. These names double as the individuals' indexes in the present generation.

74c ⟨*Add all individuals, Ch. 9* 74c⟩≡ (74b)
```
for i := 0; i < *optN; i++ {
        tr = append(tr, i)
}
```

The list of individuals is comma-separated. The user enters individuals as one-based numbers, but internally we use 0-based indexes. So we subtract 1 from the numbers provided by the user.

74d ⟨*Extract individuals, Ch. 9* 74d⟩≡ (74b)
```
fields := strings.Split(*optT, ",")
for _, field := range fields {
        i, err := strconv.Atoi(field)
        if err != nil {
                log.Fatalf("can't convert %q", field)
        }
        tr = append(tr, i-1)
}
```

We import `strings`, `strconv`, and `log`.

74e ⟨*Imports, Ch. 9* 73b⟩+≡ (71) ◁73f 75b▷
```
"strings"
"strconv"
"log"
```

We seed the random number generator either from the user input or from the current time.

75a ⟨*Seed random number generator, Ch. 9* 75a⟩≡ (74a)
```
seed := *optS
if seed == 0 {
        seed = time.Now().UnixNano()
}
rand.Seed(seed)
```

75b ⟨*Imports, Ch. 9* 73b⟩+≡ (71) ◁74e 76c▷
```
"time"
"math/rand"
```

A genealogy consists of individuals. An individual has two ancestors, a number of descendants in the present, is either male or female, holds its own name and the names of the ancestors of its two genes, one or both of which genes may be ancestral. An the individual may lie on a path to be drawn and may be a universal ancestor or a non-ancestor.

75c ⟨*Types, Ch. 9* 75c⟩≡ (71)
```
type indiv struct {
        a [2]*indiv
        p int
        isMale bool
        n string
        ag [2]string
        g [2]bool
        isOnPath bool
        isUa, isNonUa bool
}
```

The genealogy is an $m \times n$ population of individuals, where m is the number of generations and n the population size. We construct it, pick ancestors, and trace the individuals. If desired, we also trace individual genes.

75d ⟨*Construct genealogy, Ch. 9* 75d⟩≡ (73a)
```
m := *optG
n := *optN
pop := make([][]*indiv, m)
⟨Construct matrix, Ch. 9 76a⟩
⟨Pick ancestors, Ch. 9 77a⟩
⟨Count the descendants, Ch. 9 77c⟩
⟨Trace individuals, Ch. 9 78b⟩
if *optGG {
        ⟨Trace genes, Ch. 9 78e⟩
}
```

We iterate over the generations and allocate all individuals. We pick the sex of an individual randomly and count the number of females per generation to ensure both sexes are present. Initially all individuals are female.

76a ⟨*Construct matrix, Ch. 9* 76a⟩≡ (75d)

```
for i := 0; i < m; i++ {
        pop[i] = make([]*indiv, n)
        nf := n
        for j := 0; j < n; j++ {
                ⟨Construct individual, Ch. 9 76b⟩
        }
        ⟨Ensure both sexes present, Ch. 9 76d⟩
}
```

An individual has a name, which we construct from its row and column indexes. It also has a sex, which we pick randomly and then count the females.

76b ⟨*Construct individual, Ch. 9* 76b⟩≡ (76a)

```
pop[i][j] = new(indiv)
pop[i][j].n = fmt.Sprintf("i_%d_%d", i, j)
if rand.Float64() < 0.5 {
        pop[i][j].isMale = true
        nf--
}
```

We import `fmt`.

76c ⟨*Imports, Ch. 9* 73b⟩+≡ (71) ◁75b

```
"fmt"
```

If the number of females is equal to zero, we switch a random individual to female, if it is equal to n, we switch a random individual to male.

76d ⟨*Ensure both sexes present, Ch. 9* 76d⟩≡ (76a)

```
if nf == 0 {
        r := rand.Intn(n)
        pop[i][r].isMale = false
} else if nf == n {
        r := rand.Intn(n)
        pop[i][r].isMale = true
}
```

We pick the two ancestors of each individual, except for the individuals in the generation furthest back in time, generation 0. The ancestors must be male and female. From the ancestors we pick the ancestral genes.

77a ⟨*Pick ancestors, Ch. 9* 77a⟩≡ (75d)
```
for i := m -1; i > 0; i-- {
        for j := 0; j < n; j++ {
                pop[i][j].a[0] = pop[i-1][rand.Intn(n)]
                pop[i][j].a[1] = pop[i-1][rand.Intn(n)]
                for pop[i][j].a[0].isMale == pop[i][j].a[1].isMale {
                        pop[i][j].a[1] = pop[i-1][rand.Intn(n)]
                }
                ⟨Pick ancestral genes, Ch. 9 77b⟩
        }
}
```

The name of an ancestral gene is the name of the individual followed by `_0` or `_1`, depending on which gene we picked.

77b ⟨*Pick ancestral genes, Ch. 9* 77b⟩≡ (77a)
```
for k := 0; k < 2; k++ {
        r := rand.Intn(2)
        name := fmt.Sprintf("%s_%d", pop[i][j].a[k].n, r)
        pop[i][j].ag[k] = name
}
```

We count the descendants. To begin with, we initialize the number of descendants in the present, i. e. in generation $m - 1$, to one. Then we iterate over the individuals. For each one we add the number of its descendants to that of its ancestor. Then we determine ancestor status of all individuals.

77c ⟨*Count the descendants, Ch. 9* 77c⟩≡ (75d)
```
for i := 0; i < n; i++ {
        pop[m-1][i].p = 1
}
for i := m-1; i > 0; i-- {
        for j := 0; j < n; j++ {
                ⟨Add descendants to ancestors, Ch. 9 77d⟩
        }
}
⟨Determine ancestor status, Ch. 9 78a⟩
```

The number of ancestors grows exponentially and can thus quickly overflow. So we only calculate it if it hasn't reached the population size yet.

77d ⟨*Add descendants to ancestors, Ch. 9* 77d⟩≡ (77c)
```
for k := 0; k < 2; k++ {
        if pop[i][j].a[k].p < n {
                pop[i][j].a[k].p += pop[i][j].p
        }
}
```

Depending on its number of descendants in the present, an individual can be universal ancestor or a non-ancestor, or neither.

78a ⟨*Determine ancestor status, Ch. 9* 78a⟩≡ (77c)

```
for i := 0; i < m; i++ {
        for j := 0; j < n; j++ {
                if pop[i][j].p >= n {
                        pop[i][j].isUa = true
                } else if pop[i][j].p == 0 {
                        pop[i][j].isNonUa = true
                }
        }
}
```

78b ⟨*Trace individuals, Ch. 9* 78b⟩≡ (75d)

⟨*Initialize paths, Ch. 9* 78c⟩
⟨*Complete paths, Ch. 9* 78d⟩

We initialize the paths in the present.

78c ⟨*Initialize paths, Ch. 9* 78c⟩≡ (78b)

```
for _, t := range tr {
        pop[m-1][t].isOnPath = true
        pop[m-1][t].a[0].isOnPath = true
        pop[m-1][t].a[1].isOnPath = true
        if *optGG {
                pop[m-1][t].g[0] = true
                pop[m-1][t].g[1] = true
        }
}
```

We complete the paths from the second generation onward.

78d ⟨*Complete paths, Ch. 9* 78d⟩≡ (78b)

```
for i := m-2; i > 0; i-- {
        for j := 0; j < n; j++ {
                if pop[i][j].isOnPath {
                        pop[i][j].a[0].isOnPath = true
                        pop[i][j].a[1].isOnPath = true
                }
        }
}
```

Gene tracing is also done in two steps, initialization and completion.

78e ⟨*Trace genes, Ch. 9* 78e⟩≡ (75d)

⟨*Initialize gene paths, Ch. 9* 79a⟩
⟨*Complete gene paths, Ch. 9* 79b⟩

Gene tracing is done via the g field, with which we switch genes on. The genes of the focal individual are all switched on in the first generation. Each ancestor has two genes, of which we switch on the one already written down as the ancestral gene.

79a ⟨*Initialize gene paths, Ch. 9* 79a⟩≡ (78e)

```
for _, t := range tr {
        for i := 0; i < 2; i++ {
                pop[m-1][t].g[i] = true
                l := len(pop[m-1][t].ag[i])
                if pop[m-1][t].ag[i][l-1] == '0' {
                        pop[m-1][t].a[i].g[0] = true
                } else {
                        pop[m-1][t].a[i].g[1] = true
                }
        }
}
```

We iterate over the genes in every individual.

79b ⟨*Complete gene paths, Ch. 9* 79b⟩≡ (78e)

```
for i := m-2; i > 0; i-- {
        for j := 0; j < n; j++ {
                ⟨Iterate over genes, Ch. 9 79c⟩
        }
}
```

For every gene that's switched on, we switch on the gene in its ancestor we've already determinedc'

79c ⟨*Iterate over genes, Ch. 9* 79c⟩≡ (79b)

```
for k := 0; k < 2; k++ {
        if pop[i][j].g[k] {
                l := len(pop[i][j].ag[k])
                if pop[i][j].ag[k][l-1] == '0' {
                        pop[i][j].a[k].g[0] = true
                } else {
                        pop[i][j].a[k].g[1] = true
                }
        }
}
```

We either print the ancestor statistics or the genealogy. For the ancestor statistics, we determine them and then print them. For the genealogy, we print the individuals, the genes, and the lines of descent. These three graph elements are sandwiched by a graph header and footer.

80a ⟨*Print genealogy, Ch. 9* 80a⟩≡ (73a)

```
if *optA {
        ⟨Determine ancestor statistics, Ch. 9 80b⟩
        ⟨Print ancestor statistics, Ch. 9 81c⟩
} else {
        ⟨Print graph header, Ch. 9 81d⟩
        ⟨Print individuals, Ch. 9 82a⟩
        ⟨Print genes, Ch. 9 83b⟩
        ⟨Print lines of descent, Ch. 9 83f⟩
        ⟨Print graph footer, Ch. 9 84d⟩
}
```

There are two ancestor statistics we determine, time to the first universal ancestor and time to the disappearance of partial ancestors.

80b ⟨*Determine ancestor statistics, Ch. 9* 80b⟩≡ (80a)

```
⟨Determine time to first universal ancestor, Ch. 9 80c⟩
⟨Determine time to disappearance of partial ancestors, Ch. 9 81a⟩
```

We walk from the first generation after the present into the past until we find the first universal ancestor.

80c ⟨*Determine time to first universal ancestor, Ch. 9* 80c⟩≡ (80b)

```
tua := 0
foundUa := false
for i := m-1; i > -1; i-- {
        for j := 0; j < n; j++ {
                if pop[i][j].isUa {
                        foundUa = true
                        break
                }
        }
        if foundUa { break }
        tua++
}
```

Again, we walk from the first generation after the present into the past. This time we count the partial ancestors in each generation. Once this number drops to zero, we've found the extinction point of the partial ancestors and we break.

81a ⟨*Determine time to disappearance of partial ancestors, Ch. 9* 81a⟩≡ (80b)

```
tpa := 0
foundPa := false
for i := m-1; i > -1; i-- {
        npa := 0
        ⟨Count partial ancestors, Ch. 9 81b⟩
        if npa == 0 {
                foundPa = true
                break
        }
        tpa++
}
```

We iterate over one generation and count the partial ancestors.

81b ⟨*Count partial ancestors, Ch. 9* 81b⟩≡ (81a)

```
for j := 0; j < n; j++ {
        if !pop[i][j].isUa && !pop[i][j].isNonUa {
                npa++
        }
}
```

We print our two ancestor statistics, unless we didn't find them, which we indicate by a 0.

81c ⟨*Print ancestor statistics, Ch. 9* 81c⟩≡ (80a)

```
if !foundUa { tua = 0 }
if !foundPa { tpa = 0 }
m1 := "Generations_to_first_universal_ancestor\t%d\n"
m2 := "Generations_to_no_partial_ancestor\t%d\n"
fmt.Printf(m1, tua)
fmt.Printf(m2, tpa)
```

The graph header consists of a comment and a graph declaration. In the comment we explain the graph's origin and how to render it. Then we declare a undirected graph.

81d ⟨*Print graph header, Ch. 9* 81d⟩≡ (80a)

```
fmt.Println("# Genealogy generated with drag.")
fmt.Println("# Render with neato.")
fmt.Println("graph g {")
```

We draw the individuals of a generation in one row. The row is flanked by a forward counter of generations, g_i, and a backward counter, b_j. We also label the individuals in the present.

82a ⟨*Print individuals, Ch. 9* 82a⟩≡ (80a)
```
f := *optF
t := "%c_%d[shape=plaintext,pos=\"%.4g,%.4g!\"];"
for i := 0; i < m; i++ {
        y := float64(m-i) * f
        fmt.Printf("\t" + t, 'g', i+1, 0.0, y)
        for j := 0; j < n; j++ {
                in := pop[i][j]
                ⟨Draw individual, Ch. 9 82b⟩
        }
        fmt.Printf(t + "\n", 'b', m-1-i, float64(n+1)*f, y)
}
⟨Label individuals, Ch. 9 83a⟩
```

An individual has color, shape, and position.

82b ⟨*Draw individual, Ch. 9* 82b⟩≡ (82a)
```
var c, s string
var x float64
⟨Determine color, Ch. 9 82c⟩
⟨Determine shape, Ch. 9 82d⟩
⟨Determine position, Ch. 9 82e⟩
tmpl := "%s[label=\"\",color=%s,shape=%s," +
        "style=filled,pos=\"%.4g,%.4g!\"];"
fmt.Printf(tmpl, in.n, c, s, x, y)
```

Universal ancestors are salmon, universal non-ancestors light blue, and partial ancestors light green.

82c ⟨*Determine color, Ch. 9* 82c⟩≡ (82b)
```
c = "lightgreen"
if in.isUa {
        c = "salmon"
} else if in.isNonUa {
        c = "lightblue"
}
```

Males are boxes, females ellipses.

82d ⟨*Determine shape, Ch. 9* 82d⟩≡ (82b)
```
s = "ellipse"
if in.isMale {
        s = "box"
}
```

We already know the y coordinate, the x coordinate is just the scaled column index.

82e ⟨*Determine position, Ch. 9* 82e⟩≡ (82b)
```
x = float64(j+1) * f
```

Individuals are labeled $i_1, i_2, ...$

83a ⟨*Label individuals, Ch. 9* 83a⟩≡ (82a)
```
y := -0.0 * f
for i := 0; i < n; i++ {
        x := float64(i+1) * f
        fmt.Printf(t, 'i', i+1, x, y)
}
```

Genes are points. For each individual, we calculate its position we draw two genes nearby.

83b ⟨*Print genes, Ch. 9* 83b⟩≡ (80a)
```
fmt.Printf("\tnode[shape=point,penwidth=4];\n")
for i := 0; i < m; i++ {
        fmt.Printf("\t")
        for j := 0; j < n; j++ {
                in := pop[i][j]
                ⟨Calculate position of individual, Ch. 9 83c⟩
                ⟨Draw first gene, Ch. 9 83d⟩
                ⟨Draw second gene, Ch. 9 83e⟩
        }
        fmt.Printf("\n")
}
```

As before, an individual's position is a function of its row and column indexes.

83c ⟨*Calculate position of individual, Ch. 9* 83c⟩≡ (83b)
```
x := float64(j+1) * f
y := float64(m-i) * f
```

The first gene is 0.15 to the left of the hosting individual. Its name is that of the hosting individual appended by `_0`.

83d ⟨*Draw first gene, Ch. 9* 83d⟩≡ (83b)
```
name := in.n + "_0"
tmpl := "%s[pos=\"%.4g,%.4g!\"];"
fmt.Printf(tmpl, name, x-0.15, y)
```

The second gene is 0.15 to the right of the host and its name ends in `_1`.

83e ⟨*Draw second gene, Ch. 9* 83e⟩≡ (83b)
```
name = in.n + "_1"
fmt.Printf(tmpl, name, x+0.15, y)
```

For every individual on the path we draw the edges.

83f ⟨*Print lines of descent, Ch. 9* 83f⟩≡ (80a)
```
fmt.Printf("edge[color=black]")
for i := 1; i < m; i++ {
        for j := 0; j < n; j++ {
                in := pop[i][j]
                if in.isOnPath {
                        ⟨Draw edges, Ch. 9 84a⟩
                }
        }
}
```

We either draw the edges for an individual or for genes.

84a ⟨*Draw edges, Ch. 9* 84a⟩≡ (83f)
```
if *optGG {
        ⟨Draw edges for genes, Ch. 9 84b⟩
} else {
        ⟨Draw edges for individual, Ch. 9 84c⟩
}
```

For each gene that's switched on, we print the edge to its ancestor.

84b ⟨*Draw edges for genes, Ch. 9* 84b⟩≡ (84a)
```
for k := 0; k < 2; k++ {
        if in.g[k] {
                so := fmt.Sprintf("%s_%d", in.n, k)
                de := fmt.Sprintf("%s", in.ag[k])
                fmt.Printf("\t%s--%s\n", so, de)
        }
}
```

When tracing individuals, there are always two edges, one for each gene.

84c ⟨*Draw edges for individual, Ch. 9* 84c⟩≡ (84a)
```
fmt.Printf("\t%s_0--%s;", in.n, in.ag[0])
fmt.Printf("%s_1--%s;\n", in.n, in.ag[1])
```

The graph is closed by a curly bracket.

84d ⟨*Print graph footer, Ch. 9* 84d⟩≡ (80a)
```
fmt.Printf("}\n")
```

We've finished `drag`, let's test it.

Testing

The outline of our test of `drag` has hooks for imports and the testing logic.

84e ⟨*drag_test.go* 84e⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 9 85a⟩
)

func TestDrag(t *testing.T) {
        ⟨Testing, Ch. 9 84f⟩
}
```

We construct a set of tests and run them.

84f ⟨*Testing, Ch. 9* 84f⟩≡ (84e)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 9 85b⟩
for i, test := range tests {
        ⟨Run test, Ch. 9 85d⟩
}
```

We import `exec`.

85a ⟨*Testing imports, Ch. 9* 85a⟩≡ (84e) 85e▷
```
"os/exec"
```

We construct a first batch of six tests. The first is without any options apart from the seed for the random number generator. The second sets the number of generations, the third the number of individuals, the fours traces individuals 3 and 4, the fifth traces all individuals, and the sixth prints the ancestor stats.

85b ⟨*Construct tests, Ch. 9* 85b⟩≡ (84f) 85c▷
```
test := exec.Command("./drag", "-s", "1")
tests = append(tests, test)
test = exec.Command("./drag", "-s", "1", "-g", "5")
tests = append(tests, test)
test = exec.Command("./drag", "-s", "1", "-n", "5")
tests = append(tests, test)
test = exec.Command("./drag", "-s", "1", "-t", "3,4")
tests = append(tests, test)
test = exec.Command("./drag", "-s", "1", "-t", "-1")
tests = append(tests, test)
test = exec.Command("./drag", "-s", "1", "-a")
tests = append(tests, test)
```

We also test the option for tracing only the genes of individual 5.

85c ⟨*Construct tests, Ch. 9* 85b⟩+≡ (84f) ◁85b
```
test = exec.Command("./drag", "-s", "1", "-G", "-t", "5")
tests = append(tests, test)
```

When running a test, we compare the result we get with the result we want. The result we want is stored in one of the files `r1.txt`, `r2.txt`, and so on.

85d ⟨*Run test, Ch. 9* 85d⟩≡ (84f)
```
get, err := test.Output()
if err != nil { t.Errorf("can't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("can't open %q", f) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

85e ⟨*Testing imports, Ch. 9* 85a⟩+≡ (84e) ◁85a
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 10

Program `drawf`: Draw Wright-Fisher Population

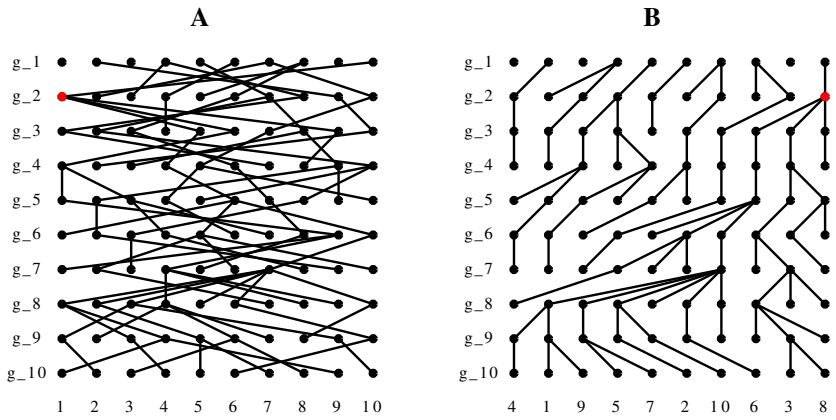


Figure 10.1: Tangled (**A**) and untangled (**B**) Wright-Fisher population; the red gene in generation g_2 is the most recent common ancestor of the population.

Introduction

The Wright-Fisher model is a simple but widely used model of evolution within populations. It consists of n genes, which are passed between generations by sampling with replacement. The program `drawf` draws the resulting lines of descent either tangled or untangled (Figure 10.1). To untangle the lines of descent, we reorder the descendants according to their ancestors' position in the previous generation. A special gene is the most recent common ancestor of the population, which `drawf` can mark in red, as shown in generation g_2 in Figure 10.1.

The Wright-Fisher population is written in the dot notation of the graphviz package and visualized using the program `neato`, also part of graphviz.

Implementation

The outline of `drawf` has hooks for imports, types, and the logic of the main function.

87 ⟨*drawf.go* 87⟩≡

```
package main

import (
        ⟨Imports, Ch. 10 88b⟩
)
⟨Types, Ch. 10 89b⟩
func main() {
        ⟨Main function, Ch. 10 88a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, run the simulation, and print the result.

88a ⟨*Main function, Ch. 10* 88a⟩≡ (87)
```
util.PrepLog("drawf")
⟨Set usage, Ch. 10 88c⟩
⟨Declare options, Ch. 10 88e⟩
⟨Parse options, Ch. 10 89a⟩
⟨Run simulation, Ch. 10 90e⟩
⟨Print simulation, Ch. 10 91b⟩
```

We import `util`.

88b ⟨*Imports, Ch. 10* 88b⟩≡ (87) 88d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `drawf`, and an example command.

88c ⟨*Set usage, Ch. 10* 88c⟩≡ (88a)
```
u := "drawf [-h] [option]..."
p := "Draw Wright-Fisher population."
e := "drawf | neato -T x11"
clio.Usage(u, p, e)
```

We import `clio`.

88d ⟨*Imports, Ch. 10* 88b⟩+≡ (87) ◁88b 88f▷
```
"github.com/evolbioinf/clio"
```

We declare seven options, the number of genes, (`-n`), the number of generations (`-g`), untangled lines of descent (`-u`), the seed for the random number generator (`-s`), a scaling factor for the plot (`-f`), marked most recent common ancestor (`-m`), and the version (`-v`). I found the default scaling factor of 0.4 by trial and error.

88e ⟨*Declare options, Ch. 10* 88e⟩≡ (88a)
```
var optN = flag.Int("n", 10, "number of genes")
var optG = flag.Int("g", 10, "number of generations")
var optU = flag.Bool("u", false, "untangled lines of descent")
var optS = flag.Int64("s", 0, "seed for random number generator")
var optF = flag.Float64("f", 0.4, "scaling factor for plot")
var optM = flag.Bool("m", false, "mark most recent common " +
          "ancestor")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

88f ⟨*Imports, Ch. 10* 88b⟩+≡ (87) ◁88d 90a▷
```
"flag"
```

We parse the options and first respond to the version, as this stops the program. Then we construct the Wright-Fisher population, and seed the random number generator.

89a ⟨*Parse options, Ch. 10* 89a⟩≡ (88a)
```
flag.Parse()
if *optV {
        util.PrintInfo("drawf")
}
⟨Construct Wright-Fisher population, Ch. 10 89c⟩
⟨Seed random number generator, Ch. 10 90c⟩
```

A Wright-Fisher population consists of genes. A gene has an ancestor, a list of descendants, an ID, and a label. A gene can also be the most recent common ancestor of the whole population. To find the most recent common ancestor, we note the number of descendants in the present.

89b ⟨*Types, Ch. 10* 89b⟩≡ (87)
```
type gene struct {
        a *gene
        d []*gene
        i int
        l string
        isMrca bool
        p int
}
```

We construct the population as an $m \times n$ matrix of genes. At the end we set the descendants in the present.

89c ⟨*Construct Wright-Fisher population, Ch. 10* 89c⟩≡ (89a)
```
m := *optG
n := *optN
wfp := make([][]*gene, m)
for i := 0; i < m; i++ {
        wfp[i] = make([]*gene, n)
        for j := 0; j < n; j++ {
                ⟨Construct gene, Ch. 10 89d⟩
        }
}
⟨Set descendants in the present, Ch. 10 90b⟩
```

A gene's ID is its column index, its label a string representation of the row and column index separated by an underscore. The separator is important because pairs of numbers with more than two digits are indistinguishable without it. For example, $(11, 1)$ differs from $(1, 11)$, but without a separator we'd write 111 both times.

89d ⟨*Construct gene, Ch. 10* 89d⟩≡ (89c)
```
wfp[i][j] = new(gene)
wfp[i][j].d = make([]*gene, 0)
wfp[i][j].i = j
wfp[i][j].l = fmt.Sprintf("i%d_%d", i, j)
```

We import `fmt`.

90a ⟨*Imports, Ch. 10* 88b⟩+≡ (87) ◁88f 90d▷
```
"fmt"
```

Initially, the only genes that have descendants in the present are the genes in the present.

90b ⟨*Set descendants in the present, Ch. 10* 90b⟩≡ (89c)
```
genes := wfp[m-1]
for _, gene := range genes {
        gene.p = 1
}
```

If the user provided a seed for the random number generator, we use that, otherwise we use the current time.

90c ⟨*Seed random number generator, Ch. 10* 90c⟩≡ (89a)
```
seed := *optS
if seed == 0 {
        seed = time.Now().UnixNano()
}
source := rand.NewSource(seed)
r := rand.New(source)
```

We import `time` and `rand`.

90d ⟨*Imports, Ch. 10* 88b⟩+≡ (87) ◁90a
```
"time"
"math/rand"
```

In the simulation, each gene picks a random ancestor from the previous generation. This ancestor has the current gene as one of its descendants. If requested, we also determine the most recent common ancestor.

90e ⟨*Run simulation, Ch. 10* 90e⟩≡ (88a)
```
for i := 1; i < m; i++ {
        for j := 0; j < n; j++ {
                p := r.Intn(n)
                a := wfp[i-1][p]
                a.d = append(a.d, wfp[i][j])
                wfp[i][j].a = a
        }
}
if *optM {
        ⟨Find most recent common ancestor, Ch. 10 91a⟩
}
```

The most recent common ancestor is the first gene with all present genes in its tree of descendants. To find it, we walk from the present generation into the past and add the number of descendants of the current gene to its ancestor. The most recent common ancestor is the gene with as many descendants as the population size. We abandon the search as soon as we've found it.

91a ⟨*Find most recent common ancestor, Ch. 10* 91a⟩≡ (90e)

```
found := false
for i := m-1; i > 0; i-- {
        for _, gene := range wfp[i] {
                gene.a.p += gene.p
                if gene.a.p == n {
                        gene.a.isMrca = true
                        found = true
                        break
                }
        }
        if found { break }
}
```

If the user requested untangled lines of descent, we untangle them. Then we print the graph header. A Wright-Fisher population consists of nodes and edges. We first print the nodes, then the edges. If so desired, we mark the most recent common ancestor. This is done *after* the edges have been drawn to cover them with a nice red dot, rather than one that intersects the ends of the edges. We end with the graph footer.

91b ⟨*Print simulation, Ch. 10* 91b⟩≡ (88a)

```
if *optU {
        ⟨Untangle lines of descent, Ch. 10 91c⟩
}
⟨Print header, Ch. 10 92a⟩
⟨Print nodes, Ch. 10 92b⟩
⟨Print edges, Ch. 10 93a⟩
if *optM {
        ⟨Mark most recent common ancestor, Ch. 10 93b⟩
}
⟨Print footer, Ch. 10 93c⟩
```

To untangle lines of descent, we walk from the past to the present and always reorder the next generation according to its ancestors' positions.

91c ⟨*Untangle lines of descent, Ch. 10* 91c⟩≡ (91b)

```
for i := 0; i < m-1; i++ {
        k := 0
        for j := 0; j < n; j++ {
                for _, d := range wfp[i][j].d {
                        wfp[i+1][k] = d
                        k++
                }
        }
}
```

In the header we first explain in a commend the graph's origin and how to visualize it using `neato`. Then we declare a directional graph.

92a ⟨*Print header, Ch. 10* 92a⟩≡ (91b)
```
fmt.Println("# Wright-Fisher population generated with drawf.")
fmt.Println("# Render with neato, e.g.")
fmt.Println("# $ neato -T x11 foo.dot")
fmt.Println("digraph g {")
```

We declare the nodes as points and give them fixed positions using the "pos" attribute with exclamation mark. Each row of genes starts with a generation label. The last row of nodes are the gene IDs.

92b ⟨*Print nodes, Ch. 10* 92b⟩≡ (91b)
```
fmt.Println("\tnode [shape=point, penwidth=4.0];")
f := *optF
for i, genes := range wfp {
        fmt.Printf("\tg_%d[shape=plaintext,pos=\"%.4g,%.4g!\"];",
                i+1, 0.0, float64(m-i) * f)
        ⟨Print row of nodes, Ch. 4 92c⟩
        fmt.Printf("\n")
}
⟨Print gene IDs, Ch. 10 92d⟩
```

If a gene is the most recent common ancestor, we print it in red.

92c ⟨*Print row of nodes, Ch. 4* 92c⟩≡ (92b)
```
for j, gene := range genes {
        fmt.Printf("%s[pos=\"%.4g,%.4g!\"", gene.l,
                float64(j+1) * f, float64(m-i) * f)
        // if gene.isMrca {
        //    fmt.Printf(",color=\"red\"")
        // }
        fmt.Printf("];")
}
```

We render the gene IDs as i_1, i_2, and so on.

92d ⟨*Print gene IDs, Ch. 10* 92d⟩≡ (92b)
```
genes = wfp[m-1]
fmt.Println("\tnode [shape=plaintext]")
fmt.Printf("\t")
for i, gene := range genes {
        x := float64(i+1) * f
        fmt.Printf("%d[pos=\"%.4g,%.4g!\"];",
                gene.i+1, x, 0.0)
}
fmt.Printf("\n")
```

For each gene we draw a line to its ancestor in the previous generation. Edges are lines without arrowheads.

93a ⟨*Print edges, Ch. 10* 93a⟩≡ (91b)
```
fmt.Println("\tedge [arrowhead=none,penwidth=2.0];")
for i := 1; i < m; i++ {
        genes := wfp[i]
        fmt.Printf("\t")
        for _, g := range genes {
                fmt.Printf("%s->%s;", g.l, g.a.l)
        }
        fmt.Printf("\n")
}
```

We mark the most recent common ancestor in red.

93b ⟨*Mark most recent common ancestor, Ch. 10* 93b⟩≡ (91b)
```
for i, genes := range wfp {
        for j, gene := range genes {
                if gene.isMrca {
                        fmt.Printf("mrca[pos=\"%.4g,%.4g!\"",
                                float64(j+1) * f, float64(m-i) * f)
                        fmt.Printf("shape=point,color=\"red\"];")
                }
        }
}
```

The graph is closed by a curly bracket.

93c ⟨*Print footer, Ch. 10* 93c⟩≡ (91b)
```
fmt.Println("}")
```

We're done with `drawf`, let's test it.

Testing

Our testing code has hooks for imports and the testing logic.

93d ⟨*drawf_test.go* 93d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 10 94b⟩
)
func TestDrawf(t *testing.T) {
        ⟨Testing, Ch. 10 94a⟩
}
```

We construct the tests and then iterate over them.

94a ⟨*Testing, Ch. 10* 94a⟩≡ (93d)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 10 94c⟩
for i, test := range tests {
        ⟨Run test, Ch. 10 94d⟩
}
```

We import `exec`.

94b ⟨*Testing imports, Ch. 10* 94b⟩≡ (93d) 94e ▷
```
"os/exec"
```

We test untangling and marking.

94c ⟨*Construct tests, Ch. 10* 94c⟩≡ (94a)
```
test := exec.Command("./drawf", "-s", "4")
tests = append(tests, test)
test = exec.Command("./drawf", "-s", "4", "-u")
tests = append(tests, test)
test = exec.Command("./drawf", "-s", "4", "-m")
tests = append(tests, test)
```

For each tests we compare the result we get with the result we want. The results we want are stored in files `r1.txt`, `r2.txt`, and `r3.txt`.

94d ⟨*Run test, Ch. 10* 94d⟩≡ (94a)
```
get, err := test.Output()
if err != nil { t.Errorf("can't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("can't open %q", f) }
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

94e ⟨*Testing imports, Ch. 10* 94b⟩+≡ (93d) ◁ 94b
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 11

Program `drawGenes`: Convert Gene Coordinates to x/y Coordinates

A

100	400	+
600	1500	-

B

100	0
100	1
400	1
400	0
600	0
600	-1
1500	-1
1500	0
100	0

C

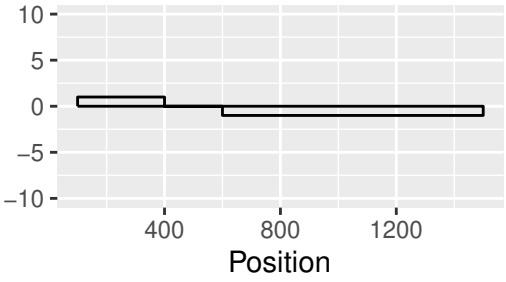


Figure 11.1: Gene coordinates (**A**) are transformed by `drawGene` to x/y coordinates (**B**), which can be plotted with `plotLine` (**C**).

Introduction

Genes are often represented by their start and end positions, and their strand. An example are the two genes in Figure 11.1A, the first on the forward strand, the second on the reverse.

A simple way to draw such data is to convert it to x/y coordinates with genes on the forward strand represented by rectangles above the zero line and genes on the reverse strand below, as shown in Figure 11.1B and plotted in Figure 11.1C. The program `drawGenes` takes data like that in Figure 11.1A and converts it the x/y coordinates in Figure 11.1B, ready for plotting.

Implementation

Our outline of `drawGenes` has hooks for imports and functions.

96 ⟨*drawGenes.go* 96⟩≡

```
package main

import (
        ⟨Imports, Ch. 11 97b⟩
)

⟨Functions, Ch. 11 98a⟩
func main() {
        ⟨Main function, Ch. 11 97a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

97a ⟨*Main function, Ch. 11* 97a⟩≡ (96)
```
util.PrepLog("drawGenes")
⟨Set usage, Ch. 11 97c⟩
⟨Declare options, Ch. 11 97e⟩
⟨Parse options, Ch. 11 97g⟩
⟨Parse input files, Ch. 11 97h⟩
```

We import `util`.

97b ⟨*Imports, Ch. 11* 97b⟩≡ (96) 97d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `drawGenes`, and an example command.

97c ⟨*Set usage, Ch. 11* 97c⟩≡ (97a)
```
u := "drawGenes [-h|-v] foo.txt"
p := "Convert gene coordinates to x/y coordinates for plotting."
e := "drawGenes foo.txt | plotLine -x Position -Y \"-10:10\""
clio.Usage(u, p, e)
```

We import `clio`.

97d ⟨*Imports, Ch. 11* 97b⟩+≡ (96) ◁97b 97f▷
```
"github.com/evolbioinf/clio"
```

We only declare the version, `-v`.

97e ⟨*Declare options, Ch. 11* 97e⟩≡ (97a)
```
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

97f ⟨*Imports, Ch. 11* 97b⟩+≡ (96) ◁97d 98b▷
```
"flag"
```

We parse the options and respond to `-v`, as this would stop the program.

97g ⟨*Parse options, Ch. 11* 97g⟩≡ (97a)
```
flag.Parse()
if *optV {
        util.PrintInfo("drawGenes")
}
```

The remaining tokens on the command line are taken as the names of input files. Each of them is parsed by `scan`.

97h ⟨*Parse input files, Ch. 11* 97h⟩≡ (97a)
```
files := flag.Args()
clio.ParseFiles(files, scan)
```

Inside `scan` we iterate over the input and draw a box for each gene. At the end we close the boxes with a line to the smallest x-position.

98a ⟨*Functions, Ch. 11* 98a⟩≡ (96)

```
func scan(r io.Reader, args ...interface{}) {
        sc := bufio.NewScanner(r)
        min := math.MaxFloat64
        for sc.Scan() {
                fields := strings.Fields(sc.Text())
                ⟨Draw gene, Ch. 11 98c⟩
        }
        fmt.Printf("%g\t0\n", min)
}
```

We import `io`, `bufio`, `math`, `strings`, and `fmt`.

98b ⟨*Imports, Ch. 11* 97b⟩+≡ (96) ◁97f 98d▷

```
"io"
"bufio"
"math"
"strings"
"fmt"
```

A gene has a start position, an end position, and a strand. Start and end position determine its x-coordinates, the strand the y-coordinate.

98c ⟨*Draw gene, Ch. 11* 98c⟩≡ (98a)

```
x1, err := strconv.ParseFloat(fields[0], 64)
if err != nil { log.Fatalf("can't convert %q", fields[0]) }
x2, err := strconv.ParseFloat(fields[1], 64)
if err != nil { log.Fatalf("can't convert %q", fields[0]) }
y := 1
if fields[2] == "-" { y = -1 }
fmt.Printf("%g\t0\n%g\t%d\n%g\t%d\n%g\t0\n",
        x1, x1, y, x2, y, x2)
if x1 < min { min = x1 }
```

We import `strconv` and `log`.

98d ⟨*Imports, Ch. 11* 97b⟩+≡ (96) ◁98b

```
"strconv"
"log"
```

We have finished `drawGenes`, let's test it.

Testing

The outline of our testing code has hooks for imports and the testing logic.

99a ⟨*drawGenes_test.go* 99a⟩≡
```
package main
import (
        "testing"
        ⟨Testing imports, Ch. 11 99c⟩
)
func TestDrawGenes(t *testing.T) {
        ⟨Testing, Ch. 11 99b⟩
}
```

We construct a test by applying `drawGenes` to the gene coordinates in `t.txt`. We compare the result we get with the result we want contained in `r.txt`.

99b ⟨*Testing, Ch. 11* 99b⟩≡ (99a)
```
test := exec.Command("./drawGenes", "t.txt")
get, err := test.Output()
if err != nil { t.Errorf("can't run %q", test) }
want, err := ioutil.ReadFile("r.txt")
if err != nil { t.Errorf("can't open %q", "r.txt") }
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `exec`, `ioutil`, and `bytes`.

99c ⟨*Testing imports, Ch. 11* 99c⟩≡ (99a)
```
"os/exec"
"io/ioutil"
"bytes"
```

Chapter 12

Program `drawKt`: Draw Keyword Tree

Introduction

A keyword tree is a data structure for efficient set matching [2], and the program `drawKt` draws keyword trees as plain text or in in LaTeX. Take, for example, the five patterns

- $p_1 =$ `ATTT`
- $p_2 =$ `ATTC`
- $p_3 =$ `AT`
- $p_4 =$ `TG`
- $p_5 =$ `TT`

The LaTeX-version of their keyword tree is shown in Figure 12.1A. Each character is drawn along match link, the failure links are red arks. When thinking about trees, it is often useful to label each node. Figure 12.1B shows the keyword tree with labeled nodes and patterns labeled p_i instead of plain i. The corresponding text version (Figure 12.1C) is in Newick format[1]. An example node is

```
5[T->9{1,5}]
```

which means that node 5 has an incoming edge labeled `T`, a failure link referring to node 9, and an output set comprising p_1 and p_5—just as shown in Figure 12.1B.

Implementation

The program outline contains hooks for imports, functions, and the logic of the main function.

101a ⟨*drawKt.go* 101a⟩≡

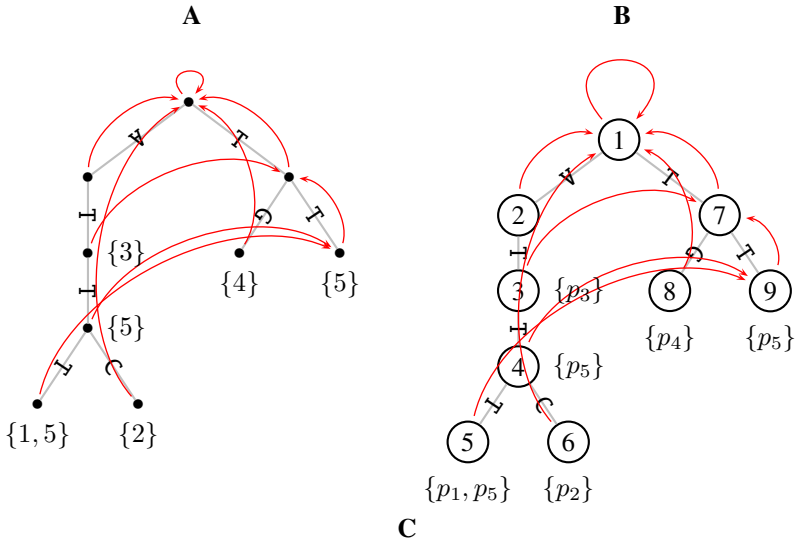
```
package main

import (
        ⟨Imports, Ch. 12 102a⟩
)
⟨Functions, Ch. 12 104a⟩
func main() {
        ⟨Main function, Ch. 12 101b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare and parse the options, get the patterns, and draw their tree.

101b ⟨*Main function, Ch. 12* 101b⟩≡ (101a)

```
util.PrepLog("drawKt")
⟨Set usage, Ch. 12 102b⟩
⟨Declare options, Ch. 12 102d⟩
⟨Parse options, Ch. 12 103b⟩
⟨Get patterns, Ch. 12 103c⟩
⟨Draw tree, Ch. 12 103e⟩
```

[1] `evolution.genetics.washington.edu/phylip/newick_doc.html`



C

```
((((5[T->9{1,5}],6[C->1{2}])4[T->9{5}])3[T->7{3}],
  (8[G->1{4}],9[T->7{5}])7[T->1])2[A->1])1[->1];
```

Figure 12.1: Keyword tree of the fire patterns p_1 = `ATTT`, p_2 = `ATTC`, p_3 = `AT`, p_4 = `TG`, p_5 = `TT` in LaTeX (**A**), again in LaTeX but with labeled nodes (**B**) and more elaborately labeled patterns, and in text format (**C**).

We import `util`.

102a ⟨*Imports, Ch. 12* 102a⟩≡ (101a) 102c ▷

```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts, the actual usage message, an explanation of the program's purpose, and an example command.

102b ⟨*Set usage, Ch. 12* 102b⟩≡ (101b)

```
u := "drawKt [-h] [options] [patterns]"
p := "Draw the keyword tree of a set of patterns"
e := "drawKt ATTT ATTC AT TG TT > kt.tex"
clio.Usage(u, p, e)
```

We import `clio`.

102c ⟨*Imports, Ch. 12* 102a⟩+≡ (101a) ◁ 102a 103a ▷

```
"github.com/evolbioinf/clio"
```

Apart from the default help option, there are five declared options, the user can request a LaTeX wrapper, labeled instead of plain nodes, plain text instead of LaTeX, and the program version.

102d ⟨*Declare options, Ch. 12* 102d⟩≡ (101b)

```
var optW = flag.String("w", "", "LaTeX wrapper file")
var optL = flag.Bool("l", false, "labeled nodes; default: plain")
var optT = flag.Bool("t", false, "plain text; default: LaTeX")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

103a ⟨*Imports, Ch. 12* 102a⟩+≡ (101a) ◁102c 103d▷
```
"flag"
```

When parsing the options, we respond to `-v`.

103b ⟨*Parse options, Ch. 12* 103b⟩≡ (101b)
```
flag.Parse()
if *optV {
        util.PrintInfo("drawKt")
}
```

Patterns are read either from the command line or from the standard input stream.

103c ⟨*Get patterns, Ch. 12* 103c⟩≡ (101b)
```
var patterns []string
if len(flag.Args()) > 0 {
        patterns = flag.Args()
} else {
        sc := bufio.NewScanner(os.Stdin)
        for sc.Scan() {
                patterns = append(patterns, sc.Text())
        }
}
```

We import `bufio` and `os`.

103d ⟨*Imports, Ch. 12* 102a⟩+≡ (101a) ◁103a 103f▷
```
"bufio"
"os"
```

The tree is drawn either as plain text or in LaTeX. If drawn in LaTeX, we also write a wrapper upon request to make the output more useful.

103e ⟨*Draw tree, Ch. 12* 103e⟩≡ (101b)
```
tree := kt.NewKeywordTree(patterns)
if *optT {
        fmt.Println(tree)
} else {
        fmt.Println(writeLatex(tree, *optL))
        if *optW != "" {
                ⟨Write wrapper, Ch. 12 109b⟩
        }
}
```

We import `kt` and `fmt`.

103f ⟨*Imports, Ch. 12* 102a⟩+≡ (101a) ◁103d 104b▷
```
"github.com/evolbioinf/kt"
"fmt"
```

The function `writeLatex` converts a keyword tree to LaTeX and print it to a byte buffer that is returned as a string. We implement the actual printing by first working out the coordinates of each node. Then we print the nodes, the match & failure links, and the output sets.

104a ⟨*Functions, Ch. 12* 104a⟩≡ (101a) 104d▷

```
func writeLatex(root *kt.Node, optL bool) string {
        w := new(bytes.Buffer)
        ⟨Calculate y-coordinates, Ch. 12 104c⟩
        ⟨Calculate x-coordinates, Ch. 12 104e⟩
        ⟨Print nodes, Ch. 12 106c⟩
        ⟨Print match links, Ch. 12 107d⟩
        ⟨Print failure links, Ch. 12 107f⟩
        ⟨Print output sets, Ch. 12 108d⟩
        return(w.String())
}
```

We import `bytes`.

104b ⟨*Imports, Ch. 12* 102a⟩+≡ (101a) ◁103f 109d▷

```
"bytes"
```

To work out the y-coordinates, take another look at the keyword tree in Figure 12.1A. The bottom left hand corner of the image has coordinates $(0, 0)$, the top right hand corner $(\text{width}, \text{md})$, where md is the maximum depth. A y-coordinate is thus $\text{md} - \text{depth}$, which means we need to first compute the maximum depth. We do this using the function `BreadthFirst`, which traverses the tree breadth-first and applies the function `findMaxDepth` with argument `findMaxDepth` to every node.

104c ⟨*Calculate y-coordinates, Ch. 12* 104c⟩≡ (104a)

```
var maxDepth int
kt.BreadthFirst(root, findMaxDepth, &maxDepth)
```

In `findMaxDepth`, we retrieves the variable `maxDepth` by reflection and update it where appropriate.

104d ⟨*Functions, Ch. 12* 104a⟩+≡ (101a) ◁104a 105b▷

```
func findMaxDepth(v *kt.Node, args ...interface{}) {
        maxDepth := args[0].(*int)
        if v.Depth > *maxDepth {
                *maxDepth = v.Depth
        }
}
```

The organizing principle of the x-coordinates is that the leaves are spread evenly along the horizontal axis and a parent's position is in the middle of its children. Thus we compute the horizontal distance between leaves and then calculate the x-coordinate of every node.

104e ⟨*Calculate x-coordinates, Ch. 12* 104e⟩≡ (104a)

```
⟨Compute distance between leaves, Ch. 12 105a⟩
⟨Compute x-coordinates, Ch. 12 105c⟩
```

The distance between two neighboring leaves is the width of the tree divided by the number of leaves minus one. The number of leaves is taken as the tree's width.

105a ⟨*Compute distance between leaves, Ch. 12* 105a⟩≡ (104e)
```
var nl int
kt.BreadthFirst(root, countLeaves, &nl)
dist := float64(nl) / (float64(nl) - 1.0)
```

The function `countLeaves` takes as argument a node and a pointer to an integer.

105b ⟨*Functions, Ch. 12* 104a⟩+≡ (101a) ◁104d 105d▷
```
func countLeaves(n *kt.Node, args ...interface{}) {
        nl := args[0].(*int)
        if n.Child == nil {
                *nl = *nl + 1
        }
}
```

To compute the x-coordinates, we need space to store them. So we allocate a slice with as many entries as nodes and refer to a particular entry by a node's ID. Then we traverse the tree postorder [22, p. 334] to ensure the leaves are encountered before their parents.

105c ⟨*Compute x-coordinates, Ch. 12* 105c⟩≡ (104e)
```
xcoords := make([]float64, kt.NodeCount())
var curX float64
postorder(root, setX, &curX, dist, xcoords)
```

We write the postorder traversal.

105d ⟨*Functions, Ch. 12* 104a⟩+≡ (101a) ◁105b 105e▷
```
func postorder(v *kt.Node, fn kt.NodeAction, args ...interface{}) {
        if v != nil {
                postorder(v.Child, fn, args...)
                fn(v, args...)
                postorder(v.Sib, fn, args...)
        }
}
```

The function `setX` retrieves the arguments just passed and places the node, which is either a leaf or a parent.

105e ⟨*Functions, Ch. 12* 104a⟩+≡ (101a) ◁105d 106e▷
```
func setX(v *kt.Node, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 12 105f⟩
        if v.Child == nil {
                ⟨Place leaf, Ch. 12 106a⟩
        } else {
                ⟨Place parent, Ch. 12 106b⟩
        }
}
```

The arguments are retrieved via reflection.

105f ⟨*Retrieve arguments, Ch. 12* 105f⟩≡ (105e)
```
curX := args[0].(*float64)
dist := args[1].(float64)
xcoords := args[2].([]float64)
```

A leaf is placed at the current x-position, which is then incremented.

106a ⟨*Place leaf, Ch. 12* 106a⟩≡ (105e)
```
xcoords[v.Id] = *curX
*curX = *curX + dist
```

A parent is placed in the middle of its children.

106b ⟨*Place parent, Ch. 12* 106b⟩≡ (105e)
```
x1 := xcoords[v.Child.Id]
cp := v.Child
for cp.Sib != nil {
        cp = cp.Sib
}
x2 := xcoords[cp.Id]
xcoords[v.Id] = (x1 + x2) / 2.0
```

Nodes are printed in a `pspicture`, which has dimensions we still need to determine before we can declare it. Then we apply the function `writeLatexNode` to each node of the tree.

106c ⟨*Print nodes, Ch. 12* 106c⟩≡ (104a)
```
⟨Calculate picture coordinates, Ch. 12 106d⟩
fmt.Fprintf(w, "\\begin{pspicture}(%.2g,%.2g)(%.2g,%.2g)\n",
        x1, y1, x2, y2)
fmt.Fprint(w, "%% Nodes\n")
kt.BreadthFirst(root, writeLatexNode, w, xcoords, maxDepth, optL)
```

The picture coordinates depend on whether or not we are using labeled nodes, because labeled nodes are larger.

106d ⟨*Calculate picture coordinates, Ch. 12* 106d⟩≡ (106c)
```
var x1 float64
y1 := -0.8
x2 := float64(nl)
y2 := float64(maxDepth)+0.7
if optL {
        x1 -= 0.3
        y1 -= 0.2
        x2 += 0.3
        y2 += 0.5
}
```

To write a node, we retrieve the arguments just passed, compute the node's coordinates, and print it.

106e ⟨*Functions, Ch. 12* 104a⟩+≡ (101a) ◁105e 107e▷
```
func writeLatexNode(v *kt.Node, args ...interface{}) {
        ⟨Retrieve LaTeX arguments, Ch. 12 107a⟩
        ⟨Compute coordinates, Ch. 12 107b⟩
        ⟨Print node, Ch. 12 107c⟩
}
```

The argumetns are retrieved by reflection.

107a ⟨*Retrieve LaTeX arguments, Ch. 12* 107a⟩≡ (106e)
```
w := args[0].(*bytes.Buffer)
xcoords := args[1].([]float64)
maxDepth := args[2].(int)
optL := args[3].(bool)
```

The y-coordinate is the inverse of the depth, the x-coordinate is looked up.

107b ⟨*Compute coordinates, Ch. 12* 107b⟩≡ (106e)
```
y := maxDepth - v.Depth
x := xcoords[v.Id]
```

By default the tree consists of unlabeled `dotnodes` (•) but the user can request `cnodes` with one-based labels instead, (1).

107c ⟨*Print node, Ch. 12* 107c⟩≡ (106e)
```
if optL {
        fmt.Fprintf(w, "\\cnodeput(%.3g,%d){%d}{%d}\n",
                x, y, v.Id, v.Id+1)
} else {
        fmt.Fprintf(w, "\\dotnode(%.3g,%d){%d}\n",
                x, y, v.Id)
}
```

Light gray match links are added in a second traversal.

107d ⟨*Print match links, Ch. 12* 107d⟩≡ (104a)
```
fmt.Fprint(w, "%% Match links\n")
fmt.Fprintf(w, "\\psset{linecolor=lightgray}")
kt.BreadthFirst(root, writeMatchLink, w)
```

Except for the root, each node has one incoming match-link.

107e ⟨*Functions, Ch. 12* 104a⟩+≡ (101a) ◁106e 108a▷
```
func writeMatchLink(v *kt.Node, args ...interface{}) {
        w := args[0].(*bytes.Buffer)
        if v.Parent != nil {
                p := v.Parent.Id
                n := v.Id
                c := v.In
                fmt.Fprintf(w, "\\ncline{%d}{%d}" +
                        "\\ncput[nrot=:U]{\\texttt{%c}}\n",
                        p, n, c)
        }
}
```

Failure links are red arks $v \to f(v)$. There is a 2pt gap between the arrows and their start and the end nodes. To draw them, we need the relative x-positions of v and fail(v); so the x-coordinates are passed into the computation.

107f ⟨*Print failure links, Ch. 12* 107f⟩≡ (104a)
```
fmt.Fprint(w, "%% Failure links\n")
fmt.Fprint(w, "\\psset{linecolor=red,linewidth=0.5pt,nodesep=2pt}\n")
kt.BreadthFirst(root, writeFailureLink, w, xcoords)
```

Every node has a failure link that points to another node, except for the root, whose failure link points to itself. So we distinguish these two types of failure links.

108a ⟨*Functions, Ch. 12* 104a⟩+≡ (101a) ◁107e 108e▷
```
func writeFailureLink(v *kt.Node, args ...interface{}) {
        w := args[0].(*bytes.Buffer)
        xcoords := args[1].([]float64)
        if v.Parent == nil {
                ⟨Write failure link for root, Ch. 12 108b⟩
        } else {
                ⟨Write failure link for non-root, Ch. 12 108c⟩
        }
}
```

After reading up on nodes and their connections in [29, p.162] and some trial and error, I found that the root's self-referential failure link is best drawn with `nccurve`.

108b ⟨*Write failure link for root, Ch. 12* 108b⟩≡ (108a)
```
fmt.Fprint(w, "\\nccurve[angleA=130,angleB=50,ncurv=6]" +
        "{->}{0}{0}\n")
```

An ordinary failure link is either left-tilted or right-tilted.

108c ⟨*Write failure link for non-root, Ch. 12* 108c⟩≡ (108a)
```
angle := 50
x1 := xcoords[v.Id]
x2 := xcoords[v.Fail.Id]
if x1 > x2 { angle = -50 }
fmt.Fprintf(w, "\\ncarc[arcangle=%d]{->}{%d}{%d}\n",
        angle, v.Id, v.Fail.Id)
```

The output sets are printed last, and we omit the newline from the last line.

108d ⟨*Print output sets, Ch. 12* 108d⟩≡ (104a)
```
fmt.Fprint(w, "%% Output sets\n")
kt.BreadthFirst(root, writeOutputSet, w, optL)
fmt.Fprintf(w, "\\end{pspicture}")
```

Output sets of leaves are placed below them at an angle of -90, output sets of other nodes to their right, at an angle of 0. As you can see when comparing Figures 12.1A and B, there is a simple and a slightly more fancy version of the output set, and the distinction requires a bit of extra reasoning.

108e ⟨*Functions, Ch. 12* 104a⟩+≡ (101a) ◁108a
```
func writeOutputSet(v *kt.Node, args ...interface{}) {
        if len(v.Output) == 0 { return }
        w := args[0].(*bytes.Buffer)
        optL := args[1].(bool)
        angle := 0
        if v.Child == nil { angle = -90 }
        ⟨Print simple or fancy output set 109a⟩
}
```

The simple

109a ⟨*Print simple or fancy output set* 109a⟩≡ (108e)
```
fmt.Fprintf(w, "\\nput{%d}{%d}{$\\{", angle, v.Id)
if optL { fmt.Fprintf(w, "p_") }
fmt.Fprintf(w, "%d", v.Output[0]+1)
for i := 1; i < len(v.Output); i++ {
          fmt.Fprintf(w, ",")
          if optL { fmt.Fprintf(w, "p_") }
          fmt.Fprintf(w, "%d", v.Output[i]+1)
}
fmt.Fprint(w, "\\}$}\n")
```

When writing the wrapper, we open a file, write to it, and tell the user what we have done.

109b ⟨*Write wrapper, Ch. 12* 109b⟩≡ (103e)
```
⟨Open file, Ch. 12 109c⟩
⟨Write to file, Ch. 12 109e⟩
⟨Tell user, Ch. 12 109f⟩
```

We open the file passed with `-w`.

109c ⟨*Open file, Ch. 12* 109c⟩≡ (109b)
```
f, err := os.Create(*optW)
if err != nil {
          log.Fatalf("couldn't open %q\n", *optW)
}
```

We import `os` and `log`.

109d ⟨*Imports, Ch. 12* 102a⟩+≡ (101a) ◁104b 109g▷
```
"os"
"log"
```

We write brief LaTeX code to the wrapper file and close it again.

109e ⟨*Write to file, Ch. 12* 109e⟩≡ (109b)
```
fmt.Fprintf(f, "\\documentclass{article}\n")
fmt.Fprintf(f, "\\usepackage{pst-all}\n")
fmt.Fprintf(f, "\\begin{document}\n")
fmt.Fprintf(f, "\\begin{center}\n\\input{kt.tex}\n\\end{center}\n")
fmt.Fprintf(f, "\\end{document}\n")
f.Close()
```

We tell the user how to use the wrapper.

109f ⟨*Tell user, Ch. 12* 109f⟩≡ (109b)
```
old := *optW
new := strings.TrimSuffix(old, ".tex")
fmt.Fprintf(os.Stderr, "# Wrote wrapper %s; if the keyword tree is in " +
          "kt.tex, run \n# latex %s\n# dvips %s -o -q\n# " +
          "ps2pdf %s.ps\n", old, new, new, new)
```

We import `strings`.

109g ⟨*Imports, Ch. 12* 102a⟩+≡ (101a) ◁109d
```
"strings"
```

The drawing program is done, time for testing fun.

Testing

The outline of the testing program contains hooks for imports and the testing logic.

110a ⟨*drawKt_test.go* 110a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 12 110c⟩
)

func TestDrawKt(t *testing.T) {
        ⟨Testing, Ch. 12 110b⟩
}
```

We define test cases and compare the output we get with the pre-computed output we want. To avoid repeating ourselves, we first construct the test commands and the output files, and then run loop over the commands.

110b ⟨*Testing, Ch. 12* 110b⟩≡ (110a)
```
var commands []*exec.Cmd
⟨Declare commands, Ch. 12 110d⟩
⟨Construct list of output files, Ch. 12 110e⟩
for i, command := range commands {
        ⟨Run command, Ch. 12 111b⟩
}
```

We import `exec`.

110c ⟨*Testing imports, Ch. 12* 110c⟩≡ (110a) 111a▷
```
"os/exec"
```

We run the program three times, in default mode, text mode, and with labeled nodes.

110d ⟨*Declare commands, Ch. 12* 110d⟩≡ (110b)
```
c := exec.Command("./drawKt", "ATTT", "ATTC", "AT", "TG", "TT")
commands = append(commands, c)
c = exec.Command("./drawKt", "-t", "ATTT", "ATTC", "AT", "TG", "TT")
commands = append(commands, c)
c = exec.Command("./drawKt", "-l", "ATTT", "ATTC", "AT", "TG", "TT")
commands = append(commands, c)
```

For each command, there is an output file.

110e ⟨*Construct list of output files, Ch. 12* 110e⟩≡ (110b)
```
var names []string
for i, _ := range commands {
        s := "r" + strconv.Itoa(i+1) + ".txt"
        names = append(names, s)
}
```

We import `strconv`.

111a ⟨*Testing imports, Ch. 12* 110c⟩+≡ (110a) ◁110c 111c▷

```
"strconv"
```

For each command, we compare the output we get with what we want.

111b ⟨*Run command, Ch. 12* 111b⟩≡ (110b)

```
get, err := command.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", command)
}
want, err := ioutil.ReadFile(names[i])
if err != nil {
          t.Errorf("couldnt' open %q\n", names[i])
}
if !bytes.Equal(want, get) {
          t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `ioutil` and `bytes`.

111c ⟨*Testing imports, Ch. 12* 110c⟩+≡ (110a) ◁111a

```
"io/ioutil"
"bytes"
```

Chapter 13

Program `drawSt`: Draw Suffix Tree

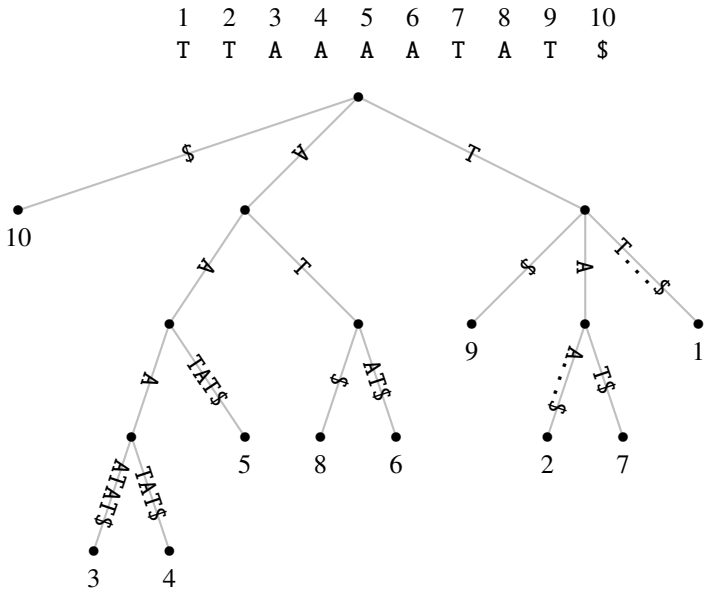


Figure 13.1: Suffix tree of t=TTAAAATAT$.

Introduction

When you read a novel, you might like to look up the pages where a particular character is mentioned. So you leaf through the book and scan the pages for the character's name. If it's a long novel, this takes longer than if its a short novel. But regardless of the novel's length, this would be easier if it had an index. Most novels don't, but most textbooks do. To look up a word in a textbook, just find it in its index. In other words, by using an index, searching a text becomes independent of its length.

A suffix tree is a perfect index in the sense that any word can be looked up in it, not just particular terms considered important by an author. Figure 13.1 shows the suffix tree of the text

$$t = \texttt{TTAAAATAT}$$

The program `drawSt` takes as input a FASTA-formatted sequence and draws its suffix tree.

Each leaf in the suffix tree in Figure 13.1 is labeled by a number. This refers to the starting position of the suffix obtained by concatenating the characters on the path from the root to that leaf. For example, leaf 7 has the path label `TAT$`, the suffix starting at position 7. This also means the tree has as many leaves as there are suffixes—and by the same token characters—in t. The last character, `$`, is a sentinel that mismatches every ordinary character to ensure that all suffixes end in a mismatch, and hence a leaf. Each internal node is also labeled by a circled number, the length of its path label, also known as the node's depth.

As explained in Chapter 41, suffix trees are computed by traversing their alphabetically ordered suffixes, their suffix array. Table 13.1 shows the suffix array of t, sa. If you read it top to bottom, you get the same list of suffixes as reading the leaves of the suffix tree left to right. Far left is the 10, followed by 3, then 4, and so on.

The transformation of a suffix array to a suffix tree requires an auxiliary array of the lengths of the matching prefixes of $\mathrm{suf}[i]$ and $\mathrm{suf}[i-1]$. This array is called the longest

Table 13.1: Suffix array of $t =$ `TTAAAATAT$`.

i	sa$[i]$	lcp$[i]$	suf$[i]$
1	10	-1	`$`
2	3	0	`AAAATAT$`
3	4	3	`AAATAT$`
4	5	2	`AATAT$`
5	8	1	`AT$`
6	6	2	`ATAT$`
7	9	0	`T$`
8	2	1	`TAAAATAT$`
9	7	2	`TAT$`
10	1	1	`TTAAAATAT$`

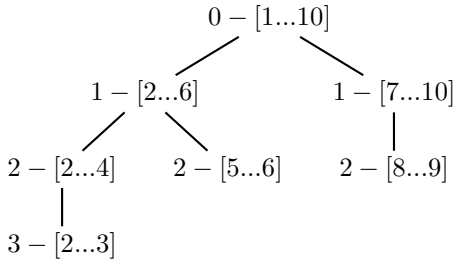


Figure 13.2: Suffix tree of $t =$ `TTAAAATAT$` in interval notation.

common prefix array, lcp, and is also shown in Table 13.1. For example, suf$[4]$ = `AATAT$` matches suf$[3]$ = `AAATAT$` in the first two nucleotides, hence lcp$[4] = 2$.

A suffix tree is computed by finding the intervals in sa corresponding to its internal nodes. For example, the root corresponds to interval $[1..10]$, the node colored in red to the interval $[2..4]$. These intervals are augmented by their depths, d, to give nodes of the form $d - [\ell..r]$. Figure 13.2 shows this interval version of the suffix tree in Figure 13.1. Its shape is that of the suffix tree in Figre 13.1 stripped of its leaves. `DrawSt` can also draw suffix trees in the interval style.

We compute the interval tree using Algorithm 1 [25, p. 94]. Nodes are written as quartets d, ℓ, r, c, where d is the depth, ℓ and r the left and right interval borders, and c the child. An as yet unknown right border is -1, no child is $\perp$. This interval tree is then converted to the full suffix tree.

Apart from the trees in Figures 13.1 and 13.2, `drawSt` can also produce suffix trees in the Newick[1] notation used for phylogenies. Figure 13.3A shows the Newick tree string of the. This string can be converted to a proper tree (Figure 13.3B. Now it isn't just the branch order that carries meaning, but also the branch lengths, as they are proportional to the length of the path label.

[1] `evolution.genetics.washington.edu/phylip/newick_doc.html`

Algorithm 1 Algorithm for computing suffix tree [25, p. 94].

Require: sa {suffix array}
Require: lcp {-1 appended}
Require: n {length of lcp array}
Ensure: Suffix tree in interval notation

```
v ← ⊥
push(0, 1, −1, ⊥) {push root onto stack}
for i ← 2 to n do
  ℓ ← i − 1
  while stack not empty and lcp[i] < top().d do
    top().r ← i − 1
    v ← pop()
    ℓ ← v.ℓ
    if stack not empty and lcp[i] ≤ top().d then
      top().addChild(v)
      v ← ⊥
    end if
  end while
  if stack not empty and lcp[i] > top().d then
    push(lcp[i], ℓ, −1, v)
    v ← ⊥
  end if
end for
```

A

```
(10:1,(((3:5,4:4):4,5:4):4,(8:1,6:3):3):1,(9:1,(2:7,7:2):2,1:9):9):1;
```

B

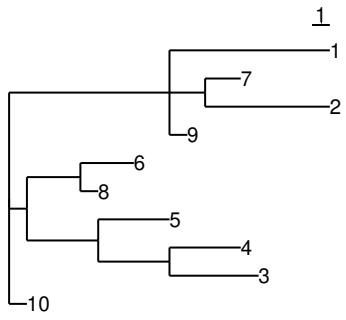


Figure 13.3: Suffix tree drawn like a phylogeny. (**A**) Newick notation; (**B**) drawn as tree; the scale corresponds to one character.

Implementation

The program outline contains hooks for imports, variables, types, methods, functions, and the logic of the main function.

116a ⟨*drawSt.go* 116a⟩≡
```
package main
import (
        ⟨Imports, Ch. 13 116c⟩
)
⟨Variables, Ch. 13 119b⟩
⟨Types, Ch. 13 118d⟩
⟨Methods, Ch. 13 118e⟩
⟨Functions, Ch. 13 117e⟩
func main() {
        ⟨Main function, Ch. 13 116b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare and parse the options, and parse the input files.

116b ⟨*Main function, Ch. 13* 116b⟩≡ (116a)
```
util.PrepLog("drawSt")
⟨Set usage, Ch. 13 116d⟩
⟨Declare options, Ch. 13 117a⟩
⟨Parse options, Ch. 13 117c⟩
⟨Parse input files, Ch. 13 117d⟩
```

We import `util`.

116c ⟨*Imports, Ch. 13* 116c⟩≡ (116a) 116e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts, the actual usage message, an explanation of the program's purpose, and an example command.

116d ⟨*Set usage, Ch. 13* 116d⟩≡ (116b)
```
m := "drawSt [-h] [options] [files]"
p := "Draw suffix tree."
e := "drawSt foo.fasta"
clio.Usage(m, p, e)
```

We import `clio`.

116e ⟨*Imports, Ch. 13* 116c⟩+≡ (116a) ◁ 116c 117b ▷
```
"github.com/evolbioinf/clio"
```

By default, the program specifies the tree in LaTeX using the conventional notation of Figure 13.1. We declare options for the two alternative formats, interval notation (`-i`, Figure 13.2), and Newick notation (`-n`, Figure 13.3A). The user can also print the node depth (`-d`), label the nodes (`-l`), change the default x- and y-units (`-x` & `-y`), add a sentinel character (`-w`), write a LaTeX wrapper (`-w`), and print the program version (`-v`).

117a ⟨*Declare options, Ch. 13* 117a⟩≡ (116b)
```
var optI = flag.Bool("i", false, "interval notation, LaTeX")
var optN = flag.Bool("n", false, "Newick notation, plain text")
var optD = flag.Bool("d", false, "show node depth")
var optL = flag.Bool("l", false, "label nodes")
var optX = flag.Float64("x", 1, "x-unit in LaTeX")
var optY = flag.Float64("y", 1.5, "y-unit in LaTeX")
var optS = flag.Bool("s", false, "add sentinel character")
var optW = flag.String("w", "", "LaTeX wrapper file")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

117b ⟨*Imports, Ch. 13* 116c⟩+≡ (116a) ◁116e 117f▷
```
"flag"
```

We parse the options and respond to `-v`.

117c ⟨*Parse options, Ch. 13* 117c⟩≡ (116b)
```
flag.Parse()
if *optV {
        util.PrintInfo("drawSt")
}
```

The remaining tokens on the command line are interpreted as input files, which get parsed using the function `scan`. It takes as arguments the seven options specifying the tree format. Once we've parsed the input files, we print the LaTeX wrapper, if desired.

117d ⟨*Parse input files, Ch. 13* 117d⟩≡ (116b)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optI, *optN, *optD, *optL, *optX,
        (*optY), *optS)
if *optW != "" {
        ⟨Write wrapper, Ch. 13 129e⟩
}
```

In `scan`, we retrieve the options before iterating over the sequences in the file.

117e ⟨*Functions, Ch. 13* 117e⟩≡ (116a) 119a▷
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve options, Ch. 13 118a⟩
        ⟨Iterate over sequences, Ch. 13 118b⟩
}
```

We import `io`.

117f ⟨*Imports, Ch. 13* 116c⟩+≡ (116a) ◁117b 118c▷
```
"io"
```

The seven options just passed are retrieved by reflection.

118a ⟨*Retrieve options, Ch. 13* 118a⟩≡ (117e)
```
optI := args[0].(bool)
optN := args[1].(bool)
optD := args[2].(bool)
optL := args[3].(bool)
optX := args[4].(float64)
optY := args[5].(float64)
optS := args[6].(bool)
```

For each sequence, we extract the sequence data, compute the suffix tree, and draw it.

118b ⟨*Iterate over sequences, Ch. 13* 118b⟩≡ (117e)
```
scanner := fasta.NewScanner(r)
for scanner.ScanSequence() {
        sequence := scanner.Sequence()
        data := sequence.Data()
        ⟨Compute suffix tree, Ch. 13 120a⟩
        ⟨Draw suffix tree, Ch. 13 122c⟩
}
```

We import `fasta`.

118c ⟨*Imports, Ch. 13* 116c⟩+≡ (116a) ◁117f 120c▷
```
"github.com/evolbioinf/fasta"
```

A suffix tree consists of nodes, which in turn consist of a depth, a left border, and a right border. The tree topology is established through references to a child node and a sibling. In a conventional suffix tree (Figure 13.1), each incoming edge of a node has a label. We don't store the label but compute it whenever required. To do that, we need to know not only the current node's depth, but also its parent's, and hence include a pointer to parent. A node also has an ID for easy reference and a level in the tree.

118d ⟨*Types, Ch. 13* 118d⟩≡ (116a) 119c▷
```
type node struct {
        d, l, r, id, level int
        child, sib, parent *node
}
```

For easy printing of nodes we implement `String`.

118e ⟨*Methods, Ch. 13* 118e⟩≡ (116a) 119d▷
```
func (v *node) String() string {
        if v == nil {
                return "!"
        }
        s := fmt.Sprintf("%d-[%d..%d]", v.d, v.l, v.r)
        return s
}
```

A new node is constructed as a function of its depth, left and right borders, and child node.

119a ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁117e 122e▷
```
func newNode(d, l, r int, child *node) *node {
        n := new(node)
        n.d = d
        n.l = l
        n.r = r
        n.id = nodeId
        nodeId++
        if child != nil {
                n.child = child
                child.parent = n
        }
        return n
}
```

The node identifier is kept in a global variable.

119b ⟨*Variables, Ch. 13* 119b⟩≡ (116a)
```
var nodeId int
```

According to Algorithm 1, the nodes are kept on a stack, which we implement as a slice of node pointers [7, p. 92].

119c ⟨*Types, Ch. 13* 118d⟩+≡ (116a) ◁118d 123b▷
```
type stack []*node
```

We implement the three canonical stack functions, push, pop, and top.

119d ⟨*Methods, Ch. 13* 118e⟩+≡ (116a) ◁118e 121b▷
```
func (s *stack) push(n *node) { *s = append(*s, n) }
func (s *stack) pop() *node {
        n := (*s)[len(*s)-1]
        *s = (*s)[0:len(*s)-1]
        return n
}
func (s *stack) top() *node { return (*s)[len(*s)-1] }
```

To compute the suffix tree, we first prepare the data. The basis for our suffix tree is the enhanced suffix array consisting of the suffix array proper and the lcp array. The lcp array gets a -1 appended to ensure all nodes are eventually popped from the stack in the while loop of Algorithm 1. We also initialize the focal node, v, and the stack, onto which we push the root. Then we traverse the lcp array.

120a ⟨*Compute suffix tree, Ch. 13* 120a⟩≡ (118b)
```
⟨Prepare sequence data, Ch. 13 120b⟩
sa := esa.Sa(data)
lcp := esa.Lcp(data, sa)
lcp = append(lcp, -1)
n := len(lcp)
var v *node
root := newNode(0, 0, -1, nil)
stack := new(stack)
stack.push(root)
⟨Traverse lcp array, Ch. 13 120d⟩
```

If the user requested a sentinel character, we append that to the sequence.

120b ⟨*Prepare sequence data, Ch. 13* 120b⟩≡ (120a)
```
if optS {
        data = append(data, '$')
}
```

We import `esa`.

120c ⟨*Imports, Ch. 13* 116c⟩+≡ (116a) ◁118c 125b▷
```
"github.com/evolbioinf/esa"
```

Algorithm 1 says that for each value in the lcp array, nodes with depths greater $\mathrm{lcp}[i]$ are popped from the stack. If the top node's depth is then less than $\mathrm{lcp}[i]$, a new node is pushed.

120d ⟨*Traverse lcp array, Ch. 13* 120d⟩≡ (120a)
```
for i := 1; i < n; i++ {
        l := i - 1
        for len(*stack) > 0 && lcp[i] < stack.top().d {
                ⟨Pop node, Ch. 13 121a⟩
        }
        if len(*stack) > 0 && lcp[i] > stack.top().d {
                ⟨Push node, Ch. 13 122b⟩
        }
}
```

When popping nodes from the stack, we check whether they are children of the top node. Since adding a child is an operation we need repeatedly, we delegate it to a method.

121a ⟨*Pop node, Ch. 13* 121a⟩≡ (120d)
```
stack.top().r = i - 1
v = stack.pop()
l = v.l
if len(*stack) > 0 && lcp[i] <= stack.top().d {
        p := stack.top()
        p.addChild(v)
        v = nil
}
```

When adding a child to a node, we first assign the child's parent link. Then we either assign the child to the parent's child link, or insert it in the correct position among its siblings. This position is either on the left of its siblings, in between them, or to their right.

121b ⟨*Methods, Ch. 13* 118e⟩+≡ (116a) ◁119d
```
func (p *node) addChild(c *node) {
        c.parent = p
        if p.child == nil {
                p.child = c
        } else {
                ⟨Insert child on the left of siblings?, Ch. 13 121c⟩
                ⟨Insert child between siblings?, Ch. 13 121d⟩
                ⟨Insert child on the right of siblings?, Ch. 13 122a⟩
        }
}
```

If the child's left border is to the left of its first sibling, the new child becomes the first sibling. This child has now been taken care of, so the function returns.

121c ⟨*Insert child on the left of siblings?, Ch. 13* 121c⟩≡ (121b)
```
w := p.child
if c.l < w.l {
        p.child = c
        c.sib = w
        return
}
```

If the new child's left border is between that of two siblings, it is inserted between them.

121d ⟨*Insert child between siblings?, Ch. 13* 121d⟩≡ (121b)
```
for w.sib != nil {
        if c.l > w.r && c.l < w.sib.l {
                c.sib = w.sib
                w.sib = c
                return
        }
        w = w.sib
}
```

If the child is still not assigned, it becomes the last sibling.

122a ⟨*Insert child on the right of siblings?, Ch. 13* 122a⟩≡ (121b)
```
w.sib = c
```

After removing nodes from the stack, we might also push a new node.

122b ⟨*Push node, Ch. 13* 122b⟩≡ (120d)
```
w := newNode(lcp[i], l, -1, v)
stack.push(w)
v = nil
```

There are three tree formats, conventional (Figure 13.1), interval (Figure 13.2), and Newick (Figure 13.3). If we are not drawing an interval tree, we need to add leaves and node levels to our tree.

122c ⟨*Draw suffix tree, Ch. 13* 122c⟩≡ (118b)
```
if !optI {
        ⟨Add leaves, Ch. 13 122f⟩
        ⟨Add levels, Ch. 13 122d⟩
}
if optI {
        ⟨Print interval tree, Ch. 13 127c⟩
} else if optN {
        ⟨Print Newick tree, Ch. 13 128c⟩
} else {
        ⟨Print conventional tree, Ch. 13 124c⟩
}
```

We add the node levels, for which we need a preorder traversal. Since we won't reuse this, we write it as a simple recursion.

122d ⟨*Add levels, Ch. 13* 122d⟩≡ (122c)
```
preorder(root)
```

A node's level is that of its parent plus one.

122e ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁119a 123a▷
```
func preorder(v *node) {
        if v != nil {
                if v.parent != nil {
                        v.level = v.parent.level + 1
                }
                preorder(v.child)
                preorder(v.sib)
        }
}
```

Adding leaves requires another tree traversal. This time, we delegate it to a reusable function, `traverse`. `traverse` takes as argument the root of a tree and a function it applies to each node, in this case `addLeaves`. `AddLeaves` in turn takes as argument the suffix array.

122f ⟨*Add leaves, Ch. 13* 122f⟩≡ (122c)
```
traverse(root, addLeaves, sa)
```

To make `traverse` useful in diverse traversals, it takes a variadic variable consisting of empty interfaces. These and the current node are the arguments of the function applied to every node. We add leaves in a post order traversal, so we implement this variant.

123a ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁122e 123c▷
```
func traverse(v *node, fn nodeAction, args ...interface{}) {
        if v != nil {
                traverse(v.child, fn, args...)
                traverse(v.sib, fn, args...)
                fn(v, args...)
        }
}
```

We define the type of this function, `nodeAction`.

123b ⟨*Types, Ch. 13* 118d⟩+≡ (116a) ◁119c
```
type nodeAction func(*node, ...interface{})
```

If we are dealing with a leaf of the interval tree, all elements of its interval become leaves. Otherwise, we have to compare its interval to that of its children.

123c ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁123a 125a▷
```
func addLeaves(p *node, args ...interface{}) {
        sa := args[0].([]int)
        l := len(sa)
        if p.child == nil {
                for i := p.l; i <= p.r; i++ {
                        c := newNode(l - sa[i], i, i, nil)
                        p.addChild(c)
                }
        } else {
                ⟨Compare parent interval to child intervals, Ch. 13 123d⟩
        }
}
```

Any part of an internal node's interval not found in its children is filled with leaves. We discover these gaps by considering the left, middle and right parts of the parent's interval.

123d ⟨*Compare parent interval to child intervals, Ch. 13* 123d⟩≡ (123c)
```
⟨Add leaves on the left, Ch. 13 123e⟩
⟨Add leaves in the middle, Ch. 13 124a⟩
⟨Add leaves on the right, Ch. 13 124b⟩
```

We fill the gap between the parent's left border and the first child's.

123e ⟨*Add leaves on the left, Ch. 13* 123e⟩≡ (123d)
```
for i := p.l; i < p.child.l; i++ {
        c := newNode(l - sa[i], i, i, nil)
        p.addChild(c)
}
```

We fill the gaps between siblings.

124a ⟨*Add leaves in the middle, Ch. 13* 124a⟩≡ (123d)

```
v := p.child
for v.sib != nil {
        x := v.sib.l
        for i := v.r+1; i < x; i++ {
                c := newNode(l - sa[i], i, i, nil)
                p.addChild(c)
        }
        v = v.sib
}
```

We fill the gap between the right border of the last sibling and the parent's right border.

124b ⟨*Add leaves on the right, Ch. 13* 124b⟩≡ (123d)

```
for i := v.r+1; i <= p.r; i++ {
        c := newNode(l - sa[i], i, i, nil)
        p.addChild(c)
}
```

The tree is now ready to be drawn. For a string of length n, it has n leaves in the x-dimension and n characters in the y-dimension. So we could try to fit our suffix tree into an $n \times n$ square. And while the x-dimension really is a function of n, it turns out that the y-dimension is better taken from the maximum node level.

For the conventional tree (Figure 13.1), we print the picture header, the edges, the nodes, and the picture footer. The picture header takes as arguments the picture dimensions. The x-dimension is equal to the lengrh of the input string times the x-scale factor. The y-dimension is equal to the negative of the largest node level times the y-scale factor.

124c ⟨*Print conventional tree, Ch. 13* 124c⟩≡ (122c)

```
l := len(data)
x := float64(l) * optX
m := maxNodeLevel(root, 0)
y := float64(m) * optY
fmt.Printf("\\begin{pspicture}(%.2g,%.2g)(%.2g,%.2g)\n",
        0.0, -y, x, 0.0)
fmt.Printf("\\psset{xunit=%.3g, yunit=%.3g}\n", optX, optY)
traverse(root, drawCedge, sa, data)
traverse(root, drawCnode, sa, optL, optD)
fmt.Printf("\\end{pspicture}\n")
```

We find the maximum node level by recursion.

125a ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁123c 125c▷
```
func maxNodeLevel(v *node, m int) int {
        if v != nil {
                if v.level > m {
                        m = v.level
                }
                m = maxNodeLevel(v.child, m)
                m = maxNodeLevel(v.sib, m)
        }
        return m
}
```

We import `fmt`.

125b ⟨*Imports, Ch. 13* 116c⟩+≡ (116a) ◁120c 127b▷
```
"fmt"
```

Nodes are drawn in three steps, the arguments are retrieved, the node is drawn and labeled.

125c ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁125a 126b▷
```
func drawCnode(v *node, args ...interface{}) {
        ⟨Retrieve conventional node arguments, Ch. 13 125d⟩
        ⟨Write conventional node, Ch. 13 125e⟩
        ⟨Label conventional node, Ch. 13 126a⟩
}
```

We retrieve the arguments just passed by reflection.

125d ⟨*Retrieve conventional node arguments, Ch. 13* 125d⟩≡ (125c)
```
sa := args[0].([]int)
nodeLabel := args[1].(bool)
depth := args[2].(bool)
```

The x-coordinate of a node is the middle of its interval, the y-coordinate the node level. Nodes are either dots or boxes around a node identifier.

125e ⟨*Write conventional node, Ch. 13* 125e⟩≡ (125c)
```
x := float64(v.l + v.r) / 2.0
if nodeLabel {
        fmt.Printf("\\rput(%.3g,%d){\\rnode{%d}{" +
                "\\psframebox[linecolor=lightgray]{%d}}}",
                x, -v.level, v.id, v.id)
} else {
        fmt.Printf("\\dotnode(%.3g,%d){%d}\n",
                x, -v.level, v.id)
}
```

Leaves are labeled by the suffix position. Internal nodes might be labeled by their string depth, which we place in a box to distinguish it from the labels of nodes and leaves.

126a ⟨*Label conventional node, Ch. 13* 126a⟩≡ (125c)
```
if v.child == nil {
        fmt.Printf("\\nput{-90}{%d}{%d}\n",
                v.id, sa[v.l]+1)
} else if depth {
        fmt.Printf("\\nput{0}{%d}{" +
                "\\ovalnode[linecolor=lightgray]{%d}{%d}}\n",
                v.id, v.id, v.d)
}
```

For each node that isn't the root, we draw an edge to its parent. This is labeled with a substring of the input string. The starting point of the label is the starting point of the suffix minus the parent's depth. The length of the label is the current depth minus the parent's depth.

126b ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁125c 127d▷
```
func drawCedge(v *node, args ...interface{}) {
        if v.parent == nil { return }
        sa := args[0].([]int)
        seq := args[1].([]byte)
        start := sa[v.l] + v.parent.d
        l := v.d - v.parent.d
        label := string(seq[start:start+l])
        ⟨Print edge label, Ch. 13 126c⟩
}
```

To print the edge label, we prepare it and then place it.

126c ⟨*Print edge label, Ch. 13* 126c⟩≡ (126b)
```
⟨Prepare edge label, Ch. 13 126d⟩
⟨Place edge label, Ch. 13 127a⟩
```

We abridge long edge labels using interval notation. The resulting label may contain a dollar character, the classical terminator symbol. As the dollar is part of the LaTeX syntax, we escape it.

126d ⟨*Prepare edge label, Ch. 13* 126d⟩≡ (126c)
```
ll := len(label)
if ll > 5 {
        label = label[:1] + "..." + label[ll-1:ll]
}
label = strings.Replace(label, "$", "\\$", 1)
```

We place the label in the center of a text path along a line from the parent to the child.

127a ⟨*Place edge label, Ch. 13* 127a⟩≡ (126c)
```
x1 := float64(v.parent.l + v.parent.r) / 2.0
y1 := -v.parent.level
x2 := float64(v.l + v.r) / 2.0
y2 := -v.level
tp := "\\pstextpath[c]{\\psline[linecolor=lightgray](%.3g,%d)" +
        "(%.3g,%d)}{\\texttt{%s}}\n"
fmt.Printf(tp, x1, y1, x2, y2, label)
```

We import `strings`.

127b ⟨*Imports, Ch. 13* 116c⟩+≡ (116a) ◁125b 129g▷
```
"strings"
```

We are done with the conventional tree (Figure 13.1) and move to the interval tree (Figure 13.2). We print this as a ps-tree with node separation of 2 points and level separation of 1 cm.

127c ⟨*Print interval tree, Ch. 13* 127c⟩≡ (122c)
```
fmt.Printf("\\psset{nodesep=2pt, levelsep=1cm}\n")
printIntervals(root)
```

The function `printIntervals` is recursive. Inside it, we distinguish between leaves and internal nodes of the interval tree.

127d ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁126b 128d▷
```
func printIntervals(i *node) {
        if i == nil { return }
        ⟨Is i a leaf? Ch. 13 127e⟩
        ⟨Is i an internal node? Ch. 13 127f⟩
}
```

If i is a leaf, we print its ps-tree representation.

127e ⟨*Is i a leaf? Ch. 13* 127e⟩≡ (127d)
```
if i.child == nil {
        s := "\\Tr{$%d-[%d...%d]$}\n"
        fmt.Printf(s, i.d, i.l+1, i.r+1)
}
```

If i is an internal node, we open it and add children and siblings to the subtree rooted on it.

127f ⟨*Is i an internal node? Ch. 13* 127f⟩≡ (127d)
```
⟨Open internal node, Ch. 13 127g⟩
⟨Add child to internal node, Ch. 13 128a⟩
⟨Add sibling to internal node, Ch. 13 128b⟩
```

If the current node has children, it is an internal node and the root of a subtree.

127g ⟨*Open internal node, Ch. 13* 127g⟩≡ (127f)
```
if i.child != nil {
        s := "\\pstree{\\Tr{$%d-[%d...%d]$}}{\n"
        fmt.Printf(s, i.d, i.l+1, i.r+1)
}
```

We follow the child link and note whether or not the subtree is closed.

128a ⟨*Add child to internal node, Ch. 13* 128a⟩≡ (127f)
```
printIntervals(i.child)
closed := false
if i.child != nil {
          fmt.Printf("}\n")
          closed = true
}
```

We follow the child link and if we end up with an as yet open internal node, we close that.

128b ⟨*Add sibling to internal node, Ch. 13* 128b⟩≡ (127f)
```
printIntervals(i.sib)
if i.child != nil && !closed {
          fmt.Printf("}\n")
}
```

The interval tree is finished and we get to the third and last tree version, Newick (Figure 13.3A). To write the tree in that format, we follow the explanation given in the keyword tree package, `kt`[2]. We call a new traversal function with the suffix array as argument, for labeling the leaves.

128c ⟨*Print Newick tree, Ch. 13* 128c⟩≡ (122c)
```
printNewick(root, sa)
```

In the implementation, we test whether a node is *not* the first child, whether it's a leaf, whether it's an internal node, and whether it's the root.

128d ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁127d 129a▷
```
func printNewick(v *node, args ...interface{}) {
          if v == nil { return }
          sa := args[0].([]int)
          ⟨Is v not a first child? Ch. 13 128e⟩
          ⟨Is v a leaf? Ch. 13 128f⟩
          ⟨Is v an internal node? Ch. 13 129c⟩
          ⟨Is v the root? Ch. 13 129d⟩
}
```

Nodes subsequent to the first child are preceded by commas.

128e ⟨*Is v not a first child? Ch. 13* 128e⟩≡ (128d)
```
if v.parent != nil && v.parent.child.id != v.id {
          fmt.Printf(",")
}
```

Leaves are labeled.

128f ⟨*Is v a leaf? Ch. 13* 128f⟩≡ (128d)
```
if v.child == nil {
        label(v, sa)
}
```

[2] `github.com/evolbioinf/kt`

A node label consists of the starting position of the corresponding suffix and a branch length.

129a ⟨*Functions, Ch. 13* 117e⟩+≡ (116a) ◁128d
```
func label(v *node, sa []int) {
        fmt.Printf("%d", sa[v.l] + 1)
        ⟨Branch length, Ch. 13 129b⟩
}
```

The branch length consists of the number of characters on the incoming edge.

129b ⟨*Branch length, Ch. 13* 129b⟩≡ (129)
```
if v.parent != nil {
        l := v.d - v.parent.d
        fmt.Printf(":%d", l)
}
```

Internal nodes are enclosed in brackets and come with the length of the edge to the parent.

129c ⟨*Is v an internal node? Ch. 13* 129c⟩≡ (128d)
```
if v.child != nil { fmt.Printf("(") }
printNewick(v.child, sa)
printNewick(v.sib, sa)
if v.parent != nil && v.sib == nil {
        fmt.Printf(")")
        ⟨Branch length, Ch. 13 129b⟩
}
```

The root is denoted by a semicolon and a newline.

129d ⟨*Is v the root? Ch. 13* 129d⟩≡ (128d)
```
if v.parent == nil {
        fmt.Printf(";\n")
}
```

We've now written the tree, but the user might also have requested a LaTeX wrapper. We open the file for it, write the wrapper to that file, and tell the user about it.

129e ⟨*Write wrapper, Ch. 13* 129e⟩≡ (117d)
```
⟨Open file, Ch. 13 129f⟩
⟨Write to file, Ch. 13 130a⟩
⟨Tell user, Ch. 13 130b⟩
```

We open the file passed with `-w` and bail on fail.

129f ⟨*Open file, Ch. 13* 129f⟩≡ (129e)
```
f, err := os.Create(*optW)
if err != nil {
        log.Fatalf("couldn't open %q\n", *optW)
}
```

We import `os` and `log`.

129g ⟨*Imports, Ch. 13* 116c⟩+≡ (116a) ◁127b
```
"os"
"log"
```

We write a LaTeXarticle that wraps the hypothetical input file `st.tex`.

130a ⟨*Write to file, Ch. 13* 130a⟩≡ (129e)
```
fmt.Fprintf(f, "\\documentclass{article}\n")
fmt.Fprintf(f, "\\usepackage{pst-all}\n")
fmt.Fprintf(f, "\\begin{document}\n")
fmt.Fprintf(f, "\\begin{center}\n\\input{st}\n\\end{center}\n")
fmt.Fprintf(f, "\\end{document}\n")
f.Close()
```

We tell the user via the standard error stream how to use the wrapper. The file names used in the instructions are constructed by trimming `.tex` off the wrapper file name.

130b ⟨*Tell user, Ch. 13* 130b⟩≡ (129e)
```
old := *optW
new := strings.TrimSuffix(old, ".tex")
fmt.Fprintf(os.Stderr, "# Wrote wrapper to %s; if the suffix tree is in " +
          "st.tex, run\n# latex %s\n# dvips %s\n# " +
          "ps2pdf %s.ps\n", old, new, new, new)
```

The program `drawSt` is finished, so we test it next.

Testing

Our testing outline has hooks for imports and the testing logic.

130c ⟨*drawSt_test.go* 130c⟩≡
```
package main

import (
          "testing"
          ⟨Testing imports, Ch. 13 130e⟩
)

func TestDrawSt(t *testing.T) {
          ⟨Testing logic, Ch. 13 130d⟩
}
```

We test in two steps. First we construct the tests, then we iterate over them and run them.

130d ⟨*Testing logic, Ch. 13* 130d⟩≡ (130c)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 13 131a⟩
for i, test := range tests {
          ⟨Run test, Ch. 13 131b⟩
}
```

We import `exec`.

130e ⟨*Testing imports, Ch. 13* 130e⟩≡ (130c) 131c ▷
```
"os/exec"
```

We test the three trees that `drawSt` can draw: conventional (Figure 13.1), interval (Figure 13.2), and Newick (Figure 13.3A). Every time we use the sequence `TTAAAATAT` with sentinel.

131a ⟨*Construct tests, Ch. 13* 131a⟩≡ (130d)

```
file := "test.fasta"
cmd := exec.Command("./drawSt", "-s", file)
tests = append(tests, cmd)
cmd = exec.Command("./drawSt", "-s", "-i", file)
tests = append(tests, cmd)
cmd = exec.Command("./drawSt", "-s", "-n", file)
tests = append(tests, cmd)
```

When running a test, we compare the output we get with the output we want.

131b ⟨*Run test, Ch. 13* 131b⟩≡ (130d)

```
get, err := test.Output()
if err != nil { t.Error(err.Error()) }
file = "res" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(file)
if err != nil { t.Error(err.Error()) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n",
                  string(get), string(want))
}
```

We import `strconv`, `ioutil`, and `bytes`.

131c ⟨*Testing imports, Ch. 13* 130e⟩+≡ (130c) ◁130e

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 14

Program `fasta2tab`: Convert FASTA to Tabular Format

Introduction

A FASTA file contains one or more sequences in FASTA format. Each sequence consists of a header line followed by multiple lines of sequence data, for example

```
>Seq
TCTCA
CAAAA
```

This format makes it easy for us to read sequences. However, it makes it difficult to parse FASTA sequences as pairs consisting of a name and a sequence. Such a key/value interpretation of sequence data is easiest if it consists of a column of names and a column of corresponding sequences. Our example data would then become

```
Seq         TCTCACAAAA
```

The program `fasta2tab` generates such a table from FASTA-formatted input.

Implementation

Our program outline contains hooks for imports, functions, and the logic of the main function.

133a ⟨*fasta2tab.go* 133a⟩≡

```
package main

import (
        ⟨Imports, Ch. 14 133c⟩
)
⟨Functions, Ch. 14 135a⟩
func main() {
        ⟨Main function, Ch. 14 133b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare and parse the options, and parse the input files.

133b ⟨*Main function, Ch. 14* 133b⟩≡ (133a)

```
util.PrepLog("fasta2tab")
⟨Set usage, Ch. 14 133d⟩
⟨Declare options, Ch. 14 134b⟩
⟨Parse options, Ch. 14 134d⟩
⟨Parse files, Ch. 14 134h⟩
```

We import `util`.

133c ⟨*Imports, Ch. 14* 133c⟩≡ (133a) 134a ▷

```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts, the actual usage message, an explanation of the program's purpose, and an example command.

133d ⟨*Set usage, Ch. 14* 133d⟩≡ (133b)

```
u := "fasta2tab [-h] [option] [file]..."
p := "Convert sequences in FASTA to tabular format."
e := "fasta2tab foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

134a ⟨*Imports, Ch. 14* 133c⟩+≡ (133a) ◁133c 134c▷
```
"github.com/evolbioinf/clio"
```

We declare one option specific to this program, the column delimiter. By default, this is the TAB character. In addition, the user can ask for the version.

134b ⟨*Declare options, Ch. 14* 134b⟩≡ (133b)
```
var optD = flag.String("d", "\t", "field delimiter")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

134c ⟨*Imports, Ch. 14* 133c⟩+≡ (133a) ◁134a 134f▷
```
"flag"
```

We parse the options and respond to `-d` and `-v`.

134d ⟨*Parse options, Ch. 14* 134d⟩≡ (133b)
```
flag.Parse()
⟨Respond to -d, Ch. 14 134e⟩
⟨Respond to -v, Ch. 14 134g⟩
```

To get at the character passed as delimiter, we unquote it.

134e ⟨*Respond to -d, Ch. 14* 134e⟩≡ (134d)
```
delim, err := strconv.Unquote(`"` + *optD + `"`)
if err != nil {
        fmt.Fprintf(os.Stderr,
                "please enter delimiter in quotes\n")
        os.Exit(1)
}
```

We import `strconv`, `fmt`, and `os`.

134f ⟨*Imports, Ch. 14* 133c⟩+≡ (133a) ◁134c 135b▷
```
"strconv"
"fmt"
"os"
```

In response to `-v` we call `PrintInfo` on our program.

134g ⟨*Respond to -v, Ch. 14* 134g⟩≡ (134d)
```
if *optV {
        util.PrintInfo("fasta2tab")
}
```

The remaining tokens on the command line are the input files. We pass them to the function `ParseFiles`. It applies the function `scan` to every file. `scan`, in turn, takes as argument the delimiter.

134h ⟨*Parse files, Ch. 14* 134h⟩≡ (133b)
```
files := flag.Args()
clio.ParseFiles(files, scan, delim)
```

Inside `scan`, we retrieve the delimiter, iterate over the sequences and print each one as the header, followed by the delimiter, followed by the sequence.

135a ⟨*Functions, Ch. 14* 135a⟩≡ (133a)
```
func scan(r io.Reader, args ...interface{}) {
        delim := args[0].(string)
        scanner := fasta.NewScanner(r)
        for scanner.ScanSequence() {
                s := scanner.Sequence()
                fmt.Printf("%s%s%s\n", s.Header(), delim,
                        string(s.Data()))
        }
}
```

We import `io` and `fasta`.

135b ⟨*Imports, Ch. 14* 133c⟩+≡ (133a) ◁134f
```
"io"
"github.com/evolbioinf/fasta"
```

Our program is written, let's test it.

Testing

The outline for our testing code contains hooks for imports and the testing logic.

135c ⟨*fasta2tab_test.go* 135c⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 14 136a⟩
)
func TestFasta2tab(t *testing.T) {
        ⟨Testing, Ch. 14 135d⟩
}
```

We first construct the tests and then iterate over them to run each one.

135d ⟨*Testing, Ch. 14* 135d⟩≡ (135c)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 14 135e⟩
for i, test := range tests {
        ⟨Run test, Ch. 14 136b⟩
}
```

135e ⟨*Construct tests, Ch. 14* 135e⟩≡ (135d)
```
f := "test.fasta"
cmd := exec.Command("./fasta2tab", f)
tests = append(tests, cmd)
cmd = exec.Command("./fasta2tab", "-d", "\\t", f)
tests = append(tests, cmd)
cmd = exec.Command("./fasta2tab", "-d", "\\n", f)
tests = append(tests, cmd)
```

We import `exec`.

136a ⟨*Testing imports, Ch. 14* 136a⟩≡ (135c) 136c ▷
```
"os/exec"
```

When running a test, we check we get what we want.

136b ⟨*Run test, Ch. 14* 136b⟩≡ (135d)
```
get, err := test.Output()
if err != nil { t.Error(err.Error()) }
f = "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Error(err.Error()) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv` and `ioutil`, and `bytes`.

136c ⟨*Testing imports, Ch. 14* 136a⟩+≡ (135c) ◁ 136a
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 15

Program geco: Explore the Genetic Code

A

	T	C	A	G	
T	Phe	Ser	Tyr	Cys	T
	Phe	Ser	Tyr	Cys	C
	Leu	Ser	Ter	Ter	A
	Leu	Ser	Ter	Trp	G
C	Leu	Pro	His	Arg	T
	Leu	Pro	His	Arg	C
	Leu	Pro	Gln	Arg	A
	Leu	Pro	Gln	Arg	G
A	Ile	Thr	Asn	Ser	T
	Ile	Thr	Asn	Ser	C
	Ile	Thr	Lys	Arg	A
	Met	Thr	Lys	Arg	G
G	Val	Ala	Asp	Gly	T
	Val	Ala	Asp	Gly	C
	Val	Ala	Glu	Gly	A
	Val	Ala	Glu	Gly	G

B

	T	C	A	G	
T	Cys	Asn	Arg	Val	T
	Cys	Asn	Arg	Val	C
	Lys	Asn	Ter	Ter	A
	Lys	Asn	Ter	Gln	G
C	Lys	Gly	Glu	Trp	T
	Lys	Gly	Glu	Trp	C
	Lys	Gly	Ala	Trp	A
	Lys	Gly	Ala	Trp	G
A	Tyr	Ser	Phe	Asn	T
	Tyr	Ser	Phe	Asn	C
	Tyr	Ser	Ile	Trp	A
	Leu	Ser	Ile	Trp	G
G	Pro	His	Met	Thr	T
	Pro	His	Met	Thr	C
	Pro	His	Asp	Thr	A
	Pro	His	Asp	Thr	G

Figure 15.1: Natural genetic code (**A**) and a shuffled version (**B**).

Introduction

Similar codons often specify similar amino acids. The extent to which the genetic code is evolved to minimize chemical change can be investigated by testing the hypothesis that amino acids are randomly assigned to codons. In the program `geco` we implement one version of this idea, where the amino acids are shuffled between codon groups [13]. So the natural code shown in Figure 15.1A might be shuffled into Figure 15.1B. Notice that the degeneracy classes remain unchanged. For example, the three sixfold degenerate amino acids leucine, serine, and arginine of the natural code in Figure 15.1A are relabeled lysine, asparagine, and tryptophane in the shuffled code in Figure 15.1B. The stop codon is the only "amino acid" that always retains its position.

The program `geco` reads a file of amino acid properites, for example the polarity values listed in Table 15.1. It then calculates the mean squared difference in polarity between the amino acids in the given genetic code and the amino acids of all one-step mutations. We call this mean squared difference d. `geco` then prints the natural genetic code and its d-value. It can also shuffle the amino acids repeatedly and print the shuffled codes and their d-values.

Implementation

The outline of `geco` contains hooks for imports, types, methods, functions, and the logic of the main function.

138 ⟨*geco.go* 138⟩≡

```
package main

import (
        ⟨Imports, Ch. 15 139b⟩
```

Table 15.1: Polarity values taken from Table 1 in [13].

amino acid	polarity	amino acid	polarity
Ala	7.0	Leu	4.9
Arg	9.1	Lys	10.1
Asp	13.0	Met	5.3
Asn	10.0	Phe	5.0
Cys	4.8	Pro	6.6
Glu	12.5	Ser	7.5
Gln	8.6	Thr	6.6
Gly	7.9	Trp	5.2
His	8.4	Tyr	5.4
Ile	4.9	Val	5.6

```
)
⟨Types, Ch. 15 141a⟩
⟨Methods, Ch. 15 145d⟩
⟨Functions, Ch. 15 141b⟩
func main() {
        ⟨Main function, Ch. 15 139a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

139a ⟨*Main function, Ch. 15* 139a⟩≡ (138)
```
util.PrepLog("geco")
⟨Set usage, Ch. 15 139c⟩
⟨Declare options, Ch. 15 140a⟩
⟨Parse options, Ch. 15 140c⟩
⟨Parse input files, Ch. 15 140f⟩
```

We import `util`.

139b ⟨*Imports, Ch. 15* 139b⟩≡ (138) 139d ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `geco`, and an example command.

139c ⟨*Set usage, Ch. 15* 139c⟩≡ (139a)
```
u := "geco [-h] [option]... property.dat"
p := "Explore the genetic code."
e := "geco -n 10000 polarity.dat | grep '^d'"
clio.Usage(u, p, e)
```

We import `clio`.

139d ⟨*Imports, Ch. 15* 139b⟩+≡ (138) ◁ 139b 140b ▷
```
"github.com/evolbioinf/clio"
```

Apart from the version (`-v`), we declare an option for the number of iterations (`-n`), and the seed of the random number generator, (`-s`).

140a ⟨*Declare options, Ch. 15* 140a⟩≡ (139a)
```
var optV = flag.Bool("v", false, "version")
var optN = flag.Int("n", 0, "number of iterations")
var optS = flag.Int("s", 0, "seed of random number generator")
```

We import `flag`.

140b ⟨*Imports, Ch. 15* 139b⟩+≡ (138) ◁139d 140e▷
```
"flag"
```

We parse the options and first respond to `-v`, as this stops the program. If the program carries out shuffling, we also seed the random number generator.

140c ⟨*Parse options, Ch. 15* 140c⟩≡ (139a)
```
flag.Parse()
if *optV {
        util.PrintInfo("geco")
}
if *optN > 0 {
        ⟨Seed random number generator, Ch. 15 140d⟩
}
```

The random number generator is either initialized with the seed the user supplied, or with the current time.

140d ⟨*Seed random number generator, Ch. 15* 140d⟩≡ (140c)
```
seed := int64(*optS)
if seed == 0 {
        seed = time.Now().UnixNano()
}
rand.Seed(seed)
```

We import `time` and `rand`.

140e ⟨*Imports, Ch. 15* 139b⟩+≡ (138) ◁140b 144b▷
```
"time"
"math/rand"
```

The remaining tokens on the command line are taken as the names of input files. They are parsed with the function `scan`, which takes as parameters the number of iterations and the genetic code. We construct the genetic code as a variable of type `geneticCode`.

140f ⟨*Parse input files, Ch. 15* 140f⟩≡ (139a)
```
files := flag.Args()
gc := newGeneticCode()
clio.ParseFiles(files, scan, *optN, gc)
```

The genetic code consists of five components:

1. a slice of codons
2. the genetic code, a map between strings representing codons and integers representing amino acids

3. a map between codons and their one-step mutants; precomputing them is the same work as computing them once for every shuffling

4. an integer slice to map from one amino acid integer to another; by shuffling this array, we can later shuffle the amino acids between codon classes.

5. a string slice to look up amino acid names

141a ⟨*Types, Ch. 15* 141a⟩≡ (138)
```
type geneticCode struct {
          codons []string
          mutants map[string][]string
          codon2int map[string]int
          int2int   []int
          int2aa    []string
}
```

In `newGeneticCode` we allocate the components of the genetic code and construct it.

141b ⟨*Functions, Ch. 15* 141b⟩≡ (138) 144a▷
```
func newGeneticCode() *geneticCode {
          gc := new(geneticCode)
          gc.codons = make([]string, 0)
          gc.mutants = make(map[string][]string)
          gc.codon2int = make(map[string]int)
          gc.int2int = make([]int, 0)
          gc.int2aa = make([]string, 21)
          ⟨Construct genetic code, Ch. 15 141c⟩
          return gc
}
```

We construct the five components of the genetic code, the slice of codons, the mutants, the codon map, the integer array, and the amino acid array.

141c ⟨*Construct genetic code, Ch. 15* 141c⟩≡ (141b)
```
⟨Construct codons, Ch. 15 142a⟩
⟨Construct mutants, Ch. 15 142b⟩
⟨Construct codon map, Ch. 15 143a⟩
⟨Construct integer array, Ch. 15 143c⟩
⟨Construct amino acid names, Ch. 15 143d⟩
```

We construct the codons in the order in which they appear in the standard genetic code (Figure 15.1A).

142a ⟨*Construct codons, Ch. 15* 142a⟩≡ (141c)

```
dna := "TCAG"
for i := 0; i < 4; i++ {
        for j := 0; j < 4; j++ {
                for k := 0; k < 4; k++ {
                        codon := dna[i:i+1]
                        codon += dna[j:j+1]
                        codon += dna[k:k+1]
                        gc.codons = append(gc.codons, codon)
                }
        }
}
```

We iterate over the codons and construct the mutants for each. Since strings are immutable character slices, we construct the mutant codons using a byte slice.

142b ⟨*Construct mutants, Ch. 15* 142b⟩≡ (141c)

```
b := make([]byte, 3)
for _, codon := range gc.codons {
        mutants := make([]string, 0)
        ⟨Mutate codon, Ch. 15 142c⟩
        gc.mutants[codon] = mutants
}
```

We construct the mutants for the given codon in a triple-nested loop.

142c ⟨*Mutate codon, Ch. 15* 142c⟩≡ (142b)

```
for i := 0; i < 3; i++ {
        for j := 0; j < 3; j++ { b[j] = codon[j] }
        for j := 0; j < 4; j++ {
                b[i] = dna[j]
                if b[i] != codon[i] {
                        mutants = append(mutants, string(b))
                }
        }
}
```

To construct the codon map, we assign an integer to each amino acid. The amino acids appear in the order in which they occur in the standard genetic code, except for the stop codon, which encodes no amino acid and is thus exempt from the analysis. Then we iterate across the 64 codons we've just constructed and assign the corresponding integer.

143a ⟨*Construct codon map, Ch. 15* 143a⟩≡ (141c)
```
aa := "FLSYCWPHQRIMTNKVADEG*"
ai := make(map[byte]int)
for i, a := range aa {
          ai[byte(a)] = i
}
aaTab := "FFLLSSSSYY**CC*W" +
          "LLLLPPPPHHQQRRRR" +
          "IIIMTTTTNNKKSSRR" +
          "VVVVAAAADDEEGGGG"
```
⟨*Iterate over codons, Ch. 15* 143b⟩

We iterate over the codons and map them to integers.

143b ⟨*Iterate over codons, Ch. 15* 143b⟩≡ (143a)
```
for i, codon := range gc.codons {
          gc.codon2int[codon] = ai[aaTab[i]]
}
```

We map the 21 codons to their integers.

143c ⟨*Construct integer array, Ch. 15* 143c⟩≡ (141c)
```
for _, a := range aa {
          gc.int2int = append(gc.int2int, ai[byte(a)])
}
```

We construct the array of three-letter amino acid names; notice the stop codon is last, as above.

143d ⟨*Construct amino acid names, Ch. 15* 143d⟩≡ (141c)
```
names := []string{
          "Phe", "Leu", "Ser", "Tyr", "Cys",
          "Trp", "Pro", "His", "Gln", "Arg",
          "Ile", "Met", "Thr", "Asn", "Lys",
          "Val", "Ala", "Asp", "Glu", "Gly",
          "Ter"}
for i := 0; i < 21; i++ {
          gc.int2aa[i] = names[i]
}
```

Inside `scan`, we retrieve the arguments passed and read the input, a map of amino acids and their properties. Then we carry out the analysis.

144a ⟨*Functions, Ch. 15* 141b⟩+≡ (138) ◁141b 145c▷

```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 15 144e⟩
        aap := make(map[string]float64)
        sc := bufio.NewScanner(r)
        for sc.Scan() {
                ⟨Fill amino acid property map, Ch. 15 144c⟩
        }
        ⟨Carry out analysis, Ch. 15 145a⟩
}
```

We import `io` and `bufio`.

144b ⟨*Imports, Ch. 15* 139b⟩+≡ (138) ◁140e 144d▷

```
"io"
"bufio"
```

The line we just scanned is either a header starting with a hash or a data line. We skip the header; a data line consists of two fields, the amino acid, and its property value. We convert the property value from string to number.

144c ⟨*Fill amino acid property map, Ch. 15* 144c⟩≡ (144a)

```
fields := strings.Fields(sc.Text())
if fields[0][0] == '#' { continue }
aa := fields[0]
x, err := strconv.ParseFloat(fields[1], 64)
if err != nil {
        log.Fatalf("can't convert %q", fields[1])
}
aap[aa] = x
```

We import `strings`, `strconv`, and `log`.

144d ⟨*Imports, Ch. 15* 139b⟩+≡ (138) ◁144b 145b▷

```
"strings"
"strconv"
"log"
```

We retrieve the two arguments, number of iterations and genetic code.

144e ⟨*Retrieve arguments, Ch. 15* 144e⟩≡ (144a)

```
n := args[0].(int)
gc := args[1].(*geneticCode)
```

In the analysis, we calculate d and print it together with the genetic code. This is either done once or repeatedly after shuffling.

145a ⟨*Carry out analysis, Ch. 15* 145a⟩≡ (144a)
```
if n == 0 {
        d := meanDiff(gc, aap)
        fmt.Printf("%sd: %.4g\n", gc, d)
} else {
        for i := 0; i < n; i++ {
                ⟨Shuffle genetic code, Ch. 15 147a⟩
                d := meanDiff(gc, aap)
                fmt.Printf("%sd: %.4g\n", gc, d)
        }
}
```

We import `fmt`.

145b ⟨*Imports, Ch. 15* 139b⟩+≡ (138) ◁144d 146c▷
```
"fmt"
```

To calculate the mean squared difference, d, we iterate over all codons and check whether it's a stop codon, in which case we skip the rest of the analysis. For all other codons, we retrieve the corresponding amino acid and store its property. Then we iterate over the mutants of the codon.

145c ⟨*Functions, Ch. 15* 141b⟩+≡ (138) ◁144a
```
func meanDiff(gc *geneticCode, aap map[string]float64) float64 {
        var d, c float64
        for _, codon := range gc.codons {
                if gc.codon2int[codon] == 20 { continue }
                aa := gc.aa(codon)
                x := aap[aa]
                mutants := gc.mutants[codon]
                ⟨Iterate over codon mutants, Ch. 15 146a⟩
        }
        return d / c
}
```

We implement the method `aa` to look up the amino acid that corresponds to a codon in three steps: look up its integer, map the integer, and retrieve the amino acid corresponding to that integer.

145d ⟨*Methods, Ch. 15* 145d⟩≡ (138) 146b▷
```
func (g *geneticCode) aa(codon string) string {
        ai1 := g.codon2int[codon]
        ai2 := g.int2int[ai1]
        return g.int2aa[ai2]
}
```

We iterate over the codon mutants and again skip the stop codon.

146a ⟨*Iterate over codon mutants, Ch. 15* 146a⟩≡ (145c)
```
for _, mutant := range mutants {
        if gc.codon2int[mutant] == 20 { continue }
        aa := gc.aa(mutant)
        y := aap[aa]
        d += (x-y) * (x-y)
        c++
}
```

We implement the `String` method to print the table holding the genetic code. We format this table using a tab writer.

146b ⟨*Methods, Ch. 15* 145d⟩+≡ (138) ◁145d
```
func (g *geneticCode) String() string {
        buf := new(bytes.Buffer)
        w := tabwriter.NewWriter(buf, 1, 0, 2, ' ', 0)
        ⟨Print table header, Ch. 15 146d⟩
        ⟨Print table body, Ch. 15 146e⟩
        w.Flush()
        return buf.String()
}
```

We import `bytes` and `tabwriter`.

146c ⟨*Imports, Ch. 15* 139b⟩+≡ (138) ◁145b
```
"bytes"
"text/tabwriter"
```

The table header consists of the four nucleotides offset by one tab.

146d ⟨*Print table header, Ch. 15* 146d⟩≡ (146b)
```
dna := "TCAG"
for i := 0; i < 4; i++ {
        fmt.Fprintf(w, "\t %c", dna[i])
}
fmt.Fprint(w, "\n")
```

We use another triple nested loop to generate the table body.

146e ⟨*Print table body, Ch. 15* 146e⟩≡ (146b)
```
for i := 0; i < 4; i++ {
        fmt.Fprintf(w, "%c", dna[i])
        for j := 0; j < 4; j++ {
                for k := 0; k < 4; k++ {
                        c := dna[i:i+1] + dna[k:k+1] +
                                dna[j:j+1]
                        fmt.Fprintf(w, "\t%s", g.aa(c))
                }
                fmt.Fprintf(w, "\t%c\n", dna[j])
        }
}
```

The genetic code is shuffled by shuffling the first twenty entries in the integer map—the twenty-first entry is the stop codon. That leaves the position of the stop codon unchanged, as desired.

147a ⟨*Shuffle genetic code, Ch. 15* 147a⟩≡ (145a)
```
rand.Shuffle(20, func(i, j int) {
          gc.int2int[i], gc.int2int[j] =
                  gc.int2int[j], gc.int2int[i]
})
```

We're finished writing `geco`, let's test it.

Testing

The outline of our testing program has hooks for imports and the testing logic.

147b ⟨*geco_test.go* 147b⟩≡
```
package main

import (
          "testing"
          ⟨Testing imports, Ch. 15 147d⟩
)
func TestGeco(t *testing.T) {
          ⟨Testing, Ch. 15 147c⟩
}
```

We construct a set of tests and then iterate over them.

147c ⟨*Testing, Ch. 15* 147c⟩≡ (147b)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 15 147e⟩
for i, test := range tests {
          ⟨Run test, Ch. 15 148a⟩
}
```

We import `exec`.

147d ⟨*Testing imports, Ch. 15* 147d⟩≡ (147b) 148b ▷
```
"os/exec"
```

We construct two tests, the first without shuffling, the second with two shuffling steps. When shuffling, we also seed the random number generator to ensure predictable results. Both tests run on the polarity data contained in `polarity.dat`.

147e ⟨*Construct tests, Ch. 15* 147e⟩≡ (147c)
```
f := "polarity.dat"
test := exec.Command("./geco", f)
tests = append(tests, test)
test = exec.Command("./geco", "-n", "2", "-s", "13", f)
tests = append(tests, test)
```

When we run a test, we compare the results we get with the results we want, which are contained in files `r1.txt` and `r2.txt`.

148a ⟨*Run test, Ch. 15* 148a⟩≡ (147c)

```
get, err := test.Output()
if err != nil { t.Errorf("can't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("can't open %q", f) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

148b ⟨*Testing imports, Ch. 15* 147d⟩+≡ (147b) ◁147d

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 16

Program `genTree`: Generate Random Tree

A

```
(((T1:33,T2:37):5,(T3:45,T4:
43):1):3,((T5:23,T6:30):17,(
(T7:10,(T8:2,T9:9):5):24,T10
:37):2):5):0;
```

B

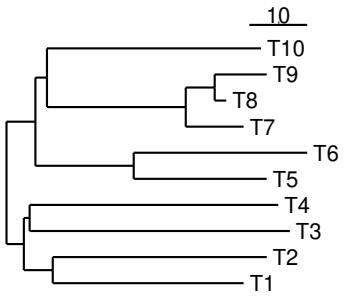


Figure 16.1: A random tree generated by `genTree` in Newick format (**A**) and drawn with `plotTree` (**B**).

Introduction

It is often handy to have a source of random phylogenies and `genTree` prints random phylogenies in Newick format. An example is shown in Figure 16.1A, which is drawn using `plotTree` in Figure 16.1B.

Implementation

The outline of `genTree` has hooks for imports, functions, and the logic of the main function.

150a ⟨*genTree.go* 150a⟩≡

```
package main

import (
        ⟨Imports, Ch. 16 151a⟩
)
⟨Functions, Ch. 16 154c⟩
func main() {
        ⟨Main function, Ch. 16 150b⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and compute the trees.

150b ⟨*Main function, Ch. 16* 150b⟩≡ (150a)

```
util.PrepLog("genTree")
⟨Set usage, Ch. 16 151b⟩
⟨Declare options, Ch. 16 151d⟩
⟨Parse options, Ch. 16 152a⟩
⟨Calculate trees, Ch. 16 152c⟩
```

We import `util`.

151a ⟨*Imports, Ch. 16* 151a⟩≡ (150a) 151c ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts. The actual usage message, an explanation of the program's purpose, and an example command.

151b ⟨*Set usage, Ch. 16* 151b⟩≡ (150b)
```
u := "genTree [-h] [option]..."
p := "Generate random trees."
e := "genTree -n 15"
clio.Usage(u, p, e)
```

We import `clio`.

151c ⟨*Imports, Ch. 16* 151a⟩+≡ (150a) ◁151a 151e ▷
```
"github.com/evolbioinf/clio"
```

We declare eight options:

- `-n` sample size
- `-i` number of iterations
- `-t` mutation rate, $\theta = 2N\mu$
- `-c` coalescent instead of phylogeny
- `-a` absolute branch lengths instead of number of mutations
- `-l` label internal nodes in addition to leaves
- `-s` seed for random number generator
- `-v` version

151d ⟨*Declare options, Ch. 16* 151d⟩≡ (150b)
```
var optN = flag.Int("n", 10, "sample size")
var optI = flag.Int("i", 1, "iterations")
var optT = flag.Float64("t", 1000, "theta=2Nu")
var optC = flag.Bool("c", false, "coalescent")
var optA = flag.Bool("a", false, "absolute branch lengths")
var optL = flag.Bool("l", false, "label internal branches")
var optS = flag.Int("s", 0, "seed for random number generator")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

151e ⟨*Imports, Ch. 16* 151a⟩+≡ (150a) ◁151c 152b ▷
```
"flag"
```

We parse the options and respond to `-v` by printing program information. We also respond to `-s` by seeding the random number generator. The seed is either given by the user or we take the current Unix time in nanoseconds.

152a ⟨*Parse options, Ch. 16* 152a⟩≡ (150b)
```
flag.Parse()
if *optV {
        util.PrintInfo("genTree")
}
seed := int64(*optS)
if seed == 0 {
        seed = time.Now().UnixNano()
}
ran := rand.New(rand.NewSource(seed))
```

We import `time` and `rand`.

152b ⟨*Imports, Ch. 16* 151a⟩+≡ (150a) ◁151e 152d▷
```
"time"
"math/rand"
```

We represet a tree as a slice of nodes. For each iteration, we construct the tree, add mutations if desired, and print it.

152c ⟨*Calculate trees, Ch. 16* 152c⟩≡ (150b)
```
n := *optN
tree := make([]*nwk.Node, 2*n-1)
for ii := 0; ii < *optI; ii++ {
        ⟨Construct tree, Ch. 16 152e⟩
        if !*optA {
                ⟨Add mutations, Ch. 16 154d⟩
        }
        ⟨Print tree, Ch. 16 155d⟩
}
```

We import `nwk`.

152d ⟨*Imports, Ch. 16* 151a⟩+≡ (150a) ◁152b 155c▷
```
"github.com/evolbioinf/nwk"
```

A tree is constructed in four steps: We allocate the nodes, set their times, construct the topology, and determine the branch lengths.

152e ⟨*Construct tree, Ch. 16* 152e⟩≡ (152c)
```
⟨Allocate nodes, Ch. 16 152f⟩
⟨Set node times, Ch. 16 153a⟩
⟨Construct topology, Ch. 16 153b⟩
⟨Determine branch lengths, Ch. 16 154b⟩
```

We allocate the nodes by calling `NewNode`.

152f ⟨*Allocate nodes, Ch. 16* 152f⟩≡ (152e)
```
for i := 0; i < 2*n-1; i++ {
        tree[i] = nwk.NewNode()
}
```

Coalescence times are exponentially distributed with parameter $\binom{i}{2}$, where i is the number of lines. Their computation is summarized in Algorithm 2, which is adapted from [17]. Curiously, the tree looks more like a phylogeny if we use the constant $\binom{n}{2}$ as parameter instead of $\binom{i}{2}$. We also note that the node actually has a branch length.

Algorithm 2 Generate node times in coalescent.

Require: n {sample size}
Require: tree {Array: n leaves, $n-1$ internal nodes}
Ensure: Node times
1: **for** $i \leftarrow 1$ **to** n **do**
2: $\quad \text{tree}[i].\text{time} \leftarrow 0$
3: **end for**
4: $t \leftarrow 0$
5: **for** $i \leftarrow n$ **to** 2 **do**
6: $\quad \lambda \leftarrow \binom{i}{2}$ {Parameter for exponential distribution}
7: $\quad r \leftarrow \text{rexp}(\lambda)$ {Random time to CA with expectation $1/\lambda$}
8: $\quad t \leftarrow t + r$ {Times always increase}
9: $\quad v \leftarrow 2n - i + 1$ {Internal node $n+1, n+2, ..., 2n-1$}
10: $\quad \text{tree}[v].\text{time} \leftarrow t$
11: **end for**

153a ⟨*Set node times, Ch. 16* 153a⟩≡ (152e)

```
t := 0.0
for i := 0; i < n; i++ { tree[i].HasLength = true }
for i := n; i > 1; i-- {
        lambda := float64(n * (n-1) / 2)
        if *optC { lambda = float64(i * (i-1) / 2) }
        t += rand.ExpFloat64() / lambda
        j := 2 * n - i
        tree[j].Length = t
        tree[j].HasLength = true
}
```

We generate the topology of the coalescent by going through the internal nodes and thinking of them as the current parent node. A parent, for which we pick two children [17].

153b ⟨*Construct topology, Ch. 16* 153b⟩≡ (152e)

```
for i := n; i > 1; i-- {
        p := tree[2 * n - i]
        ⟨Pick first child, Ch. 16 153c⟩
        ⟨Pick second child, Ch. 16 154a⟩
}
```

As detailed in Algorithm 3, we pick the first child from among the nodes in positions $0, ..., i-1$ and replace it by node $i-1$.

153c ⟨*Pick first child, Ch. 16* 153c⟩≡ (153b)

```
r := ran.Intn(i)
c := tree[r]
p.AddChild(c)
tree[r] = tree[i-1]
```

Algorithm 3 Generate 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: $c \leftarrow i \times \mathrm{ran}() + 1$ {Draw first child, $1 \leq c \leq i$}
4: $\mathrm{tree}[p].\mathrm{child1} \leftarrow \mathrm{tree}[c]$
5: $\mathrm{tree}[p].\mathrm{child1.parent} \leftarrow \mathrm{tree}[p]$
6: $\mathrm{tree}[c] \leftarrow \mathrm{tree}[i]$ {Replace first child}
7: $c \leftarrow (i-1) \times \mathrm{ran}() + 1$ {Draw second child $1 \leq c \leq i-1$}
8: $\mathrm{tree}[p].\mathrm{child2} \leftarrow \mathrm{tree}[c]$
9: $\mathrm{tree}[p].\mathrm{child2.parent} \leftarrow \mathrm{tree}[p]$
10: $\mathrm{tree}[c] \leftarrow \mathrm{tree}[p]$ {Replace second child by parent}
11: **end for**

We pick the second child and replace it by the current parent, which thus becomes a child candidate in the next round.

154a ⟨*Pick second child, Ch. 16* 154a⟩≡ (153b)
```
r = ran.Intn(i-1)
c = tree[r]
p.AddChild(c)
tree[r] = p
```

The branch lengths are determined by traversing the tree from the root located in the last entry of the tree array.

154b ⟨*Determine branch lengths, Ch. 16* 154b⟩≡ (152e)
```
root := tree[len(tree)-1]
setBranchLen(root)
```

Inside `setBranchLen` we determine the branch length as the difference between the current node position and that of its parent, starting from the leaves.

154c ⟨*Functions, Ch. 16* 154c⟩≡ (150a) 155a ▷
```
func setBranchLen(v *nwk.Node) {
        if v == nil { return }
        setBranchLen(v.Child)
        l := 0.0
        if v.Parent != nil {
                l = v.Parent.Length - v.Length
        }
        v.Length = l
        setBranchLen(v.Sib)
}
```

The number of mutations is a determined in another tree traversal conditioned on the population mutation rate, θ. The number of mutations is a random variable, so the function also takes the random number generator as argument.

154d ⟨*Add mutations, Ch. 16* 154d⟩≡ (152c)
```
addMut(root, *optT, ran)
```

Inside `addMut`, we compute the number of mutations as a function of the branch length and θ.

155a ⟨*Functions, Ch. 16* 154c⟩+≡ (150a) ◁154c 156a▷
```
func addMut(v *nwk.Node, t float64, r *rand.Rand) {
        if v == nil { return }
        ⟨Calculate the number of mutations, Ch. 16 155b⟩
        addMut(v.Child, t, r)
        addMut(v.Sib, t, r)
}
```

The number of mutations is a Poisson-distributed random variable with mean $\lambda = \ell/2\theta$, where ℓ is the branch length. We calculate this random number using a method I took from [23, p. 137].

155b ⟨*Calculate the number of mutations, Ch. 16* 155b⟩≡ (155a)
```
lambda := t * v.Length / 2.0
x := math.Exp(-lambda)
p := 1.0
c := 0.0
for p > x {
        p *= r.Float64()
        c++
}
v.Length = c
```

We import `math`.

155c ⟨*Imports, Ch. 16* 151a⟩+≡ (150a) ◁152d 155e▷
```
"math"
```

Before we print the tree, we label the leaves by calling `labelLeaves` on the root with a node counter. If desired, we also label the internal nodes. Then we print the tree by calling the `String` method on its root.

155d ⟨*Print tree, Ch. 16* 155d⟩≡ (152c)
```
nc := 0
nc = labelLeaves(root, nc)
if *optL {
        nc = 0
        labelInternalNodes(root, nc)
}
fmt.Println(root)
```

We import `fmt`.

155e ⟨*Imports, Ch. 16* 151a⟩+≡ (150a) ◁155c 156b▷
```
"fmt"
```

We label the leaves.

156a ⟨*Functions, Ch. 16* 154c⟩+≡ (150a) ◁155a 156c▷
```
func labelLeaves(v *nwk.Node, nc int) int {
        if v == nil { return nc }
        nc = labelLeaves(v.Child, nc)
        if v.Child == nil {
                nc++
                v.Label = "T" + strconv.Itoa(nc)
        }
        nc = labelLeaves(v.Sib, nc)
        return nc
}
```

We import `strconv`.

156b ⟨*Imports, Ch. 16* 151a⟩+≡ (150a) ◁155e
```
"strconv"
```

And we label the internal nodes.

156c ⟨*Functions, Ch. 16* 154c⟩+≡ (150a) ◁156a
```
func labelInternalNodes(v *nwk.Node, nc int) int {
        if v == nil { return nc }
        if v.Child != nil {
                nc++
                v.Label = "N" + strconv.Itoa(nc)
        }
        nc = labelInternalNodes(v.Child, nc)
        nc = labelInternalNodes(v.Sib, nc)
        return nc
}
```

We have finished `genTree`, let's test it.

Testing

The outline of our testing program contains hooks for imports and the testing logic.

156d ⟨*genTree_test.go* 156d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 16 157b⟩
)

func TestGenTree(t *testing.T) {
        ⟨Testing, Ch. 16 157a⟩
}
```

We construct a set of tests and iterate over them.

157a ⟨*Testing, Ch. 16* 157a⟩≡ (156d)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 16 157c⟩
for i, test := range tests {
        ⟨Run test, Ch. 16 157e⟩
}
```

We import `exec`.

157b ⟨*Testing imports, Ch. 16* 157b⟩≡ (156d) 158 ▷
```
"os/exec"
```

There are seven program-specific options. We seed the random number generator in every test, which leaves six options to test, plus a test without any options. So we have seven tests in total. We begin by testing all defaults, absolute branch lengths (`-a`), coalescent (`-c`), and iterations (`-i`).

157c ⟨*Construct tests, Ch. 16* 157c⟩≡ (157a) 157d ▷
```
test := exec.Command("./genTree", "-s", "13")
tests = append(tests, test)
test = exec.Command("./genTree", "-s", "13", "-a")
tests = append(tests, test)
test = exec.Command("./genTree", "-s", "13", "-c")
tests = append(tests, test)
test = exec.Command("./genTree", "-s", "13", "-i", "2")
tests = append(tests, test)
```

The second set of tests contains labels for internal branches (`-l`), sample size (`-n`), and population mutation rate (`-t`).

157d ⟨*Construct tests, Ch. 16* 157c⟩+≡ (157a) ◁ 157c
```
test = exec.Command("./genTree", "-s", "13", "-l")
tests = append(tests, test)
test = exec.Command("./genTree", "-s", "13", "-n", "9")
tests = append(tests, test)
test = exec.Command("./genTree", "-s", "13", "-t", "500")
```

For each test we compare what we get with what we want, which is stored in files labeled `r1.txt`, `r2.txt`, and so on.

157e ⟨*Run test, Ch. 16* 157e⟩≡ (157a)
```
get, err := test.Output()
if err != nil {
        t.Errorf("couldn't run %q", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
        t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

158 ⟨*Testing imports, Ch. 16* 157b⟩+≡ (156d) ◁157b

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 17

Program `getSeq`: Get Sequence

Introduction

The program `getSeq` gets sequences from a data stream whose headers match a regular expression.

Implementation

The program outline contains hooks for imports, variables, functions, and the guts of the main function.

160a ⟨*getSeq.go* 160a⟩≡
```
package main

import (
        ⟨Imports, Ch. 17 160c⟩
)
⟨Variables, Ch. 17 161a⟩
⟨Functions, Ch. 17 162a⟩
func main() {
        ⟨Main function, Ch. 17 160b⟩
}
```

In the main function we prepare the `log` package, set a usage message and parse the options set by the user. We then compile the regular expression and use it to get the matching sequences.

160b ⟨*Main function, Ch. 17* 160b⟩≡ (160a)
```
util.PrepLog("getSeq")
⟨Set usage, Ch. 17 160d⟩
⟨Parse options, Ch. 17 161c⟩
⟨Compile regex, Ch. 17 161e⟩
⟨Get matching sequences, Ch. 17 161g⟩
```

We import `util`.

160c ⟨*Imports, Ch. 17* 160c⟩≡ (160a) 160e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage message consists of the usage proper, an explanation of the program's purpose, and an example of its application.

160d ⟨*Set usage, Ch. 17* 160d⟩≡ (160b)
```
u := "getSeq [-h] [options] regex [files]"
p := "Extract sequences with headers matching a regex."
e := "getSeq \"coli*\" *.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

160e ⟨*Imports, Ch. 17* 160c⟩+≡ (160a) ◁ 160c 161b ▷
```
"github.com/evolbioinf/clio"
```

In response to `-c`, the program prints the *complement*, that is, all sequences that are not matching. Apart from that, there is the usual `-v` option.

161a ⟨*Variables, Ch. 17* 161a⟩≡ (160a)
```
var optC = flag.Bool("c", false, "get complement")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

161b ⟨*Imports, Ch. 17* 160c⟩+≡ (160a) ◁160e 161d▷
```
"flag"
```

After parsing the options, we make sure a regular expression was provided, and test for `-v`.

161c ⟨*Parse options, Ch. 17* 161c⟩≡ (160b)
```
flag.Parse()
if len(flag.Args()) < 1 {
        fmt.Fprintf(os.Stderr, "please provide a regular expression\n")
        os.Exit(0)
}
if *optV {
        util.PrintInfo("getSeq")
}
```

We import `fmt` and `os`.

161d ⟨*Imports, Ch. 17* 160c⟩+≡ (160a) ◁161b 161f▷
```
"fmt"
"os"
```

and compile the regular expression.

161e ⟨*Compile regex, Ch. 17* 161e⟩≡ (160b)
```
rs := flag.Args()[0]
r, err := regexp.Compile(rs)
if err != nil {
        log.Fatalf("Could not compile %q.\n", rs)
}
```

We import the packages `regexp` and `log`.

161f ⟨*Imports, Ch. 17* 160c⟩+≡ (160a) ◁161d 162b▷
```
"regexp"
"log"
```

To get the matching sequences, the input files are parsed with the function `scan`, which takes the regex and the complement marker as input.

161g ⟨*Get matching sequences, Ch. 17* 161g⟩≡ (160b)
```
files := flag.Args()[1:]
clio.ParseFiles(files, scan, r, *optC)
```

Before a file is scanned, we retrieve the arguments just passed. Then we go through it line by line and open or close the printing channel. After scanning the file we flush the scanner.

162a ⟨*Functions, Ch. 17* 162a⟩≡ (160a)
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 17 162c⟩
        var open bool
        sc := fasta.NewScanner(r)
        for sc.ScanLine() {
                l := sc.Line()
                ⟨Deal with header, Ch. 17 162d⟩
                if open { fmt.Println(string(l)) }
        }
        ⟨Flush scanner, Ch. 17 162e⟩
}
```

Import `io`, `fasta`, and `fmt`.

162b ⟨*Imports, Ch. 17* 160c⟩+≡ (160a) ◁161f
```
"io"
"github.com/evolbioinf/fasta"
"fmt"
```

There are two arguments to retrieve, the regular expression and the complement indicator.

162c ⟨*Retrieve arguments, Ch. 17* 162c⟩≡ (162a)
```
re := args[0].(*regexp.Regexp)
optC := args[1].(bool)
```

When a header is found, there are four possible combinations of finding a match and being asked for the complement:

match	complement	print
yes	yes	no
yes	no	yes
no	yes	yes
no	no	no

162d ⟨*Deal with header, Ch. 17* 162d⟩≡ (162a)
```
if sc.IsHeader() {
        m := re.Find(l[1:])
        if m != nil && optC { open = false
        } else if m != nil && !optC { open = true
        } else if m == nil && optC { open = true
        } else {open = false}
}
```

We flush the scanner and print any remaining bytes.

162e ⟨*Flush scanner, Ch. 17* 162e⟩≡ (162a)
```
l := sc.Flush()
if open && len(l) > 0 {
        fmt.Println(string(sc.Flush()))
}
```

This completes `getSeq`, time to test it.

Testing

We set up the testing framework.

163a ⟨*getSeq_test.go* 163a⟩≡
```
package main
import (
        "testing"
        ⟨Testing imports, Ch. 17 163c⟩
)
func TestGetSeq(t *testing.T) {
        ⟨Testing, Ch. 17 163b⟩
}
```

We test on the file `test.fasta`. It contains the ten sequences, Seq1, Seq2,..., Seq10. We begin by matching "Seq1".

163b ⟨*Testing, Ch. 17* 163b⟩≡ (163a) 163d▷
```
cmd := exec.Command("./getSeq", "Seq1", "test.fasta")
o, err := cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
```

We import `exec`.

163c ⟨*Testing imports, Ch. 17* 163c⟩≡ (163a) 163e▷
```
"os/exec"
```

This should retrieve two sequences, Seq1 and Seq10, which are contained in `res1.txt`.

163d ⟨*Testing, Ch. 17* 163b⟩+≡ (163a) ◁163b 163f▷
```
e, err := ioutil.ReadFile("res1.txt")
if !bytes.Equal(o, e) {
        t.Errorf("want:\n%s\nget:\n%s\n", e, o)
}
```

We import `ioutil` and `bytes`.

163e ⟨*Testing imports, Ch. 17* 163c⟩+≡ (163a) ◁163c
```
"io/ioutil"
"bytes"
```

Now we retrieve "Seq1" alone. It is contained in `res2.txt`.

163f ⟨*Testing, Ch. 17* 163b⟩+≡ (163a) ◁163d 164a▷
```
cmd = exec.Command("./getSeq", "1$", "test.fasta")
o, err = cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
e, err = ioutil.ReadFile("res2.txt")
if !bytes.Equal(o, e) {
        t.Errorf("want:\n%s\nget:\n%s\n", e, o)
}
```

Retrieve Seq1, Seq2, and Seq3 using a character set. The expected result is in `res3.txt`.

164a ⟨*Testing, Ch. 17* 163b⟩+≡ (163a) ◁163f 164b ▷

```
cmd = exec.Command("./getSeq", "[123]$", "test.fasta")
o, err = cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
e, err = ioutil.ReadFile("res3.txt")
if !bytes.Equal(o, e) {
        t.Errorf("want:\n%s\nget:\n%s\n", e, o)
}
```

Finally, we test the complement option.

164b ⟨*Testing, Ch. 17* 163b⟩+≡ (163a) ◁164a

```
cmd = exec.Command("./getSeq", "-c", "[123]$",
        "test.fasta")
o, err = cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
e, err = ioutil.ReadFile("res4.txt")
if !bytes.Equal(o, e) {
        t.Errorf("want:\n%s\nget:\n%s\n", e, o)
}
```

Chapter 18

Program `histogram`: Compute Histogram

Table 18.1: Example input (**A**) and output (**B**) of `histogram`.

A	**B**	
9	0	0
2	0	0.005
2	1	0.005
2	1	0
4	1	0.026
...	...	

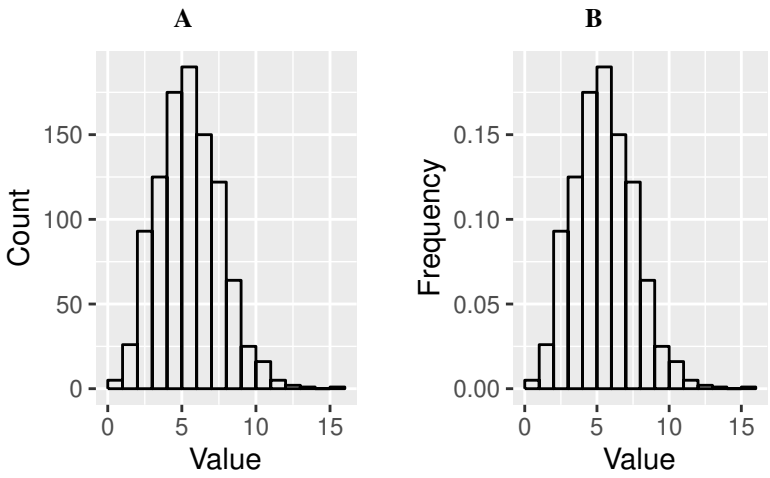


Figure 18.1: Plotting counts (**A**) and frequencies (**B**) of the example data in Table 18.1A using `plotLine`.

Introduction

Given a list of numbers, we'd often like to visualize their frequency distribution. The program `histogram` reads a column of floating point numbers and prints their frequency distribution as pairs of x/y values that can then be plotted using, for example, `plotLine` (Ch. 34). Table 18.1A shows some abridged example input and Table 18.1B the corresponding output. The default output is raw counts, which is plotted in Figure 18.1A. The user can also opt to plot frequencies, shown in Figure 18.1B.

The default number of bins, k, is computed using Sturge's rule,

$$k = 1 + 3.322 \log(n), \tag{18.1}$$

but the user can set the number of bins. The default range starts at the floor of the minimum input value and ends at the floor of the maximum value plus 1. Again, the user is free to set a different range.

Implementation

The outline of `histogram` provides hooks for imports, functions, and the logic of the main function.

167a ⟨*histogram.go* 167a⟩≡
```
package main

import (
        ⟨Imports, Ch. 18 167c⟩
)

⟨Functions, Ch. 18 169a⟩
func main() {
        ⟨Main function, Ch. 18 167b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

167b ⟨*Main function, Ch. 18* 167b⟩≡ (167a)
```
util.PrepLog("histogram")
⟨Set usage, Ch. 18 167d⟩
⟨Declare options, Ch. 18 167f⟩
⟨Parse options, Ch. 18 168b⟩
⟨Parse input files, Ch. 18 168f⟩
```

We import `util`.

167c ⟨*Imports, Ch. 18* 167c⟩≡ (167a) 167e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the program's purpose, and an example command.

167d ⟨*Set usage, Ch. 18* 167d⟩≡ (167b)
```
u := "histogram [-h] [option]... [foo.dat]... | plotLine"
p := "Convert a column of numbers to histogram coordinates."
e := "histogram -b 20 foo.dat"
clio.Usage(u, p, e)
```

We import `clio`.

167e ⟨*Imports, Ch. 18* 167c⟩+≡ (167a) ◁ 167c 168a ▷
```
"github.com/evolbioinf/clio"
```

Apart from the version, we declare three program-specific options, the number of bins, the range, and whether frequencies should be printed instead of the default raw counts.

167f ⟨*Declare options, Ch. 18* 167f⟩≡ (167b)
```
var optV = flag.Bool("v", false, "version")
var optB = flag.Int("b", 0, "number of bins")
var optR = flag.String("r", "xmin:xmax", "range")
var optF = flag.Bool("f", false, "print frequencies")
```

We import `flag`.

168a ⟨*Imports, Ch. 18* 167c⟩+≡ (167a) ◁167e 168e▷
```
"flag"
```

We parse the options and first respond to `-v`, as this terminates the program. Then we respond to the range option, `-r`.

168b ⟨*Parse options, Ch. 18* 168b⟩≡ (167b)
```
flag.Parse()
⟨Respond to -v, Ch. 18 168c⟩
⟨Respond to -r, Ch. 18 168d⟩
```

We respond to `-v` by printing standardized information about `histogram`.

168c ⟨*Respond to -v, Ch. 18* 168c⟩≡ (168b)
```
if *optV {
        util.PrintInfo("histogram")
}
```

If the user set a range, we store its values.

168d ⟨*Respond to -r, Ch. 18* 168d⟩≡ (168b)
```
fields := strings.Split(*optR, ":")
var xmin, xmax float64
var err error
if fields[0] != "xmin" && fields[1] != "xmax" {
        xmin, err = strconv.ParseFloat(fields[0], 64)
        if err != nil { log.Fatal("broken range") }
        xmax, err = strconv.ParseFloat(fields[1], 64)
        if err != nil { log.Fatal("broken range") }
}
```

We import `strings`.

168e ⟨*Imports, Ch. 18* 167c⟩+≡ (167a) ◁168a 169b▷
```
"strings"
```

The remaining tokens on the command line are interpreted as the names of input files. Each of these files is now analyzed with the function `scan`, which takes as arguments options for the number of bins, the range, and whether or not frequencies are requested.

168f ⟨*Parse input files, Ch. 18* 168f⟩≡ (167b)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optB, xmin, xmax, *optF)
```

Inside `scan`, we retrieve the arguments passed and read the data into a slice of floats. Then we calculate the number of bins, their ranges, and their counts. Taking our cue from [11, p. 313ff], `counts[i]` is the number of values between `ranges[i]` and `ranges[i+1]`. The lower boundary is included, the upper excluded. Using `counts` and `ranges`, we write the histogram of counts or frequencies.

169a ⟨*Functions, Ch. 18* 169a⟩≡ (167a)

```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 18 169c⟩
        var data []float64
        ⟨Read data, Ch. 18 169d⟩
        ⟨Calculate number of bins, Ch. 18 170a⟩
        ⟨Calculate ranges, Ch. 18 170b⟩
        ⟨Calculate counts or frequencies, Ch. 18 171a⟩
        ⟨Write histogram, Ch. 18 172a⟩
}
```

We import `io`.

169b ⟨*Imports, Ch. 18* 167c⟩+≡ (167a) ◁168e 169e▷

```
"io"
```

The number of bins and the minimum and maximum x-values are retrieved through type assertion.

169c ⟨*Retrieve arguments, Ch. 18* 169c⟩≡ (169a)

```
numBins := args[0].(int)
xmin := args[1].(float64)
xmax := args[2].(float64)
printFreq := args[3].(bool)
```

We scan the input, convert numbers from string to float, and store them.

169d ⟨*Read data, Ch. 18* 169d⟩≡ (169a)

```
sc := bufio.NewScanner(r)
for sc.Scan() {
        ns := strings.Fields(sc.Text())[0]
        f, err := strconv.ParseFloat(ns, 64)
        if err != nil {
                log.Fatal("malformed input")
        }
        data = append(data, f)
}
```

We import `bufio`, `strconv` and `log`.

169e ⟨*Imports, Ch. 18* 167c⟩+≡ (167a) ◁169b 170c▷

```
"bufio"
"strconv"
"log"
```

If the user didn't set the number of bins, we compute it from equation (18.1).

170a ⟨*Calculate number of bins, Ch. 18* 170a⟩≡ (169a)
```
if numBins == 0 {
        l := len(data)
        nb := 1.0 + 3.322 * math.Log(float64(l))
        numBins = int(math.Round(nb))
}
```

If the user did not set an x-range, we determine it from the data. Since our subsequent binning step requires sorted data, we sort at this point, which gives us easy access to the minimum and maximum values.

170b ⟨*Calculate ranges, Ch. 18* 170b⟩≡ (169a)
```
sort.Float64s(data)
if xmin == xmax && xmin == 0.0 {
        ⟨Determine xmin and xmax, Ch. 18 170d⟩
}
⟨Set ranges, Ch. 18 170f⟩
```

We import `sort`.

170c ⟨*Imports, Ch. 18* 167c⟩+≡ (167a) ◁169e 170e▷
```
"sort"
```

Let m and a be the minimum and the maximum input values, then the minimum of x is floor(m) and the maximum floor(a).

170d ⟨*Determine xmin and xmax, Ch. 18* 170d⟩≡ (170b)
```
xmin = math.Floor(data[0])
xmax = math.Floor(data[len(data)-1]+1.0)
```

We import `math`.

170e ⟨*Imports, Ch. 18* 167c⟩+≡ (167a) ◁170c 172b▷
```
"math"
```

If there are n bins, there are $n + 1$ entries in `ranges`, the smallest being `xmin`, the largest `xmax`.

170f ⟨*Set ranges, Ch. 18* 170f⟩≡ (170b)
```
counts := make([]float64, numBins)
d := (xmax - xmin) / float64(numBins)
ranges := make([]float64, numBins + 1)
ranges[0] = xmin
for i := 1; i <= numBins; i++ {
        ranges[i] = ranges[i-1] + d
}
```

To calculate the counts, we find the start of the range and then count the entries in each bin. Then we calculate the frequencies, if desired.

171a ⟨*Calculate counts or frequencies, Ch. 18* 171a⟩≡ (169a)

```
⟨Find start of range, Ch. 18 171b⟩
i := 0
for j, _ := range counts {
        for i < len(data) && data[i] < ranges[j+1] {
                counts[j]++
                i++
        }
}
if printFreq {
        ⟨Calculate frequencies, Ch. 18 171c⟩
}
```

We make sure the first element in `data` is an element of the first bin.

171b ⟨*Find start of range, Ch. 18* 171b⟩≡ (171a)

```
for i, d := range data {
        if d >= ranges[0] {
                data = data[i:]
                break
        }
}
```

We sum the counts and divide the counts to get the frequencies.

171c ⟨*Calculate frequencies, Ch. 18* 171c⟩≡ (171a)

```
s := 0.0
for _, c := range counts {
        s += c
}
for i, c := range counts {
        counts[i] = c / s
}
```

The histogram consists of two columns of x/y data, which we write with a tabwriter. Let c be the count for a bin, and x_1, x_2 its boundaries. Then we represent each bar in the histogram by three points,

$$\begin{array}{ll} x_1 & 0 \\ x_1 & c \\ x_2 & c \end{array}$$

This leaves the last bar without a line on its right, and all bars open at the bottom. We fix this in a finishing step.

172a ⟨*Write histogram, Ch. 18* 172a⟩≡ (169a)
```
w := tabwriter.NewWriter(os.Stdout, 2, 1, 2, ' ', 0)
for i, c := range counts {
        x1 := ranges[i]
        x2 := ranges[i+1]
        y := c
        fmt.Fprintf(w, "%g\t0\n", x1)
        fmt.Fprintf(w, "%g\t%g\n", x1, y)
        fmt.Fprintf(w, "%g\t%g\n", x2, y)
}
⟨Finish plot, Ch. 18 172c⟩
w.Flush()
```

We import `os`, `tabwriter` and `fmt`.

172b ⟨*Imports, Ch. 18* 167c⟩+≡ (167a) ◁170e
```
"os"
"text/tabwriter"
"fmt"
```

To finish the plot, we close the last bar, and draw a bottom line.

172c ⟨*Finish plot, Ch. 18* 172c⟩≡ (172a)
```
x1 := ranges[0]
x2 := ranges[len(ranges)-1]
fmt.Fprintf(w, "%.3g\t0\n", x2)
fmt.Fprintf(w, "%.3g\t0\n", x1)
```

The program `histogram` is finished, time to test it.

Testing

The outline of our testing code has hooks for imports and the testing logic.

172d ⟨*histogram_test.go* 172d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 18 173b⟩
)

func TestHistogram(t *testing.T) {
        ⟨Testing, Ch. 18 173a⟩
}
```

We construct the tests and then iterate over them.

173a ⟨*Testing, Ch. 18* 173a⟩≡ (172d)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 18 173c⟩
for i, test := range tests {
        ⟨Run test, Ch. 18 173d⟩
}
```

We import `exec`.

173b ⟨*Testing imports, Ch. 18* 173b⟩≡ (172d) 173e ▷
```
"os/exec"
```

We run `histogram` twice on our test data in `test.dat`. The first run returns counts, the second frequencies. In both cases we set the range and the number of bins.

173c ⟨*Construct tests, Ch. 18* 173c⟩≡ (173a)
```
test := exec.Command("./histogram", "-r", "0:16", "-b", "16",
        "test.dat")
tests = append(tests, test)
test = exec.Command("./histogram", "-r", "0:16", "-b", "16",
        "-f", "test.dat")
tests = append(tests, test)
```

For each test we compare what we get with what we want, which is stored in `r1.txt` and `r2.txt`.

173d ⟨*Run test, Ch. 18* 173d⟩≡ (173a)
```
get, err := test.Output()
if err != nil { t.Error(err.Error()) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Error(err.Error()) }
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

173e ⟨*Testing imports, Ch. 18* 173b⟩+≡ (172d) ◁ 173b
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 19

Program `huff`: Huffman Encoding

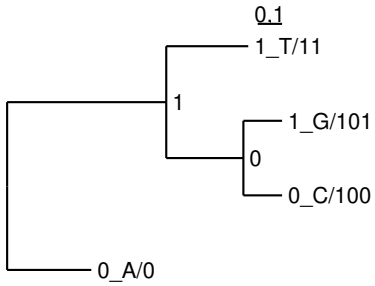


Figure 19.1: Huffman tree of the *M. genitalium* genome sequence.

A

```
>gi|84626123|gb|L43967.2| Mycoplasma genitalium...
TAAGTTATTATTTAGTTAATACTTTTAACAATATTATTAAGGTATTTAAA
AAATACTATTATAGTATTTAACATAGTTAAATACCTTCCTTAATACTGTT
```

B

```
>gi|84626123|gb|L43967.2| Mycoplasma genitalium...
11001011111011110111111010111110011010011111111001
00001101111011110010110111011111100000011010011011
11011010111011111100100011010111110001101001001111
10010011110011010011101111
```

Figure 19.2: The first 100 nuceleotides of the genome file of *M. genitalium* printed as nucleotides (**A**) and as binary codes (**B**) according to the Huffman tree in Figure 19.1.

Introduction

Given a Huffman tree, the program `huff` encodes input sequences into their binary representation. For example, Figure 19.1 shows the Huffman tree for the genome of *M. genitalium* calculated with the program `hut`. Given this tree, `huff` converts the DNA sequence in Figure 19.2A into the string representation of a bit stream in Figure 19.2B. `huff` can also reverse this step and decode such a stream of zeros and ones.

Implementation

Our outline of `huff` contains hooks for imports, functions, and the logic of the main function.

175 ⟨*huff.go* 175⟩≡

```
package main

import (
        ⟨Imports, Ch. 19 176b⟩
)

⟨Functions, Ch. 19 177e⟩
```

```
func main() {
        ⟨Main function, Ch. 19 176a⟩
}
```

In the main function we prepare the `log` package, set the usage of `huff`, declare its options, parse the options, and parse the input files.

176a ⟨*Main function, Ch. 19* 176a⟩≡ (175)
```
util.PrepLog("huff")
⟨Set usage, Ch. 19 176c⟩
⟨Declare options, Ch. 19 176e⟩
⟨Parse options, Ch. 19 176g⟩
⟨Parse input files, Ch. 19 177a⟩
```

We import `util`.

176b ⟨*Imports, Ch. 19* 176b⟩≡ (175) 176d ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `huff`, and an example command.

176c ⟨*Set usage, Ch. 19* 176c⟩≡ (176a)
```
u := "huff [-h] [option]... [file]..."
p := "Convert residue sequences to bit sequences given " +
        "a Huffman tree computed with hut."
e := "hut foo.fasta > foo.nwk; huff foo.nwk foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

176d ⟨*Imports, Ch. 19* 176b⟩+≡ (175) ◁176b 176f ▷
```
"github.com/evolbioinf/clio"
```

We declare two options, version (`-v`), and decoding (`-d`).

176e ⟨*Declare options, Ch. 19* 176e⟩≡ (176a)
```
var optV = flag.Bool("v", false, "version")
var optD = flag.Bool("d", false, "decode")
```

We import `flag`.

176f ⟨*Imports, Ch. 19* 176b⟩+≡ (175) ◁176d 177b ▷
```
"flag"
```

We parse the options and respond to `-v`, as this stops `huff`.

176g ⟨*Parse options, Ch. 19* 176g⟩≡ (176a)
```
flag.Parse()
if *optV {
        util.PrintInfo("huff")
}
```

We interpret the remaining tokens on the command line as file names. The first of these is assumed to be the name of the file containing the code tree. If it doesn't exist, we bail asking for a tree file. If it does exist, we iterate over the trees.

177a ⟨*Parse input files, Ch. 19* 177a⟩≡ (176a)
```
files := flag.Args()
if len(files) == 0 {
        m := "please provide a file containing one " +
                "or more code trees computed with hut"
        log.Fatal(m)
}
⟨Iterate over code trees, Ch. 19 177c⟩
```

We import `log`.

177b ⟨*Imports, Ch. 19* 176b⟩+≡ (175) ◁176f 177d▷
```
"log"
```

For each tree we scan the sequence files using the function `scan`. The function `scan` takes as argument the code tree, represented by its root, and the decoding option.

177c ⟨*Iterate over code trees, Ch. 19* 177c⟩≡ (177a)
```
tf, err := os.Open(files[0])
if err != nil { log.Fatalf("cannot open %q", files[0]) }
defer tf.Close()
files = files[1:]
sc := nwk.NewScanner(tf)
for sc.Scan() {
        root := sc.Tree()
        clio.ParseFiles(files, scan, root, *optD)
}
```

We import `os` and `nwk`.

177d ⟨*Imports, Ch. 19* 176b⟩+≡ (175) ◁177b 177f▷
```
"os"
"github.com/evolbioinf/nwk"
```

Inside `scan`, we retrieve the two arguments just passed and deal with each sequence in the file.

177e ⟨*Functions, Ch. 19* 177e⟩≡ (175) 179a▷
```
func scan(r io.Reader, args ...interface{}) {
        root := args[0].(*nwk.Node)
        dec := args[1].(bool)
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                seq := sc.Sequence()
                ⟨Deal with sequence, Ch. 19 178a⟩
        }
}
```

We import `io` and `fasta`.

177f ⟨*Imports, Ch. 19* 176b⟩+≡ (175) ◁177d 178b▷
```
"io"
"github.com/evolbioinf/fasta"
```

Sequences consist of headers and data. Both are transformed depending on whether the sequence is being decoded or encoded. The header is derived from the current header by appending `huff` with or without the decoding switch. The actual sequence data requires a bit more thought. When we are done with the data, we wrap it together with the header in a new sequence and print it.

178a ⟨*Deal with sequence, Ch. 19* 178a⟩≡ (177e)

```
header := seq.Header() + " - huff"
var data []byte
if dec {
        header += " -d"
        ⟨Decode sequence, Ch. 19 178c⟩
} else {
        ⟨Encode sequence, Ch. 19 180a⟩
}
seq = fasta.NewSequence(header, data)
fmt.Println(seq)
```

We import `fmt`.

178b ⟨*Imports, Ch. 19* 176b⟩+≡ (175) ◁177f

```
"fmt"
```

To decode a sequence, we first extract the decoder from the code tree. Then we iterate over the zeros and ones in the original data. These are part of a code, which we read and for which we store the corresponding byte.

178c ⟨*Decode sequence, Ch. 19* 178c⟩≡ (178a)

```
⟨Extract decoder, Ch. 19 178d⟩
od := seq.Data()
i := 0
for i < len(od) {
        ⟨Read code, Ch. 19 179b⟩
        ⟨Store decoded character, Ch. 19 179d⟩
}
```

The decoder is a map between leaf IDs and bytes. The leaf representing a code is discovered through a traversal of the code tree, which we delegate to a call to the function `extractDecoder`.

178d ⟨*Extract decoder, Ch. 19* 178d⟩≡ (178c)

```
i2c := make(map[int]byte)
i2c = extractDecoder(root, i2c)
```

Inside `extractDecoder`, we concentrate on the leaves. A leaf is labeled by a zero or a one, followed by an underscore, the encoded character, a slash, and the actual code. We summarize this pattern as

```
[0|1]_c/011
```

where `c` is the character the leaf stands for. We extract that character and store it as a function of the leaf ID.

179a ⟨*Functions, Ch. 19* 177e⟩+≡ (175) ◁177e 179c▷

```
func extractDecoder(v *nwk.Node, i2c map[int]byte) map[int]byte {
        if v == nil { return i2c }
        if v.Child == nil {
                i2c[v.Id] = v.Label[2]
        }
        i2c = extractDecoder(v.Child, i2c)
        i2c = extractDecoder(v.Sib, i2c)
        return i2c
}
```

We read a code by walking into the bit array and the code tree until the function `search` returns a nil node or we run out of "bits".

179b ⟨*Read code, Ch. 19* 179b⟩≡ (178c)

```
id := -1
v := root
for v != nil && i < len(od) {
        v, id = search(v, od[i])
        if v != nil { i++ }
}
```

Inside `search`, we might have reached a leaf, in which case we return a nil node and the ID. Otherwise, we look for the child with the bit passed. As we've already seen, that bit is always represented as the first byte in a node label.

179c ⟨*Functions, Ch. 19* 177e⟩+≡ (175) ◁179a 180b▷

```
func search(v *nwk.Node, b byte) (*nwk.Node, int) {
        if v.Child == nil { return nil, v.Id }
        if v.Child.Label[0] == b {
                return v.Child, v.Child.Id
        } else {
                return v.Child.Sib, v.Child.Sib.Id
        }
}
```

The leaf ID we've just found corresponds to a character that we append to the new sequence data. When looking up this character, we check we actually found a mapping and bail otherwise.

179d ⟨*Store decoded character, Ch. 19* 179d⟩≡ (178c)

```
c, ok := i2c[id]
if !ok {
        log.Fatalf("couldn't decode leaf %d", id)
}
data = append(data, c)
```

We're done decoding. To *encode* a sequence, we again traverse the code tree, this time using the function `extractEncoder`. The encoder it returns is a map between a byte and a byte slice. Then we iterate over the bytes in the original data and store the corresponding bits.

180a ⟨*Encode sequence, Ch. 19* 180a⟩≡ (178a)
```
byte2bits := make(map[byte][]byte)
byte2bits = extractEncoder(root, byte2bits)
od := seq.Data()
for _, b := range od {
        code := byte2bits[b]
        data = append(data, code...)
}
```

In `extractEncoder` we seek out the leaves again. Whenever we find one, we store its code as a function of its character.

180b ⟨*Functions, Ch. 19* 177e⟩+≡ (175) ◁179c
```
func extractEncoder(v *nwk.Node,
        b2b map[byte][]byte) map[byte][]byte {
        if v == nil { return b2b }
        if v.Child == nil {
                ⟨Store code, Ch. 19 180c⟩
        }
        b2b = extractEncoder(v.Child, b2b)
        b2b = extractEncoder(v.Sib, b2b)
        return b2b
}
```

The character encoded is the third byte in the label, the code starts at the fifth byte.

180c ⟨*Store code, Ch. 19* 180c⟩≡ (180b)
```
c := v.Label[2]
code := []byte(v.Label[4:])
b2b[c] = code
```

We've finished `huff`, let's test it.

Testing

Our testing code has hooks for imports and the testing logic.

180d ⟨*huff_test.go* 180d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 19 181b⟩
)

func TestHuff(t *testing.T) {
        ⟨Testing, Ch. 19 181a⟩
}
```

We construct our tests and run them.

181a ⟨*Testing, Ch. 19* 181a⟩≡ (180d)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 19 181c⟩
for i, test := range tests {
        ⟨Run test, Ch. 19 181d⟩
}
```

We import `exec`.

181b ⟨*Testing imports, Ch. 19* 181b⟩≡ (180d) 181e▷
```
"os/exec"
```

There are two tests. One for default encoding, the other for decoding. The decoding is applied to the output of the encoding.

181c ⟨*Construct tests, Ch. 19* 181c⟩≡ (181a)
```
test := exec.Command("./huff", "mght.nwk", "test.fasta")
tests = append(tests, test)
test = exec.Command("./huff", "-d", "mght.nwk", "r1.txt")
tests = append(tests, test)
```

We store the result we get from the test and compare it to the result we want, which is stored in files `r1.txt` and `r2.txt`.

181d ⟨*Run test, Ch. 19* 181d⟩≡ (181a)
```
get, err := test.Output()
if err != nil {
        t.Errorf("couldn't run %s", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
        t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

181e ⟨*Testing imports, Ch. 19* 181b⟩+≡ (180d) ◁181b
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 20

Program `hut`: Calculate Huffman tree

A

Nuc.	Freq.
C	0.158
G	0.159
T	0.337
A	0.346

B

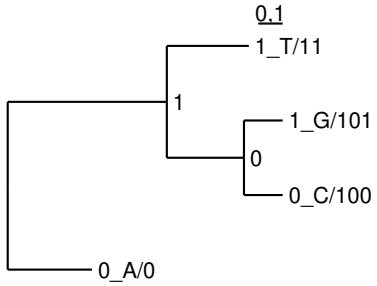


Figure 20.1: Nucleotide frequencies in the genome of *M. genitalium* (**A**) and the corresponding Huffman tree (**B**).

Introduction

Characters typically take up one byte of computer memory. A byte consists of 8 bits, enough to encode the 256 characters of the extended ASCII code. This space requirement of eight bits per character applies even for texts over alphabets much smaller than the ASCII code, or where characters frequencies vary significantly. Huffman [18] first proposed a method for finding variable length codes that occupy the smallest amount of space possible. These codes are represented in the form of binary trees, Huffman trees.

A Huffman tree is constructed starting with the leaves, which contain the characters to be encoded and their frequencies [6, p. 385ff]. The frequencies are interpreted as node weights, and the two lightest leaves are merged into a parent whose weight is the sum of its children's weights. This procedure is repeated until the last two nodes are merged into the root.

To read optimal codes from such a tree, the incoming edge for each node is labeled. Edges leading to a left child are labeled 0, edges leading to a right child are labeled 1. Now the optimal code for the character of a leaf consists of its path label.

The program `hut` reads sequences and prints their Huffman trees. For example, Figure 20.1A shows the nucleotide frequencies in the genome of *M. genitalium*. When `hut` is applied to the genome sequence of *M. genitalium*, it returns the tree in Figure 20.1B, where branch lengths are proportional to node weights. Instead of a tree, `hut` can also just compute the number of bits required to encode the sequence.

Implementation

The outline of `hut` contains hooks for imports, types, methods, functions, and the logic of the main function.

183 ⟨*hut.go* 183⟩≡

```
package main

import (
        ⟨Imports, Ch. 20 184b⟩
)
⟨Types, Ch. 20 187c⟩
```

```
⟨Methods, Ch. 20 187d⟩
⟨Functions, Ch. 20 185a⟩
func main() {
        ⟨Main function, Ch. 20 184a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

184a ⟨*Main function, Ch. 20* 184a⟩≡ (183)
```
util.PrepLog("hut")
⟨Set usage, Ch. 20 184c⟩
⟨Declare options, Ch. 20 184e⟩
⟨Parse options, Ch. 20 184g⟩
⟨Parse input files, Ch. 20 184h⟩
```

We import `util`.

184b ⟨*Imports, Ch. 20* 184b⟩≡ (183) 184d ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `hut`, and an example command.

184c ⟨*Set usage, Ch. 20* 184c⟩≡ (184a)
```
m := "hut [-h] [option]... [file]..."
p := "Convert sequences into their Huffman trees."
e := "hut foo.fasta"
clio.Usage(m, p, e)
```

We import `clio`.

184d ⟨*Imports, Ch. 20* 184b⟩+≡ (183) ◁184b 184f ▷
```
"github.com/evolbioinf/clio"
```

There are two options to declare, the version, `-v`, and the bit computation, `-b`.

184e ⟨*Declare options, Ch. 20* 184e⟩≡ (184a)
```
var optV = flag.Bool("v", false, "version")
var optB = flag.Bool("b", false, "bits")
```

We import `flag`.

184f ⟨*Imports, Ch. 20* 184b⟩+≡ (183) ◁184d 185b ▷
```
"flag"
```

We parse the options and respond to `-v`.

184g ⟨*Parse options, Ch. 20* 184g⟩≡ (184a)
```
flag.Parse()
if *optV {
        util.PrintInfo("hut")
}
```

The remaining tokens on the command line are interpreted as input files, which we parse using the function `scan`. It takes as argument the bits option, `-b`.

184h ⟨*Parse input files, Ch. 20* 184h⟩≡ (184a)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optB)
```

Inside `scan` we retrieve the bits option and analyze the sequences in the file.

185a ⟨*Functions, Ch. 20* 185a⟩≡ (183) 188a▷
```
func scan(r io.Reader, args ...interface{}) {
        bits := args[0].(bool)
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                seq := sc.Sequence()
                ⟨Analyze sequence, Ch. 20 185c⟩
        }
}
```

We import `io` and `fasta`.

185b ⟨*Imports, Ch. 20* 184b⟩+≡ (183) ◁184f 186d▷
```
"io"
"github.com/evolbioinf/fasta"
```

A sequence is analyzed by counting the characters and constructing the Huffman tree.

To analyze a sequence, we count its characters and construct the Huffman tree. The tree is either printed, or we just print the number of bits it implies.

185c ⟨*Analyze sequence, Ch. 20* 185c⟩≡ (185a)
```
⟨Count characters, Ch. 20 185d⟩
⟨Construct tree, Ch. 20 186b⟩
if bits {
        ⟨Calculate bits, Ch. 20 189a⟩
        ⟨Print bits, Ch. 20 189d⟩
} else {
        ⟨Print tree, Ch. 20 189e⟩
}
```

To count the characters, we reserve space for counting the 256 possible characters. We can only build the desired binary tree from them if there are at least two characters. So we ensure that's the case.

185d ⟨*Count characters, Ch. 20* 185d⟩≡ (185c)
```
counts := make([]int, 256)
for _, c := range seq.Data() {
        counts[c]++
}
⟨Ensure at least two characters, Ch. 20 186a⟩
```

We count the distinct characters. If there's only one, we add a dummy character nucleotide.

186a ⟨*Ensure at least two characters, Ch. 20* 186a⟩≡ (185d)

```
n := 0
for _, count := range counts {
        if count > 0 { n++ }
}
if n == 1 {
        if counts['A'] == 0 {
                counts['A'] = 1
        } else {
                counts['C'] = 1
        }
}
```

We construct the tree in three steps. First, we construct the leaves. Then we cluster the leaves into the tree topology. Given the topology, we look up the character codes.

186b ⟨*Construct tree, Ch. 20* 186b⟩≡ (185c)

```
⟨Construct tree leaves, Ch. 20 186c⟩
⟨Construct tree topology, Ch. 20 187a⟩
⟨Construct character codes, Ch. 20 187f⟩
```

For each character with at least one occurrence we make a new leaf and store it in a slice of leaves. The labels of the incoming edges, 0 or 1, will later be written as the node label. So we keep the characters that label the leaves in a separate map referenced by their IDs.

186c ⟨*Construct tree leaves, Ch. 20* 186c⟩≡ (186b)

```
leaves := make([]*nwk.Node, 0)
labels := make(map[int]byte)
for i, c := range counts {
        if c > 0 {
                n := nwk.NewNode()
                n.Length = float64(c) / float64(len(seq.Data()))
                n.HasLength = true
                labels[n.Id] = byte(i)
                leaves = append(leaves, n)
        }
}
```

We import `nwk`.

186d ⟨*Imports, Ch. 20* 184b⟩+≡ (183) ◁185b 187b▷

```
"github.com/evolbioinf/nwk"
```

To construct the tree topology, we sort the leaves, label the two lightest 0 and 1, and merge them into a new node.

187a ⟨*Construct tree topology, Ch. 20* 187a⟩≡ (186b)
```
for len(leaves) > 1 {
        sort.Sort(leafSlice(leaves))
        leaves[0].Label = "0"
        leaves[1].Label = "1"
        ⟨Merge two lightest nodes, Ch. 20 187e⟩
}
root := leaves[0]
```

We import `sort`.

187b ⟨*Imports, Ch. 20* 184b⟩+≡ (183) ◁186d 188e▷
```
"sort"
```

To enable node sorting, we define the type `leafSlice`.

187c ⟨*Types, Ch. 20* 187c⟩≡ (183)
```
type leafSlice []*nwk.Node
```

We also implement the methods of the `Sort` interface, `Len`, `Less`, and `Swap` on `leafSlice`.

187d ⟨*Methods, Ch. 20* 187d⟩≡ (183)
```
func (l leafSlice) Len() int { return len(l) }
func (l leafSlice) Less(i, j int) bool {
        return l[i].Length < l[j].Length
}
func (l leafSlice) Swap(i, j int) {
        l[i], l[j] = l[j], l[i]
}
```

We merge the two lightest nodes into a new node. Its edge length is the sum of its child lengths. The new node is appended to the slice of leaves and those we just merged are sliced off.

187e ⟨*Merge two lightest nodes, Ch. 20* 187e⟩≡ (187a)
```
n := nwk.NewNode()
n.AddChild(leaves[0])
n.AddChild(leaves[1])
n.Length = leaves[0].Length + leaves[1].Length
n.HasLength = true
leaves = append(leaves, n)
leaves = leaves[2:]
```

Character codes reside in the leaves. To find them, we traverse the tree.

187f ⟨*Construct character codes, Ch. 20* 187f⟩≡ (186b)
```
traverse(root, labels)
```

Inside `traverse`, we seek out the leaves. For a given leaf, we initialize its code with the current label. Then we climb to the root to find the rest of the code in right-to-left orientation. We reverse the code before storing it as part of the leaf label.

188a ⟨*Functions, Ch. 20* 185a⟩+≡ (183) ◁185a 189c▷

```
func traverse(n *nwk.Node, labels map[int]byte) {
        if n == nil { return }
        if n.Child == nil {
                code := n.Label
                ⟨Climb to root, Ch. 20 188b⟩
                ⟨Reverse code, Ch. 20 188c⟩
                ⟨Store leaf label, Ch. 20 188d⟩
        }
        traverse(n.Child, labels)
        traverse(n.Sib, labels)
}
```

On our climb to the root we extend the code by the labels we encounter.

188b ⟨*Climb to root, Ch. 20* 188b⟩≡ (188a)

```
v := n.Parent
for v != nil {
        code += v.Label
        v = v.Parent
}
```

To reverse the code string, we convert it into a byte slice, reverse that, and convert the bytes back to a string.

188c ⟨*Reverse code, Ch. 20* 188c⟩≡ (188a)

```
b := []byte(code)
for i, j := 0, len(b)-1; i < j; i, j = i+1, j-1 {
        b[i], b[j] = b[j], b[i]
}
code = string(b)
```

We label a leaf for character `c` using a constant-length prefix followed by the variable-length code. The prefix consists of either a zero or a one, a hyphen, the character encoded, and a slash. In other words, the label follows the pattern

```
[0|1]-c/code
```

188d ⟨*Store leaf label, Ch. 20* 188d⟩≡ (188a)

```
n.Label = fmt.Sprintf("\"%s-%c/%s\"", n.Label, labels[n.Id], code)
```

We import `fmt`.

188e ⟨*Imports, Ch. 20* 184b⟩+≡ (183) ◁187b 189b▷

```
"fmt"
```

Given the code tree, we can calculate the number of bits required to encode the sequence. This is the product of the sequence length and the sum of leaf weights, which we round to the correct integer.

189a ⟨*Calculate bits, Ch. 20* 189a⟩≡ (185c)
```
sl := len(seq.Data())
lw := 0.0
lw = sumLeafWeights(root, lw)
nb := float64(sl) * lw
nb = math.Ceil(nb)
```

We import `math`.

189b ⟨*Imports, Ch. 20* 184b⟩+≡ (183) ◁188e
```
"math"
```

We traverse the code tree with `sumLeafWeights`. Whenever we find a leaf, we multiply its weight with the length of its code, which is the length of its label minus the length of the code prefix, 4, minus the two quotes that frame the label. The result gets added to the current weight and returned.

189c ⟨*Functions, Ch. 20* 185a⟩+≡ (183) ◁188a
```
func sumLeafWeights(v *nwk.Node, w float64) float64 {
        if v == nil { return w }
        x := 0.0
        if v.Child == nil {
                cl := float64(len(v.Label) - 6)
                w += v.Length * cl
        }
        w = sumLeafWeights(v.Child, w+x)
        w = sumLeafWeights(v.Sib, w+x)
        return w
}
```

We print the sequence header and the number of bits.

189d ⟨*Print bits, Ch. 20* 189d⟩≡ (185c)
```
fmt.Printf(">%s\n", seq.Header())
fmt.Printf("Bits: %d\n", int(nb))
```

Rather than printing the number of bits, the standard use case is to print the tree. We do this by applying a print function to the root.

189e ⟨*Print tree, Ch. 20* 189e⟩≡ (185c)
```
fmt.Println(root)
```

We've finished `hut`, time to test it.

Testing

The outline of our testing program has hooks for imports and the testing logic.

190a ⟨*hut_test.go* 190a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 20 190d⟩
)

func TestHut(t *testing.T) {
        ⟨Testing, Ch. 20 190b⟩
}
```

We construct the tests and run them.

190b ⟨*Testing, Ch. 20* 190b⟩≡ (190a)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 20 190c⟩
for i, test := range tests {
        ⟨Run test, Ch. 20 191a⟩
}
```

We apply `hut` to a short random sequence contained in `test.fasta` and construct two tests; one for the tree output and one for the bits output.

190c ⟨*Construct tests, Ch. 20* 190c⟩≡ (190b)
```
f := "test.fasta"
test := exec.Command("./hut", f)
tests = append(tests, test)
test = exec.Command("./hut", "-b", f)
tests = append(tests, test)
```

We import `exec`.

190d ⟨*Testing imports, Ch. 20* 190d⟩≡ (190a) 191b ▷
```
"os/exec"
```

We run the test and compare the result we get with the result we want, which is stored in the files `r1.txt` and `r2.txt`.

191a ⟨*Run test, Ch. 20* 191a⟩≡ (190b)

```
get, err := test.Output()
if err != nil {
          t.Errorf("couldn't run %q", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
          t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

191b ⟨*Testing imports, Ch. 20* 190d⟩+≡ (190a) ◁190d

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 21

Program `kerror`: k-Error Alignment

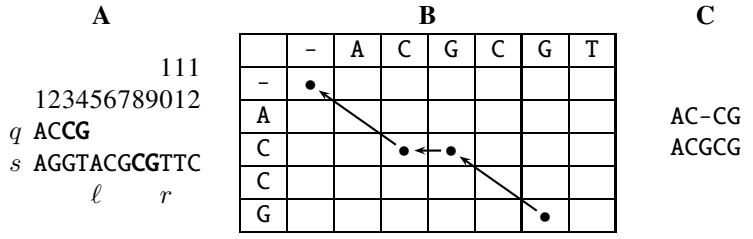


Figure 21.1: Example for picking the subject interval, $s[\ell...r]$ for aligning with the query, q with $k = 1$ using the exact match in bold (**A**); the corresponding dynamic programming matrix (**B**) with a possible trace back, and the final alignment (**C**).

Introduction

Blast is an algorithm for quickly finding local alignments between a short query and a potentially long subject. However, we might be interested in finding global rather than local alignments of the query in the subject. For example, the query might be a PCR primer and the subject the chromosome we'd like to scan for binding sites. So the alignment we are looking for is global in the query and local in the subject. One method to calculate this is known as k-error alignment [4], where an alignment can contain up to k gaps or mismatches.

Again, we can use exact matching to zoom in on promising subject regions. How do we choose the regions of the query for matching? Let's say we are allowed a single error. It must be located either in the left or the right half of the query, leaving the other half error free. In other words, if we divide the query into $k + 1$ regions, one of them must be error-free and can be detected by exact matching. So we divide the query into regions of length $|q|/(k + 1)$.

Let one such region match at $q[i]$ and $s[j]$. Then we align q with the corresponding fragment in s expanded on either side by k positions, that is, with $s[\ell...r]$, where $\ell = j-i-k+1$ and $r = j+|q|-i+k$. An example for picking the subject region is shown in Figure 21.1A, which leads to the dynamic programming matrix in Figure 21.1B.

The trace back starts at the maximum of the bottom row of the programming matrix and ends upon reaching the top row (Figure 21.1C). This can be thought of as an overlap alignment, where any overhanging subject residues in the final result are chopped off.

The program `kerror` reads query sequences from file and iterates over files containing subject sequences. Each subject is aligned with all queries and for each combination the viable alignments are printed.

Implementation

Our implementation of `kerror` has hooks for imports, types, functions, and the logic of the main function.

193 ⟨*kerror.go* 193⟩≡
```
package main

import (
        ⟨Imports, Ch. 21 194b⟩
```

```
)
⟨Types, Ch. 21 195e⟩
⟨Functions, Ch. 21 196d⟩
func main() {
        ⟨Main function, Ch. 21 194a⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

194a ⟨*Main function, Ch. 21* 194a⟩≡ (193)
```
util.PrepLog("kerror")
⟨Set usage, Ch. 21 194c⟩
⟨Declare options, Ch. 21 194e⟩
⟨Parse options, Ch. 21 195a⟩
⟨Parse input files, Ch. 21 196a⟩
```

We import `util`.

194b ⟨*Imports, Ch. 21* 194b⟩≡ (193) 194d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the program's purpose, and an example command.

194c ⟨*Set usage, Ch. 21* 194c⟩≡ (194a)
```
u := "kerror [-h] [option]... query.fasta [subject.fasta]..."
p := "Calculate k-error alignments between a short " +
        "query and a long subject."
e := "kerror -k 3 query.fasta subject.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

194d ⟨*Imports, Ch. 21* 194b⟩+≡ (193) ◁194b 194f▷
```
"github.com/evolbioinf/clio"
```

Apart from the version, `-v`, we declare an option for the number of errors allowed, options for scoring pairs of residues and gaps, and an option to print the fragment list.

194e ⟨*Declare options, Ch. 21* 194e⟩≡ (194a)
```
var optV = flag.Bool("v", false, "version")
var optK = flag.Int("k", 1, "number of errors")
var optA = flag.Float64("a", 1, "match")
var optI = flag.Float64("i", -3, "mismatch")
var optM = flag.String("m", "", "file containing score matrix")
var optO = flag.Float64("o", -5, "gap opening")
var optE = flag.Float64("e", -2, "gap extension")
var optL = flag.Bool("l", false, "print fragment list")
```

We import `flag`.

194f ⟨*Imports, Ch. 21* 194b⟩+≡ (193) ◁194d 195d▷
```
"flag"
```

We parse the options, respond to `-v` as this stops the program, respond to `-m`, and collect the remaining option values.

195a ⟨*Parse options, Ch. 21* 195a⟩≡ (194a)
```
flag.Parse()
⟨Respond to -v, Ch. 21 195b⟩
⟨Respond to -m, Ch. 21 195c⟩
⟨Collect option values, Ch. 21 195f⟩
```

We print the version.

195b ⟨*Respond to -v, Ch. 21* 195b⟩≡ (195a)
```
if *optV {
        util.PrintInfo("kerror")
}
```

We generate the score matrix either by reading it from a file or by constructing it from the match and the mismatch score.

195c ⟨*Respond to -m, Ch. 21* 195c⟩≡ (195a)
```
var sm *pal.ScoreMatrix
if *optM == "" {
        sm = pal.NewScoreMatrix(*optA, *optI)
} else {
        f, err := os.Open(*optM)
        if err != nil { log.Fatalf("can't open %q", *optM) }
        sm = pal.ReadScoreMatrix(f)
        f.Close()
}
```

We import `os`, `log`, and `pal`.

195d ⟨*Imports, Ch. 21* 194b⟩+≡ (193) ◁194f 196b▷
```
"os"
"log"
"github.com/evolbioinf/pal"
```

There are four option values we pass to the alignment algorithm, gap opening and closing, k, and list printing. We do this via the struct `opts`.

195e ⟨*Types, Ch. 21* 195e⟩≡ (193)
```
type opts struct {
        o, e float64
        k int
        l bool
}
```

We collect the option values.

195f ⟨*Collect option values, Ch. 21* 195f⟩≡ (195a)
```
op := new(opts)
op.k = *optK
op.o = *optO
op.e = *optE
op.l = *optL
```

The remaining tokens on the command line are taken as input files. The first of these is the query file, from which we read the query sequences. If it doesn't exist, we bail with message. Then we call `ParseFiles`, which takes as arguments a list of files, to each of which it applies the function `scan`, which in turn takes as arguments the name of the queries, the score matrix, and the options we just collected.

196a ⟨*Parse input files, Ch. 21* 196a⟩≡ (194a)
```
files := flag.Args()
var queries []*fasta.Sequence
if len(files) < 1 {
        log.Fatal("please enter query file")
} else {
        ⟨Read queries, Ch. 21 196c⟩
}
clio.ParseFiles(files[1:], scan, queries, sm, op)
```

We import `fasta`.

196b ⟨*Imports, Ch. 21* 194b⟩+≡ (193) ◁195d 196e▷
```
"github.com/evolbioinf/fasta"
```

We read the queries.

196c ⟨*Read queries, Ch. 21* 196c⟩≡ (196a)
```
f, err := os.Open(files[0])
if err != nil { log.Fatalf("can't open %q", files[0]) }
sc := fasta.NewScanner(f)
for sc.ScanSequence() {
        q := sc.Sequence()
        queries = append(queries, q)
}
f.Close()
```

Inside `scan`, we retrieve the arguments, iterate across the subject sequences, and for each subject iterate across the queries.

196d ⟨*Functions, Ch. 21* 196d⟩≡ (193)
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 21 196f⟩
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                subject := sc.Sequence()
                ⟨Iterate across queries, Ch. 21 197a⟩
        }
}
```

We import `io`.

196e ⟨*Imports, Ch. 21* 194b⟩+≡ (193) ◁196b 197e▷
```
"io"
```

We retrieve the arguments by type assertion.

196f ⟨*Retrieve arguments, Ch. 21* 196f⟩≡ (196d)
```
queries := args[0].([]*fasta.Sequence)
sm := args[1].(*pal.ScoreMatrix)
op := args[2].(*opts)
```

As we iterate across the queries, we divide each one into $k + 1$ fragments. These fragments are either printed, or we use them to align the query with the subject.

197a ⟨*Iterate across queries, Ch. 21* 197a⟩≡ (196d)
```
for _, query := range queries {
        ⟨Divide query into fragments, Ch. 21 197b⟩
        if op.l {
                ⟨Print query fragments, Ch. 21 197d⟩
        } else {
                ⟨Align query with subject, Ch. 21 198a⟩
        }
}
```

We divide the query into fragments and also store the fragment starts. At the end we adjust the last fragment added.

197b ⟨*Divide query into fragments, Ch. 21* 197b⟩≡ (197a)
```
fragments := make([]string, 0)
starts := make([]int, 0)
q := query.Data()
m := len(q)
r := m / (op.k + 1)
for i := 0; i <= m-r; i += r {
  f := q[i:i+r]
  fragments = append(fragments, string(f))
  starts = append(starts, i)
}
⟨Adjust last fragment, Ch. 21 197c⟩
```

Since fragment lengths are integers, it is quite likely that they do not add up to the query length. In that case the last fragment doesn't end at the query end; we make sure it does.

197c ⟨*Adjust last fragment, Ch. 21* 197c⟩≡ (197b)
```
nf := len(fragments)
fragments[nf-1] = string(q[starts[nf-1]:])
```

We print the query fragments using a tab writer and one-based positions.

197d ⟨*Print query fragments, Ch. 21* 197d⟩≡ (197a)
```
w := tabwriter.NewWriter(os.Stdout, 1, 0, 1, ' ', 0)
fmt.Fprintf(w, "#Id\tStart\tFragment\n")
for i, f := range fragments {
        fmt.Fprintf(w, "%d\t%d\t%s\n", i+1, starts[i]+1, f)
}
w.Flush()
```

We import `tabwriter` and `fmt`.

197e ⟨*Imports, Ch. 21* 194b⟩+≡ (193) ◁196e 198b▷
```
"text/tabwriter"
"fmt"
```

To align the query with the subject, we search for the fragments. Next, we iterate over the matches and for each match construct the subject fragment, align it with the query, and print the result. A subject fragment has coordinates $\ell...r$ and we construct

it only from fragments that lie outside the last fragment aligned. Hence we declare variables ℓ, r outside the search loop.

198a ⟨*Align query with subject, Ch. 21* 198a⟩≡ (197a)
```
var matches []kt.Match
⟨Search for fragments, Ch. 21 198c⟩
var l, r int
for _, match := range matches {
        ⟨Construct subject fragment, Ch. 21 198d⟩
        ⟨Align query with fragment, Ch. 21 198e⟩
        ⟨Print alignment, Ch. 21 198f⟩
}
```

We import `kt`.

198b ⟨*Imports, Ch. 21* 194b⟩+≡ (193) ◁197e
```
"github.com/evolbioinf/kt"
```

We look for the query fragments using a keyword tree.

198c ⟨*Search for fragments, Ch. 21* 198c⟩≡ (198a)
```
ktree := kt.NewKeywordTree(fragments)
matches = ktree.Search(subject.Data(), fragments)
```

If we have a match outside the previous fragment interval, we construct the new subject fragment as explained in the Introduction. We make sure the fragment borders don't lie outside the subject sequence.

198d ⟨*Construct subject fragment, Ch. 21* 198d⟩≡ (198a)
```
if match.Position < r && match.Position > l { continue }
i := starts[match.Pattern]
j := match.Position
l = j - i - op.k
if l < 0 { l = 0 }
r = j + m - i + op.k
if r > len(subject.Data()) { r = len(subject.Data()) }
sbjctFrag := subject.Data()[l:r]
sf := fasta.NewSequence(subject.Header(), sbjctFrag)
```

We align the query and the subject fragment and trim any flanking gaps from the query.

198e ⟨*Align query with fragment, Ch. 21* 198e⟩≡ (198a)
```
oal := pal.NewOverlapAlignment(query, sf, sm, op.o, op.e)
oal.Align()
oal.SetSubjectStart(l)
oal.TrimQuery()
```

We print only alignments that have fewer than k errors. We also set the subject length, as this is not the same as the length of the fragment we used in the dynamic programming.

198f ⟨*Print alignment, Ch. 21* 198f⟩≡ (198a)
```
e := oal.Mismatches() + oal.Gaps()
if e <= op.k {
        oal.SetSubjectLength(len(subject.Data()))
        fmt.Printf("%s\n", oal)
}
```

We're done with `kerror`, time to test it.

Testing

Our outline for testing has hooks for imports and for the testing logic.

199a ⟨*kerror_test.go* 199a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 21 199c⟩
)

func TestKerror(t *testing.T) {
        ⟨Testing, Ch. 21 199b⟩
}
```

We construct a set of tests and then run them.

199b ⟨*Testing, Ch. 21* 199b⟩≡ (199a)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 21 199d⟩
for i, test := range tests {
        ⟨Run test, Ch. 21 199e⟩
}
```

We import `exec`.

199c ⟨*Testing imports, Ch. 21* 199c⟩≡ (199a) 200▷
```
"os/exec"
```

We construct two tests, one for printing the fragment list, the other for running an alignment. In each case the query is located in `q.fasta` and the subject in `s.fasta`.

199d ⟨*Construct tests, Ch. 21* 199d⟩≡ (199b)
```
test := exec.Command("./kerror", "-l", "q.fasta", "s.fasta")
tests = append(tests, test)
test = exec.Command("./kerror", "-k", "6", "q.fasta", "s.fasta")
tests = append(tests, test)
```

For each test, we compare what we get with what we want, which is contained in files `r1.txt` and `r2.txt`.

199e ⟨*Run test, Ch. 21* 199e⟩≡ (199b)
```
get, err := test.Output()
if err != nil { t.Errorf("can't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("can't open %q", f) }
if !bytes.Equal(get, want) {
        t.Errorf("want:\n%s\nget:\n%s", get, want)
}
```

We import `strconv`, `ioutil` and `bytes`.

200 ⟨*Testing imports, Ch. 21* 199c⟩+≡ (199a) ◁199c

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 22

Program `keyMat`: Matching Keywords

Introduction

The program `keyMat` finds the starting positions of one or more patterns in one or more sequences. Its outline contains hooks for imports, function, and the logic of the main function.

202a ⟨*keyMat.go* 202a⟩≡
```
package main

import (
        ⟨Imports, Ch. 22 202c⟩
)

⟨Functions, Ch. 22 205a⟩
func main() {
        ⟨Main function, Ch. 22 202b⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare and parse the options, and run the search.

202b ⟨*Main function, Ch. 22* 202b⟩≡ (202a)
```
util.PrepLog("keyMat")
⟨Set usage, Ch. 22 202d⟩
⟨Declare options, Ch. 22 202f⟩
⟨Parse options, Ch. 22 203b⟩
⟨Run search, Ch. 22 204e⟩
```

We import `util`.

202c ⟨*Imports, Ch. 22* 202c⟩≡ (202a) 202e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage statement, an explanation of the program's purpose, and an example command.

202d ⟨*Set usage, Ch. 22* 202d⟩≡ (202b)
```
u := "keyMat [-h] [options] [patterns] [file(s)]"
p := "Match one or more patterns in sequence data."
e := "keyMat -r ATTC,ATTG foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

202e ⟨*Imports, Ch. 22* 202c⟩+≡ (202a) ◁ 202c 203a ▷
```
"github.com/evolbioinf/clio"
```

Apart from the default help option, `-h`, we declare three additional options:

1. `-f`: file containing patterns
2. `-r`: include reverse strand
3. `-v`: program version

202f ⟨*Declare options, Ch. 22* 202f⟩≡ (202b)
```
var optP = flag.String("p", "", "file with FASTA-formatted patterns")
var optR = flag.Bool("r", false, "include reverse strand")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

203a ⟨*Imports, Ch. 22* 202c⟩+≡ (202a) ◁202e 203d▷
```
"flag"
```

When parsing the options, we respond to a request for the version, collect the patterns, and check them.

203b ⟨*Parse options, Ch. 22* 203b⟩≡ (202b)
```
flag.Parse()
if *optV {
        util.PrintInfo("keyMat")
}
⟨Collec patterns, Ch. 22 203c⟩
⟨Check patterns, Ch. 22 204c⟩
```

The patterns are collected from the command line or a file. Since the file contains FASTA-formatted sequences, all patterns are stored in this format. If no pattern file is given, the first token on the command line is assumed to contain the patterns and is lopped of the beginning of the list of input files.

203c ⟨*Collec patterns, Ch. 22* 203c⟩≡ (203b)
```
var patterns []*fasta.Sequence
files := flag.Args()
if *optP != "" {
        ⟨Collect patterns from file, Ch. 22 204a⟩
} else if len(files) > 0 {
        ⟨Collect patterns from command line, Ch. 22 203e⟩
        files = files[1:]
}
```

We import `fasta`.

203d ⟨*Imports, Ch. 22* 202c⟩+≡ (202a) ◁203a 203f▷
```
"github.com/evolbioinf/fasta"
```

Patterns on the command line are delimited by commas. When converting the patterns to sequences, we use the sequence as header.

203e ⟨*Collect patterns from command line, Ch. 22* 203e⟩≡ (203c)
```
p := strings.Split(files[0], ",")
for _, s := range p {
        seq := fasta.NewSequence(s, []byte(s))
        patterns = append(patterns, seq)
}
```

We import `strings`.

203f ⟨*Imports, Ch. 22* 202c⟩+≡ (202a) ◁203d 204b▷
```
"strings"
```

We use a scanner to collect the patterns from the file.

204a ⟨*Collect patterns from file, Ch. 22* 204a⟩≡ (203c)
```
file, err := os.Open(*optP)
if err != nil {
        log.Fatalf("couldn't open %q\n", *optP)
}
scanner := fasta.NewScanner(file)
for scanner.ScanSequence() {
        patterns = append(patterns, scanner.Sequence())
}
file.Close()
```

We import `os` and `log`.

204b ⟨*Imports, Ch. 22* 202c⟩+≡ (202a) ◁203f 204d▷
```
"os"
"log"
```

There are several ways in which we could check the patterns; we might look for duplicates, test for non-UPAC symbols, or just make sure we have any patterns at all. Here we just count the patterns and abort if there are none.

204c ⟨*Check patterns, Ch. 22* 204c⟩≡ (203b)
```
if len(patterns) == 0 {
        fmt.Fprintf(os.Stderr, "please enter at least one pattern\n")
        os.Exit(-1)
}
```

We import `fmt`.

204d ⟨*Imports, Ch. 22* 202c⟩+≡ (202a) ◁204b 204f▷
```
"fmt"
```

When running the search, we iterate over the input files, which are the remaining command line arguments. We apply the function `scan` to each file. This function takes as argument the keyword tree constructed from the patterns, the patterns in sequence and string format, and the `-r` flag.

204e ⟨*Run search, Ch. 22* 204e⟩≡ (202b)
```
var sp []string
for _, s := range patterns {
        sp = append(sp, string(s.Data()))
}
tree := kt.NewKeywordTree(sp)
clio.ParseFiles(files, scan, tree, patterns, sp, *optR)
```

We import `kt`.

204f ⟨*Imports, Ch. 22* 202c⟩+≡ (202a) ◁204d 205b▷
```
"github.com/evolbioinf/kt"
```

In `scan` we retrieve the arugments just passed and search each sequence in turn.

205a ⟨*Functions, Ch. 22* 205a⟩≡ (202a) 205e▷
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 22 205c⟩
        scanner := fasta.NewScanner(r)
        for scanner.ScanSequence() {
                seq := scanner.Sequence()
                ⟨Search sequence, Ch. 22 205d⟩
        }
}
```

We import `io`.

205b ⟨*Imports, Ch. 22* 202c⟩+≡ (202a) ◁204f
```
"io"
```

The arguments are retrieved via reflection.

205c ⟨*Retrieve arguments, Ch. 22* 205c⟩≡ (205a)
```
tree := args[0].(*kt.Node)
pseq := args[1].([]*fasta.Sequence)
pstr := args[2].([]string)
optR := args[3].(bool)
```

We search the forward and possibly also the reverse strand of a sequence, and print the matches.

205d ⟨*Search sequence, Ch. 22* 205d⟩≡ (205a)
```
matches := tree.Search(seq.Data(), pstr)
fmt.Printf("# %s\n", seq.Header())
printMatches(matches, pseq)
if optR {
        seq.ReverseComplement()
        matches = tree.Search(seq.Data(), pstr)
        fmt.Printf("# %s - Reverse\n", seq.Header())
        printMatches(matches, pseq)
}
```

Matches are printed as pairs of positions and names.

205e ⟨*Functions, Ch. 22* 205a⟩+≡ (202a) ◁205a
```
func printMatches(matches []kt.Match, patterns []*fasta.Sequence) {
        for _, m := range matches {
                s := patterns[m.Pattern]
                fmt.Printf("%d\t%s\n", m.Position+1, s.Header())
        }
}
```

The program is finished, let's tests it.

Testing

The outline of the testing program contains hooks for imports and the testing logic.

206a ⟨*keyMat_test.go* 206a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 22 206c⟩
)

func TestKeyMat(t *testing.T) {
        ⟨Testing, Ch. 22 206b⟩
}
```

We declare a few testing commands. Each of them corresponds to a file of precomputed output. Then the commands are run and the results we get compared to the results we want.

206b ⟨*Testing, Ch. 22* 206b⟩≡ (206a)
```
var commands []*exec.Cmd
⟨Construct commands, Ch. 22 206d⟩
⟨Construct list of output files, Ch. 22 207b⟩
for i, command := range commands {
        ⟨Run command, Ch 22 207d⟩
}
```

We import `exec`.

206c ⟨*Testing imports, Ch. 22* 206c⟩≡ (206a) 207c ▷
```
"os/exec"
```

The commands look for matches in a test sequence. To write them succinctly, we first prepare a set of variables covering the command to be run, the list of patterns, the pattern file, and the input file.

206d ⟨*Construct commands, Ch. 22* 206d⟩≡ (206b)
```
r := "./keyMat"
p := "ATTT,ATTC,AT,TG,TT"
f := "patterns.fasta"
i := "test.fasta"
⟨Write commands, Ch. 22 207a⟩
```

The commands go through the options.

207a ⟨*Write commands, Ch. 22* 207a⟩≡ (206d)
```
c := exec.Command(r, p, i)
commands = append(commands, c)
c = exec.Command(r, "-r", p, i)
commands = append(commands, c)
c = exec.Command(r, "-p", f, i)
commands = append(commands, c)
c = exec.Command(r, "-p", f, "-r", i)
commands = append(commands, c)
```

For each command, there is an output file.

207b ⟨*Construct list of output files, Ch. 22* 207b⟩≡ (206b)
```
var files []string
for i, _ := range commands {
          f := "r" + strconv.Itoa(i+1) + ".txt"
          files = append(files, f)
}
```

We import `strconv`.

207c ⟨*Testing imports, Ch. 22* 206c⟩+≡ (206a) ◁206c 207e▷
```
"strconv"
```

When running a command, we compare the result we get with the result we want.

207d ⟨*Run command, Ch 22* 207d⟩≡ (206b)
```
get, err := command.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", command)
}
want, err := ioutil.ReadFile(files[i])
if err != nil {
          t.Errorf("couldn't open %q\n", files[i])
}
if !bytes.Equal(want, get) {
          t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `ioutil` and `bytes`.

207e ⟨*Testing imports, Ch. 22* 206c⟩+≡ (206a) ◁207c
```
"io/ioutil"
"bytes"
```

Chapter 23

Program `maf`: Calculate Match Factors

Introduction

When compressing a string, it is often split into longest repeated substrings. For example, the sequence `TACTA` splits into `TA.C.TA`. Notice that the unique character, `C`, is treated as if it was a repeated substring. We call these repeated substrings *match factors*.

To find the match factors, we traverse the lcp array of the input sequence. The trick is to do this in the order in which the suffixes appear in the input. This is achieved using the inverse suffix array

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

For each position in the sequence we look up the longest match starting there, skip the match, and repeat. This procedure is summarized in Algorithm 4.

Algorithm 4 Computing the match factor decomposition.

Require: S {input sequence}
Require: lcp {longest common prefix array of S}
Require: isa {inverse suffix array of S}
Require: n {length of S}
Ensure: Match decomposition
1: $i \leftarrow 1$ {set index to first position in S}
2: $\text{lcp}[n+1] \leftarrow 0$ {prevent out of bounds error}
3: **while** $i \leq n$ **do**
4: $l_1 \leftarrow \text{lcp}[\text{isa}[i]]$
5: $l_2 \leftarrow \text{lcp}[\text{isa}[i]+1]$
6: $j \leftarrow i + \max(l_1, l_2, 1) - 1$
7: `reportMatchFactor`$(S[i...j])$
8: $i \leftarrow j + 1$
9: **end while**

The program `maf` reads a FASTA-formatted sequence and writes its match factors. Alternatively, it just writes the number of match factors.

Implementation

The outline of `maf` has hooks for imports, functions, and the logic of the main function.

209 ⟨*maf.go* 209⟩≡

```
package main

import (
        ⟨Imports, Ch. 23 210b⟩
)
⟨Functions, Ch. 23 211c⟩
func main() {
        ⟨Main function, ch. 23 210a⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

210a ⟨*Main function, ch. 23* 210a⟩≡ (209)
```
util.PrepLog("maf")
⟨Set usage, Ch. 23 210c⟩
⟨Declare options, Ch. 23 210e⟩
⟨Parse options, Ch. 23 210g⟩
⟨Parse input files, Ch. 23 211b⟩
```

We import `util`.

210b ⟨*Imports, Ch. 23* 210b⟩≡ (209) 210d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `maf`, and an example command.

210c ⟨*Set usage, Ch. 23* 210c⟩≡ (210a)
```
u := "maf [-h] [option]... [foo.fasta]..."
p := "Compute the match factors of a sequence."
e := "maf foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

210d ⟨*Imports, Ch. 23* 210b⟩+≡ (209) ◁210b 210f▷
```
"github.com/evolbioinf/clio"
```

Apart from the version, the user can request the number of factors, rather than the actual factors.

210e ⟨*Declare options, Ch. 23* 210e⟩≡ (210a)
```
var optV = flag.Bool("v", false, "version")
var optN = flag.Bool("n", false, "print number of factors " +
          "instead of factors")
```

We import `flag`.

210f ⟨*Imports, Ch. 23* 210b⟩+≡ (209) ◁210d 211a▷
```
"flag"
```

We parse the options and first respond to `-v`, as this stops the program. Then we respond to `-d`, the request for the number of factors. The number of factors is printed in a table consisting of four columns, accession, number of factors, number of residues, and the number of factors per residue.

210g ⟨*Parse options, Ch. 23* 210g⟩≡ (210a)
```
flag.Parse()
if *optV {
          util.PrintInfo("maf")
}
var w *tabwriter.Writer
if *optN {
          w = tabwriter.NewWriter(os.Stdout, 1, 0, 2, ' ' , 0)
          fmt.Fprintf(w, "#acc\tfactors\tresidues\t" +
                  "factors/residues\n")
}
```

We import `tabwriter`, `os`, and `fmt`.

211a ⟨*Imports, Ch. 23* 210b⟩+≡ (209) ◁210f 211d▷
```
"text/tabwriter"
"os"
"fmt"
```

The remaining tokens on the command line are taken as the names of input files. They are parsed with the function `scan`, which takes as argument whether or not to print the number of factors and the tab writer.

211b ⟨*Parse input files, Ch. 23* 211b⟩≡ (210a)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optN, w)
```

Inside `scan` we retrieve the `-d` option and parse the sequences. For each sequence, we prepare its factorization, factorize it, and print the factorization.

211c ⟨*Functions, Ch. 23* 211c⟩≡ (209) 212b▷
```
func scan(r io.Reader, args ...interface{}) {
        printNum := args[0].(bool)
        w := args[1].(*tabwriter.Writer)
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                seq := sc.Sequence()
                ⟨Prepare factorization, Ch. 23 211e⟩
                ⟨Factorize sequence, Ch. 23 212a⟩
                ⟨Print factorization, Ch. 23 212c⟩
        }
        if printNum { w.Flush() }
}
```

We import `io` and `fasta`.

211d ⟨*Imports, Ch. 23* 210b⟩+≡ (209) ◁211a 211f▷
```
"io"
"github.com/evolbioinf/fasta"
```

As shown in Algorithm 4, the factorization relies on the longest common prefix array, lcp, and the inverse suffix array, isa, which we calculate from the suffix array, sa.

211e ⟨*Prepare factorization, Ch. 23* 211e⟩≡ (211c)
```
t := seq.Data()
sa := esa.Sa(t)
isa := make([]int, len(sa))
lcp := esa.Lcp(t, sa)
lcp = append(lcp, 0)
for i, s := range sa {
  isa[s] = i
}
```

We import `esa`.

211f ⟨*Imports, Ch. 23* 210b⟩+≡ (209) ◁211d 212d▷
```
"github.com/evolbioinf/esa"
```

We factorize the sequence into byte slices.

212a ⟨*Factorize sequence, Ch. 23* 212a⟩≡ (211c)
```
factors := make([][]byte, 0)
i := 0
for i < len(sa) {
        l1 := lcp[isa[i]]
        l2 := lcp[isa[i] + 1]
        j := i + max(max(l1, l2), 1)
        factors = append(factors, t[i:j])
        i = j
}
```

We implement `max`.

212b ⟨*Functions, Ch. 23* 211c⟩+≡ (209) ◁211c
```
func max(i, j int) int {
        if i > j {
                return i
        }
        return j
}
```

If we're asked to print the number of factors, we fill in one line of the factors table consisting of accession, number of factors, number of residues, and factors per residue. Otherwise, we construct the factorized sequence and print it.

212c ⟨*Print factorization, Ch. 23* 212c⟩≡ (211c)
```
if printNum {
        n := len(factors)
        m := len(t)
        a := strings.Fields(seq.Header())[0]
        fmt.Fprintf(w, "%s\t%d\t%d\t%.3g\n", a, n,
                m, float64(n)/float64(m))
} else {
        var fs *fasta.Sequence
        ⟨Construct factorized sequence, Ch. 23 213a⟩
        fmt.Println(fs)
}
```

We import `strings` and `fmt`.

212d ⟨*Imports, Ch. 23* 210b⟩+≡ (209) ◁211f
```
"strings"
"fmt"
```

We construct the factorized sequence by concatenating the factors separated by dots. We also append "match factors" to the header.

213a ⟨*Construct factorized sequence, Ch. 23* 213a⟩≡ (212c)
```
fd := make([]byte, 0)
n := len(factors)
for i := 0; i < n-1; i++ {
        fd = append(fd, factors[i]...)
        fd = append(fd, '.')
}
fd = append(fd, factors[n-1]...)
h := seq.Header() + " - match factors"
fs = fasta.NewSequence(h, fd)
```

We've finished `maf`, time to test it.

Testing

Our code for testing `maf` has hooks for imports and the testing logic.

213b ⟨*maf_test.go* 213b⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 23 213d⟩
)

func TestMaf(t *testing.T) {
        ⟨Testing, Ch. 23 213c⟩
}
```

We construct a set of tests and iterate over them.

213c ⟨*Testing, Ch. 23* 213c⟩≡ (213b)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 23 213e⟩
for i, test := range tests {
        ⟨Run test, Ch. 23 214a⟩
}
```

We import `exec`.

213d ⟨*Testing imports, Ch. 23* 213d⟩≡ (213b) 214b ▷
```
"os/exec"
```

We construct two tests, one for the factorization, the other for the factor counting. Both run on the same input, 630 bp of the *Mycoplasma genitalium* genome contained in `t.fasta`.

213e ⟨*Construct tests, Ch. 23* 213e⟩≡ (213c)
```
f := "t.fasta"
test := exec.Command("./maf", f)
tests = append(tests, test)
test = exec.Command("./maf", "-n", f)
tests = append(tests, test)
```

When running a test, we compare the results we get with the results we want contained in files `r1.txt` and `r2.txt`.

214a ⟨*Run test, Ch. 23* 214a⟩≡ (213c)

```
get, err := test.Output()
if err != nil { t.Errorf("can't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
want = append(want, '\n')
if err != nil { t.Errorf("can't open %q", f) }
if bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

214b ⟨*Testing imports, Ch. 23* 213d⟩+≡ (213b) ◁213d

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 24

Program `midRoot`: Midpoint Rooting of Phylogenies

Introduction

A phylogeny summarizes the evolutionary relationships between a sample of organisms. For example, the imaginary Newick tree in Figure 24.1A plotted Figure 24.1B shows the genealogy of six organisms, also called *taxa*. The branches have lengths proportional to a measure of evolutionary change, often the number of mutations per site. The phylogeny is drawn in radial layout to emphasize that it has no root. In other words, the most recent common ancestor of the taxa is unknown. This is a standard result of many algorithms for reconstructing phylogenies, for example the neighbor joining algorithm [27]. Still, in reality there was a common ancestor. A popular method for placing it on the tree is called midpoint rooting, where the root is located midpoint between the most distant taxa. In our example tree, the most distant pair of taxa is t_1/t_4. Rooting the phylogeny midpoint between them returns Figure 24.1C, where the branch leading to t_1 was broken into a branch leading from the root to t_1 and a shorter branch leading from the root to the rest of the tree.

A

```
(t4:0.009106,(t1:0.053575,(t3:0.002138,t5:0.002862)
:0.019425):0.015394,(t2:0.001169,t6:0.002031):0.005694);
```

B **C**

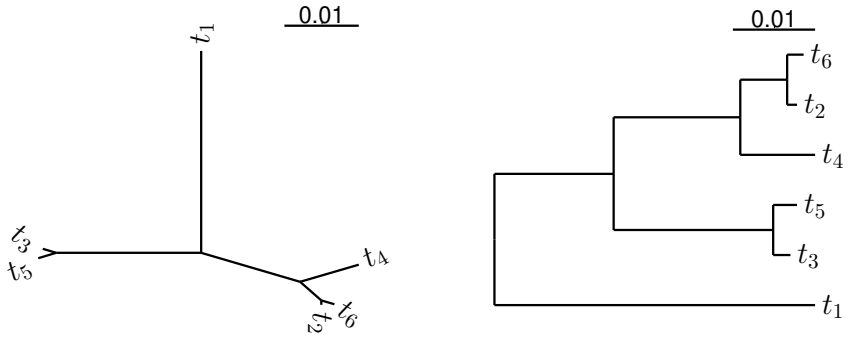


Figure 24.1: An example phylogeny in Newick notation (**A**) plotted as an unrooted tree (**B**) and as a rooted tree after midpoint rooting between t_1 and t_4 (**C**) using `midRoot`.

The program `midRoot` reads one or more trees and prints their midpoint rooted versions.

Implementation

The outline of `midRoot` has hooks for imports, functions, and the logic of the main function.

216 ⟨*midRoot.go* 216⟩≡

```
package main
```

```
import (
        ⟨Imports, Ch. 24 217b⟩
)
⟨Functions, Ch. 24 218b⟩
func main() {
        ⟨Main function, Ch. 24 217a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

217a ⟨*Main function, Ch. 24* 217a⟩≡ (216)
```
util.PrepLog("midRoot")
⟨Set usage, Ch. 24 217c⟩
⟨Declare options, Ch. 24 217e⟩
⟨Parse options, Ch. 24 217g⟩
⟨Parse input files, Ch. 24 218a⟩
```

We import `util`.

217b ⟨*Imports, Ch. 24* 217b⟩≡ (216) 217d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `midRoot`, and an example command.

217c ⟨*Set usage, Ch. 24* 217c⟩≡ (217a)
```
u := "midRoot [-h] [option]... [foo.nwk]..."
p := "Add midpoint root to a tree."
e := "midRoot foo.nwk"
clio.Usage(u, p, e)
```

We import `clio`.

217d ⟨*Imports, Ch. 24* 217b⟩+≡ (216) ◁217b 217f▷
```
"github.com/evolbioinf/clio"
```

Apart from the version (`-v`), we declare an option of printing the pair of most distant taxa (`-p`).

217e ⟨*Declare options, Ch. 24* 217e⟩≡ (217a)
```
var optV = flag.Bool("v", false, "version")
var optP = flag.Bool("p", false, "print most distant pair")
```

We import `flag`.

217f ⟨*Imports, Ch. 24* 217b⟩+≡ (216) ◁217d 218c▷
```
"flag"
```

We parse the options and respond to `-v` as stops the program.

217g ⟨*Parse options, Ch. 24* 217g⟩≡ (217a)
```
flag.Parse()
if *optV {
        util.PrintInfo("midRoot")
}
```

The remaining tokens on the command line are taken as the names of input files. We parse each of these files using the function `scan`, which takes the pair printing option as argument.

218a ⟨*Parse input files, Ch. 24* 218a⟩≡ (217a)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optP)
```

Inside `scan`, we retrieve the pair printing option and iterate over the input. For each tree we read, we find the most distant taxa, perhaps print them, reroot the tree, and print it.

218b ⟨*Functions, Ch. 24* 218b⟩≡ (216) 219a▷
```
func scan (r io.Reader, args ...interface{}) {
        printPair := args[0].(bool)
        sc := nwk.NewScanner(r)
        for sc.Scan() {
                root := sc.Tree()
                ⟨Find most distant taxa, Ch. 24 218e⟩
                ⟨Print most distant taxa?, Ch. 24 219d⟩
                ⟨Reroot tree, Ch. 24 219e⟩
                fmt.Println(root)
        }
}
```

We import `io` and `fmt`.

218c ⟨*Imports, Ch. 24* 217b⟩+≡ (216) ◁217f 218d▷
```
"fmt"
```

218d ⟨*Imports, Ch. 24* 217b⟩+≡ (216) ◁218c 218f▷
```
"io"
```

To find the most distant taxa, we collect the leaves of the tree. Then we calculate their pairwise distances and remember the maximum.

218e ⟨*Find most distant taxa, Ch. 24* 218e⟩≡ (218b)
```
var leaves []*nwk.Node
leaves = collectLeaves(root, leaves)
n := len(leaves)
max := -math.MaxFloat64
var mi, mj int
for i := 0; i < n-1; i++ {
        for j := i+1; j < n; j++ {
                ⟨Calculate distance, Ch. 24 219b⟩
        }
}
```

We import `nwk` and `math`.

218f ⟨*Imports, Ch. 24* 217b⟩+≡ (216) ◁218d
```
"github.com/evolbioinf/nwk"
"math"
```

We collect the leaves.

219a ⟨*Functions, Ch. 24* 218b⟩+≡ (216) ◁218b 222a▷
```
func collectLeaves(v *nwk.Node, l []*nwk.Node) []*nwk.Node {
        if v == nil { return l }
        l = collectLeaves(v.Child, l)
        l = collectLeaves(v.Sib, l)
        if v.Child == nil {
                l = append(l, v)
        }
        return l
}
```

To find the distance between two leaves, we find their lowest common ancestor and sum the distances from that ancestor to the leaves. Then we compare the distance to the current maximum.

219b ⟨*Calculate distance, Ch. 24* 219b⟩≡ (218e)
```
l1 := leaves[i]
l2 := leaves[j]
a := l1.LCA(l2)
d := l1.UpDistance(a) + l2.UpDistance(a)
```
⟨*Compare distance to maximum, Ch. 24* 219c⟩

If the current distance exceeds the last maximum distance we found, we update the maximum and its indexes.

219c ⟨*Compare distance to maximum, Ch. 24* 219c⟩≡ (219b)
```
if max < d {
        max = d
        mi = i
        mj = j
}
```

We might be asked to print the two taxa we just found and their distance.

219d ⟨*Print most distant taxa?, Ch. 24* 219d⟩≡ (218b)
```
if printPair {
        fmt.Printf("# d(%s, %s): %.3g\n",
                leaves[mi].Label, leaves[mj].Label, max)
}
```

Consider again the tree in Figure 24.1A. The pair of leaves with the largest distance happens to be t_1 and t_4. Since t_1 is further removed from their lowest common ancestor, we start climbing at t_1 and immediately find the edge to split.

To clarify what we do next, consider Figure 24.2A, which is Figure 24.1A with all nodes labeled and no branch lengths. We find the edge to split (Figure 24.2B), add the new root, n_5 (Figure 24.2C), and rearrange the tree such that n_5 ends up as the root (Figure 24.2D).

219e ⟨*Reroot tree, Ch. 24* 219e⟩≡ (218b)
⟨*Find edge to split, Ch. 24* 221a⟩
⟨*Insert root, Ch. 24* 221b⟩
⟨*Adjust branch lengths, Ch. 24* 221c⟩
⟨*Rearrange tree, Ch. 24* 221d⟩

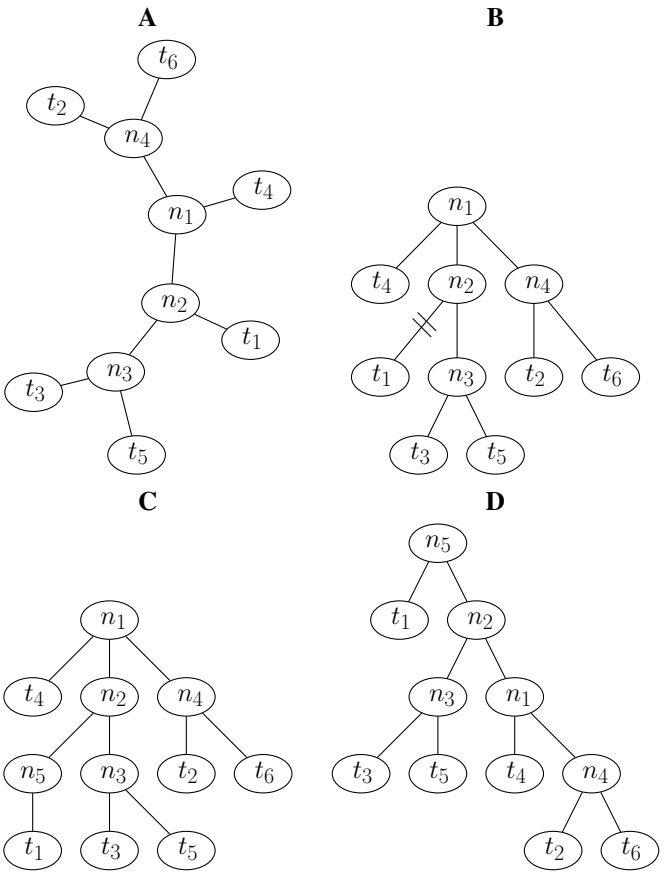


Figure 24.2: Rooting a tree. (**A**) is drawn in the unrooted, radial layout often used in biology, even though internally it is rooted on n_1. This rooting is made explicit in (**B**), where we wish to reroot the tree on edge (t_1, n_2) marked by $||$. In (**C**) the future new root n_5 is added to the tree by splitting (t_1, n_2). (**D**) is the newly rooted tree. It is obtained by picking up (**C**) at n_5 and shaking it [22, p. 373].

To find the edge to split, we climb towards to root either from node i or node j, whichever is most distant from their common ancestor.

221a ⟨*Find edge to split, Ch. 24* 221a⟩≡ (219e)
```
l1 := leaves[mi]
l2 := leaves[mj]
a := l1.LCA(l2)
v := l1
if l1.UpDistance(a) < l2.UpDistance(a) { v = l2 }
s := v.Length
for s < max / 2.0 {
          v = v.Parent
          s += v.Length
}
```

The edge (v, p) is to be split. We create the new root, r. Then we add r as a child to p, remove v as a child from p, and add v as a child of r.

221b ⟨*Insert root, Ch. 24* 221b⟩≡ (219e)
```
r := nwk.NewNode()
p := v.Parent
p.AddChild(r)
p.RemoveChild(v)
r.AddChild(v)
```

To find the new branch lengths, $d(v, r)$ and $d(p, r)$, let m be the distance between the leaves we're considering and s the sum of the branch lengths up to p. Then $d(v, r) = s - m/2$ and $d(p, r) = v$.Length $- d(v, r)$.

221c ⟨*Adjust branch lengths, Ch. 24* 221c⟩≡ (219e)
```
x2 := s - max / 2.0
x1 := v.Length - x2
v.Length = x1
r.Length = x2
```

We now turn the provisional new root r into the actual root. Donald Knuth described the process for this as picking up the tree by n_5 in Figure 24.2C and shaking it to get figure 24.2D [22, p. 371]. More formally, this is done by climbing from n_5 to the old root, n_1, and at every step converting the parent to to a child node and adjusting the branch lengths accordingly. For this purpose we apply the function `parentToChild` to the new root, before assigning it to the variable `root`, which gets us from Figure 24.2C to Figure 24.2D.

221d ⟨*Rearrange tree, Ch. 24* 221d⟩≡ (219e)
```
parentToChild(r)
root = r
```

Inside `parentToChild`, we climb as far as we can and then exchange parent and child.

222a ⟨*Functions, Ch. 24* 218b⟩+≡ (216) ◁219a

```
func parentToChild(v *nwk.Node) {
        if v.Parent.Parent != nil {
                parentToChild(v.Parent)
        }
        p := v.Parent
        p.RemoveChild(v)
        v.AddChild(p)
        p.Length = v.Length
        p.HasLength = true
}
```

The program `midRoot` is finished, so let's test it.

Testing

The outline of our testing program has hooks for imports and the testing logic.

222b ⟨*midRoot_test.go* 222b⟩≡

```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 24 223⟩
)

func TestMidRoot(t *testing.T) {
        ⟨Testing, Ch. 24 222c⟩
}
```

We run the program with the pair printing option on the tree shown in Figure 24.1A stored in `test.nwk`. Then we compare the tree we get to the tree we want stored in `r.txt`.

222c ⟨*Testing, Ch. 24* 222c⟩≡ (222b)

```
cmd := exec.Command("./midRoot", "-p", "test.nwk")
get, err := cmd.Output()
if err != nil {
        t.Errorf("can't run %q", cmd)
}
want, err := ioutil.ReadFile("r.txt")
if err != nil {
        t.Errorf("can't optn r.txt")
}
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `exec`, `ioutil`, and `bytes`.

223 ⟨*Testing imports, Ch. 24* 223⟩≡ (222b)

```
"os/exec"
"io/ioutil"
"bytes"
```

Chapter 25

Program `mtf`: Move to Front

Introduction

Compression works best on runs of small integers, like zeros and ones. However, zeros and ones don't directly correspond to residues. Moreover, we'd like to transform runs of any kind of residue to runs of zeros.

To do this, we start from a text, for example

```
TTTCCCAAAGGG
```

This implies an alphabet, where each of the characters in the text is associated with a number:

Residue	Number
A	0
C	1
G	2
T	3

We start enoding the first T. It corresponds to number 3 in the alphabet. Then we move the T to the front of the alphabet to get

Residue	Number
T	0
A	1
C	2
G	3

The next residue is another T, this time encoded as 0. The alphabet remains unchanged until we encounter the C, a 2, but next a 0, and so on. The transformed text is

3 0 0 2 0 0 2 0 0 3 0 0

Not surprisingly, this technique is called "move to front". Its defining characteristic is that it transforms runs of *any* character into runs of zeros. [5]. Move to front is reversible using the same reasoning. We start from the original alphabet and a sequence of integers, say that above. The initial 3 corresponds to T. Then we move the T to front, find another T for 0, and so on.

The program `mtf` reads a text in FASTA format and encodes it as blank-separated integers by move to front. It stores the alphabet as the last field in the FASTA header of its output. It can also reverse this step and decode the output of a previous `mtf` step.

Implementation

Our outline of `mtf` has hooks for imports, functions, and the logic of the main function.

225 ⟨*mtf.go* 225⟩≡

```
package main

import (
        ⟨Imports, Ch. 25 226b⟩
)
⟨Functions, Ch. 25 227a⟩
func main() {
        ⟨Main function, Ch. 25 226a⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

226a ⟨*Main function, Ch. 25* 226a⟩≡ (225)
```
util.PrepLog("mtf")
⟨Set usage, Ch. 25 226c⟩
⟨Declare options, Ch. 25 226e⟩
⟨Parse options, Ch. 25 226g⟩
⟨Parse input files, Ch. 25 226h⟩
```

We import `util`.

226b ⟨*Imports, Ch. 25* 226b⟩≡ (225) 226d ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `mtf`, and an example command.

226c ⟨*Set usage, Ch. 25* 226c⟩≡ (226a)
```
u := "mtf [-h] [option]... [foo.fasta]..."
p := "Perform move to front encoding and decoding."
e := "mtf -d encoded.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

226d ⟨*Imports, Ch. 25* 226b⟩+≡ (225) ◁226b 226f ▷
```
"github.com/evolbioinf/clio"
```

As options we declare the version, `-v`, and decoding, `-d`.

226e ⟨*Declare options, Ch. 25* 226e⟩≡ (226a)
```
var optV = flag.Bool("v", false, "version")
var optD = flag.Bool("d", false, "decode")
```

We import `flag`.

226f ⟨*Imports, Ch. 25* 226b⟩+≡ (225) ◁226d 227b ▷
```
"flag"
```

We parse the options and respond to `-v` as this stops the program.

226g ⟨*Parse options, Ch. 25* 226g⟩≡ (226a)
```
flag.Parse()
if *optV {
        util.PrintInfo("mtf")
}
```

The remaining tokens on the command line are taken as the names of input files. Each file is scanned with the function `scan`, which takes as argument the decoding switch, `-d`.

226h ⟨*Parse input files, Ch. 25* 226h⟩≡ (226a)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optD)
```

Inside scan, we retrieve the decoding option. Then we declare the alphabet, which we still have to actually determine, and go on to either decode or encode the FASTA formatted input.

227a ⟨*Functions, Ch. 25* 227a⟩≡ (225) 228e ▷
```
func scan(r io.Reader, args ...interface{}) {
        dec := args[0].(bool)
        var alphabet []byte
        if dec {
                ⟨Decode, Ch. 25 227c⟩
        } else {
                ⟨Encode, Ch. 25 229b⟩
        }
}
```

We import `io`.

227b ⟨*Imports, Ch. 25* 226b⟩+≡ (225) ◁226f 227d ▷
```
"io"
```

When decoding, we expect rows of blank-separated integers as input, separated by FASTA headers.

227c ⟨*Decode, Ch. 25* 227c⟩≡ (227a)
```
var seq []byte
first := true
header := ""
sc := bufio.NewScanner(r)
for sc.Scan() {
        if sc.Text()[0] == '>' {
                ⟨Deal with header in decoding, Ch. 25 227e⟩
        } else {
                ⟨Deal with data in decoding, Ch. 25 228c⟩
        }
}
⟨Deal with last sequence in decoding, Ch. 25 229a⟩
```

We import `bufio`.

227d ⟨*Imports, Ch. 25* 226b⟩+≡ (225) ◁227b 228a ▷
```
"bufio"
```

The first header is just noted. Subsequent headers close a previous sequence, so we print the sequence decoded thus far, print it, and reset it.

227e ⟨*Deal with header in decoding, Ch. 25* 227e⟩≡ (227c)
```
⟨Get alphabet from header, Ch. 25 228b⟩
if first {
        first = false
} else {
        s := fasta.NewSequence(header, seq)
        fmt.Println(s)
        seq = seq[:0]
}
header = sc.Text()[1:] + " - decoded"
```

We import `fasta` and `fmt`.

228a ⟨*Imports, Ch. 25* 226b⟩+≡ (225) ◁227d 228d▷
```
"github.com/evolbioinf/fasta"
"fmt"
```

The last field in the header contains the alphabet. This is surrounded by quotes, which we remove.

228b ⟨*Get alphabet from header, Ch. 25* 228b⟩≡ (227e)
```
fields := strings.Fields(sc.Text())
al := fields[len(fields)-1]
al = al[1:len(al)-1]
for _, c := range al {
        alphabet = append(alphabet, byte(c))
}
```

A row of data consists of strings representing integers. We convert them to actual integers and iterate over them.

228c ⟨*Deal with data in decoding, Ch. 25* 228c⟩≡ (227c)
```
fields := strings.Fields(sc.Text())
for _, field := range fields {
        i, err := strconv.Atoi(field)
        if err != nil { log.Fatalf("can't convert %q", field) }
        r, err := decode(i, alphabet)
        if err == nil {
                seq = append(seq, r)
        } else { log.Fatalf(err.Error()) }
}
```

We import `strings`, `strconv`, and `log`.

228d ⟨*Imports, Ch. 25* 226b⟩+≡ (225) ◁228a
```
"strings"
"strconv"
"log"
```

In the function `decode`, we decode the string representing an integer and rearrange the alphabet.

228e ⟨*Functions, Ch. 25* 227a⟩+≡ (225) ◁227a 230a▷
```
func decode(k int, a []byte) (byte, error) {
        for i, c := range a {
                if i == k {
                        copy(a[1:], a[:i])
                        a[0] = c
                        return c, nil
                }
        }
        return 0, fmt.Errorf("can't decode %d", k)
}
```

We print the last sequence in the file.

229a ⟨*Deal with last sequence in decoding, Ch. 25* 229a⟩≡ (227c)
```
s := fasta.NewSequence(header, seq)
fmt.Println(s)
seq = seq[:0]
```

For encoding, we deduce the alphabet from the sequence, encode the sequence, and print it.

229b ⟨*Encode, Ch. 25* 229b⟩≡ (227a)
```
sc := fasta.NewScanner(r)
var ns []int
for sc.ScanSequence() {
        seq := sc.Sequence()
        ⟨Get alphabet from sequence, Ch. 25 229c⟩
        ⟨Encode sequence, Ch. 25 229d⟩
        ⟨Print encoded sequence, Ch. 25 230b⟩
        ns = ns[:0]
}
```

To construct the alphabet, we keep track of the distinct characters in the data using a map. Having established the alphabet, we keep a copy of its original order, before any move to front has occurred. This is later used for decoding.

229c ⟨*Get alphabet from sequence, Ch. 25* 229c⟩≡ (229b)
```
cm := make(map[byte]bool)
data := seq.Data()
for _, c := range data {
        if !cm[c] {
                alphabet = append(alphabet, c)
                cm[c] = true
        }
}
oa := string(alphabet)
```

We iterate over the residues (or characters) and encode each one as an integer.

229d ⟨*Encode sequence, Ch. 25* 229d⟩≡ (229b)
```
for _, c := range data {
        i, err := encode(c, alphabet)
        if err == nil {
                ns = append(ns, i)
        } else { log.Fatalf(err.Error()) }
}
```

We encode a byte into an integer. If we can't find the character submitted, we throw an error.

230a ⟨*Functions, Ch. 25* 227a⟩+≡ (225) ◁228e
```
func encode(c byte, a []byte) (int, error) {
        for i, x := range a {
                if x == c {
                        copy(a[1:], a[:i])
                        a[0] = c
                        return i, nil
                }
        }
        return -1, fmt.Errorf("can't encode %q", c)
}
```

We print an encoded sequence as a FASTA header with the original alphabet as the last field, followed by rows of integers separated by blanks.

230b ⟨*Print encoded sequence, Ch. 25* 230b⟩≡ (229b)
```
fmt.Printf(">%s - mtf %q\n", seq.Header(), oa)
ll := fasta.DefaultLineLength
n := len(ns)
for i := 0; i < n; i += ll {
        for j := 0; i+j < n && j < ll; j++ {
                if j > 0 { fmt.Printf(" ") }
                fmt.Printf("%d", ns[i+j])
        }
        fmt.Printf("\n")
}
```

We've finished writing `mtf`, let's test it.

Testing

The outline of our testing code contains hooks for imports and the testing logic.

230c ⟨*mtf_test.go* 230c⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 25 231b⟩
)
func TestMtf(t *testing.T) {
        ⟨Testing, Ch. 25 231a⟩
}
```

We construct the tests and run them.

231a ⟨*Testing, Ch. 25* 231a⟩≡ (230c)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 25 231c⟩
for i, test := range tests {
        ⟨Run test, Ch. 25 231d⟩
}
```

We import `exec`.

231b ⟨*Testing imports, Ch. 25* 231b⟩≡ (230c) 231e ▷
```
"os/exec"
```

We run two tests, one for encoding, the second for decoding. The input for encoding is `t1.fasta`, the input for decoding the output from encoding, `r1.fasta`.

231c ⟨*Construct tests, Ch. 25* 231c⟩≡ (231a)
```
test := exec.Command("./mtf", "t1.fasta")
tests = append(tests, test)
test = exec.Command("./mtf", "-d", "r1.fasta")
tests = append(tests, test)
```

When running a test, we compare the result we get with the result we want, which is stored in `r1.fasta` and `r2.fasta`.

231d ⟨*Run test, Ch. 25* 231d⟩≡ (231a)
```
get, err := test.Output()
if err != nil { t.Errorf("can't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".fasta"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("can't open %q", f) }
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

231e ⟨*Testing imports, Ch. 25* 231b⟩+≡ (230c) ◁231b
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 26

Program `mum2plot`: Transform MUMmer Output for Plotting

Table 26.1: Example output from MUMmer (**A**); after transformation with `mum2plot` (**B**)

A			**B**			
> s1						
128	42	45	128	42	172	86
129	163	51	129	163	179	213
75	114	56	75	114	130	169
> s1 Reverse			253	50	272	31
253	50	20	253	57	273	37
253	57	21	305	60	324	41
305	60	20				

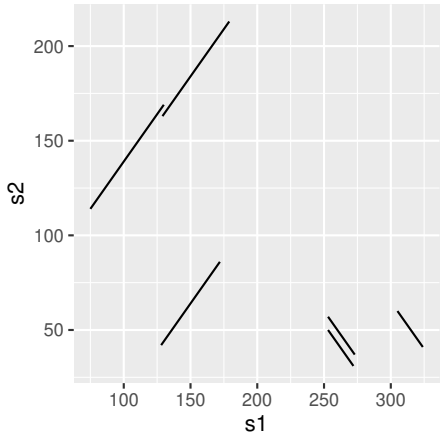


Figure 26.1: A plot of the MUMmer data shown in Table 26.1A after it was transformed to the data in Table 26.1B using `mum2plot | plotSeg`.

Introduction

MUMmer is a software package for quickly aligning pairs of genomes [24]. Its output consists of the coordinates of exact matches between the two input genomes. The matches are written in two lists, one for the forward, the other for the reverse strand. Each list is opened by a header line that looks like a FASTA header (Table 26.1A). The header of the reverse list ends in the word "Reverse".

Each entry in the two match lists consists of the starting point of a line and its length. For example, the first line starts at $x_1 = 128, y_1 = 42$ and has $\ell = 45$. In the forward list, the implied end point of this line is $x_2 = x_1 + \ell - 1 = 172$ and $y_2 = y_1 + \ell - 1 = 86$. Reverse matches lean in the opposite direction, so the x-coordinate of their end-points is $x_2 = x_1 - \ell + 1$.

The program `mum2plot` converts MUMmer output to quartets (x_1, y_2, x_2, y_2). Thus the data in Table 26.1A becomes Table 26.1B, which can be plotted with `plotSeg` to give Figure 26.1.

Implementation

The outline of `mum2plot` contains hooks for imports, functions, and the logic of the main function.

234a ⟨*mum2plot.go* 234a⟩≡
```
package main

import (
        ⟨Imports, Ch. 26 234c⟩
)
⟨Functions, Ch. 26 235c⟩
func main() {
        ⟨Main function, Ch. 26 234b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

234b ⟨*Main function, Ch. 26* 234b⟩≡ (234a)
```
util.PrepLog("mum2plot")
⟨Set usage, Ch. 26 234d⟩
⟨Declare options, Ch. 26 234f⟩
⟨Parse options, Ch. 26 235a⟩
⟨Parse input files, Ch. 26 235b⟩
```

We import `util`.

234c ⟨*Imports, Ch. 26* 234c⟩≡ (234a) 234e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts. The actual usage message, an explanation of the program's purpose, and an example command.

234d ⟨*Set usage, Ch. 26* 234d⟩≡ (234b)
```
u := "mum2plot [-h -v] [file]..."
p := "Convert MUMmer output to x/y coordinates."
e := "mummer -b -c s1.fasta s2.fasta | mum2plot | plotSeg"
clio.Usage(u, p, e)
```

We import `clio`.

234e ⟨*Imports, Ch. 26* 234c⟩+≡ (234a) ◁234c 234g ▷
```
"github.com/evolbioinf/clio"
```

We declare only the version option.

234f ⟨*Declare options, Ch. 26* 234f⟩≡ (234b)
```
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

234g ⟨*Imports, Ch. 26* 234c⟩+≡ (234a) ◁234e 235d ▷
```
"flag"
```

We parse the options and respond to `-v`.

235a ⟨*Parse options, Ch. 26* 235a⟩≡ (234b)
```
flag.Parse()
if *optV {
        util.PrintInfo("mum2plot")
}
```

The remaining tokens on the command line are taken as file names. Each file is parsed with the function `scan`.

235b ⟨*Parse input files, Ch. 26* 235b⟩≡ (234b)
```
files := flag.Args()
clio.ParseFiles(files, scan)
```

Inside `scan` we create a scanner to go through the file line-wise. We shall need to know whether are in the reverse list, so we reserve a variable for that.

235c ⟨*Functions, Ch. 26* 235c⟩≡ (234a)
```
func scan(r io.Reader, args ...interface{}) {
        sc := bufio.NewScanner(r)
        reverse := false
        for sc.Scan() {
                ⟨Deal with line, Ch. 26 235e⟩
        }
}
```

We import `io` and `bufio`.

235d ⟨*Imports, Ch. 26* 234c⟩+≡ (234a) ◁234g 235f▷
```
"io"
"bufio"
```

We split each line into its fields and decide whether we are dealing with a header or with data. If we are dealing with a header, we need to decide the strand of the list we are currently reading.

235e ⟨*Deal with line, Ch. 26* 235e⟩≡ (235c)
```
fields := strings.Fields(sc.Text())
if fields[0][0] == '>' {
        ⟨Decide strand, Ch. 26 236a⟩
} else {
        ⟨Deal with data, Ch. 26 236b⟩
}
```

We import `strings`.

235f ⟨*Imports, Ch. 26* 234c⟩+≡ (234a) ◁235d 236d▷
```
"strings"
```

If a header ends in `Reverse`, we are in the reverse list, otherwise we aren't. We also accept lower case `reverse` as a marker.

236a ⟨*Decide strand, Ch. 26* 236a⟩≡ (235e)
```
suf := fields[len(fields)-1]
if suf == "Reverse" || suf == "reverse" {
          reverse = true
} else {
          reverse = false
}
```

The first thing we do with a data line is to check the number of fields it contains. Then we extract its start coordinates and length. From that we compute its end coordinates and print them.

236b ⟨*Deal with data, Ch. 26* 236b⟩≡ (235e)
```
⟨Check number of fields, Ch. 26 236c⟩
⟨Extract start coordinates and length, Ch. 26 236e⟩
⟨Compute end coordinates, Ch. 26 236g⟩
⟨Print coordinates, Ch. 26 237a⟩
```

The data line should contain three fields. If it doesn't, we bail and say why.

236c ⟨*Check number of fields, Ch. 26* 236c⟩≡ (236b)
```
if len(fields) != 3 {
          log.Fatal("malformed input")
}
```

We import `log`.

236d ⟨*Imports, Ch. 26* 234c⟩+≡ (234a) ◁235f 236f▷
```
"log"
```

We convert the start position and length from strings to integers.

236e ⟨*Extract start coordinates and length, Ch. 26* 236e⟩≡ (236b)
```
x1, err := strconv.Atoi(fields[0])
if err != nil { log.Fatal(err) }
y1, err := strconv.Atoi(fields[1])
if err !=  nil { log.Fatal(err) }
le, err := strconv.Atoi(fields[2])
if err != nil { log.Fatal(err) }
```

We import `strconv`.

236f ⟨*Imports, Ch. 26* 234c⟩+≡ (234a) ◁236d 237b▷
```
"strconv"
```

We compute the end coordinates for matches, (x_2, y_2). The value of x_2 doesn't depend on the strand, the value of y_2 does.

236g ⟨*Compute end coordinates, Ch. 26* 236g⟩≡ (236b)
```
x2 := x1 + le - 1
y2 := y1 + le - 1
if reverse {
          y2 = y1 - le + 1
}
```

Each line corresponds a quartet of numbers, (x_1, y_1, x_2, y_2).

237a ⟨*Print coordinates, Ch. 26* 237a⟩≡ (236b)
```
fmt.Printf("%d\t%d\t%d\t%d\n", x1, y1, x2, y2)
```

We import `fmt`.

237b ⟨*Imports, Ch. 26* 234c⟩+≡ (234a) ◁236f
```
"fmt"
```

We've finished writing `mum2plot`, so let's test it.

Testing

The outline of our testing program contains hooks for imports and the testing logic.

237c ⟨*mum2plot_test.go* 237c⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 26 237f⟩
)

func TestMum2plot(t *testing.T) {
        ⟨Testing, Ch. 26 237d⟩
}
```

We construct one test and run it.

237d ⟨*Testing, Ch. 26* 237d⟩≡ (237c)
```
⟨Construct test, Ch. 26 237e⟩
⟨Run test, Ch. 26 237g⟩
```

We run `mum2plot` on a small file with MUMmer output, `test.mum`.

237e ⟨*Construct test, Ch. 26* 237e⟩≡ (237d)
```
test := exec.Command("./mum2plot",  "test.mum")
```

We import `exec`.

237f ⟨*Testing imports, Ch. 26* 237f⟩≡ (237c) 238▷
```
"os/exec"
```

We run the test and check we get what we want, which is stored in `r.txt`.

237g ⟨*Run test, Ch. 26* 237g⟩≡ (237d)
```
get, err := test.Output()
if err != nil { t.Error(err.Error()) }
want, err := ioutil.ReadFile("r.txt")
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `ioutil` and `bytes`.

238 ⟨*Testing imports, Ch. 26* 237f⟩+≡ (237c) ◁237f

```
"io/ioutil"
"bytes"
```

Chapter 27

Program `mutator`: Mutate Sequences

Introduction

Much of evolutionary biology is based on sequence comparison, and many of the software tools of the trade take related sequences as input. So it is often useful to transform a sequence into its diverged sibling. The program `mutator` implements this transformation. By default, it mutates positions drawn with replacement with a given probability. However, the user can also opt to supply a list of positions to be mutated, or the number of mutations.

Implementation

The program outline contains hooks for imports, functions, and the logic of the main function.

240a ⟨*mutator.go* 240a⟩≡
```
package main

import (
        ⟨Imports, Ch. 27 240c⟩
)
⟨Functions, Ch. 27 243b⟩
func main() {
        ⟨Main function, Ch. 27 240b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare and parse the options, and parse the input files.

240b ⟨*Main function, Ch. 27* 240b⟩≡ (240a)
```
util.PrepLog("mutator")
⟨Set usage, Ch. 27 240d⟩
⟨Declare options, Ch. 27 241a⟩
⟨Parse options, Ch. 27 241c⟩
⟨Parse input files, Ch. 27 243a⟩
```

We import `util`.

240c ⟨*Imports, Ch. 27* 240c⟩≡ (240a) 240e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three strings, the usage message proper, an explanation of the program's purpose, and an example command.

240d ⟨*Set usage, Ch. 27* 240d⟩≡ (240b)
```
u := "mutator [-h] [options] [fasta file(s)]"
p := "Mutate input sequences."
e := "mutator -p 1,10,100 foo.fasta"
clio.Usage(u, p, e)
```

We import the package `clio`.

240e ⟨*Imports, Ch. 27* 240c⟩+≡ (240a) ◁240c 241b ▷
```
"github.com/evolbioinf/clio"
```

There are six options,

1. `-v` to print the program's version,
2. `-m` to set the mutation rate,
3. `-p` to set a list of positions to be mutated,
4. `-n` to set the number of mutations,
5. `-P` to switch from DNA to protein sequences, and
6. `-s` the seed for the random number generator.

241a ⟨*Declare options, Ch. 27* 241a⟩≡ (240b)
```
var optV = flag.Bool("v", false, "version")
var optM = flag.Float64("m", 0.01, "mutation rate")
var optP = flag.String("p", "", "positions to be mutated; " +
          "comma-separated, one-based")
var optN = flag.Int("n", 0, "number of mutations")
var optPP = flag.Bool("P", false, "protein instead of DNA")
var optS = flag.Int("s", 0, "seed for random number genrator; " +
          "default: internal")
```

We import `flag`.

241b ⟨*Imports, Ch. 27* 240c⟩+≡ (240a) ◁240e 241e▷
```
"flag"
```

The options are parsed and we extract the positions to be mutated, set the residue "alphabet", and set up the random number generator.

241c ⟨*Parse options, Ch. 27* 241c⟩≡ (240b)
```
flag.Parse()
if *optV { util.PrintInfo("mutator") }
⟨Extract Positions, Ch. 27 241d⟩
⟨Set alphabet, Ch. 27 242e⟩
⟨Set up random number generator, Ch. 27 242f⟩
```

A string of positions is split into individual strings, which are converted to integers, checked, and stored.

241d ⟨*Extract Positions, Ch. 27* 241d⟩≡ (241c)
```
var positions []int
if *optP != "" {
          str := strings.Split(*optP, ",")
          for _, ps := range str {
                    ⟨Convert position, Ch. 27 242a⟩
                    ⟨Check and store position, Ch. 27 242c⟩
          }
}
```

We import `strings`.

241e ⟨*Imports, Ch. 27* 240c⟩+≡ (240a) ◁241b 242b▷
```
"strings"
```

If a position cannot be converted, we abort. If converted, it is transformed to zero-based.

242a ⟨*Convert position, Ch. 27* 242a⟩≡ (241d)
```
position, err := strconv.Atoi(ps)
if err != nil {
        log.Fatalf("couldn't convert %q\n", ps)
}
position--
```

We import `strconv` and `log`.

242b ⟨*Imports, Ch. 27* 240c⟩+≡ (240a) ◁241e 242d▷
```
"strconv"
"log"
```

If the position is negative, we warn the user, otherwise we store it.

242c ⟨*Check and store position, Ch. 27* 242c⟩≡ (241d)
```
if position < 0 {
        fmt.Fprintf(os.Stderr, "position %d cannot be mutated\n",
                position+1)
} else {
        positions = append(positions, position)
}
```

We import `fmt` and `os`.

242d ⟨*Imports, Ch. 27* 240c⟩+≡ (240a) ◁242b 242g▷
```
"fmt"
"os"
```

The alphabet can be switched from DNA to protein.

242e ⟨*Set alphabet, Ch. 27* 242e⟩≡ (241c)
```
alphabet := "ACGT"
if *optPP {
        alphabet = "ACDEFGHIKLMNPQRSTVWY"
}
```

If the user gave no seed for the random number generator, a seed is generated from the current time.

242f ⟨*Set up random number generator, Ch. 27* 242f⟩≡ (241c)
```
seed := int64(*optS)
if seed == 0 {
        seed = time.Now().UnixNano()
}
ran := rand.New(rand.NewSource(seed))
```

We import `time` and `rand`.

242g ⟨*Imports, Ch. 27* 240c⟩+≡ (240a) ◁242d 243c▷
```
"time"
"math/rand"
```

Scanning the input files is delegated to the function `ParseFiles`. It takes as arguments the names of the input files, and the name of a function, `scan`, applied to each file. The arguments of `scan` are the alphabet, the number of mutations, the mutation rate, the positions, and the random number generator.

243a ⟨*Parse input files, Ch. 27* 243a⟩≡ (240b)
```
f := flag.Args()
clio.ParseFiles(f, scan, alphabet, *optN, *optM, positions, ran)
```

Inside `scan`, we retrieve the arguments just passed, and iterate over the sequences contained in the file represented by the `Reader`.

243b ⟨*Functions, Ch. 27* 243b⟩≡ (240a) 244c▷
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 27 243d⟩
        ⟨Iterate over sequences, Ch. 27 243e⟩
}
```

We import `io`.

243c ⟨*Imports, Ch. 27* 240c⟩+≡ (240a) ◁242g 243f▷
```
"io"
```

The arguments are retrieved with type assertions.

243d ⟨*Retrieve arguments, Ch. 27* 243d⟩≡ (243b)
```
alphabet := args[0].(string)
n := args[1].(int)
mu := args[2].(float64)
pos := args[3].([]int)
ran := args[4].(*rand.Rand)
```

We iterate over the sequences with a dedicated scanner. For each sequence, we extract the residues, mutate them, and print the mutated sequence.

243e ⟨*Iterate over sequences, Ch. 27* 243e⟩≡ (243b)
```
sc := fasta.NewScanner(r)
for sc.ScanSequence() {
        seq := sc.Sequence()
        res := seq.Data()
        ⟨Mutate residues, Ch. 27 244a⟩
        ⟨Print mutated sequence, Ch. 27 245b⟩
}
```

We import `fasta`.

243f ⟨*Imports, Ch. 27* 240c⟩+≡ (240a) ◁243c
```
"github.com/evolbioinf/fasta"
```

We either generate n mutations, or mutate the residues position-wise, or according to the mutation rate.

244a ⟨*Mutate residues, Ch. 27* 244a⟩≡ (243e)
```
if n > 0 {
        ⟨Generate n mutations, Ch. 27 244b⟩
} else if len(pos) > 0 {
        ⟨Mutate position-wise, Ch. 27 244d⟩
} else {
        ⟨Mutate with rate, Ch. 27 245a⟩
}
```

We generate n mutations, where n can be larger than the number of residues. In that case positions might be mutated more than once.

244b ⟨*Generate n mutations, Ch. 27* 244b⟩≡ (244a)
```
l := len(res)
for i := 0; i < n; i++ {
        p := ran.Intn(l)
        res[p] = mutate(res[p], ran, alphabet)
}
```

The function `mutate` takes as arguments a random number generator, the alphabet, and the residue to be mutated.

244c ⟨*Functions, Ch. 27* 243b⟩+≡ (240a) ◁243b
```
func mutate(res byte, ran *rand.Rand, alphabet string) byte {
        n := len(alphabet)
        new := res
        for new == res {
                p := ran.Intn(n)
                new = alphabet[p]
        }
        return new
}
```

We read the positions to be mutated and carry out the actual mutation with a dedicated function. If the user is trying to mutate a position beyond the end of the sequence, we send a warning.

244d ⟨*Mutate position-wise, Ch. 27* 244d⟩≡ (244a)
```
for _, p := range pos {
        l := len(res)
        if p < l {
                res[p] = mutate(res[p], ran, alphabet)
        } else {
                fmt.Fprintf(os.Stderr, "trying to mutate " +
                        "position %d, but sequence only " +
                        "contains %d residues\n", p+1, l)
        }
}
```

For each residue, we draw a random number and check whether it is less than the mutation rate. If so, we pick a random position in the sequence and mutate it.

245a ⟨*Mutate with rate, Ch. 27* 245a⟩≡ (244a)
```
l := len(res)
for i := 0; i < l; i++ {
        if ran.Float64() < mu {
                r := ran.Intn(l)
                res[r] = mutate(res[r], ran, alphabet)
        }
}
```

To print the mutated sequence, we generate a new `Sequence` object with the mutated residues and append *mutated* to the header. The actual printing is done by the `String` method of `Sequence`.

245b ⟨*Print mutated sequence, Ch. 27* 245b⟩≡ (243e)
```
h := seq.Header() + " - mutated"
ns := fasta.NewSequence(h, res)
fmt.Println(ns)
```

The `mutator` is written, time to test it.

Testing

The testing framework contains hooks for imports and the actual testing logic.

245c ⟨*mutator_test.go* 245c⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 27 246a⟩
)
func TestMutator(t *testing.T) {
        ⟨Testing, Ch. 27 245d⟩
}
```

The testing logic consists in running the program under a variety of conditions and comparing the results we get to the results we want, which are stored in sequentially numbered output files. So we construct a set of commands, a list with the names of the result files, and run each test.

245d ⟨*Testing, Ch. 27* 245d⟩≡ (245c)
```
commands := make([]*exec.Cmd, 0)
⟨Construct commands, Ch. 27 246b⟩
⟨Construct list of result files, Ch. 27 246c⟩
for i, cmd := range commands {
        ⟨Run test, Ch. 27 246e⟩
}
```

We carry out three tests on the file `dna.fa`, which contains two random DNA sequences length 100, a fourth test on `pro.fa`, which contains two human protein sequences, and a fifth test to generate n mutations. In each test we set the seed of the random number gnerator to make the run reproducible.

1. Run with defaults.
2. Mutate positions.
3. Change mutation rate.
4. Mutate protein sequence.
5. Generate n mutations.

Import `exec`.

246a ⟨*Testing imports, Ch. 27* 246a⟩≡ (245c) 246d ▷
```
"os/exec"
```

246b ⟨*Construct commands, Ch. 27* 246b⟩≡ (245d)
```
c := exec.Command("./mutator", "-s", "3", "dna.fa")
commands = append(commands, c)
c = exec.Command("./mutator", "-s", "3", "-p", "0,1,3,100,101",
          "dna.fa")
commands = append(commands, c)
c = exec.Command("./mutator", "-s", "3", "-m", "0.2", "dna.fa")
commands = append(commands, c)
c = exec.Command("./mutator", "-s", "3", "-P", "pro.fa")
commands = append(commands, c)
c = exec.Command("./mutator", "-s", "3", "-n", "2", "dna.fa")
commands = append(commands, c)
```

The results we want are contained in as many files as there are commands.

246c ⟨*Construct list of result files, Ch. 27* 246c⟩≡ (245d)
```
results := make([]string, len(commands))
for i, _ := range commands {
          results[i] = "r" + strconv.Itoa(i+1) + ".fa"
}
```

We import `strconv`.

246d ⟨*Testing imports, Ch. 27* 246a⟩+≡ (245c) ◁246a 247 ▷
```
"strconv"
```

For each test we compare what we get with what we want.

246e ⟨*Run test, Ch. 27* 246e⟩≡ (245d)
```
get, err := cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
want, err := ioutil.ReadFile(results[i])
if err != nil {
          t.Errorf("couldn't open %q\n", results[i])
}
if !bytes.Equal(get, want) {
          t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `ioutil` and `bytes`.

247 ⟨*Testing imports, Ch. 27* 246a⟩+≡ (245c) ◁246d

```
"io/ioutil"
"bytes"
```

Chapter 28

Program `naiveMatcher`: Match Pattern in Text

Introduction

The naïve method for finding a pattern, p, in a text, t, is to write two nested loops. The outer iterates over t, the inner over p. If the inner finishes, p has been found. The program `naiveMatcher` implements this algorithm.

Implementation

The outline of `naiveMatcher` contains hooks for imports, variables, functions, and the main function.

249a ⟨*naiveMatcher.go* 249a⟩≡
```
package main

import (
        ⟨Imports, Ch. 28 249c⟩
)
⟨Variables, Ch. 28 249f⟩
⟨Functions, Ch. 28 251a⟩
func main() {
        ⟨Main function, Ch. 28 249b⟩
}
```

In the main function, we prepare the `log` package, set the usage, parse the options, and scan the input files.

249b ⟨*Main function, Ch. 28* 249b⟩≡ (249a)
```
util.PrepLog("naiveMatcher")
⟨Set usage, Ch. 28 249d⟩
⟨Parse options, Ch. 28 250b⟩
⟨Scan input files, Ch. 28 250f⟩
```

We import `util`.

249c ⟨*Imports, Ch. 28* 249c⟩≡ (249a) 249e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of a usage message, an explanation of the program's purpose, and an example usage.

249d ⟨*Set usage, Ch. 28* 249d⟩≡ (249b)
```
u := "naiveMatcher [-h] [options] pattern [file(s)]"
p := "Demonstrate naive matching algorithm."
e := "naiveMatcher ATTGC foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

249e ⟨*Imports, Ch. 28* 249c⟩+≡ (249a) ◁ 249c 250a ▷
```
"github.com/evolbioinf/clio"
```

We declare two options, `-v` to print the program version, and `-p` to enter a file of patterns.

249f ⟨*Variables, Ch. 28* 249f⟩≡ (249a)
```
var optV = flag.Bool("v", false, "version")
var optP = flag.String("p", "", "file of patterns")
```

We import `flag>`

250a ⟨*Imports, Ch. 28* 249c⟩+≡ (249a) ◁249e 250d▷
```
"flag"
```

When parsing the options, we check for `-v`, and get the pattern or pattern file and the input files.

250b ⟨*Parse options, Ch. 28* 250b⟩≡ (249b)
```
flag.Parse()
if *optV {
        util.PrintInfo("naiveMatcher")
}
⟨Get pattern or pattern file, Ch. 28 250c⟩
⟨Get input files, Ch. 28 250e⟩
```

The pattern is the first entry in the argument slice, unless a pattern file was given.

250c ⟨*Get pattern or pattern file, Ch. 28* 250c⟩≡ (250b)
```
p = ""
a := flag.Args()
if *optP == "" {
        if len(a) < 1 {
                fmt.Fprintf(os.Stderr, "please enter a pattern " +
                        "or a pattern file via -p\n")
                os.Exit(0)
        }
        p = a[0]
}
```

We import `fmt` and `os`.

250d ⟨*Imports, Ch. 28* 249c⟩+≡ (249a) ◁250a 251b▷
```
"fmt"
"os"
```

The input files are the remaining elements in the argument slice.

250e ⟨*Get input files, Ch. 28* 250e⟩≡ (250b)
```
var f []string
if p == "" {
        f = a[0:]
} else {
        f = a[1:]
}
```

The pattern is searched in all input files using the function `scan`, which takes as argument the pattern, p, and the name of the pattern file.

250f ⟨*Scan input files, Ch. 28* 250f⟩≡ (249b)
```
clio.ParseFiles(f, scan, p, *optP)
```

In `scan`, we first retrieve the arguments of `scan`, then get one or more patterns, iterate over the text sequences, and search for the pattern(s) in them.

251a ⟨*Functions, Ch. 28* 251a⟩≡ (249a)
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 28 251c⟩
        ⟨Get patterns, Ch. 28 251d⟩
        textSc := fasta.NewScanner(r)
        for textSc.ScanSequence() {
                t := textSc.Sequence().Data()
                th := textSc.Sequence().Header()
                ⟨Iterate over patterns, Ch. 28 251e⟩
        }
}
```

We import `io` and `fasta`.

251b ⟨*Imports, Ch. 28* 249c⟩+≡ (249a) ◁250d
```
"io"
"github.com/evolbioinf/fasta"
```

The arguments consist of a pattern read from the command line and a pattern file.

251c ⟨*Retrieve arguments, Ch. 28* 251c⟩≡ (251a)
```
pc := args[0].(string)
pfn := args[1].(string)
```

Patterns are read either from the command line or from the pattern file.

251d ⟨*Get patterns, Ch. 28* 251d⟩≡ (251a)
```
ps := make([]fasta.Sequence, 0)
if pc != "" {
        ps = append(ps, *fasta.NewSequence(pc, []byte(pc)))
} else {
        pf, err := os.Open(pfn)
        if err != nil { fmt.Errorf("couldn't open %q\n", pfn) }
        sc := fasta.NewScanner(pf)
        for sc.ScanSequence() {
                ps = append(ps, *sc.Sequence())
        }
        pf.Close()
}
```

Given a text, we iterate over the patterns, write a comment line identifying both, followed by the positions of the pattern in the text.

251e ⟨*Iterate over patterns, Ch. 28* 251e⟩≡ (251a)
```
for _, pattern := range ps {
        p := pattern.Data()
        fmt.Printf("# %s / %s\n", pattern.Header(), th)
        ⟨Search for pattern, Ch. 28 252a⟩
}
```

The pattern search consists of a nested loop. Whenever a pattern is found, its starting position is printed. Positions are one-based.

252a ⟨*Search for pattern, Ch. 28* 252a⟩≡ (251e)

```
j := 0
m := len(t) - len(p) + 1
n := len(p)
for i := 0; i < m; i++ {
        for j = 0; j < n; j++ {
                if t[i+j] != p[j] { break }
        }
        if j == len(p) {
                fmt.Println(i+1)
        }
}
```

The `naiveMatcher` is written, time to test it.

Testing

The testing outline contains hooks for imports and the testing logic.

252b ⟨*naiveMatcher_test.go* 252b⟩≡

```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 28 253a⟩
)

func TestNaiveMatcher(t *testing.T) {
        ⟨Testing, Ch. 28 252c⟩
}
```

We search in the alcohol dehydrogenase locus of two `Drosophila` species, `D. melanogaster` and `D. guanche`. In the first test, the pattern `ATTA` is passed on the command line. The result we want is contained in `r1.txt`, which we compare to what we get.

252c ⟨*Testing, Ch. 28* 252c⟩≡ (252b) 253b ▷

```
cmd := exec.Command("./naiveMatcher", "ATTA",
        "dmAdhAdhdup.fasta", "dgAdhAdhdup.fasta")
get, err := cmd.Output()
if err != nil { t.Errorf("couldn't run %q\n", cmd) }
want, err := ioutil.ReadFile("r1.txt")
if err != nil { t.Errorf("couldn't open r1.txt\n") }
if !bytes.Equal(get, want) {
        t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `exec`, `ioutil`, and `bytes`.

253a ⟨*Testing imports, Ch. 28* 253a⟩≡ (252b)
```
"os/exec"
"io/ioutil"
"bytes"
```

In the second and final test, two patterns, `ATTA` and `ATTT` are passed via `p.fasta`. The result we want is in `r2.txt`

253b ⟨*Testing, Ch. 28* 252c⟩+≡ (252b) ◁252c
```
cmd = exec.Command("./naiveMatcher", "-p", "p.fasta",
          "dmAdhAdhdup.fasta", "dgAdhAdhdup.fasta")
get, err = cmd.Output()
if err != nil { t.Errorf("couldn't run %q\n", cmd) }
want, err = ioutil.ReadFile("r2.txt")
if err != nil { t.Errorf("couldn't open r1.txt\n") }
if !bytes.Equal(get, want) {
          t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

Chapter 29

Program `num2char`: Convert Numbers to Characters

Introduction

The program `mtf` creates a stream of numbers under the move to front algorithm. This output makes it easy to trace the move to front method, but difficult for character-based programs to parse the result. The program `num2char` bridges this gap by translating the numbers 0–127 into letters and digits. The numbers are presented in the FASTA format generated by the program `mtf`. For example, the input

```
>example - mtf
2 1 3 2 1 1 3 0 0 2 3 3 0 2 1 1 1 1 3 0
```

is converted by `num2char` into

```
>example - mtf - num2char
#"$#""$!!#$$!#"""""$!
```

`num2char` can also reverse this step by decoding characters into numbers.

Implementation

The implementation of `mtf` has hooks for imports, functions, and the logic of the main function.

255a ⟨*num2char.go* 255a⟩≡

```
package main

import (
        ⟨Imports, Ch. 29 255c⟩
)
⟨Functions, Ch. 29 257d⟩
func main() {
        ⟨Main function, Ch. 29 255b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

255b ⟨*Main function, Ch. 29* 255b⟩≡ (255a)

```
util.PrepLog("num2char")
⟨Set usage, Ch. 29 256a⟩
⟨Declare options, Ch. 29 256c⟩
⟨Parse options, Ch. 29 256e⟩
⟨Parse input files, Ch. 29 257c⟩
```

We import `util`.

255c ⟨*Imports, Ch. 29* 255c⟩≡ (255a) 256b ▷

```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `num2char`, and an example command.

256a ⟨*Set usage, Ch. 29* 256a⟩≡ (255b)
```
u := "num2char [-h] [option]... [file]..."
p := "Convert FASTA-formatted numbers 0-127 to " +
        "printable characters."
e := "mtf foo.fasta | num2char"
clio.Usage(u, p, e)
```

We import `clio`.

256b ⟨*Imports, Ch. 29* 255c⟩+≡ (255a) ◁255c 256d▷
```
"github.com/evolbioinf/clio"
```

We declare an option for printing the version (`-v`) and an option for decoding characters into numbers (`-d`).

256c ⟨*Declare options, Ch. 29* 256c⟩≡ (255b)
```
var optV = flag.Bool("v", false, "version")
var optD = flag.Bool("d", false, "decode")
```

We import `flag`.

256d ⟨*Imports, Ch. 29* 255c⟩+≡ (255a) ◁256b 257a▷
```
"flag"
```

We parse the two options and respond to `-v` first, as this would stop the program. Then we construct the encoding dictionary. If the user chose decoding, we convert the encoding dictionary into its decoding version.

256e ⟨*Parse options, Ch. 29* 256e⟩≡ (255b)
```
flag.Parse()
dict := make(map[string]string)
if *optV {
        util.PrintInfo("num2char")
}
⟨Construct encoding dictionary, Ch. 29 256f⟩
if *optD {
        ⟨Construct decoding dictionary, Ch. 29 257b⟩
}
```

In the encoding dictionary we map string representations of the numbers 0–127 to letters and digits

256f ⟨*Construct encoding dictionary, Ch. 29* 256f⟩≡ (256e)
```
j := 0
for i := 0; i < 128; i++ {
        for i+j < 256 {
                r := rune(i + j)
                if unicode.IsLetter(r) || unicode.IsDigit(r) {
                        dict[strconv.Itoa(i)] = string(r)
                        break
                }
                j++
        }
}
```

We import `unicode` and `strconv`.

257a ⟨*Imports, Ch. 29* 255c⟩+≡ (255a) ◁256d 257e▷
```
"unicode"
"strconv"
```

In the decoding dictionary the values of the encoding dictionary are mapped onto its keys.

257b ⟨*Construct decoding dictionary, Ch. 29* 257b⟩≡ (256e)
```
nd := make(map[string]string)
for i := 0; i < 128; i++ {
        s := strconv.Itoa(i)
        nd[dict[s]] = s
}
dict = nd
```

The remaining tokens on the command line are interpreted as the names of input files. These are parsed with the function `scan`, which also takes the `-d` option and the dictionary as arguments.

257c ⟨*Parse input files, Ch. 29* 257c⟩≡ (255b)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optD, dict)
```

Inside `scan`, we retrieve the `-d` option through type assertion and scan the lines contained in the file.

257d ⟨*Functions, Ch. 29* 257d⟩≡ (255a) 258b▷
```
func scan(r io.Reader, args ...interface{}) {
        dec := args[0].(bool)
        dict := args[1].(map[string]string)
        sc := fasta.NewScanner(r)
        ⟨Scan lines, Ch. 29 257f⟩
}
```

We import `io` and `fasta`.

257e ⟨*Imports, Ch. 29* 255c⟩+≡ (255a) ◁257a 258d▷
```
"io"
"github.com/evolbioinf/fasta"
```

We scan the lines and deal with each one. After the end of the file we flush the scanner and deal with the "line" returned.

257f ⟨*Scan lines, Ch. 29* 257f⟩≡ (257d)
```
for sc.ScanLine() {
        line := sc.Line()
        ⟨Deal with line, Ch. 29 258a⟩
}
line := sc.Flush()
⟨Deal with line, Ch. 29 258a⟩
```

A line is either decoded or encoded, which we both delegate to function calls.

258a ⟨*Deal with line, Ch. 29* 258a⟩≡ (257f)
```
if dec {
        decode(line, dict)
} else {
        encode(line, dict)
}
```

Inside decode we first check there is some data to process. This is either a header or decode a line.

258b ⟨*Functions, Ch. 29* 257d⟩+≡ (255a) ◁257d 259a▷
```
func decode(data []byte, dict map[string]string) {
        if len(data) == 0 { return }
        if data[0] == '>' {
                ⟨Decode header, Ch. 29 258c⟩
        } else {
                ⟨Decode data, Ch. 29 258e⟩
        }
}
```

We decode the header by printing it with a note about the transformation appended.

258c ⟨*Decode header, Ch. 29* 258c⟩≡ (258b)
```
h := string(data) + " - num2char -d"
fmt.Printf("%s\n", h)
```

We import `fmt`.

258d ⟨*Imports, Ch. 29* 255c⟩+≡ (255a) ◁257e 258f▷
```
"fmt"
```

A line to be decoded consists of characters that we convert to strings representing numbers. We print these strings delimited by blanks. first construct a slice of number strings, which we then convert. We bail if we cannot look up an input string and close the line with a newline.

258e ⟨*Decode data, Ch. 29* 258e⟩≡ (258b)
```
l := len(data) - 1
for i, c := range data {
        k := string(c)
        v, ok := dict[k]
        if !ok { log.Fatalf("cannot decode %s", k) }
        fmt.Printf("%s", v)
        if i < l { fmt.Printf(" ") }
}
fmt.Printf("\n")
```

We import `log`.

258f ⟨*Imports, Ch. 29* 255c⟩+≡ (255a) ◁258d 259d▷
```
"log"
```

Inside `encode` we again first check we have data to process. Then we encode either a header or data.

259a ⟨*Functions, Ch. 29* 257d⟩+≡ (255a) ◁258b
```
func encode(data []byte, dict map[string]string) {
        if len(data) == 0 { return }
        if data[0] == '>' {
                ⟨Encode header, Ch. 29 259b⟩
        } else {
                ⟨Encode data, Ch. 29 259c⟩
        }
}
```

We encode a header by printing it with a note of the transformation appended.

259b ⟨*Encode header, Ch. 29* 259b⟩≡ (259a)
```
h := string(data) + " - num2char"
fmt.Printf("%s\n", h)
```

Data to be encoded consists of strings that we convert to characters. The strings make up fields in the byte slice holding the data. We extract these fields, iterate over them, and print the corresponding characters. As before, we mark the transformation at the end of its header and end the line with a newline.

259c ⟨*Encode data, Ch. 29* 259c⟩≡ (259a)
```
bs := bytes.Fields(data)
for _, n := range bs {
        v , ok := dict[string(n)]
        if !ok { log.Fatalf("cannot encode %s", string(n)) }
        fmt.Printf("%s", v)
}
fmt.Printf("\n")
```

We import `bytes`.

259d ⟨*Imports, Ch. 29* 255c⟩+≡ (255a) ◁258f
```
"bytes"
```

We've finished writing `num2char`, time to test it.

Testing

The outline of our testing code contains hooks for imports and the testing logic.

259e ⟨*num2char_test.go* 259e⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 29 260b⟩
)

func TestNum2char(t *testing.T) {
        ⟨Testing, Ch. 29 260a⟩
}
```

We construct the tests and run them.

260a ⟨*Testing, Ch. 29* 260a⟩≡ (259e)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 29 260c⟩
for i, test := range tests {
        ⟨Run test, Ch. 29 260d⟩
}
```

We import `exec`.

260b ⟨*Testing imports, Ch. 29* 260b⟩≡ (259e) 260e ▷
```
"os/exec"
```

We construct two tests, one to run `num2char` in default encoding mode, one to run it in decoding mode. The input to the first test is some small output generated with `mtf`, the input to the second test is the expected result of the first.

260c ⟨*Construct tests, Ch. 29* 260c⟩≡ (260a)
```
test := exec.Command("./num2char", "mtf.fasta")
tests = append(tests, test)
test = exec.Command("./num2char", "-d", "r1.txt")
tests = append(tests, test)
```

We store the result we get with the result we want, which is stored in files `r1.txt` and `r2.txt`.

260d ⟨*Run test, Ch. 29* 260d⟩≡ (260a)
```
get, err := test.Output()
if err != nil { t.Errorf("couldn't run %s", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("couldn't open %q", f) }
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

260e ⟨*Testing imports, Ch. 29* 260b⟩+≡ (259e) ◁260b
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 30

Program `numAl`: Number of Global Alignments

Introduction

Given two sequences of a single nucleotide, $S_1 =$ `A` and $S_2 =$ `T`, there are three possible alignments:

```
-     A     A
T     -     T
```

Since every alignment of two sequences ends in one of these three configurations—gap/residue, residue/gap, or residue/residue—the number of possible global alignments of two sequences of lengths m and n can be expressed recursively as the sum of three terms [14, p.32f]: The number of global alignments between sequences of lengths $m-1$ and n, between sequences of lengths m and $n-1$, and between sequences of lengths $m-1, n-1$:

$$f(m,n) = f(m-1,n) + f(m,n-1) + f(m-1,n-1) \tag{30.1}$$

Perhaps confusingly, the same function, f, appears on the left and the right of this equation. Such self-referential equations are called *recursions*. To prevent them from going on for ever, they need a stopping criterion. In our case, whenever one of the sequences has length zero, that is, consists only of gaps, there is only one possible alignment,

$$f(m,0) = f(0,n) = f(0,0) = 1. \tag{30.2}$$

This is the stopping criterion, or boundary condition, for our recursion.

There are two ways to compute $f(m,n)$, top down by direct recursion, and bottom up by working from the boundary conditions and saving the intermediate results. Say, we wish to compute $f(2,3)$; we begin by writing down a programming matrix of $(m+1) \times (n+1)$ cells:

	0	1	2	3
0				
1				
2				

We initialize the matrix according to the boundary condition by filling the first row and column with 1:

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

Then we fill-in the remaining blanks by summing their three neighbors:

	0	1	2	3
0	1	1	1	1
1	1	3	5	7
2	1	5	13	25

The number of global alignments between two sequences of lengths 2 and 3 is the entry in the bottom right-hand corner, 25. The program `numAl` implements this bottom-up method as its default. The run-time of the top-down method is proportional to its result, which quickly becomes very large with longer sequences. We include it to demonstrate the value of the bottom-up approach when solving recursions. By way of demonstration the user can also print the data structure underlying the chosen computation, the recursion tree or the programming matrix.

Implementation

The program layout contains hooks for imports, types, variables, functions, and the logic of the main function.

263a ⟨*numAl.go* 263a⟩≡
```
package main

import (
        ⟨Imports, Ch. 30 263c⟩
)
⟨Types, Ch. 30 267a⟩
⟨Variables, Ch. 30 264a⟩
⟨Functions, Ch. 30 265c⟩
func main() {
        ⟨Main function, Ch. 30 263b⟩
}
```

In the main function we prepare the `log` package, set the usage, parse the options, and compute the number of alignments or print the data structure underlying the computation.

263b ⟨*Main function, Ch. 30* 263b⟩≡ (263a)
```
util.PrepLog("numAl")
⟨Set usage, Ch. 30 263d⟩
⟨Parse options, Ch. 30 264c⟩
if !*optP {
        ⟨Compute number of alignments, Ch. 30 265a⟩
} else {
        ⟨Print data structure, Ch. 30 266c⟩
}
```

We import `util`.

263c ⟨*Imports, Ch. 30* 263c⟩≡ (263a) 263e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the usage message itself, a statement of the program's purpose, and an example command.

263d ⟨*Set usage, Ch. 30* 263d⟩≡ (263b)
```
u := "numAl [-h] [options] m n"
p := "Compute the number of possible global alignments " +
        "between two sequences of lengths m and n"
e := "numAl 5 10"
clio.Usage(u, p, e)
```

We import the package `clio`.

263e ⟨*Imports, Ch. 30* 263c⟩+≡ (263a) ◁263c 264b ▷
```
"github.com/evolbioinf/clio"
```

As to options, there is the standard version switch, `-v`. In addition, the user can request top-down computation instead of bottom-up (`-t`).

264a ⟨*Variables, Ch. 30* 264a⟩≡ (263a) 267c ▷
```
var optV = flag.Bool("v", false, "version")
var optT = flag.Bool("t", false, "top-down (default bottom-up)")
var optP = flag.Bool("p", false, "print data structure (default result)")
```

We import `flag`.

264b ⟨*Imports, Ch. 30* 263c⟩+≡ (263a) ◁263e 264d ▷
```
"flag"
```

We parse the flags, check for version printing, extract the sequence lengths, and warn the user if one or both of them are missing.

264c ⟨*Parse options, Ch. 30* 264c⟩≡ (263b)
```
flag.Parse()
if *optV {
        util.PrintInfo("numAl")
}
args := flag.Args()
if len(args) < 2 {
        fmt.Fprintf(os.Stderr, "please provide two " +
                "sequence lengths\n")
        os.Exit(0)
}
⟨Convert lengths, Ch. 30 264e⟩
```

We import `fmt` and `os`.

264d ⟨*Imports, Ch. 30* 263c⟩+≡ (263a) ◁264b 264f ▷
```
"fmt"
"os"
```

Conversion of the arguments is accompanied by error checking.

264e ⟨*Convert lengths, Ch. 30* 264e⟩≡ (264c)
```
m, err := strconv.Atoi(args[0])
if err != nil {
        fmt.Fprintf(os.Stderr, "couldn't convert %q\n", args[0])
        os.Exit(0)
}
n, err := strconv.Atoi(args[1])
if err != nil {
        fmt.Fprintf(os.Stderr, "couldn't convert %q\n", args[1])
        os.Exit(0)
}
```

We import `strconv`.

264f ⟨*Imports, Ch. 30* 263c⟩+≡ (263a) ◁264d 265b ▷
```
"strconv"
```

The actual computation depends on whether top-down or bottom-up is requested. We time this part of the program to show what a difference algorithms can make.

265a ⟨*Compute number of alignments, Ch. 30* 265a⟩≡ (263b)
```
var du time.Duration
var na float64
start := time.Now()
if *optT {
        na = topDown(m, n)
} else {
        mat := bottomUp(m, n)
        na = mat[m][n]
}
end := time.Now()
du = end.Sub(start)
fmt.Printf("f(%d, %d) = %g (%g s)\n", m, n, na, du.Seconds())
```

We import `time`.

265b ⟨*Imports, Ch. 30* 263c⟩+≡ (263a) ◁264f 268a▷
```
"time"
```

For the top-down approach we simply rephrase equations (30.1) and (30.2).

265c ⟨*Functions, Ch. 30* 265c⟩≡ (263a) 265d▷
```
func topDown(m, n int) float64 {
        if m > 0 && n > 0 {
                r := topDown(m-1, n) + topDown(m, n-1) +
                        topDown(m-1, n-1)
                return r
        } else {
                return 1.0
        }
}
```

In the bottom-up calculation, we construct a matrix, initialize it, fill it in, and return it.

265d ⟨*Functions, Ch. 30* 265c⟩+≡ (263a) ◁265c 267b▷
```
func bottomUp(m, n int) [][]float64 {
        var mat [][]float64
        ⟨Construct matrix, Ch. 30 265e⟩
        ⟨Initialize matrix, Ch. 30 266a⟩
        ⟨Fill-in matrix, Ch. 30 266b⟩
        return mat
}
```

The matrix is constructed as a slice of slices.

265e ⟨*Construct matrix, Ch. 30* 265e⟩≡ (265d)
```
mat = make([][]float64, m+1)
for i := 0; i <= m; i++ {
        mat[i] = make([]float64, n+1)
}
```

We fill the first row and column with 1's.

266a ⟨*Initialize matrix, Ch. 30* 266a⟩≡ (265d)
```
for i := 0; i <= m; i++ {
        mat[i][0] = 1
}
for i := 1; i <= n; i++ {
        mat[0][i] = 1
}
```

We fill-in the rest of the matrix.

266b ⟨*Fill-in matrix, Ch. 30* 266b⟩≡ (265d)
```
for i := 1; i <= m; i++ {
        for j := 1; j <= n; j++ {
                mat[i][j] = mat[i-1][j] +
                        mat[i][j-1] +
                        mat[i-1][j-1]
        }
}
```

Instead of calculating and printing the result, we can also print the data structure used in the calculation. This is either the recursion tree or the programming matrix.

266c ⟨*Print data structure, Ch. 30* 266c⟩≡ (263b)
```
if *optT {
        ⟨Print recursion tree, Ch. 30 266d⟩
} else {
        mat := bottomUp(m, n)
        ⟨Print programming matrix, Ch. 30 267f⟩
}
```

We print the recursion tree as a graph in dot notation. This has a header, a main body generated by calling a recursion, and a footer.

266d ⟨*Print recursion tree, Ch. 30* 266d⟩≡ (266c)
```
⟨Print graph header, Ch. 30 266e⟩
r := newNode(m, n)
printTopDown(r)
⟨Print graph footer, Ch. 30 267e⟩
```

We start the header with a comment saying where the graph comes from and how it can be rendered. Then we declare a directed graph, declare the nodes as points and remove the arrow heads from the edges.

266e ⟨*Print graph header, Ch. 30* 266e⟩≡ (266d)
```
fmt.Println("# Recursion tree for computing the number of alignments.")
fmt.Println("# Generated with numAl, render with")
fmt.Println("# $ dot -T x11 foo.dot")
fmt.Println("digraph g {")
fmt.Println("\tnode[shape=point]")
fmt.Println("\tedge[arrowhead=none]")
```

A node in the recursion tree has a value for m, n, and an identifier.

267a ⟨*Types, Ch. 30* 267a⟩≡ (263a)
```
type node struct {
        m, n, i int
}
```

A new node is created as a function of m and n. It is also assigned a unique identifier.

267b ⟨*Functions, Ch. 30* 265c⟩+≡ (263a) ◁265d 267d▷
```
func newNode(m, n int) *node {
        v := new(node)
        v.m = m
        v.n = n
        v.i = nodeId
        nodeId++
        return v
}
```

We declare `nodeId` as a global variable.

267c ⟨*Variables, Ch. 30* 264a⟩+≡ (263a) ◁264a
```
var nodeId int
```

In the recursion we check whether we've reached a leaf. If not, we create three children, print them, and recurse into them.

267d ⟨*Functions, Ch. 30* 265c⟩+≡ (263a) ◁267b
```
func printTopDown(v *node) {
        if v.m == 0 || v.n == 0 { return }
        c1 := newNode(v.m-1, v.n)
        c2 := newNode(v.m, v.n-1)
        c3 := newNode(v.m-1, v.n-1)
        fmt.Printf("\tn%d->n%d\n", v.i, c1.i)
        fmt.Printf("\tn%d->n%d\n", v.i, c2.i)
        fmt.Printf("\tn%d->n%d\n", v.i, c3.i)
        printTopDown(c1)
        printTopDown(c2)
        printTopDown(c3)
}
```

The graph footer us just a closing curly bracket.

267e ⟨*Print graph footer, Ch. 30* 267e⟩≡ (266d)
```
fmt.Println("}")
```

The programming matrix is a table, so we print it using a `tabwriter`. First we print the first row of the programming matrix, then its remainder. At the end we flush the writer.

267f ⟨*Print programming matrix, Ch. 30* 267f⟩≡ (266c)
```
w := tabwriter.NewWriter(os.Stdout, 1, 1, 1, ' ',
        tabwriter.AlignRight)
⟨Print first row of programming matrix, Ch. 30 268b⟩
⟨Print rest of progamming matrix, Ch. 30 268c⟩
w.Flush()
```

We import `tabwriter`.

268a ⟨*Imports, Ch. 30* 263c⟩+≡ (263a) ◁265b
```
"text/tabwriter"
```

The first row of the programming matrix are its column indexes.

268b ⟨*Print first row of programming matrix, Ch. 30* 268b⟩≡ (267f)
```
for i := 0; i <= n; i++ {
        fmt.Fprintf(w, "\t%d", i)
}
fmt.Fprintf(w, "\t\n")
```

We print the rest of the matrix.

268c ⟨*Print rest of progamming matrix, Ch. 30* 268c⟩≡ (267f)
```
for i := 0; i <= m; i++ {
        fmt.Fprintf(w, "%d", i)
        for j := 0; j <= n; j++ {
                fmt.Fprintf(w, "\t%d", int(mat[i][j]))
        }
        fmt.Fprintf(w, "\t\n")
}
```

We're done writing `numAl`, here comes the test.

Testing

The testing outline contains hooks for imports and the testing logic itself.

268d ⟨*numAl_test.go* 268d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 30 268f⟩
)

func TestNumAl(t *testing.T) {
        ⟨Testing, Ch. 30 268e⟩
}
```

We construct a set of tests and run them.

268e ⟨*Testing, Ch. 30* 268e⟩≡ (268d)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 30 269a⟩
for i, test := range tests {
        ⟨Run test, Ch. 30 269b⟩
}
```

We import `exec`.

268f ⟨*Testing imports, Ch. 30* 268f⟩≡ (268d) 269c▷
```
"os/exec"
```

We construct four tests, bottom up, top down, and both with printing.

269a ⟨*Construct tests, Ch. 30* 269a⟩≡ (268e)
```
test := exec.Command("./numAl", "10", "10")
tests = append(tests, test)
test = exec.Command("./numAl", "-t", "10", "10")
tests = append(tests, test)
test = exec.Command("./numAl", "-p", "3", "3")
tests = append(tests, test)
test = exec.Command("./numAl", "-p", "-t", "3", "3")
tests = append(tests, test)
```

We run a test and compare the result we get with the result we want, which is stored in files `r1.txt`, `r2.txt`, and so on. The first two test cases include time measurements in their output. This cannot be reproduced reliably, so we cut it off.

269b ⟨*Run test, Ch. 30* 269b⟩≡ (268e)
```
get, err := test.Output()
if i < 2 { get = get[:24] }
if err != nil { t.Errorf("couldn't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("couldn't open %q", f) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

269c ⟨*Testing imports, Ch. 30* 268f⟩+≡ (268d) ◁268f
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 31

Program `nj`: Compute Neighbor-Joining Tree

A

5					
S1	0	0.19	0.00502	0.0537	0.186
S2	0.19	0	0.19	0.188	0.0489
S3	0.00502	0.19	0	0.0539	0.186
S4	0.0537	0.188	0.0539	0	0.184
S5	0.186	0.0489	0.186	0.184	0

B

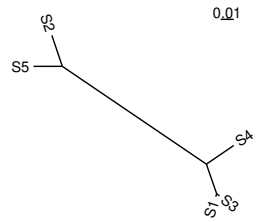


Figure 31.1: A distance matrix (**A**) is transformed by the program `nj` into a tree, which can be plotted using the program `plotTree` (**B**).

Phylogeny Reconstruction

Phylogeny reconstruction from distance matrices is often done using one of two methods, UPGMA and neighbor joining. The UPGMA method is implemented in the program `upgma` described in Chapter 55, while we now implement the neighbor joining method in the program `nj`. `nj` takes as input a distance matrix like the one in Figure 31.1A and returns the corresponding tree in Newick format, plotted as Figure 31.1B.

We implement the neighbor-joining algorithm as described in [14, p. 110f]. Given the $n \times n$ distance matrix d,

- compute the row sums of d
$$r_i = \sum_j d_{ij}$$
- compute a supplementary matrix, s,
$$s_{ij} = d_{ij} - (r_i + r_j)/(n-2)$$
- cluster a pair of taxa with smallest s_{ij} in node c with
$$d_{kc} = (d_{ik} + d_{jk} - d_{ij})/2$$
- calculate the branch lengths from the new cluster, c, to its children, i and j,
$$d_{ic} = \frac{(n-2)d_{ij} + r_i - r_j}{2(n-2)}$$
$$d_{jc} = \frac{(n-2)d_{ij} + r_j - r_i}{2(n-2)}$$

Repeat this until there are only three clusters left, call them i, j, k. Then form the root, r, and add the remaining clusters as its children with branch lengths

$$d_{ir} = (d_{ij} + d_{ik} - d_{jk})/2$$
$$d_{jr} = (d_{ji} + d_{jk} - d_{ik})/2$$
$$d_{kr} = (d_{ki} + d_{kj} - d_{ij})/2$$

Implementation

The outline of `nj` has hooks for imports, functions, and the logic of the main function.

272a ⟨*nj.go* 272a⟩≡
```
package main

import (
        ⟨Imports, Ch. 31 272c⟩
)

⟨Functions, Ch. 31 273d⟩
func main() {
        ⟨Main function, Ch. 31 272b⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

272b ⟨*Main function, Ch. 31* 272b⟩≡ (272a)
```
util.PrepLog("nj")
⟨Set usage, Ch. 31 272d⟩
⟨Declare options, Ch. 31 272f⟩
⟨Parse options, Ch. 31 273b⟩
⟨Parse input files, Ch. 31 273c⟩
```

We import `util`.

272c ⟨*Imports, Ch. 31* 272c⟩≡ (272a) 272e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of `nj`'s purpose, and an example command.

272d ⟨*Set usage, Ch. 31* 272d⟩≡ (272b)
```
u := "nj [-h] [option]... [foo.dist]..."
p := "Calculate neighbor-joining tree."
e := "nj foo.dist"
clio.Usage(u, p, e)
```

We import `clio`.

272e ⟨*Imports, Ch. 31* 272c⟩+≡ (272a) ◁ 272c 273a ▷
```
"github.com/evolbioinf/clio"
```

Apart from the version (`-v`), we declare an option for printing the intermediate distance matrices, `-m`. Also, the neighbor-joining algorithm allows negative branch lengths. These make little biological sense and are usually set to zero. However, users might be interested in the result of "pure" neighbor joining, hence we also declare an option for allowing negative branch lengths.

272f ⟨*Declare options, Ch. 31* 272f⟩≡ (272b)
```
var optV = flag.Bool("v", false, "version")
var optM = flag.Bool("m", false, "print intermediate " +
          "matrices")
var optN = flag.Bool("n", false, "allow negative branch lengths")
```

We import `flag`.

273a ⟨*Imports, Ch. 31* 272c⟩+≡ (272a) ◁272e 273e▷
```
"flag"
```

We parse the options and respond to `-v`, as this terminates the program.

273b ⟨*Parse options, Ch. 31* 273b⟩≡ (272b)
```
flag.Parse()
if *optV {
        util.PrintInfo("nj")
}
```

The remaining tokens on the command line are taken as the names of input files. These are parsed using the function `scan`, which in turn takes the options `-m` and `-n` as arguments.

273c ⟨*Parse input files, Ch. 31* 273c⟩≡ (272b)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optM, *optN)
```

Inside `scan`, we retrieve `-m` and `-n`, and iterate over the distance matrices in the input.

273d ⟨*Functions, Ch. 31* 273d⟩≡ (272a) 274b▷
```
func scan(r io.Reader, args ...interface{}) {
        printMat := args[0].(bool)
        negBr := args[1].(bool)
        sc := dist.NewScanner(r)
        for sc.Scan() {
                dm := sc.DistanceMatrix()
                ⟨Process distance matrix, Ch. 31 273f⟩
        }
}
```

We import `io` and `dist`.

273e ⟨*Imports, Ch. 31* 272c⟩+≡ (272a) ◁273a 273g▷
```
"io"
"github.com/evolbioinf/dist"
```

We make the distance matrix symmetrical and calculate its supplement. Then we calculate the tree and print it.

273f ⟨*Process distance matrix, Ch. 31* 273f⟩≡ (273d)
```
dm.MakeSymmetrical()
⟨Calculate supplementary matrix, Ch. 31 274a⟩
var root *nwk.Node
⟨Calculate tree, Ch. 31 274d⟩
fmt.Println(root)
```

We import `nwk` and `fmt`.

273g ⟨*Imports, Ch. 31* 272c⟩+≡ (272a) ◁273e 275e▷
```
"github.com/evolbioinf/nwk"
"fmt"
```

We calculate the row sums and from them the supplementary matrix by function calls.

274a ⟨*Calculate supplementary matrix, Ch. 31* 274a⟩≡ (273f)
```
r := rowSums(dm)
sm := smat(dm, r)
```

We calculate the row sums.

274b ⟨*Functions, Ch. 31* 273d⟩+≡ (272a) ◁273d 274c▷
```
func rowSums(dm *dist.DistMat) []float64 {
        n := len(dm.Names)
        r := make([]float64, n)
        for i := 0; i < n; i++ {
                for j := 0; j < n; j++ {
                        r[i] += dm.Matrix[i][j]
                }
        }
        return r
}
```

We calculate the supplementary matrix.

274c ⟨*Functions, Ch. 31* 273d⟩+≡ (272a) ◁274b 275d▷
```
func smat(dm *dist.DistMat, r []float64) *dist.DistMat {
        n := len(dm.Names)
        sm := dist.NewDistMat(n)
        for i := 0; i < n-1; i++ {
                for j := i+1; j < n; j++ {
                        sm.Matrix[i][j] = dm.Matrix[i][j] -
                                (r[i] + r[j]) / float64(n - 2)
                        sm.Matrix[j][i] = sm.Matrix[i][j]
                }
        }
        return sm
}
```

Apart from the two distance matrices, we also need the node array as prerequisite for calculating the tree. While we iterate over the steps of the tree computation, we print the current distance matrix, if desired. Then we construct the intermediate tree.

After the loop to construct the tree, we finish its construction.

274d ⟨*Calculate tree, Ch. 31* 274d⟩≡ (273f)
```
⟨Construct node array, Ch. 31 275b⟩
for i := n; i > 3; i-- {
        if printMat {
                ⟨Print matrices, Ch. 31 275c⟩
        }
        ⟨Construct intermediate tree, Ch. 31 275a⟩
}
if printMat {
        ⟨Print matrices, Ch. 31 275c⟩
}
⟨Finish tree, Ch. 31 277b⟩
```

The intermediate tree is constructed by picking a pair of nodes, clustering it, and replacing the clustered nodes by their new parent.

275a ⟨*Construct intermediate tree, Ch. 31* 275a⟩≡ (274d)
```
⟨Pick nodes for clustering, Ch. 31 276a⟩
⟨Cluster nodes, Ch. 31 276b⟩
⟨Replace clustered nodes, Ch. 31 276c⟩
```

The node array starts out with the n leaves.

275b ⟨*Construct node array, Ch. 31* 275b⟩≡ (274d)
```
n := len(dm.Names)
t := make([]*nwk.Node, n)
for i := 0; i < n; i++ {
        t[i] = nwk.NewNode()
        t[i].Label = dm.Names[i]
}
```

We delegate matrix printing to a function call.

275c ⟨*Print matrices, Ch. 31* 275c⟩≡ (274d)
```
printMatrices(dm, sm, r)
```

For each pair of matrices, we print a single matrix in PHYLIP format with the distances in the top triangle and the supplementary distances in the bottom triangle. The last column holds the row sums. Printing is done using a tab writer.

275d ⟨*Functions, Ch. 31* 273d⟩+≡ (272a) ◁274c 278a▷
```
func printMatrices(dm, sm *dist.DistMat, r []float64) {
        w := tabwriter.NewWriter(os.Stdout, 1, 0, 2, ' ', 0)
        n := len(dm.Names)
        fmt.Fprintf(w, "%d\n", n)
        w.Flush()
        for i := 0; i < n; i++ {
                fmt.Fprintf(w, "%s", dm.Names[i])
                ⟨Print row of distances, Ch. 31 275f⟩
                fmt.Fprintf(w, "\t%.3g\n", r[i])
        }
        w.Flush()
}
```

We import `tabwriter` and `os`.

275e ⟨*Imports, Ch. 31* 272c⟩+≡ (272a) ◁273g
```
"text/tabwriter"
"os"
```

Within a row we switch between distances and supplementary distance.

275f ⟨*Print row of distances, Ch. 31* 275f⟩≡ (275d)
```
for j := 0; j < n; j++ {
        x := sm.Matrix[i][j]
        if i < j {
                x = dm.Matrix[i][j]
        }
        fmt.Fprintf(w, "\t%.3g", x)
}
```

We pick the two nodes to be clustered and set their branch lengths.

276a ⟨*Pick nodes for clustering, Ch. 31* 276a⟩≡ (275a)

```
_, mj, mk := sm.Min()
c1 := t[mj]
c2 := t[mk]
root = nwk.NewNode()
l := fmt.Sprintf("(%s,%s)", c1.Label, c2.Label)
root.Label = l
x := float64(i-2) * dm.Matrix[mj][mk]
denom := float64(2*(i-2))
c1.Length = (x + r[mj] - r[mk]) / denom
c2.Length = (x + r[mk] - r[mj]) / denom
c1.HasLength = true
c2.HasLength = true
```

We cluster the nodes by adding them as child nodes to `root`.

276b ⟨*Cluster nodes, Ch. 31* 276b⟩≡ (275a)

```
root.AddChild(c1)
root.AddChild(c2)
```

We replace the nodes just clustered in the distance matrix and in the node array. Then we recalculate the row sums and the supplementary matrix.

276c ⟨*Replace clustered nodes, Ch. 31* 276c⟩≡ (275a)

```
⟨Replace entries in matrix, Ch. 31 276d⟩
⟨Replace entries in node array, Ch. 31 277a⟩
r = rowSums(dm)
sm = smat(dm, r)
```

We calculate the new distances in the original matrix, delete the taxon pair from it, and append the new distances.

276d ⟨*Replace entries in matrix, Ch. 31* 276d⟩≡ (276c)

```
data := make([]float64, i-2)
k := 0
for j := 0; j < i; j++ {
          if j == mj || j == mk { continue }
          data[k] = (dm.Matrix[j][mj] +
                    dm.Matrix[j][mk] - dm.Matrix[mj][mk]) / 2.0
          k++
}
dm.DeletePair(mj, mk)
dm.Append(root.Label, data)
```

We remove the nodes picked and append their parent, the current root.

277a ⟨*Replace entries in node array, Ch. 31* 277a⟩≡ (276c)
```
k = 0
for j := 0; j < i; j++ {
        if j == mj || j == mk { continue }
        t[k] = t[j]
        k++
}
t = t[:k]
t = append(t, root)
```

To cluster the last three nodes, we set their branch lengths and add them to their root.

There are now three taxa left. We connect them to generate the final tree. In that tree, we have labeled the internal nodes to help make sense of the printed matrices. So we remove these labels again, as phylogenies only have leaf labels. Finally, we set negative branch lengths to zero, unless the user allowed negative branch lengths.

277b ⟨*Finish tree, Ch. 31* 277b⟩≡ (274d)
```
⟨Cluster last three nodes, Ch. 31 277c⟩
⟨Reset internal node labels, Ch. 31 277e⟩
if !negBr {
        ⟨Set negative branch lengths to zero, Ch. 31 278b⟩
}
```

277c ⟨*Cluster last three nodes, Ch. 31* 277c⟩≡ (277b)
```
c1 := t[0]
c2 := t[1]
c3 := t[2]
⟨Set branch lengths, Ch. 31 277d⟩
root = nwk.NewNode()
root.AddChild(c1)
root.AddChild(c2)
root.AddChild(c3)
```

We set the branch length as described in the Introduction.

277d ⟨*Set branch lengths, Ch. 31* 277d⟩≡ (277c)
```
c1.Length = (dm.Matrix[0][1] + dm.Matrix[0][2] -
        dm.Matrix[1][2]) / 2.0
c2.Length = (dm.Matrix[1][0] + dm.Matrix[1][2] -
        dm.Matrix[0][2]) / 2.0
c3.Length = (dm.Matrix[2][0] + dm.Matrix[2][1] -
        dm.Matrix[0][1]) / 2.0
c1.HasLength = true
c2.HasLength = true
c3.HasLength = true
```

We reset the internal node labels by a function call.

277e ⟨*Reset internal node labels, Ch. 31* 277e⟩≡ (277b)
```
resetLabels(root)
```

The function `resetLabels` recursively traverses the tree. At each internal node we set the label to the empty string.

278a ⟨*Functions, Ch. 31* 273d⟩+≡ (272a) ◁275d 278c▷

```
func resetLabels(v *nwk.Node) {
        if v == nil { return }
        resetLabels(v.Child)
        resetLabels(v.Sib)
        if v.Child != nil {
                v.Label = ""
        }
}
```

We set the negative branch lengths by calling the function `correctBranchLenghts`.

278b ⟨*Set negative branch lengths to zero, Ch. 31* 278b⟩≡ (277b)

```
correctBranchLengths(root)
```

The function `correctBranchLenghts` recursively visits each node and sets negative branch lengths to zero.

278c ⟨*Functions, Ch. 31* 273d⟩+≡ (272a) ◁278a

```
func correctBranchLengths(v *nwk.Node) {
        if v == nil { return }
        correctBranchLengths(v.Child)
        correctBranchLengths(v.Sib)
        if v.Length < 0 {
                v.Length = 0.0
        }
}
```

We have finished `nj`, time to test it.

Testing

The outline of our testing code has hooks for imports and the testing logic.

278d ⟨*nj_test.go* 278d⟩≡

```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 31 279b⟩
)

func TestNj(t *testing.T) {
        ⟨Testing, Ch. 31 279a⟩
}
```

We test `nj` by running it on the distance matrix in Figure 31.1A, which is contained in the file `test.phy`. We run the test with printing of the intermediate matrices. Then we compare the result we get with the result we want, which is stored in `r.txt`.

279a ⟨*Testing, Ch. 31* 279a⟩≡ (278d)

```
cmd := exec.Command("./nj", "-m", "test.phy")
get, err := cmd.Output()
if err != nil {
          t.Errorf("can't run %q", cmd)
}
want, err := ioutil.ReadFile("r.txt")
if err != nil {
          t.Errorf("can't open r.txt")
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `exec`, `ioutil`, and `bytes`.

279b ⟨*Testing imports, Ch. 31* 279b⟩≡ (278d)

```
"os/exec"
"io/ioutil"
"bytes"
```

Chapter 32

Program `olga`: Compute Overlap Graph

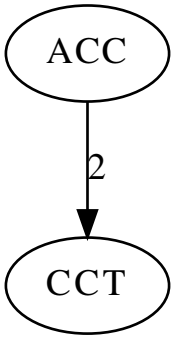


Figure 32.1: Overlap graph for ACC and CCT.

Introduction

An overlap graph shows the suffix/prefix matches between all pairs of strings. For example ACC and CCT have an overlap of CC and their overlap graph would be Figure 32.1. The nodes in the graph are the strings, the edge indicates the existence of an overlap and the edge label the length of the overlap.

The program olga reads a set of strings and prints the corresponding overlap graph in the dot language.

Implementation

The outline of olga has hooks for imports, functions, and the logic of the main function.

281 ⟨*olga.go* 281⟩≡

```
package main

import (
        ⟨Imports, Ch. 32 282b⟩
)
⟨Functions, Ch. 32 283c⟩
func main() {
        ⟨Main function, Ch. 32 282a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, and parse the options and the input files. Then we calculate the overlap graph and draw it.

282a ⟨*Main function, Ch. 32* 282a⟩≡ (281)
```
util.PrepLog("olga")
⟨Set usage, Ch. 32 282c⟩
⟨Declare options, Ch. 32 282e⟩
⟨Parse options, Ch. 32 282g⟩
⟨Parse input files, Ch. 32 283a⟩
⟨Calculate overlap graph, Ch. 32 283e⟩
⟨Print overlap graph, Ch. 32 284e⟩
```

We import `util`.

282b ⟨*Imports, Ch. 32* 282b⟩≡ (281) 282d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `olga`, and an example command.

282c ⟨*Set usage, Ch. 32* 282c⟩≡ (282a)
```
u := "olga [-v|-h] [file]..."
p := "Calculate overlap graph from input strings."
e := "olga foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

282d ⟨*Imports, Ch. 32* 282b⟩+≡ (281) ◁282b 282f▷
```
"github.com/evolbioinf/clio"
```

Apart from the version, we declare only one option, the minimum overlap, which by default is 1.

282e ⟨*Declare options, Ch. 32* 282e⟩≡ (282a)
```
var optV = flag.Bool("v", false, "version")
var optK = flag.Int("k", 1, "overlap")
```

We import `flag`.

282f ⟨*Imports, Ch. 32* 282b⟩+≡ (281) ◁282d 282h▷
```
"flag"
```

We parse the options and respond to `-v` as this stops the program. We also make sure the requested overlap isn't negative.

282g ⟨*Parse options, Ch. 32* 282g⟩≡ (282a)
```
flag.Parse()
if *optV {
        util.PrintInfo("olga")
}
if *optK < 0 {
        log.Fatal("please enter positive minimum overlap")
}
```

We import `log`.

282h ⟨*Imports, Ch. 32* 282b⟩+≡ (281) ◁282f 283b▷
```
"log"
```

The remaining tokens on the command line are taken as names of input files. These files are parsed with the function `scan` and all reads are collected in a slice of sequences.

283a ⟨*Parse input files, Ch. 32* 283a⟩≡ (282a)
```
files := flag.Args()
reads := make([]*fasta.Sequence, 0)
clio.ParseFiles(files, scan, &reads)
```

We import `fasta`.

283b ⟨*Imports, Ch. 32* 282b⟩+≡ (281) ◁282h 283d▷
```
"github.com/evolbioinf/fasta"
```

Inside `scan` we retrieve the pointer to the reads slice and store sequences in that slice.

283c ⟨*Functions, Ch. 32* 283c⟩≡ (281) 284a▷
```
func scan(r io.Reader, args ...interface{}) {
        reads := args[0].(*([]*fasta.Sequence))
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                s := sc.Sequence()
                (*reads) = append(*reads, s)
        }
}
```

We import `io`.

283d ⟨*Imports, Ch. 32* 282b⟩+≡ (281) ◁283b 284d▷
```
"io"
```

To calculate the overlap graph, we construct the graph matrix and fill it in.

283e ⟨*Calculate overlap graph, Ch. 32* 283e⟩≡ (282a)
```
⟨Construct graph matrix, Ch. 32 283f⟩
⟨Fill in graph matrix, Ch. 32 283g⟩
```

The graph matrix for n strings is an $n \times n$ matrix of integers, the overlaps.

283f ⟨*Construct graph matrix, Ch. 32* 283f⟩≡ (283e)
```
n := len(reads)
graph := make([][]int, n)
for i := 0; i < n; i++ {
        graph[i] = make([]int, n)
}
```

For each pair of stings we calculate and store the overlap.

283g ⟨*Fill in graph matrix, Ch. 32* 283g⟩≡ (283e)
```
for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
                if i == j { continue }
                a := reads[i].Data()
                b := reads[j].Data()
                graph[i][j] = overlap(a, b, *optK)
        }
}
```

The function `overlap` takes two byte slices, a and b, and a minimum overlap, k, as input and returns their overlap. Initially, it checks that k is sensible, then it finds the overlap.

284a ⟨*Functions, Ch. 32* 283c⟩+≡ (281) ◁283c
```
func overlap(a, b []byte, k int) int {
        p := 0
        ⟨Check k, Ch. 32 284b⟩
        ⟨Find overlap, Ch. 32 284c⟩
        return p
}
```

If k is greater than the length of `b`, we bail with message.

284b ⟨*Check k, Ch. 32* 284b⟩≡ (284a)
```
if k > len(b) {
        log.Fatal("can't have longer overlaps than reads")
}
```

We calculate the overlap in an infinite loop. Inside the loop we search for the first occurrence of $b[1...k]$ in $a[s...]$. Then we try to extend this match to the end of a. If that fails, we increment s and try again.

284c ⟨*Find overlap, Ch. 32* 284c⟩≡ (284a)
```
for true {
        s := bytes.Index(a[p:], b[0:k])
        if s == -1 { return 0 }
        if bytes.HasPrefix(b, a[p+s:]) {
                return len(a) - s - p
        }
        p += s+1
}
```

We import `bytes`.

284d ⟨*Imports, Ch. 32* 282b⟩+≡ (281) ◁283d 284g▷
```
"bytes"
```

For the actual graph we first print a header, then the graph body, and finally the footer.

284e ⟨*Print overlap graph, Ch. 32* 284e⟩≡ (282a)
⟨*Print graph header, Ch. 32* 284f⟩
⟨*Print graph body, Ch. 32* 285a⟩
⟨*Print graph footer, Ch. 32* 285d⟩

We start the graph with comments as to its origin and how it is rendered. Then we open the actual graph, which is a directed graph.

284f ⟨*Print graph header, Ch. 32* 284f⟩≡ (284e)
```
fmt.Println("# Overlap graph generated with olga.")
fmt.Println("# Render: dot foo.dot")
fmt.Println("digraph G {")
```

We import `fmt`.

284g ⟨*Imports, Ch. 32* 282b⟩+≡ (281) ◁284d
```
"fmt"
```

The graph body consists of nodes and edges.

285a ⟨*Print graph body, Ch. 32* 285a⟩≡ (284e)
```
⟨Print nodes, Ch. 32 285b⟩
⟨Print edges, Ch. 32 285c⟩
```

The reads are the nodes. Since there might be repeated strings in the input, we distinguish between the ID and the label of a node.

285b ⟨*Print nodes, Ch. 32* 285b⟩≡ (285a)
```
for i := 0; i < n; i++ {
        fmt.Printf("\tn%d [label=\"%s\"]\n", i,
                string(reads[i].Data()))
}
```

We print all non-zero edges labeled by the overlap. The edge label starts with a blank so that it doesn't touch the edge.

285c ⟨*Print edges, Ch. 32* 285c⟩≡ (285a)
```
for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
                if graph[i][j] == 0 { continue }
                fmt.Printf("\tn%d -> n%d [label=\" %d\"]\n",
                        i, j, graph[i][j])
        }
}
```

The footer is just the closing curly bracket.

285d ⟨*Print graph footer, Ch. 32* 285d⟩≡ (284e)
```
fmt.Println("}")
```

We've finished writing `olga`, time to test it.

Testing

Out testing code has hooks for imports and the testing logic.

285e ⟨*olga_test.go* 285e⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 32 286b⟩
)

func TestOlga(t *testing.T) {
        ⟨Testing, Ch. 32 286a⟩
}
```

We construct a set of tests and iterate over them.

286a ⟨*Testing, Ch. 32* 286a⟩≡ (285e)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 32 286c⟩
for i, test := range tests {
        ⟨Run test, Ch. 32 286d⟩
}
```

We import `exec`.

286b ⟨*Testing imports, Ch. 32* 286b⟩≡ (285e) 286e ▷
```
"os/exec"
```

We construct two tests, one with default parameters, the other with $k = 3$.

286c ⟨*Construct tests, Ch. 32* 286c⟩≡ (286a)
```
f := "reads.fasta"
test := exec.Command("./olga", f)
tests = append(tests, test)
test = exec.Command("./olga", "-k", "3", f)
tests = append(tests, test)
```

We store the result we get from a test and compare it to the result we want, which is stored in files `r1.txt` and `r2.txt`.

286d ⟨*Run test, Ch. 32* 286d⟩≡ (286a)
```
get, err := test.Output()
if err != nil {
        t.Errorf("can't run %q", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
        t.Errorf("can't open %q", f)
}
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

286e ⟨*Testing imports, Ch. 32* 286b⟩+≡ (285e) ◁286b
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 33

Program `pam`: Compute PAM Score Matrices

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	3	-3	0	0	-3	-1	0	1	-3	-1	-3	-2	-2	-4	1	1	1	-7	-4	0
R	-3	6	-1	-3	-4	1	-3	-4	1	-2	-4	2	-1	-4	-1	-1	-2	1	-6	-3
N	0	-1	4	2	-5	0	1	0	2	-2	-4	1	-3	-4	-2	1	0	-5	-2	-3
D	0	-3	2	5	-7	1	3	0	0	-3	-5	-1	-4	-7	-2	0	-1	-8	-5	-3
C	-3	-4	-5	-7	9	-7	-7	-5	-4	-3	-7	-7	-5	-6	-4	-1	-3	-8	-1	-2
Q	-1	1	0	1	-7	6	2	-3	3	-3	-2	0	-1	-6	0	-2	-2	-6	-5	-3
E	0	-3	1	3	-7	2	5	-1	-1	-3	-4	-1	-3	-6	-1	-1	-2	-8	-4	-3
G	1	-4	0	0	-5	-3	-1	5	-4	-4	-5	-3	-4	-5	-2	1	-1	-8	-6	-2
H	-3	1	2	0	-4	3	-1	-4	7	-4	-3	-2	-4	-2	-1	-2	-3	-4	-1	-3
I	-1	-2	-2	-3	-3	-3	-3	-4	-4	6	1	-2	1	0	-3	-2	0	-7	-2	3
L	-3	-4	-4	-5	-7	-2	-4	-5	-3	1	5	-4	3	0	-3	-4	-3	-4	-3	1
K	-2	2	1	-1	-7	0	-1	-3	-2	-2	-4	5	0	-6	-2	-1	-1	-5	-6	-4
M	-2	-1	-3	-4	-5	-1	-3	-4	-4	1	3	0	8	-1	-3	-2	-1	-6	-4	1
F	-4	-4	-4	-7	-6	-6	-6	-5	-2	0	0	-6	-1	8	-5	-3	-4	-1	4	-3
P	1	-1	-2	-2	-4	0	-1	-2	-1	-3	-3	-2	-3	-5	6	1	-1	-7	-6	-2
S	1	-1	1	0	-1	-2	-1	1	-2	-2	-4	-1	-2	-3	1	3	2	-2	-3	-2
T	1	-2	0	-1	-3	-2	-2	-1	-3	0	-3	-1	-1	-4	-1	2	4	-6	-3	0
W	-7	1	-5	-8	-8	-6	-8	-8	-4	-7	-4	-5	-6	-1	-7	-2	-6	12	-1	-8
Y	-4	-6	-2	-5	-1	-5	-4	-6	-1	-2	-3	-6	-4	4	-6	-3	-3	-1	8	-3
V	0	-3	-3	-3	-2	-3	-3	-2	-3	3	1	-4	1	-3	-2	-2	0	-8	-3	5

Figure 33.1: PAM120 amino acid substitution matrix.

Introduction

The 20 amino acids that make up proteins have diverse chemical properties. Some are large, like the two-ring system of tryptophane, others are small, like the hydrogen atom of glycine. Aspartate and glutamate are acidic and water soluble, leucine and isoleucine are aliphatic and water insoluble. So changes between amino acids have widely differing effects on protein structure, depending on the distance in chemical space traversed. Moreover, amino acids are encoded by triplet codons. The distance between codons in sequence space thus ranges from one to three. As a result, pairs of amino acids are scored individually rather than as binary matches/mismatches.

The match scores in Figure 33.1 along the main diagonal range from 3 for alanine (`A`) and serine (`S`) to 12 for tryptophane (`W`). The mismatch scores range from -8 (`W`, `[CDEGV]`) to 4 (`F`, `Y`).

Apart from their chemical heterogeneity, there is another complication when scoring pairs amino acids. The probability of change itself changes over time. Initially, just after divergence, it is very low, but grows as time passes. This time-dependence of mutation probabilities is also true for nucleotides, where it is usually ignored. Not so with amino acids, for which whole series of substitution matrices have been devised, covering closely related to highly divergent sequences. The PAM series is an early one that is still used today. PAM stands for Percent Accepted Mutations, Figure 33.1 shows an example from the series.

PAM matrices are computed from the mutation probabilities for each pair of amino acids found in proteins separated by an evolutionary distance of 1 PAM. Figure 33.2 shows this probability matrix with entries multiplied by 10000. This is also the format later used in the computations. An entry, $m_{i,j}$, is the probability of amino acid j changing into amino acid i. These probabilities were originally obtained from multiple

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	9867	2	9	10	3	8	17	21	2	6	4	2	6	2	22	35	32	0	2	18
R	1	9913	1	0	1	10	0	0	10	3	1	19	4	1	4	6	1	8	0	1
N	4	1	9822	36	0	4	6	6	21	3	1	13	0	1	2	20	9	1	4	1
D	6	0	42	9859	0	6	53	6	4	1	0	3	0	0	1	5	3	0	0	1
C	1	1	0	0	9973	0	0	0	1	1	0	0	1	0	0	5	1	0	3	2
Q	3	9	4	5	0	9876	27	1	23	1	3	6	4	0	6	2	2	0	0	1
E	10	0	7	56	0	35	9865	4	2	3	1	4	1	0	3	4	2	0	1	2
G	21	1	12	11	1	3	7	9935	1	0	1	2	1	1	3	21	3	0	0	5
H	1	8	18	3	1	20	1	0	9912	0	1	1	0	2	3	1	1	1	4	1
I	2	2	3	1	2	1	2	0	0	9872	9	2	12	7	0	1	7	0	1	33
L	3	1	3	0	0	6	1	1	4	22	9947	2	45	13	3	1	3	4	2	15
K	2	37	25	6	0	12	7	2	2	4	1	9926	20	0	3	8	11	0	1	1
M	1	1	0	0	0	2	0	0	0	5	8	4	9874	1	0	1	2	0	0	4
F	1	1	1	0	0	0	0	1	2	8	6	0	4	9946	0	2	1	3	28	0
P	13	5	2	1	1	8	3	2	5	1	2	2	1	1	9926	12	4	0	0	2
S	28	11	34	7	11	4	6	16	2	2	1	7	4	3	17	9840	38	5	2	2
T	22	2	13	4	1	3	2	2	1	11	2	8	6	1	5	32	9871	0	2	9
W	0	2	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	9976	1	0
Y	1	0	3	0	3	0	1	0	4	1	1	0	0	21	0	1	1	2	9945	1
V	13	2	1	1	3	2	2	3	3	57	11	1	17	1	3	2	10	0	2	9901

Figure 33.2: Amino acid mutation probabilities for sequences separated by an evolutionary distance of 1 PAM. The probabilities are multiplied by 10,000

sequence alignments and are thus not symmetrical. For instance, the probability of valine changing into isoleucine, `V` $\rightarrow$ `I`, is 0.33%, while the reverse probability is 0.57%.

Let the product of two $n \times n$ matrices $A = (a_{i,j})$, $B = (b_{i,j})$ be $C = c_{i,j}$, where $c_{i,j}$ is the sum of the elements in row a_i multiplied by the elements in column b_j,

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}. \tag{33.1}$$

The great thing about this mechanism is that multiplication of the probability matrix in Figure 33.2 with itself generates the probabilities for an evolutionary distance of 2 PAM. This can be repeated as often as we like, to give the probabilities for any evolutionary distance measured in PAM. In this case, matrix multiplication is a simple way to simulate evolution.

Clearly, the number of PAMs elapsed is not the same as the resulting percent difference between two homologous protein sequences. This percent difference can be calculated from the following consideration: The main diagonal of a mutation matrix like Figure 33.2 gives us the probability, $m_{i,i}$, that an amino acid has not changed. The probability of finding that particular amino acid is its background frequency, f_i. So the expected %-difference between protein sequences is

$$d = \left(1 - \sum_{i=1}^{20} m_{i,i} f_i\right) \times 100. \tag{33.2}$$

Amino acid frequencies are supplied in a file with 20 rows of pairs of amino acids and frequencies:

```
A 0.087
C 0.033
D 0.047
E 0.050
...
```

The order of rows doesn't matter.

The mutation probabilities $m_{i,j}$ are normalized by division by the frequency of the amino acid mutated into, f_i. Another way to look at f_i is as the probability of randomly drawing amino acid i. The result of this division is a relatedness measure,

$$r_{i,j} = \frac{m_{i,j}}{f_i}.$$

The relatedness matrix is symmetrical.

In a third and final step, the relatedness values are log-transformed and rounded to the nearest integer. So if $m_{i,j} = f_i$, the score is zero. That is, if the probability of mutating amino acid j to amino acid i is the same as randomly picking a partner for j, the score is 0. Similarly, if $m_{i,j} < f_i$, the score is negative like most off-diagonal entries in Figure 33.1. If $m_{i,j} > f_i$, the score is positive, like all on-diagonal and some off-diagonal entries in Figure 33.1.

The program `pam` implements the three steps for computing an amino acid substitution matrix: matrix multiplication, normalization, and log-transformation.

Implementation

The outline of `pam` contains hooks for imports, functions, and the logic of the main function.

290a ⟨*pam.go* 290a⟩≡
```
package main

import (
        ⟨Imports, Ch. 33 290c⟩
)
⟨Functions, Ch. 33 292d⟩
func main() {
        ⟨Main function, Ch. 33 290b⟩
}
```

In the main function, we prepare the `log` package, state the usage, declare and parse the options, and parse the input file.

290b ⟨*Main function, Ch. 33* 290b⟩≡ (290a)
```
util.PrepLog("pam")
⟨Set usage, Ch. 33 290d⟩
⟨Declare options, Ch. 33 291b⟩
⟨Parse options, Ch. 33 291d⟩
⟨Parse input file, Ch. 33 292c⟩
```

We import `util`.

290c ⟨*Imports, Ch. 33* 290c⟩≡ (290a) 291a ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts: The usage message, an explanation of the program's purpose, and an example command.

290d ⟨*Set usage, Ch. 33* 290d⟩≡ (290b)
```
u := "pam [-h] [options] [files]"
p := "Compute PAM matrices."
e := "pam -n 120 pam1.txt | pam -a aa.txt | pam"
clio.Usage(u, p, e)
```

We import `clio`.

291a ⟨*Imports, Ch. 33* 290c⟩+≡ (290a) ◁290c 291c▷
```
"github.com/evolbioinf/clio"
```

There are four options:

1. `-n` n: Compute $(m_{i,j})^n$
2. `-a aa.txt`: Normalize with the amino acid frequencies contained in `aa.txt`
3. `-b` f: Log-odds ratios given in f bits.
4. `-v`: Print version

291b ⟨*Declare options, Ch. 33* 291b⟩≡ (290b)
```
var optN = flag.Int("n", 0, "compute matrix^n; " +
          "default: log-transformation")
var optA = flag.String("a", "", "normalize by frequencies " +
          "in file; default: log-transformation")
var optB = flag.Float64("b", 0.5, "bits")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

291c ⟨*Imports, Ch. 33* 290c⟩+≡ (290a) ◁291a 291f▷
```
"flag"
```

The options are parsed and we respond to `-v` by printing the version, and to `-a` by opening the file and reading the amino acid frequencies.

291d ⟨*Parse options, Ch. 33* 291d⟩≡ (290b)
```
flag.Parse()
if *optV {
          util.PrintInfo("pam")
}
frequencies := make(map[byte]float64)
if *optA != "" {
          ⟨Open frequencies file, Ch. 33 291e⟩
          ⟨Read frequencies, Ch. 33 292a⟩
}
```

If the frequencies file cannot be opened, we abort.

291e ⟨*Open frequencies file, Ch. 33* 291e⟩≡ (291d)
```
f, err := os.Open(*optA)
if err != nil {
          log.Fatalf("couldn't open %q\n", *optA)
}
```

We import `os` and `log`.

291f ⟨*Imports, Ch. 33* 290c⟩+≡ (290a) ◁291c 292b▷
```
"os"
"log"
```

The frequencies are read with a scanner that splits the line into strings at word boundaries. The first character of the first string contains the amino acid, the second string the frequency. Lines starting with a hash are ignored.

292a ⟨*Read frequencies, Ch. 33* 292a⟩≡ (291d)
```
sc := bufio.NewScanner(f)
for sc.Scan() {
        line := sc.Text()
        str := strings.Fields(line)
        a := str[0][0]
        if a == '#' { continue }
        x, err := strconv.ParseFloat(str[1], 64)
        if err != nil {
                log.Fatalf("couldn't parse %q\n", str[1])
        }
        frequencies[a] = x
}
```

We import `bufio`, `strings`, and `strconv`.

292b ⟨*Imports, Ch. 33* 290c⟩+≡ (290a) ◁291f 292e▷
```
"bufio"
"strings"
"strconv"
```

The remaining argument is the input file. This is parsed using the function `ParseFiles`, which takes as first argument a list of file names. In our case this list contains at most one entry. The input is scanned with the function `scan`, which takes as arguments the exponent and the frequencies.

292c ⟨*Parse input file, Ch. 33* 292c⟩≡ (290b)
```
f := flag.Args()
if len(f) > 1 {
        f = f[:1]
}
clio.ParseFiles(f, scan, *optN, frequencies, *optB)
```

In the function `scan`, we retrieve the arguments, read the matrix, transform, and print the output.

292d ⟨*Functions, Ch. 33* 292d⟩≡ (290a)
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 33 293a⟩
        ⟨Read matrix, Ch. 33 293b⟩
        ⟨Transform matrix, Ch. 33 293d⟩
        ⟨Print output, Ch. 33 295b⟩
}
```

We import `io`.

292e ⟨*Imports, Ch. 33* 290c⟩+≡ (290a) ◁292b 293c▷
```
"io"
```

The exponent, frequencies, and bits just passed are retrieved using type assertions.

293a ⟨*Retrieve arguments, Ch. 33* 293a⟩≡ (292d)
```
exp := args[0].(int)
freq := args[1].(map[byte]float64)
bits := args[2].(float64)
```

We read the matrix using a dedicated function and extract the entries.

293b ⟨*Read matrix, Ch. 33* 293b⟩≡ (292d)
```
aa := "ARNDCQEGHILKMFPSTWYV"
sm := util.ReadScoreMatrix(r)
m := len(aa)
ma := make([][]float64, m)
for i := 0; i < m; i++ {
        ma[i] = make([]float64, m)
}
for i := 0; i < m; i++ {
        for j := 0; j < m; j++ {
                ma[i][j] = sm.Score(aa[i], aa[j])
        }
}
```

We import `util`.

293c ⟨*Imports, Ch. 33* 290c⟩+≡ (290a) ◁292e 295a▷
```
"github.com/evolbioinf/biobox/util"
```

The matrix is transformed depending on whether an exponent was set or frequencies were passed. If an exponent *and* frequencies were set, we do the sensible thing and carry out the matrix multiplication before the normalization.

293d ⟨*Transform matrix, Ch. 33* 293d⟩≡ (292d)
```
if exp > 0 {
        ⟨Multiply matrix, Ch. 33 294a⟩
}
if len(freq) > 0 {
        ⟨Normalize matrix, Ch. 33 294d⟩
}
if exp == 0 && len(freq) == 0 {
        ⟨Log-transform matrix, Ch. 33 294e⟩
}
```

To multiply a matrix, A, repeatedly with itself, it is first copied to obtain B, and the we repeat

$$A \leftarrow A \times B.$$

So we first copy the matrix and then carry out the multiplication.

294a ⟨*Multiply matrix, Ch. 33* 294a⟩≡ (293d)
```
⟨Copy matrix, Ch. 33 294b⟩
for i := 1; i < exp; i++ {
        for j := 0; j < m; j++ {
                for k := 0; k < m; k++ {
                        ⟨Compute matrix entry, Ch. 33 294c⟩
                }
        }
}
```

To copy the matrix, we construct a new one and use the built-in function `copy`.

294b ⟨*Copy matrix, Ch. 33* 294b⟩≡ (294a)
```
mo := make([][]float64, m)
for i := 0; i < m; i++ {
          mo[i] = make([]float64, m)
          copy(mo[i], ma[i])
}
```

The matrix entry is computed using equation (33.1).

294c ⟨*Compute matrix entry, Ch. 33* 294c⟩≡ (294a)
```
s := 0.0
for l := 0; l < m; l++ {
          s += ma[j][l] * mo[l][k]
}
ma[j][k] = s
```

The matrix is normalized by dividing by the frequency of the amino acid mutated into.

294d ⟨*Normalize matrix, Ch. 33* 294d⟩≡ (293d)
```
for i := 0; i < m; i++ {
          for j := 0; j < m; j++ {
                  ma[i][j] /= freq[aa[i]]
          }
}
```

The scores are odds measured in bits, hence the log-transformation is to the basis of 2. The result is rounded to the nearest integer.

294e ⟨*Log-transform matrix, Ch. 33* 294e⟩≡ (293d)
```
for i := 0; i < m; i++ {
          for j := 0; j < m; j++ {
                  ma[i][j] = math.Log2(ma[i][j]) / bits
                  ma[i][j] = math.Round(ma[i][j])
          }
}
```

We import `math`.

295a ⟨*Imports, Ch. 33* 290c⟩+≡ (290a) ◁293c 295f▷
```
"math"
```

Having computed the new matrix, we print the output. This consists of two components, the %-difference according to equation (33.2) and the matrix. However, the %-difference is a function of the amino acid frequencies, so we only print it if we have them.

295b ⟨*Print output, Ch. 33* 295b⟩≡ (292d)
```
if len(freq) > 0 {
        ⟨Print percent difference, Ch. 33 295c⟩
}
⟨Print matrix, Ch. 33 295d⟩
```

We compute the %-difference from equation 33.2. It is hashed to hide it from subsequent analyses.

295c ⟨*Print percent difference, Ch. 33* 295c⟩≡ (295b)
```
sum := 0.0
for i := 0; i < m; i++ {
        f := freq[aa[i]]
        sum += ma[i][i] * f * f
}
pd := (1.0 - sum) * 100.0
fmt.Printf("# percent_diff: %.2f\n", pd)
```

To line up the columns, we use a `tabwriter`. Once constructed, we fill it with the header and body of the table.

295d ⟨*Print matrix, Ch. 33* 295d⟩≡ (295b)
```
⟨Construct tabwriter, Ch. 33 295e⟩
⟨Print table header, Ch. 33 295g⟩
⟨Print table body, Ch. 33 296b⟩
```

The `tabwriter` is used to write right-aligned columns at least four positions wide with a single blank as padding.

295e ⟨*Construct tabwriter, Ch. 33* 295e⟩≡ (295d)
```
var buf []byte
buffer := bytes.NewBuffer(buf)
w := new(tabwriter.Writer)
w.Init(buffer, 1, 0, 1, ' ', tabwriter.AlignRight)
```

We import `tabwriter` and `bytes`.

295f ⟨*Imports, Ch. 33* 290c⟩+≡ (290a) ◁295a 296a▷
```
"text/tabwriter"
"bytes"
```

The header consists of the amino acids.

295g ⟨*Print table header, Ch. 33* 295g⟩≡ (295d)
```
fmt.Fprintf(w, "\t")
for _, a := range aa {
        fmt.Fprintf(w, "  %c\t", a)
}
fmt.Fprintf(w, "\n")
```

We import `fmt`.

296a ⟨*Imports, Ch. 33* 290c⟩+≡ (290a) ◁295f
```
"fmt"
```

Each row of the table is labeled by an amino acid, which is followed by the table entries. When all the data are entered, the `tabwriter` is flushed to the buffer, which is printed.

296b ⟨*Print table body, Ch. 33* 296b⟩≡ (295d)
```
for i := 0; i < m; i++ {
        fmt.Fprintf(w, "%c\t", aa[i])
        for j := 0; j < m; j++ {
                ⟨Print table entry, Ch. 33 296c⟩
        }
        fmt.Fprintf(w, "\n")
}
w.Flush()
fmt.Printf("%s", buffer)
```

An entry is either a fraction or an integer.

296c ⟨*Print table entry, Ch. 33* 296c⟩≡ (296b)
```
if exp > 0 || len(freq) > 0 {
        fmt.Fprintf(w, "%.4f\t", ma[i][j])
} else {
        if ma[i][j] == 0.0 {
                fmt.Fprintf(w, "%v\t", 0.0)
        } else {
                fmt.Fprintf(w, "%v\t", ma[i][j])
        }
}
```

The program is finished, let's test.

Testing

The outline for the testing program provides hooks for imports and the testing logic.

296d ⟨*pam_test.go* 296d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 33 297c⟩
)

func TestPam(t *testing.T) {
        ⟨Testing, Ch. 33 297a⟩
}
```

We construct a series of commands and run them. Then we compare what we get with what we want, which is stored in a corresponding set of pre-computed results files.

297a ⟨*Testing, Ch. 33* 297a⟩≡ (296d)
```
commands := make([]*exec.Cmd, 0)
⟨Construct commands, Ch. 33 297b⟩
⟨Construct list of result files, Ch. 33 297d⟩
for i, cmd := range commands {
        ⟨Run test, Ch. 33 297f⟩
}
```

We test each of the three modes of the program, matrix multiplication, normalization, and log transformation. The latter is repeated with a different bit-value.

297b ⟨*Construct commands, Ch. 33* 297b⟩≡ (297a)
```
c := exec.Command("./pam", "-n", "170", "pam1.txt")
commands = append(commands, c)
c = exec.Command("./pam", "-a", "aa.txt", "p170.txt")
commands = append(commands, c)
c = exec.Command("./pam", "p170n.txt")
commands = append(commands, c)
c = exec.Command("./pam", "-b", "0.3333", "p170n.txt")
commands = append(commands, c)
```

We import `exec`.

297c ⟨*Testing imports, Ch. 33* 297c⟩≡ (296d) 297e ▷
```
"os/exec"
```

The results wanted are contained in as many files as we just defined commands.

297d ⟨*Construct list of result files, Ch. 33* 297d⟩≡ (297a)
```
results := make([]string, len(commands))
for i, _ := range commands {
        results[i] = "r" + strconv.Itoa(i+1) + ".txt"
}
```

We import `strconv`.

297e ⟨*Testing imports, Ch. 33* 297c⟩+≡ (296d) ◁297c 298 ▷
```
"strconv"
```

When a command is run, we compare what we get to what we want.

297f ⟨*Run test, Ch. 33* 297f⟩≡ (297a)
```
get, err := cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
want, err := ioutil.ReadFile(results[i])
if err != nil {
        t.Errorf("couldn't open %q\n", results[i])
}
if !bytes.Equal(want, get) {
        t.Errorf("%s\nwant:\n%s\nget:\n%s\n", cmd, want, get)
}
```

We import `ioutil` and `bytes`.

298 ⟨*Testing imports, Ch. 33* 297c⟩+≡ (296d) ◁297e

```
"io/ioutil"
"bytes"
```

Chapter 34

Program `plotLine`: Plotting Lines

A

1	1	g1
2	2	g1
4	4	g1
1	2	g2
2	4	g2
4	8	g2

B

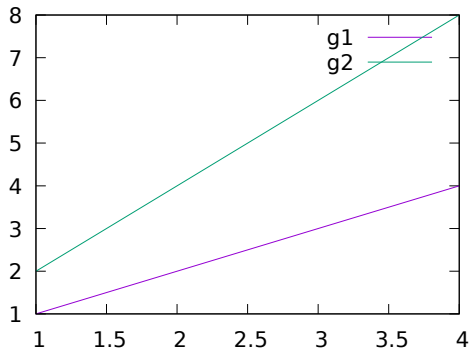


Figure 34.1: Example data (**A**) plotted with `plotLine` (**B**).

Introduction

The program `plotLine` plots lines using `gnuplot` [19]. It takes as input either two or three columns of data. The first two columns are the x- and y-coordinates, the optional third column is the group. Figure 34.1A shows some example data for two groups, g1 and g2, and Figure 34.1B its plot.

Implementation

The outline of `plotLine` has hooks for imports, types, functions, and the logic of the main function.

300 ⟨*plotLine.go* 300⟩≡

```
package main

import (
        ⟨Imports, Ch. 34 301b⟩
)
⟨Types, Ch. 34 302f⟩
⟨Functions, Ch. 34 304b⟩
func main() {
        ⟨Main function, Ch. 34 301a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, and parse the options. Then we set the type of the output window and parse the input files.

301a ⟨*Main function, Ch. 34* 301a⟩≡ (300)
```
util.PrepLog("plotLine")
⟨Set usage, Ch. 34 301c⟩
⟨Declare options, Ch. 34 301e⟩
⟨Parse options, Ch. 34 302d⟩
⟨Set type of output window, Ch. 34 303d⟩
⟨Parse input files, Ch. 34 304a⟩
```

We import `util`.

301b ⟨*Imports, Ch. 34* 301b⟩≡ (300) 301d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `plotLine`, and an example command.

301c ⟨*Set usage, Ch. 34* 301c⟩≡ (301a)
```
u := "plotLine [-h] [option]... [file]..."
p := "Plot lines from columns of x/y data " +
          "and an optional group column."
e := "plotLine foo.dat"
clio.Usage(u, p, e)
```

We import `clio`.

301d ⟨*Imports, Ch. 34* 301b⟩+≡ (300) ◁301b 301f▷
```
"github.com/evolbioinf/clio"
```

Apart from the version, we declare options concerning the axes, the plot type, and the graphics device. We also declare a catch-all "general option".

301e ⟨*Declare options, Ch. 34* 301e⟩≡ (301a)
```
optV := flag.Bool("v", false, "version")
⟨Declare axes options, Ch 34 301g⟩
⟨Declare plot type options, Ch. 34 302a⟩
⟨Declare device options, Ch. 34 302b⟩
⟨Declare general option, Ch. 34 302c⟩
```

We import `flag`.

301f ⟨*Imports, Ch. 34* 301b⟩+≡ (300) ◁301d 303f▷
```
"flag"
```

The options for axes define their labels, ranges, scales, and whether or not one or both of them are unset.

301g ⟨*Declare axes options, Ch 34* 301g⟩≡ (301e)
```
optX := flag.String("x", "", "x-label")
optY := flag.String("y", "", "y-label")
optXX := flag.String("X", "*:*", "x-range")
optYY := flag.String("Y", "*:*", "y-range")
optL := flag.String("l", "", "log-scale (x|y|xy)")
optU := flag.String("u", "", "unset axis (x|y|xy)")
```

By default, the plot consists of lines. But the user can opt for lines and points, or lines only. She can also opt to write the `gnuplot` script to file.

302a ⟨*Declare plot type options, Ch. 34* 302a⟩≡ (301e)
```
optPP := flag.Bool("P", false, "points only")
optLL := flag.Bool("L", false, "lines and points")
optS := flag.String("s", "", "write gnuplot script to file")
```

The default output destination is the screen and the user can set the terminal. Alternatively, the user can write the output to a file of encapsulated postscript. We provide three default plot dimensions, 640×384 pixels for screens, 5×3.5 in for postscript, and 79×24 characters for the "dumb" terminal. Users can also set the dimensions themselves.

302b ⟨*Declare device options, Ch. 34* 302b⟩≡ (301e)
```
optT := flag.String("t", "",
          "terminal (default wxt, qt on darwin)")
optP := flag.String("p", "", "encapsulated postscript file")
defScrDim := "640,384"
defPsDim := "5,3.5"
defDumbDim := "79,24"
optD := flag.String("d", defScrDim, "plot dimensions; " +
          "pixels for screen, " + defPsDim + " in for ps, " +
          defDumbDim + " char for dumb")
```

At its core, `gnuplot` is a graphing programming language. To make the most of the resulting versatility, power users can enter arbitrary `gnuplot` code.

302c ⟨*Declare general option, Ch. 34* 302c⟩≡ (301e)
```
optG := flag.String("g", "", "gnuplot code")
```

We parse the options and respond to `-v` first, as this might terminate `plotLine`. Then we collect the option values.

302d ⟨*Parse options, Ch. 34* 302d⟩≡ (301a)
```
flag.Parse()
⟨Respond to -v, Ch. 34 302e⟩
args := new(Args)
⟨Collect option values, Ch. 34 303a⟩
```

We respond to `-v` by printing information about `plotLine`.

302e ⟨*Respond to -v, Ch. 34* 302e⟩≡ (302d)
```
if *optV {
          util.PrintInfo("plotLine")
}
```

We declare the type `Args` and specify its fields as we go along.

302f ⟨*Types, Ch. 34* 302f⟩≡ (300)
```
type Args struct {
          ⟨Args fields, Ch. 34 303b⟩
}
```

We collect the option values for, `-x`, `-y`, `-X`, `-Y`, `-u`, `-d`, `-P`, `-L`, `-l`, `-s`, `-p`, and `-g`.

303a ⟨*Collect option values, Ch. 34* 303a⟩≡ (302d) 303c ▷
```
args.Xlab = *optX
args.Ylab = *optY
args.Xrange = *optXX
args.Yrange = *optYY
args.Unset = *optU
args.Dim = *optD
args.Points = *optPP
args.LinesPoints = *optLL
args.Log = *optL
args.Script = *optS
args.Ps = *optP
args.Gp = *optG
```

We add the corresponding fields to the structure `Args`.

303b ⟨*`Args` fields, Ch. 34* 303b⟩≡ (302f) 303e ▷
```
Xlab, Ylab, Xrange, Yrange, Unset, Dim string
Points, LinesPoints bool
Log, Script, Ps, Gp string
```

If the user requested postscript output and didn't set a size, we set the default postscript size. Similarly, if the user requested the dumb terminal and didn't set a size, we set the default dumb terminal size.

303c ⟨*Collect option values, Ch. 34* 303a⟩+≡ (302d) ◁303a
```
args.Win = *optT
if args.Dim == defScrDim {
        if args.Ps != "" {
                args.Dim = defPsDim
        } else if args.Win == "dumb" {
                args.Dim = defDumbDim
        }
}
```

The default output window is wxt. If we're running on the macOS command line, we switch to qt.

303d ⟨*Set type of output window, Ch. 34* 303d⟩≡ (301a)
```
if args.Win == "" {
        args.Win = "wxt"
        if runtime.GOOS == "darwin" {
                args.Win = "qt"
        }
}
```

We add the `Win` field to `Args`.

303e ⟨*`Args` fields, Ch. 34* 303b⟩+≡ (302f) ◁303b
```
Win string
```

We import `runtime`.

303f ⟨*Imports, Ch. 34* 301b⟩+≡ (300) ◁301f 304c ▷
```
"runtime"
```

The remaining tokens on the command line are taken as input files. These are parsed with the function `ParseFiles`, which applies the function `scan` to each file. `scan`, in turn, takes as argument the variable `args` we just filled.

304a ⟨*Parse input files, Ch. 34* 304a⟩≡ (301a)
```
files := flag.Args()
clio.ParseFiles(files, scan, args)
```

Inside `scan`, we retrieve the variable `args` by type assertion, read the data, and plot it.

304b ⟨*Functions, Ch. 34* 304b⟩≡ (300)
```
func scan(r io.Reader, a ...interface{}) {
        args := a[0].(*Args)
        ⟨Read data, Ch. 34 304d⟩
        ⟨Extract categories, Ch. 34 305b⟩
        ⟨Plot data, Ch. 34 305c⟩
}
```

We import `io`.

304c ⟨*Imports, Ch. 34* 301b⟩+≡ (300) ◁303f 304e▷
```
"io"
```

While reading the data, we skip comments. For each data point we are either given a category, or we add an empty category. Having read the data, we check the number of columns.

304d ⟨*Read data, Ch. 34* 304d⟩≡ (304b)
```
var data [][]string
sc := bufio.NewScanner(r)
for sc.Scan() {
        if sc.Text()[0] == '#' { continue }
        f := strings.Fields(sc.Text())
        if len(f) == 2 { f = append(f, "") }
        data = append(data, f)
}
⟨Check number of columns, Ch. 34 304f⟩
```

We import `bufio` and `strings`.

304e ⟨*Imports, Ch. 34* 301b⟩+≡ (300) ◁304c 305a▷
```
"bufio"
"strings"
```

If the data doesn't consist of either two or three columns, there's bound to be a fundamental problem, so we bail with a friendly message.

304f ⟨*Check number of columns, Ch. 34* 304f⟩≡ (304d)
```
ncol := 0
if len(data) > 0 {
        ncol = len(data[0])
}
if ncol < 2 || ncol > 3 {
        m := "there should be 2 or 3 columns " +
                "in the input, but you have %d\n"
        log.Fatalf(m, ncol)
}
```

We import `log`.

305a ⟨*Imports, Ch. 34* 301b⟩+≡ (300) ◁304e 305e▷
```
"log"
```

To extract the categories, we track them with a map and store them in a slice.

305b ⟨*Extract categories, Ch. 34* 305b⟩≡ (304b)
```
var categories []string
cm := make(map[string]bool)
for _, d := range data {
        if !cm[d[2]] {
                categories = append(categories, d[2])
                cm[d[2]] = true
        }
}
```

We plot the data by constructing an output stream. We write `gnuplot` code to this output stream inside a goroutine and close it again. Outside of the goroutine we run `gnuplot`, unless the user opted for the script as output.

305c ⟨*Plot data, Ch. 34* 305c⟩≡ (304b)
```
⟨Construct output stream, Ch. 34 305d⟩
done := make(chan struct{})
go func() {
        ⟨Write gnuplot code to output stream, Ch. 34 306a⟩
        ⟨Close output stream, Ch. 34 308c⟩
        done <- struct{}{}
}()
if args.Script == "" {
        ⟨Run gnuplot, Ch. 34 308d⟩
}
<-done
```

The output stream is either the standard input stream of the `gnuplot` command, or a script, the name of which was supplied by the user.

305d ⟨*Construct output stream, Ch. 34* 305d⟩≡ (305c)
```
var w io.WriteCloser
var gcmd *exec.Cmd
var err error
if args.Script == "" {
        gcmd = exec.Command("gnuplot")
        w, err = gcmd.StdinPipe()
        if err != nil { log.Fatal(err) }
} else {
        w, err = os.Create(args.Script)
        if err != nil { log.Fatal(err) }
}
```

We import `exec` and `os`.

305e ⟨*Imports, Ch. 34* 301b⟩+≡ (300) ◁305a 306c▷
```
"os/exec"
"os"
```

When writing the `gnuplot` code, we begin with the terminal. Then we write the axes, the plot(s), and finally the data.

306a ⟨*Write* `gnuplot` *code to output stream, Ch. 34* 306a⟩≡ (305c)
⟨*Write terminal, Ch. 34* 306b⟩
⟨*Write axes, Ch. 34* 306f⟩
⟨*Write plot, Ch. 34* 307d⟩
⟨*Write data, Ch. 34* 308b⟩

The terminal is either encapsulated postscript or a window. If it is an interactive window, that window is persistent. We also set the plot size.

306b ⟨*Write terminal, Ch. 34* 306b⟩≡ (306a) 306d ▷
```
t := "set terminal"
if args.Ps != "" {
        t += " postscript eps color"
} else {
        t += " " + args.Win
}
if util.IsInteractive(args.Win) && args.Ps == "" {
        t += " persist"
}
t += " size " + args.Dim
fmt.Fprintf(w, "%s\n", t)
```

We import `fmt`.

306c ⟨*Imports, Ch. 34* 301b⟩+≡ (300) ◁305e
```
"fmt"
```

`gnuplot` version 5.4 patch level 3 generates screen plots with red background—at least on macOS. We make sure our plots are white.

306d ⟨*Write terminal, Ch. 34* 306b⟩+≡ (306a) ◁306b 306e ▷
```
if util.IsInteractive(args.Win) && args.Ps == "" {
        c := "set object 1 rectangle from screen 0,0 " +
                "to screen 1,1 fillcolor rgb 'white' behind"
        fmt.Fprintf(w, "%s\n", c)
}
```

If the terminal is postscript, we also set the output file.

306e ⟨*Write terminal, Ch. 34* 306b⟩+≡ (306a) ◁306d
```
if args.Ps != "" {
        fmt.Fprintf(w, "set output \"%s\"\n", args.Ps)
}
```

For the axes we write the labels, the log scale, and note the axes that have been unset.

306f ⟨*Write axes, Ch. 34* 306f⟩≡ (306a)
⟨*Write axis labels, Ch. 34* 307a⟩
⟨*Write log scale, Ch. 34* 307b⟩
⟨*Unset axes, Ch. 34* 307c⟩

We label the x-axis and the y-axis.

307a ⟨*Write axis labels, Ch. 34* 307a⟩≡ (306f)
```
if args.Xlab != "" {
        fmt.Fprintf(w, "set xlabel \"%s\"\n", args.Xlab)
}
if args.Ylab != "" {
        fmt.Fprintf(w, "set ylabel \"%s\"\n", args.Ylab)
}
```

The user can log-scale the x-axis, the y-axis, or both.

307b ⟨*Write log scale, Ch. 34* 307b⟩≡ (306f)
```
if strings.ContainsAny(args.Log, "xX") {
        fmt.Fprintf(w, "set logscale x\n")
}
if strings.ContainsAny(args.Log, "yY") {
        fmt.Fprintf(w, "set logscale y\n")
}
```

We unset the required axes.

307c ⟨*Unset axes, Ch. 34* 307c⟩≡ (306f)
```
if strings.ContainsAny(args.Unset, "xX") {
        fmt.Fprintf(w, "unset xtics\n")
}
if strings.ContainsAny(args.Unset, "yY") {
        fmt.Fprintf(w, "unset ytics\n")
}
```

The plot consists of a plot command, a style, and a separate plot for each category.

307d ⟨*Write plot, Ch. 34* 307d⟩≡ (306a)
```
⟨Write plot command, Ch. 34 307e⟩
⟨Construct style, Ch. 34 307f⟩
⟨Write one plot per category, Ch. 34 308a⟩
```

Right in front of the plot command we print the external `gnuplot` code, which we mark by comments.

307e ⟨*Write plot command, Ch. 34* 307e⟩≡ (307d)
```
if args.Gp != "" {
        m := "#Start external\n%s\n#End external\n"
        fmt.Fprintf(w, m, args.Gp)
}
fmt.Fprintf(w, "plot[%s][%s]", args.Xrange, args.Yrange)
```

The default style is “lines”, or `l`. However, the user might have opted either for “linespoints” (`lp`), or for “points” (`p`). The line is black if there is only one category and we always use dots, which correspond to line type 7, as points.

307f ⟨*Construct style, Ch. 34* 307f⟩≡ (307d)
```
style := "l"
if args.Points { style = "p pt 7" }
if args.LinesPoints { style = "lp pt 7" }
if len(categories) == 1 {
        style += " lc \"black\""
}
```

We write the instruction for the first plot and then append an instruction for each remaining category.

308a ⟨*Write one plot per category, Ch. 34* 308a⟩≡ (307d)
```
fmt.Fprintf(w, " \"-\" t \"%s\" w %s", categories[0], style)
for i := 1; i < len(categories); i++ {
        fmt.Fprintf(w, ", \"-\" t \"%s\" w %s",
                categories[i], style)
}
fmt.Fprintf(w, "\n")
```

For each category, we write the corresponding data set and terminate it with `e`.

308b ⟨*Write data, Ch. 34* 308b⟩≡ (306a)
```
for i, c := range categories {
        if i > 0 { fmt.Fprintf(w, "e\n") }
        for _, d := range data {
                if d[2] == c {
                        fmt.Fprintf(w, "%s\t%s\n",
                                d[0], d[1])
                }
        }
}
```

We close the output stream.

308c ⟨*Close output stream, Ch. 34* 308c⟩≡ (305c)
```
w.Close()
```

We run `gnuplot`, check for errors, and print its output, if any.

308d ⟨*Run gnuplot, Ch. 34* 308d⟩≡ (305c)
```
out, err := gcmd.Output()
util.CheckGnuplot(err)
if len(out) > 0 {
        fmt.Printf("%s", out)
}
```

We've finished `plotLine`, let's test it.

Testing

Out testing outline has hooks for imports and the testing logic.

308e ⟨*plotLine_test.go* 308e⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 34 309b⟩
)

func TestPlotLine(t *testing.T) {
        ⟨Testing, Ch. 34 309a⟩
}
```

We prepare a set of tests and then run them. In each test we write the output to a temporary `gnuplot` script, so that we can compare what we get to what we want. After testing, we remove the `gnuplot` script again.

309a ⟨*Testing, Ch. 34* 309a⟩≡ (308e)
```
var tests []*exec.Cmd
gf, err := ioutil.TempFile(".", "tmp_*.gp")
if err != nil { log.Fatal("can't open script file") }
g := gf.Name()
⟨Prepare tests, Ch. 34 309c⟩
for i, test := range tests {
        ⟨Run test, Ch. 34 312a⟩
}
err = os.Remove(g)
if err != nil { log.Fatalf("can't delete %q", g) }
```

We import `os` and `ioutil`.

309b ⟨*Testing imports, Ch. 34* 309b⟩≡ (308e) 310a ▷
```
"os"
"io/ioutil"
```

We create six sets of tests, lines & points, axis labels, plot size, log-scaling, ranges, and combinations of log-scaling and ranges. We also unset axes' tic marks, set external code, and use the dumb terminal.

309c ⟨*Prepare tests, Ch. 34* 309c⟩≡ (309a)
```
⟨Test lines and dots, Ch. 34 309d⟩
⟨Test axis labels, Ch. 34 310b⟩
⟨Test plot size, Ch. 34 310c⟩
⟨Test log-scaling, Ch. 34 310d⟩
⟨Test ranges, Ch. 34 310e⟩
⟨Test combinations of log-scaling and ranges, Ch. 34 311a⟩
⟨Test unsetting axes, Ch. 34 311b⟩
⟨Test external code, Ch. 34 311c⟩
⟨Test dumb terminal, Ch. 34 311d⟩
```

We have two sets of input data, `test3.dat` with the three columns of data shown in Figure 34.1A, and `test2.dat`, with just the first two columns of group g1.

We begin testing by drawing a plot of each, then we concentrate on `test3.dat`. Every time our output is a `gnuplot` file, which we can later compare with the output we want.

309d ⟨*Test lines and dots, Ch. 34* 309d⟩≡ (309c)
```
test := exec.Command("./plotLine", "-s", g, "test2.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g, "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g, "-P", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g, "-L", "test3.dat")
tests = append(tests, test)
```

We import `exec`.

310a ⟨*Testing imports, Ch. 34* 309b⟩+≡ (308e) ◁309b 312b▷
```
"os/exec"
```

We label the axes individually and then both of them.

310b ⟨*Test axis labels, Ch. 34* 310b⟩≡ (309c)
```
test = exec.Command("./plotLine", "-s", g,
          "-x", "x", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-y", "y", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-x", "x", "-y", "y", "test3.dat")
tests = append(tests, test)
```

We set the plot size in combination with postscript output.

310c ⟨*Test plot size, Ch. 34* 310c⟩≡ (309c)
```
test = exec.Command("./plotLine", "-s", g,
          "-p", "test.ps", "-d", "340,340", "test3.dat")
tests = append(tests, test)
```

We set the x-axis to log-scale, the y-axis, and both of them.

310d ⟨*Test log-scaling, Ch. 34* 310d⟩≡ (309c)
```
test = exec.Command("./plotLine", "-s", g,
          "-l", "x", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-l", "y", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-l", "xy", "test3.dat")
tests = append(tests, test)
```

We set a range for the x-axis, for the y-axis, and for both axes.

310e ⟨*Test ranges, Ch. 34* 310e⟩≡ (309c)
```
test = exec.Command("./plotLine", "-s", g,
          "-X", "0.1:10", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-Y", "0.2:100", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g, "-X", "0.1:10",
          "-Y", "0.2:100", "test3.dat")
tests = append(tests, test)
```

We combine log-scaling and limits.

311a ⟨*Test combinations of log-scaling and ranges, Ch. 34* 311a⟩≡ (309c)

```
test = exec.Command("./plotLine", "-s", g,
          "-X", "0.1:10", "-l", "x", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-Y", "0.2:100", "-l", "x", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-X", "0.1:10", "-l", "xy", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-X", "0.1:10", "-Y", "0.2:100", "-l", "xy", "test3.dat")
tests = append(tests, test)
```

We unset the x-axis, the y-axis, and both.

311b ⟨*Test unsetting axes, Ch. 34* 311b⟩≡ (309c)

```
test = exec.Command("./plotLine", "-s", g,
          "-u", "x", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-u", "y", "test3.dat")
tests = append(tests, test)
test = exec.Command("./plotLine", "-s", g,
          "-u", "xy", "test3.dat")
tests = append(tests, test)
```

We set a title using external code.

311c ⟨*Test external code, Ch. 34* 311c⟩≡ (309c)

```
test = exec.Command("./plotLine", "-s", g,
          "-g", "set title \"External Title\"",
          "test3.dat")
tests = append(tests, test)
```

We test the dumb terminal.

311d ⟨*Test dumb terminal, Ch. 34* 311d⟩≡ (309c)

```
test = exec.Command("./plotLine", "-s", g,
          "-t", "dumb", "test3.dat")
tests = append(tests, test)
```

For each test we compare what we get in `g` with what we want in `results/r1.gp`, `results/r2.gp`, and so on. On a darwin system, the results files are called `results/r1d.gp`, `results/r2d.gp`, and so on.

312a ⟨*Run test, Ch. 34* 312a⟩≡ (309a)

```
err := test.Run()
if err != nil { log.Fatalf("can't run %q", test) }
get, err := ioutil.ReadFile(g)
f := "results/r" + strconv.Itoa(i+1)
if runtime.GOOS == "darwin" { f += "d" }
f += ".gp"
want, err := ioutil.ReadFile(f)
if err != nil { log.Fatalf("can't open %q", f) }
if !bytes.Equal(get, want) {
          t.Errorf("%s:\nget:\n%s\nwant:\n%s\n",
                  test, string(get), string(want))
}
```

We import `log`, `ioutil`, `strconv`, `runtime`, and `bytes`.

312b ⟨*Testing imports, Ch. 34* 309b⟩+≡ (308e) ◁310a

```
"log"
"io/ioutil"
"strconv"
"runtime"
"bytes"
```

Chapter 35

Program `plotSeg`: Plotting Segments

A

1	1	57	57
65	65	229	229
214	226	235	247
226	244	309	327
303	320	392	409

B

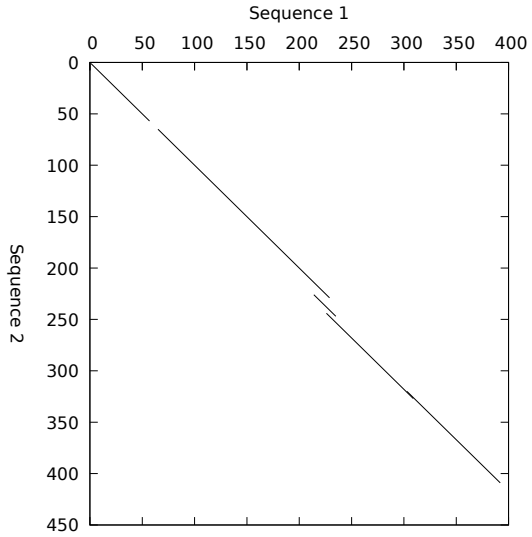


Figure 35.1: Example data (**A**) plotted with `plotSeg` (**B**).

Introduction

Segment plots, also known as dot plots, display a potentially large number of exact matches between two sequences. Each exact match is a segment, which is denoted by a quartet of numbers (x_1, y_1, x_2, y_2), where (x_1, y_1) is the start of the segment and (x_2, y_2) its end. The program `mumPlot` (Ch. 26) generates such quartets from MUMmer output. Figure 35.1A shows some sample input data and Figure 35.1B its plot generated with `plotSeg`. Like the other `plot*` programs, `plotSeg` calls `gnuplot` [19] for rendering.

Implementation

The outline of `plotSeg` contains hooks for imports, types, functions, and the logic of the main function.

314 ⟨*plotSeg.go* 314⟩≡

```
package main

import (
        ⟨Imports, Ch. 35 315b⟩
)

⟨Types, Ch. 35 316e⟩
⟨Functions, Ch. 35 317f⟩
func main() {
        ⟨Main function, Ch. 35 315a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

315a ⟨*Main function, Ch. 35* 315a⟩≡ (314)
```
util.PrepLog("plotSeg")
⟨Set usage, Ch. 35 315c⟩
⟨Declare options, Ch. 35 315e⟩
⟨Parse options, Ch. 35 316c⟩
⟨Parse input files, Ch. 35 317e⟩
```

We import `util`.

315b ⟨*Imports, Ch. 35* 315b⟩≡ (314) 315d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `plotSeg`, and an example command.

315c ⟨*Set usage, Ch. 35* 315c⟩≡ (315a)
```
u := "plotSeg [-h] [option]... [foo.dat]..."
p := "Generate segment plots, also known as dot plots."
e := "mum2plot eco_x_y.mum | plotSeg"
clio.Usage(u, p, e)
```

We import `clio`.

315d ⟨*Imports, Ch. 35* 315b⟩+≡ (314) ◁315b 315f▷
```
"github.com/evolbioinf/clio"
```

Apart from the obligatory version option, we declare options for the axes and the graphics device. We also declare a catch-all "general option".

315e ⟨*Declare options, Ch. 35* 315e⟩≡ (315a)
```
optV := flag.Bool("v", false, "version")
⟨Declare axis options, Ch. 35 315g⟩
⟨Declare device options, Ch. 35 316a⟩
⟨Declare general option, Ch. 35 316b⟩
```

We import `flag`.

315f ⟨*Imports, Ch. 35* 315b⟩+≡ (314) ◁315d 317b▷
```
"flag"
```

The user can label the axes and set their ranges. `gnuplot` code.

315g ⟨*Declare axis options, Ch. 35* 315g⟩≡ (315e)
```
optX := flag.String("x", "", "x-label")
optY := flag.String("y", "", "y-label")
optXX := flag.String("X", "*:*", "x-range")
optYY := flag.String("Y", "*:*", "y-range")
```

The user can chose the terminal for displaying the plot. Moreover, instead of showing the plot in a window, the user can write it as encapsulated postscript and set its size. To guide the user, we provide three default sizes, 640×384 pixels for screen, 5×3.5 in for postscript, and 79×24 characters for the "dumb" terminal. The user can also ask for the `gnuplot` script of the plot.

316a ⟨*Declare device options, Ch. 35* 316a⟩≡ (315e)
```
optT := flag.String("t", "", "terminal (default wxt, qt on darwin)")
optP := flag.String("p", "", "encapsulated postscript file")
defScrDim := "640,384"
defPsDim := "5,3.5"
defDumbDim := "79,24"
optD := flag.String("d", defScrDim, "plot dimensions; " +
          "pixels for screen, " + defPsDim + " in for ps, " +
          defDumbDim + " char for dumb")
optS := flag.String("s", "", "write gnuplot script to file")
```

The "general option" is a switch for submitting arbitrary `gnuplot` code.

316b ⟨*Declare general option, Ch. 35* 316b⟩≡ (315e)
```
optG := flag.String("g", "", "gnuplot code")
```

We parse the options and respond to a request for the version first (`-v`), as this terminates `plotSeg`. Then we declare the variable `opts` for holding the options and collect their values.

316c ⟨*Parse options, Ch. 35* 316c⟩≡ (315a)
```
flag.Parse()
⟨Respond to -v, Ch. 35 316d⟩
opts := new(Options)
⟨Collect option values, Ch. 35 316f⟩
```

We respond to `-v` by printing a standardized message.

316d ⟨*Respond to -v, Ch. 35* 316d⟩≡ (316c)
```
if *optV {
          util.PrintInfo("plotSeg")
}
```

We declare the type for holding the options and fill it with fields as we go along.

316e ⟨*Types, Ch. 35* 316e⟩≡ (314)
```
type Options struct {
          ⟨Options fields, Ch. 35 317c⟩
}
```

We collect the axis labels, axis ranges, the plot size, and the names of the files for postscript and `gnuplot` output.

316f ⟨*Collect option values, Ch. 35* 316f⟩≡ (316c) 317a▷
```
opts.Xlab = *optX
opts.Ylab = *optY
opts.Xrange = *optXX
opts.Yrange = *optYY
opts.Dim = *optD
opts.Ps = *optP
opts.Script = *optS
opts.Gp = *optG
```

The one option we haven't collected yet is the terminal. This is wxt by default, but if we are running on darwin under macOS, the default is qt.

317a ⟨*Collect option values, Ch. 35* 316f⟩+≡ (316c) ◁316f 317d▷
```
opts.Win = *optT
if opts.Win == "" {
        opts.Win = "wxt"
        if runtime.GOOS == "darwin" {
                opts.Win = "qt"
        }
}
```

We import `runtime`.

317b ⟨*Imports, Ch. 35* 315b⟩+≡ (314) ◁315f 317g▷
```
"runtime"
```

We add the fields we just used to `Options`.

317c ⟨*`Options` fields, Ch. 35* 317c⟩≡ (316e)
```
Xlab, Ylab, Xrange, Yrange, Dim string
Width, Height float64
Win, Ps, Script, Gp string
```

If the user requested postscript output but didn't set a size, we set the default postscript size.

If the user requested postscript or dumb and didn't set a size, we set the default.

317d ⟨*Collect option values, Ch. 35* 316f⟩+≡ (316c) ◁317a
```
if opts.Dim == defScrDim {
        if opts.Ps != "" {
                opts.Dim = defPsDim
        } else if opts.Win == "dumb" {
                opts.Dim = defDumbDim
        }
}
```

The remaining tokens on the command line are taken as input files. The function `ParseFiles` applies the function `scan` to each input file. `scan` takes as argument the options.

317e ⟨*Parse input files, Ch. 35* 317e⟩≡ (315a)
```
files := flag.Args()
clio.ParseFiles(files, scan, opts)
```

Inside `scan`, we retrieve the options, read the segments, and plot them.

317f ⟨*Functions, Ch. 35* 317f⟩≡ (314)
```
func scan(r io.Reader, args ...interface{}) {
        opts := args[0].(*Options)
        ⟨Read segments, Ch. 35 318a⟩
        ⟨Construct output stream, Ch. 35 318c⟩
        ⟨Plot segments, Ch. 35 319a⟩
}
```

We import `io`.

317g ⟨*Imports, Ch. 35* 315b⟩+≡ (314) ◁317b 318b▷
```
"io"
```

We read the segments, skip comments, and check each data line has four columns. If not, something has gone wrong and we bail.

318a ⟨*Read segments, Ch. 35* 318a⟩≡ (317f)
```
sc := bufio.NewScanner(r)
var segments [][]string
for sc.Scan() {
        row := sc.Text()
        if row[0] == '#' { continue }
        fields := strings.Fields(row)
        l := len(fields)
        if l != 4 {
                log.Fatalf("get %d columns, want 4\n", l)
        }
        segments = append(segments, fields)
}
```

We import `bufio`, `strings`, and `log`.

318b ⟨*Imports, Ch. 35* 315b⟩+≡ (314) ◁317g 318d▷
```
"bufio"
"strings"
"log"
```

The output stream is either the standard input stream of the `gnuplot` command, or a script of `gnuplot` code.

318c ⟨*Construct output stream, Ch. 35* 318c⟩≡ (317f)
```
var w io.WriteCloser
var gcmd *exec.Cmd
var err error
if opts.Script == "" {
        gcmd = exec.Command("gnuplot")
        w, err = gcmd.StdinPipe()
        if err != nil { log.Fatal(err) }
} else {
        w, err = os.Create(opts.Script)
        if err != nil { log.Fatal(err) }
}
```

We import `exec` and `os`.

318d ⟨*Imports, Ch. 35* 315b⟩+≡ (314) ◁318b 319d▷
```
"os/exec"
"os"
```

We use a Go routine to write the `gnuplot` code to that output stream, which we also close inside that routine. After the Go routine, we run `gnuplot`, unless the user opted for the script instead.

319a ⟨*Plot segments, Ch. 35* 319a⟩≡ (317f)
```
done := make(chan struct{})
go func() {
        ⟨Write gnuplot code to output stream, Ch. 35 319b⟩
        ⟨Close output stream, Ch. 35 320e⟩
        done <- struct{}{}
}()
if opts.Script == "" {
        ⟨Run gnuplot, Ch. 35 320f⟩
}
<-done
```

The `gnuplot` code describes the terminal, the axes, the plot, and the segments.

319b ⟨*Write* `gnuplot` *code to output stream, Ch. 35* 319b⟩≡ (319a)
```
⟨Write terminal, Ch. 35 319c⟩
⟨Write axes, Ch. 35 320b⟩
⟨Write plot, Ch. 35 320c⟩
⟨Write segments, Ch. 35 320d⟩
```

The terminal is either monochrome encapsulate postscript or a window. If it is an interactive window, that window is persistent. We also set the plot size.

319c ⟨*Write terminal, Ch. 35* 319c⟩≡ (319b) 319e ▷
```
t := "set terminal"
if opts.Ps != "" {
        t += " postscript eps monochrome"
} else {
        t += " " + opts.Win
}
if util.IsInteractive(opts.Win) && opts.Ps == "" {
        t += " persist"
}
t += " size " + opts.Dim
fmt.Fprintf(w, "%s\n", t)
```

We import `fmt`.

319d ⟨*Imports, Ch. 35* 315b⟩+≡ (314) ◁318d
```
"fmt"
```

`gnuplot` version 5.4 patch level 3 generates screen plots with red background—at least on macOS. We make sure our plots are white.

319e ⟨*Write terminal, Ch. 35* 319c⟩+≡ (319b) ◁319c 320a ▷
```
if util.IsInteractive(opts.Win) && opts.Ps == "" {
        c := "set object 1 rectangle from screen 0,0 " +
                "to screen 1,1 fillcolor rgb 'white' behind"
        fmt.Fprintf(w, "%s\n", c)
}
```

If the terminal is postscript, we also set the output file.

320a ⟨*Write terminal, Ch. 35* 319c⟩+≡ (319b) ◁319e
```
if opts.Ps != "" {
        fmt.Fprintf(w, "set output \"%s\"\n", opts.Ps)
}
```

We set the x2 axis as the primary x axis. This involves removing the tics from the x axis and switching on the x2 axis and mirroring its tic marks. We also set the x2 range and the y range. The y range is inverted. Then we label the x axis and the y axis. The label on the y axis is rotated by -90 degrees to track the direction of the sequence.

320b ⟨*Write axes, Ch. 35* 320b⟩≡ (319b)
```
fmt.Fprintf(w, "set format x ''\n")
fmt.Fprintf(w, "unset xtics\n")
fmt.Fprintf(w, "set x2tics mirror\n")
fmt.Fprintf(w, "set xrange[%s]\n", opts.Xrange)
fmt.Fprintf(w, "set yrange [%s] reverse\n", opts.Yrange)
fmt.Fprintf(w, "set x2label '%s'\n", opts.Xlab)
fmt.Fprintf(w, "set ylabel rotate by -90 '%s'\n", opts.Ylab)
```

Just before drawing the actual plot, we enter the `gnuplot` code submitted by the user. Then we write the plot as an untitled black line plot.

320c ⟨*Write plot, Ch. 35* 320c⟩≡ (319b)
```
if opts.Gp != "" { fmt.Fprintf(w, "%s\n", opts.Gp) }
fmt.Fprintf(w, "plot \"-\" t '' w l lc \"black\"\n")
```

The segments are written as pairs of points separated by a blank line. This is interpreted by `gnuplot` as a line disconnected from the next line, a *segment*.

320d ⟨*Write segments, Ch. 35* 320d⟩≡ (319b)
```
for _, s := range segments {
        fmt.Fprintln(w, s[0], s[1])
        fmt.Fprintln(w, s[2], s[3])
        fmt.Fprintln(w)
}
```

We close the output stream.

320e ⟨*Close output stream, Ch. 35* 320e⟩≡ (319a)
```
w.Close()
```

We run `gnuplot`, check its error, and print its output, if any.

320f ⟨*Run gnuplot, Ch. 35* 320f⟩≡ (319a)
```
out, err := gcmd.Output()
util.CheckGnuplot(err)
if len(out) > 0 {
        fmt.Printf("%s", out)
}
```

The program `plotSeg` is finished, let's test it.

Testing

Our program for testing `plotSeg` contains hooks for imports and the testing logic.

321a ⟨*plotSeg_test.go* 321a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 35 321c⟩
)

func TestPlotSeg(t *testing.T) {
        ⟨Testing, Ch. 35 321b⟩
}
```

We write the individual tests and run them in a loop. Each test is written to the same unique `gnuplot` file, which we remove again after the tests.

321b ⟨*Testing, Ch. 35* 321b⟩≡ (321a)
```
gf, err := ioutil.TempFile(".", "tmp_*.gp")
if err != nil { log.Fatal("cant open output file") }
g := gf.Name()
var tests []*exec.Cmd
⟨Construct tests, Ch. 35 321d⟩
for i, test := range tests {
        ⟨Run test, Ch. 35 323a⟩
}
err = os.Remove(g)
if err != nil { log.Fatalf("can't delete %q", g) }
```

We import `ioutil`, `log`, `exec`, and `os`.

321c ⟨*Testing imports, Ch. 35* 321c⟩≡ (321a) 323b ▷
```
"io/ioutil"
"log"
"os/exec"
"os"
```

We test setting axis labels, ranges, plot size, submitting `gnuplot` code, and the dumb terminal.

321d ⟨*Construct tests, Ch. 35* 321d⟩≡ (321b)
```
⟨Test axis labels, Ch. 35 322a⟩
⟨Test axis ranges, Ch. 35 322b⟩
⟨Test plot size, Ch. 35 322c⟩
⟨Test gnuplot code, Ch. 35 322d⟩
⟨Test dumb terminal, Ch. 35 322e⟩
```

We begin by running `plotSeg` on the example data in `test.dat` without any other options. Then we set the x-label, the y-label, and both.

322a ⟨*Test axis labels, Ch. 35* 322a⟩≡ (321d)

```
f := "test.dat"
te := exec.Command("./plotSeg", "-s", g, f)
tests = append(tests, te)
te = exec.Command("./plotSeg", "-s", g, "-x", "x", f)
tests = append(tests, te)
te = exec.Command("./plotSeg", "-s", g, "-y", "y", f)
tests = append(tests, te)
te = exec.Command("./plotSeg", "-s", g, "-x", "x",
          "-y", "y", f)
tests = append(tests, te)
```

We set the x-range, the y-range, and both.

322b ⟨*Test axis ranges, Ch. 35* 322b⟩≡ (321d)

```
te = exec.Command("./plotSeg", "-s", g, "-X", "100:500", f)
tests = append(tests, te)
te = exec.Command("./plotSeg", "-s", g, "-Y", "100:500", f)
tests = append(tests, te)
te = exec.Command("./plotSeg", "-s", g, "-X", "100:500",
          "-Y", "100:500", f)
tests = append(tests, te)
```

We set the plot dimensions.

322c ⟨*Test plot size, Ch. 35* 322c⟩≡ (321d)

```
te = exec.Command("./plotSeg", "-s", g, "-d", "300,300", f)
tests = append(tests, te)
```

We set a title via `gnuplot` code.

322d ⟨*Test* `gnuplot` *code, Ch. 35* 322d⟩≡ (321d)

```
te = exec.Command("./plotSeg", "-s", g, "-g",
          "set title \"External Title\"", f)
tests = append(tests, te)
```

We test the dumb terminal.

322e ⟨*Test dumb terminal, Ch. 35* 322e⟩≡ (321d)

```
te = exec.Command("./plotSeg", "-s", g, "-t", "dumb", f)
tests = append(tests, te)
```

For each test we compare what we get with what we want, which is stored in a file the name of which we still need to construct.

323a ⟨*Run test, Ch. 35* 323a⟩≡ (321b)
```
err = test.Run()
if err != nil { log.Fatalf("can't run %q", test) }
get, err := ioutil.ReadFile(g)
if err != nil { log.Fatalf("can't read %q", g) }
⟨Construct file name, Ch. 35 323c⟩
want, err := ioutil.ReadFile(f)
if err != nil { log.Fatalf("can't read %q", f) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n",
                  string(get), string(want))
}
```

We import `bytes`.

323b ⟨*Testing imports, Ch. 35* 321c⟩+≡ (321a) ◁321c 323d▷
```
"bytes"
```

On default systems, the results we want are stored in files `results/r1.gp`, `results/r2.gp`, and so on. On darwin systems, the names are extended by "d".

323c ⟨*Construct file name, Ch. 35* 323c⟩≡ (323a)
```
f = "results/r" + strconv.Itoa(i+1)
if runtime.GOOS == "darwin" {
          f += "d"
}
f += ".gp"
```

We import `strconv` and `runtime`.

323d ⟨*Testing imports, Ch. 35* 321c⟩+≡ (321a) ◁323b
```
"strconv"
"runtime"
```

Chapter 36

Program `plotTree`: Plotting Trees

A

```
(((One:0.2,Two:0.3)75:0.3,(Three:0.5,Four:0.3)69:0.2):0.3,Five:0.7):0.0;
```

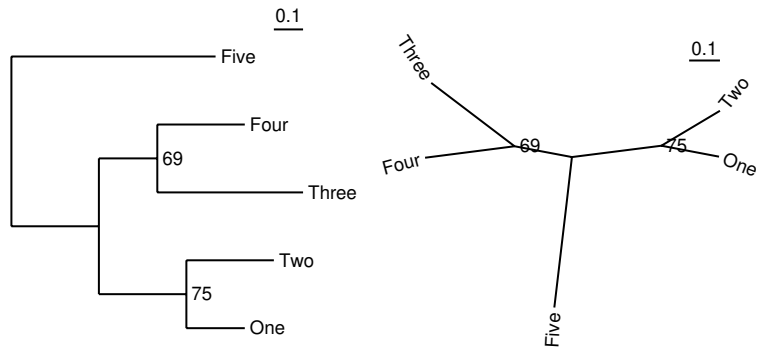


Figure 36.1: The program `drawTree` reads a tree in Newick format (**A**) and converts it into a rooted (**B**) or an unrooted (**C**) phylogeny.

Introduction

We'd often like to draw a quick phylogeny from a tree given in Newick format, for example the one shown in Figure 36.1A. The program `plotTree` converts such a tree string either into a rooted phylogeny (Figure 36.1B) or an unrooted phylogeny (Figure 36.1C). The user can also opt for an encapsulated postscript file and may omit the node labels. The phylogeny is rendered in `gnuplot`.

Implementation

The outline of `plotTree` has hooks for imports, types, functions, and the logic of the main function.

325 ⟨*plotTree.go* 325⟩≡

```
package main

import (
        ⟨Imports, Ch. 36 326b⟩
)
⟨Types, Ch. 36 327d⟩
⟨Functions, Ch. 36 329a⟩
func main() {
        ⟨Main function, Ch. 36 326a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options and parse the options. Then we parse the input files.

326a ⟨*Main function, Ch. 36* 326a⟩≡ (325)
```
util.PrepLog("plotTree")
⟨Set usage, Ch. 36 326c⟩
⟨Declare options, Ch. 36 326e⟩
⟨Parse options, Ch. 36 327c⟩
⟨Parse input files, Ch. 36 328e⟩
```

We import `util`.

326b ⟨*Imports, Ch. 36* 326b⟩≡ (325) 326d▷
```
"github.com/evolbioinf/biobox/util"
```

User Interaction

The usage consists of three parts, the actual usage message, an explanation of the purpose of `plotTree`, and an example command.

326c ⟨*Set usage, Ch. 36* 326c⟩≡ (326a)
```
u := "plotTree [-h] [option]... [foo.nwk]..."
p := "Plot Newick-formatted trees."
e := "plotTree foo.nwk"
clio.Usage(u, p, e)
```

We import the package `clio`.

326d ⟨*Imports, Ch. 36* 326b⟩+≡ (325) ◁326b 326f▷
```
"github.com/evolbioinf/clio"
```

Apart from the built-in help option (`-h`), we declare a version option (`-v`), and a set of program-specific options: By default, we interpret a bifurcating root as belonging to a rooted tree drawn like Figure 36.1B, and a trifurcating root as belonging to an unrooted tree drawn like Figure 36.1C. However, the user can enforce a rooted (`-r`) or an unrooted (`-u`) layout irrespective of the degree of the root.

326e ⟨*Declare options, Ch. 36* 326e⟩≡ (326a) 326g▷
```
optV := flag.Bool("v", false, "version")
optR := flag.Bool("r", false, "rooted tree (default input)")
optU := flag.Bool("u", false, "unrooted tree (default input)")
```

We import `flag`.

326f ⟨*Imports, Ch. 36* 326b⟩+≡ (325) ◁326d 328d▷
```
"flag"
```

The nodes of a Newick tree may or may not be labeled. By default, `plotTree` draws the labels, but the user can omit them (`-n`).

326g ⟨*Declare options, Ch. 36* 326e⟩+≡ (326a) ◁326e 327a▷
```
optN := flag.Bool("n", false, "no node labels (default input)")
```

The default output is drawn to the screen, for which the user can set the terminal. Alternatively, the user can draw the tree to a postscript file (`-p`) and give the plot custom dimensions (`-d`). To guide the user, we provide three default dimensions, 640×384 pixels for screen, 5×3.5 in for postscript, and 79×24 characters for the "dumb" terminal. The user can also set the plot margins as a fraction of the plot size (`-m`), set the scale of the tree (`-c`), and inject arbitrary `gnuplot` code.

327a ⟨*Declare options, Ch. 36* 326e⟩+≡ (326a) ◁326g 327b▷

```
optT := flag.String("t", "", "terminal (default wxt, qt on darwin)")
optP := flag.String("p", "", "encapsulated postscript file")
defScrDim := "640,384"
defPsDim := "5,3.5"
defDumbDim := "79,24"
optD := flag.String("d", defScrDim, "plot dimensions; " +
          "pixels for screen, " + defPsDim + " in for ps, " +
          defDumbDim + " char for dumb")
optM := flag.Float64("m", 0.2, "margin")
optC := flag.Float64("c", 0.0, "scale")
optG := flag.String("g", "", "gnuplot code")
```

Finally, the user may opt to print the `gnuplot` script.

327b ⟨*Declare options, Ch. 36* 326e⟩+≡ (326a) ◁327a

```
optS := flag.String("s", "", "write gnuplot script to file")
```

We parse the options and immediately respond to `-v` as this might stop `plotTree`. Then we create the variable `opts` and store the option values in it.

327c ⟨*Parse options, Ch. 36* 327c⟩≡ (326a) 328c▷

```
flag.Parse()
if *optV {
          util.PrintInfo("plotTree")
}
opts := new(opts)
⟨Store options, Ch. 36 327e⟩
```

We declare the `opts` type as a struct with a hook for the fields we need.

327d ⟨*Types, Ch. 36* 327d⟩≡ (325) 329e▷

```
type opts struct {
          ⟨Opts fields, Ch. 36 328a⟩
}
```

We store the the options.

327e ⟨*Store options, Ch. 36* 327e⟩≡ (327c) 328b▷

```
opts.Rooted = *optR
opts.Unrooted = *optU
opts.NoLabels = *optN
opts.Ps = *optP
opts.Dim = *optD
opts.Margin = *optM
opts.Scale = *optC
opts.Script = *optS
opts.Win = *optT
opts.Code = *optG
```

We declare the fields we just used.

328a ⟨*Opts fields, Ch. 36* 328a⟩≡ (327d) 338f▷
```
Rooted, Unrooted, NoLabels bool
Ps, Dim string
Margin, Scale float64
Script, Win, Code string
```

If the user chose postscript or dumb and didn't set a size, we set the default.

328b ⟨*Store options, Ch. 36* 327e⟩+≡ (327c) ◁327e
```
if opts.Dim == defScrDim {
        if opts.Ps != "" {
                opts.Dim = defPsDim
        } else if opts.Win == "dumb" {
                opts.Dim = defDumbDim
        }
}
```

If the user didn't set a terminal, we set it to wxt unless we are running on the darwin system (macOS), in which case we switch to qt.

328c ⟨*Parse options, Ch. 36* 327c⟩+≡ (326a) ◁327c
```
if opts.Win == "" {
        opts.Win = "wxt"
        if runtime.GOOS == "darwin" {
                opts.Win = "qt"
        }
}
```

We import `runtime`.

328d ⟨*Imports, Ch. 36* 326b⟩+≡ (325) ◁326f 329b▷
```
"runtime"
```

Scan Input Files

The remaining tokens on the command line are interpreted as tree files. Each of them is parsed with the function `scan`, which takes the file names, a file counter, and the options as arguments.

328e ⟨*Parse input files, Ch. 36* 328e⟩≡ (326a)
```
files := flag.Args()
fileCounter := 0
clio.ParseFiles(files, scan, files, &fileCounter, opts)
```

Inside `scan`, we retrieve the arguments. Then we iterate over the trees, and count each one. A tree is represented by its root node, which we convert from the root returned by the scanner. Then we draw the newly converted tree. Having drawn all the trees in the file, we increment the file counter.

329a ⟨*Functions, Ch. 36* 329a⟩≡ (325) 329d▷
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 36 329c⟩
        sc := nwk.NewScanner(r)
        treeCounter := 0
        for sc.Scan() {
                treeCounter++
                root := convertTree(sc.Tree())
                ⟨Draw tree, Ch. 36 330c⟩
        }
        ⟨Increment file counter, Ch. 36 342d⟩
}
```

We import `io` and `nwk`.

329b ⟨*Imports, Ch. 36* 326b⟩+≡ (325) ◁328d 335b▷
```
"io"
"github.com/evolbioinf/nwk"
```

We retrieve the files, the file counter, and the options.

329c ⟨*Retrieve arguments, Ch. 36* 329c⟩≡ (329a)
```
files := args[0].([]string)
fileCounter := args[1].(*int)
opts := args[2].(*opts)
```

We convert the new tree from nodes of type `nwk.Node` to our local node type. This allows us to tailor the nodes to fit the layout task in hand.

329d ⟨*Functions, Ch. 36* 329a⟩+≡ (325) ◁329a 330a▷
```
func convertTree(v *nwk.Node) *node {
        root := new(node)
        cpTree(v, root)
        return root
}
```

A `node` replicates the fields of a `nwk.Node` and has a hook for additional fields we add later.

329e ⟨*Types, Ch. 36* 327d⟩+≡ (325) ◁327d 331a▷
```
type node struct {
        child, sib, parent *node
        label string
        length float64
        hasLength bool
        ⟨Node fields, Ch. 36 332b⟩
}
```

In the function `cpTree` we copy each node. We begin by copying the label and the branch length, then we copy the tree topology.

330a ⟨*Functions, Ch. 36* 329a⟩+≡ (325) ◁329d 332a▷

```
func cpTree(v *nwk.Node, n *node) {
          if v == nil { return }
          n.label = v.Label
          n.length = v.Length
          n.hasLength = v.HasLength
          ⟨Copy tree topology, Ch. 36 330b⟩
          cpTree(v.Child, n.child)
          cpTree(v.Sib, n.sib)
}
```

The tree topology consists of references to child, sib, and parent.

330b ⟨*Copy tree topology, Ch. 36* 330b⟩≡ (330a)

```
if v.Child != nil {
          c := new(node)
          c.parent = n
          n.child = c

}
if v.Sib != nil {
          s := new(node)
          s.parent = n.parent
          n.sib = s
}
```

Draw Tree

We draw a tree by constructing its segments and its title. Then we construct an output stream and write the segments to it.

330c ⟨*Draw tree, Ch. 36* 330c⟩≡ (329a)

```
⟨Construct tree segments, Ch. 36 330d⟩
⟨Construct plot title, Ch. 36 338d⟩
⟨Construct output stream, Ch. 36 339a⟩
⟨Write segments to output stream, Ch. 36 339c⟩
```

We decide whether the tree is to be drawn in rooted or unrooted format. Then we layout the tree accordingly and store its segments. A special segment is the scale, which we add last.

330d ⟨*Construct tree segments, Ch. 36* 330d⟩≡ (330c)

```
var segments []segment
rooted := false
⟨Is the tree rooted or unrooted? Ch. 36 331b⟩
if rooted {
          ⟨Layout rooted tree, Ch. 36 331e⟩
} else {
          ⟨Layout unrooted tree, Ch. 36 334a⟩
}
⟨Add scale, Ch. 36 336c⟩
```

A segment consists of a start and an end position, a label of the start position, an angle of that label, and an orientation. The orientation is either `l` for *left* or `r` for *right*.

331a ⟨*Types, Ch. 36* 327d⟩+≡ (325) ◁329e 337a▷

```
type segment struct {
          x1, y1, x2, y2 float64
          l string
          a, h, v float64
          o string
}
```

By default, we base the layout on the root's number of children. However, the user might have used the switches for rooted or unrooted layout.

331b ⟨*Is the tree rooted or unrooted? Ch. 36* 331b⟩≡ (330d)

⟨*Base layout on the root's number of children, Ch. 36* 331c⟩
⟨*Base layout on user options, Ch. 36* 331d⟩

At this point the tree is treated as unrooted, but if the root has two children or less, we set it to rooted.

331c ⟨*Base layout on the root's number of children, Ch. 36* 331c⟩≡ (331b)

```
w := root.child
n := 0
for w != nil {
          n++
          w = w.sib
}
if n <= 2 {
          rooted = true
}
```

The default layout can be overridden by the user.

331d ⟨*Base layout on user options, Ch. 36* 331d⟩≡ (331b)

```
if opts.Rooted {
          rooted = true
}
if opts.Unrooted {
          rooted = false
}
```

We layout a rooted tree by setting the x and y coordinates of its nodes in a tree traversal. For setting the y coordinate we refer to a "global" y position. Then we collect the branches.

331e ⟨*Layout rooted tree, Ch. 36* 331e⟩≡ (330d)

```
setXcoords(root)
y := 0.0
y = setYcoords(root, y)
segments = collectBranchesR(root, segments, opts)
```

The x coordinates are set recursively by adding the branch length to the parent's x coordinate. This means the root's x coordinate is zero.

332a ⟨*Functions, Ch. 36* 329a⟩+≡ (325) ◁330a 332c▷

```
func setXcoords(v *node) {
        if v == nil { return }
        if v.parent != nil {
                l := v.length
                if !v.hasLength { l = 1.0 }
                v.x = l + v.parent.x
        }
        setXcoords(v.child)
        setXcoords(v.sib)
}
```

We declare fields for the x and y coordinates of a node.

332b ⟨*Node fields, Ch. 36* 332b⟩≡ (329e) 334b▷

```
x, y float64
```

The y coordinates are set separately for leaves and internal nodes.

332c ⟨*Functions, Ch. 36* 329a⟩+≡ (325) ◁332a 333a▷

```
func setYcoords(v *node, y float64) float64 {
        if v == nil { return y }
        y = setYcoords(v.child, y)
        if v.child == nil {
                ⟨Set y coordinate of leaf, Ch. 36 332d⟩
        } else {
                ⟨Set y coordinate of internal node, Ch. 36 332e⟩
        }
        y = setYcoords(v.sib, y)
        return y
}
```

The leaves are spaced evenly along the y axis using the y coordinate passed.

332d ⟨*Set y coordinate of leaf, Ch. 36* 332d⟩≡ (332c)

```
v.y = y
y++
```

Internal nodes are centered on their children.

332e ⟨*Set y coordinate of internal node, Ch. 36* 332e⟩≡ (332c)

```
w := v.child
min := w.y
for w.sib != nil {
        w = w.sib
}
max := w.y
v.y = (min + max) / 2.0
```

We collect the branches of the rooted tree recursively, treating the root separately from all other nodes.

333a ⟨*Functions, Ch. 36* 329a⟩+≡ (325) ◁332c 334c▷

```
func collectBranchesR(v *node, segments []segment, o *opts) []segment {
        if v == nil { return segments }
        if v.parent == nil {
                ⟨Treat root in rooted tree, Ch. 36 333b⟩
        } else {
                ⟨Treat other node in rooted tree, Ch. 36 333c⟩
        }
        segments = collectBranchesR(v.child, segments, o)
        segments = collectBranchesR(v.sib, segments, o)
        return segments
}
```

The root may be labeled, in which case we add an empty segment with the label. To prevent a label from touching the point labeled, we pad the label with a blank.

333b ⟨*Treat root in rooted tree, Ch. 36* 333b⟩≡ (333a)

```
if v.label != "" && !o.NoLabels {
        label := " " + v.label
        seg := segment{x1: v.x, y1: v.y, x2: v.x,
                y2: v.y, l: label, o: "l"}
        segments = append(segments, seg)
}
```

For the other nodes we again pad the label on the left. Then we draw two segments each. Let (v_x, v_y) be the coordinates of the current node, (p_x, p_y) the coordinates of its parent. Then we draw one segment from the parent to the height of v, $(p_x, p_y), (p_x, v_y)$, and one from v to that point, $(v_x, v_y), (p_x, v_y)$. The first segment is not labeled, the second one might be labeled at its starting position.

333c ⟨*Treat other node in rooted tree, Ch. 36* 333c⟩≡ (333a)

```
label := ""
if v.label != "" && !o.NoLabels {
        label = " " + v.label
}
p := v.parent
s1 := segment{x1: p.x, y1: p.y, x2: p.x, y2: v.y}
s2 := segment{x1: v.x, y1: v.y, x2: p.x,
        y2: v.y, l: label, o: "l"}
segments = append(segments, s1)
segments = append(segments, s2)
```

The layout of the unrooted tree is based on the number of leaves in the subtree of each internal node. So we compute this, before setting the node coordinates with `setCoords`. For this we initialize the omega and tau parameters of the root to -1. Then we collect the branches with `collectBranchesU`.

334a ⟨*Layout unrooted tree, Ch. 36* 334a⟩≡ (330d)
```
numLeaves(root)
totalLeaves := root.nl
root.omega = -1.0
root.tau = -1.0
setCoords(root, totalLeaves)
segments = collectBranchesU(root, segments, opts)
```

We declare the node field `nl` to hold the number of leaves in the node's subtree.

334b ⟨*Node fields, Ch. 36* 332b⟩+≡ (329e) ◁332b 335c▷
```
nl int
```

The function `numLeaves` is a depth-first traversal that passes the number of leaves up from child to parent.

334c ⟨*Functions, Ch. 36* 329a⟩+≡ (325) ◁333a 334d▷
```
func numLeaves(v *node) {
        if v == nil { return }
        numLeaves(v.child)
        numLeaves(v.sib)
        if v.child == nil {
                v.nl = 1
        }
        if v.parent != nil {
                v.parent.nl += v.nl
        }
}
```

When setting the node coordinates, we place the current node, unless it is the root, and then place its children [3].

334d ⟨*Functions, Ch. 36* 329a⟩+≡ (325) ◁334c 335e▷
```
func setCoords(v *node, nl int) {
        if v == nil { return }
        if v.parent != nil {
                ⟨Place node, Ch. 36 335a⟩
        }
        ⟨Place children, Ch. 36 335d⟩
        setCoords(v.child, nl)
        setCoords(v.sib, nl)
}
```

We place the node according to the formulae given in [3].

335a ⟨*Place node, Ch. 36* 335a⟩≡ (334d)
```
p := v.parent
l := v.length
if !v.hasLength { l = 1.0 }
v.x = p.x + l *
          (math.Cos(v.tau + v.omega / 2.0))
v.y = p.y + l *
          (math.Sin(v.tau + v.omega / 2.0))
```

We import `math`.

335b ⟨*Imports, Ch. 36* 326b⟩+≡ (325) ◁329b 338c▷
```
"math"
```

We declare the new node fields `tau` and `omega`.

335c ⟨*Node fields, Ch. 36* 332b⟩+≡ (329e) ◁334b
```
tau, omega float64
```

We place the children [3].

335d ⟨*Place children, Ch. 36* 335d⟩≡ (334d)
```
eta := v.tau
w := v.child
for w != nil {
          w.omega = float64(w.nl) / float64(nl) * 2.0 * math.Pi
          w.tau = eta
          eta += w.omega
          w = w.sib
}
```

Each node with a parent corresponds to a segment.

335e ⟨*Functions, Ch. 36* 329a⟩+≡ (325) ◁334d 337b▷
```
func collectBranchesU(v *node, segments []segment,
          o *opts) []segment {
          if v == nil { return segments }
          if v.parent != nil {
                    ⟨Construct segment, Ch. 36 336a⟩
          }
          segments = collectBranchesU(v.child, segments, o)
          segments = collectBranchesU(v.sib, segments, o)
          return segments
}
```

A segment starts at the child and ends at the parent. It has a label and if the child is a leaf, that label should have the same direction as the branch [3]. However, we'd like to avoid labels that are upside down and we also have to pad the label with a blank; so we adjust the angles and labels. Once we've constructed the segment, we store it in the slice of segments.

336a ⟨*Construct segment, Ch. 36* 336a⟩≡ (335e)

```
p := v.parent
a := 0.0
ori := "l"
label := ""
if v.child == nil {
        a = (v.tau + v.omega / 2.0) * 180.0 / math.Pi
}
⟨Adjust angle and label, Ch. 36 336b⟩
seg := segment{x1: v.x, y1: v.y, x2: p.x, y2: p.y,
        l: label, a: a, o: ori}
segments = append(segments, seg)
```

If the label is greater than 90 degrees and less than 270 degrees, we add 180 degrees to it in order to flip it. In that case the label is padded with on or two blanks on the right hand side, depending on the OS. In that case we also set the orientation to *right*, `r`.

336b ⟨*Adjust angle and label, Ch. 36* 336b⟩≡ (336a)

```
if a > 90 && a < 270 {
        a += 180
        ori = "r"
        if !o.NoLabels {
                pad := " "
                if runtime.GOOS == "darwin" { pad += " " }
                label = v.label + pad
        }
} else if !o.NoLabels {
        label = " " + v.label
}
```

The scale is located at the top right hand corner of the tree. To find it, we calculate the plot dimensions.

336c ⟨*Add scale, Ch. 36* 336c⟩≡ (330d)

```
⟨Calculate plot dimensions, Ch. 36 336d⟩
⟨Construct scale, Ch. 36 337c⟩
```

The plot dimensions are calculated using the function `findDim`. It takes as argument a structure holding the maxima and minima of x and y. We initialize the maxima to the smallest number and the minima to the largest number.

336d ⟨*Calculate plot dimensions, Ch. 36* 336d⟩≡ (336c)

```
dim := new(dimension)
dim.xMin = math.MaxFloat64
dim.xMax = -dim.xMin
dim.yMin = dim.xMin
dim.yMax = dim.xMax
findDim(root, dim)
```

We declare a `dimension` to hold the minima and maxima of x and y.

337a ⟨*Types, Ch. 36* 327d⟩+≡ (325) ◁331a
```
type dimension struct {
        xMin, xMax float64
        yMin, yMax float64
}
```

Inside `findDim`, the x and y values passed are compared to that of the current node and updated if necessary.

337b ⟨*Functions, Ch. 36* 329a⟩+≡ (325) ◁335e
```
func findDim(v *node, d *dimension) {
        if v == nil { return }
        if d.xMax < v.x { d.xMax = v.x }
        if d.yMax < v.y { d.yMax = v.y }
        if d.xMin > v.x { d.xMin = v.x }
        if d.yMin > v.y { d.yMin = v.y }
        findDim(v.child, d)
        findDim(v.sib, d)
}
```

The scale consists of a line and a number.

337c ⟨*Construct scale, Ch. 36* 337c⟩≡ (336c)
```
⟨Draw scale line, Ch. 36 337d⟩
⟨Draw scale number, Ch. 36 338b⟩
```

To draw the scale line, we need its length and its coordinates. The scale length is either given by the user or calculated by us.

337d ⟨*Draw scale line, Ch. 36* 337d⟩≡ (337c)
```
scaleLen := opts.Scale
width := dim.xMax - dim.xMin
if scaleLen == 0.0 {
        ⟨Determine scale length, Ch. 36 337e⟩
}
⟨Determine scale coordinates, Ch. 36 338a⟩
s1 := segment{x1: x1, y1: y, x2: x2, y2: y}
segments = append(segments, s1)
```

Let w be the plot width and the offset ℓ the decadic logarithm of w rounded to the nearest integer,

$$\ell = \text{round}(\log_{10}(w)).$$

Then we choose $10^\ell/10$ as the length of the scale. Recall, the user can set a length, but this should be a good starting point.

337e ⟨*Determine scale length, Ch. 36* 337e⟩≡ (337d)
```
y := math.Round(math.Log10(width))
scaleLen = math.Pow(10, y) / 10.0
```

We place it by the margin, m, above the plot height, h, so it starts at $(x_\mathrm{m}, y_\mathrm{m} + h \times m)$.

338a ⟨*Determine scale coordinates, Ch. 36* 338a⟩≡ (337d)
```
x1 := dim.xMax
height := dim.yMax - dim.yMin
y := dim.yMax + height / 10.0
x2 := x1 - scaleLen
```

The label of the scale is placed in its middle. We raise it above the line by 1/20-th of the plot height.

338b ⟨*Draw scale number, Ch. 36* 338b⟩≡ (337c)
```
x := (x1+x2) / 2.0
y += height / 20.0
l := strconv.FormatFloat(scaleLen, 'g', 3, 64)
s1 = segment{x1: x, y1: y, x2: x, y2: y, l: l, o: "c"}
segments = append(segments, s1)
```

We import `strconv`.

338c ⟨*Imports, Ch. 36* 326b⟩+≡ (325) ◁335b 338e▷
```
"strconv"
```

The plot title is the root of the file name plus the counter. If there are no input files, we set the name to *stdin*.

338d ⟨*Construct plot title, Ch. 36* 338d⟩≡ (330c)
```
if opts.Ps != "" {
        opts.Title = ""
} else {
        fn := "stdin"
        if len(files) > *fileCounter {
                fn = files[*fileCounter]
        }
        title := strings.Split(path.Base(fn), ".")[0]
        title += "_" + strconv.Itoa(treeCounter)
        opts.Title = title
}
```

We import `strings` and `path`.

338e ⟨*Imports, Ch. 36* 326b⟩+≡ (325) ◁338c 339b▷
```
"strings"
"path"
```

We add the option field `Title`.

338f ⟨*Opts fields, Ch. 36* 328a⟩+≡ (327d) ◁328a
```
Title string
```

The output stream is either the standard input stream of the `gnuplot` command or the script file requested by the user.

339a ⟨*Construct output stream, Ch. 36* 339a⟩≡ (330c)
```
var wr io.WriteCloser
var gcmd *exec.Cmd
var err error
if opts.Script == "" {
          gcmd = exec.Command("gnuplot")
          wr, err = gcmd.StdinPipe()
          if err != nil { log.Fatal(err) }
} else {
          wr, err = os.Create(opts.Script)
          if err != nil { log.Fatal(err) }
}
```

We import `exec`, `os`, and `log`.

339b ⟨*Imports, Ch. 36* 326b⟩+≡ (325) ◁338e 340b▷
```
"os/exec"
"os"
"log"
```

We write the segments to an the output stream in a Go routine, where we also close the output stream again. Unless the user opted for the script, we run `gnuplot`.

339c ⟨*Write segments to output stream, Ch. 36* 339c⟩≡ (330c)
```
done := make(chan struct{})
go func() {
          ⟨Write gnuplot code to output stream, Ch. 36 339d⟩
          ⟨Close output stream, Ch. 36 342b⟩
          done <- struct{}{}
}()
if opts.Script == "" {
          ⟨Run gnuplot, Ch. 36 342c⟩
}
<-done
```

We write the `gnuplot` code in seven steps: We write the terminal, add the code passed by the user, remove the axes, write the labels, write the plot, its segments, and its margins.

339d ⟨*Write gnuplot code to output stream, Ch. 36* 339d⟩≡ (339c)
```
⟨Write terminal, Ch. 36 340a⟩
⟨Write gnuplot code, Ch. 36 340e⟩
⟨Remove axes, Ch. 36 340f⟩
⟨Write labels, Ch. 36 341a⟩
⟨Write plot, Ch. 36 341b⟩
⟨Write segments, Ch. 36 341d⟩
⟨Write margins, Ch. 36 341e⟩
```

The terminal is either encapsulated postscript or window. If the window is interactive, we make it persistent. We also set the plot dimensions.

340a ⟨*Write terminal, Ch. 36* 340a⟩≡ (339d) 340c ▷
```
t := "set terminal"
if opts.Ps != "" {
        t += " postscript eps monochrome"
} else {
        t += " " + opts.Win
}
if util.IsInteractive(opts.Win) && opts.Ps == "" {
        t += " persist"
}
t += " size " + opts.Dim
fmt.Fprintf(wr, "%s\n", t)
```

We import `fmt`.

340b ⟨*Imports, Ch. 36* 326b⟩+≡ (325) ◁339b
```
"fmt"
```

`gnuplot` version 5.4 patch level 3 generates screen plots with red background—at least on macOS. We make sure our plots are white.

340c ⟨*Write terminal, Ch. 36* 340a⟩+≡ (339d) ◁340a 340d ▷
```
if util.IsInteractive(opts.Win) && opts.Ps == "" {
        c := "set object 1 rectangle from screen 0,0 " +
                "to screen 1,1 fillcolor rgb 'white' behind"
        fmt.Fprintf(wr, "%s\n", c)
}
```

We also set the postscript file.

340d ⟨*Write terminal, Ch. 36* 340a⟩+≡ (339d) ◁340c
```
if opts.Ps != "" {
        fmt.Fprintf(wr, "set output \"%s\"\n", opts.Ps)
}
```

If the user passed `gnuplot` code, we print it, surrounded by comments.

340e ⟨*Write* `gnuplot` *code, Ch. 36* 340e⟩≡ (339d)
```
if opts.Code != "" {
        fmt.Fprintf(wr, "# Start of external code\n")
        fmt.Fprintf(wr, "%s\n", opts.Code)
        fmt.Fprintf(wr, "# End of external code\n")
}
```

To remove the plot axes, we remove the tics of the x- and y-axes, and remove the border.

340f ⟨*Remove axes, Ch. 36* 340f⟩≡ (339d)
```
fmt.Fprintf(wr, "unset xtics\n")
fmt.Fprintf(wr, "unset ytics\n")
fmt.Fprintf(wr, "unset border\n")
```

We iterate over the segments and write any labels they might contain.

341a ⟨*Write labels, Ch. 36* 341a⟩≡ (339d)
```
t = "set label \"%s\" %s rotate by %d at %.4g,%.4g front\n"
for _, s := range segments {
        if s.l != "" {
                a := int(math.Round(s.a))
                fmt.Fprintf(wr, t, s.l,
                        s.o, a, s.x1, s.y1)
        }
}
```

Our plot is a line plot in black that might have a title.

341b ⟨*Write plot, Ch. 36* 341b⟩≡ (339d) 341c ▷
```
if opts.Title != "" {
        fmt.Fprintf(wr, "set title \"%s\"\n",
                opts.Title)
}
fmt.Fprintf(wr, "plot \"-\" t \"\" w l lc \"black\"")
```

For postscript output we set the line width to 3 before we terminate the plot command with a newline.

341c ⟨*Write plot, Ch. 36* 341b⟩+≡ (339d) ◁ 341b
```
if opts.Ps != "" {
        fmt.Fprintf(wr, " lw 3")
}
fmt.Fprintf(wr, "\n")
```

Segments are pairs of points set off by a blank line.

341d ⟨*Write segments, Ch. 36* 341d⟩≡ (339d)
```
for i, s := range segments {
        if i > 0 {
                fmt.Fprintf(wr, "\n")
        }
        fmt.Fprintf(wr, "%.4g %.4g\n%.4g %.4g\n",
                s.x1, s.y1, s.x2, s.y2)
}
```

We write the margins for the leaf labels. All trees have labels on the right hand side, so we place an extra dot there. In addition, unrooted trees require margins on the other three sides.

341e ⟨*Write margins, Ch. 36* 341e⟩≡ (339d)
```
xOffset := width * opts.Margin
x := dim.xMax + xOffset
fmt.Fprintf(wr, "\n%.4g 0\n", x)
if !rooted {
        ⟨Add margins to unrooted tree, Ch. 36 342a⟩
}
```

We add margins to the top, bottom, and left.

342a ⟨*Add margins to unrooted tree, Ch. 36* 342a⟩≡ (341e)
```
yOffset := height * opts.Margin
y := height + yOffset
fmt.Fprintf(wr, "\n0 %.4g\n", y)
y = dim.yMin - yOffset
fmt.Fprintf(wr, "\n0 %.4g\n", y)
x = dim.xMin - xOffset
fmt.Fprintf(wr, "\n%.4g 0\n", x)
```

Having written all `gnuplot` instructions to the output stream, we close it again.

342b ⟨*Close output stream, Ch. 36* 342b⟩≡ (339c)
```
wr.Close()
```

We run the `gnuplot` command, check for errors, and print the output, if any.

342c ⟨*Run gnuplot, Ch. 36* 342c⟩≡ (339c)
```
out, err := gcmd.Output()
util.CheckGnuplot(err)
if len(out) > 0 {
        fmt.Printf("%s", out)
}
```

There is still the file counter to increment.

342d ⟨*Increment file counter, Ch. 36* 342d⟩≡ (329a)
```
*fileCounter++
```

We've finished `plotTree`, time to test it.

Testing

The outline of our testing program contains hooks for imports and the testing logic.

342e ⟨*plotTree_test.go* 342e⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 36 343b⟩
)

func TestPlotTree(t *testing.T) {
        ⟨Testing, Ch. 36 343a⟩
}
```

We construct a set of tests. In each test we compare the `gnuplot` output we get with the precomputed `gnuplot` output we want. The `gnuplot` output of the test runs is written to a unique temporary file that we delete after we have iterated over the tests.

343a ⟨*Testing, Ch. 36* 343a⟩≡ (342e)
```
var tests []*exec.Cmd
gf, err := ioutil.TempFile(".", "tmp_*.gp")
if err != nil { t.Error("can't open temp file") }
g := gf.Name()
⟨Construct tests, Ch. 36 343c⟩
for i, test := range tests {
        ⟨Run test, Ch. 36 343d⟩
}
err = os.Remove(g)
if err != nil { t.Errorf("can't remove %q", g) }
```

We import `exec`, `ioutil`, and `os`.

343b ⟨*Testing imports, Ch. 36* 343b⟩≡ (342e) 344a▷
```
"os/exec"
"io/ioutil"
"os"
```

Our tests run on the Newick tree shown in Figure 36.1A, which is stored in `newick.nwk`. We draw rooted and unrooted versions of it, with and without node labels.

343c ⟨*Construct tests, Ch. 36* 343c⟩≡ (343a)
```
f := "newick.nwk"
test := exec.Command("./plotTree", "-r", "-s", g, f)
tests = append(tests, test)
test = exec.Command("./plotTree", "-u", "-s", g, f)
tests = append(tests, test)
test = exec.Command("./plotTree", "-r", "-s", g, "-n", f)
tests = append(tests, test)
test = exec.Command("./plotTree", "-u", "-s", g, "-n", f)
tests = append(tests, test)
test = exec.Command("./plotTree", "-t", "dumb", "-s", g, f)
tests = append(tests, test)
```

When we run a test, we compare the result we get to the result we want, which is stored in files with names we construct next.

343d ⟨*Run test, Ch. 36* 343d⟩≡ (343a)
```
err := test.Run()
if err != nil { t.Errorf("couldn't run %q", test) }
get, err := ioutil.ReadFile(g)
if err != nil { t.Errorf("couldn't open %q", g) }
⟨Construct file name, Ch. 36 344b⟩
want , err := ioutil.ReadFile(f)
if err != nil { t.Errorf("couldn't open %q", f) }
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `bytes`.

344a ⟨*Testing imports, Ch. 36* 343b⟩+≡ (342e) ◁343b 344c▷
```
"bytes"
```

By default, the results are stored in files called `r1.gp`, `r2.gp`, and so on, inside the directory `results`. On darwin systems, these names are extended by "d".

344b ⟨*Construct file name, Ch. 36* 344b⟩≡ (343d)
```
f := "results/r" + strconv.Itoa(i+1)
if runtime.GOOS == "darwin" {
          f += "d"
}
f += ".gp"
```

We import `strconv` and `runtime`.

344c ⟨*Testing imports, Ch. 36* 343b⟩+≡ (342e) ◁344a
```
"strconv"
"runtime"
```

Chapter 37

Program `pps`: Print Polymorphic Sites

A	B
	>Positions (2)
>S1	2 3
AC-GT	>S1 - polymorphic
>S2	C-
AGGGT	>S2 - polymorphic
	GG

Figure 37.1: An alignment of two sequences (**A**) is transformed by `pps` to its polymorphic sites (**B**).

Introduction

The program `pps` prints polymorphic sites. It reads one or more sets of aligned sequences and for each set prints the polymorphic positions and residues in the alignment. For example, Figure 37.1A shows the alignment of two DNA sequences, the first is four bp long, the second five. `pps` turns this into the three FASTA entries shown in Figure 37.1B. The first entry lists the two polymorphic positions, the next two the polymorphic residues in the two sequences.

There are two types of polymorphisms, gaps and mutations. By default, `pps` extracts them all, but the user can exclude gaps. The user can also opt to code positions that agree with the top row of the alignment as dots.

Implementation

The outline of our implementation of `pps` contains hooks for imports, functions, and the logic of the main function.

346a ⟨*pps.go* 346a⟩≡

```
package main

import (
        ⟨Imports, Ch. 37 347a⟩
)
⟨Functions, Ch. 37 347h⟩
func main() {
        ⟨Main function, Ch. 37 346b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

346b ⟨*Main function, Ch. 37* 346b⟩≡ (346a)

```
util.PrepLog("pps")
⟨Set usage, Ch. 37 347b⟩
⟨Declare options, Ch. 37 347d⟩
⟨Parse options, Ch. 37 347f⟩
⟨Parse input files, Ch. 37 347g⟩
```

We import `util`.

347a ⟨*Imports, Ch. 37* 347a⟩≡ (346a) 347c ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `pps`, and an example command.

347b ⟨*Set usage, Ch. 37* 347b⟩≡ (346b)
```
u := "pps [-h] [option]... [foo.fasta]..."
p := "Extract polymorphic sites from alignment."
e := "pps foo.fasta | getSeq -c Pos"
clio.Usage(u, p, e)
```

We import `clio`.

347c ⟨*Imports, Ch. 37* 347a⟩+≡ (346a) ◁347a 347e ▷
```
"github.com/evolbioinf/clio"
```

There are four options, the version, `-v`, the line length, `-l`, whether we exclude gaps (`-g`), and whether to dot matching sites (`d`).

347d ⟨*Declare options, Ch. 37* 347d⟩≡ (346b)
```
var optV = flag.Bool("v", false, "version")
var optL = flag.Int ("l", fasta.DefaultLineLength,
          "line length")
var optG = flag.Bool("g", false, "exclude gaps")
var optD = flag.Bool("d", false, "dot matches with top row")
```

We import `flag` and `fasta`.

347e ⟨*Imports, Ch. 37* 347a⟩+≡ (346a) ◁347c 348a ▷
```
"flag"
"github.com/evolbioinf/fasta"
```

We parse the options and respond to `-v` as this ends `pps`.

347f ⟨*Parse options, Ch. 37* 347f⟩≡ (346b)
```
flag.Parse()
if *optV {
          util.PrintInfo("pps")
}
```

The remaining tokens on the command line are taken as the names of input files. Each of these is parsed with the function `scan`, which takes as argument the line length, whether we exclude gaps, and whether to dot matches.

347g ⟨*Parse input files, Ch. 37* 347g⟩≡ (346b)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optL, *optG, *optD)
```

Inside `scan`, we retrieve the arguments just passed, collect all sequences in the stream, find the polymorphic sites, and print them.

347h ⟨*Functions, Ch. 37* 347h⟩≡ (346a)
```
func scan(r io.Reader, args ...interface{}) {
          ⟨Retrieve arguments, Ch. 37 348b⟩
          ⟨Collect sequences, Ch. 37 348c⟩
          ⟨Find polymorphic sites, Ch. 37 348f⟩
          ⟨Print polymorphic sites, Ch. 37 349c⟩
}
```

We import `io`.

348a ⟨*Imports, Ch. 37* 347a⟩+≡ (346a) ◁347e 348e▷
```
"io"
```

We retrieve the line length, whether to exclude gaps, and whether to dot matches.

348b ⟨*Retrieve arguments, Ch. 37* 348b⟩≡ (347h)
```
ll := args[0].(int)
exclGaps := args[1].(bool)
dot := args[2].(bool)
```

We iterate over the sequences using a scanner and collect them into a slice of sequences, the alignment. Once collected, we check the alignment.

348c ⟨*Collect sequences, Ch. 37* 348c⟩≡ (347h)
```
al := make([]*fasta.Sequence, 0)
sc := fasta.NewScanner(r)
for sc.ScanSequence() {
        al = append(al, sc.Sequence())
}
⟨Check alignment, Ch. 37 348d⟩
```

If the sequences in the alignment don't all have the same length, they cannot be aligned and we bail.

348d ⟨*Check alignment, Ch. 37* 348d⟩≡ (348c)
```
for i, _ := range al {
        if i > 0 {
                l1 := len(al[i].Data())
                l2 := len(al[i-1].Data())
                if l1 != l2 {
                        log.Fatal("sequences not aligned")
                }
        }
}
```

We import `log`.

348e ⟨*Imports, Ch. 37* 347a⟩+≡ (346a) ◁348a 350a▷
```
"log"
```

An alignment is an $m \times n$ matrix of residues. We store its dimensions and allocate space for the polymorphic sites. Then we identify all polymorphisms and remove the gaps, if desired.

348f ⟨*Find polymorphic sites, Ch. 37* 348f⟩≡ (347h)
```
m := len(al)
n := len(al[0].Data())
ps := make([]int, 0)
⟨Find all polymorphisms, Ch. 37 349a⟩
if exclGaps {
        ⟨Remove gaps, Ch. 37 349b⟩
}
```

We go through the n columns of the alignment and compare its residues to the residue in the top row. If they differ, we've found a polymorphic site and move on to the next column.

349a ⟨*Find all polymorphisms, Ch. 37* 349a⟩≡ (348f)
```
for i := 0; i < n; i++ {
        c1 := al[0].Data()[i]
        for j := 1; j < m; j++ {
                c2 := al[j].Data()[i]
                if c1 != c2 {
                        ps = append(ps, i)
                        break
                }
        }
}
```

We iterate over the polymorphic sites and remove those that contain a gap.

349b ⟨*Remove gaps, Ch. 37* 349b⟩≡ (348f)
```
var k, j int
for _, p := range ps {
        for k = 0; k < m; k++ {
                if al[k].Data()[p] == '-' { break }
        }
        if k == m {
                ps[j] = p
                j++
        }
}
ps = ps[:j]
```

We print the positions and convert matches to dots if desired. Then we iterate over the sequences in the alignment print their residues. For both the positions and the residues we print the header followed by the remainder.

349c ⟨*Print polymorphic sites, Ch. 37* 349c⟩≡ (347h)
```
⟨Print header of positions, Ch. 37 349d⟩
⟨Print positions, Ch. 37 350b⟩
if dot {
        ⟨Convert matches to dots, Ch. 37 350d⟩
}
for _, s := range al {
        ⟨Print header of residues, Ch. 37 350e⟩
        ⟨Print residues, Ch. 37 350f⟩
}
```

The header of the positions contains the number of polymorphic sites. We distinguish singular and plural.

349d ⟨*Print header of positions, Ch. 37* 349d⟩≡ (349c)
```
fmt.Printf(">Position")
n = len(ps)
if n != 1 { fmt.Printf("s") }
fmt.Printf(" (%d)\n", n)
```

We import `fmt`.

350a ⟨*Imports, Ch. 37* 347a⟩+≡ (346a) ◁348e 350c ▷
```
"fmt"
```

We print the positions one line at a time. The positions are one-based and we align them in columns using a tabwriter.

350b ⟨*Print positions, Ch. 37* 350b⟩≡ (349c)
```
w := tabwriter.NewWriter(os.Stdout, 1, 0, 1, ' ', 0)
for i := 0; i < n; i += ll {
        for j := 0; i+j < n && j < ll; j++ {
                if j > 0 { fmt.Fprintf(w, "\t") }
                fmt.Fprintf(w, "%d", ps[i+j] + 1)
        }
        fmt.Fprintf(w, "\n")
}
w.Flush()
```

We import `tabwriter` and `os`.

350c ⟨*Imports, Ch. 37* 347a⟩+≡ (346a) ◁350a
```
"text/tabwriter"
"os"
```

We convert matches to the top row to dots.

350d ⟨*Convert matches to dots, Ch. 37* 350d⟩≡ (349c)
```
d1 := al[0].Data()
for i := 1; i < len(al); i++ {
        dx := al[i].Data()
        for j := 0; j < len(dx); j++ {
                if dx[j] == d1[j] { dx[j] = byte('.') }
        }
}
```

We print the header of the residues with a reminder that these are just the polymorphic sites.

350e ⟨*Print header of residues, Ch. 37* 350e⟩≡ (349c)
```
fmt.Printf(">%s - polymorphic\n", s.Header())
```

Like the positions, we print the residues one line at a time.

350f ⟨*Print residues, Ch. 37* 350f⟩≡ (349c)
```
d := s.Data()
for i := 0; i < n; i += ll {
        for j := 0; i+j < n && j < ll; j++ {
                p := ps[i+j]
                fmt.Printf("%c", d[p])
        }
        fmt.Printf("\n")
}
```

We've finished `pps`, let's test it.

Testing

The outline of our testing code contains hooks for imports and the testing logic.

351a ⟨*pps_test.go* 351a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 37 351c⟩
)
func TestPps(t *testing.T) {
        ⟨Testing, Ch. 37 351b⟩
}
```

We construct a set of tests and then iterate over them.

351b ⟨*Testing, Ch. 37* 351b⟩≡ (351a)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 37 351d⟩
for i, test := range tests {
        ⟨Run test, Ch. 37 351e⟩
}
```

We import `exec`.

351c ⟨*Testing imports, Ch. 37* 351c⟩≡ (351a) 352▷
```
"os/exec"
```

We construct four tests, with gaps, without, with custom line length, and with dots. The input is always the file `hom.fasta`, which contains an aligned region of *Hominidae* mitochondrial genomes.

351d ⟨*Construct tests, Ch. 37* 351d⟩≡ (351b)
```
f := "hom.fasta"
test := exec.Command("./pps", f)
tests = append(tests, test)
test = exec.Command("./pps", "-g", f)
tests = append(tests, test)
test = exec.Command("./pps", "-l", "20", f)
tests = append(tests, test)
test = exec.Command("./pps", "-d", f)
tests = append(tests, test)
```

When we run a test, we compare the output we get with the output we want, which is contained in files `r1.fasta`, `r2.fasta`, and `r3.fasta`.

351e ⟨*Run test, Ch. 37* 351e⟩≡ (351b)
```
get, err := test.Output()
if err != nil { t.Errorf("can't run %s", test) }
f = "r" + strconv.Itoa(i+1) + ".fasta"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("can't open %q", f) }
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

352 ⟨*Testing imports, Ch. 37* 351c⟩+≡ (351a) ◁351c

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 38

Program `randomizeSeq`: Shuffle DNA Sequence

Introduction

We often compare the properties of a given sequence with those of its shuffled version. The program `randomizeSeq` carries out this shuffling.

Implementation

The program outline contains hooks for imports, variable, functions, and the logic of the main function.

354a ⟨*randomizeSeq.go* 354a⟩≡
```
package main

import (
        ⟨Imports, Ch. 38 354c⟩
)
⟨Variables, Ch. 38 355b⟩
⟨Functions, Ch. 38 355e⟩
func main() {
        ⟨Main function, Ch. 38 354b⟩
}
```

In the main function, we prepare the `log` package, set the usage, and parse the options and the input.

354b ⟨*Main function, Ch. 38* 354b⟩≡ (354a)
```
util.PrepLog("randomizeSeq")
⟨Set usage, Ch. 38 354d⟩
⟨Parse options, Ch. 38 354f⟩
⟨Parse input, Ch. 38 355c⟩
```

We import `util`.

354c ⟨*Imports, Ch. 38* 354c⟩≡ (354a) 354e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage has three parts, the actual usage statement, a description, and an example.

354d ⟨*Set usage, Ch. 38* 354d⟩≡ (354b)
```
u := "randomizeSeq [-h] [options] [files]"
p := "Shuffle sequences."
e := "randomizeSeq *.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

354e ⟨*Imports, Ch. 38* 354c⟩+≡ (354a) ◁ 354c 355a ▷
```
"github.com/evolbioinf/clio"
```

We parse the options and check whether the version is to be printed.

354f ⟨*Parse options, Ch. 38* 354f⟩≡ (354b)
```
flag.Parse()
if *optV {
        util.PrintInfo("randomizeSeq")
}
```

We import `flag`.

355a ⟨*Imports, Ch. 38* 354c⟩+≡ (354a) ◁354e 355d▷
```
"flag"
```

and declare the options. Apart from version (`-v`), the user can seed the random number generator (`-s`). By default, the seed is generated internally.

355b ⟨*Variables, Ch. 38* 355b⟩≡ (354a)
```
var optV = flag.Bool("v", false, "version")
var optS = flag.Int("s", 0, "seed for random number generator; " +
          "default: internal")
```

The input is parsed with the function `ParseFiles`. It takes as arguments the names of the files to be parsed, the function `scan` applied to each file, and its argument, a pointer to the random number generator.

355c ⟨*Parse input, Ch. 38* 355c⟩≡ (354b)
```
var rn *rand.Rand
if *optS != 0 {
          rn = rand.New(rand.NewSource(int64(*optS)))
} else {
          t := time.Now().UnixNano()
          rn = rand.New(rand.NewSource(t))
}
files := flag.Args()
clio.ParseFiles(files, scan, rn)
```

We import `rand` and `time`.

355d ⟨*Imports, Ch. 38* 354c⟩+≡ (354a) ◁355a 355f▷
```
"math/rand"
"time"
```

In the function `scan` we retrieve the random number generator, and print a shuffled version of each sequence. We append `SHUFFLED` to the header to notify the user.

355e ⟨*Functions, Ch. 38* 355e⟩≡ (354a)
```
func scan(r io.Reader, args ...interface{}) {
          rn := args[0].(*rand.Rand)
          sc := fasta.NewScanner(r)
          for sc.ScanSequence() {
                  seq := sc.Sequence()
                  seq.Shuffle(rn)
                  seq.AppendToHeader(" - SHUFFLED")
                  fmt.Println(seq)
          }
}
```

We import `io`, `fasta`, and `fmt`.

355f ⟨*Imports, Ch. 38* 354c⟩+≡ (354a) ◁355d
```
"io"
"github.com/evolbioinf/fasta"
"fmt"
```

This concludes our implementation `randomizeSeq`, time to test it.

Testing

We begin with an outline containing hooks for imports and the testing logic.

356a ⟨*randomizeSeq_test.go* 356a⟩≡

```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 38 356c⟩
)

func TestRandomizeSeq (t *testing.T) {
        ⟨Testing, Ch. 38 356b⟩
}
```

If we run `randomizeSeq` with a given seed, we can pre-compute its result, and thus know that `test.fasta` becomes `shuf.fasta`. So we can compare what we get when we shuffle `test.fasta` with what we want.

356b ⟨*Testing, Ch. 38* 356b⟩≡ (356a)

```
cmd := exec.Command("./randomizeSeq", "-s", "13", "test.fasta")
g, err := cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err := ioutil.ReadFile("shuf.fasta")
if err != nil {
        t.Errorf("couldn't open file %q\n", "shuf.fasta")
}
if !bytes.Equal(g, w) {
        t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

We import `exec`, `ioutil`, and `bytes`.

356c ⟨*Testing imports, Ch. 38* 356c⟩≡ (356a)

```
"os/exec"
"io/ioutil"
"bytes"
```

Chapter 39

Program `ranDot`: Random Graph in dot Notation

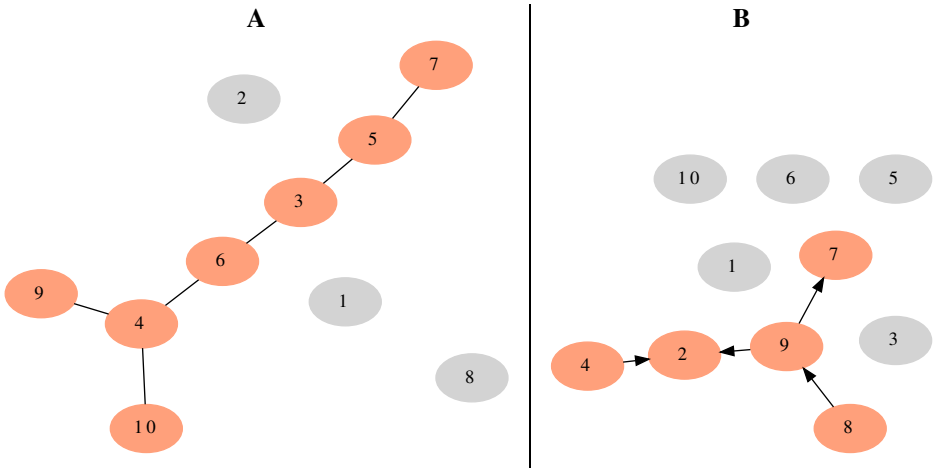


Figure 39.1: Two random graphs; undirected (**A**) and directed (**B**).

Introduction

When studying graphs, it's convenient to have a ready source of graph data. The program `ranDot` generates random graphs in dot notation, which can then be rendered by the programs of the free package GraphViz[1]. Graphs may be undirected (Figure 39.1A) or directed (Figure 39.1B).

The user of `ranDot` can set the number of nodes, the probability of an edge between pairs of nodes, and the color of nodes, depending on whether they are connected or not. For example, in Figures 39.1A and B the connected nodes are shown in light salmon, unconnected—or singleton—nodes in light gray. The colors and their names are listed on the GraphViz web site:

`www.graphviz.org/doc/info/colors.html`

Implementation

The outline of `ranDot` has hooks for imports and the main function.

358 ⟨*ranDot.go* 358⟩≡

```
package main

import (
        ⟨Imports, Ch. 39 359b⟩
)

func main() {
        ⟨Main function, Ch. 39 359a⟩
}
```

[1] `graphviz.org`

In the main function we prepare the `log` package, set the usage, declare and parse the options, construct the graph, and print it.

359a ⟨*Main function, Ch. 39* 359a⟩≡ (358)
```
util.PrepLog("ranDot")
⟨Set usage, Ch. 39 359c⟩
⟨Declare options, Ch. 39 359e⟩
⟨Parse options, Ch. 39 360b⟩
⟨Construct graph, Ch. 39 360f⟩
⟨Print graph, Ch. 39 361b⟩
```

We import `util`.

359b ⟨*Imports, Ch. 39* 359b⟩≡ (358) 359d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `ranDot`, and an example command.

359c ⟨*Set usage, Ch. 39* 359c⟩≡ (359a)
```
u := "ranDot [-h] [option]..."
p := "Draw random graph in dot notation."
e := "ranDot -c lightsalmon -C lightgray -d"
clio.Usage(u, p, e)
```

We import `clio`.

359d ⟨*Imports, Ch. 39* 359b⟩+≡ (358) ◁359b 360a▷
```
"github.com/evolbioinf/clio"
```

We declare eight options:

1. `-n` size of graph
2. `-p` edge probability
3. `-d` directed edges
4. `-S` allow edge to self
5. `-c` color of connected nodes
6. `-C` color of singletons
7. `-s` seed of random number generator
8. `-v`

359e ⟨*Declare options, Ch. 39* 359e⟩≡ (359a)
```
var optN = flag.Int("n", 10, "number of nodes")
var optP = flag.Float64("p", 0.05, "edge probability")
var optD = flag.Bool("d", false, "directed edges")
var optSS = flag.Bool("S", false, "allow edge to self")
var optC = flag.String("c", "", "color of connected nodes")
var optCC = flag.String("C", "", "color of singleton nodes; " +
          "color names: www.graphviz.org/doc/info/colors.html")
var optS = flag.Int64("s", 0, "seed for random number generator " +
          "(default internal)")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

360a ⟨*Imports, Ch. 39* 359b⟩+≡ (358) ◁359d 360e▷
```
"flag"
```

We parse the options and respond to the version (`-v`) and the seed (`-s`).

360b ⟨*Parse options, Ch. 39* 360b⟩≡ (359a)
```
flag.Parse()
⟨Respond to -v, Ch. 39 360c⟩
⟨Respond to -s, Ch. 39 360d⟩
```

If requested, we print the version.

360c ⟨*Respond to -v, Ch. 39* 360c⟩≡ (360b)
```
if *optV {
        util.PrintInfo("ranDot")
}
```

If the user didn't set a seed, we take the number of nanoseconds since the beginning of the UNIX epoch. Then we seed the random number generator.

360d ⟨*Respond to -s, Ch. 39* 360d⟩≡ (360b)
```
seed := *optS
if seed == 0 {
        seed = time.Now().UnixNano()
}
source := rand.NewSource(seed)
r := rand.New(source)
```

We import `time` and `rand`.

360e ⟨*Imports, Ch. 39* 359b⟩+≡ (358) ◁360a 361d▷
```
"time"
"math/rand"
```

A graph consists of nodes and edges, which we construct in two separate steps.

360f ⟨*Construct graph, Ch. 39* 360f⟩≡ (359a)
```
⟨Construct nodes, Ch. 39 360g⟩
⟨Construct edges, Ch. 39 360h⟩
```

We store the n nodes as an integer slice, where the value is the name of the node, $1, 2, ..., n$.

360g ⟨*Construct nodes, Ch. 39* 360g⟩≡ (360f)
```
n := *optN
nodes := make([]int, n)
for i := 0; i < n; i++ {
        nodes[i] = i + 1
}
```

The edges are represented by an $n \times n$ matrix of boolean variables. If a cell, $m_{i,j}$, is true, there is an edge $v_i - v_j$. After its construction, we fill the matrix.

360h ⟨*Construct edges, Ch. 39* 360h⟩≡ (360f)
```
edges := make([][]bool, n)
for i := 0; i < n; i++ {
        edges[i] = make([]bool, n)
}
⟨Fill edge matrix, Ch. 39 361a⟩
```

We go through the edge matrix and draw a random number for each cell. If the random number is less or equal to the probability of an edge, we set the cell to true. Self-referential edges are on the main diagonal. They are a special case, we set them only if asked to do so.

361a ⟨*Fill edge matrix, Ch. 39* 361a⟩≡ (360h)
```
for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
                if i == j && !*optSS { continue }
                if r.Float64() <= *optP {
                        edges[i][j] = true
                }
        }
}
```

The graph is printed in three portions, header, body, and footer.

361b ⟨*Print graph, Ch. 39* 361b⟩≡ (359a)
```
⟨Print header, Ch. 39 361c⟩
⟨Print body, Ch. 39 361e⟩
⟨Print footer, Ch. 39 363a⟩
```

In the header, we say in a comment that the graph was produced with `ranDot` and how to render it. Then we open it.

361c ⟨*Print header, Ch. 39* 361c⟩≡ (361b)
```
fmt.Println("# Graph written by ranDot.")
fmt.Println("# Render: dot|neato|circo foo.dot")
fmt.Println("graph G {")
```

We import `fmt`.

361d ⟨*Imports, Ch. 39* 359b⟩+≡ (358) ◁360e
```
"fmt"
```

We allow the user to color-code connected nodes and singletons. So we print them separately, first the singletons, then the connected nodes. If the singletons are colored, but the connected nodes aren't, we reset the style so that the connected nodes are drawn in the default style.

361e ⟨*Print body, Ch. 39* 361e⟩≡ (361b)
```
if *optCC != "" {
        fmt.Printf("node [style=filled, color=%s]\n", *optCC)
}
⟨Print singleton nodes, Ch. 39 362a⟩
if *optCC != "" && *optC == "" {
        fmt.Printf("node [style=\"\", color=\"\"]\n")
}
if *optC != "" {
        fmt.Printf("node [style=filled, color=%s]\n", *optC)
}
⟨Print connected nodes, Ch. 39 362c⟩
```

We reserve space for the singletons and mark them. Then we print them.

362a ⟨*Print singleton nodes, Ch. 39* 362a⟩≡ (361e)
```
singletons := make([]bool, n)
⟨Mark singletons, Ch. 39 362b⟩
for i := 0; i < n; i++ {
        if singletons[i] {
                fmt.Printf("\t%d\n", nodes[i])
        }
}
```

To mark the singletons, we set start by setting every node to singleton status. Then we traverse the edge matrix and set connected by an edge to false.

362b ⟨*Mark singletons, Ch. 39* 362b⟩≡ (362a)
```
for i := 0; i < n; i++ { singletons[i] = true }
for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
                if edges[i][j] {
                        singletons[i] = false
                        singletons[j] = false
                }
        }
}
```

We visit every entry in the edge matrix and write a node pair whenever we find an edge. Once we've written an edge, we delete it.

362c ⟨*Print connected nodes, Ch. 39* 362c⟩≡ (361e)
```
for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
                if  edges[i][j] {
                        ⟨Write node pair, Ch. 39 362d⟩
                        edges[i][j] = false
                }
        }
}
```

The edge connecting a node pair may be directed, depending on `-d`. If directed, it may be reciprocal or just forward. For reciprocal edges re delete the partner to avoid duplications.

362d ⟨*Write node pair, Ch. 39* 362d⟩≡ (362c)
```
fmt.Printf("\t%d -- %d", nodes[i], nodes[j])
if *optD {
        fmt.Printf("[dir=")
        if edges[j][i] {
                fmt.Printf("both]")
                edges[j][i] = false
        } else {
                fmt.Printf("forward]")
        }
}
fmt.Printf("\n")
```

The footer simply closes the curly bracket opened at the beginning of the graph.

363a ⟨*Print footer, Ch. 39* 363a⟩≡ (361b)
```
fmt.Println("}")
```

We are finished writing `ranDot`, now comes the test.

Testing

Our testing program has hooks for imports and the testing logic.

363b ⟨*ranDot_test.go* 363b⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 39 364a⟩
)

func TestRanDot(t *testing.T) {
        ⟨Testing, Ch. 39 363c⟩
}
```

We test `ranDot` in twp steps. First, we construct the tests, then we loop over them and run them.

363c ⟨*Testing, Ch. 39* 363c⟩≡ (363b)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 39 363d⟩
for i, test := range tests {
        ⟨Run test, Ch. 39 364d⟩
}
```

We first test the options with arguments, then those without.

363d ⟨*Construct tests, Ch. 39* 363d⟩≡ (363c)
```
⟨Test options with arguments, Ch. 39 363e⟩
⟨Test options without arguments, Ch. 39 364c⟩
```

There are five options with arguments. We just concatenate them, starting with the seed for the random number generator. This means each test runs with a seed, which makes it reproducible. Whenever we have constructed the next argument list, we construct the corresponding test and add it to the list of tests.

363e ⟨*Test options with arguments, Ch. 39* 363e⟩≡ (363d)
```
var test *exec.Cmd
args := []string{"-s", "13"}
⟨Add test, Ch. 39 364b⟩
args = append(args, "-C", "lightgray")
⟨Add test, Ch. 39 364b⟩
args = append(args, "-c", "lightsalmon")
⟨Add test, Ch. 39 364b⟩
args = append(args, "-n", "11")
⟨Add test, Ch. 39 364b⟩
args = append(args, "-p", "0.5")
⟨Add test, Ch. 39 364b⟩
```

We import `exec`.

364a ⟨*Testing imports, Ch. 39* 364a⟩≡ (363b) 364e ▷
```
"os/exec"
```

We add a new test to the list of tests.

364b ⟨*Add test, Ch. 39* 364b⟩≡ (363e 364c)
```
test = exec.Command("./ranDot", args...)
tests = append(tests, test)
```

We test the two options without arguments.

364c ⟨*Test options without arguments, Ch. 39* 364c⟩≡ (363d)
```
args = append(args, "-S")
⟨Add test, Ch. 39 364b⟩
args = append(args, "-d")
⟨Add test, Ch. 39 364b⟩
```

When we run a test, we check we get what we want, which is saved in numbered files `r1.dot`, `r2.dot`, and so on.

364d ⟨*Run test, Ch. 39* 364d⟩≡ (363c)
```
get, err := test.Output()
if err != nil { t.Error(err.Error()) }
f := "r" + strconv.Itoa(i+1) + ".dot"
want, err := ioutil.ReadFile(f)
if err != nil { t.Error(err.Error()) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

364e ⟨*Testing imports, Ch. 39* 364a⟩+≡ (363b) ◁ 364a
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 40

Program `ranseq`: Random DNA Sequence

Introduction

We often need a bit of random DNA sequence as input to other programs. The program `ranseq` generates such random sequences. The user can set their length, number, and G/C content.

Implementation

The outline provides hooks for imports, variables, and the logic of the main function.

366a ⟨*ranseq.go* 366a⟩≡
```
package main

import (
        ⟨Imports, Ch. 40 366c⟩
)

⟨Variables, Ch. 40 367c⟩

func main() {
        ⟨Main function, Ch. 40 366b⟩
}
```

In the main function, we prepare the `log` package, set the usage, parse the user options, and generate the sequences.

366b ⟨*Main function, Ch. 40* 366b⟩≡ (366a)
```
util.PrepLog("ranseq")
⟨Set usage, Ch. 40 366d⟩
⟨Parse options, Ch. 40 367a⟩
⟨Generate sequences, Ch. 40 367d⟩
```

We import `util`.

366c ⟨*Imports, Ch. 40* 366c⟩≡ (366a) 366e ▷
```
"github.com/evolbioinf/biobox/util"
```

When setting the usage, we state the usage proper and give description of the program plus an example.

366d ⟨*Set usage, Ch. 40* 366d⟩≡ (366b)
```
u := "ranseq [-h] [options]"
d := "Generate random sequence."
e := "ranseq -l 1000"
clio.Usage(u, d, e)
```

We import `clio`.

366e ⟨*Imports, Ch. 40* 366c⟩+≡ (366a) ◁366c 367b ▷
```
"github.com/evolbioinf/clio"
```

We parse the options and check immediately whether the user requested the program version.

367a ⟨*Parse options, Ch. 40* 367a⟩≡ (366b)
```
flag.Parse()
if *optV {
        util.PrintInfo("ranseq")
}
```

We import `flag`.

367b ⟨*Imports, Ch. 40* 366c⟩+≡ (366a) ◁366e 367e▷
```
"flag"
```

We also declare the options. Apart from `-v`, there is the sequence length, `-l`, the number of sequences, `-n`, the G/C content, `-g`, and the seed for the random number generator, `-s`. By default the seed is zero, which prompts the program to generate it internally. This is expected to be the default usage, but occasionally someone might like to exactly reproduce a "random" sequence.

367c ⟨*Variables, Ch. 40* 367c⟩≡ (366a)
```
var optV = flag.Bool("v", false, "version")
var optL = flag.Int("l", 100, "sequence length")
var optN = flag.Int("n", 1,   "number of sequences")
var optG = flag.Float64("g", 0.5, "G/C content")
var optS = flag.Int("s", 0, "seed for random number generator; " +
        "default: internal")
```

To generate the requested sequences, we first initialize the random number generator, `r`, and declare a byte array for a random sequence, `s`, and a single byte for a random nucleotide, `c`.

367d ⟨*Generate sequences, Ch. 40* 367d⟩≡ (366b)
```
var r *rand.Rand
⟨Prepare random number generator, Ch. 40 367f⟩
var s []byte
var c byte
for i := 0; i < *optN; i++ {
        ⟨Generate one sequence, Ch. 40 368b⟩
}
```

We import the package `math/rand`.

367e ⟨*Imports, Ch. 40* 366c⟩+≡ (366a) ◁367b 368a▷
```
"math/rand"
```

The random number generator is either seeded with the number passed via the option `-s`, or with the current UNIX time.

367f ⟨*Prepare random number generator, Ch. 40* 367f⟩≡ (367d)
```
if *optS != 0 {
        r = rand.New(rand.NewSource(int64(*optS)))
} else {
        t := time.Now().UnixNano()
        r = rand.New(rand.NewSource(t))
}
```

We import the package `time`.

368a ⟨*Imports, Ch. 40* 366c⟩+≡ (366a) ◁367e 368c ▷
```
"time"
```

When generating a new sequence, we first erase the old one by reslicing, and generate a header, `Rand1`, `Rand2`, and so on. Then we pick as many random nucleotides as set via `-l`. After that loop, we construct a new sequence from the header and nucleotide slice and print it.

368b ⟨*Generate one sequence, Ch. 40* 368b⟩≡ (367d)
```
s = s[:0]
h := "Rand" + strconv.Itoa(i+1)
for j := 0; j < *optL; j++ {
        ⟨Pick random nucleotide, Ch. 40 368d⟩
        s = append(s, c)
}
seq := fasta.NewSequence(h, s)
fmt.Println(seq)
```

We import `strconv`, `fasta`, and `fmt`.

368c ⟨*Imports, Ch. 40* 366c⟩+≡ (366a) ◁368a
```
"strconv"
"github.com/evolbioinf/fasta"
"fmt"
```

A single random nucleotide is picked by first deciding whether it's a `G/C` or an `A/T`, depending on the G/C content set by the user. Within these two categories, the chance of picking either nucleotide is the same.

368d ⟨*Pick random nucleotide, Ch. 40* 368d⟩≡ (368b)
```
if r.Float64() < *optG {
        if r.Float64() < 0.5 { c = 'G'
        } else { c = 'C'
        }
} else {
        if r.Float64() < 0.5 { c = 'A'
        } else { c = 'T'
        }
}
```

We're done writing `ranseq`, so let's test it.

Testing

Our outline contains hooks for imports and the actual testing.

369a ⟨*ranseq_test.go* 369a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 40 369c⟩
)

func TestRanseq(t *testing.T) {
        ⟨Testing, Ch. 40 369b⟩
}
```

We run `ranseq` with a fixed seed and compare the result we get with the result we want stored in `res1.fasta`.

369b ⟨*Testing, Ch. 40* 369b⟩≡ (369a) 369d ▷
```
cmd := exec.Command("./ranseq", "-s", "13")
g, err := cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err := ioutil.ReadFile("res1.fasta")
if !bytes.Equal(g, w) {
        t.Errorf("want:\n%s\nget\n%s\n", w, g)
}
```

We import `exec`, `ioutil`, and `bytes`.

369c ⟨*Testing imports, Ch. 40* 369c⟩≡ (369a)
```
"os/exec"
"io/ioutil"
"bytes"
```

We repeat the test without setting the seed. This time the result should *differ* from that in `res1.fasta`.

369d ⟨*Testing, Ch. 40* 369b⟩+≡ (369a) ◁ 369b 370a ▷
```
cmd = exec.Command("./ranseq")
g, err = cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res1.fasta")
if bytes.Equal(g, w) {
        t.Errorf("don't want:\n%s\nbut do get\n%s\n", w, g)
}
```

We continue testing the seed fixed and generate two sequences this time.

370a ⟨*Testing, Ch. 40* 369b⟩+≡ (369a) ◁369d 370b ▷

```
cmd = exec.Command("./ranseq", "-s", "13", "-n", "2")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res2.fasta")
if !bytes.Equal(g, w) {
          t.Errorf("want:\n%s\nget\n%s\n", w, g)
}
```

As a final test, we change the GC-content.

370b ⟨*Testing, Ch. 40* 369b⟩+≡ (369a) ◁370a

```
cmd = exec.Command("./ranseq", "-s", "13", "-g", "0.3")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res3.fasta")
if !bytes.Equal(g, w) {
          t.Errorf("want:\n%s\nget\n%s\n", w, g)
}
```

Chapter 41

Program `repeater`: Find Maximal Repeats

Table 41.1: Enhanced suffix array of $t =$ `CTAATAATG`.

i	$\mathrm{sa}[i]$	$\mathrm{lcp}[i]$	$\mathrm{suf}[i]$
1	3	-1	`AATAATG`
2	6	3	`AATG`
3	4	1	`ATAATG`
4	7	2	`ATG`
5	1	0	`CTAATAATG`
6	9	0	`G`
7	2	0	`TAATAATG`
8	5	4	`TAATG`
9	8	1	`TG`

The repeat structure of molecular sequences, particularly DNA sequences, plays an important role in the design of genetic markers and read mapping. However, some care needs to be taken when describing this structure to avoid excessive output. For example, we might be tempted to extract all pairs of repeated substrings, but their number grows very quickly with sequence length. Consider for example the sequence $t =$ `AAA` and let's write a repeated pair as a triple consisting of the two starting positions, and the length. Our tiny example sequence would already produce 4 repeat pairs, $(1, 2, 1)$, $(1, 3, 1)$, $(2, 3, 1)$, and $(1, 2, 2)$. To avoid this behavior, we restrict our attention to maximal substrings.

In a second example sequence $t =$ `CAGATAT`, `A` is repeated three times. The pairs $(2, 4, 1)$ and $(2, 6, 1)$ cannot be extended to the left or to the right without losing the repeat. Such pairs of substrings are called maximal [12, p. 143]. In contrast, the pair $(4, 6, 1)$ can be extended to the right, so it is not maximal.

Any sequence of non-trivial length tends to contain many maximal pairs. To obtain a more compact description of repetitiveness, a maximal repeat is defined as a substring that participates in a maximal pair. So `A` would be a maximal repeat in t as it participates in the maximal pair $(2, 4, 1)$, even though it also participates in the pair $(4, 6, 1)$, which is not maximal. The program `repeater` finds all maximal repeats in a sequence.

The maximal repeats of a string are discovered from its suffix tree. Figure 41.1 shows the suffix tree for a third example sequence, $t =$ `CTAATAATG`. Any path label that ends at an internal node can be extended by at least two distinct nucleotides, which makes it right-diverse. Maximal repeats are those right-diverse path labels that are also left-diverse. Left-diversity is discovered by traversing the suffix tree from the leaves upward. During such a bottom-up traversal we encounter leaves and internal nodes. If we are at a leaf, we store the character to its left in the parent node. If we are at an internal node, we pass characters already collected to the parent. As soon as two or more characters are found, the node is left-diverse and this property propagates up the tree [12, p. 145].

In practice, suffix trees have been replaced by enhanced suffix arrays. Table 41.1 shows the enhanced suffix array corresponding to the suffix tree in Figure 41.1. It consists of the suffix array, sa, the longest common prefix array, lcp, and the suffixes, suf. Table 41.1 is in the customary vertical orientation. However, it's easier to get from the array to the tree if we rotate it and omit the suffixes as shown in Table 41.2. Notice

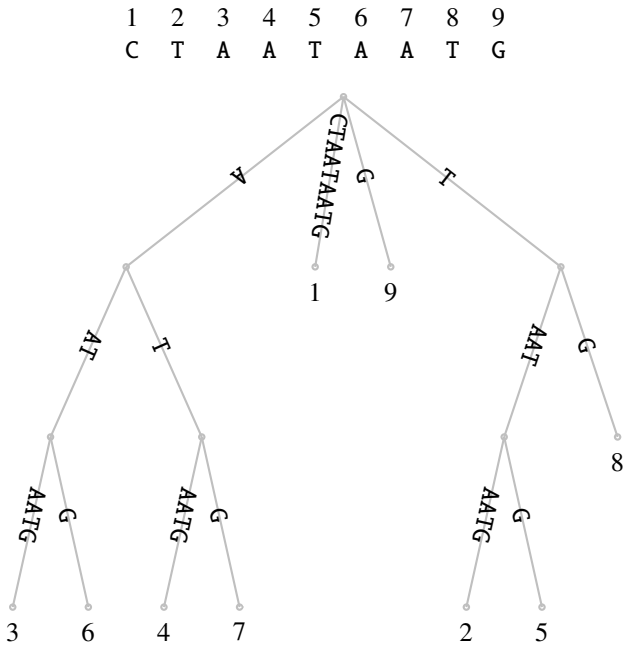


Figure 41.1: Suffix tree of $t =$ CTAATAATG.

Table 41.2: Setup for converting an enhanced suffix array to its suffix tree.

i	1	2	3	4	5	6	7	8	9	10
$\mathrm{sa}[i]$	3	6	4	7	1	9	2	5	8	
$\mathrm{lcp}[i]$	-1	3	1	2	0	0	0	4	1	-1

that the order of entries in sa is the same as the order of leaves in the suffix tree. So by annotating sa with parenthesis, we get the tree:

$$(((3\ 6)(4\ 7))1\ 9((7\ 5)8))$$

To find these parentheses, we begin by appending a last lcp value that is smaller than all legitimate values, -1. We initialize the empty Table 41.2 by writing the opening parenthesis of the root node, which we also annotate with its depth, 0.

i	$\dot{1}$	2	3	4	5	6	7	8	9	10
$\mathrm{sa}[i]$	$(_0$ 3	6	4	7	1	9	2	5	8	
$\mathrm{lcp}[i]$	-1	3	1	2	0	0	0	4	1	-1

We denote this half-open root node as the pair $(0, 1)$ and place it on a stack that we manipulate throughout tree construction using the functions $\mathrm{push}(v)$ to add node v to the stack, $\mathrm{top}()$ to refer to the uppermost node, and $\mathrm{pop}()$ to remove the top node. We're done with $i = 1$ and tick it off with a dot.

The repeated part of the procedure starts at $i = 2$. From now on we ask at every i, whether $\mathrm{lcp}[i]$ is less than or greater than $\mathrm{top}().d$. While $\mathrm{lcp}[i] < \mathrm{top}().d$, we write closing parentheses at $i-1$ and remove, or pop, the corresponding opening parentheses from the stack. If $\mathrm{lcp}[i] > \mathrm{top}().d$, we place an opening parenthesis at the position of the last opening parenthesis removed, or, if none was removed, at $i - 1$. Here is a summary of the procedure:

$\ell \leftarrow i - 1$ {Left border of node.}
while $\mathrm{lcp}[i] < \mathrm{top}().d$ **do**
  write ")" at $i - 1$ {Right border of node.}
  $\ell \leftarrow \mathrm{top}().\ell$
  dot the depth of the corresponding "("
  $\mathrm{pop}()$
end while
if $\mathrm{lcp}[i] > \mathrm{top}().d$ **then**
  $d \leftarrow \mathrm{lcp}[i]$
  write "$(_d$" at ℓ
  $\mathrm{push}(d, \ell)$
end if

Now, $\mathrm{lcp}[2] > \mathrm{top}().d = 0$ so we write $(_3$ at $i = 1$ to push $(3, 1)$.

i		$\dot{1}$	$\dot{2}$	3	4	5	6	7	8	9	10
$\mathrm{sa}[i]$	$(_0$	$(_3$ 3	6	4	7	1	9	2	5	8	
$\mathrm{lcp}[i]$		-1	3	1	2	0	0	0	4	1	-1

At $i = 3$ we close $(_3$ and dot it to remove it from the stack. Now, $\mathrm{lcp}[3] = 1$ is greater than the depth of the new top node, 0, so we write $(_1$ at $i = 1$ to push $(1, 1)$.

i		$\dot{1}$	$\dot{2}$	$\dot{3}$	4	5	6	7	8	9	10
$\mathrm{sa}[i]$	$(_0$	$(_1$ $(_{\dot{3}}$ 3	6)	4	7	1	9	2	5	8	
$\mathrm{lcp}[i]$		-1	3	1	2	0	0	0	4	1	-1

At $i = 4$ we write $(_4$ at $i = 3$ to push $(2, 3)$.

i		$\dot{1}$	$\dot{2}$	$\dot{3}$	$\dot{4}$	5	6	7	8	9	10
$\mathrm{sa}[i]$	$(_0$	$(_1$ $(_{\dot{3}}$ 3	6)	$(_2$ 4	7	1	9	2	5	8	
$\mathrm{lcp}[i]$		-1	3	1	2	0	0	0	4	1	-1

At $i = 5$ we write two closing parentheses at $i = 4$ and dot the corresponding depths to pop $(2, 3)$ and $(1, 1)$.

i		$\dot{1}$	$\dot{2}$	$\dot{3}$	$\dot{4}$	$\dot{5}$	6	7	8	9	10
$\mathrm{sa}[i]$	$(_0$	$(_{\dot{1}}$ $(_{\dot{3}}$ 3	6)	$(_{\dot{2}}$ 4	7))	1	9	2	5	8	
$\mathrm{lcp}[i]$		-1	3	1	2	0	0	0	4	1	-1

At $i = 6$ and $i = 7$ nothing changes, so we skip forward to $i = 8$, where we write $(_4$ at $i = 7$ to push $(4, 7)$.

i		$\dot{1}$	$\dot{2}$	$\dot{3}$	$\dot{4}$	$\dot{5}$	$\dot{6}$	$\dot{7}$	$\dot{8}$	9	10
$\mathrm{sa}[i]$	$(_0$	$(_{\dot{1}}$ $(_{\dot{3}}$ 3	6)	$(_{\dot{2}}$ 4	7))	1	9	$(_4$ 2	5	8	
$\mathrm{lcp}[i]$		-1	3	1	2	0	0	0	4	1	-1

At $i = 9$ we write) at $i = 8$ and pop $(4, 7)$; then we write $(_1$ at $i = 7$ to push $(1, 7)$.

i		$\dot{1}$	$\dot{2}$	$\dot{3}$	$\dot{4}$	$\dot{5}$	$\dot{6}$	$\dot{7}$	$\dot{8}$	$\dot{9}$	10
$\mathrm{sa}[i]$	($_{\dot{0}}$	($_{\dot{1}}$ ($_{\dot{3}}$ 3	6)	($_{\dot{2}}$ 4	7))	1	9	($_{1}$ ($_{\dot{4}}$ 2	5)	8	
$\mathrm{lcp}[i]$		-1	3	1	2	0	0	0	4	1	-1

Finally, at $i = 10$ we close and pop the nodes with depths 1 and 0 remaining on the stack. And, voilá, there's our tree:

i		$\dot{1}$	$\dot{2}$	$\dot{3}$	$\dot{4}$	$\dot{5}$	$\dot{6}$	$\dot{7}$	$\dot{8}$	$\dot{9}$	$\dot{1}0$
$\mathrm{sa}[i]$	($_{\dot{0}}$	($_{\dot{1}}$ ($_{\dot{3}}$ 3	6)	($_{\dot{2}}$ 4	7))	1	9	($_{\dot{1}}$ ($_{\dot{4}}$ 2	5)	8))	
$\mathrm{lcp}[i]$		-1	3	1	2	0	0	0	4	1	-1

This gives us a paper-and-pencil construction of suffix trees from lcp arrays. It also gives us a new perspective on the search for maximal repeats. I said earlier that these could be found by walking up the suffix tree in Figure 41.1 and noting whether any of the suffixes in a node's subtree differed in their left characters.

Now, during our paper-and-pencil procedure we encountered the suffixes in the order in which they occur in the tree. So we just monitor the last position at which a suffix differed from its neighbor. If that position is greater than the left border of the current node, the node is a maximal repeat, unless, of course, it's the root.

This reasoning is summarized in Algorithm 5 [25, p. 149]. In its last if-clause, we compare the characters to the left of two suffixes. For this to work, we conceptually place a sentinel character to the left of the first proper character in t, $t[0] = \$$. We also refer to $\mathrm{sa}[i]$, where i ranges from 1 to n, which is the length of the lcp-array. However, the lcp-array is one element longer than sa, so we have to make sure we don't overstep its right-hand border. The program `repeater` implements this algorithm to find the maximal repeats of one or more sequences.

Implementation

The outline of `repeater` contains hooks for imports, types, functions, and the logic of the main function.

375a ⟨*repeater.go* 375a⟩≡

```
package main

import (
        ⟨Imports, Ch. 41 377a⟩
)
⟨Types, Ch. 41 380c⟩
⟨Methods, Ch. 41 380e⟩
⟨Functions, Ch. 41 378b⟩
func main() {
        ⟨Main function, Ch. 41 375b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare and parse the options, and parse the input files.

375b ⟨*Main function, Ch. 41* 375b⟩≡ (375a)

```
util.PrepLog("repeater")
⟨Set usage, Ch. 41 377b⟩
⟨Declare options, Ch. 41 377d⟩
⟨Parse options, Ch. 41 377f⟩
⟨Parse input files, Ch. 41 377g⟩
```

Algorithm 5 Finding maximal repeats by bottom-up suffix tree traversal [25, p. 149].

Require: t {Text with $t[0] = \$$.}
Require: lcp {lcp array with -1 appended.}
Require: n {Length of lcp array}
Require: sa {Suffix array of $t[1...n-1]$}
Ensure: Maximal repeats.
  $\mathrm{push}(0, 1)$ {Initialize stack.}
  $\delta \leftarrow 1$ {Position of most recent left-difference.}
  for $i \leftarrow 2$ **to** n **do**
    $\ell \leftarrow i - 1$ {Left border of node.}
    while stack not empty **and** $\mathrm{lcp}[i] < \mathrm{top}().d$ **do**
      $v \leftarrow \mathrm{pop}()$
      $\ell \leftarrow v.\ell$
      $d \leftarrow v.d$
      if $d > 0$ **and** $\delta > \ell$ **then**
        $m \leftarrow t[\mathrm{sa}[\ell]...\mathrm{sa}[\ell] + d - 1]$ {The maximal repeat.}
        report maximal repeat m at $\mathrm{sa}[j], \ell \leq j \leq i - 1$
      end if
    end while
    if stack not empty **and** $\mathrm{lcp}[i] > \mathrm{top}().d$ **then**
      $\mathrm{push}(\mathrm{lcp}[i], \ell)$
    end if
    if $i < n$ **and** $t[\mathrm{sa}[i-1] - 1] \neq t[sa[i] - 1]$ **then**
      $\delta \leftarrow i$
    end if
  end for

We import `util`.

377a ⟨*Imports, Ch. 41* 377a⟩≡ (375a) 377c▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts, the usage message proper, an explanation of the program's purpose, and an example command.

377b ⟨*Set usage, Ch. 41* 377b⟩≡ (375b)
```
u := "repeater [-h] [options] [files]"
p := "Find maximal repeats."
e := "repeater foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

377c ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁377a 377e▷
```
"github.com/evolbioinf/clio"
```

We declare five options,

1. `-m` m: print only repeats of minimum length m
2. `-r`: include reverse strand
3. `-p`: print all positions
4. `-s`: print full sequences
5. `-v` print program version

377d ⟨*Declare options, Ch. 41* 377d⟩≡ (375b)
```
var optM = flag.Int("m", 0, "minimum repeat length; default: longest")
var optR = flag.Bool("r", false, "include reverse strand")
var optP = flag.Bool("p", false, "print all positions")
var optS = flag.Bool("s", false, "print full sequences")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

377e ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁377c 378a▷
```
"flag"
```

We parse the options and respond to `-v`.

377f ⟨*Parse options, Ch. 41* 377f⟩≡ (375b)
```
flag.Parse()
if *optV {
        util.PrintInfo("repeater")
}
```

The remaining arguments on the command line are interpreted as input files. They are parsed by applying the function `scan` to each of them. It takes as arguments the options.

377g ⟨*Parse input files, Ch. 41* 377g⟩≡ (375b)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optR, *optP, *optS, *optM)
```

We import `fasta`.

378a ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁377e 378c▷
```
"github.com/evolbioinf/fasta"
```

Inside `scan` we retrieve the arguments passed, collect the sequences, and analyze them.

378b ⟨*Functions, Ch. 41* 378b⟩≡ (375a) 380a▷
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 41 378d⟩
        ⟨Collect sequences, Ch. 41 378e⟩
        ⟨Analyze sequences, Ch. 41 378f⟩
}
```

We import `io`.

378c ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁378a 379e▷
```
"io"
```

The arguments are retrieved via reflection.

378d ⟨*Retrieve arguments, Ch. 41* 378d⟩≡ (378b)
```
optR := args[0].(bool)
optP := args[1].(bool)
optS := args[2].(bool)
optM := args[3].(int)
```

We store the sequences contained in the file.

378e ⟨*Collect sequences, Ch. 41* 378e⟩≡ (378b)
```
var sequences []*fasta.Sequence
scanner := fasta.NewScanner(r)
for scanner.ScanSequence() {
        sequence := scanner.Sequence()
        sequences = append(sequences, sequence)
}
```

We concatenate the sequences and compute the enhanced suffix array. To understand the relationship between the sequences known to the user and the concatenated version we analyze, consider $t =$ `AC$AC$A`. Taken at face value, this contains the maximal repeat `AC$A`. In other words, there's nothing in the unprocessed enhanced suffix array that stops a repeat from crossing sequence borders. So we check the lcp array and trim values that run over. Then we compute the maximal repeats, determine their minimum length, and collect the repeats that conform to that minimum. They are sorted by size and printed.

378f ⟨*Analyze sequences, Ch. 41* 378f⟩≡ (378b)
```
⟨Concatenate sequences, Ch. 41 379a⟩
⟨Compute enhanced suffix array, Ch. 41 379d⟩
⟨Check lcp-values for run over, Ch. 41 379f⟩
⟨Compute maximal repeats, Ch. 41 380b⟩
⟨Determine minimum repeat length, Ch. 41 382b⟩
⟨Collect repeats of minimum length, Ch. 41 383a⟩
⟨Sort repeats by size, Ch. 41 383d⟩
⟨Print repeats, Ch. 41 383f⟩
```

We concatenate the forward strands and, if requested, the reverse strands, too.

379a ⟨*Concatenate sequences, Ch. 41* 379a⟩≡ (378f)
```
⟨Concatenate forward strands, Ch. 41 379b⟩
if optR {
        ⟨Concatenate reverse strands, Ch. 41 379c⟩
}
```

The sequence data is concatenated into a byte slice. Each sequence is terminated by the zero byte as separator. Any match across sequence boundaries would thus be flagged in the output as an unprintable character.

379b ⟨*Concatenate forward strands, Ch. 41* 379b⟩≡ (379a)
```
var cat []byte
var ends []int
for i, sequence := range sequences {
        if i > 0 {
                cat = append(cat, 0)
        }
        cat = append(cat, sequence.Data()...)
        ends = append(ends, len(cat))
}
```

To simplify the analysis, we pretend for now the reverse strands are just another batch of forward strands. They are appended in the same order as the forward strand.

379c ⟨*Concatenate reverse strands, Ch. 41* 379c⟩≡ (379a)
```
for _, sequence := range sequences {
        sequence.ReverseComplement()
        cat = append(cat, 0)
        cat = append(cat, sequence.Data()...)
        ends = append(ends, len(cat))
}
```

The enhanced suffix array consists of the suffix array proper and the lcp array.

379d ⟨*Compute enhanced suffix array, Ch. 41* 379d⟩≡ (378f)
```
sa := esa.Sa(cat)
lcp := esa.Lcp(cat, sa)
```

We import `esa`.

379e ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁378c 382d▷
```
"github.com/evolbioinf/esa"
```

To check for out of bounds repeats, convert suffix positions to sequence identifiers. As we shall need this conversion again when printing the repeats, we delegate it to the function `positionToSequence`. Any out of bound lcp-value we do find is trimmed to the correct length.

379f ⟨*Check* lcp*-values for run over, Ch. 41* 379f⟩≡ (378f)
```
for i, p := range sa {
        seq := positionToSequence(p, ends)
        l := ends[seq] - p
        if p + lcp[i] > ends[seq] {
                lcp[i] = l
        }
}
```

In the function `positionToSequence` we iterate across the sequence ends until we find the interval containing the position.

380a ⟨*Functions, Ch. 41* 378b⟩+≡ (375a) ◁378b 384d▷
```
func positionToSequence(p int, ends []int) int {
        var start, end, seq int
        for seq, end = range ends {
                if p >= start && p <= end {
                        break
                }
        }
        return seq
}
```

The maximal repeats are computed by implementing Algorithm 5 to traverse the suffix tree of the input. For this tree traversal we prepare the required variables before walking across the lcp array.

380b ⟨*Compute maximal repeats, Ch. 41* 380b⟩≡ (378f)
⟨*Prepare variables, Ch. 41* 381a⟩
⟨*Iterate over* lcp *array, Ch. 41* 381b⟩

A suffix tree consists of nodes. In the pseudocode of Algorithm 5, they consist of two fields, depth, d, and left border, ℓ. In our implementation we add the right border, r, so that we can store all the nodes that signify maximal repeats before printing them, rather than mixing printing and traversal.

380c ⟨*Types, Ch. 41* 380c⟩≡ (375a) 380d▷
```
type node struct {
        d, l, r int
}
```

The nodes are kept on a stack, which we implement as a slice [7, p. 92].

380d ⟨*Types, Ch. 41* 380c⟩+≡ (375a) ◁380c 383b▷
```
type stack []node
```

We implement the three conventional stack functions, `top`, `pop`, and `push`.

380e ⟨*Methods, Ch. 41* 380e⟩≡ (375a) 383c▷
```
func (s *stack) top() node { return (*s)[len(*s)-1] }
func (s *stack) push(n node) { *s = append(*s, n) }
func (s *stack) pop() node {
        n := (*s)[len(*s)-1]
        *s = (*s)[0:len(*s)-1]
        return n
}
```

As shown in Algorithm 5, we append the final -1 to the lcp array, denote its length as n, and create a stack onto which we push the root node. We also create a slice for storing the nodes that correspond to maximal repeats. The last significant variable, δ, is the position of the most recent left-diverse suffix.

381a ⟨*Prepare variables, Ch. 41* 381a⟩≡ (380b)
```
lcp = append(lcp, -1)
n := len(lcp)
s := new(stack)
root := node{d: 0, l: 1}
s.push(root)
var repeats []node
var delta int
```

Each step in the iteration over the lcp array consists of two parts, the while loop, and the if clauses.

381b ⟨*Iterate over* lcp *array, Ch. 41* 381b⟩≡ (380b)
```
for i := 1; i < n; i++ {
        l := i - 1
        ⟨While loop, Ch. 41 381c⟩
        ⟨If clauses, Ch. 41 381d⟩
}
```

In the while loop we remove nodes from the stack until it is either empty or the current lcp value is greater or equal to the depth of the top node. The right border of these nodes is $i - 1$. Left-diverse nodes represent maximal repeats and are stored.

381c ⟨*While loop, Ch. 41* 381c⟩≡ (381b)
```
for len(*s) > 0 && lcp[i] < s.top().d {
        v := s.pop()
        l = v.l
        if delta > l && v.d > 0 {
                v.r = i - 1
                repeats = append(repeats, v)
        }
}
```

In the first if clause outside the while loop, we first make sure the stack isn't empty. Then we ask whether the current lcp value is greater than the depth of the top node. If so, we push a new node with the left border of the most recently popped node.

381d ⟨*If clauses, Ch. 41* 381d⟩≡ (381b) 382a▷
```
if len(*s) > 0 && lcp[i] > s.top().d {
        s.push(node{d: lcp[i], l: l})
}
```

If the current suffix left-differs from its neighbor, we update δ. We avoid referencing the character to the left of the first character and simply note a difference instead. We also ensure that the sentinel character, 0, differs from all characters, even itself.

382a ⟨*If clauses, Ch. 41* 381d⟩+≡ (381b) ◁381d

```
if i >= n-1 { continue }
pos1 := sa[i-1] - 1
pos2 := sa[i] - 1
if pos1 < 0 || pos2 < 0 {
        delta = i
} else if cat[pos1] == 0 || cat[pos2] == 0 {
          delta = i
} else if cat[pos1] != cat[pos2] {
          delta = i
}
```

The minimum repeat length is either set by the user or is the maximum repeat length. Now, the user may have requested repeats longer than available, in which case we'd like to send a message. So to determine the minimum repeat length we first compute the maximum repeat length.

382b ⟨*Determine minimum repeat length, Ch. 41* 382b⟩≡ (378f) 382c▷

```
max := 0
for _, repeat := range repeats {
          if max < repeat.d {
                  max = repeat.d
          }
}
```

If the user set a minimum repeat length, we check this against the maximum just determined. If the user requested a length greater than the maximum, we reset the minimum to the maximum and send a waring. Otherwise the maximum becomes the minimum.

382c ⟨*Determine minimum repeat length, Ch. 41* 382b⟩+≡ (378f) ◁382b

```
min := 0
if optM == 0 {
          min = max
} else {
          if optM <= max {
                  min = optM
          } else {
                  min = max
                  fmt.Fprintf(os.Stderr, "there aren't any " +
                          "repeats longer than %d\n", min)
          }
}
```

We import `os`.

382d ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁379e 383e▷

```
"os"
```

We collect the repeats of minimum length

383a ⟨*Collect repeats of minimum length, Ch. 41* 383a⟩≡ (378f)
```
var mRepeats = make([]node, 0)
for _, repeat := range repeats {
        if repeat.d >= min {
                mRepeats = append(mRepeats, repeat)
        }
}
```

To sort the repeats by descending length, we declare the type `nodes`,

383b ⟨*Types, Ch. 41* 380c⟩+≡ (375a) ◁380d
```
type nodes []node
```

and make it sortable.

383c ⟨*Methods, Ch. 41* 380e⟩+≡ (375a) ◁380e
```
func (n nodes) Len() int { return len(n) }
func (n nodes) Less(i, j int) bool { return n[i].d > n[j].d }
func (n nodes) Swap(i, j int) { n[i], n[j] = n[j], n[i] }
```

Then we sort the repeats.

383d ⟨*Sort repeats by size, Ch. 41* 383d⟩≡ (378f)
```
sort.Sort(nodes(mRepeats))
```

We import `sort`.

383e ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁382d 383h▷
```
"sort"
```

Repeats are printed in a table using a tab writer. This is set up before we print the table header and iterate over the repeats of minimum length. For each repeat, we convert the position of its first instance from a coordinate in the concatenated sequence to a position the user understands. After the last repeat, the table is printed.

383f ⟨*Print repeats, Ch. 41* 383f⟩≡ (378f)
```
⟨Setup tab writer, Ch. 41 383g⟩
⟨Write table header, Ch. 41 384a⟩
for _, repeat := range mRepeats {
        ⟨Convert array position to user position, Ch. 41 384c⟩
        ⟨Write a repeat, Ch. 41 385a⟩
}
⟨Print table, Ch. 41 386c⟩
```

A tab writer writes to a buffer, which we initialize to a column width of 1, tabs of zero characters, and padding with two blanks.

383g ⟨*Setup tab writer, Ch. 41* 383g⟩≡ (383f)
```
var buf []byte
buffer := bytes.NewBuffer(buf)
w := new(tabwriter.Writer)
w.Init(buffer, 1, 0, 2, ' ', 0)
```

We import `bytes` and `tabwriter`.

383h ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁383e 384b▷
```
"bytes"
"text/tabwriter"
```

The table consists of four columns, length, count, sequence, and positions. By default only one of the positions is printed, but the user can request all of them, in which case we change the column header *Position* to plural.

384a ⟨*Write table header, Ch. 41* 384a⟩≡ (383f)
```
fmt.Fprint(w, "#\tLength\tCount\tSequence\tPosition")
if optP {
        fmt.Fprint(w, "s")
}
fmt.Fprint(w, "\n")
```

We import `fmt`.

384b ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁383h 386b▷
```
"fmt"
```

Given a suffix position, we need its strand, sequence identifier, and position within that sequence. As we might want to calculate these values for each instance of a repeat, we delegate this to the function `position`. It takes as arguments a repeat position, its length, the end positions, and whether or not the reverse strand was included.

384c ⟨*Convert array position to user position, Ch. 41* 384c⟩≡ (383f)
```
strand, seqId, pos := position(sa[repeat.l], repeat.d, ends, optR)
```

Given a position, we check whether it's on the forward or reverse strands.

384d ⟨*Functions, Ch. 41* 378b⟩+≡ (375a) ◁380a 386a▷
```
func position(p, l int, ends []int, rev bool) (byte, int, int) {
        seqId := positionToSequence(p, ends)
        strand := byte('f')
        if rev && p > ends[len(ends)/2-1] {
                strand = byte('r')
                ⟨Determine position on reverse strand, Ch. 41 384e⟩
        } else {
                ⟨Determine position on forward strand, Ch. 41 384f⟩
        }
        return strand, seqId, p
}
```

On the reverse strand, a position is mapped to its forward equivalent by subtracting it from the end. We also adjust the sequence identifier.

384e ⟨*Determine position on reverse strand, Ch. 41* 384e⟩≡ (384d)
```
p = ends[seqId] - p - 1
seqId -= len(ends) / 2
```

On the forward strand, a position minus the start gives the absolute position in that sequence. The start is either zero or follows the end of its predecessor.

384f ⟨*Determine position on forward strand, Ch. 41* 384f⟩≡ (384d)
```
start := 0
if seqId > 0 {
        start = ends[seqId-1] + 1
}
p -= start
```

A repeat is written in three steps, its length and count, its sequence, and its positions.

385a ⟨*Write a repeat, Ch. 41* 385a⟩≡ (383f)
```
⟨Write length and count, Ch. 41 385b⟩
⟨Write sequence, Ch. 41 385c⟩
⟨Write positions, Ch. 41 385d⟩
```

The count of a repeat is the length of the node interval.

385b ⟨*Write length and count, Ch. 41* 385b⟩≡ (385a)
```
count := repeat.r - repeat.l + 1
fmt.Fprintf(w, "\t%d\t%d", repeat.d, count)
```

Repeats can be very long, so by default we write any repeat longer than 13 residues as the first five residues, followed by three dots, followed by the last five residues. However, the user can request the full sequence.

385c ⟨*Write sequence, Ch. 41* 385c⟩≡ (385a)
```
p := sa[repeat.l]
seq := cat[p:p+repeat.d]
if optS || repeat.d <= 13 {
          fmt.Fprintf(w, "\t%s", seq)
} else {
          fmt.Fprintf(w, "\t%s", seq[0:5])
          fmt.Fprintf(w, "...")
          fmt.Fprintf(w, "%s", seq[repeat.d-5:repeat.d])
}
```

A position in the concatenated sequence is converted to a string consisting of sequence identifier, position, and strand. By default we write a single position, but the user can request them all. The formatting of individual positions is delegated to the function `posStr`.

385d ⟨*Write positions, Ch. 41* 385d⟩≡ (385a)
```
str := posStr(strand, seqId+1, pos+1, len(sequences), optR)
fmt.Fprintf(w, "\t%s", str)
if optP {
          for i := repeat.l + 1; i <= repeat.r; i++ {
                  strand, seqId, pos = position(sa[i], repeat.d, ends, optR)
                  str = posStr(strand, seqId+1, pos+1, len(sequences), optR)
                  fmt.Fprintf(w, " %s", str)
          }
}
fmt.Fprintf(w, "\n")
```

The function `posStr` prints strand, sequence identifier, and position as

$$\mathrm{f|r}i : p$$

The strand information is dropped if only the forward strand was analyzed, the sequence identifier is dropped if only one sequence was analyzed.

386a ⟨*Functions, Ch. 41* 378b⟩+≡ (375a) ◁384d
```
func posStr(strand byte, seq, pos, num int, rev bool) string {
        str := ""
        if rev { str += string(strand) }
        if num > 1 { str += strconv.Itoa(seq) }
        if rev || num > 1 { str += ":" }
        str += strconv.Itoa(pos)
        return str
}
```

We import `strconv`.

386b ⟨*Imports, Ch. 41* 377a⟩+≡ (375a) ◁384b
```
"strconv"
```

The table of repeats is now written, but before we print the buffer, we flush any remaining bytes from the writer.

386c ⟨*Print table, Ch. 41* 386c⟩≡ (383f)
```
w.Flush()
fmt.Printf("%s", buffer)
```

The program is done, time for a test run.

Testing

The testing outline has hooks for imports and the testing logic.

386d ⟨*repeater_test.go* 386d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 41 387a⟩
)

func TestRepeater(t *testing.T) {
        ⟨Testing, Ch. 41 386e⟩
}
```

Testing proceeds in three phases. We construct a set of test commands and a list of files containing the results we want. Then we run the commands.

386e ⟨*Testing, Ch. 41* 386e⟩≡ (386d)
```
var commands []*exec.Cmd
⟨Construct test commands, Ch. 41 387b⟩
⟨Construct list of result files, Ch. 41 387c⟩
for i, command := range commands {
        ⟨Run test command, Ch. 41 387e⟩
}
```

We import `exec`.

387a ⟨*Testing imports, Ch. 41* 387a⟩≡ (386d) 387d▷
```
"os/exec"
```

We construct five test commands. One without any options, and one for each of the four options.

387b ⟨*Construct test commands, Ch. 41* 387b⟩≡ (386e)
```
c := exec.Command("./repeater", "test.fasta")
commands = append(commands, c)
c = exec.Command("./repeater", "-m", "13", "test.fasta")
commands = append(commands, c)
c = exec.Command("./repeater", "-r", "test.fasta")
commands = append(commands, c)
c = exec.Command("./repeater", "-p", "test.fasta")
commands = append(commands, c)
c = exec.Command("./repeater", "-s", "test.fasta")
commands = append(commands, c)
```

There is one result file per command.

387c ⟨*Construct list of result files, Ch. 41* 387c⟩≡ (386e)
```
var results []string
for i, _ := range commands {
          name := "r" + strconv.Itoa(i+1) + ".txt"
          results = append(results, name)
}
```

We import `strconv`.

387d ⟨*Testing imports, Ch. 41* 387a⟩+≡ (386d) ◁387a 387f▷
```
"strconv"
```

An individual command is run and the result we get compared to what we want.

387e ⟨*Run test command, Ch. 41* 387e⟩≡ (386e)
```
get, err := command.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", command)
}
want, err := ioutil.ReadFile(results[i])
if err != nil {
          t.Errorf("couldnt' open %q\n", results[i])
}
if !bytes.Equal(want, get) {
          t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `ioutil` and `bytes`.

387f ⟨*Testing imports, Ch. 41* 387a⟩+≡ (386d) ◁387d
```
"io/ioutil"
"bytes"
```

Chapter 42

Program `rep2plot`: Plot `repeater` Output

Introduction

The program `repeater` (Chapter 41) prints repeats within or between sequences. For example, here is the `repeater` command for comparing the forward and reverse (`-r`) strands of the *Adh* locus in the fruit flies *Drosohpila melanogaster* and *D. guanche* by plotting all matches (`-p`) with a minimum length of 12 (`-m`):

```
$ cat dmAdhAdhdup.fasta dgAdhAdhdup.fasta | repeater -m 12 -r -p | head
#  Length  Count  Sequence       Positions
   37      2      AGCAA...GAGTG  f1:3292 f2:3287
   37      2      CACTC...TTGCT  r2:3287 r1:3292
   27      2      ATTTG...ATGTT  r1:3949 r2:3741
   27      2      AACAT...CAAAT  f1:3949 f2:3741
   26      2      CTTAC...AAGTT  r1:2569 r2:2529
   26      2      GTGGT...TAGTT  r2:2370 r1:2410
   26      2      AACTT...GTAAG  f2:2529 f1:2569
   26      2      AACTA...ACCAC  f2:2370 f1:2410
   23      2      ACCTC...TTCAT  r1:3865 r2:3657
...
```

Each match has a length, a count of at least two, the sequence, and at least two positions. So, the first match is 37 nucleotides long and starts on the forward strand of sequence 1 at position 3292 and on the forward strand of sequence 2 on position 3287. Such a match can be read as a segment in a dot plot. If we write its start and end positions (x_1, y_1, x_2, y_2), we get $(3292, 3287, 3328, 3323)$. The program `rep2plot` transforms `repeater` output to such segments, which can then be rendered with `plotSeg` (Chapter 35) to give Figure 42.1.

There is one slight complication, though. DNA is double-stranded. So each matching string occurs twice, on the forward and on the reverse strand, where repeater also gives the forward coordinates. For example, the second match in our list is identical to the first, bar the strand. We'd like to avoid printing each segment twice, and do this by grouping duplicates through sorting, which helps us remove them.

Implementation

The outline of `rep2plot` has hooks for imports, types, methods, functions, and the logic of the main function.

389 ⟨*rep2plot.go* 389⟩≡

```
package main

import (
        ⟨Imports, Ch. 42 391b⟩
)
⟨Types, Ch. 42 392d⟩
⟨Methods, Ch. 42 396a⟩
⟨Functions, Ch. 42 392a⟩
func main() {
        ⟨Main function, Ch. 42 391a⟩
}
```

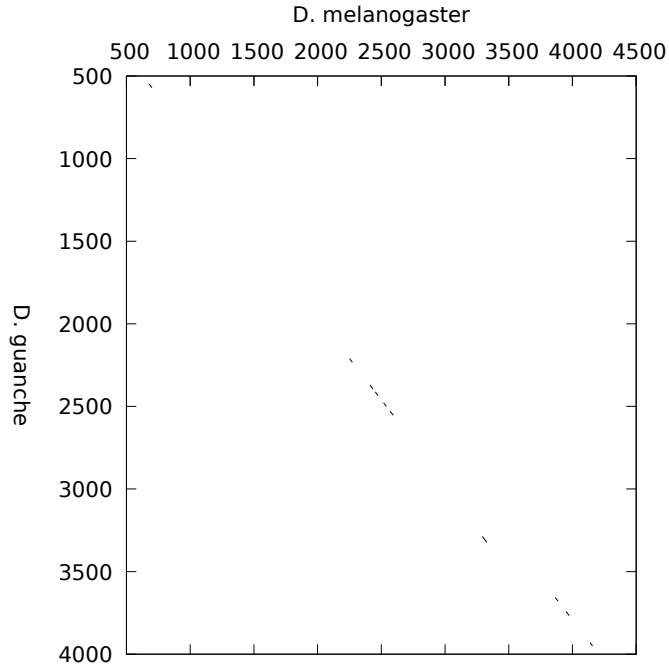


Figure 42.1: Segment plot of the matches between the *Adh* loci in *Drosophila melanogaster* and *D. guanche* using `rep2plot` and `plotSeg` (Chapter 35).

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

391a ⟨*Main function, Ch. 42* 391a⟩≡ (389)
```
util.PrepLog("rep2plot")
⟨Set usage, Ch. 42 391c⟩
⟨Declare options, Ch. 42 391e⟩
⟨Parse options, Ch. 42 391g⟩
⟨Parse input files, Ch. 42 391h⟩
```

We import `util`.

391b ⟨*Imports, Ch. 42* 391b⟩≡ (389) 391d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the program's purpose, and an example command.

391c ⟨*Set usage, Ch. 42* 391c⟩≡ (391a)
```
u := "rep2plot [-h -v] [file]..."
p := "Convert repeater output to plotSeg input."
e := "cat f1.fasta f2.fasta | repeater -m 12 -r -p | " +
        "rep2plot | plotSeg"
clio.Usage(u, p, e)
```

We import `clio`.

391d ⟨*Imports, Ch. 42* 391b⟩+≡ (389) ◁391b 391f▷
```
"github.com/evolbioinf/clio"
```

The help option, `-h`, is always implied, so we only declare the version, `-v`.

391e ⟨*Declare options, Ch. 42* 391e⟩≡ (391a)
```
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

391f ⟨*Imports, Ch. 42* 391b⟩+≡ (389) ◁391d 392b▷
```
"flag"
```

We parse the options and respond to `-v`.

391g ⟨*Parse options, Ch. 42* 391g⟩≡ (391a)
```
flag.Parse()
if *optV {
        util.PrintInfo("rep2plot")
}
```

The remaining tokens on the command line are taken as input files. We scan them with the function `scan`.

391h ⟨*Parse input files, Ch. 42* 391h⟩≡ (391a)
```
files := flag.Args()
clio.ParseFiles(files, scan)
```

Inside scan, we create a reader, use it to scan the file, and print the segments read during the scan.

392a ⟨*Functions, Ch. 42* 392a⟩≡ (389)
```
func scan(r io.Reader, args ...interface{}) {
        reader := bufio.NewReader(r)
        ⟨Scan file, Ch. 42 392c⟩
        ⟨Print segments, Ch. 42 395a⟩
}
```

We import `io` and `bufio`.

392b ⟨*Imports, Ch. 42* 391b⟩+≡ (389) ◁391f 393b▷
```
"io"
"bufio"
```

We iterate across the input and ignore hashed lines. The remaining lines are converted to segments, which are printed after we've collected all of them. So we declare a variable for segments. And since segments are built from x- and y-positions, we also declare variables for them. Each position is either on the forward strand or not, and we reserve space for that information, too.

Note that we use a buffered reader rather than a buffered scanner to read the input. That's because there may be very long lines in our input and a reader imposes no limits on line length, while a scanner does.

392c ⟨*Scan file, Ch. 42* 392c⟩≡ (392a)
```
var xp, yp []int
var xf, yf []bool
var segments []Segment
line, err := reader.ReadString('\n')
for err == nil {
        if line[0] != '#' {
                ⟨Convert line to segments, Ch. 42 393a⟩
        }
        line, err = reader.ReadString('\n')
}
```

A segment consists of a pair of points, which we denote by a quartet of integers.

392d ⟨*Types, Ch. 42* 392d⟩≡ (389) 395g▷
```
type Segment struct {
        x1, y1, x2, y2 int
}
```

We split the line into its fields, extract the match length, and analyze the matches. Since matches consist of position and strand variables, we reset their storage beforehand.

393a ⟨*Convert line to segments, Ch. 42* 393a⟩≡ (392c)
```
fields := strings.Fields(line)
ml, err := strconv.Atoi(fields[0])
if err != nil { log.Fatalf("can't convert %q", fields[0]) }
matches := fields[3:]
⟨Reset coordinate variables, Ch. 42 393c⟩
for _, match := range matches {
        ⟨Extract x- and y-coordinates, Ch. 42 393d⟩
}
⟨Construct segments, Ch. 42 394c⟩
```

We import `strings`, `strconv`, and `log`.

393b ⟨*Imports, Ch. 42* 391b⟩+≡ (389) ◁392b 395b▷
```
"strings"
"strconv"
"log"
```

There are four coordinate variables denoting position and strand on the x- and y-axes.

393c ⟨*Reset coordinate variables, Ch. 42* 393c⟩≡ (393a)
```
xp = xp[:0]
yp = yp[:0]
xf = xf[:0]
yf = yf[:0]
```

As we saw in the Introduction, a match consists of a strand, a sequence ID, and a position, which is separated by a colon. We interpret a position on the first sequence as an x-coordinate, on the second sequence as a y-coordinate.

393d ⟨*Extract x- and y-coordinates, Ch. 42* 393d⟩≡ (393a)
```
sa := strings.Split(match, ":")
if len(sa) < 2 {
        m := "please stream 2 sequences though repeater"
        log.Fatal(m)
}
p, err := strconv.Atoi(sa[1])
if err != nil { log.Fatalf("can't convert %q", sa[1]) }
if match[0] == '1' || match[1] == '1' {
        ⟨Record position on x-axis, Ch. 42 394a⟩
} else {
        ⟨Record position on y-axis, Ch. 42 394b⟩
}
```

We record the position and the strandedness of a point on the x-axis.

394a ⟨*Record position on x-axis, Ch. 42* 394a⟩≡ (393d)
```
xp = append(xp, p)
if match[0] == 'f' {
        xf = append(xf, true)
} else {
        xf = append(xf, false)
}
```

We do the same for a point on the y-axis.

394b ⟨*Record position on y-axis, Ch. 42* 394b⟩≡ (393d)
```
yp = append(yp, p)
if match[0] == 'f' {
        yf = append(yf, true)
} else {
        yf = append(yf, false)
}
```

We have now established a set of positions on the x- and y-axes and their strandedness. We also know the length of the match. So we are now ready to construct the segments. We do this by forming all pairs of x- and y-positions. A segment might lean forward or backward, so we orient it before we store it.

394c ⟨*Construct segments, Ch. 42* 394c⟩≡ (393a)
```
for i, x1 := range xp {
        for j, y1 := range yp {
                y2 := y1 + ml - 1
                x2 := x1 + ml - 1
                s := Segment{x1: x1, y1: y1, x2: x2, y2: y2}
                ⟨Orient segment, Ch. 42 394d⟩
                segments = append(segments, s)
        }
}
```

If the positions of a match of length ℓ are both located on the forward strand or both are on the reverse strand, $x_2 \leftarrow x_1 + \ell - 1$. Such a match is a forward-leaning segment. Otherwise, the match leans backward, which is achieved by swapping x_1 and x_2.

394d ⟨*Orient segment, Ch. 42* 394d⟩≡ (394c)
```
if xf[i] != yf[j] {
        s.x1, s.x2 = s.x2, s.x1
}
```

We print the segments as four tab-delimited columns that we align using a tab writer. But recall that each segment may appear twice, so we remove duplicates before we print.

395a ⟨*Print segments, Ch. 42* 395a⟩≡ (392a)
```
⟨Set up tab writer, Ch. 42 395c⟩
⟨Remove duplicated segments, Ch. 42 395e⟩
for _, s := range segments {
        fmt.Fprintf(w, "%d\t%d\t%d\t%d\n", s.x1, s.y1,
                s.x2, s.y2)
}
w.Flush()
fmt.Printf("%s", buffer)
```

We import `fmt`.

395b ⟨*Imports, Ch. 42* 391b⟩+≡ (389) ◁393b 395d▷
```
"fmt"
```

We set up a tab writer with blank-separated columns.

395c ⟨*Set up tab writer, Ch. 42* 395c⟩≡ (395a)
```
var buf []byte
buffer := bytes.NewBuffer(buf)
w := new(tabwriter.Writer)
w.Init(buffer, 1, 0, 2, ' ', 0)
```

We import `bytes` and `tabwriter`.

395d ⟨*Imports, Ch. 42* 391b⟩+≡ (389) ◁395b 395f▷
```
"bytes"
"text/tabwriter"
```

To remove duplicate segments, we sort them and squeeze the duplicates from the sorted slice.

395e ⟨*Remove duplicated segments, Ch. 42* 395e⟩≡ (395a)
```
sort.Sort(SegmentSlice(segments))
j := 1
for i := 1; i < len(segments); i++ {
        if segments[i-1] != segments[i] {
                segments[j] = segments[i]
                j++
        }
}
if len(segments) > 0 {
        segments = segments[:j]
}
```

We import `sort`.

395f ⟨*Imports, Ch. 42* 391b⟩+≡ (389) ◁395d
```
"sort"
```

We declare the type `SegmentSlice`.

395g ⟨*Types, Ch. 42* 392d⟩+≡ (389) ◁392d
```
type SegmentSlice []Segment
```

We make `SegmentSlice` sortable by attaching the methods `Len`, `Swap`, and `Less`.

396a ⟨*Methods, Ch. 42* 396a⟩≡ (389)
```
func (s SegmentSlice) Len() int { return len(s) }
func (s SegmentSlice) Less(i, j int) bool {
        return s[i].x1 < s[j].x1
}
func (s SegmentSlice) Swap(i, j int) {
        s[i], s[j] = s[j], s[i]
}
```

We've finished `rep2plot`, time to test it.

Testing

The outline of our testing program has hooks for imports and the testing logic.

396b ⟨*rep2plot_test.go* 396b⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 42 396d⟩
)

func TestRep2plot(t *testing.T) {
        ⟨Testing, Ch. 42 396c⟩
}
```

We construct a set of tests and then run them.

396c ⟨*Testing, Ch. 42* 396c⟩≡ (396b)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 42 396e⟩
for i, test := range tests {
        ⟨Run test, Ch. 42 397a⟩
}
```

We import `exec`.

396d ⟨*Testing imports, Ch. 42* 396d⟩≡ (396b) 397b ▷
```
"os/exec"
```

We run two tests, one on data obtained just from the forward strands of two sequences (`test1.txt`, the other on data obtained from the forward and reverse strands (`test2.txt`).

396e ⟨*Construct tests, Ch. 42* 396e⟩≡ (396c)
```
test := exec.Command("./rep2plot", "test1.txt")
tests = append(tests, test)
test = exec.Command("./rep2plot", "test2.txt")
tests = append(tests, test)
```

When running a test, we compare the result we get with the result we want in files `r1.txt` and `r2.txt`.

397a ⟨*Run test, Ch. 42* 397a⟩≡ (396c)

```
get, err := test.Output()
if err != nil { t.Errorf("can't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("cant' open %q", f) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

397b ⟨*Testing imports, Ch. 42* 396d⟩+≡ (396b) ◁396d

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 43

Program `revComp`: Reverse-Complement DNA Sequence

DNA sequences are double stranded. The reverse strand is inferred from the forward strand by what's known as "reverse complementation". This is implemented in the program `revComp`. Its outline provides hooks for imports, variables, functions, and the logic of the main function.

399a ⟨*revComp.go* 399a⟩≡
```
package main

import (
        ⟨Imports, Ch. 43 399c⟩
)
⟨Variables, Ch. 43 400a⟩
⟨Functions, Ch. 43 400c⟩

func main() {
        ⟨Main function, Ch. 43 399b⟩
}
```

In the main function we prepare the `log` package, set the usage, scan the options, and scan the input.

399b ⟨*Main function, Ch. 43* 399b⟩≡ (399a)
```
util.PrepLog("revComp")
⟨Set usage, Ch. 43 399d⟩
⟨Parse options, Ch. 43 399f⟩
⟨Parse input, Ch. 43 400b⟩
```

We import `util`.

399c ⟨*Imports, Ch. 43* 399c⟩≡ (399a) 399e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the usage proper, a description of `revComp`, and an example command.

399d ⟨*Set usage, Ch. 43* 399d⟩≡ (399b)
```
u := "revComp [-h] [options] [files]"
d := "Reverse-complement DNA sequences."
e := "revComp *.fasta"
clio.Usage(u, d, e)
```

We import `clio`.

399e ⟨*Imports, Ch. 43* 399c⟩+≡ (399a) ◁399c 399g ▷
```
"github.com/evolbioinf/clio"
```

We parse the options and check whether the user requested the version.

399f ⟨*Parse options, Ch. 43* 399f⟩≡ (399b)
```
flag.Parse()
if *optV {
        util.PrintInfo("revComp")
}
```

We import `util`.

399g ⟨*Imports, Ch. 43* 399c⟩+≡ (399a) ◁399e 400d ▷
```
"flag"
```

We declare the options for version, `-v`, and for printing just the reverse, `-r`.

400a ⟨*Variables, Ch. 43* 400a⟩≡ (399a)
```
var optV = flag.Bool("v", false, "version")
var optR = flag.Bool("r", false, "reverse only")
```

When scanning the input files, we pass the reverse and complement options.

400b ⟨*Parse input, Ch. 43* 400b⟩≡ (399b)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optR)
```

In function `scan` we retrieve the option just passed by type assertion, then reverse or reverse-complement the sequence, and finally print it.

400c ⟨*Functions, Ch. 43* 400c⟩≡ (399a)
```
func scan(r io.Reader, args ...interface{}) {
        optR := args[0].(bool)
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                seq := sc.Sequence()
                ⟨Reverse or reverse-complement? Ch. 43 400e⟩
                fmt.Println(seq)
        }
}
```

We import `io`, `fasta`, and `fmt`.

400d ⟨*Imports, Ch. 43* 399c⟩+≡ (399a) ◁399g
```
"io"
"github.com/evolbioinf/fasta"
"fmt"
```

We decide whether to just reverse the sequence or compute the full reverse complement. In each case we add the corresponding information to the header.

400e ⟨*Reverse or reverse-complement? Ch. 43* 400e⟩≡ (400c)
```
seq.AppendToHeader(" - reverse")
if optR {
        seq.Reverse()
} else {
        seq.ReverseComplement()
        seq.AppendToHeader("_complement")
}
```

We're done with `revComp`, so let's test it. The outline for testing contains hooks for imports and the function to be tested.

401a ⟨*revComp_test.go* 401a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 43 401c⟩
)

func TestRevComp(t *testing.T) {
        ⟨Testing, Ch. 43 401b⟩
}
```

We apply `revComp` to `test.fasta` and compare what we get to what we want, which is stored in `res11.fasta`.

401b ⟨*Testing, Ch. 43* 401b⟩≡ (401a) 401d▷
```
cmd := exec.Command("./revComp", "test.fasta")
g, err := cmd.Output()
if err != nil {
        t.Errorf("coutdn't run %q\n", cmd)
}
w, err := ioutil.ReadFile("res1.fasta")
if err != nil {
        t.Error("couldn't open res1.fasta")
}
if !bytes.Equal(g, w) {
        t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

We import `exec`, `ioutil`, and `bytes`.

401c ⟨*Testing imports, Ch. 43* 401c⟩≡ (401a)
```
"os/exec"
"io/ioutil"
"bytes"
```

We repeat the test, only this time just compute the reverse.

401d ⟨*Testing, Ch. 43* 401b⟩+≡ (401a) ◁401b
```
cmd = exec.Command("./revComp", "-r", "test.fasta")
g, err = cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res2.fasta")
if err != nil {
        t.Error("couldnt' open res2.fasta")
}
if !bytes.Equal(g, w) {
        t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

Chapter 44

Program `rpois`: Draw Poisson-Distributed Random Variable

Introduction

The Poisson distribution gives the number of occurrences per unit time, N, of an event that can occur at any moment. The program `rpois` takes as input the mean number of occurrences, μ, and returns the corresponding N. This is computed by drawing random numbers until their product, $u_1 u_2 ... u_m \leq e^{-\mu}$. Then $N \leftarrow m - 1$ [23, p. 137].

Implementation

Our program outline contains hooks for imports and the logic of the main function.

403a ⟨*rpois.go* 403a⟩≡
```
package main
import (
        ⟨Imports, Ch. 44 403c⟩
)

func main() {
        ⟨Main function, Ch. 44 403b⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and compute the random variable n times.

403b ⟨*Main function, Ch. 44* 403b⟩≡ (403a)
```
util.PrepLog("rpois")
var n int
⟨Set usage, Ch. 44 403d⟩
⟨Declare options, Ch. 44 404a⟩
⟨Parse options, Ch. 44 404c⟩
for i := 0; i < n; i++ {
        ⟨Compute random variable, Ch. 44 405a⟩
}
```

We import `util`.

403c ⟨*Imports, Ch. 44* 403c⟩≡ (403a) 403e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts, the actual usage message, an explanation of the program's purpose, and an example command.

403d ⟨*Set usage, Ch. 44* 403d⟩≡ (403b)
```
u := "rpois [-h] [option]..."
p := "Draw Poisson-distributed random number."
e := "rpois -m 2"
clio.Usage(u, p, e)
```

We import `clio`.

403e ⟨*Imports, Ch. 44* 403c⟩+≡ (403a) ◁403c 404b ▷
```
"github.com/evolbioinf/clio"
```

The user can set the mean number of events, `-m`, the sample size (`-n`), and the seed for the random number generator, `-s`, which is a long integer. (S)he can also ask for the version.

404a ⟨*Declare options, Ch. 44* 404a⟩≡ (403b)
```
var optM = flag.Float64("m", 1, "mean")
var optN = flag.Int("n", 1, "sample size")
var optS = flag.Int64("s", 0, "seed for random number generator; " +
          "default: internal")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

404b ⟨*Imports, Ch. 44* 403c⟩+≡ (403a) ◁403e 404f▷
```
"flag"
```

We parse the options, set the sample size, and respond to `-v` and `-s`.

404c ⟨*Parse options, Ch. 44* 404c⟩≡ (403b)
```
flag.Parse()
n = *optN
⟨Respond to -v, Ch. 44 404d⟩
⟨Respond to -s, Ch. 44 404e⟩
```

If requested to do so, we print information about `rpois`.

404d ⟨*Respond to -v, Ch. 44* 404d⟩≡ (404c)
```
if *optV {
          util.PrintInfo("rpois")
}
```

If the user didn't set a seed, we take the number of nanoseconds since the beginning of the UNIX epoch. Then we seed the generator.

404e ⟨*Respond to -s, Ch. 44* 404e⟩≡ (404c)
```
seed := *optS
if seed == 0 {
          seed = time.Now().UnixNano()
}
source := rand.NewSource(seed)
r := rand.New(source)
```

We import `time` and `rand`.

404f ⟨*Imports, Ch. 44* 403c⟩+≡ (403a) ◁404b 405b▷
```
"time"
"math/rand"
```

Wit the preliminaries taken care of, we compute the random variable and print it.

405a ⟨*Compute random variable, Ch. 44* 405a⟩≡ (403b)
```
t := math.Exp(- *optM)
N := 0
pr := 1.0
un := r.Float64()
pr *= un
for pr >= t {
          un = r.Float64()
          pr *= un
          N++
}
fmt.Println(N)
```

We import `math` and `fmt`.

405b ⟨*Imports, Ch. 44* 403c⟩+≡ (403a) ◁404f
```
"math"
"fmt"
```

Our little program is written, so we test it.

Testing

The testing program has hooks for imports and the testing logic.

405c ⟨*rpois_test.go* 405c⟩≡
```
package main
import (
          "testing"
          ⟨Testing imports, Ch. 44 405e⟩
)

func TestRpois(t *testing.T) {
          ⟨Testing, Ch. 44 405d⟩
}
```

We set up the tests and run them.

405d ⟨*Testing, Ch. 44* 405d⟩≡ (405c)
```
var tests []*exec.Cmd
⟨Set up tests, Ch. 44 406a⟩
for i, test := range tests {
          ⟨Run test, Ch. 44 406c⟩
}
```

We import `exec`.

405e ⟨*Testing imports, Ch. 44* 405e⟩≡ (405c) 406b▷
```
"os/exec"
```

We set up five tests, which means we set up a slice of five seeds and construct the tests from them. We also set up a slice of the five results we want.

406a ⟨*Set up tests, Ch. 44* 406a⟩≡ (405d)

```
seeds := []int{ 1, 2, 3, 4, 5 }
for i := 0; i < len(seeds); i++ {
          s := strconv.Itoa(seeds[i])
          cmd := exec.Command("./rpois", "-s", s)
          tests = append(tests, cmd)
}
want := []string{ "3", "0", "3", "0", "3" }
```

We import `strconv`.

406b ⟨*Testing imports, Ch. 44* 405e⟩+≡ (405c) ◁405e

```
"strconv"
```

In an individual test we check we get what we want. The string we get is terminated by a newline, which we cut off.

406c ⟨*Run test, Ch. 44* 406c⟩≡ (405d)

```
get, err := test.Output()
get = get[0:len(get)-1]
if err != nil {
          t.Error(err.Error())
}
if string(get) != want[i] {
          t.Errorf("get: %s\nwant: %s\n",
                  get, want[i])
}
```

Chapter 45

Program `sass`: Simple Assembly

Figure 45.1: Assembly of two reads (**A**) that overlap by four nucleotides (**B**) into one contig (**C**)

Introduction

Forty years after its invention, shotgun sequencing remains the method of choice for sequencing any significant piece of DNA [28]. Shotgun sequencing consists of two steps, sequencing and assembly. `sass` is a simple assembly program. It reads a set of sequencing reads, which form the initial set of contigs. Then it aligns all pairs of contigs and merges the pair the best score. This is repeated until only one contig is left, or the remaining contigs don't overlap.

`sass` really *is* simple—it knows nothing about quality scores or paired-end reads to name but two glaring omissions compared to modern assemblers. To get a specific idea of what it *can* do, consider the two reads in Fig. 45.1A, which overlap as shown in Fig. 45.1B. `sass` takes these reads, merges them, and returns the resulting contig, Fig. 45.1C. While not much of a genome assembler, this is enough to demonstrate the principal of sequence assembly.

Implementation

The outline of `sass` has hooks for imports, types, methods, functions, and the logic of the main function.

408 ⟨*sass.go* 408⟩≡

```
package main

import (
        ⟨Imports, Ch. 45 409b⟩
)
⟨Types, Ch. 45 414c⟩
⟨Methods, Ch. 45 414d⟩
⟨Functions, Ch. 45 411b⟩
func main() {
        ⟨Main function, Ch. 45 409a⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files containing the sequencing reads. Then we calculate the assembly.

409a ⟨*Main function, Ch. 45* 409a⟩≡ (408)
```
util.PrepLog("sass")
⟨Set usage, Ch. 45 409c⟩
⟨Declare options, Ch. 45 409e⟩
⟨Parse options, Ch. 45 410c⟩
⟨Read sequencing reads, Ch. 45 410f⟩
⟨Calculate assembly, Ch. 45 411d⟩
```

We import `util`.

409b ⟨*Imports, Ch. 45* 409b⟩≡ (408) 409d ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `sass`, and an example command.

409c ⟨*Set usage, Ch. 45* 409c⟩≡ (409a)
```
u := "sass [option]... [file]..."
p := "Calculate assembly using a simple algorithm."
e := "sass -r reads.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

409d ⟨*Imports, Ch. 45* 409b⟩+≡ (408) ◁409b 409f ▷
```
"github.com/evolbioinf/clio"
```

In addition to the version, we declare two kinds of options, options that affect the alignment algorithm and options that affect the assembly proper.

409e ⟨*Declare options, Ch. 45* 409e⟩≡ (409a)
```
var optV = flag.Bool("v", false, "version")
⟨Declare alignment options, Ch. 45 409g⟩
⟨Declare assembly options, Ch. 45 410b⟩
```

We import `flag`.

409f ⟨*Imports, Ch. 45* 409b⟩+≡ (408) ◁409d 410e ▷
```
"flag"
```

An alignment is determined by its score scheme, which consists of residue scores and gap scores.

409g ⟨*Declare alignment options, Ch. 45* 409g⟩≡ (409e)
```
⟨Declare residue scores, Ch. 45 409h⟩
⟨Declare gap scores, Ch. 45 410a⟩
```

Residue scores are either match/mismatch or summarized in a score matrix.

409h ⟨*Declare residue scores, Ch. 45* 409h⟩≡ (409g)
```
var optA = flag.Float64("a",  1, "match")
var optI = flag.Float64("i", -3, "mismatch")
var optM = flag.String("m", "", "file containing score matrix")
```

Gaps are scored according to existence and length.

410a ⟨*Declare gap scores, Ch. 45* 410a⟩≡ (409g)
```
var optO = flag.Float64("o", -5, "gap opening")
var optE = flag.Float64("e", -2, "gap extension")
```

As to the actual assembly, the user can opt to include the reverse strand (`-r`), print the merge steps (`-M`), and set the score threshold (`-t`).

410b ⟨*Declare assembly options, Ch. 45* 410b⟩≡ (409e)
```
var optR = flag.Bool("r", false, "include reverse strand")
var optMM = flag.Bool("M", false, "print merge steps")
var optT = flag.Float64("t", 15.0, "score threshold")
```

We parse the options and respond to the version, as this stops the program. We also read the score matrix and collect the alignment parameters for easy handling.

410c ⟨*Parse options, Ch. 45* 410c⟩≡ (409a)
```
flag.Parse()
if *optV {
        util.PrintInfo("sass")
}
```
⟨*Get score matrix, Ch. 45* 410d⟩

A score matrix is either constructed from the match and mismatch scores or read from a file given by the user. For match/mismatch we allow any byte in the input.

410d ⟨*Get score matrix, Ch. 45* 410d⟩≡ (410c)
```
var sm *pal.ScoreMatrix
if *optM == "" {
        sm = pal.NewByteScoreMatrix(*optA, *optI)
} else {
        f, err := os.Open(*optM)
        if err != nil {
                log.Fatalf("couldn't open score matrix %q\n",
                        (*optM))
        }
        sm = pal.ReadScoreMatrix(f)
        f.Close()
}
```

We import `pal`, `os`, and `log`.

410e ⟨*Imports, Ch. 45* 409b⟩+≡ (408) ◁409f 411a▷
```
"github.com/evolbioinf/pal"
"os"
"log"
```

The remaining tokens on the command line are interpreted as the names of read files. We apply the function `scan` to each of them, which saves the reads in the initial slice of contigs.

410f ⟨*Read sequencing reads, Ch. 45* 410f⟩≡ (409a)
```
files := flag.Args()
contigs := make([]*fasta.Sequence, 0)
clio.ParseFiles(files, scan, &contigs)
```

We import `fasta`.

411a ⟨*Imports, Ch. 45* 409b⟩+≡ (408) ◁410e 411c▷
```
"github.com/evolbioinf/fasta"
```

Inside `scan`, we retrieve the slice of contigs, iterate over the sequences in the file, and store them.

411b ⟨*Functions, Ch. 45* 411b⟩≡ (408) 411e▷
```
func scan(r io.Reader, args ...interface{}) {
        contigs := args[0].(*([]*fasta.Sequence))
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                s := sc.Sequence()
                (*contigs) = append(*contigs, s)
        }
}
```

We import `io`.

411c ⟨*Imports, Ch. 45* 409b⟩+≡ (408) ◁411a 411f▷
```
"io"
```

The assembly is calculated by repeatedly finding the best pair of alignments and merging them into contigs. After the last merger, we sort the remaining contigs by length and print them. We delegate the search for the best alignment to the function `bestAl`.

411d ⟨*Calculate assembly, Ch. 45* 411d⟩≡ (409a)
```
i, j, bal := bestAl(contigs, sm, *optR, *optO, *optE)
for len(contigs) > 1 && bal.Score() >= *optT  {
        ⟨Merge contigs i and j, Ch. 45 413a⟩
        i, j, bal = bestAl(contigs, sm, *optR, *optO, *optE)
}
⟨Sort contigs by length, Ch. 45 414b⟩
⟨Print contigs, Chr. 45 414f⟩
```

Inside `bestAl`, we set up variables to hold the indexes of the sequences that make up the best alignment and the actual alignment. Then we iterate over all piaris of contigs and return the best alignment and its indexes.

411e ⟨*Functions, Ch. 45* 411b⟩+≡ (408) ◁411b 413d▷
```
func bestAl(contigs []*fasta.Sequence, sm *pal.ScoreMatrix,
        rev bool, optO, optE float64) (i, j int,
        oal *pal.OverlapAlignment) {
        var mi, mj int
        var mo *pal.OverlapAlignment
        ms := -1.0
        ⟨Iterate over pairs of contigs, Ch. 45 412a⟩
        return mi, mj, mo
}
```

We import `pal`.

411f ⟨*Imports, Ch. 45* 409b⟩+≡ (408) ◁411c 414e▷
```
"github.com/evolbioinf/pal"
```

We iterate over all pairs of contigs and calculate an alignment for each. If the user opted to include the reverse strand, we also do that. You might wonder why inclusion of the reverse strand is not the default behavior. This is to also allow protein sequences to be assembled, which isn't a standard use case but might be interesting in demos.

412a ⟨*Iterate over pairs of contigs, Ch. 45* 412a⟩≡ (411e)

```
for i := 0; i < len(contigs); i++ {
        for j := i+1; j < len(contigs); j++ {
                ⟨Align forward, Ch. 45 412b⟩
                if rev {
                        ⟨Align reverse, Ch. 45 412d⟩
                }
        }
}
```

We calculate the overlap alignment of the forward strand and check its score.

412b ⟨*Align forward, Ch. 45* 412b⟩≡ (412a)

```
oal := pal.NewOverlapAlignment(contigs[i], contigs[j],
        sm, optO, optE)
oal.Align()
⟨Check score, Ch. 45 412c⟩
```

If the score is better than the previous best score, we store the alignment.

412c ⟨*Check score, Ch. 45* 412c⟩≡ (412)

```
if oal.Score() > ms {
        mo = oal
        mi = i
        mj = j
        ms = oal.Score()
}
```

Similarly, we align the reverse strand of the shorter contig and check for a new maximum score.

412d ⟨*Align reverse, Ch. 45* 412d⟩≡ (412a)

```
if len(contigs[i].Data()) < len(contigs[j].Data()) {
        contigs[i].ReverseComplement()
} else {
        contigs[j].ReverseComplement()
}
oal = pal.NewOverlapAlignment(contigs[i], contigs[j],
        sm, optO, optE)
oal.Align()
⟨Check score, Ch. 45 412c⟩
```

To merge two contigs, we generate the merged contig, store it, and remove the contigs just merged.

413a ⟨*Merge contigs i and j, Ch. 45* 413a⟩≡ (411d)
```
⟨Generate merged contig, Ch. 45 413b⟩
if *optMM {
        ⟨Print merged contig, Ch. 45 413c⟩
}
⟨Store merged contig, Ch. 45 413e⟩
⟨Remove merged contigs, Ch. 45 414a⟩
```

We generate the merged contig from the raw alignment data of the best alignment.

413b ⟨*Generate merged contig, Ch. 45* 413b⟩≡ (413a)
```
a1, a2 := bal.RawAlignment()
var m []byte
for i, c := range a1 {
        if c != '-' {
                m = append(m, c)
        } else {
                m = append(m, a2[i])
        }
}
```

We print the cleaned source contigs and their merger.

413c ⟨*Print merged contig, Ch. 45* 413c⟩≡ (413a)
```
s1 := string(clean(a1))
s2 := string(clean(a2))
s3 := string(m)
fmt.Println(s1, s2, s3)
```

In the function `clean` we generate a version of the sequence without flanking gaps.

413d ⟨*Functions, Ch. 45* 411b⟩+≡ (408) ◁411e
```
func clean(b []byte) []byte {
        n := 0
        for i := 0; i < len(b); i++ {
                if b[i] != '-' {
                        b[n] = b[i]
                        n++
                }
        }
        b = b[:n]
        return b
}
```

We convert the merged string into a new contig and append it to the list of contigs. We leave the header of the new contig blank for now.

413e ⟨*Store merged contig, Ch. 45* 413e⟩≡ (413a)
```
contig := fasta.NewSequence("", m)
contigs = append(contigs, contig)
```

We remove the contigs we've just merged.

414a ⟨*Remove merged contigs, Ch. 45* 414a⟩≡ (413a)
```
n := 0
for k := 0; k < len(contigs); k++ {
        if k != i && k != j {
                contigs[n] = contigs[k]
                n++
        }
}
contigs = contigs[:n]
```

To sort the contigs, we cast them to a sortable type and apply `Sort`.

414b ⟨*Sort contigs by length, Ch. 45* 414b⟩≡ (411d)
```
sc := sortableContigs(contigs)
sort.Sort(sc)
```

We declare `sortableContigs`.

414c ⟨*Types, Ch. 45* 414c⟩≡ (408)
```
type sortableContigs []*fasta.Sequence
```

We implement the three methods of the `Sort` interface on `sortableContigs`, `Len`, `Less`, and `Swap`.

414d ⟨*Methods, Ch. 45* 414d⟩≡ (408)
```
func (s sortableContigs) Len() int {
        return len(s)
}
func (s sortableContigs) Less(i, j int) bool {
        return len(s[i].Data()) < len(s[j].Data())
}
func (s sortableContigs) Swap(i, j int) {
        s[i], s[j] = s[j], s[i]
}
```

We import `Sort`.

414e ⟨*Imports, Ch. 45* 409b⟩+≡ (408) ◁411f 415a▷
```
"sort"
```

We name the sorted contigs, contig1, contig2, and so on. If a read has not been incorporated into a contig, we just leave the header as is. Then we print the contigs in reverse order.

414f ⟨*Print contigs, Chr. 45* 414f⟩≡ (411d)
```
nc := 0
for i := len(sc)-1; i >= 0; i-- {
        if len(sc[i].Header()) == 0 {
                sc[i].AppendToHeader("Contig_" +
                        strconv.Itoa(nc+1))
                nc++
        }
        fmt.Println(sc[i])
}
```

We import `strcovn` and `fmt`.

415a ⟨*Imports, Ch. 45* 409b⟩+≡ (408) ◁414e
```
"strconv"
"fmt"
```

We've finished writing `sass`, let's test it.

Testing

The testing code for `sass` contains hooks for imports and the testing logic.

415b ⟨*sass_test.go* 415b⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 45 415d⟩
)

func TestSass(t *testing.T) {
        ⟨Testing, Ch. 45 415c⟩
}
```

To test `sass`, we construct a set of tests and then run each one in a loop.

415c ⟨*Testing, Ch. 45* 415c⟩≡ (415b)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 45 415e⟩
for i, test := range tests {
        ⟨Run test, Ch. 45 416a⟩
}
```

We import `exec`.

415d ⟨*Testing imports, Ch. 45* 415d⟩≡ (415b) 416b▷
```
"os/exec"
```

We construct a test for a default run and for each of the three options concerned with the assembly itself, print the merge steps (`-M`), include the reverse strand (`-r`), and set the score threshold (`-t`). Each test is run on the same set of three fragments in `f.fasta`.

415e ⟨*Construct tests, Ch. 45* 415e⟩≡ (415c)
```
f := "f.fasta"
test := exec.Command("./sass", f)
tests = append(tests, test)
test = exec.Command("./sass", "-r", f)
tests = append(tests, test)
test = exec.Command("./sass", "-r", "-M", f)
tests = append(tests, test)
test = exec.Command("./sass", "-r", "-t", "20", f)
tests = append(tests, test)
```

We run the test and compare the result we get to the result we want, which is contained in files `r1.txt`, `r2.txt`, and so on.

416a ⟨*Run test, Ch. 45* 416a⟩≡ (415c)

```
get, err := test.Output()
if err != nil {
          t.Errorf("couldn't run %q", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
          t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

416b ⟨*Testing imports, Ch. 45* 415d⟩+≡ (415b) ◁415d

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 46

Program `sblast`: Simple BLAST

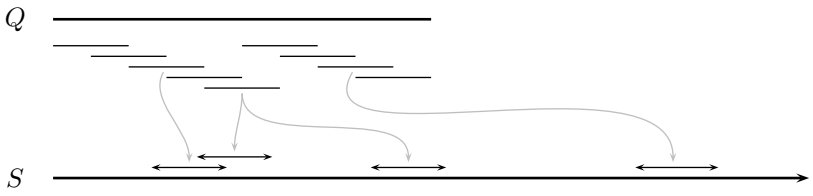


Figure 46.1: Cartoon of the BLAST algorithm.

Introduction

BLAST calculates local alignments between pairs of sequences and is used extensively in molecular biology to annotate sequences. The members of a sequence pair aligned with BLAST are called query and subject, where the query is searched in the subject. The search algorithm has three steps, division of the query into short, overlapping words, w, search for the words in the subject, and extension of matches into alignments. These three steps are illustrated in Figure 46.1 and we implement them in a simple BLAST program for DNA sequences, `sblast`.

Before we write any code, let's look at the three steps of the algorithm in a bit more detail starting with the construction of the word list. Let `GTCGA` be our query and the word length $w = 4$, then the word list is $\{$`GTC`, `TCG`, `CGA`$\}$. In real implementations, w is typically at least 11 for DNA sequences. To emphasize the importance of the word list in the BLAST algorithm, the user of `sblast` can print it out for inspection.

The query words are looked up in the subject by exact matching using a keyword tree. This is a tree structure built from the query words. As illustrated in Chapter 12, its construction takes some effort. To persuade the user of `sblast` that this effort is worth while, we also implement naïve matching as an alternative.

Each match of a query word in the subject is extended to the left and the right until the score of the alignment doesn't grow any further. Now, a word might be flanked by a mismatch, in which case the score drops on the first extension, but clearly we shouldn't give up immediately. So there is a maximum number of extension steps we are willing to wait for the last maximum score to improve until we give up and fall back to the position that generated the maximum. We call this the number of idle extension steps.

This gives us enough understanding of BLAST to get coding.

Implementation

Our program outline contains hooks for imports, types, methods, functions, and the logic of the main function.

418 ⟨*sblast.go* 418⟩≡

```
package main

import (
        ⟨Imports, Ch. 46 419b⟩
)
```

```
⟨Types, Ch. 46 420c⟩
⟨Methods, Ch. 46 428b⟩
⟨Functions, Ch. 46 421c⟩
func main() {
        ⟨Main function, Ch. 46 419a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input files.

419a ⟨*Main function, Ch. 46* 419a⟩≡ (418)
```
util.PrepLog("sblast")
⟨Set usage, Ch. 46 419c⟩
⟨Declare options, Ch. 46 419e⟩
⟨Parse options, Ch. 46 420a⟩
⟨Parse input files, Ch. 46 421a⟩
```

We import `util`.

419b ⟨*Imports, Ch. 46* 419b⟩≡ (418) 419d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `sblast`, and an example command.

419c ⟨*Set usage, Ch. 46* 419c⟩≡ (419a)
```
u := "sblast [-h] [option]... query.fasta [subject.fasta]..."
p := "Carry out a simple version of BLAST."
e := "sblast query.fasta subject.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

419d ⟨*Imports, Ch. 46* 419b⟩+≡ (418) ◁419b 419f▷
```
"github.com/evolbioinf/clio"
```

Apart from help (`-h`), which is already given by the `flag` package, we provide eight additional options. The algorithm is specified by match and mismatch scores, the word length, and the maximum number of idle extension steps. There is a threshold score, below which an alignment is not printed. The matching method may be switched to naïve and the user can print the word list. These options and their default values are listed in Table 46.1. Wherever I could, I took the defaults from BLAST.

419e ⟨*Declare options, Ch. 46* 419e⟩≡ (419a)
```
var optA = flag.Float64("a", 1.0, "match")
var optI = flag.Float64("i", -3.0, "mismatch")
var optW = flag.Int("w", 11, "word length")
var optS = flag.Int("s", 30, "maximum number " +
          "of idle extension steps")
var optT = flag.Float64("t", 50.0, "threshold score")
var optN = flag.Bool("n", false, "naive matching")
var optL = flag.Bool("l", false, "print word list")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

419f ⟨*Imports, Ch. 46* 419b⟩+≡ (418) ◁419d 421b▷
```
"flag"
```

Table 46.1: User options of `sblast` and their defaults.

#	Option	Meaning	Default
1	`-a`	match	1
2	`-i`	mismatch	-3
3	`-w`	word length	11
4	`-s`	idle extension steps	30
5	`-t`	threshold score	50
6	`-n`	naïve matching	false
7	`-l`	print word list	false
8	`-v`	print version	false

We parse the options and respond to `-v`, as this would terminate the program. Then we collect the remaining option values.

420a ⟨*Parse options, Ch. 46* 420a⟩≡ (419a)
```
flag.Parse()
if *optV {
        util.PrintInfo("sblast")
}
```
⟨*Collect option values, Ch. 46* 420b⟩

There are seven options we later pass to the BLAST algorithm. To make this easy, we collect them in the variable `opts`.

420b ⟨*Collect option values, Ch. 46* 420b⟩≡ (420a)
```
opts := new(Opts)
opts.a = *optA
opts.i = *optI
opts.w = *optW
opts.s = *optS
opts.t = *optT
opts.n = *optN
opts.l = *optL
```

We declare the type `Opts`.

420c ⟨*Types, Ch. 46* 420c⟩≡ (418) 423d ▷
```
type Opts struct {
        a, i, t float64
        w, s int
        n, l bool
}
```

The remaining tokens on the command line are taken as the names of input files. The first of these contains the query sequences, any subsequent file the subject sequences. If there is no query file, we bail with a friendly message. If there is, we call `ParseFiles`, which has as first parameter the names of the subject files, and second parameter the function `scan`. This function is applied to each subject file and takes as arguments the options and the query file. It also takes as argument a tab writer to align the columns of the output. This is initialized with the column headers and flushed after the run is finished.

421a ⟨*Parse input files, Ch. 46* 421a⟩≡ (419a)
```
files := flag.Args()
if len(files) == 0 {
        log.Fatal("please provide a query")
}
out := tabwriter.NewWriter(os.Stdout, 2, 1, 2, ' ' ,0)
if !opts.l {
        fmt.Fprintf(out, "#qa\tsa\tqs\tqe\tss\tse\tscore\n")
} else {
        fmt.Fprintf(out, "#qa\tn\tword\n")
}
clio.ParseFiles(files[1:], scan, opts, files[0], out)
out.Flush()
```

We import `tabwriter` and `fmt`.

421b ⟨*Imports, Ch. 46* 419b⟩+≡ (418) ◁419f 421d▷
```
"text/tabwriter"
"fmt"
```

Inside `scan`, we retrieve the arguments, iterate across the subject sequences, and for each one iterate across the queries.

421c ⟨*Functions, Ch. 46* 421c⟩≡ (418) 423a▷
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 46 421e⟩
        sScanner := fasta.NewScanner(r)
        for sScanner.ScanSequence() {
                subject := sScanner.Sequence()
                ⟨Iterate across queries, Ch. 46 422a⟩
        }
}
```

We import `io` and `fasta`.

421d ⟨*Imports, Ch. 46* 419b⟩+≡ (418) ◁421b 422b▷
```
"io"
"github.com/evolbioinf/fasta"
```

The options and the queries are retrieved by type assertion.

421e ⟨*Retrieve arguments, Ch. 46* 421e⟩≡ (421c)
```
opts := args[0].(*Opts)
qName := args[1].(string)
out := args[2].(*tabwriter.Writer)
```

We open the query file and analyze each sequence it contains.

422a ⟨*Iterate across queries, Ch. 46* 422a⟩≡ (421c)
```
qFile, err := os.Open(qName)
if err != nil {
        log.Fatalf("couldn't open %s\n", qName)
}
defer qFile.Close()
qScanner := fasta.NewScanner(qFile)
for qScanner.ScanSequence() {
        query := qScanner.Sequence()
        ⟨Analyze query, Ch. 46 422c⟩
}
```

We import `os` and `log`.

422b ⟨*Imports, Ch. 46* 419b⟩+≡ (418) ◁421d 422e▷
```
"os"
"log"
```

A query either gets its word list printed or is aligned to the subject.

422c ⟨*Analyze query, Ch. 46* 422c⟩≡ (422a)
```
if opts.l {
        ⟨Print word list, Ch. 46 422d⟩
} else {
        ⟨Align query, Ch. 46 423b⟩
}
```

A word list is started by the header of the sequence. The list itself consists of numbered words, one per line. We only write the words on the forward strand. Since we might write the word lists for more than one query, we extract the query accession as the first token on the command line.

422d ⟨*Print word list, Ch. 46* 422d⟩≡ (422c)
```
words := getWords(query, opts.w)
qa := strings.Fields(query.Header())[0]
for i, word := range words {
        fmt.Fprintf(out, "%s\t%d\t%s\n", qa, i+1, word)
}
```

We import `strings`.

422e ⟨*Imports, Ch. 46* 419b⟩+≡ (418) ◁422b 425c▷
```
"strings"
```

The function `getWords` takes as argument a sequence and a word length and returns all words of that length.

423a ⟨*Functions, Ch. 46* 421c⟩+≡ (418) ◁421c 423c▷
```
func getWords(seq *fasta.Sequence, w int) []string {
        var words []string
        d := seq.Data()
        l := len(d)
        for i := 0; i <= l - w; i++ {
                word := string(d[i:i+w])
                words = append(words, word)
        }
        return words
}
```

We align the query first along its forward strand, then along its reverse strand. We print the resulting alignments.

423b ⟨*Align query, Ch. 46* 423b⟩≡ (422c)
```
forward := true
alignments := align(query, subject, opts, forward)
query.ReverseComplement()
forward = false
a := align(query, subject, opts, forward)
alignments = append(alignments, a...)
⟨Print alignments, Ch. 46 430b⟩
```

Inside the function `align`, we calculate the alignments and return them.

423c ⟨*Functions, Ch. 46* 421c⟩+≡ (418) ◁423a
```
func align(query, subject *fasta.Sequence,
        opts *Opts, forward bool) []Alignment {
        var alignments []Alignment
        ⟨Calculate alignments, Ch. 46 423e⟩
        return alignments
}
```

An alignment consists of query start and end, subject start and end, a score, and a strand.

423d ⟨*Types, Ch. 46* 420c⟩+≡ (418) ◁420c 428a▷
```
type Alignment struct {
        qs, qe, ss, se int
        score float64
        forward bool
}
```

As shown in Figure 46.1, we initialize alignments through exact matching and then extend the matches to the left and to the right. Then we filter the alignments and sort them by score.

423e ⟨*Calculate alignments, Ch. 46* 423e⟩≡ (423c)
```
⟨Exact matching, Ch. 46 424a⟩
⟨Extend alignments, Ch. 46 425f⟩
⟨Filter alignments, Ch. 46 427b⟩
⟨Sort alignments by score, Ch. 46 429d⟩
```

As shown in Figure 46.1, in the exact matching phase of the algorithm query words are located in the subject. We store these matches as mini alignments, which we either find by naïve matching or by matching with a keyword tree.

424a ⟨*Exact matching, Ch. 46* 424a⟩≡ (423e)

```
if opts.n {
        ⟨Naïve exact matching, Ch. 46 424b⟩
} else {
        ⟨Exact match with keyword tree, Ch. 46 424d⟩
}
```

In naïve exact matching, we iterate over the query to generate the patterns and then look for them in the subject.

424b ⟨*Naïve exact matching, Ch. 46* 424b⟩≡ (424a)

```
q := query.Data()
m := len(q)
s := subject.Data()
n := len(s)
w := opts.w
for i := 0; i < m - w; i++ {
        p := q[i:i+w]
        for j := 0; j < n - w; j++ {
                ⟨Look for pattern, Ch. 46 424c⟩
        }
}
```

We break off the search for a pattern at the first mismatch we encounter.

424c ⟨*Look for pattern, Ch. 46* 424c⟩≡ (424b)

```
var k int
for k = 0; k < w; k++ {
        if s[j+k] != p[k] {
                break
        }
}
if k == opts.w {
        a := Alignment{qs: i, qe: i+w-1, ss: j, se: j+w-1,
                score: float64(w) *opts.a, forward: forward}
        alignments = append(alignments, a)
}
```

With a keyword tree, we look for all patterns at the same time. So we construct the patterns and their keyword tree, search for matches in the subject, and store the matches as alignments.

424d ⟨*Exact match with keyword tree, Ch. 46* 424d⟩≡ (424a)

```
⟨Construct patterns, Ch. 46 425a⟩
⟨Construct keyword tree, Ch. 46 425b⟩
⟨Search with keyword tree, Ch. 46 425d⟩
⟨Convert matches to alignments, Ch. 46 425e⟩
```

We store the patterns as a string slice.

425a ⟨*Construct patterns, Ch. 46* 425a⟩≡ (424d)
```
var patterns []string
q := query.Data()
m := len(q)
w := opts.w
for i := 0; i <= m-w; i++ {
        p := string(q[i:i+w])
        patterns = append(patterns, p)
}
```

The keyword tree is constructed by a function call.

425b ⟨*Construct keyword tree, Ch. 46* 425b⟩≡ (424d)
```
tree := kt.NewKeywordTree(patterns)
```

We import `kt`.

425c ⟨*Imports, Ch. 46* 419b⟩+≡ (418) ◁422e 427f▷
```
"github.com/evolbioinf/kt"
```

The search with the keyword tree is also a single function call.

425d ⟨*Search with keyword tree, Ch. 46* 425d⟩≡ (424d)
```
matches := tree.Search(subject.Data(), patterns)
```

We iterate over the matches and convert them to our proto alignments.

425e ⟨*Convert matches to alignments, Ch. 46* 425e⟩≡ (424d)
```
for _, m := range matches {
        qs := m.Pattern
        ss := m.Position
        qe := qs + w - 1
        se := ss + w - 1
        sc := float64(w) * opts.a
        a := Alignment{qs: qs, ss: ss, qe: qe,
                se: se, score: sc, forward: forward}
        alignments = append(alignments, a)
}
```

We extend each alignment seed by walking to the left and to the right.

425f ⟨*Extend alignments, Ch. 46* 425f⟩≡ (423e)
```
q := query.Data()
m := len(q)
s := subject.Data()
n := len(s)
for i, _ := range alignments {
        ⟨Walk left, Ch. 46 426a⟩
        ⟨Walk right, Ch. 46 426d⟩
}
```

We walk left until we run out of query or subject, or until we run out of idle steps. In each step we compare the current pair of residues and ask whether we should adjust the alignment start.

426a ⟨*Walk left, Ch. 46* 426a⟩≡ (425f)
```
cq := alignments[i].qs - 1
cs := alignments[i].ss - 1
score := alignments[i].score
is := 0
for cq >= 0 && cs >= 0 && is <= opts.s {
        ⟨Compare current pair of residues, Ch. 46 426b⟩
        ⟨Adjust alignment start? Ch. 46 426c⟩
        cq--
        cs--
}
```

If a pair of residues is identical, we add the match score to the current score, otherwise we add the mismatch score.

426b ⟨*Compare current pair of residues, Ch. 46* 426b⟩≡ (426)
```
if q[cq] == s[cs] {
        score += opts.a
} else {
        score += opts.i
}
```

If the alignment score has grown, we shift the alignment start to the left, set the new maximum score, and reset the number of idle steps to zero. Otherwise, we've just carried out an idle step.

426c ⟨*Adjust alignment start? Ch. 46* 426c⟩≡ (426a)
```
if score > alignments[i].score {
        alignments[i].score = score
        alignments[i].qs = cq
        alignments[i].ss = cs
        is = 0
} else {
        is++
}
```

Walking to the right is similar as walking to the left, except that now we ask whether we should adjust the alignment end.

426d ⟨*Walk right, Ch. 46* 426d⟩≡ (425f)
```
cq = alignments[i].qe + 1
cs = alignments[i].se + 1
score = alignments[i].score
is = 0
for cq < m && cs < n && is <= opts.s {
        ⟨Compare current pair of residues, Ch. 46 426b⟩
        ⟨Adjust alignment end? Ch. 46 427a⟩
        cq++
        cs++
}
```

If the score has improved, we extend the alignment to the right, set the new score, and reset the number of idle steps to zero. Otherwise, we increment the idle steps.

427a ⟨*Adjust alignment end? Ch. 46* 427a⟩≡ (426d)
```
if score > alignments[i].score {
          alignments[i].score = score
          alignments[i].qe = cq
          alignments[i].se = cs
          is = 0
} else {
          is++
}
```

We filter the alignments by removing those with low scores. In addition, words that land in the same homologous region on the subject may generate alignments that are contained in each other. We remove these redundant alignments.

427b ⟨*Filter alignments, Ch. 46* 427b⟩≡ (423e)
⟨*Remove low-scoring alignments, Ch. 46* 427c⟩
⟨*Remove redundant alignments, Ch. 46* 427d⟩

We keep only alignments with a score greater or equal to the threshold score.

427c ⟨*Remove low-scoring alignments, Ch. 46* 427c⟩≡ (427b)
```
i := 0
max := -1.0
for _, al := range alignments {
          if al.score >= opts.t {
                    if max < al.score { max = al.score }
                    alignments[i] = al
                    i++
          }
}
alignments = alignments[:i]
```

Redundant alignments tend to either share a start position or an end position. So we sort by start position as primary key and reduce runs of identical start positions to the first element. This means we should sort alignments with identical start positions by score in reverse order. Then repeat for the end position.

427d ⟨*Remove redundant alignments, Ch. 46* 427d⟩≡ (427b)
⟨*Sort alignments by start, Ch. 46* 427e⟩
⟨*Delete alignments with identical start, Ch. 46* 428d⟩
⟨*Sort alignments by end, Ch. 46* 428e⟩
⟨*Delete alignments with identical end, Ch. 46* 429c⟩

We sort the alignments by their start positions using an alignment slice.

427e ⟨*Sort alignments by start, Ch. 46* 427e⟩≡ (427d)
```
sort.Sort(AlSliceStart(alignments))
```

We import `sort`.

427f ⟨*Imports, Ch. 46* 419b⟩+≡ (418) ◁425c
```
"sort"
```

We declare `AlSliceStart`.

428a ⟨*Types, Ch. 46* 420c⟩+≡ (418) ◁423d 428f▷
```
type AlSliceStart []Alignment
```

We implement the methods `Len`, `Less`, and `Swap` to make `AlSliceStart` sortable.

428b ⟨*Methods, Ch. 46* 428b⟩≡ (418) 429a▷
```
func (a AlSliceStart) Len() int {
        return len(a)
}
```
⟨*Implement* `Less` *for* `AlSliceStart`*, Ch. 46* 428c⟩
```
func (a AlSliceStart) Swap(i, j int) {
        a[i], a[j] = a[j], a[i]
}
```

We make sure that for alignments starting at the same position the highest scoring one comes first.

428c ⟨*Implement* `Less` *for* `AlSliceStart`*, Ch. 46* 428c⟩≡ (428b)
```
func (a AlSliceStart) Less(i, j int) bool {
        if a[i].ss == a[j].ss {
                return a[i].score > a[j].score
        } else {
                return a[i].ss < a[j].ss
        }
}
```

Wit the alignments sorted by their start and positions and scores, we can

428d ⟨*Delete alignments with identical start, Ch. 46* 428d⟩≡ (427d)
```
j := 0
if len(alignments) > 0 { j = 1 }
for i := 1; i < len(alignments); i++ {
        if alignments[i].ss != alignments[i-1].ss {
                alignments[j] = alignments[i]
                j++
        }
}
alignments = alignments[:j]
```

We repeat this procedure for the alignment ends and start again by sorting the alignments by their end positions using an alignment slice.

428e ⟨*Sort alignments by end, Ch. 46* 428e⟩≡ (427d)
```
sort.Sort(AlSliceEnd(alignments))
```

We declare `AlSliceEnd`.

428f ⟨*Types, Ch. 46* 420c⟩+≡ (418) ◁428a 429e▷
```
type AlSliceEnd []Alignment
```

We implement the methods `Len`, `Less`, and `Swap` to make `AlSliceStart` sortable.

429a ⟨*Methods, Ch. 46* 428b⟩+≡ (418) ◁428b 430a▷
```
func (a AlSliceEnd) Len() int {
        return len(a)
}
⟨Implement Less for AlSliceEnd, Ch. 46 429b⟩
func (a AlSliceEnd) Swap(i, j int) {
        a[i], a[j] = a[j], a[i]
}
```

We make sure that for alignments ending at the same position the highest scoring one comes first.

429b ⟨*Implement* `Less` *for* `AlSliceEnd`*, Ch. 46* 429b⟩≡ (429a)
```
func (a AlSliceEnd) Less(i, j int) bool {
        if a[i].se == a[j].se {
                return a[i].score > a[j].score
        } else {
                return a[i].se < a[j].se
        }
}
```

Wit the alignments sorted by their end and positions and scores, we can

429c ⟨*Delete alignments with identical end, Ch. 46* 429c⟩≡ (427d)
```
j = 0
if len(alignments) > 0 { j = 1 }
for i := 1; i < len(alignments); i++ {
        if alignments[i].se != alignments[i-1].se {
                alignments[j] = alignments[i]
                j++
        }
}
alignments = alignments[:j]
```

We sort the remaining alignments by score.

429d ⟨*Sort alignments by score, Ch. 46* 429d⟩≡ (423e)
```
sort.Sort(AlSliceScore(alignments))
```

We declare `AlSliceScore`.

429e ⟨*Types, Ch. 46* 420c⟩+≡ (418) ◁428f
```
type AlSliceScore []Alignment
```

We implement the `Sort` interface on `AlSliceScore` imposing an ascending order this time.

430a ⟨*Methods, Ch. 46* 428b⟩+≡ (418) ◁429a
```
func (a AlSliceScore) Len() int {
        return len(a)
}
func (a AlSliceScore) Less(i, j int) bool {
        return a[i].score > a[j].score
}
func (a AlSliceScore) Swap(i, j int) {
        a[i], a[j] = a[j], a[i]
}
```

The alignments are ready to be printed. Again, we extract the accessions from the header. Alignments on the reverse strand get their subject positions switched.

430b ⟨*Print alignments, Ch. 46* 430b⟩≡ (423b)
```
qa := strings.Fields(query.Header())[0]
sa := strings.Fields(subject.Header())[0]
for _, a := range alignments {
        if !a.forward {
                a.ss, a.se = a.se, a.ss
        }
        fmt.Fprintf(out, "%s\t%s\t%d\t%d\t%d\t%d\t%.1f\n",
                qa, sa, a.qs+1, a.qe+1, a.ss+1, a.se+1, a.score)
}
```

We have finished `sblast`, let's test it.

Testing

Our testing code has hooks for imports and the testing logic.

430c ⟨*sblast_test.go* 430c⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 46 431a⟩
)

func TestSblast(t *testing.T) {
        ⟨Testing, Ch. 46 430d⟩
}
```

We construct the tests and iterate over them.

430d ⟨*Testing, Ch. 46* 430d⟩≡ (430c)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 46 431b⟩
for i, test := range tests {
        ⟨Run test, Ch. 46 432b⟩
}
```

We import `exec`.

431a ⟨*Testing imports, Ch. 46* 431a⟩≡ (430c) 432c ▷
```
"os/exec"
```

We test the first seven options listed in Table 46.1.

431b ⟨*Construct tests, Ch. 46* 431b⟩≡ (430d)
```
⟨Test -a, Ch. 46 431c⟩
⟨Test -i, Ch. 46 431d⟩
⟨Test -w, Ch. 46 431e⟩
⟨Test -s, Ch. 46 431f⟩
⟨Test -t, Ch. 46 431g⟩
⟨Test -n, Ch. 46 431h⟩
⟨Test -l, Ch. 46 432a⟩
```

We set the match score from its default of 1 to 2. We use the file `test.fasta` as query and subject. It contains the *Adh* loci of *Drosophila melanogaster* and *D. guanche*.

431c ⟨*Test -a, Ch. 46* 431c⟩≡ (431b)
```
test := exec.Command("./sblast", "-a", "2",
          "test.fasta", "test.fasta")
tests = append(tests, test)
```

We set the mismatch score from default -3 to -2.

431d ⟨*Test -i, Ch. 46* 431d⟩≡ (431b)
```
test = exec.Command("./sblast", "-i", "-2",
          "test.fasta", "test.fasta")
tests = append(tests, test)
```

We set the word length from default 11 to 20.

431e ⟨*Test -w, Ch. 46* 431e⟩≡ (431b)
```
test = exec.Command("./sblast", "-w", "20",
          "test.fasta", "test.fasta")
tests = append(tests, test)
```

We reduce the maximum number of idle steps from default 30 to 20.

431f ⟨*Test -s, Ch. 46* 431f⟩≡ (431b)
```
test = exec.Command("./sblast", "-s", "20",
          "test.fasta", "test.fasta")
tests = append(tests, test)
```

We reduce the threshold score from 50 to 40.

431g ⟨*Test -t, Ch. 46* 431g⟩≡ (431b)
```
test = exec.Command("./sblast", "-t", "40",
          "test.fasta", "test.fasta")
tests = append(tests, test)
```

We switch from matching with a keyword tree to naïve matching.

431h ⟨*Test -n, Ch. 46* 431h⟩≡ (431b)
```
test = exec.Command("./sblast", "-n",
          "test.fasta", "test.fasta")
tests = append(tests, test)
```

We print the word list.

432a ⟨*Test -l, Ch. 46* 432a⟩≡ (431b)
```
test = exec.Command("./sblast", "-l",
          "test.fasta", "test.fasta")
tests = append(tests, test)
```

When running `sblast`, we compare what we get with what we want, which is contained in results files `r1.txt`, `r2.txt`, and so on.

432b ⟨*Run test, Ch. 46* 432b⟩≡ (430d)
```
get, err := test.Output()
if err != nil { t.Errorf("couldn't run %s\n", test) }
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("couldn't open %s\n", f) }
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n",
                  string(get), string(want))
}
```

We import `strconv`, `ioutil`, and `bytes`.

432c ⟨*Testing imports, Ch. 46* 431a⟩+≡ (430c) ◁431a
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 47

Program `sequencer`: Sequence DNA Sequences

Introduction

Sequencing machines generate reads of DNA sequences, which, depending on the sequencing technology, are usually only a few dozen to a few hundred nucleotides long. This is tiny compared to most chromosomes, which means that rather than reading a chromosome in one fell swoop, many reads have to be assembled into the underlying template. This sequencing technique is known as *shotgun sequencing*, and I think of the subsequent assembly as doing a giant jigsaw puzzle.

The popularity of shotgun sequencing has led to the development of a whole class of programs for carrying out genome assembly. To test such programs, it is handy to have a ready source of sequencing reads. The program `sequencer` generates such reads. It takes as input one ore more template sequences and sequences each one to the specified coverage. Sequencing can be single-end or paired-end, and the user can set both the read length and—in paired end sequencing—the insert length. Input sequences may be linear or circular.

Implementation

The outline of `sequencer` has hooks for imports, types, functions, and the logic of the main function.

434a ⟨*sequencer.go* 434a⟩≡
```
package main

import (
        ⟨Imports, Ch. 47 434c⟩
)
⟨Types, Ch. 47 436f⟩
⟨Functions, Ch. 47 437c⟩
func main() {
        ⟨Main function, Ch. 47 434b⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and parse the input sequences.

434b ⟨*Main function, Ch. 47* 434b⟩≡ (434a)
```
util.PrepLog("sequencer")
⟨Set usage, Ch. 47 435a⟩
⟨Declare options, Ch. 47 435c⟩
⟨Parse options, Ch. 47 436b⟩
⟨Parse input files, Ch. 47 437b⟩
```

We import `util`.

434c ⟨*Imports, Ch. 47* 434c⟩≡ (434a) 435b ▷
```
"github.com/evolbioinf/biobox/util"
```

Table 47.1: Options of `sequencer`.

#	Option	Meaning	Default
1	`-v`	version	false
2	`-c`	coverage	1
3	`-r`	mean read length	100
4	`-R`	standard deviation of read length	0
5	`-p`	paired end	false
6	`-i`	mean insert length	500
7	`-I`	standard deviation of insert length	0
8	`-e`	sequencing error	10^{-3}
9	`-s`	seed for random number generator	internal
10	`-o`	circular genome	linear
11	`-S`	shredder	

The usage consists of the actual usage message, an explanation of the purpose of `sequencer`, and an example command.

435a ⟨*Set usage, Ch. 47* 435a⟩≡ (434b)
```
u := "sequencer [-h] [option]... [foo.fasta]..."
p := "Simulate a DNA sequencing machine."
e := "sequencer -c 20 foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

435b ⟨*Imports, Ch. 47* 434c⟩+≡ (434a) ◁434c 435d▷
```
"github.com/evolbioinf/clio"
```

Apart from the version (`-v`), we declare options for the coverage, to set the mean read and insert length, paired-end *vs.* single-end, the error rate, a seed for the random number generator, whether the genome is circular, and whether or not `sequencer` works as a simple shredder. The eleven options are listed in Table 47.1. We begin by declaring the coverage and the read length, whether or not we are using paired-end sequencing, and the insert length. The standard deviation of the read and insert length is by default zero, that is, their length is constant, but the user can change that, in which their lengths are drawn from a normal distribution with the specified mean and standard deviation.

435c ⟨*Declare options, Ch. 47* 435c⟩≡ (434b) 436a▷
```
var optC = flag.Float64("c", 1.0, "coverage")
var optR = flag.Float64("r", 100.0, "mean read length")
var optRR = flag.Float64("R", 0.0, "standard deviation of " +
          "read length")
var optP = flag.Bool("p", false, "paired end")
var optI = flag.Float64("i", 500.0, "mean insert length")
var optII = flag.Float64("I", 0.0, "standard deviation of " +
          "insert length")
```

We import `flag`.

435d ⟨*Imports, Ch. 47* 434c⟩+≡ (434a) ◁435b 436e▷
```
"flag"
```

We declare the sequencing error, the seed for the random number generator, the option for circular genomes, for shredder mode, and the version.

436a ⟨*Declare options, Ch. 47* 435c⟩+≡ (434b) ◁435c
```
var optE = flag.Float64("e", 0.001, "error rate")
var optS = flag.Int("s", 0, "seed for random number generator")
var optO = flag.Bool("o", false, "circular template")
var optSS = flag.Bool("S", false, "shredder - forward strand only")
var optV = flag.Bool("v", false, "version")
```

We parse the options and first respond to `-v`, as this stops the program. We also seed the random number generator and collect the options, so that we can conveniently pass them around.

436b ⟨*Parse options, Ch. 47* 436b⟩≡ (434b)
```
flag.Parse()
⟨Respond to -v, Ch. 47 436c⟩
⟨Seed random number generator, Ch. 47 436d⟩
⟨Collect options, Ch. 47 437a⟩
```

We write the version, if desired.

436c ⟨*Respond to -v, Ch. 47* 436c⟩≡ (436b)
```
if *optV {
        util.PrintInfo("sequencer")
}
```

We seed the random number generator either from the seed provided by the user or from the current time.

436d ⟨*Seed random number generator, Ch. 47* 436d⟩≡ (436b)
```
seed := int64(*optS)
if seed == 0 {
        seed = time.Now().UnixNano()
}
rn := rand.New(rand.NewSource(int64(seed)))
```

We import `time` and `rand`.

436e ⟨*Imports, Ch. 47* 434c⟩+≡ (434a) ◁435d 437d▷
```
"time"
"math/rand"
```

To collect the options, we declare the structure `opts`, whose fields reflect the names of the options.

436f ⟨*Types, Ch. 47* 436f⟩≡ (434a)
```
type opts struct {
        c, r, R, i, I, e float64
        p, o, S bool
}
```

We initialize a variable of type `opts` and collect the options.

437a ⟨*Collect options, Ch. 47* 437a⟩≡ (436b)
```
op := new(opts)
op.c = *optC
op.r = *optR
op.R = *optRR
op.i = *optI
op.I = *optII
op.e = *optE
op.p = *optP
op.o = *optO
op.S = *optSS
```

The remaining tokens on the command line are taken as the names of input files. We parse each one of them in turn using the function `scan`, which takes the options and the random number generator as arguments.

437b ⟨*Parse input files, Ch. 47* 437b⟩≡ (434b)
```
files := flag.Args()
clio.ParseFiles(files, scan, op, rn)
```

Inside `scan`, we retrieve the arguments just passed and sequence each entry in the FASTA file.

437c ⟨*Functions, Ch. 47* 437c⟩≡ (434a) 439c ▷
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 47 437e⟩
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                seq := sc.Sequence()
                ⟨Carry out sequencing, Ch. 47 438a⟩
        }
}
```

We import `io` and `fasta`.

437d ⟨*Imports, Ch. 47* 434c⟩+≡ (434a) ◁436e 438b ▷
```
"io"
"github.com/evolbioinf/fasta"
```

We retrieve the options and the random number generator using type assertion.

437e ⟨*Retrieve arguments, Ch. 47* 437e⟩≡ (437c)
```
op := args[0].(*opts)
rn := args[1].(*rand.Rand)
```

We prepare the sequence, compute coverage as the number of nucleotides to be sequenced, and declare variables for the number of nucleotides sequenced and for counting the reads. We also construct a buffer for writing the reads. Then we iterate until the number of nucleotides sequenced exceeds the coverage. Inside this loop we sequence according to the mode chosen by the user. After the loop we flush the buffer.

438a ⟨*Carry out sequencing, Ch. 47* 438a⟩≡ (437c)
```
n := len(seq.Data())
⟨Prepare sequence, Ch. 47 438c⟩
cov := int(math.Round(float64(n) * op.c))
var ns, rc int
w := bufio.NewWriter(os.Stdout)
for ns < cov {
        ⟨Sequence according to mode, Ch. 47 438d⟩
}
w.Flush()
```

We import `bufio`, `os`, and `math`.

438b ⟨*Imports, Ch. 47* 434c⟩+≡ (434a) ◁437d 439b▷
```
"bufio"
"os"
"math"
```

In preparation of our sequencing run, we store the forward and the reverse strand in a slice of byte slices.

438c ⟨*Prepare sequence, Ch. 47* 438c⟩≡ (438a)
```
se := make([][]byte, 2)
se[0] = make([]byte, n)
copy(se[0], seq.Data())
se[1] = make([]byte, n)
seq.ReverseComplement()
copy(se[1], seq.Data())
```

We branch between two sequencing modes, paired-end and single-end.

438d ⟨*Sequence according to mode, Ch. 47* 438d⟩≡ (438a)
```
if op.p {
        ⟨Paired-end sequencing, Ch. 47 438e⟩
} else {
        ⟨Single-end sequencing, Ch. 47 440a⟩
}
```

In paired-end sequencing, we pick an insert position and length. If the insert is either located inside the template or the template is circular, we sequence the first read mate, followed by the second read mate.

438e ⟨*Paired-end sequencing, Ch. 47* 438e⟩≡ (438d)
```
pos := rn.Intn(n)
il := int(math.Round(rn.NormFloat64() * op.I + op.i))
if pos + il < n || op.o {
        ⟨Sequence first read mate, Ch. 47 439a⟩
        ⟨Sequence second read mate, Ch. 47 439d⟩
}
```

We pick a read length and sequence the first read mate. Negative read lengths are folded to positive. Each nucleotide is mutated according to the error rate using a call to `mutate`, which we write in a moment.

439a ⟨*Sequence first read mate, Ch. 47* 439a⟩≡ (438e)

```
rc++
fmt.Fprintf(w, ">Read%d mate=1\n", rc)
rl := int(math.Round(rn.NormFloat64() * op.R + op.r))
if rl < 0 { rl *= -1 }
for i := pos; i < pos + rl; i++ {
        c := se[0][i % n]
        c = mutate(c, rn, op.e)
        fmt.Fprintf(w, "%c", c)
}
fmt.Fprintf(w, "\n")
ns += rl
```

We import `fmt`.

439b ⟨*Imports, Ch. 47* 434c⟩+≡ (434a) ◁438b

```
"fmt"
```

In `mutate` we change the given nucleotide to one of the three others with the error probability.

439c ⟨*Functions, Ch. 47* 437c⟩+≡ (434a) ◁437c

```
const dna = "ACGT"
func mutate(c byte, r *rand.Rand, e float64) byte {
        if r.Float64() >= e { return c }
        m := dna[r.Intn(4)]
        for m == c {
                m = dna[r.Intn(4)]
        }
        return m
}
```

We look up the start position on the reverse strand, draw a new read length, and sequence the second read mate.

439d ⟨*Sequence second read mate, Ch. 47* 439d⟩≡ (438e)

```
pos = n - (pos + il - 1)
fmt.Fprintf(w, ">Read%d mate=2\n", rc)
rl = int(math.Round(rn.NormFloat64() * op.R + op.r))
if rl < 0 { rl *= -1 }
for i := pos; i < pos + rl; i++ {
        c := se[1][i % n]
        c = mutate(c, rn, op.e)
        fmt.Fprintf(w, "%c", c)
}
fmt.Fprintf(w, "\n")
ns += rl
```

In single-end sequencing, we pick a read position, length, and strand. Then we check that we are either inside the sequence or the sequence is circular. If so, we sequence it.

440a ⟨*Single-end sequencing, Ch. 47* 440a⟩≡ (438d)
```
pos := rn.Intn(n)
rl := int(math.Round(rn.NormFloat64() * op.R + op.r))
if rl < 0 { rl *= -1 }
strand := 0
if rn.Float64() < 0.5 && !op.S { strand = 1 }
if pos + rl <= n || op.o {
        ⟨Sequence single read, Ch. 47 440b⟩
}
```

We increment the read counter and print the read header. Then we iterate over the nucleotides of the read and print them before we add them to the number of nucleotides sequenced.

440b ⟨*Sequence single read, Ch. 47* 440b⟩≡ (440a)
```
rc++
fmt.Fprintf(w, ">Read%d\n", rc)
for i := pos; i  < pos + rl; i++ {
        c := se[strand][i % n]
        c = mutate(c, rn, op.e)
        fmt.Fprintf(w, "%c", c)
}
fmt.Fprintf(w, "\n")
ns += rl
```

We're finished with `sequencer`, let's test it.

Testing

The outline of our testing program contains hooks for imports and the testing logic.

440c ⟨*sequencer_test.go* 440c⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 47 441a⟩
)
func TestSequencer(t *testing.T) {
        ⟨Testing, Ch. 47 440d⟩
}
```

We construct our tests and then iterate over them.

440d ⟨*Testing, Ch. 47* 440d⟩≡ (440c)
```
tests := make([]*exec.Cmd, 0)
⟨Contruct tests, Ch. 47 441b⟩
for i, test := range tests {
        ⟨Run test, Ch. 47 441c⟩
}
```

We import `exec`.

441a ⟨*Testing imports, Ch. 47* 441a⟩≡ (440c) 441d▷
```
"os/exec"
```

We construct four tests, each of which uses the 1 kb random sequence in `test.fasta` as template and a seed for the random number generator to freeze the results.

441b ⟨*Contruct tests, Ch. 47* 441b⟩≡ (440d)
```
f := "test.fasta"
test := exec.Command("./sequencer", "-s", "3", f)
tests = append(tests, test)
test = exec.Command("./sequencer", "-s", "3", "-p", f)
tests = append(tests, test)
test = exec.Command("./sequencer", "-s", "3", "-c", "2", f)
tests = append(tests, test)
test = exec.Command("./sequencer", "-s", "3", "-r", "50", f)
tests = append(tests, test)
```

When we run a test, we compare the result we get with the result we want, which is stored in files `r1.txt`, `r2.txt`, and so on.

441c ⟨*Run test, Ch. 47* 441c⟩≡ (440d)
```
get, err := test.Output()
if err != nil {
          t.Errorf("can't run %q", test)
}
f = "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
          t.Errorf("can't open %q", f)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

441d ⟨*Testing imports, Ch. 47* 441a⟩+≡ (440c) ◁441a
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 48

Program `shustring`: Find Shortest Unique Substrings

Table 48.1: Shustrings starting at every position in $t =$ TATTTTTATA.

i	shu$[i]$	shustring$[i]$
1	4	TATT
2	3	ATT
3	5	TTTTT
4	5	TTTTA
5	4	TTTA
6	3	TTA
7	4	TATA
8	3	ATA

Introduction

Molecular markers are regions of DNA or protein sequences that are diagnostic for a given organism. To find such regions, it can be instructive to search for shortest unique substrings, or shustrings [15]. Consider, for example the sequence

$$t = \texttt{TATTTTTATA}$$

consisting of ten nucleotides. At every position in t we ask, what is the shortest unique substring starting there? Consider the first nucleotide in t, $t[1...1] =$ T is not unique, nor is $t[1...2] =$ TA or $t[1...3] =$ TAT, but $t[1...4] =$ TATT is. Since all extensions of TATT, such as $t[1...5] =$ TATTT, $t[1...6] =$ TATTTT, and so on, are also unique, we call TATT *shortest* unique, a shustring.

Table 48.1 shows the shustring lengths, $\mathrm{shu}[i]$ and the actual shustrings of t. They are a compact representation of a sequence's marker content, because, as we just said, once unique, always unique.

Shustrings are found using the enhanced suffix array of t. This usually consists of two tables, the *suffix array* of alphabetically ordered suffixes, sa, and the longest common prefix array of the lengths of matching, lcp. We augment these two with a third array, the inverse suffix array, isa, to map positions in t onto positions in sa.

As shown in Table 48.2, the enhanced suffix array of t consists of three columns of integers, where sa and isa refer to the starting positions and lcp to the lengths of strings. However, it is easier to think about strings by looking at them than by contemplating numbers, hence Table 48.2 also shows the suffixes, suf, of t. As I just said, their starting positions are in sa. The lengths of the prefixes matching between $\mathrm{suf}[i]$ and $\mathrm{suf}[i-1]$ are in $\mathrm{lcp}[i]$. And isa stores the positions of suffixes in text-order. For example, the first suffix in t, $t[1...]$, is located at $\mathrm{isa}[1] = 6$, the second suffix, $t[2...]$, at $\mathrm{isa}[2] = 3$, and so on. In other words,

$$\mathrm{isa}[\mathrm{sa}[i]] = i. \tag{48.1}$$

From the enhanced suffix array in Table 48.2, we can look up the shustring lengths in Table 48.1 as the lcp-value of the corresponding suffix, or that of its right-hand neighbor, whichever is larger. And since the lcp-values are the lengths of right-maximal repeats, extending them by one creates shsutrings. To summarize,

$$\mathrm{shu}[i] = \max\left(\mathrm{lcp}\left[\mathrm{isa}[i]\right], \mathrm{lcp}\left[isa[i]+1\right]\right) + 1. \tag{48.2}$$

Table 48.2: Enhanced suffix array of $t =$ `TATTTTTATA`.

i	$\mathrm{sa}[i]$	$\mathrm{lcp}[i]$	$\mathrm{isa}[i]$	$\mathrm{suf}[i]$
1	10	-1	6	`A`
2	8	1	3	`ATA`
3	2	2	10	`ATTTTTATA`
4	9	0	9	`TA`
5	7	2	8	`TATA`
6	1	3	7	`TATTTTTATA`
7	6	1	5	`TTATA`
8	5	2	2	`TTTATA`
9	4	3	4	`TTTTATA`
10	3	4	1	`TTTTTATA`

For example, to compute $\mathrm{shu}[1]$, we write

$$\begin{aligned}\mathrm{shu}[1] &= \max\left(\mathrm{lcp}\left[\mathrm{isa}[1]\right], \mathrm{lcp}\left[\mathrm{isa}[1]+1\right]\right)+1\\ &= \max\left(\mathrm{lcp}[6], \mathrm{lcp}[7]\right)+1\\ &= \max(3,1)+1\\ &= 4.\end{aligned}$$

Given the shustrings of t, we can ask, how long are the shortest shustrings? Table 48.1 tells us that's 3, and there are three of them, `ATT`, `TTA`, and `ATA`. We call such shortest shustrings *global*, to distinguish them from the containing set of all shustrings, which we call *local*. The program `shustring` computes either global or local shustrings for an arbitrary set of sequences.

Implementation

The outline of `shustring` has hooks for imports, functions, and the logic of the main function.

444 ⟨*shustring.go* 444⟩≡

```
package main

import (
        ⟨Imports, Ch. 48 445b⟩
)
⟨Functions, Ch. 48 446e⟩
func main() {
        ⟨Main function, Ch. 48 445a⟩
}
```

In the main function we prepare the `log` package, set the usage, declare and parse the options, and parse the input files.

445a ⟨*Main function, Ch. 48* 445a⟩≡ (444)
```
util.PrepLog("shustring")
⟨Set usage, Ch. 48 445c⟩
⟨Declare options, Ch. 48 445e⟩
⟨Parse options, Ch. 48 446a⟩
⟨Parse input files, Ch. 48 446c⟩
```

We import `util`.

445b ⟨*Imports, Ch. 48* 445b⟩≡ (444) 445d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts, the usage message itself, a description of the program's purpose, and an example command.

445c ⟨*Set usage, Ch. 48* 445c⟩≡ (445a)
```
u := "shustring [-h] [options] [files]"
p := "Compute shortest unique substrings."
e := "shustring foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

445d ⟨*Imports, Ch. 48* 445b⟩+≡ (444) ◁445b 445f▷
```
"github.com/evolbioinf/clio"
```

Apart from the default help option, `-h`, we declare five additional options:

1. `-l`: Local shustrings
2. `-s` r: Restrict output to sequences with names matching regular expression r
3. `-r`: Include reverse strand
4. `-q`: Quiet, don't print shustring sequences
5. `-v`: Program version

445e ⟨*Declare options, Ch. 48* 445e⟩≡ (445a)
```
var optL = flag.Bool("l", false, "local")
var optS = flag.String("s", ".", "restrict output to sequences " +
          "described by regex")
var optR = flag.Bool("r", false, "include reverse strand")
var optQ = flag.Bool("q", false, "quiet, don't print shustrings")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

445f ⟨*Imports, Ch. 48* 445b⟩+≡ (444) ◁445d 446b▷
```
"flag"
```

We parse the options and respond to `-s` and `-v`.

446a ⟨*Parse options, Ch. 48* 446a⟩≡ (445a)
```
flag.Parse()
seqReg, err := regexp.Compile(*optS)
if err != nil {
        log.Fatalf("couldn't compile %q.\n", *optS)
}
if *optV {
        util.PrintInfo("shustring")
}
```

We import `regexp` and `log`.

446b ⟨*Imports, Ch. 48* 445b⟩+≡ (444) ◁445f 446d▷
```
"regexp"
"log"
```

The arguments not parsed yet are interpreted as the names of the input files. These are parsed by applying the function `scan` to each one in turn. Scan takes as arguments the option values for local, reverse, and quiet, and the regular expression to pick sequences.

446c ⟨*Parse input files, Ch. 48* 446c⟩≡ (445a)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optL, *optR, *optQ, seqReg)
```

We import `fasta`.

446d ⟨*Imports, Ch. 48* 445b⟩+≡ (444) ◁446b 446f▷
```
"github.com/evolbioinf/fasta"
```

In `scan` we retrieve the options just passed, collect the sequences in the input, and analyze them.

446e ⟨*Functions, Ch. 48* 446e⟩≡ (444)
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 48 446g⟩
        ⟨Collect sequences, Ch. 48 447a⟩
        ⟨Analyze sequences, Ch. 48 447b⟩
}
```

We import `io`.

446f ⟨*Imports, Ch. 48* 445b⟩+≡ (444) ◁446d 448c▷
```
"io"
```

The arguments are retrieved by reflection.

446g ⟨*Retrieve arguments, Ch. 48* 446g⟩≡ (446e)
```
local := args[0].(bool)
reverse := args[1].(bool)
quiet := args[2].(bool)
seqReg := args[3].(*regexp.Regexp)
```

The sequences contained in the current file are stored in the eponymous slice.

447a ⟨*Collect sequences, Ch. 48* 447a⟩≡ (446e)
```
scanner := fasta.NewScanner(r)
var sequences []*fasta.Sequence
for scanner.ScanSequence() {
        sequence := scanner.Sequence()
        sequences = append(sequences, sequence)
}
```

To analyze the sequences, we concatenate them into one long byte slice, and, if appropriate, also add their reverse strands. Then we calculate the enhanced suffix array and the inverse suffix array of the concatenated data. From the enhanced suffix array we compute the shustrings—strictly speaking their lengths—which are analyzed and printed.

447b ⟨*Analyze sequences, Ch. 48* 447b⟩≡ (446e)
```
⟨Concatenate sequences, Ch. 48 447c⟩
if reverse {
        ⟨Concatenate reverse strands, Ch. 48 448a⟩
}
⟨Compute enhanced suffix array, Ch. 48 448b⟩
⟨Compute inverse suffix array, Ch. 48 448d⟩
⟨Compute shustrings, Ch. 48 448e⟩
⟨Analyze shustrings, Ch. 48 449b⟩
⟨Print shustrings, Ch. 48 449d⟩
```

We concatenate the sequences and note their start and end positions. However, concatenation can create new substrings at the border between the joined sequences, which may mask legitimate shustrings. Consider for example the two sequences $s_1 =$ `GTG` and $s_2 =$ `TT`. Their combined shustring inventory is `GT`, `TG`, and `TT`. However, concatenation to `GTGTT` creates a second `GT`, which masks the uniqueness of the first. To prevent this, we separate sequences by a character outside of their alphabet, the zero byte.

447c ⟨*Concatenate sequences, Ch. 48* 447c⟩≡ (447b)
```
var cat []byte
var start, end []int
start = append(start, 0)
for i, sequence := range sequences {
        if i > 0 {
                cat = append(cat, 0)
                start = append(start, end[i-1]+1)
        }
        cat = append(cat, sequence.Data()...)
        end = append(end, start[i] + len(sequence.Data()))
}
```

We reverse-complement each sequence and append it. No position information is required for the reverse strands, but we still separate sequences by the zero byte to prevent the creation of spurious substrings.

448a ⟨*Concatenate reverse strands, Ch. 48* 448a⟩≡ (447b)
```
for _, sequence := range sequences {
        sequence.ReverseComplement()
        cat = append(cat, 0)
        cat =  append(cat, sequence.Data()...)
}
```

For the upcoming shustring analysis, we process the sequence just generated into the three components of an enhanced suffix array, the suffix array proper and the longest common prefix array.

448b ⟨*Compute enhanced suffix array, Ch. 48* 448b⟩≡ (447b)
```
sa := esa.Sa(cat)
lcp := esa.Lcp(cat, sa)
```

We import `esa`.

448c ⟨*Imports, Ch. 48* 445b⟩+≡ (444) ◁446f 449c▷
```
"github.com/evolbioinf/esa"
```

We now implement the computation of shustring lengths using equation (48.2). To ensure that there always exists an element $\mathrm{lcp}[i+1]$, we extend lcp by one cell. The value of this extra cell should be smaller than the length of any legitimate common prefix, we use the same value as in the first cell, -1.

We implement the inverse suffix array according to equation (48.1).

448d ⟨*Compute inverse suffix array, Ch. 48* 448d⟩≡ (447b)
```
isa := make([]int, len(sa))
for i, _ := range sa {
        isa[sa[i]] = i
}
```

448e ⟨*Compute shustrings, Ch. 48* 448e⟩≡ (447b)
```
shu := make([]int, len(sa))
lcp = append(lcp, -1)
for i, _ := range sequences {
        for j := start[i]; j < end[i]; j++ {
                ⟨Calculate a shustring length, Ch. 48 449a⟩
        }
}
```

When calculating an individual shustring length, we mark shustrings that extend beyond the end of their host sequence as non-existent by setting them to the largest integer available.

449a ⟨*Calculate a shustring length, Ch. 48* 449a⟩≡ (448e)
```
is := isa[j]
shu[is] = lcp[is]
if lcp[is+1] > shu[is] {
        shu[is] = lcp[is+1]
}
shu[is]++
if sa[is] + shu[is] > end[i] {
        shu[is] = math.MaxInt64
}
```

In local mode, we print all shustrings, in global just the shortest. So the difference between these modes is the maximum length of a shustring to be printed. In local mode, we include all shustrings and thus set the maximum is a very large integer, but not the largest, as we have just used that to mark positions without any shustring at all. In global mode, the minimum shustring length becomes the maximum, which is computed for each sequence.

449b ⟨*Analyze shustrings, Ch. 48* 449b⟩≡ (447b)
```
var maxima []int
for i, _ := range sequences {
        maxima = append(maxima, math.MaxInt64 - 1)
        if local { continue }
        for j := start[i]; j < end[i]; j++ {
                l := shu[isa[j]]
                if l < maxima[i] {
                        maxima[i] = l
                }
        }
}
```

We import `math`.

449c ⟨*Imports, Ch. 48* 445b⟩+≡ (444) ◁448c 450a▷
```
"math"
```

We print the shustrings to a tab writer, which we prepare before iterating over the sequences.

449d ⟨*Print shustrings, Ch. 48* 449d⟩≡ (447b)
```
⟨Prepare tab writer, Ch. 48 449e⟩
⟨Iterate over sequences, Ch. 48 450b⟩
```

A tab writer is used to write to a buffer. We initialize the writer to a minimal column width of 1 and tabs zero characters wide padded with two blanks.

449e ⟨*Prepare tab writer, Ch. 48* 449e⟩≡ (449d)
```
var buf []byte
buffer := bytes.NewBuffer(buf)
w := new(tabwriter.Writer)
w.Init(buffer, 1, 0, 2, ' ', 0)
```

We import `bytes` and `tabwriter`.

450a ⟨*Imports, Ch. 48* 445b⟩+≡ (444) ◁449c 450c▷
```
"bytes"
"text/tabwriter"
```

For each sequence that matches the regular expression, we write the sequence header and a table of shustrings.

450b ⟨*Iterate over sequences, Ch. 48* 450b⟩≡ (449d)
```
for i, sequence := range sequences {
        header := []byte(sequence.Header())
        match := seqReg.Find(header)
        if match == nil {
                continue
        }
        fmt.Printf(">%s\n", sequence.Header())
        ⟨Write shustring table, Ch. 48 450d⟩
}
```

We import `fmt`.

450c ⟨*Imports, Ch. 48* 445b⟩+≡ (444) ◁450a
```
"fmt"
```

The shustring table consists of a header a body. For every table, we initially reset the buffer and flush the buffer prior to printing.

450d ⟨*Write shustring table, Ch. 48* 450d⟩≡ (450b)
```
buffer.Reset()
⟨Write table header, Ch. 48 450e⟩
⟨Write table body, Ch. 48 451a⟩
w.Flush()
fmt.Printf("%s", buffer)
```

The table header differs between the modes. In global mode, it consists of four fields, count, position, length, and sequence. In local mode, the count and the position coincide, so we drop the count. Similarly, in quiet mode we drop the column of shustring sequences.

450e ⟨*Write table header, Ch. 48* 450e⟩≡ (450d)
```
fmt.Fprintf(w, "#\t")
if !local {
        fmt.Fprint(w, "Count\t")
}
fmt.Fprint(w, "Position\tLength")
if !quiet {
        fmt.Fprintf(w, "\tShustring")
}
fmt.Fprint(w, "\n")
```

For a given sequence, we walk though the lcp array in text-order and count and write the shustrings that don't exceed the maximum.

451a ⟨*Write table body, Ch. 48* 451a⟩≡ (450d)
```
count := 0
for j := start[i]; j < end[i]; j++ {
        is := isa[j]
        if shu[is] <= maxima[i] {
                count++
                ⟨Write a shustring, Ch. 48 451b⟩
        }
}
```

We write the count of a shustring, its starting position, and, where appropriate, its sequence. This is best looked up in the concatenated sequence, as the individual sequences might have been reverse-complemented.

451b ⟨*Write a shustring, Ch. 48* 451b⟩≡ (451a)
```
s := sa[is] - start[i]
l := shu[is]
if !local { fmt.Fprintf(w, "\t%d", count) }
fmt.Fprintf(w, "\t%d\t%d", s+1, l)
if !quiet {
        s = sa[is]
        str := string(cat[s:s+l])
        fmt.Fprintf(w, "\t%s", str)
}
fmt.Fprintf(w, "\n")
```

We're done writing `shustring`, time to test it.

Testing

The testing framework has hooks for imports and the testing logic.

451c ⟨*shustring_test.go* 451c⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 48 452b⟩
)

func TestShustring(t *testing.T) {
        ⟨Testing, Ch. 48 452a⟩
}
```

We construct the test commands, the list of files containing the results we want, and run the commands.

452a ⟨*Testing, Ch. 48* 452a⟩≡ (451c)
```
var commands []*exec.Cmd
⟨Construct commands, Ch. 48 452c⟩
⟨Construct list of result files, Ch. 48 452d⟩
for i, command := range commands {
        ⟨Run command, Ch. 48 453a⟩
}
```

We import `exec`.

452b ⟨*Testing imports, Ch. 48* 452b⟩≡ (451c) 452e▷
```
"os/exec"
```

We run a test without any options, followed by one test for each of the five options, so we construct six commands in total.

452c ⟨*Construct commands, Ch. 48* 452c⟩≡ (452a)
```
p := "./shustring"
f := "test.fasta"
c := exec.Command(p, f)
commands = append(commands, c)
c = exec.Command(p, "-l", f)
commands = append(commands, c)
c = exec.Command(p, "-s", "1", f)
commands = append(commands, c)
c = exec.Command(p, "-r", f)
commands = append(commands, c)
c = exec.Command(p, "-q", f)
commands = append(commands, c)
```

For each command we construct a result file.

452d ⟨*Construct list of result files, Ch. 48* 452d⟩≡ (452a)
```
var results []string
for i, _ := range commands {
        name := "r" + strconv.Itoa(i+1) + ".txt"
        results = append(results, name)
}
```

We import `strconv`.

452e ⟨*Testing imports, Ch. 48* 452b⟩+≡ (451c) ◁452b 453b▷
```
"strconv"
```

For each command we compare what we get with what we want.

453a ⟨*Run command, Ch. 48* 453a⟩≡ (452a)

```
get, err := command.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", command)
}
want, err := ioutil.ReadFile(results[i])
if err != nil {
          t.Errorf("couldn't open %q\n", results[i])
}
if !bytes.Equal(want, get) {
          t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `ioutil` and `bytes`.

453b ⟨*Testing imports, Ch. 48* 452b⟩+≡ (451c) ◁452e

```
"io/ioutil"
"bytes"
```

Chapter 49

Program `simNorm`: Simulate Samples under the Normal Distribution

Introduction

Many random variables we observe in nature are normally distributed. As a result, many statistical tests are based on the assumption of a normal null distribution. It is thus useful to be able to generate samples under this distribution, for example to explore statistical tests. The program `simNorm` simulates samples drawn from the normal distribution and produces output that can be read by `testMeans`.

Implementation

The program outline has hooks for imports, and the logic of the main function.

455a ⟨*simNorm.go* 455a⟩≡
```
package main

import (
        ⟨Imports, Ch. 49 455c⟩
)

func main() {
        ⟨Main function, Ch. 49 455b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare and parse the options, and carry out the simulation.

455b ⟨*Main function, Ch. 49* 455b⟩≡ (455a)
```
util.PrepLog("simNorm")
⟨Set usage, Ch. 49 455d⟩
⟨Declare options, Ch. 49 456a⟩
⟨Parse options, Ch. 49 456c⟩
⟨Carry out simulation, Ch. 49 456e⟩
```

We import `util`.

455c ⟨*Imports, Ch. 49* 455c⟩≡ (455a) 455e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts, the usage message itself, an explanation of the program's purpose, and an example command.

455d ⟨*Set usage, Ch. 49* 455d⟩≡ (455b)
```
u := "simNorm [-h] [options]"
p := "Simulate samples drawn from the normal distribution."
e := "simNorm -i 3"
clio.Usage(u, p, e)
```

We import `clio`.

455e ⟨*Imports, Ch. 49* 455c⟩+≡ (455a) ◁455c 456b ▷
```
"github.com/evolbioinf/clio"
```

We declare six options:

1. `-i`: number of iterations

2. `-n`: sample size
3. `-m`: mean
4. `-d`: standard deviation
5. `-s`: seed for random number generator
6. `-v`: version

456a ⟨*Declare options, Ch. 49* 456a⟩≡ (455b)
```
var optI = flag.Int("i", 10, "number of iterations")
var optN = flag.Int("n", 8, "sample size")
var optM = flag.Float64("m", 0, "mean")
var optD = flag.Float64("d", 1, "standard deviation")
var optS = flag.Int("s", 0, "seed for random number " +
          "generator; default: internal")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

456b ⟨*Imports, Ch. 49* 455c⟩+≡ (455a) ◁455e 456d▷
```
"flag"
```

We parse the options, print the version if requested, and initialize the random number generator.

456c ⟨*Parse options, Ch. 49* 456c⟩≡ (455b)
```
flag.Parse()
if *optV {
          util.PrintInfo("simNorm")
}
seed := int64(*optS)
if seed == 0 {
          seed = time.Now().UnixNano()
}
rand.Seed(seed)
```

We import `time` and `rand`.

456d ⟨*Imports, Ch. 49* 455c⟩+≡ (455a) ◁456b 457a▷
```
"time"
"math/rand"
```

We use a `tabwriter` to arrange the output into neat columns. The `tabwriter` is first constructed, then the samples are written, before the final result is printed.

456e ⟨*Carry out simulation, Ch. 49* 456e⟩≡ (455b)
```
⟨Create tabwriter, Ch. 49 456f⟩
⟨Write samples, Ch. 49 457b⟩
⟨Print result, Ch. 49 457f⟩
```

A `tabwriter` writes to a buffer. The writer is initialized to a minimal cell width of 1, tabs of width zero, and padding with two blanks.

456f ⟨*Create tabwriter, Ch. 49* 456f⟩≡ (456e)
```
var buf []byte
buffer := bytes.NewBuffer(buf)
w := new(tabwriter.Writer)
w.Init(buffer, 1, 0, 2, ' ', 0)
```

We import `bytes` and `tabwriter`.

457a ⟨*Imports, Ch. 49* 455c⟩+≡ (455a) ◁456d 457d▷
```
"bytes"
"text/tabwriter"
```

The samples are effectively written as a table. We first write the table header, then its body.

457b ⟨*Write samples, Ch. 49* 457b⟩≡ (456e)
```
⟨Print table header, Ch. 49 457c⟩
⟨Print table body, Ch. 49 457e⟩
```

The table header has $n + 1$ entries, where n is the sample size.

457c ⟨*Print table header, Ch. 49* 457c⟩≡ (457b)
```
n := *optN
fmt.Fprintf(w, "# ID\t")
for i := 0; i < n; i++ {
        fmt.Fprintf(w, "x_%d\t", i + 1)
}
fmt.Fprintf(w, "\n")
```

We import `fmt`.

457d ⟨*Imports, Ch. 49* 455c⟩+≡ (455a) ◁457a
```
"fmt"
```

The table body consists of random entries computed as

$$r = \text{normRand}() \times s + m,$$

where s is the standard deviation set by the user and m the mean.

457e ⟨*Print table body, Ch. 49* 457e⟩≡ (457b)
```
m := *optM
s := *optD
for i := 0; i < *optI; i++ {
        fmt.Fprintf(w, "s_%d\t", i + 1)
        for j := 0; j < n; j++ {
                r := rand.NormFloat64() * s + m
                fmt.Fprintf(w, "%.3g\t", r)
        }
        fmt.Fprintf(w, "\n")
}
```

Before printing the buffer, the tabwriter is flushed.

457f ⟨*Print result, Ch. 49* 457f⟩≡ (456e)
```
w.Flush()
fmt.Printf("%s", buffer)
```

We're done with `simNorm`, let's test it.

Testing

There are three options, number of iterations, mean, and standard deviation, so we run four tests, one with all defaults, and one for each option.

The outline of the testing program has hooks for imports and the testing logic.

458a ⟨*simNorm_test.go* 458a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 49 458c⟩
)

func TestSimNorm(t *testing.T) {
        ⟨Testing, Ch. 49 458b⟩
}
```

The test results are compared to output stored in a list of files, `r1.txt`, `r2.txt`,... So we construct the tests and the list of output files, and run the tests.

458b ⟨*Testing, Ch. 49* 458b⟩≡ (458a)
```
tests := make([]*exec.Cmd, 0)
⟨Construct tests, Ch. 49 458d⟩
⟨Construct file names, Ch. 49 458e⟩
for i, test := range tests {
        ⟨Run test, Ch. 49 459b⟩
}
```

We import `exec`.

458c ⟨*Testing imports, Ch. 49* 458c⟩≡ (458a) 459a ▷
```
"os/exec"
```

Every test is run with a preset seed for the random number generator to make it reproducible.

458d ⟨*Construct tests, Ch. 49* 458d⟩≡ (458b)
```
test := exec.Command("./simNorm", "-s", "3")
tests = append(tests, test)
test = exec.Command("./simNorm", "-s", "3", "-i", "3")
tests = append(tests, test)
test = exec.Command("./simNorm", "-s", "3", "-m", "10.1")
tests = append(tests, test)
test = exec.Command("./simNorm", "-s", "3", "-d", "2.5")
tests = append(tests, test)
```

For each test there is a results file.

458e ⟨*Construct file names, Ch. 49* 458e⟩≡ (458b)
```
results := make([]string, 0)
for i, _ := range tests {
        r := "r" + strconv.Itoa(i+1) + ".txt"
        results = append(results, r)
}
```

We import `strconv`.

459a ⟨*Testing imports, Ch. 49* 458c⟩+≡ (458a) ◁458c 459c▷

```
"strconv"
```

A given test is run, and the result we get compared to the result we want.

459b ⟨*Run test, Ch. 49* 459b⟩≡ (458b)

```
get, err := test.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", test)
}
want, err := ioutil.ReadFile(results[i])
if err != nil {
          t.Errorf("couldn't open %q\n", results[i])
}
if !bytes.Equal(want, get) {
          t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `ioutil` and `bytes`.

459c ⟨*Testing imports, Ch. 49* 458c⟩+≡ (458a) ◁459a

```
"io/ioutil"
"bytes"
```

Chapter 50

Program `simOrf`: Simulate Open Reading Frames

Introduction

In procaryotes, proteins are encoded by stretches of DNA that start with an initiation codon, `ATG`, and end with a stop codon, `TAA`, `TAG`, or `TGA`. Such a stretch of DNA is called an open reading frame, or ORF. The program `simOrf` simulates the lengths of open reading frames in random DNA. This is done by drawing random codons and counting the steps until a stop codon is encountered. Our program `simOrf` implements this method. Its outline contains hooks for impports, functions, and the logic of the main function.

461a ⟨*simOrf.go* 461a⟩≡
```
package main

import (
        ⟨Imports, Ch. 50 461c⟩
)

func main() {
        ⟨Main function, Ch. 50 461b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and simulate the ORFs.

461b ⟨*Main function, Ch. 50* 461b⟩≡ (461a)
```
util.PrepLog("simOrf")
⟨Set usage, Ch. 50 461d⟩
⟨Declare options, Ch. 50 461f⟩
⟨Parse options, Ch. 50 462b⟩
⟨Simulate ORFs, Ch. 50 462f⟩
```

We import `util`.

461c ⟨*Imports, Ch. 50* 461c⟩≡ (461a) 461e ▷
```
"github.com/evolbioinf/biobox/util"
```

461d ⟨*Set usage, Ch. 50* 461d⟩≡ (461b)
```
u := "simOrf [-h] [option]..."
p := "Simulate the lengths of open reading frames in random DNA."
e := "simOrf -n 5"
clio.Usage(u, p, e)
```

We import `clio`.

461e ⟨*Imports, Ch. 50* 461c⟩+≡ (461a) ◁ 461c 462a ▷
```
"github.com/evolbioinf/clio"
```

The user can set the number of ORF-lengths printed and the seed for the random number generator. Seeds are always long integers. (S)he can also get the version.

461f ⟨*Declare options, Ch. 50* 461f⟩≡ (461b)
```
var optN = flag.Int("n", 10, "number of ORFs")
var optS = flag.Int64("s", 0, "seed for random number " +
          "generator; default: internal")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

462a ⟨*Imports, Ch. 50* 461c⟩+≡ (461a) ◁461e 462e▷
```
"flag"
```

We parse the options and respond to `-v` and `-s`.

462b ⟨*Parse options, Ch. 50* 462b⟩≡ (461b)
```
flag.Parse()
⟨Respond to -v, Ch. 50 462c⟩
⟨Respond to -s, Ch. 50 462d⟩
```

If requested, we print the version.

462c ⟨*Respond to -v, Ch. 50* 462c⟩≡ (462b)
```
if *optV {
        util.PrintInfo("simOrf")
}
```

The seed for the random number generator is either given by the user or taken as the number of nanoseconds elapsed in the UNIX epoche.

462d ⟨*Respond to -s, Ch. 50* 462d⟩≡ (462b)
```
seed := *optS
if seed == 0 {
        seed = time.Now().UnixNano()
}
```

We import `time`.

462e ⟨*Imports, Ch. 50* 461c⟩+≡ (461a) ◁462a 462h▷
```
"time"
```

To simulate the ORF lengths, we first seed the random number generator and then generate the ORF lengths.

462f ⟨*Simulate ORFs, Ch. 50* 462f⟩≡ (461b)
```
⟨Seed random number generator, Ch. 50 462g⟩
⟨Generate ORF lengths, Ch. 50 463a⟩
```

The random number generator is seeded from a source.

462g ⟨*Seed random number generator, Ch. 50* 462g⟩≡ (462f)
```
source := rand.NewSource(seed)
r := rand.New(source)
```

We import `rand`.

462h ⟨*Imports, Ch. 50* 461c⟩+≡ (461a) ◁462e 463b▷
```
"math/rand"
```

To generate an ORF length, we keep picking random numbers as long as they are greater than the probability of finding a stop, which is $3/64$.

463a ⟨*Generate ORF lengths, Ch. 50* 463a⟩≡ (462f)
```
pr := 3.0 / 64.0
for i := 0; i < *optN; i++ {
        c := 1
        for x := r.Float64(); x > pr; x = r.Float64() {
                c++
        }
        fmt.Println(c)
}
```

We import `fmt`.

463b ⟨*Imports, Ch. 50* 461c⟩+≡ (461a) ◁462h
```
"fmt"
```

We are done with `simOrf`, so let's test it.

Testing

The outline of our testing program contains hooks for imports and the testing logic.

463c ⟨*simOrf_test.go* 463c⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 50 463e⟩
)

func TestSimOrf(t *testing.T) {
        ⟨Testing, Ch. 50 463d⟩
}
```

We test in two steps. First, we generate the tests and store them in a slice of commands. Then we iterate over them and run each one.

463d ⟨*Testing, Ch. 50* 463d⟩≡ (463c)
```
var tests []*exec.Cmd
⟨Generate tests, Ch. 50 464a⟩
for i, test := range tests {
        ⟨Run test, Ch. 50 464b⟩
}
```

We import `exec`.

463e ⟨*Testing imports, Ch. 50* 463e⟩≡ (463c) 464c▷
```
"os/exec"
```

We generate two tests, each with a seed so we can control the output.

464a ⟨*Generate tests, Ch. 50* 464a⟩≡ (463d)
```
cmd := exec.Command("./simOrf", "-s", "23")
tests = append(tests, cmd)
cmd = exec.Command("./simOrf", "-s", "23", "-n", "20")
tests = append(tests, cmd)
```

We run a test and check we get what we want, which we have stored in results files.

464b ⟨*Run test, Ch. 50* 464b⟩≡ (463d)
```
get, err := test.Output()
if err != nil {
        t.Error(err.Error())
}
f := "r" + strconv.Itoa(i + 1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
        t.Error(err.Error())
}
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

464c ⟨*Testing imports, Ch. 50* 463e⟩+≡ (463c) ◁463e
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 51

Program `sops`: Sum-of-Pairs Score for Multiple Sequence Alignment

```
A-
A-
TT
```

Figure 51.1: Small multiple sequence alignment; its sum-of-pairs score is -9 if match is 1, mismatch -3, and gap -2.

Introduction

The sum-of-pairs score is a popular score for multiple sequence alignments. It is computed by iterating over the alignment columns. For each column, every pair of residues is scored and these scores are summed. Pairs of gaps are ignored. As an example, consider the alignment of three sequences in Figure 51.1. The alignment consists of two columns. Let the score scheme be match $= 1$, mismatch $= -3$, and gap $= -2$. Then the score of the first column is $1 - 3 - 3 = -5$ and the score of the second column is $-2 - 2 = -4$. So the sum-of-pairs score of the alignment is $-5 - 4 = -9$.

The program `sops` reads one or more multiple sequence alignments and prints tier sum-or-pairs scores.

Implementation

The outline of `sops` has hooks for imports, functions, and the logic of the main function.

466a ⟨*sops.go* 466a⟩≡

```
package main

import (
        ⟨Imports, Ch. 51 466c⟩
)

⟨Functions, Ch. 51 468c⟩

func main() {
        ⟨Main function, Ch. 51 466b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, and parse the multiple sequence alignments.

466b ⟨*Main function, Ch. 51* 466b⟩≡ (466a)

```
util.PrepLog("sops")
⟨Set usage, Ch. 51 467a⟩
⟨Declare options, Ch. 51 467c⟩
⟨Parse options, Ch. 51 467e⟩
⟨Parse MSAs, Ch. 51 468b⟩
```

We import `util`.

466c ⟨*Imports, Ch. 51* 466c⟩≡ (466a) 467b ▷

```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage statement, an explanation of the purpose of `sops`, and an example command.

467a ⟨*Set usage, Ch. 51* 467a⟩≡ (466b)
```
u := "sops [-h] [option]... [foo.fasta]..."
p := "Calculate the sum-of-pairs score of a multiple sequence alignment."
e := "sops msa.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

467b ⟨*Imports, Ch. 51* 466c⟩+≡ (466a) ◁466c 467d▷
```
"github.com/evolbioinf/clio"
```

We declare options for the version (`-v`), match (`-m`), mismatch (`-i`), score matrix (`-m`), and gap extension (`-g`). We ignore gap opening.

467c ⟨*Declare options, Ch. 51* 467c⟩≡ (466b)
```
var optV = flag.Bool("v", false, "version")
var optA = flag.Float64("a", 1, "match")
var optI = flag.Float64("i", -3, "mismatch")
var optM = flag.String("m", "", "score matrix")
var optG = flag.Float64("g", -2, "gap")
```

We import `flag`.

467d ⟨*Imports, Ch. 51* 466c⟩+≡ (466a) ◁467b 468a▷
```
"flag"
```

We parse the options and respond to a request for the version, as this stops `sops`. We also get the score matrix.

467e ⟨*Parse options, Ch. 51* 467e⟩≡ (466b)
```
flag.Parse()
if *optV {
        util.PrintInfo("sops")
}
⟨Get score matrix, Ch. 51 467f⟩
```

The score matrix is either constructed from the match and mismatch scores, or read from a file.

467f ⟨*Get score matrix, Ch. 51* 467f⟩≡ (467e)
```
var mat *pal.ScoreMatrix
if *optM == "" {
        mat = pal.NewScoreMatrix(*optA, *optI)
} else {
        f, err := os.Open(*optM)
        if err != nil {
                log.Fatalf("couldn't open score matrix %q",
                        (*optM))
        }
        defer f.Close()
        mat = pal.ReadScoreMatrix(f)
}
```

We import `pal`, `os`, and `log`.

468a ⟨*Imports, Ch. 51* 466c⟩+≡ (466a) ◁467d 468d▷
```
"github.com/evolbioinf/pal"
"os"
"log"
```

The remaining tokens on the command line are interpreted as input files. These are scanned with the function `scan`, which takes as argument the score matrix and the gap score.

468b ⟨*Parse MSAs, Ch. 51* 468b⟩≡ (466b)
```
f := flag.Args()
clio.ParseFiles(f, scan, mat, *optG)
```

Inside `scan`, we retrieve the score matrix and the gap score through type assertion, read the sequences into a multiple sequence alignment, check the multiple sequence alignment, and calculate its sum-of-pairs score.

468c ⟨*Functions, Ch. 51* 468c⟩≡ (466a)
```
func scan(r io.Reader, args ...interface{}) {
        mat := args[0].(*pal.ScoreMatrix)
        g := args[1].(float64)
        sc := fasta.NewScanner(r)
        var msa [][]byte
        for sc.ScanSequence() {
                msa = append(msa, sc.Sequence().Data())
        }
        ⟨Check MSA, Ch. 51 468e⟩
        ⟨Calculate sum-of-pairs, Ch. 51 469a⟩
}
```

We import `io` and `fasta`.

468d ⟨*Imports, Ch. 51* 466c⟩+≡ (466a) ◁468a 469b▷
```
"io"
"github.com/evolbioinf/fasta"
```

We read a sequence and append it to the growing multiple sequence alignment.

If sequences have unequal lengths, we are not dealing with a multiple sequence alignmet and bail with message.

468e ⟨*Check MSA, Ch. 51* 468e⟩≡ (468c)
```
for i := 1; i < len(msa); i++ {
        l1 := len(msa[i-1])
        l2 := len(msa[i])
        if l1 != l2 {
                m := "sequence %d has length %d, " +
                        "but sequence %d has length %d; " +
                        "this doesn't look like an alignment"
                log.Fatalf(m, i, l1, i+1, l2)
        }
}
```

The multiple sequence alignment is now an $(m \times n)$ matrix of residues. We score all pairs of residues in the MSA and print the result.

469a ⟨*Calculate sum-of-pairs, Ch. 51* 469a⟩≡ (468c)
```
m := len(msa)
n := len(msa[0])
s := 0.0
for i := 0; i < n; i++ {
        for j := 0; j < m-1; j++ {
                for k := j+1; k<m; k++ {
                        ⟨Score pair of residues, Ch. 51 469c⟩
                }
        }
}
fmt.Printf("sum-of-pairs_score\t%g\n", s)
```

We import `fmt`.

469b ⟨*Imports, Ch. 51* 466c⟩+≡ (466a) ◁468d
```
"fmt"
```

A pair falls in one of three categories: It consists of two residues, in which case we read its score from the score matrix; or it consists of a gap and a residue, in which case its score is the gap score, or it consists of two gaps, in which case we ignore it.

469c ⟨*Score pair of residues, Ch. 51* 469c⟩≡ (469a)
```
r1 := msa[j][i]
r2 := msa[k][i]
if r1 == '-' && r2 == '-' {
        continue
}
if r1 == '-' || r2 == '-' {
        s += g
} else {
        s += mat.Score(r1, r2)
}
```

We've finished `sops`, time to test it.

Testing

Our testing code for `sops` has hooks for imports and the testing logic.

469d ⟨*sops_test.go* 469d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 51 470b⟩
)

func TestSops(t *testing.T) {
        ⟨Testing, Ch. 51 470a⟩
}
```

We construct a set of tests and then run them.

470a ⟨*Testing, Ch. 51* 470a⟩≡ (469d)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 51 470c⟩
for i, test := range tests {
        ⟨Run test, Ch. 51 470d⟩
}
```

We import `exec`.

470b ⟨*Testing imports, Ch. 51* 470b⟩≡ (469d) 470e ▷
```
"os/exec"
```

We construct four tests, one for each option. All tests take as input the tiny alignment in Figure 51.1, which is contained in `test.fasta`.

470c ⟨*Construct tests, Ch. 51* 470c⟩≡ (470a)
```
f := "test.fasta"
test := exec.Command("./sops", "-a", "2", f)
tests = append(tests, test)
test = exec.Command("./sops", "-i", "-2", f)
tests = append(tests, test)
test = exec.Command("./sops", "-g", "-1", f)
tests = append(tests, test)
test = exec.Command("./sops", "-m", "sm.txt", f)
tests = append(tests, test)
```

When running a test, we compare the result we get with the result we want, which is contained in files `r1.txt`, `r2.txt`, and so on.

470d ⟨*Run test, Ch. 51* 470d⟩≡ (470a)
```
get, err := test.Output()
if err != nil {
        t.Errorf("couldn't run %q", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want, err := ioutil.ReadFile(f)
if err != nil {
        t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

470e ⟨*Testing imports, Ch. 51* 470b⟩+≡ (469d) ◁ 470b
```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 52

Program `testMeans`: Statistical Test of two Means

Introduction

Given two samples with means μ_1 and μ_2, `testMeans` tests the null hypothesis that $\mu_1 \approx \mu_2$. Three tests are available, Student's t-test with equal variances, Student's t-test with unequal variances, also known as Welch's test, and a Monte-Carlo test. The Monte-Carlo test establishes the frequency with which a difference in means at least as large as the one observed between μ_1 and μ_2 is found by chance alone.

The input consists of two files of matched samples. These might correspond to expression values for genes under treatment and control conditions. For example,

Samples 1

id_1	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	
id_2	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$
id_3	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$

Samples 2

id_1	$y_{1,1}$	$y_{1,2}$	$y_{1,3}$	$y_{1,4}$	
id_2	$y_{2,1}$	$y_{2,2}$	$y_{2,3}$	$y_{2,4}$	$y_{2,5}$
id_3	$y_{3,1}$	$y_{3,2}$	$y_{3,3}$	$y_{3,4}$	
id_4	$y_{4,1}$	$y_{4,2}$	$y_{4,3}$	$y_{4,4}$	

So an individual sample occupies one row and consists of an identifier followed by numerical values separated by blanks. Samples are matched by identifiers, which means they don't need to be in the same order in the two files. The program only considers those samples that have entries in both files. So for our example the output would be

ID	Mean 1	Mean 2	P
id_1	μ_1^1	μ_1^2	P_1
id_2	μ_2^1	μ_2^2	P_2
id_3	μ_3^1	μ_3^2	P_3

Implementation

The outline of `testMeans` provides hooks for imports, types, functions, and the logic of the main function.

472a ⟨*testMeans.go* 472a⟩≡

```
package main

import (
        ⟨Imports, Ch. 52 473a⟩
)
⟨Types, Ch. 52 476b⟩
⟨Functions, Ch. 52 475a⟩
func main() {
        ⟨Main function, Ch. 52 472b⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare and parse the options, read the two input files, carry out the tests, and print the results.

472b ⟨*Main function, Ch. 52* 472b⟩≡ (472a)

```
util.PrepLog("testMeans")
⟨Set usage, Ch. 52 473b⟩
⟨Declare options, Ch. 52 473d⟩
⟨Parse options, Ch. 52 473f⟩
⟨Read input files, Ch. 52 474f⟩
⟨Carry out tests, Ch. 52 476a⟩
⟨Print results, Ch. 52 477f⟩
```

We import `util`.

473a ⟨*Imports, Ch. 52* 473a⟩≡ (472a) 473c ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage message has three parts, the usage proper, an explanation of the program's purpose, and an example command. We include a sketch of the input data with the program's purpose.

473b ⟨*Set usage, Ch. 52* 473b⟩≡ (472b)
```
u := "testMeans [-h] [options] samples1.txt samples2.txt"
p := "Student's t-test for multiple experiments.\n" +
          "Data: name_1 x_1,1 x_1,2 ...\n" +
          "      name_2 x_2,1 x_2,2 ...\n" +
          "      ..."
e := "testMeans -m 10000 samples1.txt samples2.txt"
clio.Usage(u, p, e)
```

We import `clio`.

473c ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁473a 473e ▷
```
"github.com/evolbioinf/clio"
```

We declare four options,

1. `-u`: unequal variance
2. `-m`: number of iterations for Monte-Carlo test
3. `-s`: seed for random number generator
4. `-v`: version

473d ⟨*Declare options, Ch. 52* 473d⟩≡ (472b)
```
var optU = flag.Bool("u", false, "unequal variance")
var optM = flag.Int("m", 0, "Monte-Carlo iterations")
var optS = flag.Int("s", 0, "seed for random number generator")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

473e ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁473c 474a ▷
```
"flag"
```

We parse the options and respond to `-v` by printing the program version, and to `-m` by initializing the random number generator. We also determine the names of the two input files.

473f ⟨*Parse options, Ch. 52* 473f⟩≡ (472b)
```
flag.Parse()
if *optV {
          util.PrintInfo("testMeans")
}
if *optM > 0 {
          ⟨Initialize random number generator, Ch. 52 474b⟩
}
⟨Get names of input files, Ch. 52 474d⟩
```

We import `rand`.

474a ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁473e 474c▷
```
"math/rand"
```

If the user supplied a seed for the random number generator, we use that, otherwise the current time.

474b ⟨*Initialize random number generator, Ch. 52* 474b⟩≡ (473f)
```
seed := int64(*optS)
if seed == 0 {
        seed = time.Now().UnixNano()
}
rand.Seed(seed)
```

We import `time`.

474c ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁474a 474e▷
```
"time"
```

If the user hasn't supplied two input files, we kindly ask for them and abort.

474d ⟨*Get names of input files, Ch. 52* 474d⟩≡ (473f)
```
if len(flag.Args()) != 2 {
        fmt.Fprintf(os.Stderr,
                "Please supply two input files.\n")
        os.Exit(0)
}
dataFile1 := flag.Args()[0]
dataFile2 := flag.Args()[1]
```

We import `fmt` and `os`.

474e ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁474c 475b▷
```
"fmt"
"os"
```

The data files are read by calling a dedicated function.

474f ⟨*Read input files, Ch. 52* 474f⟩≡ (472b)
```
samples1, ids := readData(dataFile1)
samples2, _   := readData(dataFile2)
```

The data file is opened and scanned. Each sample in it is loaded into a map of identifiers and measurements. The identifiers listed in the first data file are also returned separately. We use them later to order the output, as the keys of a map have no stable order.

475a ⟨*Functions, Ch. 52* 475a⟩≡ (472a) 477c ▷
```
func readData(file string) (map[string][]float64, []string) {
        r, err := os.Open(file)
        if err != nil {
                log.Fatalf("couldn't open %q\n", file)
        }
        samples := make(map[string][]float64)
        sc := bufio.NewScanner(r)
        ids := make([]string, 0)
        ⟨Read samples, Ch. 52 475c⟩
        return samples, ids
}
```

We import `log` and `bufio`.

475b ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁474e 475e ▷
```
"log"
"bufio"
```

Samples are contained in unhashed lines. For each sample, the key is the identifier in the first column, and the value the numbers in the subsequent columns, which are stored in a slice.

475c ⟨*Read samples, Ch. 52* 475c⟩≡ (475a)
```
for sc.Scan() {
        if sc.Text()[0] == '#' { continue }
        fields := strings.Fields(sc.Text())
        ids = append(ids, fields[0])
        n := len(fields)
        numbers := make([]float64, 0)
        ⟨Store numbers, Ch. 52 475d⟩
        samples[fields[0]] = numbers
}
```

We import `strings`.

Before storing a number, it is converted from string.

475d ⟨*Store numbers, Ch. 52* 475d⟩≡ (475c)
```
for i := 1; i < n; i++ {
        x, err := strconv.ParseFloat(fields[i], 64)
        if err != nil {
                log.Fatalf("couldn't convert %q\n", fields[i])
        }
        numbers = append(numbers, x)
}
```

475e ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁475b 477e ▷
```
"strings"
"strconv"
```

Test results are stored in a map pairing the identifier with a result. We iterate over the identifiers and for each one choose the test requested.

476a ⟨*Carry out tests, Ch. 52* 476a⟩≡ (472b)
```
results := make(map[string]result)
for _, id := range ids {
        result := new(result)
        sample1 := samples1[id]
        sample2 := samples2[id]
        ⟨Choose test, Ch. 52 476c⟩
        results[id] = *result
}
```

A result consists of the two means, the test statistic, and its significance.

476b ⟨*Types, Ch. 52* 476b⟩≡ (472a)
```
type result struct {
        m1, m2, t, p float64
}
```

We always carry out the parametric test, the Monte-Carlo test only if desired.

476c ⟨*Choose test, Ch. 52* 476c⟩≡ (476a)
```
⟨Parametric test, Ch. 52 476d⟩
if *optM > 0 {
        ⟨Monte-Carlo test, Ch. 52 476e⟩
}
```

The parametric test is delegated to a function.

476d ⟨*Parametric test, Ch. 52* 476d⟩≡ (476c)
```
m1, m2, t, p := util.TTest(sample1, sample2, !*optU)
result.m1 = m1
result.m2 = m2
result.t = t
result.p = p
```

The Monte-Carlo test starts from the observed difference between the two sample means. The measurements are then shuffled between the samples, and the means are recomputed and compared.

476e ⟨*Monte-Carlo test, Ch. 52* 476e⟩≡ (476c)
```
result.p = 0
do := math.Abs(result.m1 - result.m2)
merged := sample1
merged = append(merged, sample2...)
l := len(sample1)
for i := 0; i < *optM; i++ {
        ⟨Shuffle values, Ch. 52 477a⟩
        ⟨Get shuffled means, Ch. 52 477b⟩
        ⟨Compare differences between means, Ch. 52 477d⟩
}
result.p /= float64(*optM)
```

Both samples are written into a single slice and shuffled.

477a ⟨*Shuffle values, Ch. 52* 477a⟩≡ (476e)
```
rand.Shuffle(len(merged), func(i, j int) {
        merged[i], merged[j] = merged[j], merged[i]
})
```

The slice just shuffled is divided into two portions the size of the original samples, and the mean of each portion is computed.

477b ⟨*Get shuffled means, Ch. 52* 477b⟩≡ (476e)
```
m1 := mean(merged[0:l])
m2 := mean(merged[l:])
```

We calculate the mean.

477c ⟨*Functions, Ch. 52* 475a⟩+≡ (472a) ◁475a
```
func mean(data []float64) float64 {
        var avg float64
        for _, d := range data {
                avg += d
        }
        avg /= float64(len(data))
        return avg
}
```

If the difference between the shuffled means is greater or equal to the difference between the original means, we count.

477d ⟨*Compare differences between means, Ch. 52* 477d⟩≡ (476e)
```
d := math.Abs(m1 - m2)
if d >= do {
        result.p++
}
```

We import `math`.

477e ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁475e 478a▷
```
"math"
```

Having performed the tests, we print the results. To line them up in neat columns, we use a `tabwriter`.

477f ⟨*Print results, Ch. 52* 477f⟩≡ (472b)
⟨*Construct* `tabwriter`*, Ch. 52* 477g⟩
⟨*Write results, Ch. 52* 478b⟩
⟨*Output, Ch. 52* 478e⟩

The `tabwriter` writes to a byte buffer. The writer is initialized to a minimal cell width of 1, tabs of width zero, and padding with two blanks.

477g ⟨*Construct* `tabwriter`*, Ch. 52* 477g⟩≡ (477f)
```
var buf []byte
buffer := bytes.NewBuffer(buf)
w := new(tabwriter.Writer)
w.Init(buffer, 1, 0, 2, ' ', 0)
```

We import `bytes` and `tabwriter`.

478a ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁477e 478c▷
```
"bytes"
"text/tabwriter"
```

The results table has a header line followed by rows of data. However, if $P = 0$ was returned by the Monte-Carlo test, we need to think again.

478b ⟨*Write results, Ch. 52* 478b⟩≡ (477f)
```
fmt.Fprintf(w, "# ID\tm1\tm2\tt\tP\t\n")
for _, id := range ids {
        r := results[id]
        ⟨Check for zero P-value, Ch. 52 478d⟩
}
w.Flush()
```

We import `fmt`.

478c ⟨*Imports, Ch. 52* 473a⟩+≡ (472a) ◁478a
```
"fmt"
```

$P = 0$ obtained by Monte-Carlo with n iterations, in fact signifies $P < 1/n$.

478d ⟨*Check for zero P-value, Ch. 52* 478d⟩≡ (478b)
```
if r.p == 0 && *optM > 0 {
        x := 1.0 / float64(*optM)
        fmt.Fprintf(w, "%s\t%.3g\t%.3g\t%.3g\t<%.3g\t\n",
                id, r.m1, r.m2, r.t, x)
} else {
        fmt.Fprintf(w, "%s\t%.3g\t%.3g\t%.3g\t%.3g\t\n",
                id, r.m1, r.m2, r.t, r.p)
}
```

The buffer contains the output.

478e ⟨*Output, Ch. 52* 478e⟩≡ (477f)
```
fmt.Printf("%s", buffer)
```

This completes `testMeans`, time to test it.

Testing

The testing outline provides hooks for imports and the testing logic.

478f ⟨*testMeans_test.go* 478f⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 52 479b⟩
)
func TestTestMeans(t *testing.T) {
        ⟨Testing, Ch. 52 479a⟩
}
```

We construct a list of tests and a list of files that contain the output we want. Then we run the tests.

479a ⟨*Testing, Ch. 52* 479a⟩≡ (478f)
```
tests := make([]*exec.Cmd, 0)
⟨Construct tests, Ch. 52 479c⟩
⟨Construct list of result files, Ch. 52 479d⟩
for i, test := range tests {
        ⟨Run test, Ch. 52 479f⟩
}
```

We import `exec`.

479b ⟨*Testing imports, Ch. 52* 479b⟩≡ (478f) 479e▷
```
"os/exec"
```

We analyze two small data files three times in the three modes of `testMeans`, Student's, Welch's, and Monte-Carlo.

479c ⟨*Construct tests, Ch. 52* 479c⟩≡ (479a)
```
test := exec.Command("./testMeans", "d1.txt", "d2.txt")
tests = append(tests, test)
test = exec.Command("./testMeans", "-u", "d1.txt", "d2.txt")
tests = append(tests, test)
test = exec.Command("./testMeans", "-s", "3", "-m", "1000",
        "d1.txt", "d2.txt")
tests = append(tests, test)
```

We construct as many results files as tests.

479d ⟨*Construct list of result files, Ch. 52* 479d⟩≡ (479a)
```
results := make([]string, 0)
for i, _ := range tests {
        r := "r" + strconv.Itoa(i+1) + ".txt"
        results = append(results, r)
}
```

We import `strconv`.

479e ⟨*Testing imports, Ch. 52* 479b⟩+≡ (478f) ◁479b 480▷
```
"strconv"
```

In a given test, we compare the result we want with the result we get.

479f ⟨*Run test, Ch. 52* 479f⟩≡ (479a)
```
want, err := ioutil.ReadFile(results[i])
if err != nil {
        t.Errorf("couldn't open %q\n", results[i])
}
get, err := test.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", test)
}
if !bytes.Equal(want, get) {
        t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `ioutil` and `bytes`.

480 ⟨*Testing imports, Ch. 52* 479b⟩+≡ (478f) ◁479e

```
"io/ioutil"
"bytes"
```

Chapter 53

Program `translate`: Translate DNA to Protein

Table 53.1: The genetic code; numbers are the codon positions.

	2				
1	T	C	A	G	3
T	F	S	Y	C	T
T	F	S	Y	C	C
T	L	S	*	*	A
T	L	S	*	W	G
C	L	P	H	R	T
C	L	P	H	R	C
C	L	P	Q	R	A
C	L	P	Q	R	G
A	I	T	N	S	T
A	I	T	N	S	C
A	I	T	K	R	A
A	M	T	K	R	G
G	V	A	D	G	T
G	V	A	D	G	C
G	V	G	E	G	A
G	V	A	E	G	G

Introduction

Life is based on the translation of DNA into protein. The program `translate` takes a DNA sequence and prints its translation according to the genetic code shown in Table 53.1. The user can set the translation frame as 1, 2, or 3 on the forward strand and -1, -2, or -3 on the reverse.

Implementation

The implementation of `translate` has hooks for imports, functions, and the logic of the main function.

482 ⟨*translate.go* 482⟩≡

```
package main

import (
        ⟨Imports, Ch. 53 483b⟩
)
⟨Functions, Ch. 53 484d⟩
func main() {
        ⟨Main function, Ch. 53 483a⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, construct the genetic code, and parse the input files.

483a ⟨*Main function, Ch. 53* 483a⟩≡ (482)
```
util.PrepLog("translate")
⟨Set usage, Ch. 53 483c⟩
⟨Declare options, Ch. 53 483e⟩
⟨Parse options, Ch. 53 483g⟩
⟨Construct genetic code, Ch. 53 484a⟩
⟨Parse input files, Ch. 53 484c⟩
```

We import `util`.

483b ⟨*Imports, Ch. 53* 483b⟩≡ (482) 483d▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts, the actual usage message, an explanation of the program's purpose, and an example command.

483c ⟨*Set usage, Ch. 53* 483c⟩≡ (483a)
```
u := "translate [-h] [option]... [foo.fasta]..."
p := "Translate DNA sequences."
e := "translate -f 2 foo.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

483d ⟨*Imports, Ch. 53* 483b⟩+≡ (482) ◁483b 483f▷
```
"github.com/evolbioinf/clio"
```

Apart from the built-in help option (`-h`), we declare an option to select a frame (`-f`) and one for printing the program version (`-v`).

483e ⟨*Declare options, Ch. 53* 483e⟩≡ (483a)
```
var optF = flag.Int("f", 1, "reading frame -3|-2|-1|1|2|3")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

483f ⟨*Imports, Ch. 53* 483b⟩+≡ (482) ◁483d 483h▷
```
"flag"
```

We parse the options and respond to `-v`, as this would stop the program. We also check that `-f` has a sensible value. If not, bail with a friendly message.

483g ⟨*Parse options, Ch. 53* 483g⟩≡ (483a)
```
flag.Parse()
if *optV {
        util.PrintInfo("tranlate")
}
if *optF < -3 || *optF > 3 {
        m := "please use a reading frame " +
                "between -3 and 3"
        log.Fatal(m)
}
```

We import `log`.

483h ⟨*Imports, Ch. 53* 483b⟩+≡ (482) ◁483f 485a▷
```
"log"
```

The genetic code is a mapping of codons to amino acids (Table 53.1). We encode it as a map between strings representing codons and bytes representing amino acids.

484a ⟨*Construct genetic code, Ch. 53* 484a⟩≡ (483a)

```
gc := make(map[string]byte)
dna := "TCAG"
aa := "FFLLSSSSYY**CC*W" +
          "LLLLPPPPHHQQRRRR" +
          "IIIMTTTTNNKKSSRR" +
          "VVVVAAAADDEEGGGG"
codon := make([]byte, 3)
n := 0
⟨Iterate over codons and amino acids, Ch. 53 484b⟩
```

We iterate over the codons and the amino acids using a triple nested loop over the nucleotides in the order in which they are used in Table 53.1, `T`, `C`, `A`, `G`. This allows us to think carefully about genes.

484b ⟨*Iterate over codons and amino acids, Ch. 53* 484b⟩≡ (484a)

```
for i := 0; i < 4; i++ {
          for j := 0; j < 4; j++ {
                  for k := 0; k < 4; k++ {
                          codon[0] = dna[i]
                          codon[1] = dna[j]
                          codon[2] = dna[k]
                          gc[string(codon)] = aa[n]
                          n++
                  }
          }
}
```

The remaining tokens on the command line are taken as input files. We iterate over them with the function `scan`, which takes as arguments the genetic code and the translation frame.

484c ⟨*Parse input files, Ch. 53* 484c⟩≡ (483a)

```
files := flag.Args()
clio.ParseFiles(files, scan, gc, *optF)
```

Inside `scan` we retrieve the options just passed and iterate over the sequences. Each sequence is translated and printed.

484d ⟨*Functions, Ch. 53* 484d⟩≡ (482)

```
func scan(r io.Reader, args ...interface{}) {
          gc := args[0].(map[string]byte)
          frame := args[1].(int)
          sc := fasta.NewScanner(r)
          for sc.ScanSequence() {
                  seq := sc.Sequence()
                  ⟨Translate sequence, Ch. 53 485b⟩
                  ⟨Print translation, Ch. 53 485c⟩
          }
}
```

We import `io`.

485a ⟨*Imports, Ch. 53* 483b⟩+≡ (482) ◁483h 485d▷
```
"io"
```

We translate a sequence

485b ⟨*Translate sequence, Ch. 53* 485b⟩≡ (484d)
```
if frame < 0 {
          seq.ReverseComplement()
          frame *= -1
}
d := seq.Data()
var aa []byte
for i := frame-1; i < len(seq.Data())-2; i += 3 {
          codon := string(d[i:i+3])
          aa = append(aa, gc[codon])
}
```

We construct a new sequence from the translation. Its header is the original header with "`- translated`" appended. We print the new sequence using its `String` method.

485c ⟨*Print translation, Ch. 53* 485c⟩≡ (484d)
```
h := seq.Header() + " - translated"
aaSeq := fasta.NewSequence(h, aa)
fmt.Println(aaSeq)
```

We import `fasta` an `fmt`.

485d ⟨*Imports, Ch. 53* 483b⟩+≡ (482) ◁485a
```
"github.com/evolbioinf/fasta"
"fmt"
```

We are finished with `translate`, let's test it.

Testing

The outline of our testing program has hooks for imports and the testing logic.

485e ⟨*translate_test.go* 485e⟩≡
```
package main

import (
          "testing"
          ⟨Testing imports, Ch. 53 486b⟩
)
func TestTranslate(t *testing.T) {
          ⟨Testing, Ch. 53 486a⟩
}
```

We construct a set of tests and then iterate over them.

486a ⟨*Testing, Ch. 53* 486a⟩≡ (485e)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 53 486c⟩
for i, test := range tests {
        ⟨Run test, Ch. 53 486f⟩
}
```

We import `exec`.

486b ⟨*Testing imports, Ch. 53* 486b⟩≡ (485e) 487▷
```
"os/exec"
```

We test translation on the forward and on the reverse strands. The input file is always `test.fasta`, a random sequence.

486c ⟨*Construct tests, Ch. 53* 486c⟩≡ (486a)
```
f := "test.fasta"
⟨Test forward translation, Ch. 53 486d⟩
⟨Test reverse translation, Ch. 53 486e⟩
```

We construct four forward tests, one with the default frame, the other three for frames 1, 2, and 3.

486d ⟨*Test forward translation, Ch. 53* 486d⟩≡ (486c)
```
test := exec.Command("./translate", f)
tests = append(tests, test)
test = exec.Command("./translate", "-f", "1", f)
tests = append(tests, test)
test = exec.Command("./translate", "-f", "2", f)
tests = append(tests, test)
test = exec.Command("./translate", "-f", "3", f)
tests = append(tests, test)
```

We also go through the three reverse frames.

486e ⟨*Test reverse translation, Ch. 53* 486e⟩≡ (486c)
```
test = exec.Command("./translate", "-f", "-1", f)
tests = append(tests, test)
test = exec.Command("./translate", "-f", "-2", f)
tests = append(tests, test)
test = exec.Command("./translate", "-f", "-3", f)
tests = append(tests, test)
```

We run a test and compare the output we get with the precomputed output we want, which is stored in files `r1.fasta`, `r2.fasta`, and so on.

486f ⟨*Run test, Ch. 53* 486f⟩≡ (486a)
```
get, err := test.Output()
if err != nil { t.Errorf("couldn't run %q", test) }
f := "r" + strconv.Itoa(i+1) + ".fasta"
want, err := ioutil.ReadFile(f)
if err != nil { t.Errorf("couldn't read %q", f) }
if !bytes.Equal(get, want) {
        t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

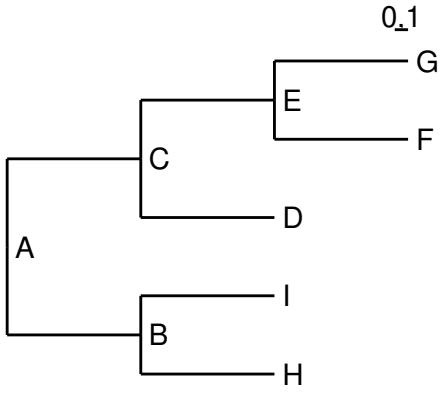
487 ⟨*Testing imports, Ch. 53* 486b⟩+≡ (485e) ◁486b

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 54

Program `travTree`: Traverse Phylogeny

A



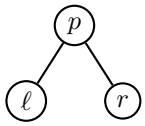
B

```
((H,I)B,(D,(F,G)E)C)A;
```

Figure 54.1: A phylogeny (**A**) and its Newick version (**B**).

Introduction

In a phylogeny each internal node is usually the parent of two children, one on the left, the other on the right:



To traverse such a binary tree, we recursively visit this trio of nodes, parent, left child, right child. Depending on whether the parent is visited first, second, or last, such a traversal is called preorder, inorder, or postorder [22, p. 318f]. These three traversal modes visit the nodes of a tree in characteristic patterns. Take for example the Tree in Figure 54.1A. When traversed preorder, its nodes are visited

A, B, H, I, C, D, E, F, G.

When visited inorder, its nodes are visited

H, I, B, D, F, G, E, C, A.

Finally, when visited postorder, its nodes are visited

I, H, G, F, E, D, C, B, A.

Table 54.1: Preorder traversal of the tree in Figure 54.1.

Label	Parent	Distance	Type
A	none	0	root
B	A	0	internal
H	B	0	leaf
I	B	0	leaf
C	A	0	internal
D	C	0	leaf
E	C	0	internal
F	E	0	leaf
G	E	0	leaf

The program `travTree` takes a tree in Newick format, like the one shown in Figure 54.1B, and prints a table of the node it visits in preorder, inorder, or postorder. For each node `travTree` prints the label, the node's parent, the length of its incoming branch, and the node type. Table 54.1 shows the preorder table for the example tree.

Implementation

The outline of `travTree` has hooks for imports, functions, and the logic of the main function.

490a ⟨*travTree.go* 490a⟩≡
```
package main

import (
        ⟨Imports, Ch. 54 490c⟩
)
⟨Functions, Ch. 54 492b⟩
func main() {
        ⟨Main function, Ch. 54 490b⟩
}
```

In the main function we prepare the `log` package, set the usage, declare the options, parse the options, prepare the output table, and parse the input files.

490b ⟨*Main function, Ch. 54* 490b⟩≡ (490a)
```
util.PrepLog("travTree")
⟨Set usage, Ch. 54 491a⟩
⟨Declare options, Ch. 54 491c⟩
⟨Parse options, Ch. 54 491e⟩
⟨Prepare output table, Ch. 54 491g⟩
⟨Parse input files, Ch. 54 492a⟩
```

We import `util`.

490c ⟨*Imports, Ch. 54* 490c⟩≡ (490a) 491b ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts, the actual usage message, an explanation of the program's purpose, and an example command.

491a ⟨*Set usage, Ch. 54* 491a⟩≡ (490b)
```
u := "travTree [-h] [option]... [foo.nwk]..."
p := "Traverse a tree given in Newick format."
e := "travTree -i foo.nwk"
clio.Usage(u, p, e)
```

We import `clio`.

491b ⟨*Imports, Ch. 54* 490c⟩+≡ (490a) ◁490c 491d▷
```
"github.com/evolbioinf/clio"
```

Apart from the built-in help option (`-h`), we declare switches for inorder (`-i`) and postorder (`-o`). If neither of these is used, the traversal is preorder. The user can also request the program version (`-v`).

491c ⟨*Declare options, Ch. 54* 491c⟩≡ (490b)
```
var optI = flag.Bool("i", false, "inorder")
var optO = flag.Bool("o", false, "postorder")
var optV = flag.Bool("v", false, "version")
```

We import `flag`.

491d ⟨*Imports, Ch. 54* 490c⟩+≡ (490a) ◁491b 491f▷
```
"flag"
```

We parse the options and respond to `-v`, as this stops the program. We make sure the user opted for only one traversal mode.

491e ⟨*Parse options, Ch. 54* 491e⟩≡ (490b)
```
flag.Parse()
if *optV {
        util.PrintInfo("travTree")
}
if *optI && *optO {
        log.Fatal("please opt for just one traversal mode")
}
```

We import `log`.

491f ⟨*Imports, Ch. 54* 490c⟩+≡ (490a) ◁491d 491h▷
```
"log"
```

The output table is written using a `tabwriter`. This writes to the standard output stream and uses blanks for padding.

491g ⟨*Prepare output table, Ch. 54* 491g⟩≡ (490b)
```
out := tabwriter.NewWriter(os.Stdout, 2, 1, 2, ' ', 0)
```

We import `tabwriter` and `os`.

491h ⟨*Imports, Ch. 54* 490c⟩+≡ (490a) ◁491f 492c▷
```
"text/tabwriter"
"os"
```

The remaining tokens on the input line are taken as file names. These files are parsed by applying the function `scan` to each one. The function `scan` takes as arguments the two options that determine the order of traversal, the `tabwriter`, and an indicator of whether we are dealing with the first tree in a potentially longer list.

492a ⟨*Parse input files, Ch. 54* 492a⟩≡ (490b)
```
files := flag.Args()
first := true
clio.ParseFiles(files, scan, *optI, *optO, out, &first)
```

Inside `scan`, we iterate over the trees. For each tree we print a table header, then traverse the tree, and afterwards flush the `tabwriter`. We also track whether we are dealing with the first tree.

492b ⟨*Functions, Ch. 54* 492b⟩≡ (490a) 493b ▷
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Retrieve arguments, Ch. 54 492d⟩
        sc := nwk.NewScanner(r)
        for sc.Scan() {
                ⟨Dealing with first tree? Ch. 54 492e⟩
                fmt.Fprint(out, "#Label\tParent\tDist.\tType\n")
                root := sc.Tree()
                ⟨Traverse tree, Ch. 54 493a⟩
                out.Flush()
        }
}
```

We import `io`, `nwk`, and `fmt`.

492c ⟨*Imports, Ch. 54* 490c⟩+≡ (490a) ◁491h
```
"io"
"github.com/evolbioinf/nwk"
"fmt"
```

We retrieve the arguments through type assertion.

492d ⟨*Retrieve arguments, Ch. 54* 492d⟩≡ (492b)
```
io := args[0].(bool)
po := args[1].(bool)
out := args[2].(*tabwriter.Writer)
first := args[3].(*bool)
```

If we are dealing with the first tree, we toggle `first`. Otherwise, we print a blank line to offset the next node table.

492e ⟨*Dealing with first tree? Ch. 54* 492e⟩≡ (492b)
```
if *first {
        *first = false
} else {
        fmt.Fprint(out, "\n")
}
```

A tree is traversed inorder, postorder, or preorder.

493a ⟨*Traverse tree, Ch. 54* 493a⟩≡ (492b)
```
if io {
          inorder(root, out)
} else if po {
          postorder(root, out)
} else {
          preorder(root, out)
}
```

During inorder traversal we determine the node type and then print a row in the node table.

493b ⟨*Functions, Ch. 54* 492b⟩+≡ (490a) ◁492b 493e▷
```
func inorder(v *nwk.Node, w *tabwriter.Writer) {
          if v == nil { return }
          inorder(v.Child, w)
          ⟨Determine node type, Ch. 54 493c⟩
          ⟨Print row in node table, Ch. 54 493d⟩
          inorder(v.Sib, w)
}
```

A node is either a leaf, an internal node, or the root.

493c ⟨*Determine node type, Ch. 54* 493c⟩≡ (493 494a)
```
typ := "leaf"
if v.Parent == nil {
          typ = "root"
} else if v.Child != nil {
          typ = "internal"
}
```

A row in the node table consists of the label, the parent's label, if there is a parent, the branch length, and the node type.

493d ⟨*Print row in node table, Ch. 54* 493d⟩≡ (493 494a)
```
p := "none"
if v.Parent != nil {
          p = v.Parent.Label
}
fmt.Fprintf(w, "%s\t%s\t%.3g\t%s\n",
          v.Label, p, v.Length, typ)
```

We implement postorder traversal.

493e ⟨*Functions, Ch. 54* 492b⟩+≡ (490a) ◁493b 494a▷
```
func postorder(v *nwk.Node, w *tabwriter.Writer) {
          if v == nil { return }
          postorder(v.Child, w)
          postorder(v.Sib, w)
          ⟨Determine node type, Ch. 54 493c⟩
          ⟨Print row in node table, Ch. 54 493d⟩
}
```

The last traversal type we implement is preorder.

494a ⟨*Functions, Ch. 54* 492b⟩+≡ (490a) ◁493e
```
func preorder(v *nwk.Node, w *tabwriter.Writer) {
        if v == nil { return }
        ⟨Determine node type, Ch. 54 493c⟩
        ⟨Print row in node table, Ch. 54 493d⟩
        preorder(v.Child, w)
        preorder(v.Sib, w)
}
```

The program `travTree` is finished, so we test it.

Testing

The outline of our testing program has hooks for imports and the testing logic.

494b ⟨*travTree_test.go* 494b⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 54 494d⟩
)

func TestTravTree(t *testing.T) {
        ⟨Testing, Ch. 54 494c⟩
}
```

We construct a set of tests and then iterate over them.

494c ⟨*Testing, Ch. 54* 494c⟩≡ (494b)
```
var tests []*exec.Cmd
⟨Construct tests, Ch. 54 494e⟩
for i, test := range tests {
        ⟨Run test, Ch. 54 495a⟩
}
```

We import `exec`.

494d ⟨*Testing imports, Ch. 54* 494d⟩≡ (494b) 495b▷
```
"os/exec"
```

We construct three tests, preorder, inorder, and postorder. Every time we analyze the tree in file `test.nwk`.

494e ⟨*Construct tests, Ch. 54* 494e⟩≡ (494c)
```
f := "test.nwk"
test := exec.Command("./travTree", f)
tests = append(tests, test)
test = exec.Command("./travTree", "-i", f)
tests = append(tests, test)
test = exec.Command("./travTree", "-o", f)
tests = append(tests, test)
```

We run a test and compare what we get with what we want, which is stored in files `r1.txt`, `r2.txt`, and `r3.txt`.

495a ⟨*Run test, Ch. 54* 495a⟩≡ (494c)

```
get, err := test.Output()
if err != nil {
          t.Errorf("couldn't run %q", test)
}
f := "r" + strconv.Itoa(i+1) + ".txt"
want , err := ioutil.ReadFile(f)
if err != nil {
          t.Errorf("couldn't open %q", f)
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `strconv`, `ioutil`, and `bytes`.

495b ⟨*Testing imports, Ch. 54* 494d⟩+≡ (494b) ◁494d

```
"strconv"
"io/ioutil"
"bytes"
```

Chapter 55

Program `upgma`: Compute UPGMA Tree

A

5					
S1	0	0.19	0.00502	0.0537	0.186
S2	0.19	0	0.19	0.188	0.0489
S3	0.00502	0.19	0	0.0539	0.186
S4	0.0537	0.188	0.0539	0	0.184
S5	0.186	0.0489	0.186	0.184	0

B

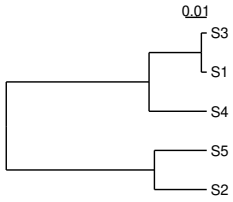


Figure 55.1: A distance matrix (**A**) gets converted by `upgma` into a UPGMA tree (**B**); tree plotted with `plotTree`.

Introduction

Of the many methods available for phylogeny reconstruction, distance methods are perhaps the simplest. The program `upgma` implements the simplest among those, UPGMA [25, Ch. 10.3]. Input is a distance matrix in PHYLIP format for example the one shown in (Figure 55.1A). For this, `upgma` returns either the tree in Figure 55.1B. Upon request, `upgma` also prints the intermediate distance matrices generated by this algorithm.

Implementation

The outline of `upgma` contains hooks for imports, functions, and the logic of the main function.

497a ⟨*upgma.go* 497a⟩≡

```
package main

import (
        ⟨Imports, Ch. 55 497c⟩
)
⟨Functions, Ch. 55 498g⟩
func main() {
        ⟨Main function, Ch. 55 497b⟩
}
```

In the main function, we prepare the `log` package, set the usage, declare the options, parse the options, and scan the input files.

497b ⟨*Main function, Ch. 55* 497b⟩≡ (497a)

```
util.PrepLog("upgma")
⟨Set usage, Ch. 55 498a⟩
⟨Declare options, Ch. 55 498c⟩
⟨Parse options, Ch. 55 498e⟩
⟨Scan input files, Ch. 55 498f⟩
```

We import `util`.

497c ⟨*Imports, Ch. 55* 497c⟩≡ (497a) 498b ▷

```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage message, an explanation of the purpose of `upgma`, and an example command.

498a ⟨*Set usage, Ch. 55* 498a⟩≡ (497b)
```
u := "upgma [-h] [option]... [foo.dist]..."
p := "Cluster a distance matrix into a tree using UPGMA."
e := "upgma foo.dist"
clio.Usage(u, p, e)
```

We import `clio`.

498b ⟨*Imports, Ch. 55* 497c⟩+≡ (497a) ◁497c 498d▷
```
"github.com/evolbioinf/clio"
```

Apart from the version (`-v`), we declare an option for printing the intermediate matrices (`-m`).

498c ⟨*Declare options, Ch. 55* 498c⟩≡ (497b)
```
var optV = flag.Bool("v", false, "version")
var optM = flag.Bool("m", false, "print intermediate " +
          "matrices")
```

We include `flag`.

498d ⟨*Imports, Ch. 55* 497c⟩+≡ (497a) ◁498b 498h▷
```
"flag"
```

We parse the options and respond to `-v`, as this terminates the program.

498e ⟨*Parse options, Ch. 55* 498e⟩≡ (497b)
```
flag.Parse()
if *optV {
          util.PrintInfo("upgma")
}
```

The remaining tokens on the command line are interpreted as file names. We scan each file with the function `scan`, which takes as arguments the two options that influence tree computation, `-u` and `-m`.

498f ⟨*Scan input files, Ch. 55* 498f⟩≡ (497b)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optM)
```

Inside `scan`, we retrieve the option just passed, and iterate over the distance matrices in the input.

498g ⟨*Functions, Ch. 55* 498g⟩≡ (497a) 502c▷
```
func scan(r io.Reader, args ...interface{}) {
          printMat := args[0].(bool)
          sc := dist.NewScanner(r)
          for sc.Scan() {
                    dm := sc.DistanceMatrix()
                    ⟨Process distance matrix, Ch. 55 499a⟩
          }
}
```

We import `io` and `dist`.

498h ⟨*Imports, Ch. 55* 497c⟩+≡ (497a) ◁498d 499b▷
```
"io"
"github.com/evolbioinf/dist"
```

The first step to process a distance matrix is to make it symmetrical and to store its dimension. Then the matrix is converted into a tree, represented by its root, and printed.

499a ⟨*Process distance matrix, Ch. 55* 499a⟩≡ (498g)

```
dm.MakeSymmetrical()
n := len(dm.Names)
var root *nwk.Node
⟨Calculate tree, Ch. 55 500⟩
fmt.Println(root)
```

We import `nwk` and `fmt`.

499b ⟨*Imports, Ch. 55* 497c⟩+≡ (497a) ◁498h

```
"github.com/evolbioinf/nwk"
"fmt"
```

We calculate the tree using two data structures: Our $n \times n$ distance matrix, d, and an array of n tree nodes, t. Tree construction consists of picking pairs of children from t and clustering them in parent nodes, r. A pair of children i, j, has the smallest entry in d, and the height of their parent is $d_{ij}/2$.

The children are removed from d and from t, and replaced by r, so a parent becomes a child in the next round.

The distance between r and the remaining nodes, k, is the average distance to the children clustered:

$$d_{rk} = (d_{ki} + d_{kj})/2.$$

After the last round, r is the root of the desired tree. Algorithm 6 summarizes these steps.

Algorithm 6 The UPGMA clustering algorithm.

Require: n {sample size}
Require: d {$n \times n$ distance matrix}
Require: t {n nodes}
Ensure: Root of tree, r

1: **for** $i \leftarrow n$ **to** 2 **do**
2: Find smallest entry in d, d_{jk}
3: Construct r as parent of t_j and t_k
4: height$(r) \leftarrow d_{jk}/2$
5: $d_{r.} \leftarrow (d_{j.} + d_{k.})/2$
6: $d \leftarrow d \backslash \{d_j, d_k\}$
7: $t \leftarrow c \backslash \{t_j, t_k\}$
8: $t \leftarrow c \cup r$
9: **end for**
10: Convert node heights to branch lengths

When we calculate a tree, we begin by constructing the node array. Then we iterate until the node array contains only two entries. In each iteration, we might print the distance matrix, if so desired. Then we pick a pair of nodes, cluster them, and replace them. Once the tree has been constructed, we convert the node heights to branch lengths.

500 ⟨*Calculate tree, Ch. 55* 500⟩≡ (499a)

```
⟨Construct node array, Ch. 55 501a⟩
for i := n; i > 1; i-- {
        if printMat {
                fmt.Printf("%s", dm)
        }
        ⟨Pick nodes to be clustered, Ch. 55 501b⟩
        ⟨Cluster nodes, Ch. 55 501c⟩
        ⟨Replace clustered nodes, Ch. 55 501d⟩
}
⟨Convert node heights to branch lengths, Ch. 55 502b⟩
```

The node array initially consists of n leaves.

501a ⟨*Construct node array, Ch. 55* 501a⟩≡ (500)
```
t := make([]*nwk.Node, n)
for i := 0; i < n; i++ {
          t[i] = nwk.NewNode()
          t[i].Label = dm.Names[i]
}
```

By finding the minimum matrix entry, we find the nodes to be clustered.

501b ⟨*Pick nodes to be clustered, Ch. 55* 501b⟩≡ (500)
```
md, mj, mk := dm.Min()
c1 := t[mj]
c2 := t[mk]
root = nwk.NewNode()
l := fmt.Sprintf("(%s,%s)", c1.Label, c2.Label)
root.Label = l
root.Length = md / 2
```

We cluster the nodes by adding nodes `c1` and `c2` as children of `root`.

501c ⟨*Cluster nodes, Ch. 55* 501c⟩≡ (500)
```
root.AddChild(c1)
root.AddChild(c2)
```

We replace the matrix entries and the entries in the nodes array.

501d ⟨*Replace clustered nodes, Ch. 55* 501d⟩≡ (500)
```
⟨Replace matrix entries, Ch. 55 501e⟩
⟨Replace entries in node array, Ch. 55 502a⟩
```

We replace the matrix entries by computing the distances between the new node and all other nodes. Then we delete the child nodes from the distance matrix and replace them by appending the new distances. The label of the new cluster is constructed from the labels of its children.

501e ⟨*Replace matrix entries, Ch. 55* 501e⟩≡ (501d)
```
data := make([]float64, i-2)
k := 0
for j := 0; j < i; j++ {
          if j == mj || j == mk { continue }
          data[k] = (dm.Matrix[j][mj] + dm.Matrix[j][mk]) / 2.0
          k++
}
dm.DeletePair(mj, mk)
dm.Append(root.Label, data)
```

We remove the nodes picked from the node array and append the current root.

502a ⟨*Replace entries in node array, Ch. 55* 502a⟩≡ (501d)
```
j := 0
for k := 0; k < i; k++ {
        if k == mj || k == mk { continue }
        t[j] = t[k]
        j++
}
t = t[:j]
t = append(t, root)
```

What remains, is to convert the node heights into branch lengths. We do this by calling a function for tree traversal. In this step we also remove the node labels again, as they were only useful for printing the matrix. In the actual phylogeny, only the leaves have labels.

502b ⟨*Convert node heights to branch lengths, Ch. 55* 502b⟩≡ (500)
```
branchLengths(root)
```

Branch lengths are computed by subtracting the height of the child from that of the parent. Let's also not forget to reset the labels of internal nodes.

502c ⟨*Functions, Ch. 55* 498g⟩+≡ (497a) ◁498g
```
func branchLengths(v *nwk.Node) {
        if v == nil { return }
        branchLengths(v.Child)
        branchLengths(v.Sib)
        if v.Child != nil { v.Label = "" }
        if v.Parent != nil {
                v.Length = v.Parent.Length - v.Length
                v.HasLength = true
        }
}
```

The program `upgma` is finished, time to test it.

Testing

The outline of our testing code has hooks for imports and the testing logic.

502d ⟨*upgma_test.go* 502d⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 55 503b⟩
)

func TestUpgma(t *testing.T) {
        ⟨Testing, Ch. 55 503a⟩
}
```

We test our program by running it on the distance matrix shown in Figure 55.1A, which is contained in the file `test.phy`. Then we compare the output we get with the output we want, which is stored in the file `r.txt`.

503a ⟨*Testing, Ch. 55* 503a⟩≡ (502d)

```
cmd := exec.Command("./upgma", "-m", "test.phy")
get, err := cmd.Output()
if err != nil {
          t.Errorf("can't run %q", cmd)
}
want, err := ioutil.ReadFile("r.txt")
if err != nil {
          t.Errorf("can't open r.txt")
}
if !bytes.Equal(get, want) {
          t.Errorf("get:\n%s\nwant:\n%s\n", get, want)
}
```

We import `exec`, `ioutil`, and `bytes`.

503b ⟨*Testing imports, Ch. 55* 503b⟩≡ (502d)

```
"os/exec"
"io/ioutil"
"bytes"
```

Chapter 56

Package `util`: Utilities

!Package `util` contains data and functions used by many of the !programs collected in the `biobox`.

The package outline provides hooks for imports, constants, types, variables, methods, and functions. Some of the computations are delegated to the GNU Scientific library, which is written in C. We therefore also provide hooks for dealing with C.

505a ⟨*util.go* 505a⟩≡
```
package util
⟨Deal with C, Ch. 56 505b⟩
import (
        ⟨Imports, Ch. 56 506c⟩
)
const (
        ⟨Constants, Ch. 56 510e⟩
)
⟨Types, Ch. 56 505c⟩
⟨Variables, Ch. 56 511c⟩
⟨Methods, Ch. 56 506d⟩
⟨Functions, Ch. 56 506a⟩
```

To deal with C, we import `C`. Our bridge between Go and C is cgo, so there is a hook for cgo commands, and one for includes.

505b ⟨*Deal with C, Ch. 56* 505b⟩≡ (505a)
```
/*
⟨Cgo, Ch. 56 519b⟩
⟨Includes, Ch. 56 519c⟩
*/
import "C"
```

56.1 Structure `Alignment`

!The structure `Alignment` holds the alignment of two !sequences.

This structure is intended for uniform printing of pairwise sequence alignments. It holds two sequences, the score matrix, start positions of the two alignments—important for local alignments—the line length in the output, and the score.

505c ⟨*Types, Ch. 56* 505c⟩≡ (505a) 511d▷
```
type Alignment struct {
        sequence1, sequence2 *fasta.Sequence
        scoreMatrix *ScoreMatrix
        length1, length2, start1, start2, lineLength int
        score float64
}
```

Function NewAlignment

!NewAlignment takes as arguments two aligned sequences, the !score matrix used in computing the alignment, lengths of the two !sequences, start positions in the two sequences, and the score. The !start positions are zero-based.

506a ⟨*Functions, Ch. 56* 506a⟩≡ (505a) 510b ▷
```
func NewAlignment(seq1, seq2 *fasta.Sequence, sm *ScoreMatrix,
        l1, l2, s1, s2 int, score float64) *Alignment {
        al := new(Alignment)
        ⟨Set Alignment fields, Ch. 56 506b⟩
        return al
}
```

506b ⟨*Set* `Alignment` *fields, Ch. 56* 506b⟩≡ (506a)
```
al.sequence1 = seq1
al.sequence2 = seq2
al.scoreMatrix = sm
al.lineLength = fasta.DefaultLineLength
al.length1 = l1
al.length2 = l2
al.start1 = s1
al.start2 = s2
al.score = score
```

We import fasta.

506c ⟨*Imports, Ch. 56* 506c⟩≡ (505a) 507b ▷
```
"github.com/evolbioinf/fasta"
```

Method SetLineLength

An Alignment is immutable. Only the line length affects its appearance and hence is accessible via a setter. !SetLineLength sets the lengths of data lines in the printout !of an alignment. If the length passed is less than one, no change is !made.

506d ⟨*Methods, Ch. 56* 506d⟩≡ (505a) 507a ▷
```
func (a *Alignment) SetLineLength(l int) {
        if l > 0 {
                a.lineLength = l
        }
}
```

Method `String`

!`String` converts an `Alignment` into a !printable string.

This string has two parts, a header and the actual alignment. They are generated by writing to a byte buffer. An `Alignment` is terminated by `//`.

507a ⟨*Methods, Ch. 56* 506d⟩+≡ (505a) ◁506d 515a▷

```
func (a *Alignment) String() string {
        var buf []byte
        buffer := bytes.NewBuffer(buf)
        ⟨Write header, Ch. 56 507c⟩
        ⟨Write data, Ch. 56 508a⟩
        buffer.Write([]byte("//"))
        return buffer.String()
}
```

We import `bytes`.

507b ⟨*Imports, Ch. 56* 506c⟩+≡ (505a) ◁506c 507e▷

```
"bytes"
```

We write the header as a table, which is formatted via `tabwriter`.

507c ⟨*Write header, Ch. 56* 507c⟩≡ (507a)

```
⟨Construct tabwriter, Ch. 56 507d⟩
⟨Write header to tabwriter, Ch. 56 507f⟩
```

Our `tabwriter` writes to the `buffer`. Its minimal cell width is 1, the width of the tab characters is zero, a single character is added for padding, which is done by blanks.

507d ⟨*Construct tabwriter, Ch. 56* 507d⟩≡ (507c)

```
w := new(tabwriter.Writer)
w.Init(buffer, 1, 0, 1, ' ', 0)
```

We import `tabwriter`.

507e ⟨*Imports, Ch. 56* 506c⟩+≡ (505a) ◁507b 511b▷

```
"text/tabwriter"
```

The header consists of the names of the sequences, their role (query or subject), their lengths, and the alignment score.

507f ⟨*Write header to tabwriter, Ch. 56* 507f⟩≡ (507c)

```
s1 := a.sequence1
s2 := a.sequence2
al := len(s2.Data())
l1 := a.length1
l2 := a.length2
fmt.Fprintf(w, "Query\t%s\t(%d residues)\t\n", s1.Header(), l1)
fmt.Fprintf(w, "Subject\t%s\t(%d residues)\t\n", s2.Header(), l2)
fmt.Fprintf(w, "Score\t%g\t\n", a.score)
w.Flush()
```

```
Query: 1    MKFLAL-F  7
            ||:| | |
Subject: 1  MKYLILLF  8
```

Figure 56.1: Example alignment with the match characters sandwiched by the sequences.

Having completed the header, we loop over the alignment and format it in triplets of lines consisting of the two sequences sandwiching a row of match characters. Figure 56.1 shows an example, the alignment of two short peptides. Pairs of identical residues get vertical lines, distinct residues with scores greater than zero like phenylalanine (`F`) and tyrosine (`Y`) a colon, and mismatches or gaps blanks. We use the same `tabwriter` as for the header, and as for the header.

508a ⟨*Write data, Ch. 56* 508a⟩≡ (507a)

```
⟨Declare variables, Ch. 56 508c⟩
for i := 0; i < al; i += a.lineLength {
        ⟨Compute line end, Ch. 56 508b⟩
        ⟨Store sequences and match characters, Ch. 56 508d⟩
}
w.Flush()
```

The variable `i` refers to the beginning of the line. Its end is either the beginning plus line length, or, if fewer residues than "line length" remain, the end of the alignment.

508b ⟨*Compute line end, Ch. 56* 508b⟩≡ (508a)

```
if i + a.lineLength < al {
        end = i + a.lineLength
} else {
        end = al
}
```

We declare the variable `end`.

508c ⟨*Declare variables, Ch. 56* 508c⟩≡ (508a) 509b ▷

```
var end int
```

In an alignment, two sequences sandwich a row of match characters (Figure 56.1). And while the sequences can be used as supplied, we still need to determine the match characters.

508d ⟨*Store sequences and match characters, Ch. 56* 508d⟩≡ (508a)

```
⟨Store first sequence, Ch. 56 509a⟩
for j := i; j < end; j++ {
        ⟨Create slice of match characters, Ch. 56 509d⟩
}
⟨Store match characters, Ch. 56 509e⟩
⟨Store second sequence, Ch. 56 509f⟩
```

We generate a row-length slice of the first sequence in the alignment. The residues in this slice are its length minus the number of gaps. From the number of residues we compute the start and end positions in the underlying sequence. If the row contains at least one residue, the left border of the interval is advanced by one from the previous end, otherwise it remains unchanged.

509a ⟨*Store first sequence, Ch. 56* 509a⟩≡ (508d)
```
data := s1.Data()[i:end]
nr := len(data) - bytes.Count(data, []byte("-"))
l := st1
if nr > 0 { l++ }
fmt.Fprintf(w, "\n\nQuery\t%d\t%s\t%d\t\n", l, data, st1+nr)
st1 += nr
```

Here `s1` is the start of the first sequence.

509b ⟨*Declare variables, Ch. 56* 508c⟩+≡ (508a) ◁508c 509c▷
```
st1 := a.start1
```

The match characters are stored in a byte slice and are determined using the score matrix.

509c ⟨*Declare variables, Ch. 56* 508c⟩+≡ (508a) ◁509b 510a▷
```
var matches []byte
sc := a.scoreMatrix
```

If neither of the characters compared is a gap, their match character is decided by looking up the score.

509d ⟨*Create slice of match characters, Ch. 56* 509d⟩≡ (508d)
```
c1 := s1.Data()[j]
c2 := s2.Data()[j]
m := byte(' ')
if c1 != '-' && c2 != '-' {
          if c1 == c2 {
                  m = '|'
          } else if sc.Score(c1, c2) > 0 {
                  m = ':'
          }
}
matches = append(matches, m)
```

The match characters are handed to the `tabwriter` and reset.

509e ⟨*Store match characters, Ch. 56* 509e⟩≡ (508d)
```
fmt.Fprintf(w, "\t\t%s\t\t\n", string(matches))
matches = matches[:0]
```

To close the alignment sandwich, we add the second sequence, again framed by its start and end in the original.

509f ⟨*Store second sequence, Ch. 56* 509f⟩≡ (508d)
```
data = s2.Data()[i:end]
nr = len(data) - bytes.Count(data, []byte("-"))
l = st2
if nr > 0 { l++ }
fmt.Fprintf(w, "Subject\t%d\t%s\t%d\t\n", l, data, st2+nr)
st2 += nr
```

We declare and initialize the start of the second sequence.

510a ⟨*Declare variables, Ch. 56* 508c⟩+≡ (508a) ◁509c
```
st2 := a.start2
```

56.2 Function `MeanVar`

!`MeanVar` takes as input a data set and returns its mean and !sample variance.

510b ⟨*Functions, Ch. 56* 506a⟩+≡ (505a) ◁506a 511a▷
```
func MeanVar(data []float64) (float64, float64) {
        var m, v float64
        ⟨Calculate mean, Ch. 56 510c⟩
        ⟨Calculate variance, Ch. 56 510d⟩
        return m, v
}
```

We calculate the mean,

510c ⟨*Calculate mean, Ch. 56* 510c⟩≡ (510b)
```
n := len(data)
for i := 0; i < n; i++ {
        m += data[i]
}
m /= float64(n)
```

and the variance.

510d ⟨*Calculate variance, Ch. 56* 510d⟩≡ (510b)
```
for i := 0; i < n; i++ {
        s := m - data[i]
        v += s * s
}
v /= float64(n - 1)
```

56.3 Function `PrintInfo`

Each program in the biobox provides the same information on author, email, and license. These are stored as constants.

510e ⟨*Constants, Ch. 56* 510e⟩≡ (505a)
```
author = "Bernhard Haubold"
email = "haubold@evolbio.mpg.de"
license = "Gnu General Public License, " +
        "https://www.gnu.org/licenses/gpl.html"
```

!`PrintInfo` prints a program's name, version, and compilation !date. It also prints the author, email address, and license of the !`biobox` package. Then it exits. To achieve this, we wrap the !generic function for printing program information from the package !`clio`.

511a ⟨*Functions, Ch. 56* 506a⟩+≡ (505a) ◁510b 511e▷
```
func PrintInfo(name string) {
        clio.PrintInfo(name, version, date, author, email,
                license)
        os.Exit(0)
}
```

We import `clio` and `os`.

511b ⟨*Imports, Ch. 56* 506c⟩+≡ (505a) ◁507e 514a▷
```
"github.com/evolbioinf/clio"
"os"
```

The values of `Version` and `Date` are injected at compile-time. Here we just declare them.

511c ⟨*Variables, Ch. 56* 511c⟩≡ (505a)
```
var version string
var date string
```

56.4 Structure `ScoreMatrix`

!A `ScoreMatrix` stores the scores of residue pairs. To save space, scores are 32-bit floats.

511d ⟨*Types, Ch. 56* 505c⟩+≡ (505a) ◁505c 515d▷
```
type ScoreMatrix struct {
        m [][]float32
}
```

Function `NewScoreMatrix`

!Function `NewScoreMatrix` generates a new score matrix, !takes as input a match and a mismatch score, and stores them.

511e ⟨*Functions, Ch. 56* 506a⟩+≡ (505a) ◁511a 512d▷
```
func NewScoreMatrix(match, mismatch float64) *ScoreMatrix {
        sm := new(ScoreMatrix)
        ⟨Allocate score matrix, Ch. 56 512a⟩
        ⟨Fill-in match scores, Ch. 56 512b⟩
        ⟨Fill-in mismatch scores, Ch. 56 512c⟩
        return sm
}
```

There are 95 printing ASCII characters. Not all of these are valid residues, but we simplify matters by allocating a 95×95 matrix.

512a ⟨*Allocate score matrix, Ch. 56* 512a⟩≡ (511e)
```
n := 95
sm.m = make([][]float32, n)
for i := 0; i < n; i++ {
        sm.m[i] = make([]float32, n)
}
```

The match scores are on its main diagonal.

512b ⟨*Fill-in match scores, Ch. 56* 512b⟩≡ (511e)
```
for i := 0; i < n; i++ {
        sm.m[i][i] = float32(match)
}
```

The mismatch scores are everywhere else.

512c ⟨*Fill-in mismatch scores, Ch. 56* 512c⟩≡ (511e)
```
for i := 0; i < n - 1; i++ {
        for j := i + 1; j < n; j++ {
                sm.m[i][j] = float32(mismatch)
                sm.m[j][i] = sm.m[i][j]
        }
}
```

Function ReadScores

!`ReadScores` reads scores from an `io.Reader`.

Figure 56.2 shows the BLOSUM62 score matrix. It starts with optional fenced-off comment lines, followed by a row of residues as column headers. Next are rows of values, each preceded by the residue in question. Columns are delimited by white space.

For reading scores, we prepare two variables. The boolean `first` marks the first line of the table; the slice of byte slices `res` holds the residues used as column headers in that line. Then we scan the input.

512d ⟨*Functions, Ch. 56* 506a⟩+≡ (505a) ◁511e 516a▷
```
func ReadScoreMatrix(r io.Reader) *ScoreMatrix {
        s := NewScoreMatrix(1, -1)
        first := true
        var res [][]byte
        sc := bufio.NewScanner(r)
        for sc.Scan() {
                b := sc.Bytes()
                ⟨Deal with score table, Ch. 56 514b⟩
        }
        return s
}
```

```
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1 -1 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4  1 -1 -4
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -3 -1 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0  3 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0  0 -1 -4
I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3 -3 -1 -4
L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4 -3 -1 -4
K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0  1 -1 -4
M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3 -1 -1 -4
F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3 -3 -1 -4
P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -1 -1 -4
S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0  0 -1 -4
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1 -1 -4
W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -3 -1 -4
Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -2 -1 -4
V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3 -2 -1 -4
B -2 -1  3  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4  1 -1 -4
Z -1  0  0  1 -3  3  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
X -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -4
,* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1
```

Figure 56.2: The BLOSUM62 amino acid substitution matrix for scoring aligned pairs of amino acids.

We import `io` and `bufio`.

514a ⟨*Imports, Ch. 56* 506c⟩+≡ (505a) ◁511b 514d▷
```
"io"
"bufio"
```

As shown in Figure 56.2, the first line that isn't fenced off, is the table header, the others make up its body.

514b ⟨*Deal with score table, Ch. 56* 514b⟩≡ (512d)
```
if b[0] != '#' {
        if first {
                ⟨Deal with header, Ch. 56 514c⟩
        } else {
                ⟨Deal with body, Ch. 56 514e⟩
        }
}
```

The header line is split into individual headers, and we remember not to do that again.

514c ⟨*Deal with header, Ch. 56* 514c⟩≡ (514b)
```
res = bytes.Fields(b)
first = false
```

We import `bytes`.

514d ⟨*Imports, Ch. 56* 506c⟩+≡ (505a) ◁514a 514f▷
```
"bytes"
```

The body of the table is split into entries, of which the first is a residue and the others are scores.

514e ⟨*Deal with body, Ch. 56* 514e⟩≡ (514b)
```
entries := bytes.Fields(b)
for i := 1; i < len(entries); i++ {
        c1 := entries[0][0]
        c2 := res[i-1][0]
        score, err := strconv.ParseFloat(string(entries[i]), 64)
        if err != nil {
                log.Fatalf("couldn't parse %q\n", entries[i])
        }
        s.setScore(c1, c2, score)
}
```

We import `strconv` and `log`.

514f ⟨*Imports, Ch. 56* 506c⟩+≡ (505a) ◁514d 515b▷
```
"strconv"
"log"
```

Method Score

!The method `Score` takes two characters as arguments and !returns their score. If one of the characters is not a printing ASCII !character, it returns the smallest float and prints a warning.

We first check we have a pair of valid characters, then return the corresponding score as a 64-bit float, because that is the standard currency of numerical work.

515a ⟨*Methods, Ch. 56* 506d⟩+≡ (505a) ◁507a 515c▷
```
func (s *ScoreMatrix) Score(c1, c2 byte) float64 {
        c1 -= 32
        c2 -= 32
        if c1 < 0 || c1 > 94 || c2 < 0 || c2 > 94 {
                fmt.Fprintf(os.Stderr, "couldn't score " +
                        "(%q, %q)\n", c1, c2)
                return -math.MaxFloat64
        }
        return float64(s.m[c1][c2])
}
```

We import `fmt` and `math`.

515b ⟨*Imports, Ch. 56* 506c⟩+≡ (505a) ◁514f 519e▷
```
"fmt"
"math"
```

Method `setScore`

The method `setScore` takes as argument two characters and their score, and stores the triple.

515c ⟨*Methods, Ch. 56* 506d⟩+≡ (505a) ◁515a 517a▷
```
func (s *ScoreMatrix) setScore(c1, c2 byte, sc float64) {
        c1 -= 32
        c2 -= 32
        if c1 < 0 || c1 > 94 || c2 < 0 || c2 > 94 {
                fmt.Fprintf(os.Stderr, "couldn't score " +
                        "(%q, %q)\n", c1, c2)
        }
        s.m[c1][c2] = float32(sc)
}
```

56.5 Structure `TransitionTab`

!A `transitionTab` indicates whether or not a pair of !nucleotides represents a transition. Nucleotides come in two types, !the purines, `A` and `G`, and the pyrimidines, !`C` and `T`. Mutations within a chemical class are !called transitions, as opposed to transversions, mutations between !the classes.

The structure holds a two-dimensional table of booleans. Nucleotides come as bytes and the are used as indexes into the table. The nucleotide alphabet starts at an *offset* from the ASCII alphabet and is n characters long.

515d ⟨*Types, Ch. 56* 505c⟩+≡ (505a) ◁511d
```
type TransitionTab struct {
        offset, n byte
        ts [][]bool
}
```

Function NewTransitionTab

!NewTransitionTab constructs and initializes a new !TransitionTab.

The table caters for nucleotides in caps, of which the smallest is A, decimal 65, and the largest T, 84. Hence the offset is 65 and $n = 84 - 65 + 1 = 20$ characters are accounted for.

516a ⟨*Functions, Ch. 56* 506a⟩+≡ (505a) ◁512d 517b▷

```
func NewTransitionTab() TransitionTab {
        var tab TransitionTab
        tab.offset = 65
        tab.n = 20
        ⟨Make transition table, Ch. 56 516b⟩
        ⟨Enter transitions, Ch. 56 516c⟩
        return tab
}
```

The transition table is a slice of boolean slices.

516b ⟨*Make transition table, Ch. 56* 516b⟩≡ (516a)

```
tab.ts = make([][]bool, tab.n)
for i := 0; i < int(tab.n); i++ {
        tab.ts[i] = make([]bool, tab.n)
}
```

We fill in the transition table to get:

	A	C	G	T
A	false	false	true	false
C	false	false	false	true
G	true	false	false	false
T	false	true	false	false

516c ⟨*Enter transitions, Ch. 56* 516c⟩≡ (516a)

```
a := byte('A') - tab.offset
c := byte('C') - tab.offset
g := byte('G') - tab.offset
t := byte('T') - tab.offset
tab.ts[a][g] = true
tab.ts[c][t] = true
tab.ts[g][a] = true
tab.ts[t][c] = true
```

Method `IsTransition`

!`IsTransition` takes a pair of nucleotides as arguments and !returns true if they are both caps and represent a transition, false !otherwise.

517a ⟨*Methods, Ch. 56* 506d⟩+≡ (505a) ◁515c

```
func (t TransitionTab) IsTransition(a, b byte) bool {
        a -= t.offset
        b -= t.offset
        if a < 0 || b < 0 || a >= t.n || b >= t.n {
                return false
        }
        return t.ts[a][b]
}
```

56.6 Function `TTest`

!`TTest` tests the equality of two sample means. It takes as !input two samples and returns their means, the value of t, and its !significance, p. It runs with equal variance (original Student's !t-test), or with unequal variances (Welch's test)

For Student's t-test, t is defined as

$$t = \frac{m_1 - m_2}{s_p\sqrt{1/n_1 + 1/n_2}}, \tag{56.1}$$

where m_1 and m_2 are the sample means, n_1 and n_2 their 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 degrees of freedom for significance testing are

$$d = n_1 + n_2 - 2. \tag{56.2}$$

For Welch's t-test, we have

$$t = \frac{m_1 - m_2}{\sqrt{v_1/n_1 + v_2/n_2}}. \tag{56.3}$$

In this case the degrees of freedom are

$$d = \frac{(v_1/n_1 + v_2/n_2)^2}{v_1^2/n_1^2/(n_1 - 1) + v_2^2/n_2^2/(n_2 - 1)}. \tag{56.4}$$

517b ⟨*Functions, Ch. 56* 506a⟩+≡ (505a) ◁516a 519d▷

```
func TTest(d1, d2 []float64, equalVar bool) (m1, m2, t, p float64) {
        ⟨Compute means and variances, Ch. 56 518a⟩
        ⟨Compute t and degrees of freedom, Ch. 56 518b⟩
        ⟨Compute P-value, Ch. 56 519a⟩
        return m1, m2, t, p
}
```

Means and variances are calculated in a single function.

518a ⟨*Compute means and variances, Ch. 56* 518a⟩≡ (517b)
```
m1, v1 := MeanVar(d1)
m2, v2 := MeanVar(d2)
```

The value of the test statistic, t, depends on whether or not we are dealing with equal variances.

518b ⟨*Compute t and degrees of freedom, Ch. 56* 518b⟩≡ (517b)
```
var d float64
n1 := float64(len(d1))
n2 := float64(len(d2))
if equalVar {
        ⟨Equal variances, Ch. 56 518c⟩
} else {
        ⟨Unequal variances, Ch. 56 518d⟩
}
```

For the equal-variance case we transcribe equations (56.1) and (56.2).

518c ⟨*Equal variances, Ch. 56* 518c⟩≡ (518b)
```
x := (n1 - 1.0) * v1 + (n2 - 1.0) * v2
if x == 0 {
        log.Fatal("util.TTest: Error, data constant.\n")
}
d  = n1 + n2 - 2.0
if d == 0 {
        log.Fatal("util.TTest: Error, samples too small.\n")
}
sp := math.Sqrt(x/d)
x = sp * math.Sqrt(1.0/n1 + 1.0/n2)
t = (m1 - m2) / x
```

For the unequal-variance case we transcribe equations (56.3) and (56.4).

518d ⟨*Unequal variances, Ch. 56* 518d⟩≡ (518b)
```
t = (m1 - m2) / math.Sqrt(v1/n1 + v2/n2)
x := (v1/n1 + v2/n2) * (v1/n1 + v2/n2)
y := v1*v1/n1/n1/(n1-1.0) + v2*v2/n2/n2/(n2-1.0)
if y == 0 {
        log.Fatal("util.TTest: Error, data constant.\n")
}
d = x / y
```

For the significance computation we call the cumulative density function of the t-distribution provided by the Gnu Scientific Library. This is written in C and we use cgo to communicate between C and Go.

519a ⟨*Compute P-value, Ch. 56* 519a⟩≡ (517b)
```
ct := C.double(t)
cd := C.double(d)
if t > 0 {
          p = float64(C.gsl_cdf_tdist_Q(ct, cd)) * 2.0
} else {
          p = float64(C.gsl_cdf_tdist_P(ct, cd)) * 2.0
}
```

We add the GSL to cgo and make sure the build also works on macOS with homebrew.

519b ⟨*Cgo, Ch. 56* 519b⟩≡ (505b)
```
#cgo CFLAGS: -I/opt/homebrew/include
#cgo LDFLAGS: -lgsl -lgslcblas -L/opt/homebrew/lib
```

We also include the header for the cumulative density functions.

519c ⟨*Includes, Ch. 56* 519c⟩≡ (505b)
```
#include <gsl/gsl_cdf.h>
```

56.7 PrepLog

In `CheckGnuplot` and elsewhere, we handle errors via the `log` package. By default, this prefixes the error message with time and date. Instead, we'd like error messages prefixed with the name of the program and with date and time suppressed.

!`PrepLog` takes as argument the program name and sets !this as the prefix for error messages from the `log` package.

519d ⟨*Functions, Ch. 56* 506a⟩+≡ (505a) ◁517b 524a▷
```
func PrepLog(name string) {
          m := fmt.Sprintf("%s: ", name)
          log.SetPrefix(m)
          log.SetFlags(0)
}
```

We import `fmt`.

519e ⟨*Imports, Ch. 56* 506c⟩+≡ (505a) ◁515b
```
"fmt"
```

56.8 Testing

The testing outline contains hooks for imports, types, and the testing logic.

520a ⟨*util_test.go* 520a⟩≡
```
package util

import (
        "testing"
        ⟨Testing imports, Ch. 56 520c⟩
)
⟨Testing types, Ch. 56 522b⟩
func TestUtil(t *testing.T) {
        ⟨Testing, Ch. 56 520b⟩
}
```

Alignment

We begin our test of `Alignment` by constructing one from the two peptide sequences shown in Figure 56.1.

520b ⟨*Testing, Ch. 56* 520b⟩≡ (520a) 520d▷
```
s1 := fasta.NewSequence("s1", []byte("MKFLAL-F"))
s2 := fasta.NewSequence("s2", []byte("MKYLILLF"))
sf, err := os.Open("BLOSUM62")
if err != nil {
        t.Error("couldn't open BLOSUM62\n")
}
sm := ReadScoreMatrix(sf)
sf.Close()
al := NewAlignment(s1, s2, sm, 7, 8, 0, 0, 19)
```

We import `fasta` and `os`.

520c ⟨*Testing imports, Ch. 56* 520c⟩≡ (520a) 521a▷
```
"github.com/evolbioinf/fasta"
"os"
```

Now we compare what we get with what we want, which is stored in the file `res1.txt`. The data in the file is terminated by an extra newline, so we add one to what we compute.

520d ⟨*Testing, Ch. 56* 520b⟩+≡ (520a) ◁520b 521b▷
```
get := []byte(al.String())
get = append(get, '\n')
want, err := ioutil.ReadFile("res1.txt")
if err != nil {
        t.Error("couldn't open res1.txt\n")
}
if !bytes.Equal(want, get) {
        t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We import `ioutil` and `bytes`.

521a ⟨*Testing imports, Ch. 56* 520c⟩+≡ (520a) ◁520c 523c▷

```
"io/ioutil"
"bytes"
```

Now we set the line length to 4.

521b ⟨*Testing, Ch. 56* 520b⟩+≡ (520a) ◁520d 521c▷

```
al.SetLineLength(4)
get = []byte(al.String())
get = append(get, '\n')
want, err = ioutil.ReadFile("res2.txt")
if err != nil {
        t.Error("couldn't open res2.txt\n")
}
if !bytes.Equal(want, get) {
        t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

We test the `ScoreMatrix` by sampling four residue pairs.

521c ⟨*Testing, Ch. 56* 520b⟩+≡ (520a) ◁521b 521d▷

```
w := []float64{4, -1, 4, -4}
g := make([]float64, 4)
g[0] = sm.Score('A', 'A')
g[1] = sm.Score('A', 'R')
g[2] = sm.Score('B', 'B')
g[3] = sm.Score('*', 'X')
for i := 0; i < 4; i++ {
        if w[i] != g[i] {
                t.Errorf("want:\n%g\nget:\n%g\n", w[i], g[i])
        }
}
```

We continue the test by setting the score for `F` and `Y` to zero and observing that the similarity character, `:` changes to blank.

521d ⟨*Testing, Ch. 56* 520b⟩+≡ (520a) ◁521c 522a▷

```
al.scoreMatrix.setScore('Y', 'F', 0)
al.scoreMatrix.setScore('F', 'Y', 0)
get = []byte(al.String())
get = append(get, '\n')
want, err = ioutil.ReadFile("res3.txt")
if err != nil {
        t.Error("couldn't open res3.txt")
}
if !bytes.Equal(get, want) {
        t.Errorf("3 want:\n%s\nget:\n%s\n", want, get)
}
```

TransitionTab

To test TransitionTab, we set up some tests and run them.

522a ⟨*Testing, Ch. 56* 520b⟩+≡ (520a) ◁521d 522e▷
```
tab := NewTransitionTab()
⟨Set up transition tests, Ch. 56 522c⟩
⟨Run transition tests, Ch. 56 522d⟩
```

Each test consists of three values, two characters and the boolean wanted. We set up the structure triplet to hold these values.

522b ⟨*Testing types, Ch. 56* 522b⟩≡ (520a)
```
type triplet struct {
        a, b byte
        w bool
}
```

We set up the table of tests.

522c ⟨*Set up transition tests, Ch. 56* 522c⟩≡ (522a)
```
tr := make([]triplet, 8)
tr = append(tr, triplet{a: 'A', b: 'G', w: true})
tr = append(tr, triplet{a: 'G', b: 'A', w: true})
tr = append(tr, triplet{a: 'C', b: 'T', w: true})
tr = append(tr, triplet{a: 'T', b: 'C', w: true})
tr = append(tr, triplet{a: 'A', b: 'C', w: false})
tr = append(tr, triplet{a: 'T', b: 'T', w: false})
tr = append(tr, triplet{a: 'T', b: 't', w: false})
tr = append(tr, triplet{a: '!', b: 'A', w: false})
```

If a pair of characters is misclassified, the test fails.

522d ⟨*Run transition tests, Ch. 56* 522d⟩≡ (522a)
```
for _, test := range tr {
        g := tab.IsTransition(test.a, test.b)
        if g != test.w {
                t.Errorf("misclassified (%c, %c)\n", test.a, test.b)
        }
}
```

TTest

We create a file for storing the results of our t-test. Once carried out, we compare the results of our test with the results we want, before deleting the file again.

522e ⟨*Testing, Ch. 56* 520b⟩+≡ (520a) ◁522a
```
⟨Create file for t-test, Ch. 56 523a⟩
⟨Carry out t-test and save results, Ch. 56 523b⟩
⟨Compare t-test results, Ch. 56 523d⟩
⟨Remove file for t-test, Ch. 56 523e⟩
```

We open a file to store our results in and throw an error if we can't.

523a ⟨*Create file for t-test, Ch. 56* 523a⟩≡ (522e)
```
fn := "tmp.txt"
outf, err := os.Create(fn)
if err != nil {
        t.Errorf("couldn't open %q\n", fn)
}
```

We construct two data sets and carry out the test with equal variances and without.

523b ⟨*Carry out t-test and save results, Ch. 56* 523b⟩≡ (522e)
```
d1 := []float64{11.961,12.401,11.661,11.96,10.454,11.584,11.175}
d2 := []float64{8.479,8.523,8.793,8.726,9.677,8.728,8.383,11.086}
m1, m2, st, p := TTest(d1, d2, true)
fmt.Fprintf(outf, "%.8g %.8g %.8g %.8g\n", m1, m2, st, p)
m1, m2, st, p = TTest(d1, d2, false)
fmt.Fprintf(outf, "%.8g %.8g %.8g %.8g\n", m1, m2, st, p)
outf.Close()
```

We import `fmt`.

523c ⟨*Testing imports, Ch. 56* 520c⟩+≡ (520a) ◁521a
```
"fmt"
```

The file `res4.txt` contains the results we want, which we compare to what we get.

523d ⟨*Compare t-test results, Ch. 56* 523d⟩≡ (522e)
```
want, err = ioutil.ReadFile("res4.txt")
if err != nil {
          t.Errorf("couldn't open res4.txt\n")
}
get, err  = ioutil.ReadFile(fn)
if err != nil {
          t.Errorf("couldn't open %q\n", fn)
}
if !bytes.Equal(want, get) {
          t.Errorf("want:\n%s\nget:\n%s\n", want, get)
}
```

523e ⟨*Remove file for t-test, Ch. 56* 523e⟩≡ (522e)
```
err = os.Remove(fn)
if err != nil {
          t.Errorf("couldn't remove %q\n", fn)
}
```

56.9 Function CheckGnuplot

!CheckGnuplot checks the error returned by a gnuplot run.

524a ⟨*Functions, Ch. 56* 506a⟩+≡ (505a) ◁519d 524b▷

```
func CheckGnuplot(err error) {
        if err != nil {
                m := "Error when plotting with gnuplot; "
                m += "you might like to try a different terminal. "
                m += "To get the list of available terminals, "
                m += "start gnuplot and enter \"set term\"."
                log.Fatal(m)
        }
}
```

56.10 Function IsInteractive

!IsInteractive checks whether a gnuplot terminal is interactive !or not.

524b ⟨*Functions, Ch. 56* 506a⟩+≡ (505a) ◁524a

```
func IsInteractive(t string) bool {
        ii := false
        if t == "wxt" || t == "x11" || t == "qt" ||
                t == "aqua" || t == "windows" {
                ii = true
        }
        return ii
}
```

Chapter 57

Program `var`: Variance

Introduction

Given a set of numbers, `var` computes their mean, variance, and standard deviation. Input is read from a single column, one number per line.

Implementation

The outline contains hooks for imports, variables, functions, and the logic of the main function.

526a ⟨*var.go* 526a⟩≡
```
package main

import (
        ⟨Imports, Ch. 57 526c⟩
)
⟨Variables, Ch. 57 527b⟩
⟨Functions, Ch. 57 527d⟩
func main() {
        ⟨Main function, Ch. 57 526b⟩
}
```

In the main function we prepare the `log` package, set the usage, parse the user options, and iterate over the input files.

526b ⟨*Main function, Ch. 57* 526b⟩≡ (526a)
```
util.PrepLog("var")
⟨Set usage, Ch. 57 526d⟩
⟨Parse options, Ch. 57 526f⟩
⟨Iterate over files, Ch. 57 527c⟩
```

We import `util`.

526c ⟨*Imports, Ch. 57* 526c⟩≡ (526a) 526e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage begins with the actual usage statement, followed by an explanation of what the program does, and an example command.

526d ⟨*Set usage, Ch. 57* 526d⟩≡ (526b)
```
u := "var [-h] [options] [files]"
p := "Compute the mean and variance of a set of numbers."
e := "var *.txt"
clio.Usage(u, p, e)
```

We import `clio`.

526e ⟨*Imports, Ch. 57* 526c⟩+≡ (526a) ◁526c 527a ▷
```
"github.com/evolbioinf/clio"
```

The flags are parsed and `PrintInfo` is called, if requested.

526f ⟨*Parse options, Ch. 57* 526f⟩≡ (526b)
```
flag.Parse()
if *optV {
        util.PrintInfo("var")
}
```

We import the package `flag`.

527a ⟨*Imports, Ch. 57* 526c⟩+≡ (526a) ◁526e 527e▷
```
"flag"
```

The variable `*optV` corresponds to option `-v`.

527b ⟨*Variables, Ch. 57* 527b⟩≡ (526a)
```
var optV = flag.Bool("v", false, "version")
```

By calling `flag.Parse()`, we consume the options. All remaining arguments on the command line are file names. We pass them to the function `clio.ParseFiles`. In addition to the file names, this takes as argument the function for scanning each file, `scan`. Results are reported per file, hence we pass a copy of the file names to `scan`.

527c ⟨*Iterate over files, Ch. 57* 527c⟩≡ (526b)
```
files := flag.Args()
var fn = make([]string, len(files))
copy(fn, files)
clio.ParseFiles(files, scan, fn)
```

In `scan` the data is first collected, then analyzed, and finally the results are printed.

527d ⟨*Functions, Ch. 57* 527d⟩≡ (526a)
```
func scan(r io.Reader, args ...interface{}) {
        ⟨Collect data, Ch. 57 527f⟩
        ⟨Analyze data, Ch. 57 527h⟩
        ⟨Print results, Ch. 57 528b⟩
}
```

We import `io`.

527e ⟨*Imports, Ch. 57* 526c⟩+≡ (526a) ◁527a 527g▷
```
"io"
```

527f ⟨*Collect data, Ch. 57* 527f⟩≡ (527d)
```
sc := bufio.NewScanner(r)
var data []float64
for sc.Scan() {
        str := string(sc.Bytes())
        x, err := strconv.ParseFloat(str, 64)
        if err != nil {
                log.Fatalf("couldn't parse %q\n", str)
        }
        data = append(data, x)
}
```

We import `bufio`, `strconv`, and `log`.

527g ⟨*Imports, Ch. 57* 526c⟩+≡ (526a) ◁527e 528a▷
```
"bufio"
"strconv"
"log"
```

The data is analyzed using the utility function `MeanVar`.

527h ⟨*Analyze data, Ch. 57* 527h⟩≡ (527d)
```
ave, variance := util.MeanVar(data)
sdev := math.Sqrt(variance)
```

We import `math`.

528a ⟨*Imports, Ch. 57* 526c⟩+≡ (526a) ◁527g 528d ▷
```
"math"
```

We print the results using a `tabwriter` to align the columns. We also echo the file name. By default, this is `stdin`, but it might be set to the name of an input file.

528b ⟨*Print results, Ch. 57* 528b⟩≡ (527d)
```
fn := args[0].([]string)
file := "stdin"
⟨Set file name, Ch. 57 528c⟩
w := new(tabwriter.Writer)
w.Init(os.Stdout, 4, 0, 1, ' ', 0)
fmt.Fprintf(w, "# File\tAvg\tVar\tSD\tn\n")
fmt.Fprintf(w, "%s\t%.6g\t%.6g\t%.6g\t%d\n",
          file, ave, variance, sdev, len(data))
w.Flush()
```

If input files were used, we assign the next one in the list to `file` and lop it off the start of the list.

528c ⟨*Set file name, Ch. 57* 528c⟩≡ (528b)
```
if len(fn) > 0 {
          file = fn[0]
          args[0] = fn[1:]
}
```

We import `tabwriter`, `os`, and `fmt`.

528d ⟨*Imports, Ch. 57* 526c⟩+≡ (526a) ◁528a
```
"text/tabwriter"
"os"
"fmt"
```

We're done writing `var`, let's test it.

Testing

We use the standard testing framework.

528e ⟨*var_test.go* 528e⟩≡
```
package main

import (
          "testing"
          ⟨Testing imports, Ch. 57 529b⟩
)

func TestVar(t *testing.T) {
          ⟨Testing, Ch. 57 529a⟩
}
```

We begin by applying `var` to `data1.txt`. The output we get is compared to the output we want in `res1.txt`.

529a ⟨*Testing, Ch. 57* 529a⟩≡ (528e) 529c ▷

```
cmd := exec.Command("./var", "data1.txt")
g, err := cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err := ioutil.ReadFile("res1.txt")
if !bytes.Equal(g, w) {
        t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

We import `exec`, `ioutil`, and `bytes`.

529b ⟨*Testing imports, Ch. 57* 529b⟩≡ (528e)

```
"os/exec"
"io/ioutil"
"bytes"
```

In the second and last test we iterate across the two input files `data[12].txt`.

529c ⟨*Testing, Ch. 57* 529a⟩+≡ (528e) ◁529a

```
cmd = exec.Command("./var", "data1.txt", "data2.txt")
g, err = cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res2.txt")
if !bytes.Equal(g, w) {
        t.Errorf("want\n%ss\nget:\n%s\n", w, g)
}
```

Chapter 58

Program `watterson`: Estimating the Number of Mutations

Introduction

The expected number of segregating sites, S, observed in a sample of aligned DNA sequences is a simple function of the sample size, n,

$$S = \theta \sum_{i=1}^{n-1} \frac{1}{i}, \tag{58.1}$$

where $\theta = 4N_{\mathrm{e}}\mu$, N_{e} the "effective" population size, and μ the mutation rate. Equation 58.1 was published in 1975 by G. A. Watterson [30], hence the name of my program.

There is also an approximate version of Watterson's equation [16],

$$S \approx \theta \left(\gamma + \frac{1-\gamma}{n-1} + \log(n-1) \right), \tag{58.2}$$

where $\gamma \approx 0.58$ is the Euler-Mascheroni constant. We implement both the exact and the approximate equation.

Implementation

The program outline contains hooks for imports, constants, variables, and the logic of the main function.

531a ⟨*watterson.go* 531a⟩≡
```
package main

import (
        ⟨Imports, Ch. 58 531c⟩
)
const (
        ⟨Constants, Ch. 58 533c⟩
)
⟨Variables, Ch. 58 532c⟩
func main() {
        ⟨Main function, Ch. 58 531b⟩
}
```

In the main function, we prepare the `log` package, set the usage, parse the options, and compute S.

531b ⟨*Main function, Ch. 58* 531b⟩≡ (531a)
```
util.PrepLog("watterson")
⟨Set usage, Ch. 58 532a⟩
⟨Parse options, Ch. 58 532e⟩
⟨Compute S, Ch. 58 533a⟩
```

We import `util`.

531c ⟨*Imports, Ch. 58* 531c⟩≡ (531a) 532b ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of three parts: The usage message itself, an explanation of the program's purpose, and an example command.

532a ⟨*Set usage, Ch. 58* 532a⟩≡ (531b)
```
u := "watterson [-h] [options}"
p := "Compute Watterson's estimator of the number " +
          "of segregating sites."
e := "watterson -n 10 -t 20"
clio.Usage(u, p, e)
```

We import `clio`.

532b ⟨*Imports, Ch. 58* 531c⟩+≡ (531a) ◁531c 532d▷
```
"github.com/evolbioinf/clio"
```

Before parsing the options, we declare them: There is the sample size (`-n`), θ (`-t`), and the possibility to use the approximation (`-a`). Finally, the user can request the program's version (`-v`).

532c ⟨*Variables, Ch. 58* 532c⟩≡ (531a)
```
var optN = flag.Int("n", 0, "sample size")
var optT = flag.Float64("t", 0, "theta = 4Nu")
var optA = flag.Bool("a", false, "use approximation")
var optV = flag.Bool("v", false, "version")
```

We import `flag`,

532d ⟨*Imports, Ch. 58* 531c⟩+≡ (531a) ◁532b 532f▷
```
"flag"
```

and parse the options. If $n < 2$ or $\theta = 0$, we prompt the user for sensible values.

532e ⟨*Parse options, Ch. 58* 532e⟩≡ (531b)
```
flag.Parse()
if *optV {
          util.PrintInfo("watterson")
}
if *optN < 2 || *optT == 0 {
          fmt.Fprintf(os.Stderr, "Please enter a sample size > 1, " +
                  "and a theta > 0\n")
          os.Exit(0)
}
```

We import `flag`, `fmt`, and `os`.

532f ⟨*Imports, Ch. 58* 531c⟩+≡ (531a) ◁532d 533b▷
```
"flag"
"fmt"
"os"
```

S is computed either with the exact formula, or with the approximation.

533a ⟨*Compute S, Ch. 58* 533a⟩≡ (531b)
```
var S float64
t := *optT
n := *optN
if *optA {
        ⟨Approximate S, Ch. 58 533d⟩
} else {
        ⟨Exact S, Ch. 58 533f⟩
}
fmt.Printf("S = %.8g\n", S)
```

We import `fmt`.

533b ⟨*Imports, Ch. 58* 531c⟩+≡ (531a) ◁532f 533e▷
```
"fmt"
```

The approximate formula is based on the Euler-Mascheroni constant, γ, which we take from the "Online Encyclopedia of Integer Sequences".

533c ⟨*Constants, Ch. 58* 533c⟩≡ (531a)
```
EulerMascheroni = 0.57721566490153286060651209008240243104215933594
```

The actual computation is a transcription of equation (58.2).

533d ⟨*Approximate S, Ch. 58* 533d⟩≡ (533a)
```
g := EulerMascheroni
S = t * (g + (1-g)/float64(n-1) + math.Log(float64(n-1)))
```

We import `math`.

533e ⟨*Imports, Ch. 58* 531c⟩+≡ (531a) ◁533b
```
"math"
```

Similarly, for the exact computation we transcribe equation (58.1).

533f ⟨*Exact S, Ch. 58* 533f⟩≡ (533a)
```
var h float64
for i := 1; i < n; i++ {
  h += 1/float64(i)
}
S = t * h
```

The implementation is finished, the rest's the test.

Testing

The testing framework contains hooks for imports and the actual testing logic.

534a ⟨*watterson_test.go* 534a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 58 534c⟩
)

func TestWatterson(t *testing.T) {
        ⟨Testing, Ch. 58 534b⟩
}
```

We begins with $n = 10$ and $\theta = 20$, and compare what we get with what we want, which is stored in `res1.txt`

534b ⟨*Testing, Ch. 58* 534b⟩≡ (534a) 535▷
```
cmd := exec.Command("./watterson", "-n", "10", "-t", "20")
g, err := cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err := ioutil.ReadFile("res1.txt")
if err != nil {
        t.Errorf("couldnt' open res1.txt")
}
if !bytes.Equal(g, w) {
        t.Errorf("get:\n%s\nwant:\n%s\n", g, w)
}
```

We import `exec`, `ioutil`, and `bytes`.

534c ⟨*Testing imports, Ch. 58* 534c⟩≡ (534a)
```
"os/exec"
"io/ioutil"
"bytes"
```

We also use the approximate formula and compare what we get with what we want in `res2.txt`.

535 ⟨*Testing, Ch. 58* 534b⟩+≡ (534a) ◁534b

```
cmd = exec.Command("./watterson", "-n", "10", "-t", "20", "-a")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res2.txt")
if err != nil {
          t.Errorf("couldnt' open res2.txt")
}
if !bytes.Equal(g, w) {
          t.Errorf("get:\n%s\nwant:\n%s\n", g, w)
}
```

Chapter 59

Program `wrapSeq`: Wrap Sequence

Introduction

The lengths of data lines in FASTA files varies a lot. The program `wrapSeq` allows wrapping them to a user-defined length.

Implementation

The program outline contains hooks for imports, variables, functions, and the logic of the main function.

537a ⟨*wrapSeq.go* 537a⟩≡
```
package main

import (
        ⟨Imports, Ch. 59 537c⟩
)
⟨Variables, Ch. 59 538a⟩
⟨Functions, Ch. 59 538f⟩
func main() {
        ⟨Main function, Ch. 59 537b⟩
}
```

In the main function we prepare the `log` package, set the usage, parse the user options, and iterate over the input files.

537b ⟨*Main function, Ch. 59* 537b⟩≡ (537a)
```
util.PrepLog("wrapSeq")
⟨Set usage, Ch. 59 537d⟩
⟨Parse options, Ch. 59 537f⟩
⟨Parse input, Ch. 59 538c⟩
```

We import `util`.

537c ⟨*Imports, Ch. 59* 537c⟩≡ (537a) 537e ▷
```
"github.com/evolbioinf/biobox/util"
```

The usage consists of the actual usage statement, an explanation of the purpose of the program, and an example of its application.

537d ⟨*Set usage, Ch. 59* 537d⟩≡ (537b)
```
u := "wrapSeq [-h] [options] [files]"
p := "Wrap lines of sequence data."
e := "wrapSeq -l 50 *.fasta"
clio.Usage(u, p, e)
```

We import `clio`.

537e ⟨*Imports, Ch. 59* 537c⟩+≡ (537a) ◁537c 538b ▷
```
"github.com/evolbioinf/clio"
```

Next we parse the options an immediately check for `-v`.

537f ⟨*Parse options, Ch. 59* 537f⟩≡ (537b)
```
flag.Parse()
if *optV {
        util.PrintInfo("wrapSeq")
}
```

We declare -v,

538a ⟨*Variables, Ch. 59* 538a⟩≡ (537a) 538d▷
```
var optV = flag.Bool("v", false, "version")
```

and import `flag`.

538b ⟨*Imports, Ch. 59* 537c⟩+≡ (537a) ◁537e 538e▷
```
"flag"
```

The command line arguments remaining after `flag.Parse` was called, are file names. We pass them to be scanned with the line length as argument. This is set via `-l`.

538c ⟨*Parse input, Ch. 59* 538c⟩≡ (537b)
```
files := flag.Args()
clio.ParseFiles(files, scan, *optL)
```

We declare the line length, `-l`; values less than 1 signal unbroken lines of data.

538d ⟨*Variables, Ch. 59* 538a⟩+≡ (537a) ◁538a
```
var optL = flag.Int("l", fasta.DefaultLineLength, "line length, " +
        "< 1 for unbroken lines")
```

We import `fasta`.

538e ⟨*Imports, Ch. 59* 537c⟩+≡ (537a) ◁538b 538g▷
```
"github.com/evolbioinf/fasta"
```

In the `scan` function we first retrieves the line length, then parse the input one `Sequence` at a time, and finally print it with the desired line length.

538f ⟨*Functions, Ch. 59* 538f⟩≡ (537a)
```
func scan(r io.Reader, args ...interface{}) {
        l := args[0].(int)
        sc := fasta.NewScanner(r)
        for sc.ScanSequence() {
                se := sc.Sequence()
                se.SetLineLength(l)
                fmt.Println(se)
        }
}
```

We import `io` and `fmt`.

538g ⟨*Imports, Ch. 59* 537c⟩+≡ (537a) ◁538e
```
"io"
"fmt"
```

Testing

We set up the testing framework.

539a ⟨*wrapSeq_test.go* 539a⟩≡
```
package main

import (
        "testing"
        ⟨Testing imports, Ch. 59 539c⟩
)

func TestWrapSeq(t *testing.T) {
        ⟨Testing, Ch. 59 539b⟩
}
```

The testing data is in `test.fasta`, which contains two sequences length 100 in lines of 70 and 30 nucleotides each.

First, run `wrapSeq` with default options. The result should be identical to the input.

539b ⟨*Testing, Ch. 59* 539b⟩≡ (539a) 539d ▷
```
cmd := exec.Command("./wrapSeq", "test.fasta")
g, err := cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err := ioutil.ReadFile("test.fasta")
if !bytes.Equal(g, w) {
        t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

We import `exec`, `ioutil`, and `bytes`.

539c ⟨*Testing imports, Ch. 59* 539c⟩≡ (539a)
```
"os/exec"
"io/ioutil"
"bytes"
```

Now we wrap into lines of 100 nucleotides.

539d ⟨*Testing, Ch. 59* 539b⟩+≡ (539a) ◁539b 540a ▷
```
cmd = exec.Command("./wrapSeq", "-l", "100", "test.fasta")
g, err = cmd.Output()
if err != nil {
        t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res1.fasta")
if !bytes.Equal(g, w) {
        t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

We effectively repeat this by asking for unbroken lines.

540a ⟨*Testing, Ch. 59* 539b⟩+≡ (539a) ◁539d 540b ▷

```
cmd = exec.Command("./wrapSeq", "-l", "0", "test.fasta")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res1.fasta")
if !bytes.Equal(g, w) {
          t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

Finally, we wrap into 50 bp lines.

540b ⟨*Testing, Ch. 59* 539b⟩+≡ (539a) ◁540a

```
cmd = exec.Command("./wrapSeq", "-l", "50", "test.fasta")
g, err = cmd.Output()
if err != nil {
          t.Errorf("couldn't run %q\n", cmd)
}
w, err = ioutil.ReadFile("res2.fasta")
if !bytes.Equal(g, w) {
          t.Errorf("want:\n%s\nget:\n%s\n", w, g)
}
```

Bibliography

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.

[2] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[3] C. Bachmaier, U. Brandes, and B. Schlieper. Drawing phylogenetic trees. In X. Deng and Du DZ, editors, *Algorithms and Computation, ISAAC 2005*, volume 3827 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2005. Springer.

[4] R. A. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. In *Proc. 3rd Symp. on Combinatorial Pattern Matching*, volume 644 of *Springer Lecture Notes in Computer Science*, pages 185–192, New York, 1992. Springer.

[5] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2001.

[7] A. A. A. Donovan and B. W. Kernighan. *The Go Programming Language*. Addison-Wesley, New York, 2016.

[8] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, New York, 1993.

[9] J. Felsenstein. Confidence limits on phylogenies: an approach using the bootstrap. *Evolution*, 39:783–791, 1985.

[10] J. Felsenstein. PHYLIP (phylogeny interference package) version 3.6, 2005.

[11] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*. Network Theory Ltd, 1.6, for gsl version 1.6, 17 march 2005 edition, 2005.

[12] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, 1997.

[13] D. Haig and L. D. Hurst. A quantitative measure of error minimization in the genetic code. *Journal of Molecular Evolution*, 33:412–417, 1991.

[14] B. Haubold and A. Börsch-Haubold. *Bioinformatics for Evolutionary Biologists. A Problems Approach*. Springer, 2017.

[15] B. Haubold, N. Pierstorff, F. Möller, and T. Wiehe. Genome comparison without alignment using shortest unique substrings. *BMC Bioinformatics*, 6:123, 2005.

[16] B. Haubold and T. Wiehe. Calculating the SNP-effective sample size from an alignment. *Bioinformatics*, 18:36–38, 2002.

[17] R. R. Hudson. Gene genealogies and the coalescent process. *Oxford Surveys in Evolutionary Biology*, 7:1–44, 1990.

[18] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E*, 40:1098–1101, 1952.

[19] P. K. Janert. *Gnuplot in Action, Understanding Data with Graphs*. Manning, Greenwich, 2010.

[20] T. H. Jukes and C. R. Cantor. Evolution of protein molecules. In H. N. Munro, editor, *Mammalian Protein Metabolism*, volume 3, pages 21–132. Academic Press, New York, 1969.

[21] M. Kimura. A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences. *Journal of Molecular Evolution*, 16:111–120, 1980.

[22] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addision Wesley, Boston, 1997.

[23] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addision Wesley, Boston, 1998.

[24] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, 2004.

[25] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Enno Ohlebusch, Ulm, 2013.

[26] D. L. T. Rohde, S. Olson, and J. T. Chang. Modelling the recent common ancestry of all living humans. *Nature*, 431:562–566, 2004.

[27] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylgenetic trees. *Molecular Biology and Evolution*, 4:406–425, 1987.

[28] F. Sanger, A. R. Coulson, G. F. Hong, D. F. Hill, and G. B. Petersen. Nucleotide sequence of bacteriophage λ DNA. *Journal of Molecular Biology*, 162:729–773, 1982.

[29] H. Voss. *PSTricks, Grafik mit PostScript für TEX und LATEX*. Dante e.V., 2nd edition, 2005.

[30] G. A. Watterson. On the number of segregating sites in genetical models without recombination. *Theoretical Population Biology*, 7:256–276, 1975.