CHAPTER 9
Text Searching

Searching text is a ubiquitous problem, present wherever information is retrieved. Its applications range from using search engines on the web to locating words in files on your computer and to biologists searching sequence databases. The general form of the problem is to find within a text t a match for a pattern p, where *text*, *pattern*, and *match* can be interpreted in different ways. In the simplest case the text is a document in memory (in your word processor, for example), and the pattern is a word you want to locate within that document. This is the *classical version* of the text-searching problem, which can be solved by brute force sequential searching in time $O(|p| * |t|)$, where $|s|$ is the length of a string s. We discuss this algorithm in Section 9.1. In Section 9.2 we present the solution by Rabin-Karp, which runs in average time $O(|p| + |t|)$. For this algorithm, Rabin and Karp used the powerful notion of hashing (fingerprinting).

The classical problem can be solved in time $O(|p| + |t|)$. The two most famous algorithms achieving this time bound are Knuth-Morris-Pratt and Boyer-Moore. We explain Knuth-Morris-Pratt in detail in Section 9.3 and sketch the basic ideas behind Boyer-Moore in Section 9.4. Implementing Boyer-Moore is complicated, so instead we present an algorithm called Boyer-Moore-Horspool whose worst-case running time is $O(|p| * |t|)$, but which performs very well in practice and is easy to implement.

Stepping back from the classical problem, we find that there are multiple possible relaxations. For example, we can allow approximate matches. It is not immediately obvious what an approximate match should be, and we investigate that question in Section 9.5 discussing two very different approaches: edit distance and the don't-care wildcard.

Finally, in some situations we want to find a match of a particular form, for example when scanning a document for phone numbers or when verifying credit card information submitted by a web-form. These problems can be solved using regular expressions, an elegant technique that grew out of formal language theory (Section 9.6).

Before we start, we need to fix some terminology for the rest of this chapter. We usually denote the pattern to be searched for as p, and the text being searched as t. We also usually think of the underlying alphabet Σ as fixed.

You can think of it as containing all of the letters and punctuation symbols of the English language. For the Rabin-Karp algorithm we restrict ourselves to the binary alphabet $\Sigma = \{0, 1\}$. The set of all (finite) words on an alphabet Σ is denoted Σ^*, so $\{0, 1\}^*$ is the set of all binary strings.

The length of a string s is denoted by $|s|$. Within a program we write *s.length* for $|s|$. We usually let $m = |p|$ and $n = |t|$ and assume that $m \leq n$. To simplify talking about strings, we write $p[i..j]$ for the substring of p starting at position i and ending at position j. Hence, $p = p[0..m-1]$ (strings begin at position 0), and $p = [0..i]$ would be an arbitrary prefix of p. If $i > j$, we define $p[i..j]$ to be the empty string ε (the unique string of length 0). We also write $p[i]$ for $p[i..i]$, so $p[0]$ is the first and $p[m-1]$ is the last letter of p. We write st for the concatenation of two strings s and t (programming languages usually use $s + t$ to denote the concatenation of the two strings).

9.1 Simple Text Search

Imagine a word-processor searching for a word (or phrase) in a document. The easiest approach to locating a word in a text is to compare the word letter by letter against the text, testing all possible locations in the text until a match is found.

Algorithm 9.1.1 Simple Text Search. This algorithm searches for an occurrence of a pattern p in a text t. It returns the smallest index i such that $t[i..i+m-1] = p$, or -1 if no such index exists.

Input Parameters: p, t
Output Parameters: None

```
simple_text_search(p, t) {
  m = p.length
  n = t.length
  i = 0
  while (i + m ≤ n) {
    j = 0
    while (t[i + j] == p[j]) {
      j = j + 1
      if (j ≥ m)
        return i
    }
    i = i + 1
  }
  return -1
}
```

Example 9.1.2. Let us have a look at a run of *simple_text_search* searching for the pattern "001" in the text "010001". Figure 9.1.1 shows the progression of the algorithm.

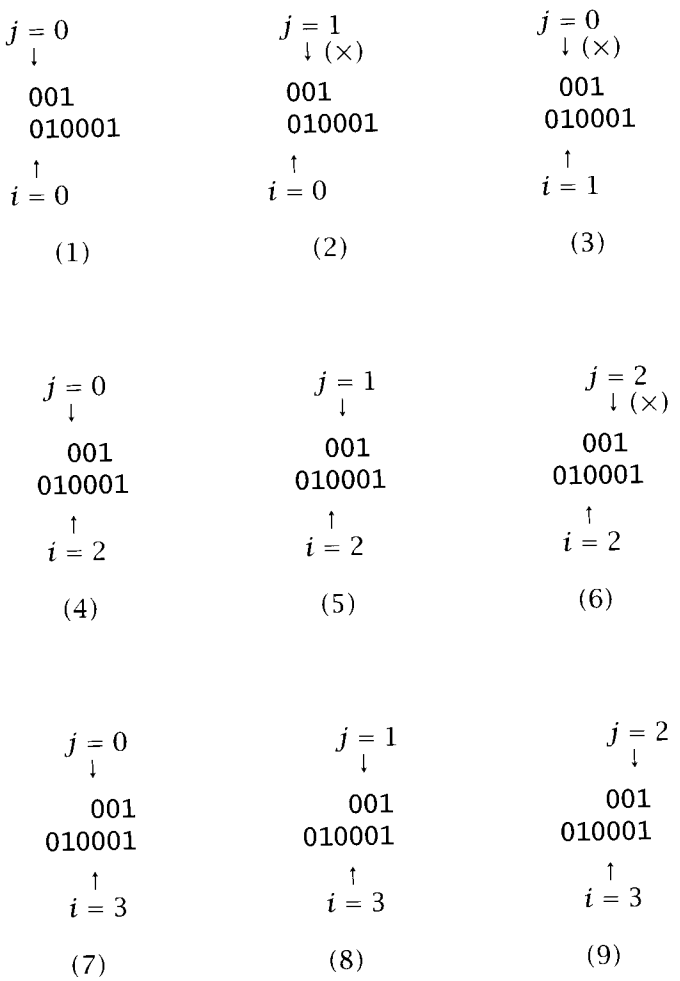

Figure 9.1.1 Searching for "001" in "010001" using *simple_text_search*. The cross (×) in steps (2), (3), and (6) marks a mismatch. □

To convince ourselves that the algorithm works correctly, we have to verify only that the pattern is checked against all possible positions. Since the pattern contains $m = |p|$ characters, the last possible match for p in $t = t[0..n-1]$ would be $t[n-m..n-1]$, and, indeed, the outer while loop allows indexes i up to and including $n - m$.

In the worst case there is no match for the pattern, and the algorithm can take time $\Theta(m(n-m+1))$ (Exercise 4 asks you to prove the lower bound).

The best case occurs if the pattern is found at the start of the text taking time $\Theta(m)$.

Theorem 9.1.3. *The algorithm simple_text_search solves the pattern-matching problem correctly in time* $O(m(n - m + 1))$.

In practice *simple_text_search* behaves better than $O(m(n - m + 1))$ because the inner while loop usually recognizes failure quite quickly. On random patterns and text, the running time of *simple_text_search* is $O(n - m)$ (see Exercise 9.1).

Exercises

1S. How many comparisons does *simple_text_search* perform when searching for the pattern "est" in the text "test"?

2. How many comparisons does *simple_text_search* perform when searching for the pattern "ita" in the text "itititit"?

3. Trace the algorithm *simple_text_search* on pattern "lai" and text "balalaika".

4S. Show that *simple_text_search* has a worst-case running time of $\Theta(m(n - m + 1))$. We proved only the upper bound $O(m(n - m + 1))$; the lower bound $\Omega(m(n - m + 1))$ still has to be proved.

5. Show that $m(n - m + 1)$ is $O(mn)$.

6. Show that $m(n - m + 1)$ is not $\Omega(mn)$. Together with Exercise 5 this implies that while $O(mn)$ is an upper bound on the running time of *simple_text_search*, it is not a *tight* upper bound.

7. Show that $m(n - m + 1)$ is not $O(m(n - m))$ even if we assume $m \leq n$.

Exercises 8-10 investigate a variant of the simple text search algorithm that finds all occurrences of a pattern.

8S. Modify *simple_text_search* so that it prints out a list of locations of all matches for a pattern in a text.

9. Give an upper bound on the worst-case running time of your algorithm that finds all occurrences of a pattern.

10. Give a lower bound on the worst-case running time of your algorithm that finds all occurrences of a pattern.

Exercises 11-13 are about string matching with the don't-care *symbol "?" (sometimes called a* wildcard*). We assume that "?" is a special character that does not belong to the alphabet from which the text is made up. The*

don't-care symbol matches any letter in the text. For example, the first match for "?01?" in "1001110" is "0011" starting in the second position of the text. Note that "?" can match different characters in the same match.

11S. Using the don't-care symbol "?", write a pattern that matches an American social security number.

12. Modify *simple_text_search* so that it returns the first match of a pattern possibly containing the don't-care symbol "?".

13. Extend the algorithm from Exercise 12 to also return the actual match that was found for the pattern.

9.2 The Rabin-Karp Algorithm

The simple text search we implemented in Section 9.1 can be ineffective for longer patterns, in particular if pattern and text contain repeated elements (see Exercise 4 in Section 9.1). The running time of the simple text search can be as bad as $\Theta(n^2)$ (for $m = n/2$, for example). Our ultimate goal is an algorithm running in linear time $O(n + m)$ in the worst case, a goal we will attain in the next section. Here we present an algorithm by Rabin and Karp that takes time $\Theta(n + m)$ on average.

The central idea of Rabin and Karp's algorithm is that, before we spend time checking for the match of a long pattern in a particular position (which might take m steps), it might pay to do some preliminary checking on whether to expect a matching pattern at that position. Suppose that a preliminary check could be implemented cheaply in time $O(1)$, say, and we could expect it to eliminate all but $1/m$th of the locations, that is $(n - m + 1)/m$ many. For these we would have to run the brute-force comparison, which takes time $O(m)$ for each location or time $O(m(n - m + 1)/m)$ altogether. Since the implementation requires $O(m)$ time for preprocessing, we get an overall average running time of $O(n + m)$.

How can we quickly exclude indexes from consideration? Suppose, for example, that we are searching for the pattern "000011" in

"0000010000100000".

The pattern has parity 0 (the number of ones in the string is even). Now consider the text we are searching: All substrings of length 6 have parity 1 (they contain an odd number of ones) with the exception of 100001. Out of $11 = 16 - 6 + 1$ positions in the text, we can rule out all but one, simply because they have the wrong parity. Rabin and Karp called this method *fingerprinting* because, instead of comparing the full pattern, we only compare a small aspect of it, its fingerprint.

Example 9.2.1. Let us see how the parity fingerprint helps in searching for p = "010111" in t = "00101101010010010011". The pattern p has parity 0,

since it contains an even number of ones. We let $f[i]$ be the parity of the string $t[i..i+5]$. For example, $f[0] = 1$, since "001011" contains an odd number of ones. The following table lists all values of $f[i]$:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$t[i]$	0	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	0	1	1
$f[i]$	1	1	0	1	0	1	0	1	0	1	0	0	1	1					

The table tells us that the only positions in t we need to check are 2, 3, 4, 6, 8, 10, and 11, since all of the others have the wrong parity. Since p is different from all of the substrings found at these positions, we conclude that there is no match for p in t. □

We could extend our simple text search algorithm to first perform a parity check and then skip positions that have the wrong parity. A straightforward implementation will not lead to an improvement, because computing the parity of m bits takes m steps, and we need the check to be in constant time. However, we can do better. When we start the algorithm, we compute the parity of the first m bits of t [taking time $O(m)$ for preprocessing]. In Example 9.2.1 this gives us a parity of 1 for the six bits starting at index 0. When, in the next step, we need the parity of the m bits starting at index 1, we need to look only at the bits in positions 0 and m, because these are the only bits that change. In Example 9.2.1, both $t[0]$ and $t[6]$ are zero; hence the parity of $t[1..6]$ is the same as the parity of $t[0..5]$, namely 0. Using this trick we can update the parity in only two steps. In general, if the parity of the m bits starting at position i of the text is x, then the parity of the m bits starting at position $i+1$ is $x + t[i] + t[i+m] \bmod 2$.

Example 9.2.2. Consider again the string t in Example 9.2.1. Let us verify that $f[i+1] = f[i] + t[i] + t[i+6] \bmod 2$ for some sample values. By inspection (precomputation) we see that $f[0] = 1$. Now $f[1] = f[0] + t[0] + t[6] \bmod 2 = 1 + 0 + 0 \bmod 2 = 1$, and $f[2] = f[1] + t[1] + t[7] \bmod 2 = 1 + 0 + 1 \bmod 2 = 0$. And $f[13] = f[12] + t[12] + t[18] \bmod 2 = 1 + 1 + 1 \bmod 2 = 1$. □

On average we expect the parity check to reject half the inputs, so that the loop would take about $m(n-m+1)/2$ steps. Unfortunately, this is still $\Theta(m(n-m+1))$. While the basic idea is sound, parity is not good enough to give us the required speed-up because too many values share the same fingerprint. If we want to obtain a speed-up by a factor of q, we need a fingerprint function that

(a) maps m-bit strings to q different values (fingerprints),

(b) distributes the m-bit strings evenly across the q values, and

(c) is easy to compute sequentially, in the sense that if we change the string at the beginning and the end only, we can quickly (in time $O(1)$) compute the new value of the function.

A function fulfilling properties (a) and (b) is called a *hash function.* It distributes a large number of values evenly among a small number of fingerprints. Hash functions have found numerous applications in practice, for example in implementing hash tables for storing data or, more recently, to compute the message digests for digital signatures.

Parity fulfills all three properties for $q = 2$, which is why it gave us a speed-up by a factor of 2. Properties (a) and (b) are needed because they imply that on average $(q-1)/q$ of the inputs get rejected, and we need to run the brute-force algorithm on only $1/q$ of the inputs. Property (c) guarantees that we can find out whether an input can be eliminated in constant time. Eliminating positions therefore takes time $O((n-m+1)(q-1)/q)$, which is $O(n)$. This leaves us with approximately $(n-m+1)/q$ positions for which we need to compare the pattern to the text letter by letter. This might take m steps for each position, leading to a running time of $O(m(n-m+1)/q)$ on the positions that are not eliminated. If we choose q larger than m, this is $O(n)$. Since the algorithm needs time $O(m)$ for preprocessing, it takes time $O(n+m)$ on average.

How can we find a function with properties (a), (b), and (c) for $q > m$? Going back to the parity example, the first idea that comes to mind would be to simply take the sum of the m bits. This function fulfills (a) and (c). Unfortunately, it violates (b) badly (for example, there is only one input each that takes on values 0 or m).

Rabin and Karp suggested the following hash function: Consider the m bits as the binary expansion of a natural number and take the remainder after division by q. More precisely, if the m bits are $s_0 s_1 \ldots s_{m-1}$, use the value

$$\sum_{j=0}^{m-1} s_j 2^{m-1-j} \bmod q.$$

Example 9.2.3. Let us return to the text t we saw in Example 9.2.1, but instead of fingerprinting it with the parity function we use

$$f[i] = \sum_{j=0}^{5} t[i+j]2^{5-j} \bmod 7.$$

The tables display the results. The third row shows the value $f[i]$.

i	0	1	2	3	4	5	6	7	8	9
$t[i]$	0	0	1	0	1	1	0	1	0	1
Σ	11	22	45	26	53	42	20	41	18	37
$f[i]$	4	1	3	5	4	0	6	6	4	2

i	10	11	12	13	14	15	16	17	18
$t[i]$	0	0	1	0	1	0	0	1	1
Σ	10	20	41	19					
$f[i]$	3	6	6	5					

Let us verify some entries in the table. For example, $t[0..5]$ ="001011" corresponds to the number $11 = 1 * 8 + 1 * 2 + 1 * 1$, and therefore $f[0] = 11 \bmod 7 = 4$. Similarly, $t[5..10]$ ="101010" corresponds to the number $32 + 8 + 2 = 42$, and therefore $f[5] = 42 \bmod 7 = 0$. □

The function $\sum_{j=0}^{m-1} s_j 2^{m-1-j} \bmod q$ fulfills property (a) because the remainder is always in the range $\{0,\ldots,q-1\}$. Property (b) depends on the choice of q. For random texts and patterns, the value of q does not matter; but in practice values of q need to be avoided. Suppose, for example, you knew that your texts are binary and are made up of roughly 30% ones and 70% zeros, and you are using $q = 2$, and $m = 2$. In that case, we expect to see mostly substrings of the form 00 and a much smaller number of substrings 01, 10, and 11, implying that there is an imbalance between strings having fingerprints 0 and strings having fingerprint 1. In practice, it has turned out that choosing a prime number $q > m$ works very well and fulfills (b). We are left with verifying property (c), which we need for implementing the algorithm. Consider $m+1$ bits $s_0 s_1 \ldots s_{m-1} s_m$. The first m bits, $s_0 s_1 \ldots s_{m-1}$, get assigned value $a = \sum_{j=0}^{m-1} s_j 2^{m-1-j} \bmod q$. The m bits starting in the next position $s_1 \ldots s_m$ in turn get assigned

$$\begin{aligned}
\sum_{j=0}^{m-1} s_{j+1} 2^{m-1-j} \bmod q &= s_m + \sum_{j=0}^{m-2} s_{j+1} 2^{m-1-j} \bmod q \\
&= s_m + 2 \sum_{j=0}^{m-2} s_{j+1} 2^{m-1-(j+1)} \bmod q \\
&= s_m + 2 \sum_{j=1}^{m-1} s_j 2^{m-1-j} \bmod q \\
&= s_m + 2(-2^{m-1} s_0 + \sum_{j=0}^{m-1} s_j 2^{m-1-j}) \bmod q \\
&= s_m + 2(a - 2^{m-1} s_0) \bmod q.
\end{aligned}$$

This formula allows us to compute the next value from the previous in constant time. It pays to precompute $2^{m-1} \bmod q$, since it is used in every step.

Example 9.2.4. Returning to Example 9.2.3, we now see that there was no need to explicitly compute the values $\sum_{j=i}^{i+5} s_j 2^{5-1-j}$. Our new formula now gives us

$$f[i+1] = t[i+6] + 2(f[i] - 32t[i]) \bmod 7 = t[i+6] + 2f[i] - t[i] \bmod 7.$$

For example, $f[1] = t[6] + 2f[0] - t[0] \bmod 7 = 0 + 8 - 0 \bmod 7 = 1$, $f[8] = t[13] + 2f[7] - t[7] \bmod 7 = 0 + 12 - 1 \bmod 7 = 4$, and $f[10] = t[15] + 2f[9] - t[9] \bmod 7 = 0 + 4 - 1 \bmod 7 = 3$. □

Algorithm 9.2.5 Rabin-Karp Search. This algorithm searches for an occurrence of a pattern p in a text t. It returns the smallest index i such that $t[i..i+m-1] = p$ or -1 if no such index exists.

```
  Input Parameters:  p, t
Output Parameters:  None

rabin_karp_search(p,t) {
  m = p.length
  n = t.length
  q = prime number larger than m
  r = 2^(m-1) mod q
  // computation of initial remainders
  f[0] = 0
  pfinger = 0
  for j = 0 to m - 1 {
    f[0] = 2 * f[0] + t[j] mod q
    pfinger = 2 * pfinger + p[j] mod q
  }
  i = 0
  while (i + m ≤ n) {
    if (f[i] == pfinger)
      if (t[i..i + m - 1] == p) // this comparison takes time O(m)
        return i
    f[i + 1] = 2 * (f[i] - r * t[i]) + t[i + m] mod q
    i = i + 1
  }
  return -1
}
```

Example 9.2.6. Let us look again at p ="010101" in

$$t = \text{"0010110101001010011"}.$$

Running *rabin_karp_search* for $q = 7$ gives us the table in Example 9.2.3. For example, the algorithm starts with $f[0] = 4$. In the next step it computes

$$f[i+1] = 2 * (f[i] - r * t[i]) + t[i+m] \bmod q$$

with $r = 2^{m-1} \bmod q = 4$; hence we obtain $f[1] = 2(4 - 4t[0]) + t[6] \bmod 7 = 2(4-0)+0 \bmod 7 = 1$. In the third step, we get $f[2] = 2(f[1]-4t[1]) + t[7] \bmod 7 = 3$, in the fourth step $f[3] = 2(f[2] - 4t[2]) + t[8] \bmod 7 = 5$, and so on.

The value of *pfinger* is $16 + 4 + 1 \bmod 7 = 0$; hence the only position we need to check in t is $i = 5$. Since $t[5 + 10]$ ="101010"≠"010101"$= p$, we know that p does not occur in t.

Let us try the same text t with another pattern p ="001010". In this case, *pfinger* is 3; hence we have to check only two positions, 2 and 10, and indeed $t[10..15] = p$. □

Before we prove correctness, let us analyze the running time of *rabin_karp_search*. In the worst case, $f[i]$ equals *pfinger* for each location i and the main loop takes as much time as the simple text search, namely $O(m(n-m+1))$. Together with the additional $O(m)$ time to compute the initial values of $f[0]$ and *pfinger*, we have a running time of $O(m(n-m+2))$, which is $O(m(n-m+1))$. On average, however, we expect the test $(f[i] == pfinger)$ to succeed only about $1/q$th of the time because we assumed that our fingerprinting function distributes inputs evenly among the q possible output values. (We did not prove this, and a technical proof is beyond the scope of this book.) Hence, we expect to run the inner loop $(n-m+1)/q$ times for $O(m)$ steps each. Since q was chosen larger than m, the expected running time for the loops is $O(n-m+1)$. With the $O(m)$ preprocessing, this yields an expected running time of $O(n+m)$.

Theorem 9.2.7. *The algorithm rabin_karp_search correctly finds the first match for a pattern in a text (if it exists) in time* $O(m(n-m+1))$.

Proof. We already did the running time analysis. We prove correctness by using the loop invariant

$$f[i] = \sum_{j=0}^{m-1} t_{i+j} 2^{m-1-j} \bmod q$$

for the while loop (see Section 2.2). We claim that the loop invariant is true every time the loop condition $(i + m \leq n)$ is tested. The loop invariant states that $f[i]$ contains the correct fingerprint of $t[i..i+m-1]$ during the ith iteration of the loop. Before we show that this is indeed correct, we show how to use the loop invariant to prove that the algorithm is correct.

In each iteration of the while loop, the algorithm checks whether $f[i]$ equals *pfinger*. If they are equal, it checks whether $t[i..i+m-1] == p$; otherwise it continues to the next i. Since we assume that the loop invariant is true, we know that $f[i]$ is the correct fingerprint of $t[i..i+m-1]$. If $f[i]$ is different from *pfinger*, we know that there cannot be a match of p starting at position i, since $p = t[i..i+m-1]$ implies that

$$\begin{aligned} f[i] &= \sum_{j=0}^{m-1} t_{i+j} 2^{m-1-j} \bmod q && \text{by the loop invariant} \\ &= \sum_{j=0}^{m-1} p_j 2^{m-1-j} \bmod q && \text{because } p = t[i..i+m-1] \\ &= pfinger && \text{definition of } pfinger. \end{aligned}$$

Therefore, the algorithm behaves correctly by not comparing p to $t[i..i+m-1]$ if $f[i]$ is different from *pfinger*. If, on the other hand, $f[i]$ and *pfinger* agree, the algorithm checks whether $t[i..i+m-1]$ and p are the same, comparing letter by letter. If they are equal, it returns i; otherwise it continues with the next i. We conclude that *rabin_karp_search* does not

miss a match for p in t if there is one. With this knowledge we can complete the argument that the algorithm works correctly. There are two ways of leaving the algorithm: because the loop condition $i + m \leq n$ fails or because a match $p = t[i..i + m - 1]$ is found. Since we argued that the algorithm did not miss any previous matches, we know that in the first case there is no match for p, and in the second case, the algorithm returns the first match for p.

We still need to verify that the loop invariant is true. Initially, the first time the loop condition is checked, $f[0]$ contains the correct value:

$$\sum_{j=0}^{m-1} t_j 2^{m-1-j} \bmod q,$$

because that is how we initialize $f[0]$. We earlier derived the formula

$$f[i + 1] = 2 * (f[i] - 2^{m-1} * t[i]) + t[i + m] \bmod q.$$

Since we use this formula to compute $f[i + 1]$ in the algorithm (with the difference that we have precomputed the value r of $2^{m-1} \bmod q$), $f[i + 1]$ is computed correctly for the next iteration of the loop, and we have verified that the loop invariant is correct. ■

We have ignored the problem of finding a prime number larger than q. Prime numbers are plentiful. Bertrand's postulate, for example, states that for any number m there is a prime number between m and $2m$, and better results are known. Hence we could find a prime larger than m in time $O(m^{3/2})$ (verifying primality by testing all possible factors). In practice, it is much faster to use randomized techniques to generate a large prime number. We would also choose q close to and smaller than the word-size of the computer since this speeds up the arithmetical computations that are at the heart of the algorithm.

Choosing the prime number at random has other advantages than just being faster. Compare the following algorithm to Algorithm 9.2.5, *rabin_karp_search*.

Algorithm 9.2.8 Monte Carlo Rabin-Karp Search. This algorithm searches for occurrences of a pattern p in a text t. It prints out a list of indexes such that with high probability $t[i..i + m - 1] = p$ for every index i on the list.

Input Parameters: p, t
Output Parameters: None

mc_rabin_karp_search(p, t) {
  m = *p.length*
  n = *t.length*
  q = randomly chosen prime number less than mn^2
  $r = 2^{m-1} \bmod q$

```
  // computation of initial remainders
  f[0] = 0
  pfinger = 0
  for j = 0 to m − 1 {
    f[0] = 2 * f[0] + t[j] mod q
    pfinger = 2 * pfinger + p[j] mod q
  }
  i = 0
  while (i + m ≤ n) {
    if (f[i] == pfinger)
      println("Match at position " + i)
    f[i + 1] = 2 * (f[i] − r * t[i]) + t[i + m] mod q
    i = i + 1
  }
}
```

The algorithm *mc_rabin_karp_search* relies entirely on fingerprints to decide whether there is a match at position i. We removed the loop testing whether $t[i..i + m - 1] = p$ when the fingerprints of $t[i..i + m - 1]$ and p are the same. This makes the algorithm much faster; it now runs in time $O(n)$. However, it also makes the algorithm incorrect: Two strings could have the same fingerprint without being identical. Rabin and Karp observed that this is very unlikely to happen if q is a randomly chosen prime less than mn^2. Indeed, the probability of the algorithm making even a single wrong decision for a particular p and t is less than $2.53/n$, which is small when n is large. Furthermore, the algorithm only errs by listing wrong matches, also called *false positives*; the list will include all correct matches, since a position is only rejected if the fingerprints do not agree. Algorithms with these two properties are called *Monte-Carlo algorithms*.

Exercises

1S. Trace the algorithm *rabin_karp_search* by hand on pattern "111" and text "011010011001110" with $q = 2$. How many comparisons between pattern and text symbols are made?

2. Trace the algorithm *rabin_karp_search* by hand on pattern "111" and text "011010011001110" with $q = 5$. How many comparisons between pattern and text symbols are made?

3. Trace the algorithm *rabin_karp_search* by hand on pattern "101" and text "011010011001110" with $q = 7$. How many comparisons between pattern and text symbols are made?

4S. Show that the worst-case running time of *rabin_karp_search* is $\Theta(m(n - m + 1))$.

5. Implement *rabin_karp_search*. To find q, write a program searching for the smallest prime larger than m. *Hint:* By Bertrand's Postulate there is always a prime number between any m and $2m$, so the search takes time at most $O(m^{3/2})$; in practice, it is much faster.

6. Modify *rabin_karp_search* to print out a list of all matches for a pattern in a text. Your modified version should still run in average time $O(n + m)$.

9.3 The Knuth-Morris-Pratt Algorithm

In this section we present the algorithm of Knuth-Morris-Pratt that solves the pattern-matching problem in time $O(|p| + |t|)$. Let us look at an example to see why the simple text search can be inefficient. Suppose we have reached the following stage running *simple_text_search*:

```
Tweedledum
Tweedledee and Tweedledum
```

In the simple text search we would now move "Tweedledum" by one letter and start matching again. However, we see that "Tweedledum" cannot match in the second position of the text because we know that the text starts with "Tweedled" as we found out when comparing the pattern to the text. As a matter of fact, "Tweedled" does not contain any "T" after the first position; so we can shift the pattern to the first letter after "Tweedled" in "Tweedledee and Tweedledum" and continue matching.

```
        Tweedledum
Tweedledee and Tweedledum
```

Let us consider another example. We trace the simple text search with pattern p ="pappar", and text t ="pappappapparrassanuaragh"[†]

```
pappar
pappappapparrassanuaragh
```

After six comparisons, the simple text search encounters a mismatch between the final "r" of "pappar" and the "p" in the corresponding position of t. It then continues by trying to match "pappar" starting with $t[1]$ ="a" at position 1. Again, we know that there cannot be a match at this position. We know that we successfully matched "pappa" against the text; hence $t[0..4]$ = "pappa" and the next potential match for "pappar" could begin three positions over.

```
   pappar
pappa
```

[†]This word, of course, is not a complete English word. The full word is "PappappapparrassannuaragheallachnatullaghmonganmacmacmacmwhackfallTheredebblenonthedubblandaddydoodled".

Hence, we can skip positions 1 and 2 of t altogether for matching p and align $p[0]$ with $t[3]$:

```
   pappar
pappappapparrassanuaragh
```

At this point we know that $p[0..1] = t[3..4]$ (this is why we shifted to that position), and we can continue comparisons at $t[5]$, the same position at which we encountered the mismatch. We conclude that whenever we match $p[0..4]$ against a text, but $p[5]$ does not match, we can shift the pattern by three positions and continue comparing in the same position at which the mismatch was found. More generally, given a search pattern p, we can compute a table that answers the following question for each k: If $p[0..k]$ matches the text, by how many positions can we shift the pattern if $p[k+1]$ does not match the text?

Example 9.3.1. Here is the table for p ="Tweedledum". We will see how to compute it later.

	T	w	e	e	d	l	e	d	u	m
k	−1	0	1	2	3	4	5	6	7	8	9
shift	1	1	2	3	4	5	6	7	8	9	10

We notice that $shift[k] = \max\{1, k+1\}$. The reason is that if we have matched $p[0..k]$ against $t[i..i+k]$, but $p[k+1] \neq t[i+k+1]$, then there cannot be a match for p starting at positions i through $i+k$ in t. There cannot be a match at position i, because we just found out that $p[k+1] \neq t[i+k+1]$. Also, no matches are possible at $i+1$ through $i+k$, since these are the same as $p[1..k]$, and therefore do not contain a "T" with which a match for p must begin. Hence, we can shift the pattern to position $i+k+1$ and continue by comparing $p[0]$ to $t[i+k+1]$. In case $k = -1$ we know that $p[0] \neq t[i]$, so we have to move the pattern to the next position, giving us a shift of 1.

So when "Tweedledum" fails matching against "Tweedledee and Tweedledum" at $p[8]$ we look up $shift[7]$ to find that we can shift the pattern by 8 positions, so the next comparison is $p[0]$ ="T" against $t[8]$ ="e". □

Example 9.3.2. For "pappar" we can use the following table:

	p	a	p	p	a	r
k	−1	0	1	2	3	4	5
shift	1	1	2	2	3	3	6

Suppose, for example, that we are matching "pappar" against "panther". After failing to match $p[2]$ against the "n" in panther we can shift the pattern by $shift[1] = 2$ positions and continue by comparing "p" to "n" as shown:

```
  j = 2                j = 0
    ↓ (×)                ↓ (×)
  pappar               pappar
  panther            panther
   ↑                    ↑
i = 0                i = 2
```

If we are matching "pappar" against "papaya tree" we can shift the pattern by $shift[2] = 2$ positions

```
  j = 3                j = 0
    ↓ (×)                ↓
  pappar               pappar
  papaya tree        papaya tree
   ↑                    ↑
i = 0                i = 2
```

and continue by comparing $p[1]$ ="a" to $t[3]$ ="a".

As a final example, consider finding a match for "pappar" in "pappappapparrassanuaragh".

```
    j = 5
      ↓ (×)
  pappar
  pappappapparrassanuaragh
   ↑
i = 0
```

After six comparisons, we know that $p[0..4] = t[0..4]$, but $p[5] \neq t[5]$. Hence, we shift the pattern by $shift[4] = 3$ positions and continue by comparing $p[2]$ to $t[5]$.

```
    j = 2
      ↓
    pappar
  pappappapparrassanuaragh
     ↑
  i = 3
```

Four comparisons later we find another mismatch: $p[5]$ is different from $t[8]$. Again we shift the pattern by $shift[4] = 3$ positions and continue by comparing $p[2]$ to $t[8]$.

```
      j = 2
        ↓
      pappar
  pappappapparrassanuaragh
       ↑
    i = 6
```

We have found a match after four more comparisons. □

Why does the technique shown in Example 9.3.2 work? Suppose $p[0..k]$, the pattern up to position k, matches the letters in the search text starting in position i that is $p[0..k] = t[i..i+k]$. Consider shifting the pattern p by s positions. For the shift to be successful, the first $k - s + 1$ characters of p have to match in the new position:

p:		0	1	2	...	s	$s+1$	...	k	...
t:	0 1 2...	i	$i+1$	$i+2$	...	$i+s$	$i+s+1$	...	$i+k$	...
p:						0	1	2 ...	$k-s$	...

Looking at the diagram, we see that for the shift to be successful we need $p[0..k-s] = t[i+s..i+k]$. However, we assumed that $t[i+s..i+k] = p[s..k]$, so we must have $p[0..k-s] = p[s..k]$. A successful shift has to fulfill this condition; also, we have to take the first such shift since otherwise we might miss a match, and therefore

$$shift[k] = \min\{s > 0 \mid p[0..k-s] = p[s..k]\}.$$

We require $s > 0$ since we do want to force a shift. The best case occurs if there is no match at all, and $s = k + 1$ (in that case both $p[0..k-s]$ and $p[s..k]$ are empty, hence equal).

Example 9.3.3. We can now justify the table in Example 9.3.1. For

$$p = \text{“Tweedledum”},$$

we know that $p[0] \neq p[j]$ for any $1 \leq j \leq 9$, and hence $shift[k] = k + 1$ for $k \geq 0$. □

Example 9.3.4. Let us return to a situation from Example 9.3.2:

```
        j = 5
          ↓ (×)
    pappar
pappappapparrassanuaragh
   ↑
 i = 3
```

In this situation, the current starting position of p in t is $i = 3$, and we are comparing $p[j]$ with $j = 5$ to $t[i + j] = t[8]$. The comparison fails, and hence we shift the pattern by $shift[j - 1]$, which is 3. That is, we add 3 to i and remove 3 from j to get $i = 8$ and $j = 2$ to continue:

```
        j = 2
          ↓
        pappar
pappappapparrassanuaragh
      ↑
    i = 6
```

In general, we have to be careful in the case that $shift[j-1] > j$:

```
j = 0
  ↓ (×)
   pappar
panther
   ↑
i = 2
```

Here $i = 2$ and $j = 0$, when a mismatch occurs. Now $shift[j-1]$ is 1; hence we change i to 3 to continue matching at $t[3]$, but we do not subtract 1 from j, since j is already 0. We continue with $i = 3$ and $j = 0$ as shown below:

```
  j = 0
    ↓ (×)
     pappar
panther
     ↑
  i = 3
```

This case always occurs when a mismatch is found for $j = 0$. □

Suppose we know how to compute the table for a pattern p. The Knuth-Morris-Pratt algorithm then works as follows:

Algorithm 9.3.5 Knuth-Morris-Pratt Search. This algorithm searches for an occurrence of a pattern p in a text t. It returns the smallest index i such that $t[i..i+m-1] = p$, or -1 if no such index exists.

Input Parameters: p, t
Output Parameters: None

```
knuth_morris_pratt_search(p, t) {
  m = p.length
  n = t.length
  knuth_morris_pratt_shift(p, shift) // compute array shift of shifts
  i = 0
  j = 0
  while (i + m ≤ n) {
    while (t[i + j] == p[j]) {
      j = j + 1
      if (j ≥ m)
        return i
    }
    i = i + shift[j - 1]
    j = max(j - shift[j - 1], 0)
  }
  return -1
}
```

Let us assume for the moment that *knuth_morris_pratt_shift* correctly computes the *shift* table in time $O(m)$ where m is the length of the pattern (we will see how to do this later). To prove correctness of *knuth_morris_pratt_search*, we need to show that our intuition about using the *shift* array is correct; that is, the shift array takes us to the next potential match without overlooking any intermediary matches. The following lemma justifies this intuition.

Lemma 9.3.6. *Assume* $p[0..j-1] = t[i..i+j-1]$ *and* $p[j] \neq t[i+j]$*; that is, we have a partial match of* p *starting at position* i *that failed at* $i+j$*. Let* $i' = i + shift[j-1]$*, and* $j' = \max\{j - shift[j-1], 0\}$*. Then*

(a) $p[0..j'-1] = t[i'..i'+j'-1]$*, and*

(b) $p \neq t[k..k+m-1]$ *for* $i \leq k < i'$*.*

That is, $p[0..j'-1]$ *matches* t *starting at position* i'*, and there are no matches for* p *in positions* $i, i+1, \ldots, i'-1$*.*

Proof. By definition $shift[j-1]$ is an allowable shift, namely

$$p[0..j-1-shift[j-1]] = p[shift[j-1]..j-1].$$

Therefore,

$$\begin{array}{lcll} p[0..j'-1] & = & p[0..j-1-shift[j-1]] & \text{by definition of } j' \\ & = & p[shift[j-1]..j-1] & \text{by definition of } shift[j-1] \\ & = & t[i+shift[j-1]..j-1] & p[0..j-1] = t[i..i+j-1] \\ & = & t[i'..i'+j'-1] & \text{by definition of } i' \text{ and } j', \end{array}$$

showing that (a) is true (note that in the case $j' = 0$ all of these strings are empty). Statement (b) follows, because $shift[j-1]$ is the smallest allowable shift; that is, we do not miss any matches by applying the shift. To prove this, assume, by way of a contradiction, that $p[0..m-1] = t[k..k+m-1]$ for some $i \leq k < i'$. Let $k' = k - i$. Then

$$\begin{array}{lcll} p[0..j-1-k'] & = & t[k..k+j-1-k'] & \text{by assumption} \\ & = & t[k..i+j-1] & \text{by definition of } k' \\ & = & p[k'..j-1] & \text{since we assumed that} \\ & & & p[0..j-1] = t[i..i+j-1], \end{array}$$

and therefore $k' \geq shift[j-1]$ (by definition of $shift[j-1]$) contradicting $k' = k - i < i' - i = shift[j-1]$. ■

Applying Lemma 9.3.6 to justify the use of shifting, we can now prove the algorithm correct.

Theorem 9.3.7. *The algorithm knuth_morris_pratt_search correctly computes the position of the first occurrence of pattern* p *in text* t *if there is one or returns* -1 *otherwise. The algorithm runs in time* $O(m+n)$*.*

Proof. We are assuming that *knuth_morris_pratt_shift*(p) correctly computes the *shift* table for p in time $O(m)$; that is,

$$shift[k] = \min\{s > 0 \mid p[0..k-s] = p[s..k]\} \quad \text{for } -1 \le k < m.$$

To reference the code, we assign line numbers:

```
(1) while (i + m ≤ n) {
(2)   while (t[i + j] == p[j]) {
(3)     j = j + 1
(4)     if (j ≥ m)
(5)       return i
(6)   }
(7)   i = i + shift[j − 1]
(8)   j = max(j − shift[j − 1], 0)
(9) }
```

We first verify that the algorithm runs in time $O(m+n)$. The outer loop (1) is performed at most n times, since i is increased in each iteration as the *shift* array contains only values greater than zero. The inner loop (2), however, might take $O(m)$ steps giving us an $O(mn)$ upper bound, which is not good enough. We try to extend the idea that i can be increased at most n times to the inner loop in which j gets increased. In a first attempt, we might trace the value of the expression $i + j$. However, that value remains constant if the inner loop fails for some $j > 0$ (if $j > 0$, then $shift[j-1] \le j$; hence $i + j$ does not change in that case). A slight variation of this idea does work: Trace the value of $2i + j$ in line (2), where the condition of the inner while loop is checked. If the condition evaluates to true, j, and therefore $2i + j$, is increased. If it fails, the value $shift[j-1]$ is added to i and at most $shift[j-1]$ is subtracted from j. In consequence, $2i + j$ is increased by at least $shift[j-1] \ge 1$. Hence, every time we return to line (2), the value of $2i + j$ has increased. Since $2i + j \le 2n + m$, this can happen at most $2n + m$ many times, giving us an upper bound of $O(m + n)$ for the running time of the loop. Since we assumed that the initialization takes time $O(m)$, we obtain an overall running time of $O(n + m)$.

To show that the algorithm is correct, we use a loop invariant. We claim that the following two statements are loop invariants for the inner loop in line (2); that is, both are true every time the algorithm performs line (2).

(a) $p[0..j-1] = t[i..i+j-1]$, and

(b) $p[0..m-1] \ne t[k..k+m-1]$ for $0 \le k < i$.

Intuitively, (a) claims that whenever we enter the loop we have a partial match of the first j letters of the pattern starting at position i of the text; (b) implies that the algorithm has not missed a match of the pattern at an earlier position.

Before we show that the two statements are indeed a loop invariant, we apply them to prove *knuth_morris_pratt_search* correct. We already showed that the algorithm terminates. Termination can happen in two ways: by

leaving the if statement in line (4) or after having finished the outer loop. Let us consider the latter case first. For the algorithm to exit the outer loop, it must be true that $i + m > n$. Then (b) implies that there no matches for the pattern up to and including position $n - m$. However, there cannot be any matches starting after that position since there are less than m letters left, and the algorithm correctly returns -1. In the other case, the algorithm leaves through the return statement in line (5). When we entered the loop in line (2) we knew that $p[0..j-1] = t[i..i+j-1]$ from (a), and $p[j] = t[i+j]$ (otherwise we would not have entered the inner loop); hence we have $p[0..j] = t[i..i+j]$. We exit only if $j+1$ is at least m, hence $j \geq m-1$ and therefore $p = p[0..m-1] = t[i..i+m-1]$, which means we have found an occurrence of the pattern starting at position i. By (b) we know that there are no earlier occurrences we missed, so the algorithm correctly returns i at that point.

We conclude the proof by verifying that (a) and (b) are indeed a loop invariant. When we first test the loop condition, (a) is true because $j = 0$ and $p[0..-1]$ and $t[i..i-1]$ are both the empty string; (b) is true because $i = 0$, which means that there are no previous positions k to consider.

We now need to show that, if (a) and (b) are true whenever we perform line (2), they are still true when we return to (2) the next time.

There are two cases, depending on whether the test in (2) fails or succeeds. If $t[i+j] = p[j]$, we increase j in line (3), so condition (a) will be true for the new j; (b) remains true, since i did not change at all. If, on the other hand, $t[i+j] \neq p[j]$, we do not enter the inner loop but continue with lines (7) and (8). This increases i by $shift[j-1]$ and decreases j by the same value. Let us call the new values $i' = i + shift[j-1]$, and $j' = \max\{j - shift[j-1], 0\}$. Lemma 9.3.6 now tells us that

(a′) $p[0..j'-1] = t[i'..i'+j'-1]$, and
(b′) $p \neq t[k..k+m-1]$ for $i \leq k < i'$.

Since (b) was true at the start of the loop, we know that $p \neq t[k..k+m-1]$ for $0 \leq k < i$. Together with (b'), this implies that $p[0..m-1] \neq t[k..k+m-1]$ for $0 \leq k < i'$. Moreover, (a') tells us that $p[0..j'-1] = t[i'..i'+j'-1]$, so (a) and (b) are still true when we return to (2) and are therefore a loop invariant. ■

We still have to show how to compute the shift table in time $O(m)$. The idea is the same as for finding the pattern in a string, with the difference that we try to locate the pattern within itself. Assume that $t = p$ and we run *knuth_morris_pratt_search*. Whenever we increase j in the inner loop, it is because we have found a partial match $p[0..j] = p[i..i+j]$, which implies that $shift[i+j] \leq i$ because

$$shift[i+j] = \min\{s > 0 \mid p[0..i+j-s] = p[s..i+j]\}.$$

On the other hand, we are making sure we are not missing any earlier matches; so, in fact, at that point $shift[i+j] = i$, and we can set this value in the *shift*

table. This suggests that we just run *knuth_morris_pratt_search* with $t = p$ and set the values of $shift[j]$ in the inner loop. However, we are actually using values from the *shift* table in *knuth_morris_pratt_search*, so how can this work? As it turns out, we only ever access values in the *shift* table that have already been computed earlier. We can therefore adapt *knuth_morris_pratt_search* to compute the shifts.

Algorithm 9.3.8 Knuth-Morris-Pratt Shift Table. This algorithm computes the shift table for a pattern p to be used in the Knuth-Morris-Pratt search algorithm. The value of $shift[k]$ is the smallest $s > 0$ such that $p[0..k-s] = p[s..k]$.

```
 Input Parameter:  p
Output Parameter:  shift
knuth_morris_pratt_shift(p, shift) {
  m = p.length
  shift[-1] = 1 // if p[0] ≠ t[i] we shift by one position
  shift[0] = 1 // p[0.. - 1] and p[1..0] are both the empty string
  i = 1
  j = 0
  while (i + j < m)
    if (p[i + j] == p[j]) {
      shift[i + j] = i
      j = j + 1;
    }
    else {
      if (j == 0)
        shift[i] = i + 1
      i = i + shift[j - 1]
      j = max(j - shift[j - 1], 0 )
    }
}
```

Example 9.3.9. The following table shows how *knuth_morris_pratt_shift* computes the shift table for "pappar":

			$shift[-1] = 1$
			$shift[0] = 1$
$i = 1, j = 0$:	$p[1] \neq p[0], j = 0$	$\rightarrow$	$shift[1] = 2$
$i = 2, j = 0$:	$p[2] = p[0]$	$\rightarrow$	$shift[2] = 2$
$i = 2, j = 1$:	$p[3] \neq p[1], j \neq 0$		
$i = 3, j = 0$:	$p[3] = p[0]$	$\rightarrow$	$shift[3] = 3$
$i = 3, j = 1$:	$p[4] = p[1]$	$\rightarrow$	$shift[4] = 3$
$i = 3, j = 2$:	$p[5] \neq p[2], j \neq 0$		
$i = 5, j = 0$:	$p[5] \neq p[0], j = 0$	$\rightarrow$	$shift[5] = 6$

This yields the table we used in Example 9.3.2. □

Example 9.3.10. As another example, we trace *knuth_morris_pratt_shift* on input "1010001":

$$
\begin{array}{lll}
 & & shift[-1] = 1 \\
 & & shift[0] = 1 \\
i = 1, j = 0: & p[1] \neq p[0], j = 0 \rightarrow & shift[1] = 2 \\
i = 2, j = 0: & p[2] = p[0], j = 0 \rightarrow & shift[2] = 2 \\
i = 2, j = 1: & p[3] = p[1], j \neq 0 \rightarrow & shift[3] = 2 \\
i = 2, j = 2: & p[4] \neq p[2], j \neq 0 & \\
i = 4, j = 0: & p[4] \neq p[0], j = 0 \rightarrow & shift[4] = 5 \\
i = 5, j = 0: & p[5] \neq p[0], j = 0 \rightarrow & shift[5] = 6 \\
i = 6, j = 0: & p[6] = p[0], j = 0 \rightarrow & shift[6] = 7
\end{array}
$$

For example, $shift[2]$ and $shift[3]$ are both 2 since the initial part "10" of "1010001" repeats immediately. □

†Correctness of Shift Table Computation

We need to show that *knuth_morris_pratt_shift* correctly computes the values

$$shift[k] = \min\{s > 0 \mid p[0..k-s] = p[s..k]\}$$

for $k = -1, 0, \ldots, m$ where $m = |p|$. We first prove a strengthening of Lemma 9.3.6.

Lemma 9.3.11. *Assume* $p[0..j-1] = p[i..i+j-1]$, *and* $p[j] \neq p[i+j]$. *Let* $i' = i + shift[j-1]$, *and* $j' = \max\{j - shift[j-1], 0\}$. *Then*

(a) $p[0..j'-1] = p[i'..i'+j'-1]$, *and*

(b) $p[0..i'+j'-1-s] \neq p[s..i'+j'-1]$ *for* $i \leq s < i'$.

That is, $p[0..j'-1]$ *matches* p *starting at position* i', *and either* $shift[i'+j'-1] < i$ *or* $shift[i'+j'-1] \geq i'$.

Proof. We prove the truth of (a) as in Lemma 9.3.6. Since $shift[j-1]$ is an allowable shift, we have

$$
\begin{array}{lll}
p[0..j'-1] & p[0..j-1-shift[j-1]] & \text{definition of } j' \\
= & p[shift[j-1]..j-1] & \text{by definition of } shift[j-1] \\
= & p[i+shift[j-1]..j-1] & p[0..j-1] = p[i..i+j-1] \\
= & p[i'..i'+j'-1] & \text{by definition of } i' \text{ and } j',
\end{array}
$$

showing that (a) is true (again, if $j' = 0$ all of the strings are empty). Suppose for a contradiction that (b) fails; that is, for some s we have

$$p[0..i'+j'-1-s] = p[s..i'+j'-1],$$

†This subsection can be omitted without loss of continuity.

and $i \le s < i'$. Note that this implies that $p[0..i+j-1-s] = p[s..i+j-1]$ since $i'+j' \ge i+j$. Let $s' = s-i$. Then

$$\begin{array}{lcll} p[0..j-1-s'] & = & p[0..i+j-1-s] & \text{by definition of } s' \\ & = & p[s..i+j-1] & \text{by assumption for contradiction} \\ & = & p[s'..j-1] & \text{since } p[0..j-1] = p[i..i+j-1] \end{array}$$

Hence $s' \ge shift[j-1]$ (by definition of $shift[j-1]$) contradicting $s' = s-i < i'-i = shift[j-1]$. ■

Lemma 9.3.12. *The algorithm knuth_morris_pratt_shift correctly computes the shift array in time $O(m)$.*

Proof. Consider the values of the expression $i+j$ as we enter the while loop. Initially that value is 1, and we claim the value never decreases. If $p[i+j] = p[j]$, then j is increased, hence $i+j$ increases. If $p[i+j] \ne p[j]$, we add $shift[j-1]$ to i and subtract the same value from j unless the difference becomes negative, in which case we let $j = 0$. In either case $i+j$ does not decrease.

We also note that the algorithm assigns a value to $shift[i+j]$ exactly when $i+j$ increases by one: if $p[i+j] = p[j]$, then $shift[i+j]$ is set, and j increased by one; if $p[i+j] \ne p[j]$, then $shift[i+j]$ is set if $j = 0$, and in that case i, and thus $i+j$, are increased by one. It follows that at any point of the algorithm, $shift[k]$ has been assigned a value for all $k < i+j$. Finally note that, if we set a value in the *shift* array, we set it to i or $i+1$, both of which are always at least one.

As in the case of the search algorithm, we can now prove termination by tracing the value of $2i+j$. This value is at most $3m$ and we claim it increases by at least one in each iteration of the loop, which implies that the loop is performed at most $3m$ times. Hence the running time is $O(m)$. There are two cases: If $p[i+j] = p[j]$ then j is increased, hence $2i+j$ increases. Otherwise, if $j \ne 0$, then $2i+j$ increases by a total of $shift[j-1]$, which is always at least one. In the final case we have $j = 0$, and i is increased by 1 and j remains zero, which increases $2i+j$ by two.

The algorithm initializes both $shift[-1]$ and $shift[0]$ to 1. Since

$$shift[-1] = \min\{s > 0 \mid p[0..-1-s] = p[s..-1]\} = 1,$$

and similarly $shift[0] = 1$, because $p[0..-1]$ and $p[0..-2]$ are empty, this is correct. Furthermore, since the while loop begins with $i+j$ having value 1, and $i+j$ never decreases, these two values are never overwritten, and hence remain correct.

The main part of the algorithm is the loop, which is captured by the following loop invariant:

(a) The values of $shift[k]$ are correct for all $-1 \le k < i+j$.

(b) $p[0..j-1] = p[i..i+j-1]$.

(c) $p[0..i+j-1-s] \neq p[s..i+j-1]$ for $0 < s < i$.

We show that these three statements are indeed a loop invariant in Lemma 9.3.13. Intuitively, (a) says that we have computed all values of *shift* correctly so far, (b) says that for the current values of i and j we have a partial match of the pattern at positions i through $i+j-1$ (which for $j = 0$ is always true), and (c) implies that $shift[i+j-1] \geq i$.

The loop invariant immediately settles the correctness of the algorithm: As the loop terminates with $i + j = m$, (a) states that all of the values in the *shift* table are correct. ■

Lemma 9.3.13. *The statements*

(a) The values of $shift[k]$ *are correct for all* $-1 \leq k < i + j$,

(b) $p[0..j-1] = p[i..i+j-1]$, *and*

(c) $p[0..i+j-1-s] \neq p[s..i+j-1]$ *for* $0 < s < i$,

are a loop invariant for our implementation of knuth_morris_pratt_shift; that is, all three conditions are true every time the loop condition is checked.

Proof. The first time the loop is entered we have $i = 1$ and $j = 0$, so (a) is true because we set $shift[-1]$ and $shift[0]$ correctly; (b) is true because both $p[0..j-1]$ and $p[i..i+j-1]$ are empty; and (c) is true because there is no s between 0 and i.

The next step is to show that if the loop invariant holds at the beginning of the execution of the loop, it still holds after the body has been executed.

We distinguish two cases.

First Case: $p[i+j] = p[j]$. In this case we execute

$$shift[i+j] = i$$
$$j = j + 1$$

We have to show that (a), (b), and (c) then hold for the new values. To distinguish between the values of i and j before and after the body of the loop is executed, call the new values $i' = i$, and $j' = j + 1$. We have to verify (a), (b), and (c) for i' and j'.

Let us begin with (b). We know that $p[0..j-1] = p[i..i+j-1]$. Since we assumed that $p[i+j] = p[j]$, we get $p[0..j'-1] = p[0..j] = p[i..i+j] = p[i'..i'+j'-1]$, which shows that (b) is true for i' and j'. Consider (c) next. We know that $p[0..i+j-1-s] \neq p[s..i+j-1]$ for $0 < s < i$, which implies $p[0..i+j-s] \neq p[s..i+j]$ for $0 < s < i$ (we are adding a single letter to strings that are already different). This, however, is the same as $p[0..i'+j'-1-s] \neq p[s..i'+j'-1]$ for $0 < s < i'$ (by definition of i' and j'). We are left with (a). We have to show that $shift[k]$ is correct for all $-1 \leq k < i' + j'$. By assumption, the values are correct for all $-1 \leq k < i + j$; so the only value we have to check is $shift[i+j]$, which was set to i. Because of (b), we know that $p[0..j] = p[i..i+j]$, and therefore $shift[i+j] \leq i$ (by definition

$shift[i+j] = \min\{s > 0 \mid p[0..i+j-s] = p[s..i+j]\}$). Assume for a contradiction that $0 < s = shift[i+j] < i$. Then $p[0..i+j-s] = p[s..i+j]$ for some s with $0 < s < i$, or (using i', j' instead), $p[0..i'+j'-1-s] = p[s..i'+j'-1]$ for some $s < i$, contradicting (c) for i' and j', which we established earlier.

Second Case: $p[i+j] \neq p[j]$. In this case we execute

```
if (j == 0)
   shift[i] = i + 1
i = i + shift[j - 1]
j = max(j - shift[j - 1], 0)
```

The values of i and j after executing this code are $i' = i + shift[j-1]$ and $j' = \max\{j - shift[j-1], 0\}$. Lemma 9.3.11 tells us that (b) and (c) remain true for i' and j', so we are left with verifying (a). We assign $shift[i] = i + 1$ only if $j = 0$. However, $j = 0$ means that we must have had $p[i] \neq p[0]$, which immediately implies $shift[i] \neq i$. But $shift[i] \geq i$ by (c), and therefore $shift[i] = i + 1$ (because $shift[k] \leq k + 1$ for $k \geq 0$). ■

Exercises

1S. Compute the *shift* table for the pattern "barbara".

2. Compute the *shift* table for the pattern "abracadabra".

3. Compute the *shift* table for the pattern "entente".

4S. Compute the *shift* table for the pattern "cancan".

5. Compute the *shift* table for the pattern "mathematics".

6S. Trace *knuth_morris_pratt_search* searching for an occurrence of "cancan" in "cacancacanccancanca". How many comparisons did the algorithm make?

7. Trace *knuth_morris_pratt_search* searching for an occurrence of "abracadabra" in "abrabracabracadabracadabracad". How many comparisons did the algorithm make?

8. Trace *knuth_morris_pratt_search* searching for an occurrence of "entente" in "tenttentetententen". How many comparisons did the algorithm make?

9S. Trace *knuth_morris_pratt_search* searching for an occurrence "mathematics" in "mathematicians and mathematics". How many comparisons did the algorithm make?

9.4 The Boyer-Moore-Horspool Algorithm

Boyer and Moore came up with a seemingly innocuous idea: Why not compare the pattern to the text from right to left instead of from left to right, while keeping the shift direction the same? For example, reimplementing the simple text search with this idea yields the following algorithm.

Algorithm 9.4.1 Boyer-Moore Simple Text Search. This algorithm searches for an occurrence of a pattern p in a text t. It returns the smallest index i such that $t[i..i+m-1] = p$ or -1 if no such index exists.

```
 Input Parameters:  p, t
Output Parameters:  None

boyer_moore_simple_text_search(p, t) {
  m = p.length
  n = t.length
  i = 0
  while (i + m ≤ n) {
    j = m - 1 // begin at the right end
    while (t[i + j] == p[j]) {
      j = j - 1
      if (j < 0)
        return i
    }
    i = i + 1
  }
  return -1
}
```

While this variant of our earlier *simple_text_search* still takes time $O(mn)$, it does lend itself to the implementation of two heuristics called the **occurrence** and the **match** heuristics, which speed it up significantly.

We first look at some examples of the occurrence heuristic.

Example 9.4.2. Suppose we are searching for the word "rum" in "conundrum". Running the Boyer-Moore version of the simple text search, we first compare the "m" of "rum" to the "n" of "conundrum":

```
rum
conundrum
```

The match fails. Now we know that "rum" does not contain the letter "n" at all, so we can actually shift the word over by three positions:

```
   rum
conundrum
```

This time "d" and "m" do not match and again we know that "rum" does not contain a "d", so we can move the pattern by another three positions:

```
      rum
conundrum
```

to finally find a match. □

Example 9.4.3. As a second example of the occurrence heuristic, let us search for "drum" in "conundrum":

```
drum
conundrum
```

Again we have a mismatch, but this time the letter "u" actually occurs in the word "drum" so we can only move the pattern over by one letter, aligning the rightmost "u" in "drum" with the "u" from "conundrum":

```
 drum
conundrum
```

The mismatch between "m" and "n" here leads to a shift of four positions:

```
    drum
conundrum
```

giving us the final match. □

Example 9.4.4. As a final example of the occurrence heuristic, let us try to match "natu" against "conundrum":

```
natu
conundrum
```

Matching "t" with "n" fails, so we shift the pattern over for the "n"s to match:

```
  natu
conundrum
```

Here "u" against "d" fails and "natu" does not contain a "d", so we need to shift the pattern beyond the "d"; and since there are only three letters of the text left after "d", the search fails. □

We summarize the occurrence heuristic as follows: Match the pattern against the text from right to left. When encountering a mismatch with a letter α from the text, shift the pattern to align the rightmost occurrence of α in the pattern with the one in the text. If α does not occur in the pattern at all, move the pattern one position beyond α.

There is one subtle problem with this rule: It might make us shift the pattern back to the left rather than to the right if the rightmost occurrence of α has already moved beyond the current position. This situation is illustrated in the next example.

Example 9.4.5. We want to find a match for "date" in "detective":

```
date
detective
```

Using the occurrence heuristic the mismatch between "a" and "e" should lead to the following alignment:

```
date
  detective
```

This would mean moving the pattern back to a position dealt with earlier. □

Boyer and Moore's solution for this problem is to shift the pattern to the right by one position in case we would end up with a negative shift (a shift to the left).

Example 9.4.6. Applying the Boyer-Moore solution to

```
date
detective
```

we obtain

```
 date
detective
```

since the occurrence heuristic would lead to a negative shift, as seen in Example 9.4.5, and we therefore move the pattern by one position to the right. □

The second technique, the match heuristic, is similar to the idea used in the Knuth-Morris-Pratt algorithm: If we fail halfway through a match, we can use the information about what we have seen of the text so far to shift the pattern to the next possible match. Since we are comparing from right to left, we need to compute the shift table for the reverse of the pattern.

Example 9.4.7. Let us use the match heuristic to find a match for p = "banana" in t ="a banana":

```
banana
a banana
```

We would match $p[3..5]$ ="ana" to $t[3..5]$ ="ana" in "a banana" and then find a mismatch of $p[2]$ = "n" against $t[2]$ = "b". A Knuth-Morris shift table for "ananab" yields $shift[3] = 2$; hence we shift p by two positions to the right:

```
  banana
a banana
```

at which point we have found a match. □

The Boyer-Moore algorithm implements both of these heuristics, choosing whichever gives the larger shift. This makes it one of the fastest search algorithms in both theory and practice. The match heuristic guarantees a running time of $O(m+n)$ (as in Knuth-Morris-Pratt), and there are numerous improvements to reduce the actual number of comparisons. It is, however, a rather complex algorithm to implement, mostly because of the match heuristic. For this reason, we consider a variant suggested by Horspool, which is known as Boyer-Moore-Horspool. It implements a modification of the occurrence heuristic and is very fast in practice, although its worst-case running time is $\Theta(mn)$.

We earlier mentioned a subtle problem we can run into with the occurrence heuristic: It can potentially lead us to negative shifts (shifts to the left) because the rightmost occurrence of the letter is already to the right of the current position as in the example

```
date
detective
```

where the mismatched "e" in the text would lead to shifting the pattern to the left by two positions. This case occurs only if we have already matched parts of the pattern. Compare this to the case in which the last letter fails to match; we would always shift to the right. Horspool's solution is to always shift according to the last letter, not to the letter that actually fails. This eliminates negative shifts.

Example 9.4.8. In the case of

```
date
detective
```

we fail matching "a" against "e". We then shift the pattern to align the fourth letter in "detective" (the "e") with the next possible match in "date". Since "date" does not contain another "e", we can shift the entire pattern beyond the "e"

```
    date
detective
```

skipping three positions. □

In summary, Horspool's version reacts to a mismatch by trying to align $t[i+m-1]$ with the rightmost occurrence of that letter in the pattern to the left of $p[m-1]$. This rule guarantees that the pattern is always shifted to the right. To perform the correct shift, we need to know for each letter of the alphabet what its rightmost occurrence is in the pattern to the left of $p[m-1]$. More formally,

$$shift[w] = \begin{cases} m-1-\max\{i < m-1 \mid p[i] = w\} & \text{if } w \text{ is in } p[0..m-2] \\ m & \text{otherwise.} \end{cases}$$

Example 9.4.9. For the word p ="kettle", we get the following shifts. If the letter in t aligned with the last letter of p is a "k", shift by 5; if it is an "e", shift by 4; if it is a "t", shift by 2; and if it is an "l", shift by 1. For any other letter, the shift is 6.

For example if t ="tea kettle", we initially align:

```
kettle
tea kettle
```

The mismatch of "l" and "k" leads to a shift of 4 because $t[5]$ = "e", and we find the match:

```
    kettle
tea kettle
```

Similarly, the mismatch of "e" and "t" in the following example leads to a shift of 2 because the last letter is a "t":

```
kettle
a kettle
```

Again, we find an immediate match. □

Algorithm 9.4.10 Boyer-Moore-Horspool Search. This algorithm searches for an occurrence of a pattern p in a text t over alphabet Σ. It returns the smallest index i such that $t[i..i+m-1] = p$ or -1 if no such index exists.

Input Parameters: p, t
Output Parameters: None

```
boyer_moore_horspool_search(p, t) {
  m = p.length
  n = t.length
  // compute the shift table
  for k = 0 to |Σ| - 1
    shift[k] = m
  for k = 0 to m - 2
    shift[p[k]] = m - 1 - k
  // search
  i = 0
  while (i + m ≤ n) {
    j = m - 1
    while (t[i + j] == p[j]) {
      j = j - 1
      if (j < 0)
        return i
    }
    i = i + shift[t[i + m - 1]] // shift by last letter
  }
  return -1
}
```

The *boyer_moore_horspool_search* algorithm runs in time $O(mn)$ if we assume that the alphabet Σ is fixed. Unlike the full Boyer-Moore algorithm and Knuth-Morris-Pratt algorithm, its space usage does not depend on p. Indeed, it uses only space $O(|\Sigma|)$, which is constant if we assume that the alphabet is fixed. In practice, the algorithm gives running times comparable to Boyer-Moore and is therefore quite useful, since it is much simpler to implement and uses less space.

Theorem 9.4.11. *The algorithm boyer_moore_horspool_search correctly computes the position of the first occurrence of pattern p in text t if there is one or returns -1 otherwise. The algorithm runs in time $O(mn)$.*

We leave the proof of the theorem to Exercise 9 (running time) and Exercises 9.13–9.15 (correctness).

Exercises

1S. In Example 9.4.5 we saw that the occurrence heuristic can lead to a negative shift. Give an example, where the pattern p is shifted by $|p| - 1$ positions to the left (the maximal possible).

2. In Example 9.4.5 we saw that the occurrence heuristic can lead to a negative shift. Is it possible that it leads to a zero shift?

3S. Compute the *boyer_moore_horspool_search* shift table for the pattern "banana".

4. Compute the *boyer_moore_horspool_search* shift table for the pattern "antenna".

5. Compute the *boyer_moore_horspool_search* shift table for the pattern "panpipe".

6S. Use *boyer_moore_horspool_search* to search for "banana" in "cananabananab". How many comparisons does the algorithm make? (Use the results from Exercise 3.)

7. Use *boyer_moore_horspool_search* to search for "pappar" in "pappappapparrassanuaragh".

8. Use the match heuristic to search for "banana" in "cananabananab".

9S. Show that *boyer_moore_horspool_search* runs in time $\Omega(mn)$ in the worst case (assume the alphabet is fixed).

10. Show that *boyer_moore_horspool_search* runs in time $O(n/m)$ in the best case (assume the alphabet is fixed). How does that compare to the best-case running times of the other algorithms we have seen?

11. We cannot always assume that the alphabet Σ is of a small, fixed size. What is the worst-case running time of *boyer_moore_horspool_search* if we take into account the size of the alphabet?

9.5 Approximate Pattern Matching

There are numerous variants of the approximate pattern-matching problem, reflecting the many situations in which a search has to be based on partial, imprecise information.

Think, for example, of comparing two files on your file system to find out how similar they are or what has changed in a new version of the file. The UNIX operating system supports the commands `diff` and `sdiff` to do that.

Example 9.5.1. We have files `prereq1.txt` containing CSC 225, CSC323, CSC343, CSC345, CSC415, CSC 416, and CSC 417 and `prereq2.txt` containing CSC 211, CSC 212, CSC 309, CSC 343, CSC 345, CSC 415, and CSC 416, each class name listed on a separate line of the file. Running the UNIX command `sdiff prereq1.txt prereq2.txt` yields:

```
CSC 225         |   CSC 211
CSC 323         |   CSC 212
                >   CSC 309
CSC 343             CSC 343
CSC 345             CSC 345
CSC 415             CSC 415
CSC 416             CSC 416
CSC 417         <
```

Here | means different, > denotes an additional line in the second file, and < denotes an additional line in the first file. □

How does `sdiff` know whether it is actually pairing up the right text if there are several possibilities? The answer, of course, is, it does not and cannot know, because it does not know how the two files are related. One way to approach this problem is to define a measure of how different the two documents are by thinking of the differences as changes made by somebody starting with one file and editing it to become identical to the second file. The smallest number of changes necessary to turn one file into the other is called the **edit distance** between the two files; and associated with it we have the sequence of edits turning one file into the other, which we can use to distinguish between text that has changed, and text that has not changed.

Example 9.5.2. In the case of `prereq1.txt` and `prereq2.txt` from Example 9.5.1, four changes are sufficient to turn `prereq1.txt` into `prereq2.txt`: Change CSC 225 to CSC 211, CSC 323 to CSC 212, add CSC 309, and remove CSC 417. Three changes would not be enough to turn one file into the other, justifying the output of `sdiff`. □

A seemingly different problem occurs when searching for a word in a text, but we are not entirely sure of the correct spelling. In that case, we want to tell the program that it should look for text similar to the one entered. It turns out that solving this problem is strongly related to comparing two texts.

A final variant we consider allows the pattern to contain wildcards such as "$*$" for any sequence of symbols and "?" for any single symbol.

Example 9.5.3. In UNIX the command `ls *.html` lists all files in the current directory that end in `.html`, such as `index.html` and `default.html`. The command `ls page?.html` would list `page2.html` but not `page11.html`. □

Here we consider only the don't-care wildcard "?" matching a single letter. The "$*$" wildcard can be handled using regular expressions, which we investigate in Section 9.6.

Edit Distance

We gave an intuitive definition of the edit distance of two texts s and t as the smallest number of editorial changes we would have to make to turn s into t. We need to specify what editorial changes are available. We allow three operations at unit cost:

(R) Replace one letter with another letter.
(D) Delete one letter.
(I) Insert one letter.

Example 9.5.4. Let us see how to turn "ghost" into "house" using three operations:

g	h	o	s	t	delete g at position 0
h	o	s	t		insert u after position 1
h	o	u	s	t	replace t by e at position 4
h	o	u	s	e	

Note that we do not require the intermediary words to be correct English words. □

Of course, there might be several ways of turning one word into another, and some of them may take more operations than others. We define the *edit distance* $\text{dist}(s,t)$ of two words s and t to be the smallest number of operations selected from (R), (D), and (I) that turns s into t. Note that we can always turn s into t; in the worst case we can delete all $|s|$ letters of s and then insert the $|t|$ letters of t in the right order. This shows that

$$\text{dist}(s,t) \leq |s| + |t|.$$

It turns out that the edit distance of two words can be computed very naturally using the dynamic-programming technique from Chapter 8. As subproblems of computing $\text{dist}(s,t)$, we first compute the edit distance between prefixes of s and t, namely $\text{dist}(s[0..i], t[0..j])$, for $0 \leq i < |s|, 0 < j < |t|$.

Example 9.5.5. Here is the table of distances for the words s = "presto", and t = "peseta":

		0	1	2	3	4	5
		p	e	s	e	t	a
0	p	0	1	2	3	4	5
1	r	1	1	2	3	4	5
2	e	2	1	2	2	3	4
3	s	3	2	1	2	3	4
4	t	4	3	2	2	2	3
5	o	5	4	3	3	3	3

For example, $\text{dist}(s[0], t[0]) = 0$ because they are both the same letter, "p". Then $\text{dist}(s[0], t[0..j]) = j$ because the quickest way of turning "p" into $t[0..j]$, the prefixes of "peseta", is by appending the j letters $t[1..j]$. Similarly, $\text{dist}(s[0..1], t[0..j]) = j$ for $j \geq 1$ because we can replace "r" by "e" and append the remaining letters. Also, $\text{dist}(s[0..4], t[0..3])$ is 2 because we can delete the "r" from "prest" and replace the "t" by "e". Finally, we can turn "presto" into "peseta" by removing the "r", inserting an "e" before the "t", and replacing the "o" by an "a". □

Let us define $d_{i,j} = \text{dist}(s[0..i], t[0..j])$. Then the edit distance of s and t is $\text{dist}(s, t) = d_{|s|-1,|t|-1}$. The question is, how do we compute the values in the table?

We need to compute $d_{i,j}$, the edit distance between $s[0..i]$, and $t[0..j]$. Let us begin at the end: How did $s[i]$, the last letter of $s[0..i]$, become $t[j]$, the last letter of $t[0..j]$? There are three operations we can use:

(R) Replace $s[i]$ by $t[j]$ and turn $s[0..i-1]$ into $t[0..j-1]$.
This takes at most $d_{i-1,j-1} + 1$ operations.

(D) Delete $s[i]$ and turn $s[0..i-1]$ into $t[0..j]$.
This takes $d_{i-1,j} + 1$ operations.

(I) Insert $t[j]$ at the end of $s[0..i]$ and turn $s[0..i]$ into $t[0..j-1]$.
This takes $d_{i,j-1} + 1$ operations.

However, there is one special case: If $s[i]$ and $t[j]$ are the same letter, then we do not actually have to replace $s[i]$ by $t[j]$ in (R), and it takes only $d_{i-1,j-1}$ steps to turn $s[0..i]$ into $t[0..j]$ by turning $s[0..i-1]$ into $t[0..j-1]$.

This discussion shows that we can turn $s[0..i]$ into $t[0..j]$ in at most

$$\min \left\{ \begin{array}{l} d_{i-1,j-1} + \left\{ \begin{array}{ll} 0 & \text{if } s[i] = t[j] \\ 1 & \text{else} \end{array} \right. \\ d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \end{array} \right\} \text{ steps.}$$

Since we exhausted all of the possibilities of how $s[i]$ can turn into $t[j]$, this formula gives us the correct value of $d_{i,j}$. The formula shows that $d_{i,j}$ can

be computed from the values of $d_{i-1,j}$, $d_{i-1,j-1}$ and $d_{i,j-1}$. In terms of the table containing the values, this means that $d_{i,j}$, the content of cell (i,j), can be computed from just three neighboring cells: cell $(i-1,j)$ to the left, cell $(i-1,j-1)$ to the upper left, and cell $(i,j-1)$ above. The easiest way to achieve this is to compute the table row by row starting in the upper left-hand corner.

Example 9.5.6. With our recursive formulas at hand, let us have another look at the table we computed in Example 9.5.5. How did we get started? In row 0 we were already applying our recursive formulas to an invisible row -1. Let us add a row and column -1 to the table. The value of $d_{-1,j}$ should be the smallest number of operations turning $s[0..-1]$ (the empty string) into $t[0..j]$. Since we can turn the empty string into $t[0..j]$ by inserting the $j+1$ symbols of t and $j+1$ operations are necessary, $d_{-1,j}$ has to be $j+1$ and similarly, $d_{i,-1} = i+1$.

Here then is the table of distances for the words s = "presto", and t = "peseta" including row -1:

		−1	0	1	2	3	4	5
			p	e	s	e	t	a
−1		0	1	2	3	4	5	6
0	p	1	0	1	2	3	4	5
1	r	2	1	1	2	3	4	5
2	e	3	2	1	2	2	3	4
3	s	4	3	2	1	2	3	4
4	t	5	4	3	2	2	2	3
5	o	6	5	4	3	3	3	3

Let us verify $d_{3,2}$. Since $s[3] = t[2]$, the value of $d_{3,2}$ is the minimum of $d_{2,2}+1 = 3$, $d_{2,1} = 1$, and $d_{3,1}+1 = 3$, which is 1. As a second example, consider $d_{5,3}$. Here $s[5] \neq t[3]$, so $d_{5,3}$ is the minimum of $d_{4,3}+1 = 3$, $d_{4,2}+1 = 3$, and $d_{5,2}+1 = 4$, which is 3.

The table tells us more than just the distances. Going back from $d_{5,5}$ we can come up with a list of three operations that turn "presto" into "peseta". The value of $d_{5,5}$ is 3 because $d_{4,4}$ is 2. Hence, the last operation was replacing "o" at position 5 by "a". In turn, $d_{4,4}$ is 2 because $d_{3,3}$ is 2 (no operation). Next, $d_{3,3}$ equals 2 because $d_{3,2}$ equals 1 (insert "e" at position 3). The value of $d_{3,2}$ traces back to $d_{2,1}$ (no operation), which in turn goes back to $d_{1,0}$ (no operation). The value of $d_{1,0}$ is determined by $d_{0,0}$ (delete "r" at position 1), which in turn goes back to $d_{-1,-1}$ (no operation). Reading backwards tells us how to turn "presto" into "peseta": Delete "r" at position 1 to get "pesto", insert "e" at position 3 to get "peseto", replace "o" at position 5 to get "peseta". □

Algorithm 9.5.7 Edit-Distance. This algorithm returns the edit distance between two words s and t.

Input Parameters: s, t
Output Parameters: None

```
edit_distance(s, t) {
  m = s.length
  n = t.length
  for i = −1 to m − 1
    dist[i, −1] = i + 1 // initialization of column −1
  for j = 0 to n − 1
    dist[−1, j] = j + 1 // initialization of row −1
  for i = 0 to m − 1
    for j = 0 to n − 1
      if (s[i] == t[j])
        dist[i, j] = min(dist[i − 1, j − 1], dist[i − 1, j] + 1,
                         dist[i, j − 1] + 1)
      else
        dist[i, j] = 1 + min(dist[i − 1, j − 1], dist[i − 1, j], dist[i, j − 1])
  return dist[m − 1, n − 1]
}
```

Theorem 9.5.8. *The algorithm edit_distance correctly computes the distance between two strings in time $O(mn)$.*

Proof. We have already verified correctness. The running time of $O(mn)$ is caused by the two nested loops. Preprocessing takes time $O(m+n)$ only. ■

Best Approximate Match

In text searching, we are often interested in finding approximate rather than exact matches. In the general problem, we are given a pattern p and a text t, and we try to find a part of the text t that is as similar to p as possible. A solution depends on the measure of similarity employed. If we use edit distance, the problem becomes:

> Given a pattern p and a text t, find a subword $w = t[i..j]$ of t such that $\text{dist}(p, w)$ is minimal.

The word w will then be considered an approximate match, and $\text{dist}(p, w)$ is a measure of how good the match is.

Example 9.5.9. The best match for "retrieve" in "retreive, retreeve, retreev" is the subword "retreeve" starting at position 10, which has a distance 1 (replace i by e). □

A simple approach to finding the best approximate match would be to compute the edit distance between p and all subwords of t and look for the smallest one. Let $m = |p|$, and $n = |t|$. Since there are n^2 subwords of t and comparing each to p takes time $O(mn)$, this approach takes time $O(mn^3)$.

This simple approach violates the basic tenet of dynamic programming, namely not to recompute information that has already been computed. Consider running the edit-distance algorithm on p and t. The algorithm computes values $d_{i,j} = \text{dist}(p[0..i], t[0..j])$, the edit distances between $p[0..i]$ and $t[0..j]$, for all $0 \le i < m$ and $0 \le j < n$. In effect, it gives us the edit distance of p from all prefixes of t. This is already very close to what we need, the difference being that we are computing the edit distance between p and all $t[0..j]$ instead of also allowing suffixes of $t[0..j]$.

Let us define

$$ad_{i,j} = \min\{\text{dist}(p[0..i], t[\ell..j]) \mid 0 \le \ell \le j+1\};$$

that is, $ad_{i,j}$ is the smallest edit distance between $p[0..i]$ and a subword of t ending in j (possibly the empty string if $\ell = j+1$). The value we are looking for, then, is the minimum of $ad_{m-1,0}, ad_{m-1,1}, \ldots, ad_{m-1,n-1}$. The new array $ad_{i,j}$ looks more complicated than $d_{i,j}$, but it can actually be computed using the same formula as for d:

$$ad_{i,j} = \min \left\{ \begin{array}{l} ad_{i-1,j-1} + \left\{ \begin{array}{ll} 0 & \text{if } p[i] = t[j] \\ 1 & \text{else} \end{array} \right. \\ ad_{i-1,j} + 1 \\ ad_{i,j-1} + 1 \end{array} \right\}.$$

The formula can be justified using the same argument we used for $d_{i,j}$. If $p[i] = t[j]$, then we can extend the match found by $ad_{i-1,j-1}$. If $p[i] \neq t[j]$, then there are three possibilities of how the last letter of $p[0..i]$ and some $t[\ell..j]$ matched: replacing $p[i]$ by $t[j]$, deleting $p[i]$, or inserting $t[j]$ at the end of p. Since these are all possibilities, the formula is correct.

The computation of the ad array differs from the computation of the d array in the initialization. In the definition of $ad_{i,j}$, as an extreme case we allow $p[0..i]$ to be reduced to the empty string. Hence, $ad_{-1,j} = 0$ for all j (because the empty string is a substring of any string at any position j).

Algorithm 9.5.10 Best Approximate Match. This algorithm returns the smallest edit distance between a pattern p and a subword of a text t.

Input Parameters: p, t
Output Parameters: None

```
best_approximate_match(p, t) {
  m = p.length
  n = t.length
  for i = −1 to m − 1
    adist[i, −1] = i + 1 // initialization of column −1
  for j = 0 to n − 1
    adist[−1, j] = 0 // initialization of row −1
```

```
    for i = 0 to m − 1
      for j = 0 to n − 1
        if (p[i] == t[j])
          adist[i, j] = min(adist[i − 1, j − 1], adist[i − 1, j] + 1,
                            adist[i, j − 1] + 1)
        else
          adist[i, j] = 1 + min(adist[i − 1, j − 1], adist[i − 1, j],
                                adist[i, j − 1])
    differences = m
    for j = 0 to n − 1
      if (adist[m − 1, j] < differences)
        differences = adist[m − 1, j]
    return differences
}
```

As in the case of *edit_distance*, we have the following theorem.

Theorem 9.5.11. *The algorithm best_approximate_match correctly computes the smallest distance between p and a subword of t in time $O(mn)$.*

Example 9.5.12. Let us use *best_approximate_match* to search for "pierce" (misspelled) in the list "james, peirce, dewey":

		−1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
			j	a	m	e	s	,	p	e	i	r	c	e	,	d	e	w	e	y
−1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	p	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	i	2	2	2	2	2	2	2	1	1	1	2	2	2	2	2	2	2	2	2
2	e	3	3	3	3	2	3	3	2	2	2	2	3	2	3	3	2	3	2	3
3	r	4	4	4	4	3	4	4	3	3	3	2	3	3	3	4	4	4	3	3
4	c	5	5	5	5	4	4	5	4	4	4	3	2	3	4	4	5	5	4	4
5	e	6	6	6	6	5	5	5	5	4	5	4	3	2	3	4	4	5	5	5

The smallest value in the last row is 2, and we therefore have a match for "pierce" of that distance. Tracing the 2 back, we find out that the match is "peirce" in positions 6 to 11. □

Call a match w in t a *k-approximate match for p*, if $\text{dist}(w, p) \leq k$. We can ask the question of whether, given p and t, a k-approximate match exists. Algorithm 9.5.10 solves the problem in time $O(mn)$. Using a data structure called *suffix trees*, some clever preprocessing improves the time to $O(kn)$.

Searching with Don't-Cares

For the purpose of this section, a pattern may contain—in addition to the letters from the alphabet—a special character, "?", called the don't-care symbol. We assume that "?" does not belong to Σ. For such a pattern p, a **(don't-care) match** in a text t is a position i such that $p[j] = t[i + j]$ or $p[j] =$ "?" for all $0 \leq j < |p|$.

In this section we present an algorithm due to Pinter to find such a match. This algorithm builds on an algorithm by Aho and Corasick, which in turn extends the Knuth-Morris-Pratt search. Aho and Corasick's algorithm solves the *multiple pattern* search problem: Given a text t and a set $P = \{p_1, \ldots, p_k\}$ of patterns (without the don't-care symbol), find all occurrences in t of every pattern in P. A simple solution would be to run the modified Knuth-Morris-Pratt algorithm from Exercise 9.10 separately for each pattern, which would give us a running time of $O(m + kn)$ where $m = |p_1| + \cdots + |p_k|$. For long texts this becomes too expensive. Aho and Corasick's algorithm solves the same problem in time $O(m + n)$ by looking at the set of patterns as a tree of patterns and implementing a shift table for that tree. Without going into further detail, we will just assume that we have an algorithm running in time $O(m + n)$ solving the multiple pattern search problem.

We return to the problem of matching don't-care patterns. We first split the pattern p into largest subpatterns not containing don't-care symbols.

Example 9.5.13. Consider the pattern "r?ss?ll". This pattern splits into three subpatterns: "r", "ss", and "ll". □

The splitting gives us a set of normal patterns $P = \{p_1, \ldots, p_k\}$, where p_j starts at index ℓ_j in p. Suppose t contains a don't-care match for p starting at position i. That means we have a normal match of p_j at position $i + \ell_j$ in t for $1 \leq j \leq k$.

Example 9.5.14. Assume we are searching the text "llssellrissulliss" for the pattern "r?ss?ll". $P = \{$"r", "ss", "ll"$\}$, and $\ell_1 = 0$, $\ell_2 = 2$, and $\ell_3 = 6$. The match of "r?ss?ll" starting at position 7 of the text corresponds to matches of "r" at position $7 + \ell_1 = 7$, "ss" at position $7 + \ell_2 = 9$, and "ll" at position $7 + \ell_3 = 13$. □

Let us reverse the viewpoint. Suppose that we find a match of pattern p_j starting at position i in t. This pattern potentially contributes to a don't-care match of p starting at position $i - \ell_j$. The idea of the algorithm now is to keep a counter $c[i]$ for each position i and increase the counter by one for each pattern p_j that contributes to a don't-care match of p starting in i. If there is a don't-care match of p starting in i, then $c[i]$ is increased k times, once for each p_j. Since the same pattern p_j cannot increase the same counter twice, a counter can become k only if all k patterns are present with the correct offset to form a don't-care match of p. In short, $c[i] = k$ if and only if there is a don't-care match of p starting at position i.

Algorithm 9.5.15 Don't-Care-Search. This algorithm searches for an occurrence of a pattern p with don't-care symbols in a text t over alphabet Σ. It returns the smallest index i such that $t[i + j] = p[j]$ or $p[j] =$ "?" for all j with $0 \leq j < |p|$, or -1 if no such index exists.

Input Parameters: p, t
Output Parameters: None

```
don't_care_search(p, t) {
  m = p.length
  k = 0
  start = 0
  for i = 0 to m
    c[i] = 0
  // compute the subpatterns of p, and store them in array sub
  for i = 0 to m
    if (p[i] =="?") {
      if (start ! = i) {
        // found the end of a don't-care free subpattern
        sub[k].pattern = p[start..i − 1]
        sub[k].start = start
        k = k + 1
      }
      start = i + 1
    }
  if (start ! = i) {
    // end of the last don't-care free subpattern
    sub[k].pattern = p[start..i − 1]
    sub[k].start = start
    k = k + 1
  }
  P = {sub[0].pattern, ..., sub[k − 1].pattern}
  aho_corasick(P, t) // we assume this algorithm is implemented
  for each match of sub[j].pattern in t starting at position i {
    c[i − sub[j].start] = c[i − sub[j].start] + 1
    if (c[i − sub[j].start] == k)
      return i − sub[j].start
  }
  return −1
}
```

The preprocessing in the algorithm takes time $O(m)$ (for determining the subpatterns) and $O(m + n)$ for running the Aho-Corasick algorithm. In each step of the main loop, a counter gets increased. There are n counters, each of which can be at most k. Hence, the body of the loop is performed at most kn times. Overall this gives us a running time of $O(m + kn)$. In case k is bounded (if we allow a limited number of don't-care symbols only, for example), then the running time is $O(m + n)$. Otherwise it could be as large as $O(mn)$. No better algorithms are currently known if k is not bounded.

Theorem 9.5.16. *The algorithm don't_care_search correctly performs a search with don't-cares in time $O(m + kn)$.*

Proof. We have already analyzed the running time of the algorithm. The correctness follows from the observation that $c[i]$ is k if and only

if $sub[j].pattern$ matches t starting at position $i + sub[j].start$ for all $1 \leq j \leq k$. ■

The reason we were using Aho-Corasick was that it performed in time $O(m+n)$ rather than $O(m+kn)$ (otherwise we could have just used Knuth-Morris-Pratt). Now it seems that this was unnecessary since *don't_care_search* ends up taking time $O(m + kn)$. However, the bottleneck in *don't_care_search* is incrementing the counters. Aho-Corasick in each step may find multiple matches of strings. If we can increase the counters for these matches simultaneously in constant time (on a parallel machine, for example), *don't_care_search* takes time $O(m + n)$ only.

Exercises

1S. Determine the edit distance between "center" and "centre" by computing the table of $d_{i,j}$ using *edit_distance*.

2. Determine the edit distance between "photographer" and "phonograph" by computing the table of $d_{i,j}$ using *edit_distance*.

3. Determine the edit distance between "banana" and "antenna" by computing the table of $d_{i,j}$ using *edit_distance*.

4S. Trace *best_approximate_match* to find the best match for "nite" in "tonight".

5. Trace *best_approximate_match* to find the best match for "specter" in "parerga et paralipomena".

6. Trace *best_approximate_match* to find the best match for "fudge" in "praeludium and fugue".

7S. Extend *best_approximate_match* to print the best approximate match it finds and the position of that match.

8. Write an algorithm that takes as input the matrix computed by *best_approximate_match* (like the matrix in Example 9.5.12), and from that matrix determines a best approximate match.

9S. Write a don't-care pattern for time and date.

10. Write a don't-care pattern for telephone numbers with area codes (assume the area code is within parentheses and groups are separated by dashes).

11. Write a don't-care pattern for IP-address, assuming all four blocks consist of three digits.

12S. Can appending a "?" to the end of a pattern change the results of a don't-care search?

13S. Trace *don't_care_search* on pattern "01?10" and text

"0110011110101010".

14. Trace *don't_care_search* on pattern "(?)" and text "(12),(2),(14)".

15. Trace *don't_care_search* on pattern "(??/??)" and text

"(1/99),(12/01),(10/8)".

16S. Trace *don't_care_search* on pattern "ind???nd?nt" and text

"Find, indented, independently."

Do not ignore spaces in the text.

9.6 Regular Expression Matching

Regular expressions are a very powerful technique for finding patterns with a given structure. In Perl (Practical Extracting and Reporting Language), for example, we could write

if ($t =~ /\d{3}-\d{4}/)...

to check whether variable t contains a local telephone number. The regular expression between the two forward slashes tells Perl that we are looking for three digits, followed by a dash, followed by another four digits. Most programming languages support regular expressions, including scripting languages such as JavaScript and VBScript for client- and server-side form validation and data extraction. The following JavaScript code could be used for verifying whether the variable *ccn* contains a correctly formatted American Express credit card number:

```
ccn.replace(/(\s|-)/,""); // remove spaces and dashes
valid = true;
if (!(ccn.match(/^3(4|7)\d{13}$/)))
   valid = false;
return valid;
```

The regular expression checks that *ccn* starts (^) with a 3 followed by either a 4 or a 7 followed by another thirteen digits before it ends ($).

Regular expressions as implemented by Perl and other programming languages are very rich, allowing features such as back-referencing in which a partial match of the pattern can be reused within the same pattern. Most of these features are there to simplify the pattern writing; but some features, such as back-referencing, make the problem more difficult. In this section we restrict ourselves to the three central constructs of regular expressions:

concatenation, alternation, and **repetition** (to be explained shortly). As basic symbols, we allow only symbols over some fixed alphabet Σ although implementing collections of symbols like $\backslash d$ for the class of digits would not be difficult. We assume that the symbols "$\cdot$" (concatenation), "|" (alternation), and "$*$" (repetition, called the *Kleene star*) are not part of the alphabet Σ, and neither are the parentheses "(", ")", which we use for grouping purposes.

The simplest pattern is a *concatenation* of letters, for example "html". We should really write "h$\cdot$t$\cdot$m$\cdot$l" to stress the fact that we are concatenating the letters, but we use the usual convention of dropping the "$\cdot$" when writing the pattern. It is understood implicitly.

Alternation allows us to offer options. For example, the pattern "jpg|gif" would match both "jpg" and "gif". If we wanted a pattern for image file extensions, we could use either ".jpg|.gif", or ".(jpg|gif)".[†] We use parentheses for grouping patterns (in this case to restrict the scope of the alternation symbol "|").

Finally the Kleene star allows us to match a pattern repeatedly. For example, "0$*$" matches any sequence of zeros (including the empty one). "01 $*$ 0" matches any word that starts with a zero, ends with a zero, and has any number of ones between the two zeros.

Example 9.6.1. A pattern for binary strings would be "(0|1)$*$". Patterns are not unique. For example, "0 $*$ (0|1)$*$" is also a pattern for binary strings. The pattern

$$\text{"\#(0|1|2|3|4|5|6|7|8|9|}A\text{|}B\text{|}C\text{|}D\text{|}E\text{|}F)*\text{"}$$

is matched by hexadecimal numbers with an initial "#". □

Regarding precedence, note that $*$ binds more strongly than concatenation, and concatenation binds more strongly than alternation.

Example 9.6.2. "01$*$" describes all strings starting with a zero followed by any number of ones. The Kleene star applies only to "1", since it binds stronger than the concatenation between "0" and "1". If we wanted to describe strings that consist of arbitrarily long repetitions of "01", we would use "(01)$*$".

We saw earlier that concatenation binds more strongly than alternation. That is why ".jpg|.gif" matches both ".jpg" and ".gif". If alternation had precedence over concatenation, the pattern would match both ".jpggif" and ".jp.gif"; if that was our intention, we would write ".jp(g|.)gif". □

We need a more formal definition of what it means for a word w to match a pattern p. We define $L(p) \subseteq \Sigma^*$ as the set of all words over the alphabet Σ that match p by induction over the structure of p. There are four cases:

[†] For a regular expression in Perl or Javascript, we would have to escape the period, "\.(jpg|gif)", because it is a special character.

p ="a", for some $a \in \Sigma$ $L(p) = \{\text{"}a\text{"}\}$,
p ="$q_1|q_2$" $L(p) = L(q_1) \cup L(q_2)$,
p ="$q_1 \cdot q_2$" $L(p) = \{w_1 \cdot w_2 \mid w_1 \in L(q_1), w_2 \in L(q_2)\}$,
p ="$q*$" $L(p) = \{w_1 \cdot w_2 \cdots w_n \mid w_1, \ldots, w_n \in L(q), n \in \mathbb{N}\}$.

We say a text t **matches a pattern** p, if $t \in L(p)$. Patterns can be *ambiguous*; that is, there might be more than one way that a text can match a pattern as illustrated by the pattern "$(a*|ab)b*$" and the text "abb".

We work with a binary tree representation of the pattern. As an example, a binary tree representation of the pattern "$1(1*|0*)1$", or "$1 \cdot (1* | 0*) \cdot 1$" to be more precise, is shown in Figure 9.6.1.

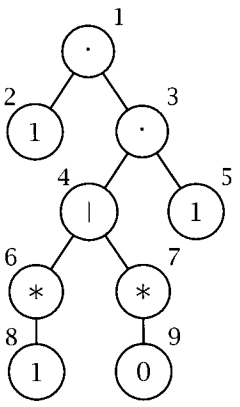

Figure 9.6.1 A binary tree representation of the pattern "$1(1*|0*)1$".

Each internal node contains an operation ($\cdot$, $|$, or $*$) and each leaf contains an element from Σ. The operators $\cdot$ and $|$ are binary and $*$ is unary. Different binary trees can represent the same pattern (see Exercise 9.44).

A node in the tree determines a subpattern of the original pattern, namely the pattern described by the tree rooted in the node.

Example 9.6.3. Consider the binary tree representation of "$1(1*|0*)1$" in Figure 9.6.1. Node 1 corresponds to the full pattern, which is the concatenation of the pattern "1" rooted in node 2 and "$(1*|0*)1$" rooted in node 3. Node 4 corresponds to "$(1*|0*)$", and node 7 corresponds to "$0*$". □

Turning the regular expression into a binary tree requires some knowledge of parsing theory, a specialized task in the area of compiler construction. Exercise 9.43 asks for such an algorithm.

Before we attack the full problem of finding a match for a regular expression in a text, let us first see how to use the tree to find out whether a word w matches a pattern p, or, more formally, whether $w \in L(p)$.

The main idea is to keep track of which letters can be matched next. Consider, for example, the pattern "$0(0|1)*0$". A corresponding tree is shown in Figure 9.6.2.

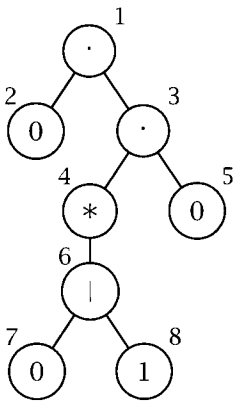

Figure 9.6.2 A binary tree representation of the pattern "0(0|1) ∗ 0".

Initially we expect to see the 0 in node 2. A 1 would lead to immediate failure. If w starts with a 0, we have matched the left subpattern of node 1, and we continue with the subpattern of node 3. The next symbol we expect could be any of the symbols in nodes 7, 8, and 5. We have to include node 5 because ∗ does allow multiplicity zero. In case w ="00", the second zero has to be the zero of node 5. If the second letter of w is a 1, we know we are in node 8. After that, again nodes 7, 8, and 5 are possible. If the second letter is a zero, we could be in node 7 or 5. Again, the next possible nodes are 7, 8, and 5. And so we continue. If we finish with the last letter of w matching node 5, then we know we have found a match. Otherwise, we know that w does not match because we considered all possibilities of how the pattern could match the word. We can therefore rephrase our basic idea more precisely: Keep track of the leaves of the pattern that can be matched next. We call these leaves the current **candidates**. With each new letter of w, we update the set of candidates to the set of leaves we could possibly match next. We still need to clarify some points. How do we find the candidates to start the process with? There might be several.

Example 9.6.4. The pattern "(0|1) ∗ 0" initially has three candidates for the first letter to match. Figure 9.6.3 shows the binary tree corresponding to the pattern. All leaves, nodes 5, 6, and 3, could be matched by the first letter of a text, and therefore make up the set of initial candidates. If w ="00", then the first zero is matched by node 5; if w ="10", the first "1" is matched by node 6, and, finally, if w ="0", then the zero is matched by node 3. □

We compute the initial set of candidates in a procedure *start*. Another procedure *match_letter* goes through the tree and determines which of the current candidates actually matches the next letter of w; it rejects the ones that do not match. After that, we apply the procedure *next_cand* that uses the information compiled by *match_letter* to compute the new set of candidates.

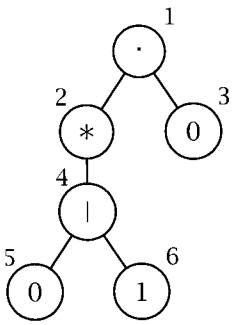

Figure 9.6.3 A binary tree representation of the pattern "$(0|1) * 0$".

In a simple implementation of these procedures, *next_cand* might take time $\Theta(n^2)$ because there could be $\Theta(n)$ candidates, each spawning $\Theta(n)$ new candidates. However, there are only n leaves to the tree, which suggests that a more sophisticated approach might lead to a better result. In fact, all three procedures can be implemented to run in time $O(n)$. To make this work, we need to precompute some information on the tree.

In Example 9.6.4 we had to allow node 3 as an initial candidate although it is the second part of a concatenation (node 1). The reason was that the pattern "$(0|1)*$", which corresponds to node 1, could match the empty word (because of the Kleene star in node 2). That is, to make the decision that node 3 had to be included in the initial set of candidates, we had to know that node 2 could match the empty string. We therefore store in each node of the tree a Boolean attribute to indicate whether the pattern corresponding to that node could match the empty word. Here is an algorithm using divide and conquer to compute this attribute that we call *eps* (for ε, the empty word). In practice we would compute this attribute while building the pattern tree.

Algorithm 9.6.5 Epsilon. This algorithm takes as input a pattern tree t. Each node contains a field *value* that is either $\cdot$, $|$, $*$ or a letter from Σ. For each node, the algorithm computes a field *eps* that is true if and only if the pattern corresponding to the subtree rooted in that node matches the empty word.

Input Parameter: t
Output Parameters: None

```
epsilon(t) {
  if (t.value == "·")
    t.eps = epsilon(t.left) && epsilon(t.right)
  else if (t.value == "|")
    t.eps = epsilon(t.left) || epsilon(t.right)
  else if (t.value == "*") {
      t.eps = true
      epsilon(t.left) // assume only child is a left child
  }
```

```
    else
        // leaf with letter in Σ
        t.eps = false
}
```

Example 9.6.6. If we run *epsilon* on the pattern "$0(0 * |1)1$", we obtain the result shown in Figure 9.6.4.

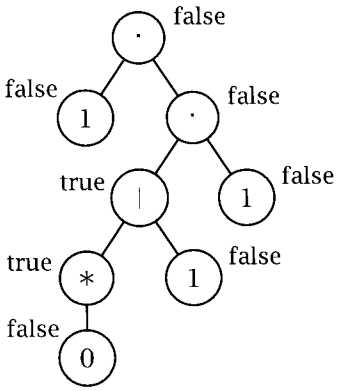

Figure 9.6.4 A binary tree representation of the pattern "$0(0*|1)1$" together with the values of the *eps* attribute. □

With the help of the *eps* attribute, we can now compute the initial set of candidates (let us call them the *start positions*) in the tree. Again we use divide and conquer to traverse the tree. If the node is an alternation node, the start positions are the start positions of both subtrees (the match could start in either one). If the node is a multiplicity node, the start positions are the start positions in its subtree. If the node is a concatenation node, the start positions are the start positions of the left subtree. If the left subtree corresponds to a pattern that could be ε, we also need to add the start positions of the right subtree (remember Example 9.6.4). This gives us the following algorithm to compute the initial set of candidates.

Algorithm 9.6.7 Initialize Candidates. This algorithm takes as input a pattern tree t. Each node contains a field *value* that is either $\cdot$, $|$, $*$ or a letter from Σ and a Boolean field *eps*. Each leaf also contains a Boolean field *cand* (initially false) that is set to true if the leaf belongs to the initial set of candidates.

Input Parameter: t
Output Parameters: None

```
start(t) {
  if (t.value == "·") {
       start(t.left)
       if (t.left.eps)
          start(t.right)
  }
  else if (t.value == "|") {
       start(t.left)
       start(t.right)
  }
  else if (t.value == "*")
       start(t.left)
  else
     // leaf with letter in Σ
     t.cand = true
}
```

Example 9.6.8. For the pattern "$0(0 * |1)1$" of Example 9.6.6, there is only one starting position, as shown in Figure 9.6.5.

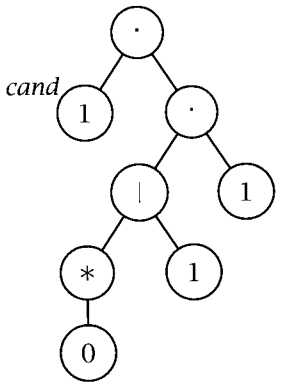

Figure 9.6.5 Only one starting position for the pattern "$0(0 * |1)1$".

The parse tree for the pattern "$(0|1) * 0$", as shown in Figure 9.6.3, contains three starting positions: 5, 6, and 3. □

The *match_letter* procedure takes as input the tree t and the next letter a of the pattern to be processed. It prepares the tree for updating by the *next_cand* procedure by computing a Boolean attribute *matched* for each inner node of the tree that is true if the letter a successfully concludes a matching of the pattern corresponding to that node.

Example 9.6.9. We saw that the starting positions for the pattern "$(0|1) * 0$" (referring to the parse tree in Figure 9.6.3) are 5, 6, and 3. Suppose the next

letter read is a 1. Since node 6 was a candidate, the pattern rooted at node 6, namely "1", is matched. So is the pattern at node 4, "(0|1)", and 2, "(0|1)∗". Similarly, reading the letter 0 would conclude matchings of the pattern "0" in node 5, the pattern "(0|1)" in node 4, the pattern "(0|1)∗" in node 2, the pattern "0" in node 3, and the full pattern, "(0|1) ∗ 0", in node 1. □

If *matched* is true for the root of the tree, we have found a match for the pattern. As before, we determine the *matched* attribute using divide and conquer. We have found a match for a concatenation node if we have completed matching its right child. An alternation node is matched if either of its children is matched, and a repetition node is matched if its child is matched. Finally, a leaf is matched if the value of the leaf (the corresponding letter in the pattern) is the same as the letter a, and the leaf is a candidate.

Algorithm 9.6.10 Match Letter. This algorithm takes as input a pattern tree t and a letter a. It computes for each node of the tree a Boolean field *matched* that is true if the letter a successfully concludes a matching of the pattern corresponding to that node. Furthermore, the *cand* fields in the leaves are reset to false.

Input Parameters: t, a
Output Parameters: None

```
match_letter(t, a) {
  if (t.value == "·") {
      match_letter(t.left, a)
      t.matched = match_letter(t.right, a)
  }
  else if (t.value == "|")
      t.matched = match_letter(t.left, a) ||match_letter(t.right, a)
  else if (t.value == "*")
      t.matched = match_letter(t.left, a)
  else {
    // leaf with letter in Σ
    t.matched = t.cand && (a == t.value)
    t.cand = false
  }
  return t.matched
}
```

Example 9.6.11. We return to the pattern "0(0|1) ∗ 0" whose pattern tree is shown in Figure 9.6.2. The initial configuration of the tree, after running *epsilon* and *start*, is shown in Figure 9.6.6(a). If we run *match_letter* on the tree with "0" as the input letter, only the leaf node 2 is matched (since it was the only candidate). □

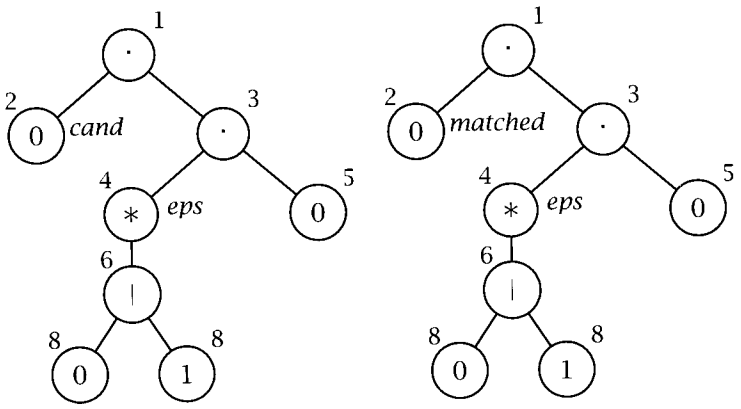

Figure 9.6.6 (a) The pattern tree of "$0(0|1) * 0$" after running *epsilon* and *start*. Instead of displaying the values of all of the Boolean attributes *eps*, *matched*, and *cand*, we display the name of the attribute at the nodes for which the attribute is true. (b) The pattern tree from (a) after running *match_letter*(t,"0") on it.

With the *matched* information, we can now compute the new set of candidates. We implement the *next* algorithm with two inputs, the tree t, and a Boolean value *mark*, which allows us to specify whether start positions of the whole pattern should also be marked as candidates (in effect rerunning the *start* procedure). The second parameter allows us to compute the set of new candidates as the starting positions of appropriate subpatterns.

Example 9.6.12. We return to the pattern "$(0|1) * 0$" and the corresponding pattern tree shown in Figure 9.6.3. Suppose the first letter of the text we read is a 1. The starting positions of the pattern are 5, 6, and 3. Hence the subpattern rooted in node 2 is matched. The new set of candidates then contains nodes 5 and 6, since node 2 is a Kleene star, and node 3, since node 2 was matched. □

We recursively run *next* on the whole tree. If the left child of a concatenation node has been matched, we select all of the start vertices of the right child as candidates. Also, if a repetition node has been matched, then all the start vertices of the left child are candidates.

As before, the start vertices of an alternation are the start vertices of both children nodes; the start vertices of a concatenation are the start vertices of the left child and, if the left child matches the empty pattern, the start vertices of the right node. The start vertices of a repetition node are the start vertices of its child.

Algorithm 9.6.13 New Candidates. This algorithm takes as input a pattern tree t that is the result of a run of *match_letter*, and a Boolean value *mark*.

It computes the new set of candidates by setting the Boolean field *cand* of the leaves.

```
 Input Parameters:  t,mark
Output Parameters:  None

next(t, mark) {
  if (t.value == "·") {
    next(t.left, mark)
    if (t.left.matched)
      next(t.right, true) // candidates following a match
    else if (t.left.eps) && mark)
      next(t.right, true)
    else
      next(t.right, false)
  else if (t.value == "|") {
    next(t.left, mark)
    next(t.right, mark)
  }
  else if (t.value == "*")
    if (t.matched)
      next(t.left, true) // candidates following a match
    else
      next(t.left, mark)
  else
    // leaf with letter in Σ
    t.cand = mark
}
```

We apply the *next* procedure by calling *next*$(t,$ false$)$ on the tree t whose candidates we want to compute.

Example 9.6.14. We continue Example 9.6.11 for pattern "$0(0|1) * 0$". We test the pattern against the text "010". Figure 9.6.6 (b) showed the pattern tree after the application of *start* and *match_letter* for the letter "0". Figure 9.6.7 (a) shows the result of running *next* on that tree, computing the new set of candidates. We then run *match_letter* for the letter "1". Nodes 4, 5, and 6 are matched as shown in Figure 9.6.7(b). Updating the tree with *next* yields nodes 7, 8, and 5 as candidates [Figure 9.6.7(c)]. Running *match_letter* for letter "0" then yields Figure 9.6.7(d). Nodes 1, 3, 5, and 7 are matched. In particular, since node 1 is matched, we can conclude that "010" matches the pattern "$0(0|1) * 0$", which, indeed, it does. □

We see that we could now eliminate the *start* algorithm, since it corresponds to a call of *next*$(t,$ true$)$ because initially all of the *matched* attributes are false.

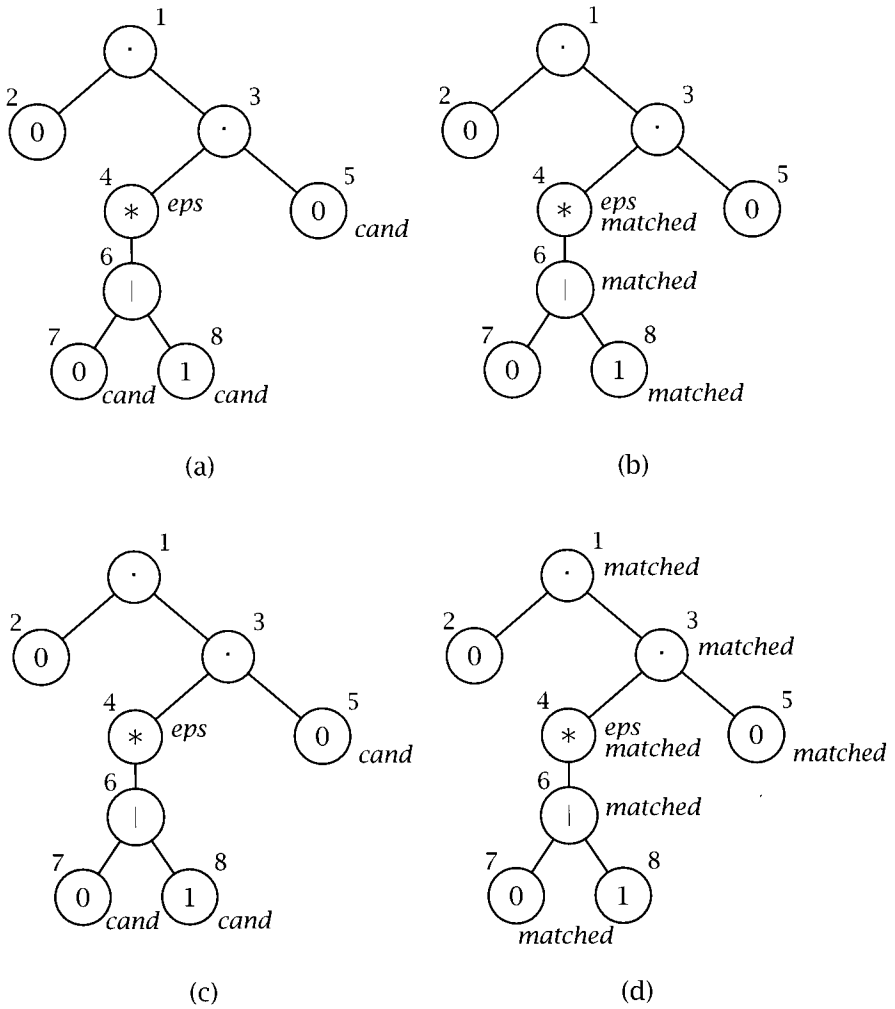

Figure 9.6.7 Matching the text "010" against the pattern "$0(0|1) * 0$". The result of running *next* on the tree in Figure 9.6.6 (b) is shown in (a). Running *match_letter* for letter "1" on the tree from (a) yields the tree in (b). In turn, running *next* on the tree from (b) yields (c), and running *match_letter* for letter "0" on (c) gives us (d).

With these algorithms, we can finally implement the actual algorithm that tests whether a word w matches a regular expression p given as a tree with root t.

Algorithm 9.6.15 Match. This algorithm takes as input a word w and a pattern tree t and returns true if a prefix of w matches the pattern described by t.

```
 Input Parameters:  w, t
Output Parameters:  None

match(w) {
  n = w.length
  epsilon(t)
  start(t)
  i = 0
  while (i < n) {
    match_letter(t, w[i])
    if (t.matched)
      return true
    next(t, false)
    i = i + 1
  }
  return false
}
```

Examples 9.6.11 and 9.6.14 trace this algorithm on pattern "$0(0|1) * 0$" and text "010".

What is the run-time of this algorithm? The subprocedures *match*, *start*, and *match_letter* all take time $\Theta(m)$, where $m = |p|$, since they each traverse the pattern tree exactly once. Hence *match* takes time $O(mn)$.

However, *match* does not solve the problem we originally set out to settle, which was finding a match for a regular expression pattern in a piece of text. At first we might think of using *match* at each position of a piece of text. In the worst case this takes time $O(mn^2)$. We can do better than that, and only a simple modification to *match* is required. We need to substitute the call to *next*$(t,$ false$)$ by *next*$(t,$ true$)$, the idea being that in each step we not only continue matching the pattern but also restart the pattern by including all of its potential starting positions. The following algorithm implements these ideas.

Algorithm 9.6.16 Find. This algorithm takes as input a text s and a pattern tree t and returns true if there is a match for the pattern described by t in s.

```
 Input Parameters:  s, t
Output Parameters:  None

find(s, t) {
  n = s.length
  epsilon(t)
  start(t)
  i = 0
  while (i < n) {
    match_letter(t, s[i])
    if (t.matched)
      return true
```

```
        next(t, true)
        i = i + 1
    }
    return false
}
```

The *find* algorithm runs in time $O(mn)$ as does *match*. The reason for the product is that for each letter from the text we might have to update as many as $\Omega(m)$ nodes in the pattern tree; that is, we might have to recompute $\Omega(m)$ candidates in each step. Let us have another look at what *match_letter* and *next* actually do. They take a set of candidates and a letter and compute a new set of candidates. That is, running *match_letter* followed by *next* is a function from sets of candidates and the alphabet to sets of candidates. Furthermore, it is a function we can precompute (without seeing any text), given the alphabet and the pattern. Since the pattern has length m, there are at most m leaves in the pattern tree; hence there are at most 2^m possible sets of candidates. For each such set and every letter in Σ, we can compute the next candidate set and store the information in a $2^m \times |\Sigma|$ table.

Example 9.6.17. In Examples 9.6.11 and 9.6.14 we computed, along the way, the following transitions:

Candidate Set	*Letter*	*Candidate Set*
$\emptyset$	*"0"*	$\emptyset$
$\emptyset$	*"1"*	$\emptyset$
$\{2\}$	*"0"*	$\{7,8,5\}$
$\{2\}$	*"1"*	$\emptyset$
$\{7,8,5\}$	*"0"*	$\{7,8,5\}$
$\{7,8,5\}$	*"1"*	$\{7,8,5\}$

Since there are four leaves in this example, the table should contain $2^4 \times 2$ rows; however, no other transitions can occur in *match*. When tracing *find* we would use the following rules:

Candidate Set	*Letter*	*Candidate Set*
$\{2\}$	*"0"*	$\{2,7,8,5\}$
$\{2\}$	*"1"*	$\{2\}$
$\{2,7,8,5\}$	*"0"*	$\{2,7,8,5\}$
$\{2,7,8,5\}$	*"1"*	$\{2,7,8,5\}$

Note that we automatically added node 2 to every candidate set to restart the pattern. □

After precomputing and storing the table of transitions, we can match each letter of the text in constant time, since we only have to look up the transition in the table instead of recomputing it. Hence running *match* or *find* with this method takes time at most $O(m2^m + n)$ (assuming the alphabet has fixed size). If we compare this to the $O(mn)$ running time of our first

implementation, we see that the new implementation is advantageous if we have short patterns and long texts or if we have a single pattern that is matched against many different texts (in which case the $O(m2^m)$ steps for precomputing the transition table are well-spent).

Exercises

1S. What is $L(\text{“}0(0|1)*\text{”})$? Give a succinct answer.

2. What is $L(\text{“}(00|01|10|11)*\text{”})$? Give a succinct answer.

3. What is $L(\text{“}0*10*\text{”})$? Give a succinct answer.

4S. What is $L(\text{“}(0|10)*\text{”})$? Give a succinct answer.

5S. Write a pattern for octal numbers.

6. Write a pattern for decimal numbers.

7. Write a pattern for binary strings containing at least two ones.

8S. Write a pattern for binary strings containing at most two ones.

9. Write a pattern for binary strings with an odd number of ones.

10. Write a pattern for binary strings whose *weight* is a multiple of three (the weight of a binary string is the number of ones it contains).

11S. Write a pattern for binary strings of odd length.

⋆12. Write a pattern for binary strings that contain both an even number of ones and an even number of zeros.

13. Write a pattern for binary strings that contain the subword “111”.

14S. Write a pattern for binary strings that do not contain the subword “111”.

15. Write a pattern that matches both the American and the British spelling of theater. Make the pattern as succinct as possible.

16. On your web-form you have an input field for year of birth into which the user can enter four digits. Write a regular expression that verifies the input (only years in the range 1880 to 2004 should be accepted).

17S. Name at least five different programming languages that support regular expressions.

18. Show that no pattern is unique; that is, for every pattern p there is a pattern q different from p such that $L(p) = L(q)$.

19S. Find a binary tree representation for the pattern "$(0*|01*)(1|10*)$" and compute the *eps* attribute of each node using *epsilon*.

20. Find a binary tree representation for the pattern "$0*((0|1)(0|1*))$" and compute the *eps* attribute of each node using *epsilon*.

21. Find a binary tree representation for the pattern "$0*1*0$" and compute the *eps* attribute of each node using *epsilon*.

22S. Trace *match* on pattern "$0*1*0$" and text "0010" by drawing the parse tree for the pattern and listing the candidate sets that *match* computes.

23. Trace *match* on pattern "$0*1*0$" and text "0101" by drawing the parse tree for the pattern and listing the candidate sets that *match* computes.

24S. Trace *match* on pattern "$(01|0)(0|10)$" and text "010" by drawing the parse tree for the pattern and listing the candidate sets that *match* computes.

25. Trace *match* on pattern "$(0|(10*1))*$" and text "010010" by drawing the parse tree for the pattern and listing the candidate sets that *match* computes.

Notes

The first $O(m+n)$ algorithm for string matching was found independently by several authors around 1970. Knuth and Pratt based their algorithm on a theoretical result by Cook that showed that an $O(m+n)$ algorithm must exist. Independently of Cook's result, Morris had discovered a similar algorithm, and a joint paper by Knuth, Morris, and Pratt describing the algorithm was published in 1977, the same year that Rivest showed that $\Omega(n-m+1)$ comparisons are necessary in the worst case.

Boyer and Moore found their algorithm in 1974. It was subsequently improved (and fixed) by many authors, including Knuth. Boyer-Moore-Horspool was described in Horspool, 1980. It is used in the UNIX `agrep` command. Stephen, 1994, discusses the algorithm.

In 1985 Pinter showed how to do don't-care searching using Aho-Corasick's algorithm for multiple pattern matches. Multiple pattern matches, and the related idea of suffix trees, are covered in detail by Gusfield, 1997.

The dynamic-programming algorithm for computing the edit distance between two strings was discovered independently by many authors in the early 1970s. Sellers observed in 1980 that this method could be used to find the best approximate match of a pattern in a string by simply changing the initial values of the problem. A fair amount of effort has been made to speed up these methods and reduce the space requirements. See the books by Stephen, 1994, and Gusfield, 1997.

The survey article by Aho, 1990, is a good, and short, introduction to the main problems in the field of text searching, with particular emphasis on the complexity of the problems. There are several good textbooks on these topics. Gusfield, 1997, comprehensively deals with text algorithms useful for computational biology. The emphasis is on practical algorithms with a detailed introduction to suffix trees. Crochemore and Rytter, 1994, cover more ground, including, for example, two-dimensional pattern matching and algorithms for parallel machines, neither of which Gusfield covers. The book by Stephen, 1994, is a comprehensive reference on several topics, including exact search, string distance, and approximate pattern matching. It is particularly valuable for its historical remarks and for material on comparing different algorithms.

Chapter Exercises

⋆**9.1.** [Requires probability theory.] Show that on random patterns and text the running time of *simple_text_search* is $O(n-m)$, assuming that the alphabet is fixed and each letter independently occurs with the same probability. *Hint:* Let $s = |\Sigma|$. The probability that index j in *simple_text_search* is increased k times is at most $(1/s)^k$. Show that this implies that the expected number of increases of j is at most $(s/(s-1))^2$, and therefore the algorithm runs in time $O(n-m)$.

Exercises 9.2–9.4 investigate the actual running time of an implementation of simple_text_search.

9.2. Implement *simple_text_search* and run it on natural text inputs. Select ten different texts and ten different patterns of varying length (you can download whole books from the World Wide Web). Keep track of the running time $t(m,n)$ for the different values of m and n, and compute $t(m,n)/(m(n-m+1))$. Do your examples show worst-case behavior?

9.3. Check whether the values $t(m,n)$ you obtained from running your implementation of *simple_text_search* meet the $O(n-m)$ upper bound predicted for random inputs.

9.4. Run your implementation of *simple_text_search* systematically on randomly generated patterns and texts for a range of values for m and n. Do your experiments bear out the $O(n-m)$ upper bound?

9.5. We implemented *rabin_karp_search* on a binary alphabet. Modify the algorithm so that it works on an arbitrary (but fixed) alphabet Σ containing d letters.

WWW **9.6.** Implement *mc_rabin_karp_search* and test how many errors it makes, that is, how many false matches it lists. Test with randomly chosen texts and patterns of lengths $n = 100, 200, 1000$ and $m = 3, 10, 50$ (giving you

nine possible combinations). *Hint:* If you do not have any software to generate a random prime in the range from 2 to mn^2, pick one by hand from a list of prime numbers.

9.7. Show how to extend *rabin_karp_search* to work for patterns containing the don't-care symbol "?". *Hint:* Aim first for a $O(kn + m)$ algorithm, where k is the number of don't-care symbols in p. Then, modify the choice of q to make the algorithm work in time $O(n + m)$ for any k.

9.8. Extend *rabin_karp_search* to work with two-dimensional inputs. The input consists of two binary matrices, p of dimension $m \times m$ and t of dimension $n \times n$, and the algorithm should search for indexes $0 \leq i, j < n$ such that $p[k, \ell] = t[i+k, j+\ell]$ for all $0 \leq k, \ell < m$. What is the running time of your algorithm?

9.9. Implement *knuth_morris_pratt_search* and compare its performance to the simple text search.

9.10. Modify *knuth_morris_pratt_search* to print out all occurrences of a pattern, not just the first one. You need to be careful to keep the running time $O(m + n)$. The straightforward approach of shifting the pattern by one after a match is found leads to a running time of $\Theta(mn)$ as demonstrated by $p = 0^m$, and $t = 0^n$. This means you need to use the shift table in this case.

9.11. Implement *boyer_moore_horspool_search* and compare its performance to the simple text search.

9.12. Implement *boyer_moore_horspool_search* and compare its performance to the Knuth-Morris-Pratt search.

Exercises 9.13–9.15 contain the proof that boyer_moore_horspool_search is correct.

9.13. Show that *boyer_moore_horspool_search* correctly computes the shift of each letter; that is, $shift[w] = m - 1 - \max\{i < m - 1 \mid p[i] = w\}$ for all letters w that occur in $p[0..m-2]$, and $shift[w] = m$ for all other letters.

9.14. Find a loop invariant for the inner loop in *boyer_moore_horspool_search* and use it to show that the algorithm works correctly. *Hint:* Use

(a) $p[j + 1..m - 1] = t[i + j - 1..i + m - 1]$ and

(b) $p[0..m - 1] \neq t[k..k + m - 1]$ for $0 \leq k < i$

as the loop invariant.

9.15. Prove that the loop invariant from Exercise 9.14 is indeed a loop invariant.

9.16. Show that $\text{dist}(s, t) \leq \max\{|s|, |t|\}$ for all words s and t.

9.17. Verify that the edit distance is really a distance function (metric); that is, it fulfills

$\text{dist}(s,s) = 0$ for all $s \in \Sigma^*$ (Reflexivity),

$\text{dist}(s,t) = 0 \Rightarrow s = t$ for all $s,t \in \Sigma^*$,

$\text{dist}(s,t) = \text{dist}(t,s)$ for all $s,t \in \Sigma^*$ (Symmetry),

$\text{dist}(s,u) \leq \text{dist}(s,t) + \text{dist}(t,u)$ for all $s,t,u \in \Sigma^*$ (Triangle Inequality).

9.18. When computing the edit distance, we fill the table row by row. For computing the values in a particular row, we need only the values in the preceding row. Use this idea to modify the algorithm so it uses only linear space.

9.19. Extend the edit-distance algorithm to output a shortest sequence of operations transforming x into y. Add your code at the end of the algorithm after the full table has been computed. Your additional code should run in time $O(m+n)$. For example, s ="presto", t ="peseta" should output (see Example 9.5.5):

```
delete ''r'' at position 1
insert ''e'' at position 3
replace ''o'' by ''a'' at position 5
```

(*Note:* The problem of returning the shortest sequence of operations becomes much harder if you want to combine it with the space-saving idea of Exercise 9.18. A solution using divide-and-conquer was found by Hirschberg.)

9.20. A frequent error in written text is the transposition of adjacent letters, as in "witner" instead of "winter". Using two replaces, or one delete and an insert, assigns a cost of 2 to this error. To account for its frequency, we want to assign it unit cost. We do this by allowing an additional operation:

(T) Transpose two adjacent letters.

Rewrite the recursive formula for computing $d_{i,j}$ to account for this new operation. *Hint:* Look two cells back.

9.21. Remember the `sdiff` UNIX command from Example 9.5.1. The difference between `sdiff` and computing the edit distance is that the edit distance works over letters whereas `sdiff` works over lines (where two lines are either equal or different). Implement `sdiff` (with graphical output as in Example 9.5.1) using the algorithm for computing the edit distance. Compare two lines until you find a mismatched symbol or the lines are equal.

Exercises 9.22–9.25 refer to the game of doublets invented by Lewis Carroll. In doublets you have to turn one word into another, replacing a single letter in each step. The problem becomes interesting by insisting that the intermediary words be real words. For example, "wet" turns into "dry" in the following sequence: "wet", "bet", "bey", "dey", "dry".

9.22. Turn "tea" into "hot".

9.23. Turn "more" into "less".

9.24. Turn "raven" into "miser".

9.25. Given a set of words L (our dictionary), we can define a distance $\text{dist}_L(x, y)$ to be the length of the shortest sequence of words from L that begins with x and ends with y, such that any two adjacent words in the sequence differ in one letter only. If no such sequence exists, let the distance be ∞. Show that dist_L is a metric (for any set L). (See Exercise 9.17 for the definition of a metric.)

9.26. If we restrict the edit operations to (R), the replacing of a letter, we obtain the *Hamming distance* d_H between two texts of the same length. Show that the *Hamming distance* is a metric (see Exercise 9.17), and is computable in time $O(n)$.

Exercises 9.27–9.29 concern a variant of the edit distance sometimes known as the Levenshtein distance. *For this variant we allow only insertions and deletions. Let us call the resulting variant of the edit distance* dist_{di}. *Since a replacement can be simulated by an insertion and a deletion,* dist_{di} *corresponds to a weighted version of the edit-distance problem where we charge twice the unit cost for replacement and unit cost for deletions and insertions.*

9.27. Show that $\text{dist}(x, y) \leq \text{dist}_{di}(x, y) \leq 2\,\text{dist}(x, y)$ for all x, y.

9.28. Show that dist_{di} is a metric (see Exercise 9.17).

9.29. Let $\text{lcs}(x, y)$ be the length of the longest common subsequence of x and y (see Section 8.4). Show that

$$2\,\text{lcs}(x, y) + \text{dist}_{di}(x, y) = n + m,$$

where $n = |x|$, and $m = |y|$. [*Note:* This relationship between dist_{di} and lcs has led to an algorithm computing dist_{di} by Hunt and Szymanski that can run as fast as $O(n \log n)$.]

9.30. How would you have to modify the code of Algorithm 9.5.15, *don't_care_search*, so that it finds all occurrences of a pattern, rather than just the first one? How does this modification affect the worst-case running time?

9.31. What is the difference between the patterns "$0 * |1*$", and "$(0|1)*$"?

9.32. What is $L(\text{"}0 * 1(0|1)*\text{"})$? Give a succinct answer.

9.33. What is $L(\text{"}(0|(10 * 1))*\text{"})$? Give a succinct answer.

9.34. What is $L(\text{"}((0|1)(0|1))*\text{"})$? Give a succinct answer.

9.35. Write a pattern for binary strings that contain the subword "101".

9.36. Write a pattern for binary strings that do not contain the subword "101".

9.37. Show that a pattern of length n can contain as many as $\Omega(n)$ starting positions (the positions computed by Algorithm 9.6.7).

Exercises 9.38–9.42 concern regular expressions as implemented in programming languages such as Perl or JavaScript and editors such as vi and emacs. The syntax of regular expressions in these languages and programs is similar to the syntax introduced in Section 9.6, the operators for concatenation, alternation, and multiplicity are the same, and we can use parentheses for grouping; for example, "`(11|00)*`*" describes binary strings made up of pairs of ones and zeros. Writing regular expressions is simplified by having several classes of characters. For example, we can write "*`\d`*" for "*`(0|1|2|3|4|5|6|7|8|9)`*", that is, a digit. Similarly, "*`\w`*" is matched by a single word-character, that is, a character that can occur in a word; and "*`\s`*" is matched by any kind of space character (hardspace, tab, etc.). You can also write your own classes; for example, "*`[abc]`*" is matched by either "a", "b", or "c". A variant of this is "*`[^abc]`*", which is matched by any single character except for "a", "b", and "c". We also have several additional operations: We can use "?" to denote an optional match; for example, "*`colou?r`*" is matched by both "color" and "colour". The operator "+" is similar to the Kleene star, "*", except that it requires at least one match of the pattern; for example, "*`\d+`*" is a nonempty sequence of digits: a number. We can also require precise cardinalities using* $\{a,b\}$*, where* a *is the lower bound and* b *the upper bound on the matches required. For example, "*`\d{15,16}`*" is a 15- or 16-digit number, and "*`\w{3}`*" is a word on three letters.*

9.38. Write a regular expression pattern that is matched by both the British and the American spelling of the word "theater".

9.39. Write a regular expression pattern that is matched by a year, either in the two digit (YY) or in the four-digit format (YYYY).

9.40. Write a regular expression pattern that is matched by a time of day, as in "7am", or "12:55pm". Make sure your pattern is not matched by ill-formed times such as 32:72pm.

9.41. Write a regular expression pattern that is matched by phone numbers. *Hint:* Restrict yourself to phone numbers of a particular country. For example, US phone numbers could be written as follows: 1-(555)-123-4567 or (555)-123-4567 or 123-4567. The parentheses could be missing: 1-555-123-4567. Instead of dashes there could be spaces: 1 555 123 4567. Finally, there might be a one- to four-digit extension: 1-555-123-4567-1010.

9.42. Write a pattern that checks that if a text contains the word "Monty", it also contains the word "Python". By this specification, a text should also be accepted if it contains neither the word "Monty" nor "Python". Also, the

exercise does not specify where the words occur in the text. They do not have to be next to each other or in any particular order. *Hint:* First simplify the problem by assuming that "Monty" occurs before "Python". Also, you need the "\b" special character that matches word boundaries in the text; for example, "\bPython\b" would be matched by "Monty Python's Flying Circus" but not by "Pythonesque".

9.43. [Requires knowledge of parsing.] Write an algorithm that takes as input a regular expression and returns a binary tree representing the expression. *Hint:* Assume that the regular expression is fully parenthesized and contains all occurrences of the concatenation operator · explicitly. Write a recursive function that translates a well-formed term into a tree.

9.44. There might be different trees associated with the same regular expression pattern. Give an example of a pattern with at least two (different) binary trees representing it.

⋆**9.45.** Can you write a regular expression pattern that matches binary strings containing the same number of ones and zeros?