Chapter 7

(Space and Time Trade-Offs)

7

Space and Time Trade-Offs

Things which matter most must never be at the mercy of things which matter less.

—Johann Wolfgang von Göethe (1749–1832)

Space and time trade-offs in algorithm design are a well-known issue for both theoreticians and practitioners of computing. Consider, as an example, the problem of computing values of a function at many points in its domain. If it is time that is at a premium, we can precompute the function's values and store them in a table. This is exactly what human computers had to do before the advent of electronic computers, in the process burdening libraries with thick volumes of mathematical tables. Though such tables have lost much of their appeal with the widespread use of electronic computers, the underlying idea has proven to be quite useful in the development of several important algorithms for other problems. In somewhat more general terms, the idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. We call this approach ***input enhancement***[1] and discuss the following algorithms based on it:

- counting methods for sorting (Section 7.1)
- Boyer-Moore algorithm for string matching and its simplified version suggested by Horspool (Section 7.2)

The other type of technique that exploits space-for-time trade-offs simply uses extra space to facilitate faster and/or more flexible access to the data. We call this approach ***prestructuring***. This name highlights two facets of this variation of the space-for-time trade-off: some processing is done before a problem in question

1. The standard terms used synonymously for this technique are ***preprocessing*** and ***preconditioning***. Confusingly, these terms can also be applied to methods that use the idea of preprocessing but do not use extra space (see Chapter 6). Thus, in order to avoid confusion, we use "input enhancement" as a special name for the space-for-time trade-off technique being discussed here.

4. Is the distribution-counting algorithm stable?

5. Design a one-line algorithm for sorting any array of size n whose values are n distinct integers from 1 to n.

6. The ***ancestry problem*** asks to determine whether a vertex u is an ancestor of vertex v in a given binary (or, more generally, rooted ordered) tree of n vertices. Design a $O(n)$ input-enhancement algorithm that provides sufficient information to solve this problem for any pair of the tree's vertices in constant time.

7. The following technique, known as ***virtual initialization***, provides a time-efficient way to initialize just some elements of a given array $A[0..n-1]$ so that for each of its elements, we can say in constant time whether it has been initialized and, if it has been, with which value. This is done by utilizing a variable $counter$ for the number of initialized elements in A and two auxiliary arrays of the same size, say $B[0..n-1]$ and $C[0..n-1]$, defined as follows. $B[0], \ldots, B[counter-1]$ contain the indices of the elements of A that were initialized: $B[0]$ contains the index of the element initialized first, $B[1]$ contains the index of the element initialized second, etc. Furthermore, if $A[i]$ was the kth element ($0 \le k \le counter-1$) to be initialized, $C[i]$ contains k.

   a. Sketch the state of arrays $A[0..7]$, $B[0..7]$, and $C[0..7]$ after the three assignments

   $$A[3] \leftarrow x; \qquad A[7] \leftarrow z; \qquad A[1] \leftarrow y.$$

   b. In general, how can we check with this scheme whether $A[i]$ has been initialized and, if it has been, with which value?

8. *Least distance sorting* There are 10 Egyptian stone statues standing in a row in an art gallery hall. A new curator wants to move them so that the statues are ordered by their height. How should this be done to minimize the total distance that the statues are moved? You may assume for simplicity that all the statues have different heights. [Azi10]

9. **a.** Write a program for multiplying two sparse matrices, a $p \times q$ matrix A and a $q \times r$ matrix B.

   b. Write a program for multiplying two sparse polynomials $p(x)$ and $q(x)$ of degrees m and n, respectively.

10. Is it a good idea to write a program that plays the classic game of tic-tac-toe with the human user by storing all possible positions on the game's 3×3 board along with the best move for each of them?

7.2 Input Enhancement in String Matching

In this section, we see how the technique of input enhancement can be applied to the problem of string matching. Recall that the problem of string matching

requires finding an occurrence of a given string of m characters called the ***pattern*** in a longer string of n characters called the ***text***. We discussed the brute-force algorithm for this problem in Section 3.2: it simply matches corresponding pairs of characters in the pattern and the text left to right and, if a mismatch occurs, shifts the pattern one position to the right for the next trial. Since the maximum number of such trials is $n - m + 1$ and, in the worst case, m comparisons need to be made on each of them, the worst-case efficiency of the brute-force algorithm is in the $O(nm)$ class. On average, however, we should expect just a few comparisons before a pattern's shift, and for random natural-language texts, the average-case efficiency indeed turns out to be in $O(n + m)$.

Several faster algorithms have been discovered. Most of them exploit the input-enhancement idea: preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text. This is exactly the idea behind the two best-known algorithms of this type: the Knuth-Morris-Pratt algorithm [Knu77] and the Boyer-Moore algorithm [Boy77].

The principal difference between these two algorithms lies in the way they compare characters of a pattern with their counterparts in a text: the Knuth-Morris-Pratt algorithm does it left to right, whereas the Boyer-Moore algorithm does it right to left. Since the latter idea leads to simpler algorithms, it is the only one that we will pursue here. (Note that the Boyer-Moore algorithm starts by aligning the pattern against the beginning characters of the text; if the first trial fails, it shifts the pattern to the right. It is comparisons within a trial that the algorithm does right to left, starting with the last character in the pattern.)

Although the underlying idea of the Boyer-Moore algorithm is simple, its actual implementation in a working method is less so. Therefore, we start our discussion with a simplified version of the Boyer-Moore algorithm suggested by R. Horspool [Hor80]. In addition to being simpler, Horspool's algorithm is not necessarily less efficient than the Boyer-Moore algorithm on random strings.

Horspool's Algorithm

Consider, as an example, searching for the pattern BARBER in some text:

$$\begin{array}{ccccccc} s_0 & \ldots & & & c & \ldots & s_{n-1} \\ & & \mathsf{B} & \mathsf{A} & \mathsf{R} & \mathsf{B} & \mathsf{E} & \mathsf{R} \end{array}$$

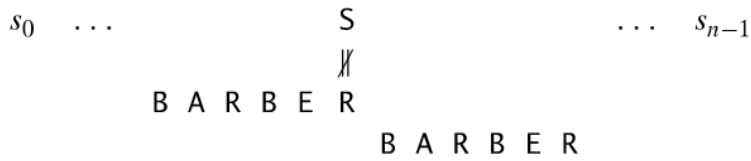
Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found. Then the search can be either stopped altogether or continued if another occurrence of the same pattern is desired.

If a mismatch occurs, we need to shift the pattern to the right. Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text. Horspool's algorithm determines the size

of such a shift by looking at the character c of the text that is aligned against the last character of the pattern. This is the case even if character c itself matches its counterpart in the pattern.

In general, the following four possibilities can occur.

Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):



Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:

Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m:

Case 4 Finally, if c happens to be the last character in the pattern and there are other c's among its first $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first $m - 1$ characters in the pattern should be aligned with the text's c:

These examples clearly demonstrate that right-to-left character comparisons can lead to farther shifts of the pattern than the shifts by only one position

always made by the brute-force algorithm. However, if such an algorithm had to check all the characters of the pattern on every trial, it would lose much of this superiority. Fortunately, the idea of input enhancement makes repetitive comparisons unnecessary. We can precompute shift sizes and store them in a table. The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters. (Note that no other information about the text in which eventual searching will be done is required.) The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad \textbf{(7.1)}$$

For example, for the pattern BARBER, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

Here is a simple algorithm for computing the shift table entries. Initialize all the entries to the pattern's length m and scan the pattern left to right repeating the following step $m-1$ times: for the jth character of the pattern ($0 \le j \le m-2$), overwrite its entry in the table with $m-1-j$, which is the character's distance to the last character of the pattern. Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence—exactly as we would like it to be.

ALGORITHM *ShiftTable*($P[0..m-1]$)

```
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and
//        filled with shift sizes computed by formula (7.1)
for i ← 0 to size − 1 do Table[i] ← m
for j ← 0 to m − 2 do Table[P[j]] ← m − 1 − j
return Table
```

Now, we can summarize the algorithm as follows:

Horspool's algorithm

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then

stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

Here is pseudocode of Horspool's algorithm.

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//        or −1 if there are no matches
ShiftTable(P[0..m − 1])      //generate Table of shifts
i ← m − 1                    //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                    //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R               B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                 B A R B E R
```

■

A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in $O(nm)$ (Problem 4 in this section's exercises). But for random texts, it is in $\Theta(n)$, and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm. In fact, as mentioned, it is often at least as efficient as its more sophisticated predecessor discovered by R. Boyer and J. Moore.

Boyer-Moore Algorithm

Now we outline the Boyer-Moore algorithm itself. If the first comparison of the rightmost character in the pattern with the corresponding character c in the text fails, the algorithm does exactly the same thing as Horspool's algorithm. Namely, it shifts the pattern to the right by the number of characters retrieved from the table precomputed as explained earlier.

The two algorithms act differently, however, after some positive number k $(0 < k < m)$ of the pattern's characters are matched successfully before a mismatch is encountered:

$$\begin{array}{ccccccccccc} s_0 & \dots & & c & & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\ & & & \not\| & & \| & & \| & & & \\ & p_0 & \dots & p_{m-k-1} & & p_{m-k} & \dots & p_{m-1} & & & \text{pattern} \end{array}$$

In this situation, the Boyer-Moore algorithm determines the shift size by considering two quantities. The first one is guided by the text's character c that caused a mismatch with its counterpart in the pattern. Accordingly, it is called the ***bad-symbol shift***. The reasoning behind this shift is the reasoning we used in Horspool's algorithm. If c is not in the pattern, we shift the pattern to just pass this c in the text. Conveniently, the size of this shift can be computed by the formula $t_1(c) - k$ where $t_1(c)$ is the entry in the precomputed table used by Horspool's algorithm (see above) and k is the number of matched characters:

$$\begin{array}{ccccccccccc} s_0 & \dots & & c & & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\ & & & \not\| & & \| & & \| & & & \\ & p_0 & \dots & p_{m-k-1} & & p_{m-k} & \dots & p_{m-1} & & & \text{pattern} \\ & & & & & p_0 & \dots & & p_{m-1} & & \end{array}$$

For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(\mathsf{S}) - 2 = 6 - 2 = 4$ positions:

$$\begin{array}{ccccccccccc} s_0 & \dots & & \mathsf{S} & \mathsf{E} & \mathsf{R} & & & \dots & s_{n-1} \\ & & & \not\| & \| & \| & & & & \\ & & \mathsf{B} & \mathsf{A} & \mathsf{R} & \mathsf{B} & \mathsf{E} & \mathsf{R} & & \\ & & & & & \mathsf{B} & \mathsf{A} & \mathsf{R} & \mathsf{B} & \mathsf{E} & \mathsf{R} \end{array}$$

The same formula can also be used when the mismatching character c of the text occurs in the pattern, provided $t_1(c) - k > 0$. For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter A, we can shift the pattern by $t_1(\mathsf{A}) - 2 = 4 - 2 = 2$ positions:

$$\begin{array}{ccccccccccc} s_0 & \dots & & \mathsf{A} & \mathsf{E} & \mathsf{R} & & & \dots & s_{n-1} \\ & & & \not\| & \| & \| & & & & \\ & & \mathsf{B} & \mathsf{A} & \mathsf{R} & \mathsf{B} & \mathsf{E} & \mathsf{R} & & \\ & & & & \mathsf{B} & \mathsf{A} & \mathsf{R} & \mathsf{B} & \mathsf{E} & \mathsf{R} \end{array}$$

If $t_1(c) - k \leq 0$, we obviously do not want to shift the pattern by 0 or a negative number of positions. Rather, we can fall back on the brute-force thinking and simply shift the pattern by one position to the right.

To summarize, the bad-symbol shift d_1 is computed by the Boyer-Moore algorithm either as $t_1(c) - k$ if this quantity is positive and as 1 if it is negative or zero. This can be expressed by the following compact formula:

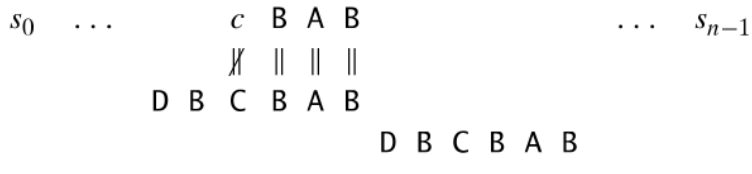
$$d_1 = \max\{t_1(c) - k, 1\}. \quad \textbf{(7.2)}$$

The second type of shift is guided by a successful match of the last $k > 0$ characters of the pattern. We refer to the ending portion of the pattern as its suffix of size k and denote it $suff(k)$. Accordingly, we call this type of shift the ***good-suffix shift***. We now apply the reasoning that guided us in filling the bad-symbol shift table, which was based on a single alphabet character c, to the pattern's suffixes of sizes $1, \ldots, m-1$ to fill in the good-suffix shift table.

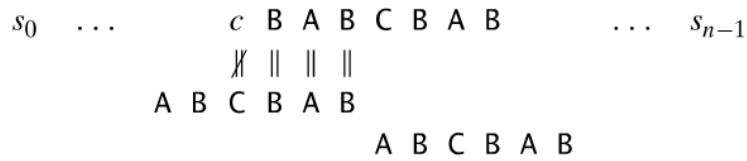
Let us first consider the case when there is another occurrence of $suff(k)$ in the pattern or, to be more accurate, there is another occurrence of $suff(k)$ not preceded by the same character as in its rightmost occurrence. (It would be useless to shift the pattern to match another occurrence of $suff(k)$ preceded by the same character because this would simply repeat a failed trial.) In this case, we can shift the pattern by the distance d_2 between such a second rightmost occurrence (not preceded by the same character as in the rightmost occurrence) of $suff(k)$ and its rightmost occurrence. For example, for the pattern ABCBAB, these distances for $k = 1$ and 2 will be 2 and 4, respectively:

k	**pattern**	d_2
1	ABC$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{AB}}$CB$\underline{\text{AB}}$	4

What is to be done if there is no other occurrence of $suff(k)$ not preceded by the same character as in its rightmost occurrence? In most cases, we can shift the pattern by its entire length m. For example, for the pattern DBCBAB and $k = 3$, we can shift the pattern by its entire length of 6 characters:



Unfortunately, shifting the pattern by its entire length when there is no other occurrence of $suff(k)$ not preceded by the same character as in its rightmost occurrence is not always correct. For example, for the pattern ABCBAB and $k = 3$, shifting by 6 could miss a matching substring that starts with the text's AB aligned with the last two characters of the pattern:



Note that the shift by 6 is correct for the pattern DBCBAB but not for ABCBAB, because the latter pattern has the same substring AB as its prefix (beginning part of the pattern) and as its suffix (ending part of the pattern). To avoid such an erroneous shift based on a suffix of size k, for which there is no other occurrence in the pattern not preceded by the same character as in its rightmost occurrence, we need to find the longest prefix of size $l < k$ that matches the suffix of the same size l. If such a prefix exists, the shift size d_2 is computed as the distance between this prefix and the corresponding suffix; otherwise, d_2 is set to the pattern's length m. As an example, here is the complete list of the d_2 values—the good-suffix table of the Boyer-Moore algorithm—for the pattern ABCBAB:

k	pattern	d_2
1	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4
4	ABCBAB	4
5	ABCBAB	4

Now we are prepared to summarize the Boyer-Moore algorithm in its entirety.

The Boyer-Moore algorithm

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the

number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases} \tag{7.3}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

Shifting by the maximum of the two available shifts when $k > 0$ is quite logical. The two shifts are based on the observations—the first one about a text's mismatched character, and the second one about a matched group of the pattern's rightmost characters—that imply that shifting by less than d_1 and d_2 characters, respectively, cannot lead to aligning the pattern with a matching substring in the text. Since we are interested in shifting the pattern as far as possible without missing a possible matching substring, we take the maximum of these two numbers.

EXAMPLE As a complete example, let us consider searching for the pattern BAOBAB in a text made of English letters and spaces. The bad-symbol table looks as follows:

c	A	B	C	D	. . .	O	. . .	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

The good-suffix table is filled as follows:

k	**pattern**	d_2
1	BAO$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{B}}$AOB$\underline{\text{AB}}$	5
3	$\overline{\text{B}}$AO$\underline{\text{BAB}}$	5
4	$\overline{\text{B}}$A$\underline{\text{OBAB}}$	5
5	$\overline{\text{B}}\underline{\text{AOBAB}}$	5

The actual search for this pattern in the text given in Figure 7.3 proceeds as follows. After the last B of the pattern fails to match its counterpart K in the text, the algorithm retrieves $t_1(\text{K}) = 6$ from the bad-symbol table and shifts the pattern by $d_1 = \max\{t_1(\text{K}) - 0, 1\} = 6$ positions to the right. The new try successfully matches two pairs of characters. After the failure of the third comparison on the space character in the text, the algorithm retrieves $t_1(\_) = 6$ from the bad-symbol table and $d_2 = 5$ from the good-suffix table to shift the pattern by $\max\{d_1, d_2\} = \max\{6 - 2, 5\} = 5$. Note that on this iteration it is the good-suffix rule that leads to a farther shift of the pattern.

The next try successfully matches just one pair of B's. After the failure of the next comparison on the space character in the text, the algorithm retrieves $t_1(\_) = 6$ from the bad-symbol table and $d_2 = 2$ from the good-suffix table to shift

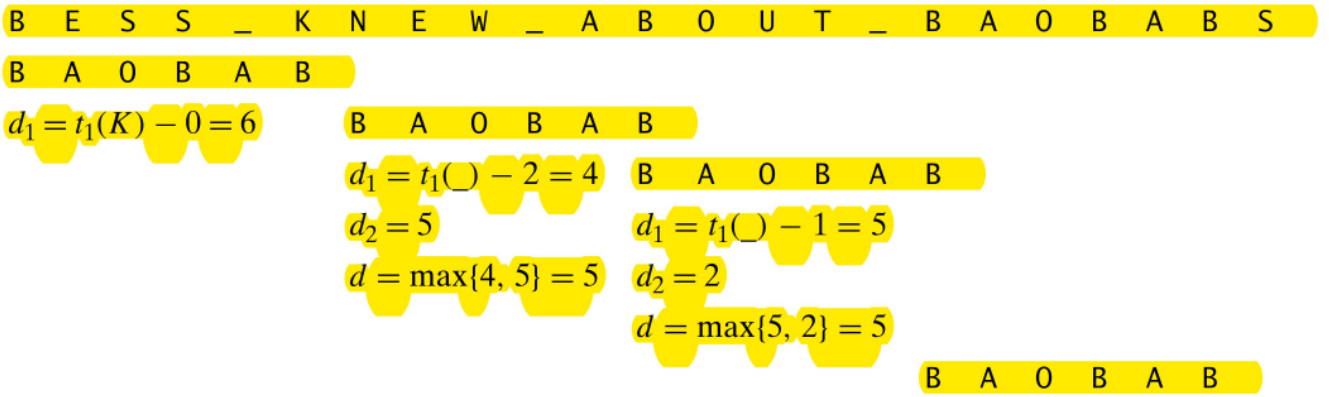


FIGURE 7.3 Example of string matching with the Boyer-Moore algorithm.

the pattern by $\max\{d_1, d_2\} = \max\{6 - 1, 2\} = 5$. Note that on this iteration it is the bad-symbol rule that leads to a farther shift of the pattern. The next try finds a matching substring in the text after successfully matching all six characters of the pattern with their counterparts in the text. ■

When searching for the first occurrence of the pattern, the worst-case efficiency of the Boyer-Moore algorithm is known to be linear. Though this algorithm runs very fast, especially on large alphabets (relative to the length of the pattern), many people prefer its simplified versions, such as Horspool's algorithm, when dealing with natural-language–like strings.

Exercises 7.2

1. Apply Horspool's algorithm to search for the pattern BAOBAB in the text

BESS_KNEW_ABOUT_BAOBABS

2. Consider the problem of searching for genes in DNA sequences using Horspool's algorithm. A DNA sequence is represented by a text on the alphabet {A, C, G, T}, and the gene or gene segment is the pattern.

a. Construct the shift table for the following gene segment of your chromosome 10:

TCCTATTCTT

b. Apply Horspool's algorithm to locate the above pattern in the following DNA sequence:

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT

3. How many character comparisons will be made by Horspool's algorithm in searching for each of the following patterns in the binary text of 1000 zeros?

a. 00001 **b.** 10000 **c.** 01010

4. For searching in a text of length n for a pattern of length m $(n \geq m)$ with Horspool's algorithm, give an example of

a. worst-case input. **b.** best-case input.

5. Is it possible for Horspool's algorithm to make more character comparisons than the brute-force algorithm would make in searching for the same pattern in the same text?

6. If Horspool's algorithm discovers a matching substring, how large a shift should it make to search for a next possible match?

7. How many character comparisons will the Boyer-Moore algorithm make in searching for each of the following patterns in the binary text of 1000 zeros?

a. 00001 **b.** 10000 **c.** 01010

8. a. Would the Boyer-Moore algorithm work correctly with just the bad-symbol table to guide pattern shifts?

b. Would the Boyer-Moore algorithm work correctly with just the good-suffix table to guide pattern shifts?

9. a. If the last characters of a pattern and its counterpart in the text do match, does Horspool's algorithm have to check other characters right to left, or can it check them left to right too?

b. Answer the same question for the Boyer-Moore algorithm.

10. Implement Horspool's algorithm, the Boyer-Moore algorithm, and the brute-force algorithm of Section 3.2 in the language of your choice and run an experiment to compare their efficiencies for matching

a. random binary patterns in random binary texts.

b. random natural-language patterns in natural-language texts.

11. You are given two strings S and T, each n characters long. You have to establish whether one of them is a right cyclic shift of the other. For example, PLEA is a right cyclic shift of LEAP, and vice versa. (Formally, T is a right cyclic shift of S if T can be obtained by concatenating the $(n - i)$-character suffix of S and the i-character prefix of S for some $1 \leq i \leq n$.)

a. Design a space-efficient algorithm for the task. Indicate the space and time efficiencies of your algorithm.

b. Design a time-efficient algorithm for the task. Indicate the time and space efficiencies of your algorithm.