Chapter 3
Exact Matching

3.1 Keyword Trees

We often need to look up a short sequence, say $P = \texttt{ACA}$, in a longer sequence, say $T = \texttt{CACAGACACAT}$. This is known as the exact matching problem: to find all occurrences of a pattern P in a text T. For our example, the solution is illustrated here:

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

P starts in T at positions 2, 6, and 8.

The simplest method for solving the exact matching problem is to align the start positions of P and T and to compare $P[1]$ and $T[1]$. If there is a mismatch, move P one position to the right and repeat, as shown in Fig. 3.1. In Step 1, a mismatch denoted by 0 is found when comparing $P[1]$ and $T[1]$. As a consequence, P is moved one step to the right and the matching resumed. This time all of P can be matched, and so on, right through to step 9, where $P[1]$ is mismatched with $T[10]$ and the algorithm stops, because P cannot be shifted further to the right. This is called the naïve string matching algorithm.

The speed of string matching is proportional to the number of comparisons. By counting all the zeros and ones in Fig. 3.1, we find that 16 comparisons were carried out. However, consider $P = \texttt{AAA}$ and $T = \texttt{AAAAAAAAAAAA}$. Now P matches at every position in T leading to $10 \times 3 = 30$ comparisons. In fact, this is the worst case with run time, which is proportional to the product of the pattern and text lengths. Such proportionality is expressed using the big-O notation, which you can think of as "order of": $O(|P| \times |T|)$. This multiplicative run time can be avoided by realizing that it is not necessary to restart matching at the first position of P every time: After AAA has been found, the first two As have been matched already. In other words, the suffix $P[2..3]$ matches the prefix $P[1..2]$. So at the next step of the algorithm, only $P[3]$ should be compared to the current text position.

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*, https://doi.org/10.1007/978-3-319-67395-0_3

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

Fig. 3.1 Naïve matching algorithm

Instead of searching for a single pattern, we might be looking for a whole set. Say we are looking for $P_1 = \texttt{AAA}$ and $P_2 = \texttt{AAT}$; whenever `AA` is found, P_1 or P_2 could be detected in the next step. So we have made two observations for improving the naïve approach: suffixes in P might match prefixes of P, and prefixes of multiple patterns might match and hence ought to be summarized. Both of these observations lead to the preprocessing of one or more patterns into a tree, a keyword tree [3]. This guarantees an additive run time $O(|P| + |T|)$ and makes set matching fast.

In this section, we begin by implementing the naïve string matching algorithm in AWK. Then, we learn how to make it go faster.

New Concepts

Name	Comment
keyword tree	fast set matching
naïve string matching	simple but fast in many situations
set matching	matching multiple patterns

New Programs

Name	Source	Help
`fold`	system	`man fold`
`gv`	package manager	`man gv`
`keywordMatcher`	book website	`keywordMatcher -h`
`latex`	package manager	`man latex`
`naiveMatcher`	book website	`naiveMatcher -h`
`revComp`	book website	`revComp -h`
`/usr/bin/time`	system	`man time`

Problem 171 Begin as usual by creating a working directory, `KeywordTrees`, and changing into it. To solve the string matching problem in AWK, we need to address a string at specific positions. The AWK function `split(s, a, fs)` splits string `s` into array `a` on field separator `fs` and returns the number of fields. Use it to write the

AWK program `split.awk` that takes an arbitrary, short text from the command line using the syntax

```
awk -v t=CACAGACACAT -f split.awk
```

to assign a string to the variable `t`. The program then splits this string and iterates over the resulting array to print each character.

Problem 172 Write the program `naive.awk`, which takes a pattern P and a text T from the command line and prints out the starting positions of P in T. Use the AWK command `break` for jumping out of a loop, e.g.,

```
for(i=1; i<=n; i++)
   if (ta[i] == "X")
       break
```

Test your program on $T =$ `CACAGACACAT` and $P =$ `ACA`.

Problem 173 To apply our program `naive.awk` to real sequences, we need to read sequences from FASTA files. AWK can execute system commands like `tail` and access the result by piping it through the AWK function `getline`:

```
BEGIN{
     cmd = "tail -n +2 " file
     while(cmd | getline)
          t = t $1
     print t
}
```

The function `getline` sets the AWK variables `$1`, `$2`, and so on as if the program were reading from a file. This could be run as

```
awk -f readFasta.awk -v file=mgGenome.fasta
```

Write `naive2.awk`, which reads a sequence from file and a pattern in FASTA format from stdin, and prints out the pattern's start positions. Where does `ACGTCG` occur in the genome of *M. genitalium*?

Problem 174 The program `revComp` computes the reverse complement of a sequence. Compute the reverse complement of `mgGenome.fasta`. Does it contain `CGGCCT`?

Problem 175 As we already said at the beginning, the naïve algorithm becomes slow when confronted with a pattern that matches everywhere, our example was $P =$ `AAA`, $T =$ `AAAAAAAAAA`. In that case, the run time is expected to be proportional to the product of the lengths of P and T, $O(|P| \times |T|)$. To explore the behavior of `naive2.awk` in this worst case, write a program that takes as input a sequence length, n, from the command line and writes a FASTA header followed by a single line of n `A`s. Call the program `monoNuc.awk`. Run its output through the UNIX program `fold` to wrap the `A`s into lines of length 80, which is the default output of `fold`.

Problem 176 Use `monoNuc.awk` and the UNIX-program `time` to get a run time for `naive2.awk` when applied to a text of 1 Mb and a pattern of 10 bp. What happens if you double the pattern and the text length? You can avoid printing all match positions to the screen by piping the output through `tail`.

Problem 177 Scripting languages like AWK tend to be slower than compiled languages like C. The program `naiveMatcher` is written in C and implements the same algorithm as `naive2.awk`. What is the run time of `naiveMatcher` compared to `naive2.awk` when searching for a 20-nucleotide pattern consisting of `A`s in `2mb.fasta`?

Problem 178 Draw the run time of `naiveMatcher` as a function of the length of the pattern consisting entirely of `A` when searching `2mb.fasta`. Use pattern lengths of 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, and 10000. Hint: Use

```
/usr/bin/time -p <command> 2>&1 | grep real
```

instead of plain `time`, to generate a table of run times ready for plotting. The command redirects the output of `time` from the error stream (2) to the standard output stream (1) and hence into the pipeline.

Problem 179 The naïve matching algorithm outlined in Fig. 3.1 can also be illustrated in a different way. Figure 3.2a shows the pattern to be matched as a graph consisting of nodes and edges depicted as arrows. Each node represents a state in the matching procedure and each arrow a response to match or mismatch. Match is illustrated by gray arrows, mismatch by orange arrows. The defining characteristic of the naïve algorithm is that upon every failure, matching resumes at the beginning of P; hence, all orange arrows in Fig. 3.2a point to the first node. However, a match to $P = \texttt{ACA}$ implies a match to the first character of P. Therefore, instead of returning to the beginning of P, the algorithm only needs to compare the characters from $P[2] = \texttt{C}$ onward. This is illustrated in Fig. 3.2b. What are the failure links for $P = \texttt{AAA}$?

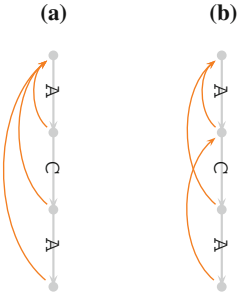


Fig. 3.2 A pattern to be matched shown as a graph. Gray arrows indicate matches, orange arrows "failure links" that are followed upon mismatch. Naïve failure links (**a**) always return to the beginning of the pattern. Better failure links (**b**) incorporate the fact that after the last `A` has been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

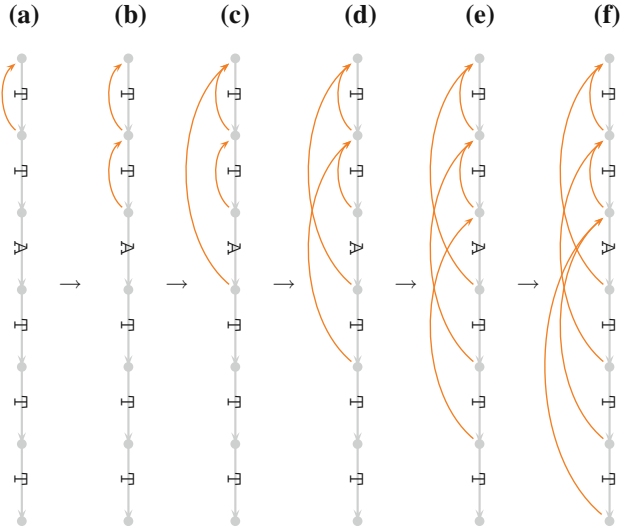


Fig. 3.3 Systematic construction of failure links going from the initialization (**a**) through to the fully preprocessed pattern (**f**)

Problem 180 To systematically construct failure links, we begin by stating that a mismatch after the first match means: return to the beginning (Fig. 3.3a). After this initialization step, we work our way from top to bottom by following failure links until the earliest match to the character directly above the node to be connected. The node we reach is the target of the next failure link. For example, in Fig. 3.3b we look for a match to `T`; after following the existing failure link, we follow the match link and have thus found the target for the failure link. Next, two existing failure links are followed without a match, and hence the new failure link points to the start node (Fig. 3.3c). Then, another failure followed by a match in Fig. 3.3d, and this pattern is repeated in Fig. 3.3e. Finally, two failure links need to be followed before a match is found leading to the completely preprocessed pattern in Fig. 3.3f. Construct the failure links for $P =$ `ATATAT`.

Problem 181 The program `keywordMatcher` looks for exact matches after preprocessing the pattern with failure links. Its name comes from the fact that the preprocessed pattern in Fig. 3.2 is called a "keyword tree". What is the run time of `keywordMatcher` when searching for a string of 20 `A` in `2mb.fasta`? Compare your result to the corresponding run time of `naiveMatcher`.

Problem 182 Apply `keywordMatcher` to the same increasingly long patterns used in Problem 178 and plot the results in the same graph as the times for `naiveMatcher`.

Problem 183 Compare the run times of `naiveMatcher` and `keywordMatcher` when searching a random sequence of length 20, say

$$P = \texttt{TTTAACCTCCGGCGGAGTTT}$$

in random sequences of lengths 1, 2, 5, 10, 20, 50, 100 Mb (`ranseq`). Plot the results in a single graph.

Problem 184 Keyword trees were originally developed for set matching [3] and the program `keywordMatcher` can search for many patterns simultaneously. Say, we wish to look for five patterns (keywords): $P_1 =$ `ACG`, $P_2 =$ `AC`, $P_3 =$ `ACT`, $P_4 =$ `CGA`, and $P_5 =$ `C`. Notice that P_2 is contained in P_1 and P_3, P_5 in all others, and P_1 and P_3 have the matching prefix `AC`. To make searching efficient, matching prefixes are summarized as common paths in the keyword tree. Its construction is shown in Fig. 3.4. The patterns are sequentially fitted into a growing tree structure. The first partial tree for $P_1 =$ `ACG` in Fig. 3.4b should look familiar from our graphs for single patterns, except for the label on the end node indicating the pattern just matched.

Repeat the keyword tree construction using paper and pencil. Then, initialize the failure link construction as in Fig. 3.4f and enter the five missing failure links.

Problem 185 Keyword trees can be drawn automatically using the `-t` option of `keywordMatcher`, for example,

```
keywordMatcher -t kt.tex -p ACA mgGenome.fasta > /dev/
    null
```

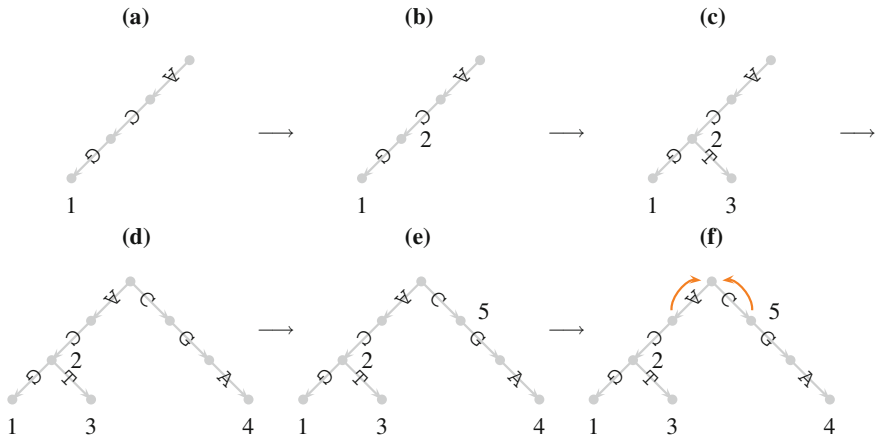


Fig. 3.4 Sequential construction of a keyword tree for the five patterns $P_1 =$ `ACG`, $P_2 =$ `AC`, $P_3 =$ `ACT`, $P_4 =$ `CGA`, and $P_5 =$ `C`

writes the tree for ACA to the file kt.tex and discards the output by writing it to the null device. The file kt.tex contains code in the typesetting language LaTeX [28, 30]. To view this, apply the program latex to the wrapper file for kt.tex, ktWrapper.tex. Like kt.tex, it is generated automatically and contains LaTeX commands. If you have never used LaTeX, take a look at ktWrapper.tex. Commands start with a backslash character. For example, the line

```
\input{kt.tex}
```

imports the keyword tree contained in the file kt.tex. Apart from commands, you can also enter ordinary text in a LaTeX document (try this). Like many programming languages, LaTeX needs to be compiled to generate the desired output, a neatly typeset page:

```
latex ktWrapper.tex
```

This generates the file ktWrapper.dvi, which is then converted to the postscript file ktWrapper.ps

```
dvips ktWrapper.dvi
```

ready for viewing:

```
gv ktWrapper.ps &
```

Automatically generate the keyword tree for our five example patterns $P_1 = \texttt{ACG}$, $P_2 = \texttt{AC}$, $P_3 = \texttt{ACT}$, $P_4 = \texttt{CGA}$, and $P_5 = \texttt{C}$. Also, what happens if you delete the line

```
\date{}
```

in ktWrapper.tex and run LaTeX again?

Problem 186 Use the keyword tree just constructed to trace with pencil and paper the search for P_1–P_5 in $T = \texttt{ACGC}$. The rule is, whenever a labeled node is reached, the corresponding pattern has been found. However, that rule is not sufficient for finding all patterns. Can you see where it fails, and how this might be fixed?

Problem 187 Write down the output set next to each node on our example keyword tree. Each output set contains at least the current label but perhaps also additional elements.

3.2 Suffix Trees

Think of your favorite book, say the first volume of *Harry Potter*. How would you find every passage that contains the word "Voldemort"? You could scan every page. But there are many repeated words and a few repeated phrases that you would scan again and again. So, would not it be useful if you could compress the book such that every repeat is only listed once together with its occurrences? An index is such a compressed version of a book. Novels usually do not have one but textbooks usually do. However, even the most detailed index does not contain every word, because readers do not need to look up insignificant words like "and". Still, sometimes it is desirable to have an index of every possible word.

A suffix tree of a text is an index that references not only every word but every possible substring [21]. Consider as text `ADAM`. To construct the corresponding suffix tree, start by drawing a root and a leaf node connected by an edge (Fig. 3.5a). Label the edge with the first suffix, `ADAM`, and the label with 1 to indicate the end of the first suffix (Fig. 3.5b). Next, take the suffix starting at position 2, `DAM`, and fit it into the tree starting from the root. Since its first character, `D`, already mismatches the first `A` of `ADAM`, create a new branch and label it as before (Fig. 3.5c). The third suffix, `AM`, can be fitted one step into the tree before a mismatch occurs; apart from a leaf, this also creates an internal node (Fig. 3.5d). The last suffix, `M`, again branches directly off the root (Fig. 3.5e).

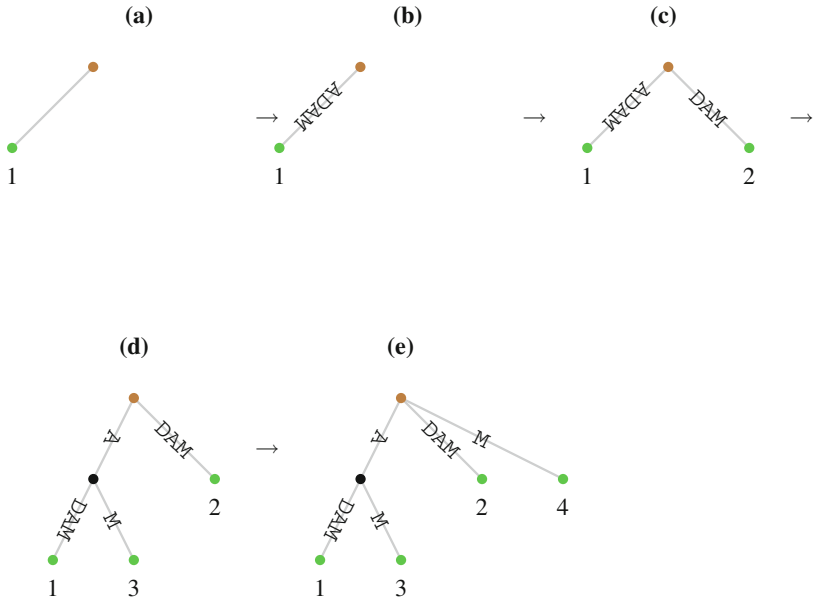


Fig. 3.5 Construction of the suffix tree that indexes `ADAM`. The root is shown in brown, leaves in green, and the single internal node in black

If you now look for the occurrence of A in ADAM, start searching at the root of the suffix tree. The leaf labels below the match, in this case 1 and 3, indicate the positions of A in ADAM. This means that—like a book index—a suffix tree enables us to look up a word in time proportional to the length of the pattern. But unlike a book index, a suffix tree references *all* possible words, or substrings, of a text. Surprisingly, a suffix tree not only provides a perfect index but it can also be computed efficiently, as we explain in this section.

New Concepts

Name	Comment
longest repeat	found using suffix tree
shortest unique substring	becomes repeat when trimmed on the right
suffix tree	text index

New Program

Name	Source	Help
`shustring`	book website	`shustring -h`

Problem 188 As we saw in Fig. 3.5, a suffix starts somewhere in a sequence and ends at its end. Consider the sequence $S =$ ACCCG and write down its suffixes.

Problem 189 Write down the suffix tree for S by following the procedure illustrated in Fig. 3.5.

Problem 190 Use the suffix tree constructed in Problem 189 to search for CC in S by matching from the root as explained for Fig. 3.5.

Problem 191 Suppose our sequence mutates at its last position from a G to an A, such that now $S =$ ACCCA. Write down the suffix tree for this sequence. What do you observe?

Problem 192 Write down the suffix tree for $S =$ ACCCA$.

Problem 193 The concatenated strings along the path leading from the root of the suffix tree to a node is called the node's "path label". The length of this path label is the node's "string depth". Add string depths to the internal nodes of the suffix tree constructed in Problem 189.

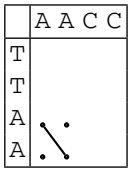
Problem 194 Every path label that leads to an internal node is a repeated string. Use the suffix tree with string depths constructed in Problem 193 to find the longest repeat in S.

Problem 195 Make the directory `SuffixTrees`, change into it, and copy the genome of *M. genitalium* contained in `mgGenome.fasta` from `Data` to your current directory. The program `repeater` computes a suffix tree of the input sequence to find longest repeats. How long is the longest repeat in the genome of *M. genitalium*?

Problem 196 Next, we compute the length of the longest repeat expected in a random version of the *M. genitalium* genome. We begin by calculating the probability of a match between two random nucleotides. Since there are four possibilities of obtaining a match, `AA`, `CC`, `TT`, and `GG`, the probability of randomly drawing two identical nucleotides from a sequence in which each nucleotide has frequency $1/4$ is $1/16 \times 4 = 1/4$. However, in real sequences, the nucleotides usually do not occur with equal frequencies. Use the program `cchar` to compute the nucleotide composition of *M. genitalium* (`mgGenome.fasta`). What is the probability of drawing `AA` when picking two random nucleotides from the genome of *M. genitalium*?

Problem 197 Write a pipeline to calculate the probability of drawing two identical nucleotides from the genome of *M. genitalium*.

Problem 198 To get from the match probability, P_m, to the expected length of the longest repeat in the genome of *M. genitalium*, consider a toy dot plot with three matches, two of length 1 and one of length 2:



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

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

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

Problem 199 To check the expected match length computed in Problem 198, randomize the sequence of *M. genitalium* using the program `randomizeSeq` and compute the longest repeat in that randomized version. Do this a few times and compare your results to the expectation.

Problem 200 Most books come without an index. One reason for this is that making an index is a lot of work. We have already seen that `repeater` is quick when applied to the genome of *M. genitalium*. However, with just half a megabase, this genome is very small compared to, say, ours with 3.2 gigabases. So the question is, how time-consuming is suffix tree construction in general? Consider the "naïve" construction

method depicted in Fig. 3.5. How does its run time scale as a function of sequence length? Approach this question by first constructing a suffix tree for a sequence consisting of only one type of nucleotide, for example, `AAAA`.

Problem 201 The program `repeater` uses a more efficient algorithm than the naïve construction in Fig. 3.5. To investigate how the more sophisticated algorithm scales with sequence length, use the program `ranseq` to simulate sequences of 1, 2, 5, 10, and 20 Mb. Apply `repeater` to them, measure its run time with `time`, and plot it. What is the run time of `repeater`?

Problem 202 Repeat the resource analysis of `repeater`, but instead of time, measure memory using commands like

```
ranseq -l 1000000 | /usr/bin/time -f "%M kb" repeater
```

on Linux or

```
ranseq -l 1000000 | /usr/bin/time -l repeater
```

on Mac OS X.

3.3 Suffix Arrrays

The nodes of suffix trees consume a lot of computer memory. This becomes a problem when computing suffix trees for very long sequences such as mammalian genomes with their billions of nucleotides. Hence, a space-saving alternative to suffix trees has been developed, which consists solely of the sorted array of suffixes [36].

Problem 203 To compute a suffix array, we again begin by writing down every suffix of the input sequence. But this time we automate the procedure. Save our sequence $S =$ `ACCCA$` in the file `s.fasta`. Then, copy the program

```
!/^>/{
     s = s $1; # read the input sequence
}END{
     L = length(s);
     for(i=1; i<=L; i++)
         print i "\t" substr(s, i);
 }
```

into the file `suf.awk` and use it to generate the suffixes of S and their starting positions. Next, sort the suffixes alphabetically. Finally, number the lines in your output. Compare your result to the suffix array shown in Fig. 3.6a.

(a)

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

(b)

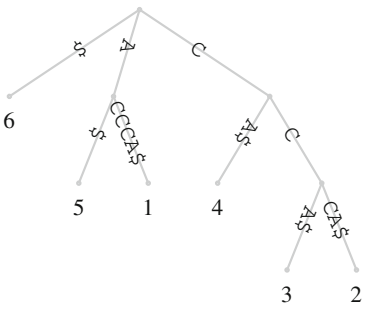


Fig. 3.6 Suffix array (**a**) and corresponding suffix tree (**b**) of ACCCA$

New Concepts

Name	Comment
enhanced suffix array	suffix array plus lcp array
inverse suffix array	suffixes in textual order
lcp array	lengths of longest common prefixes in a suffix array
lcp interval tree	suffix tree from enhanced suffix array
suffix array	sorted list of suffixes

Problem 204 Figure 3.6b shows the suffix tree for the sequence ACCCA$, which should look familiar from Chap. 3.2. It is closely related to the suffix array next to it. Take the internal node with path label A. The leaves connected to that node refer to suffixes 5 and 1. These are neighbors in the suffix array, where they occupy entries sa[2] = 5 and sa[3] = 1. We denote this interval sa[2..3]. Write down the suffix array intervals that correspond to the two remaining internal nodes and the root.

Problem 205 To make the close relationship between suffix array and suffix tree more explicit, we add to our suffix array the array cp for "common prefix", where cp[i] is the longest common prefix between sa[i] and sa[$i-1$]. For example, suf[6] in Fig. 3.6a is CCCA$, and suf[5] = CCA$; since these two suffixes have the common prefix CC, cp[6] = CC. Add cp to the suffix array and mark the prefixes in the suffix tree. The first suffix in sa cannot be compared to another suffix; hence, it is "not defined", nd.

Problem 206 To reconstruct the suffix tree from its suffix array, we need the lengths of the common prefixes, rather than the prefixes themselves. Add the lcp array to the suffix array, where lcp[i] is the length of cp[i].

Problem 207 The distinct entries in the lcp array computed in Problem 206, 0, 1, and 2, correspond to the string depths in the suffix tree in Fig. 3.6. Our aim is still to reconstruct that tree by traversing the suffix array enhanced by the lcp array. For this purpose, we first extend the lcp array by a "stop" entry, -1, as shown in Fig. 3.7a. In addition, we write next to the lcp array an empty table with the three distinct string

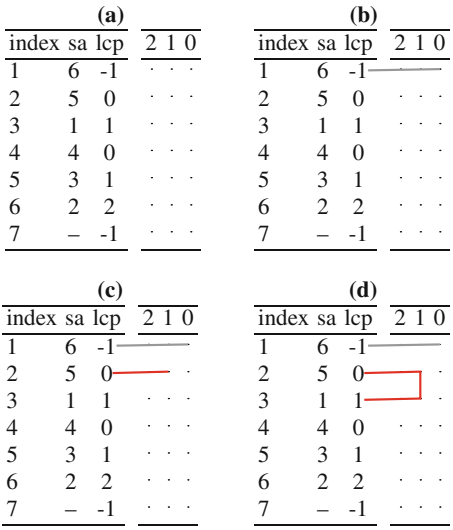


Fig. 3.7 The enhanced suffix array with an auxiliary table for reconstructing the corresponding suffix tree

depths 2, 1, and 0 as headers. Now we traverse the lcp array to find the intervals in the sa array that corresponds to the suffix tree. Starting from the top, check at each position the relationship between the current entry in the lcp-array, lcp[i] and the next entry, lcp[$i+1$]:

- If lcp[i] $<$ lcp[$i+1$], open one or more intervals. The number of intervals to open is lcp[$i+1$] $-$ lcp[i]. The string depths of the opened intervals are the values between lcp[i] $+1$ and lcp[$i+1$]. In our example, lcp[1] $=-1$ and lcp[2] $=0$; since lcp[1] $<$ lcp[2], we open $0--1=1$ interval with string depth $-1+1=0$. To denote this, we draw the gray line in Fig. 3.7b. Similarly, in the next step, we observe lcp[2] $<$ lcp[3] and open another lcp interval, as shown by the red line in Fig. 3.7c.
- If lcp[i] $>$ lcp[$i+1$], close one or more intervals where the string depth is greater than lcp[$i+1$] and has occurred in the interval to be closed. In our example, we observe in Fig. 3.7d that lcp[3] $>$ lcp[4]. Since the string depth of the red, but not of the gray interval, is greater than lcp[4], and it occurs in the interval under consideration, the red interval gets closed, while the gray interval remains open.

Find the remaining lcp intervals.

Problem 208 Convert the nested structure of the lcp intervals in Problem 207 to a tree, the lcp interval tree [38, p. 85ff]. Each node has the format $\ell - [i..j]$, where ℓ is the string depth, i is the start index, and j is the end index.

Problem 209 Let us study the relationship between enhanced suffix array and suffix tree with a longer example sequence, $S =$ `GTAAACTATT`. Write down its enhanced suffix array, the nested lcp intervals, and the suffix tree.

Problem 210 As we have seen, lcp arrays allow the conversion of suffix arrays to suffix trees. The efficient computation of lcp arrays is therefore central to the application of suffix trees. In the following couple of problems, we learn how to do this. The crucial insight is that the lcp value for a suffix $S[i..]$ forms a lower bound of the lcp value of the suffix one step to the right, $S[i+1..]$. Consider, for example, the sequence `ACCCACG`. Its first suffix matches `AC` at position 5; from this, we can conclude that its second suffix matches at least the `C` at position 6 from the previous match. In other words, if ℓ is the length of the common prefix of $S[i..]$, then $\ell - 1$ is the lower bound of the lcp for $S[i+1..]$. The emphasis here is on *lower bound*; in our example, the `CC` at the beginning of the second suffix match is the `CC` at the beginning of the third. To use this lower bound insight, we need to traverse the suffix array in the same order in which the suffixes appear in the input sequence. The mapping between sa and S is called the inverse suffix array, isa, which is defined as

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

Add the isa to the suffix array of `ACCCA$`. This is most effective when done to the left of the index i.

Problem 211 Write the program `isa.awk` that takes as input the sorted output from `suf.awk` and adds the isa as another column.

Problem 212 Algorithm 2 shows how to compute the lcp values in linear time [27]. The inverse suffix array, isa, is computed in lines 1–2, which should look familiar from Problem 211. Implement this algorithm in a program `esa.awk` that takes as input the sorted output of `suf.awk` and prints the enhanced suffix array, that is, sa and lcp. Hint: The AWK function

```
substr(s, p, n)
```

returns the substring of `s` that starts at `p` and is `n` characters long.

Problem 213 Now, we apply our program to real sequences. What is the longest repeat sequence in *Adh/Adh-dup* of *Drosophila guanche* (`dgAdhAdhdup.fasta`)?

Problem 214 What is the longest repeat sequence in *Adh/Adh-dup* of *Drosophila melanogaster* (`dmAdhAdhdup.fasta`)?

Problem 215 By typing

```
cat dmAdhAdhdup.fasta dgAdhAdhdup.fasta > dmDgAdhAdhdup
    .fasta
```

the two alcohol dehydrogenases we just investigated can be written into the same file. What is the longest repeat in the concatenated sequences? Its length will be greater than that of the repeats within the individual sequences. Can you think of why?

Algorithm 2 Algorithm for computing the lengths of common prefixes [27]

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

Problem 216 In a suffix tree, repeats are path labels ending above internal nodes. If instead we look at the path labels of leaves, we find unique sequences. For example, `CCC` in Fig. 3.6 is unique, as it is found on the edge connected to leaf 2. The extension of this motif to `CCCA` does not change its property of uniqueness as we are still on the path to leaf 2. Hence, when talking about unique motifs, we are particularly interested in *shortest* motifs, which we call shortest unique substrings, or shustrings. By traversing all leaves, the shustrings starting at every position in a sequence can be determined: At each leaf, extend its parent's path label by a single nucleotide. Write down the shustrings for each position in `ACCCA`. What is the length of the shortest shustring(s)? What might be the use of finding such shortest shustrings in real sequences?

Problem 217 Instead of using the suffix tree of `ACCCA`, we can also use the lcp array to look up the shustring lengths at every position. How is this done?

Problem 218 The program `shustring` implements the search for shustrings based on suffix arrays. What is the length of the shortest unique substrings in the genome of *M. genitalium*?

Problem 219 By default, `shustring` only searches the forward strand. Does the result change when the reverse strand is included?

Problem 220 Instead of looking for the globally shortest shustrings, you can also look for shustrings of a certain minimal (`-m`) or maximal (`-M`) length. This requires the *local* option, which is invoked for a particular sequence; in our case, there

is only one sequence to choose from, so anything that matches the header line in `mgGenome.fasta` will do the trick, for example

```
-l .
```

because in a regular expression a dot matches anything. How many shustrings of length ≤ 7 are contained in the genome of *M. genitalium*?

3.4 Text Compression

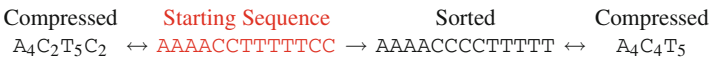


Fig. 3.8 Compressing sequences

The ability to compress data underlies many powerful programs ranging from mp3 players to read mappers in Bioinformatics. Given the starting sequence in Fig. 3.8, it can be compressed by counting repeated nucleotides and writing them as, for example, A_4. This kind of compression is easy to reverse, though only effective if there are long runs of identical nucleotides. Sorting the sequence would maximize such runs, which would lead to the greatest compression. However, sorting is irreversible. The Burrows–Wheeler transform was devised to reversibly increase the runs of identical symbols in any type of data [12]. We begin by transforming the word "Mississippi":

```
          111
123456789012
mississippi$
```

It consists of eleven characters plus a sentinel at the end. The first step in the transform is to write down all the rotations of the word

```
 1 mississippi$
 2 ississippi$m
 3 ssissippi$mi
 4 sissippi$mis
 5 issippi$miss
 6 ssippi$missi
 7 sippi$missis
 8 ippi$mississ
 9 ppi$mississi
10 pi$mississip
11 i$mississipp
12 $mississippi
```

Next, the rotations are sorted:

String	Rotations	Sorted Rotations	Transformed String
	mississippi$	$mississippi	
	ississippi$m	i$mississipp	
	ssissippi$mi	ippi$mississ	
	sissippi$mis	issippi$miss	
	issippi$miss	ississippi$m	
mississippi$	ssippi$missi	mississippi$	ipssm$pissii
	sippi$missis	pi$mississip	
	ippi$mississ	ppi$mississi	
	ppi$mississi	sippi$missis	
	pi$mississip	sissippi$mis	
	i$mississipp	ssippi$missi	
	$mississippi	ssissippi$mi	

The first column of the sorted rotations consists of the characters in Mississippi in alphabetical order. As we have already observed, this cannot be decoded to the original word. But intriguingly, the last column can, and this is our transform.

How is this transform reversed, that is, decoded? We start by sorting the transform to give us the first column of the sorted rotations, column $\mathscr{F}$ in Fig. 3.9a, while the transform is in column $\mathscr{L}$. Then we label each character in $\mathscr{F}$ and $\mathscr{L}$ with its count, so the first `i` is labeled $\texttt{i}_1$, the second $\texttt{i}_2$, etc. (Fig. 3.9b). Finally, we look for the sentinel in $\mathscr{L}$, jump across to $\mathscr{F}$, and have found the first character of the original string, $\texttt{m}_1$, in our example (Fig. 3.9c). Next, we look up $\texttt{m}_1$ in $\mathscr{L}$, and jump across to $\mathscr{F}$ to find our second character, $\texttt{i}_4$, and so on until this traceback yields the original "mississippi$". In this section we explore the effect of BWT and how it can be used to compress sequences.

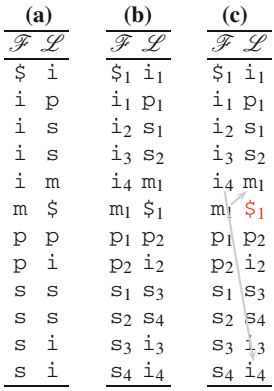
New Concepts

Name	Comment
Burrows–Wheeler transform	Reversible sorting
Compressibility	Opposite of complexity
Lempel–Ziv decomposition	Measure sequence complexity
Move to front	Reduce sequence complexity

New Programs

Name	Source	Help
`bwt`	Book website	`bwt -h`
`bzip2`	System	`man bzip2`
`du`	System	`man du`
`gzip`	System	`man gzip`
`lzd`	Book website	`lzd -h`
`mtf`	Book website	`mtf -h`

Problem 221 Write down the Burrows–Wheeler transform of the sequence `TACTA` on a piece of paper. Create the directory Bwt and change into it. Check your manual transform using the program `bwt`.



$\mathscr{F}$	$\mathscr{L}$
$	i
i	p
i	s
i	s
i	m
m	$
p	p
p	i
s	s
s	s
s	i
s	i

$\mathscr{F}$	$\mathscr{L}$
$_1$	i$_1$
i$_1$	p$_1$
i$_2$	s$_1$
i$_3$	s$_2$
i$_4$	m$_1$
m$_1$	$_1$
p$_1$	p$_2$
p$_2$	i$_2$
s$_1$	s$_3$
s$_2$	s$_4$
s$_3$	i$_3$
s$_4$	i$_4$

$\mathscr{F}$	$\mathscr{L}$
$_1$	i$_1$
i$_1$	p$_1$
i$_2$	s$_1$
i$_3$	s$_2$
i$_4$	m$_1$
m$_1$	$_1$
p$_1$	p$_2$
p$_2$	i$_2$
s$_1$	s$_3$
s$_2$	s$_4$
s$_3$	i$_3$
s$_4$	i$_4$

Fig. 3.9 Decoding the Burrows–Wheeler transform: write down the first ($\mathscr{F}$) and last ($\mathscr{L}$) columns of the rotations **a**; count characters **b**; trace back **c**, showing the first two steps

Problem 222 To observe the effect of BWT on natural language, we next transform Shakespeare's *Hamlet*, which is contained in `hamlet.fasta`. Begin by looking at the file to convince yourself that it contains the bard's words, albeit in a slightly unorthodox format:

```
less hamlet.fasta
```

When you press `d`, `less` moves down by half a page, `u` does the opposite, and `q` lets you quit. One advantage of `less` is that text can be piped into it. For example, the BWT of *Hamlet* is

```
bwt hamlet.fasta | cat -n | less
```

Take a look at the first couple of pages. What do you observe?

Problem 223 Decode by hand the BWT `CCCT$G`. Check the result using `bwt`.

Problem 224 We have seen that decoding a BWT is relatively simple, but encoding is tedious due to the rotation step. However, by using a suffix array we can avoid the rotations. To see how this works, return to our original example, $T =$ `mississippi$` and write down its suffix array using the program `suf.awk` from Problem 203. Notice that up to the sentinel rotations and suffixes are identical. Can you think of a method for finding the BWT by traversing the sa?

Problem 225 Construct the suffix array of $T =$ `TACTA$` and use it to infer the BWT of T. Check your result using `bwt`.

3.4.1 Move to Front (MTF)

To make the BWT more compressible, we can apply the move to front (MTF) procedure [2]. This works by noting the position in the alphabet of a given character. So to encode the nucleotide sequence $S = \texttt{GTTT}$, we use as alphabet the four nucleotides:

$$\begin{matrix} 0 & 1 & 2 & 3 \\ \texttt{A} & \texttt{C} & \texttt{G} & \texttt{T} \end{matrix}$$

Characters are encoded by their position in the alphabet and this position can change after each encoding step: The first G in S is encoded as 2; in the next iteration, the G is moved to the front of the alphabet

$$\begin{matrix} 0 & 1 & 2 & 3 \\ \texttt{G} & \texttt{A} & \texttt{C} & \texttt{T} \end{matrix}$$

and the T is a 3. Now the T is moved to the front

$$\begin{matrix} 0 & 1 & 2 & 3 \\ \texttt{T} & \texttt{G} & \texttt{A} & \texttt{C} \end{matrix}$$

The remaining two T are encoded as 0, yielding 2, 3, 0, 0. To decode this, we reverse the procedure. Start with ACGT; the first 2 stands for G, rearrange alphabet to GACT; the 3 stands for T, rearrange alphabet to TGAC; the two zeros stand for T, and we have GTTT back. The crucial idea of MTF is that repeats of *any* nucleotide lead to runs of zeros, which can then be compressed.

Problem 226 Use MTF to manually encode $T = \texttt{GTTAG}$. Check the result using `mtf`.

Problem 227 Let 1,0,1,3,3 be the result of MTF on the alphabet ACGT. Decode the original string.

3.4.2 Measuring Compressibility: The Lempel–Ziv Decomposition

In order to measure the effect of BWT and MTF, we need to quantify the extent to which a transform can be compressed, compared to the original sequence. One way to do this is to look for matching regions or "factors" within the input sequence. The fewer such match factors can be found, the more compressible the sequence is. Take for example the sequence $S = \texttt{CCCG}$; at every position i in S we ask, how long is the longest match that appears somewhere to the left of i? The first position has no left neighbor, so the rule for no match is invoked: $S[i]$ becomes a factor, C, and we move to the next position $S[i + 1]$. This matches the first position up to the forth, so our second factor is CC. The last position, $S[4] = \texttt{G}$, again has no match, so S is decomposed into the three factors

C.CC.G

This decomposition is called the Lempel–Ziv decomposition [32] and is widely used in compression programs, for example in `gzip`.

Problem 228 Calculate by hand the Lempel–Ziv decomposition of `TACTA`. Check your result using the program `lzd`.

Problem 229 Calculate by hand the Lempel–Ziv decomposition of the maximally redundant sequence `AAAAA`. Again use `lzd` to check your result.

Problem 230 Instead of directly measuring compressibility, it is often simpler to measure its opposite, which is complexity. Highly complex sequences cannot be compressed much, and vice versa. We define as complexity the number of Lempel–Ziv factors divided by sequence length:

$$C = \frac{\text{number of LZ factors}}{|S|}.$$

What are the complexities of the example sequences in Problems 228 and 229?

Problem 231 What are the theoretical limits of C?

Problem 232 In practice, the largest value C can take is the number of LZ factors per nucleotide in a random sequence. Use `ranseq` to generate random sequences of lengths 1, 2, 5, and 10 kb, and measure C. Plot C as a function of sequence length. Is the maximum of C constant with respect to sequence length?

Problem 233 Next, we compute C for the genome of *M. genitalium* to observe the effect of the Burrows–Wheeler transform, BWT, and move to front, MTF, on complexity. Fill in the table below:

Operation	Number of Factors	C
None		
BWT		
BWT | MTF		
MTF		
MTF | BWT		

Which combination gives the largest reduction in C?

Problem 234 We return to compression, the opposite of complexity. Two examples of popular compression programs are `gzip` and `bzip2`. The "zip" in these program names refers to Lempel–Ziv, the "b" in `bzip2` to BWT. Measure the size of `mgGemone.fasta` by using the program `du` for disk usage:

```
du -h mgGenome.fasta
```

Then compute the size of the compressed genome after it has undergone randomization, BWT, and MTF. Summarize your results in the following table:

Operation	`gzip`	`bzip2`
Nothing		
`randomizeSeq`		
BWT		
BWT | MTF		