Chapter 7

String Matching

7.1 Introduction

In this chapter, we study a number of algorithms on strings, principally, string matching algorithms. The problem of string matching is to locate all (or some) occurrences of a given *pattern* string within a given *text* string. There are many variations of this basic problem. The pattern may be a set of strings, and the matching algorithm has to locate the occurrence of any pattern in the text. The pattern may be a regular expression for which the "best" match has to be found. The text may consist of a set of strings if, for instance, you are trying to find the occurrence of "to be or not to be" in the works of Shakespeare. In some situations the text string is fixed, but the pattern changes, as in searching Shakespeare's works. Quite often, the goal is not to find an exact match but a close enough match, as in DNA sequences or Google searches.

The string matching problem is quite different from dictionary or database search. In dictionary search, you are asked to determine if a given word belongs to a set of words. Usually, the set of words —the dictionary— is fixed. A hashing algorithm suffices in most cases for such problems. Database searches can be more complex than exact matches over strings. The database entries may be images (say, thumbprints), distances among cities, positions of vehicles in a fleet or salaries of individuals. A query may involve satisfying a predicate, e.g., find any "hospital that is within 10 miles of a specific vehicle and determine the shortest path to it".

We spend most of this chapter on the exact string matching problem: given a text string t and a pattern string p over some alphabet, construct a list of positions where p occurs within t. See Table 7.1 for an example.

The naive algorithm for this problem matches the pattern against the string starting at every possible position in the text. This may take $O(m \times n)$ time where m and n are the two string lengths. We show three different algorithms all of which run much faster, and one is an $O(m + n)$ algorithm.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
text	a	g	c	t	t	a	c	g	a	a	c	g	t	a	a	c	g	a
pattern	a	a	c	g														
output									*					*				

Table 7.1: The pattern matching problem

Exercise 83

You are given strings x and y of equal length, and asked to determine if x is a rotation of y. Solve the problem through string matching. □

Solution Determine if x occurs as a substring in yy.

Notation Let the text be t and the pattern be p. The symbols in a string are indexed starting at 0. We write $t[i]$ for the ith symbol of t, and $t[i..j]$ for the substring of t starting at i and ending just before j, $i \leq j$. Therefore, the length of $t[i..j]$ is $j - i$; it is empty if $i = j$. Similar conventions apply to p.

For a string r, write $\overline{r}$ for its length. Henceforth, the length of the pattern p, $\overline{p}$, is m; so, its elements are indexed 0 through $m-1$. Text t is an infinite string. This assumption is made so that we do not have to worry about terminations of algorithms; we simply show that every substring in t that matches p will be found ultimately.

7.2 Rabin-Karp Algorithm

The idea of this algorithm is based on hashing. Given text t and pattern p, compute $val(p)$, where function val will be specified later. Then, for each substring s of t whose length is $\overline{p}$, compute $val(s)$. Since

$$p = s \;\Rightarrow\; val(p) = val(s), \text{ or}$$
$$val(p) \neq val(s) \;\Rightarrow\; p \neq s$$

we may discard string s if $val(p) \neq val(s)$. We illustrate the procedure for strings of 5 decimal digits where val returns the sum of the digits in its argument string.

Let $p = 27681$; so, $val(p) = 24$. Consider the text given in Table 7.2; the function values are also shown there (at the position at which a string ends). There are two strings for which the function value is 24, namely 27681 and 19833. We compare each of these strings against the original string, 27681, to find that there is one exact match.

Function val is similar to a hash function. It is used to remove most strings from consideration. Only when $val(s) = val(p)$ do we have a *collision*, and we match s against p. As in hashing, we require that there should be very few collisions on the average; moreover, val should be easily computable incrementally, i.e., from one string to the next.

text	2	4	1	5	7	2	7	6	8	1	9	8	3	3	7	8	1	4
$val(s)$					19	19	22	27	30	24	31	32	29	24	30	29	22	23

Table 7.2: Computing function values in Rabin-Karp algorithm

Minimizing Collisions A function like *val* partitions the set of strings into equivalence classes: two strings are in the same equivalence class if their function values are identical. Strings in the same equivalence class cause collisions, as in the case of 27681 and 19833, shown above. In order to reduce collisions, we strive to make the equivalence classes equal in size. Then, the probability of collision is $1/n$, where n is the number of possible values of *val*.

For the function that sums the digits of a 5-digit number, the possible values range from 0 (all digits are 0s) to 45 (all digits are 9s). But the 46 equivalence classes are not equal in size. Note that $val(s) = 0$ iff $s = 00000$; thus if you are searching for pattern 00000 you will never have a collision. However, $val(s) = 24$ for 5875 different 5-digit strings. So, the probability of collision is around 0.05875 (since there are 10^5 5-digit strings). If there had been an even distribution among the 46 equivalence classes, the probability of collision would have been 1/46, or around 0.02173, almost three times fewer collisions than when $val(s) = 24$.

One way of distributing the numbers evenly is to let $val(s) = s \bmod q$, for some q; we will choose q to be a prime, for efficient computation. Since the number of 5-digit strings may not be a multiple of q, the distribution may not be completely even, but no two classes differ by more than 1 in their sizes. So, this is as good as it gets.

Incremental computation of *val* The next question is how to calculate *val* efficiently, for *all* substrings in the text. If the function adds up the digits in 5-digit strings, then it is easy to compute: suppose we have already computed the sum, s, for a five digit string $b_0b_1b_2b_3b_4$; to compute the sum for the next substring $b_1b_2b_3b_4b_5$, we assign $s := s - b_0 + b_5$. I show that the modulo function can be calculated equally easily.

The main observation for performing this computation is as follows. Suppose we have already scanned a n-digit string "ar", where a is the first symbol of the string and r is its tail; let ar denote the numerical value of "ar". The function value, $ar \bmod q$, has been computed already. When we scan the digit b following r, we have to evaluate $rb \bmod q$ where rb is the numerical value of "rb". We represent rb in terms of ar, a and b. First, remove a from ar by subtracting $a \times 10^{n-1}$; this gives us r. Next, left shift r by one position, which is $r \times 10$. Finally, add b. So, $rb = (ar - a \times 10^{n-1}) \times 10 + b$. To compute $rb \bmod q$, for prime q, we need a few simple results about mod.

$$
\begin{aligned}
(a + b) \bmod q &= (a + b \bmod q) \bmod q \\
(a - b) \bmod q &= (a - b \bmod q) \bmod q \\
(a \times b) \bmod q &= (a \times b \bmod q) \bmod q
\end{aligned}
$$

Modular Simplification Rule

Let e be any expression over integers that has only addition, subtraction, multiplication and exponention as its operators. Let e' be obtained from e by replacing any subexpression t of e by $(t \bmod p)$. Then, $e \overset{\text{mod } p}{\equiv} e'$, i.e., $e \bmod p = e' \bmod p$.

Note that an exponent is not a subexpression; so, it can't be replaced by its mod.

Examples

$(20 + 5) \bmod 3 = ((20 \bmod 3) + 5) \bmod 3$
$((x \times y) + g) \bmod p = (((x \bmod p) \times y) + (g \bmod p)) \bmod p$
$x^n \bmod p = (x \bmod p)^n \bmod p$
$x^{2n} \bmod p = (x^2)^n \bmod p = (x^2 \bmod p)^n \bmod p$
$x^n \bmod p = x^{n \bmod p} \bmod p$, is wrong. □

We use this rule to compute $rb \bmod q$.

$$
\begin{array}{ll}
 & rb \bmod q \\
= & \{rb = (ar - a \times 10^{n-1}) \times 10 + b\} \\
 & ((ar - a \times 10^{n-1}) \times 10 + b) \bmod q \\
= & \{\text{replace } ar \text{ and } 10^{n-1}\} \\
 & (((ar \bmod q) - a \times (10^{n-1} \bmod q)) \times 10 + b) \bmod q \\
= & \{\text{let } u = ar \bmod q \text{ and } f = 10^{n-1} \bmod q, \text{ both already computed}\} \\
 & ((u - a \times f) \times 10 + b) \bmod q
\end{array}
$$

Example Let q be 47. Suppose we have computed $12768 \bmod 47$, which is 31. And, also $10^4 \bmod 47$ which is 36. We compute $27687 \bmod 47$ by

$$
\begin{array}{ll}
 & ((31 - 1 \times 36) \times 10 + 7) \bmod 47 \\
= & ((-5) \times 10 + 7) \bmod 47 \\
= & (-43) \bmod 47 \\
= & 4
\end{array}
$$
□

Exercise 84

Show that the equivalence classes under mod q are almost equal in size. □

Exercise 85

Derive a general formula for incremental calculation when the alphabet has d symbols, so that each string can be regarded as a d-ary number. □

7.3 Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm (KMP) locates all occurrences of a pattern in a text in linear time (in the combined lengths of the two strings). It is a refined version of the naive algorithm.

7.3.1 Informal Description

Let the pattern be "JayadevMisra". Suppose, we have matched the portion "JayadevM" against some part of the text string, but the next symbol in the text differs from 'i', the next symbol in the pattern. The naive algorithm would shift one position beyond 'J' in the text, and start the match all over, starting with 'J', the first symbol of the pattern. The KMP algorithm is based on the observation that no symbol in the text that we have already matched with "JayadevM" can possibly be the start of a full match: we have just discovered that the there is no match starting at 'J', and there is no match starting at any other symbol because none of them is a 'J'. So, we may skip this entire string in the text and shift to the next symbol beyond "JayadevM" to begin a match.

In general, we will not be lucky enough to skip the entire piece of text that we had already matched, as we could in the case of "JayadevM". For instance, suppose the pattern is "axbcyaxbts", and we have already matched "axbcyaxb"; see Table 7.3. Suppose the next symbol in the text does not match 's', the next symbol in the pattern. A possible match could begin at the second occurrence of 'a', because the text beginning at that point is "axb", a prefix of the pattern. So, we shift to that position in the text, but we avoid scanning any symbol in this portion of the text again. The formal description, given next, establishes the conditions that need to be satisfied for this scheme to work.

7.3.2 Algorithm Outline

At any point during the algorithm we have matched a portion of the pattern against the text; that is, we maintain the following invariant where l and r are indices in t defining the two ends of the matched portion.

KMP-INV:
$l \leq r \ \wedge \ t[l..r] = p[0..r-l]$, and
all occurrences of p starting prior to l in the text have been located.

The invariant is established initially by setting

$l, r := 0, 0$

In subsequent steps we compare the next symbols from the text and the pattern. If there is no next symbol in the pattern, we have found a match, and we discuss what to do next below. For the moment, assume that p has a next symbol, $p[r-l]$.

$t[r] = p[r-l] \quad \longrightarrow \ r := r+1$
{ more text has been matched }
$t[r] \neq p[r-l] \ \wedge \ r = l \quad \longrightarrow \ l := l+1; \ r := r+1$
{ we have an empty string matched so far;
the first pattern symbol differs from the next text symbol }
$t[r] \neq p[r-l] \ \wedge \ r > l \quad \longrightarrow \ l := l'$
{ a nonempty prefix of p has matched but the next symbols don't }

Note that in the third case, r is not changed; so, none of the symbols in $t[l'..r]$ will be scanned again.

The question (in the third case) is, what is l'? Abbreviate $t[l..r]$ by v and $t[l'..r]$ by u. We show below that u is a proper prefix *and* a proper suffix of v. Thus, l' is given by the longest u that is both a proper prefix and a proper suffix of v.

From the invariant, v is a prefix of p. Also, from the invariant, u is a prefix of p, and, since $l' > l$, u is a shorter prefix than v. Therefore, u is a proper prefix of v. Next, since their right ends match, $t[l'..r]$ is a proper suffix of $t[l..r]$, i.e., u is a proper suffix of v.

We describe the algorithm in schematic form in Table 7.3. Here, we have already matched the prefix "axbcyaxb", which is v. There is a mismatch in the next symbol. We shift the pattern so that the prefix "axb", which is u, is aligned with a portion of the text that matches it.

index	l					l'			r						
text	a	x	b	c	y	a	x	b	z	-	-	-	-	-	-
pattern	a	x	b	c	y	a	x	b	t	s					
newmatch						a	x	b	c	y	a	x	b	t	s

Table 7.3: Matching in the KMP algorithm

In general, there may be many strings, u, which are both proper prefix and suffix of v; in particular, the empty string satisfies this condition for any v. Which u should we choose? Any u we choose could possibly lead to a match, because we have not scanned beyond $t[r]$. So, we increment l by the minimum required amount, i.e., u is the longest string that is both a proper prefix and suffix of v; we call u the *core* of v.

The question of computing l' then reduces to the following problem: given a string v, find its core. Then $l' = l + (\text{length of } v) - (\text{length of core of } v)$. Since v is a prefix of p, we precompute the cores of all prefixes of the pattern, so that we may compute l' whenever there is a failure in the match. In the next subsection we develop a linear algorithm to compute cores of all prefixes of the pattern.

After the pattern has been completely matched, we record this fact and let $l' = l + (\text{length of } p) - (\text{length of core of } p)$.

We show that KMP runs in linear time. Observe that $l + r$ increases in each step (in the last case, $l' > l$). Both l and r are bounded by the length of the text string; so the number of steps is bounded by a linear function of the length of text. The core computation, in the next section, is linear in the size of the pattern. So, the whole algorithm is linear.

7.3.3 The Theory of Core

First, we develop a small theory about prefixes and suffixes from which the computation of the core will follow.

A Partial Order over Strings For strings u and v, we write $u \preceq v$ to mean that u is both a prefix and a suffix of v. Observe that $u \preceq v$ holds whenever u is a prefix of v and the reverse of u is a prefix of the reverse of v. As is usual, we write $u \prec v$ to mean that $u \preceq v$ and $u \neq v$.

Exercise 86

Find all strings u such that $u \prec ababab$. □

The following properties of $\preceq$ follow from the properties of the prefix relation; you are expected to develop the proofs. Henceforth, u and v denote arbitrary strings and ϵ is the empty string.

Exercise 87

1. $\epsilon \preceq u$.
2. $\preceq$ is a partial order. Use the fact that prefix relation is a partial order.
3. There is a total order among all u where $u \preceq v$, i.e.,

$$(u \preceq v \ \wedge \ w \preceq v) \ \Rightarrow \ (u \preceq w \ \vee \ w \preceq u)$$ □

7.3.3.1 Definition of core

For any nonempty v, *core* of v, written as $c(v)$, is the longest string such that $c(v) \prec v$. The core is defined for every v, $v \neq \epsilon$, because there is at least one string, namely ϵ, that is a proper prefix and suffix of every nonempty string.

Example We compute cores of several strings.

a	ab	abb	abba	abbab	abbabb	abbabba	abbabbb
ϵ	ϵ	ϵ	a	ab	abb	abba	ϵ

Table 7.4: Examples of Cores

The traditional way to formally define core of v, $c(v)$, is as follows: (1) $c(v) \prec v$, and (2) for any w where $w \prec v$, $w \preceq c(v)$. We give a different, though equivalent, definition that is more convenient for formal manipulations. For any u and v, $v \neq \epsilon$,

(core definition): $u \preceq c(v) \ \equiv \ u \prec v$

It follows, by replacing u with $c(v)$, that $c(v) \prec v$. In particular, $\overline{c(v)} < \overline{v}$. Also, every non-empty string has a unique core. To see this, let r and s be cores of v. We show that $r = s$. For any u, $u \preceq c(v) \ \equiv \ u \prec v$. Using r and s for $c(v)$, we get

$$u \preceq r \ \equiv \ u \prec v \text{ and } u \preceq s \ \equiv \ u \prec v$$

That is, $u \preceq r \ \equiv \ u \preceq s$, for all u. Setting u to r, we get $r \preceq r \ \equiv \ r \preceq s$, i.e., $r \preceq s$. Similarly, we can deduce $s \preceq r$. So, $r = s$ from the antisymmetry of $\preceq$.□

Exercise 88

Let u be a longer string than v. Is $c(u)$ necessarily longer than $c(v)$? □

Exercise 89

Show that the core function is monotonic, that is,

$$u \preceq v \ \Rightarrow \ c(u) \preceq c(v)$$ □

We write $c^i(v)$ for i-fold application of c to v, i.e., $c^i(v) = \overbrace{c(c(..(c}^{i \text{ times}}(v)..)))$ and $c^0(v) = v$. Since $\overline{c(v)} < \overline{v}$, $c^i(v)$ is defined only for some i, not necessarily all i, in the range $0 \leq i \leq \overline{v}$. Note that, $c^{i+1}(v) \prec c^i(v) \ldots c^1(v) \prec c^0(v) = v$.

Exercise 90

Compute $c^i(ababab)$ for all possible i. What is $c^i(ab)^n$, for any i, $i \leq n$? □

7.3.3.2 A characterization of $\preceq$ in terms of core

The following proposition says that any string u, where $u \prec v$, can be obtained by applying function c a sufficient (non-zero) number of times to v.

P1: For any u and v,

$$u \preceq v \ \equiv \ \langle \exists i : 0 \leq i : u = c^i(v) \rangle$$

Proof: The proof is by induction on the length of v.

• $\overline{v} = 0$:

$$\begin{array}{ll} & u \preceq v \\ \equiv & \{\overline{v} = 0, \text{ i.e., } v = \epsilon\} \\ & u = \epsilon \ \wedge \ v = \epsilon \\ \equiv & \{\text{definition of } c^0\text{: } v = \epsilon \ \Rightarrow \ c^i(v) \text{ is defined for } i = 0 \text{ only}\} \\ & \langle \exists i : 0 \leq i : u = c^i(v) \rangle \end{array}$$

• $\overline{v} > 0$:

$$
\begin{array}{ll}
 & u \preceq v \\
\equiv & \{\text{definition of } \preceq\} \\
 & u = v \ \vee \ u \prec v \\
\equiv & \{\text{definition of core}\} \\
 & u = v \ \vee \ u \preceq c(v) \\
\equiv & \{\overline{c(v)} < \overline{v}\text{; apply induction hypothesis on second term}\} \\
 & u = v \ \vee \ \langle \exists i : 0 \leq i : u = c^i(c(v)) \rangle \\
\equiv & \{\text{rewrite}\} \\
 & u = c^0(v) \ \vee \ \langle \exists i : 0 < i : u = c^i(v) \rangle \\
\equiv & \{\text{rewrite}\} \\
 & \langle \exists i : 0 \leq i : u = c^i(v) \rangle
\end{array}
$$

□

Corollary For any u and v, $v \neq \epsilon$,

$$u \prec v \ \equiv \ \langle \exists i : 0 < i : u = c^i(v) \rangle$$

□

7.3.3.3 Incremental Computation of Core

We show how to compute the core of string us, where u is a string and s a symbol, from $c(u)$, $c^2(u)$, $c^3(u)$, $\cdots$. First, suppose that us has a non-empty core. Then that core is of the form vs for some string v, because the core of us has to be a suffix of us; therefore, its last symbol has to be s.

$$
\begin{array}{ll}
 & vs \prec us \\
\equiv & \{\text{definition of } \prec\} \\
 & vs \text{ is a proper prefix of } us,\ vs \text{ is a proper suffix of } us \\
\equiv & \{vs \text{ is a proper prefix of } us \equiv v \text{ is a proper prefix of } u \text{ and } u[|v|] = s\} \\
 & v \text{ is a proper prefix of } u,\ u[|v|] = s,\ vs \text{ is a proper suffix of } us \\
\equiv & \{vs \text{ is a proper suffix of } us \equiv v \text{ is a proper suffix of } u\} \\
 & v \text{ is a proper prefix of } u,\ u[|v|] = s,\ v \text{ is a proper suffix of } u \\
\equiv & \{\text{definition of } \prec\} \\
 & v \prec u,\ u[|v|] = s \\
\equiv & \{\text{proposition P1: } v \prec u \equiv (\exists i : i > 0 : v = c^i(u))\} \\
 & (\exists i : i > 0 : v = c^i(u)),\ u[|v|] = s
\end{array}
$$

To compute the core of us find the longest v such that $v = c^i(u))$ for some positive i, and $u[|v|] = s$. So, we have to find the smallest i meeting these conditions. That is, check if $u[|c^1(u)|] = s$, if not, check if $u[|c^2(u)|] = s$, $\cdots$. These checks continue until we either find some j such that $u[|c^j(u)|] = s$ —then, $v = c^j(u)$ and the $c(us) = vs$— or, $c^j(u)$ is not defined —then $c(us) = \epsilon$.

Example In Figure 7.1, u is a string and s is the symbol following it. The prefixes of u ending at A, B, C are $c^1(u)$, $c^2(u)$ and $c^3(u)$, and the symbols following them are a, b, c, respectively. Here C is the empty string. To compute the core of us, match s against a; in case of failure, match s against b, and again in case of failure, match s against c.

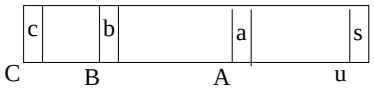

Figure 7.1: Incremental Core Computation

7.3.4 Computing Cores of all Non-empty Prefixes

The KMP algorithm needs the core of string q when the pattern match fails after having matched q. String q is a prefix of the pattern string p. Therefore, we pre-compute the cores of *all* non-empty prefixes of p.

7.3.4.1 Computing cores of all prefixes of a pattern

We compute the cores of pattern p using the scheme of Section 7.3.3.3. The core for the empty prefix is undefined. For a prefix of length 1, the core is ϵ. Next, suppose we have computed the cores of all prefixes of p up to length j; we then compute the core of the next longer prefix, of length $j+1$, using the scheme of Section 7.3.3.3.

First, let us decide how to store the cores. Any core of a prefix of p is a prefix of p. So, we can simply store the length of the core. We store the cores in array d, where $d[k]$, for $k > 0$, is the length of the core of $p[0..k]$, i.e., $d[k] = |c(p[0..k])|$. (Prefix of length 0 is ϵ which does not have a core.)

The following program has the invariant that the cores of all prefixes up to and including $p[0..j]$ are known; let u be $p[0..j]$. The goal of the program is to next compute the core of us where $s = p[j]$. To this end, we apply the scheme of Section 7.3.3.3, whereby we successively check if $u[|c^1(u)|] = s$, $u[|c^2(u)|] = s$, $\cdots$, . Suppose $u[|c^k(u)|] \neq s$ for all k, $0 < k < t$. We next have to check if $u[|c^t(u)|] = s$. Let $i = |c^t(u)|$; so the check is if $u[i] = s$, or since u is a prefix of p, $p[i] = s$, or since $p[j] = s$, if $p[i] = p[j]$. If this check succeeds, we have found the core of us, i.e., of $p[0..j+1]$; it is simply $p[0..i+1]$, or $d[j+1] = i+1$. If the check fails, we have to set $i := |c^{t+1}(u)| = |c(c^t(u))| = |c(p[0..i])| = d(i)$. However, if $i = 0$ then $d(i)$ is not defined, and we conclude that the core of $p[0..j+1] = \epsilon$, or $d[j+1] = 0$.

Below, $b \rightarrow C$, where b is a predicate and S a sequence of statements, is known as a *guarded command*; b is the guard and C the command. Command C is executed only if b holds. Below, exactly one guard is true in any iteration.

$j := 1;\ d[1] := 0;\ i := 0;$
while $j < \overline{p}$ **do**
  S_1:: $p[i] = p[j] \rightarrow d[j+1] := i+1;\ j := j+1;\ i := d[j]$
  S_2:: $p[i] \neq p[j] \wedge i \neq 0 \rightarrow i := d[i]$
  S_3:: $p[i] \neq p[j] \wedge i = 0 \rightarrow d[j+1] := 0;\ j := j+1;\ i := d[j]\ \{i = 0\}$
endwhile

7.3.4.2 Analysis of the running time of core computation

We show that the program for core computation runs in linear time in the length of pattern p. Specifically, execution of each guarded command increases $2j - i$. Since $j \leq \overline{p}$ and $i \geq 0$ (prove these as invariants), $2j - i$ never exceeds $2\overline{p}$. Initially, $2j - i = 2$. Therefore, the number of executions of all guarded commands is $O(\overline{p})$.

Now, we consider the right side of each guarded command S_k, $1 \leq k \leq 3$, and show that each one strictly increases $2j - i$, i.e.,

$$\{2j - i = n\} \quad \text{right side of } \mathrm{S}_k \quad \{2j - i > n\}$$

• Proof for S_1:

$$\begin{array}{ll}
\{2j - i = n\}\ d[j+1] := i + 1;\ j := j + 1;\ i := d[j]\ \{2j - i > n\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, goal} \\
\{2j - i = n\}\ d[j+1] := i + 1;\ j := j + 1\ \{2j - d[j] > n\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, axiom of assignment} \\
\{2j - i = n\}\ d[j+1] := i + 1\ \{2(j+1) - d[j+1] > n\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, axiom of assignment} \\
\{2j - i = n\}\ \{2(j+1) - (i+1) > n\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, axiom of assignment} \\
2j - i = n\ \Rightarrow\ 2(j+1) - (i+1) > n & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, simplify} \\
\mathit{true} & \text{, arithmetic} \qquad \square
\end{array}$$

• Proof for S_2:

$$\begin{array}{ll}
\{2j - i = n\}\ i := d[i]\ \{2j - i > n\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, goal} \\
2j - i = n\ \Rightarrow\ 2j - d[i] > n & \text{, axiom of assignment} \\
2j - i < 2j - d[i] & \text{, arithmetic} \\
d[i] < i & \text{, arithmetic} \\
\mathit{true} & \text{, } i = \overline{u} \text{ and } d[i] = \overline{c(u)} \qquad \square
\end{array}$$

• Proof for S_3:

$$\begin{array}{ll}
\{2j - i = n\}\ d[j+1] := 0;\ j := j + 1;\ i := d[j]\ \{2j - i > n\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, goal} \\
\{2j - i = n\}\ d[j+1] := 0;\ j := j + 1\ \{2j - d[j] > n\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, axiom of assignment} \\
\{2j - i = n\}\ d[j+1] := 0\ \{2(j+1) - d[j+1] > n\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, axiom of assignment} \\
\{2j - i = n\}\ \{2(j+1) - 0 > n\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad & \text{, axiom of assignment} \\
2j - i = n\ \Rightarrow\ 2(j+1) > n & \text{, simplify} \\
\mathit{true} & \text{, arithmetic and } i \geq 0 \text{ (invariant)} \qquad \square
\end{array}$$

Exercise 91

1. Show that you can match pattern p against text t by computing the cores of all prefixes of pt (pt is the concatenation of p and t).

2. Define u to be the k-core of string v, where $k \geq 0$ and $v \neq \epsilon$, if $u \prec v$, u's length is at most k and u is the longest string with this property. Show that the k-core is well-defined. Devise an algorithm to compute the k-core of a string for a given k. □

7.4 Boyer-Moore Algorithm

The next string matching algorithm we study is due to Boyer and Moore. It has the best performance, on the average, of all known algorithms for this problem. In many cases, it runs in sublinear time, because it may not even scan all the symbols of the text. Its worst case behavior could be as bad as the naive matching algorithm.

At any moment, imagine that the pattern is *aligned* with a portion of the text of the same length, though only a part of the aligned text may have been matched with the pattern. Henceforth, alignment refers to the substring of t that is aligned with p and l is the index of the left end of the alignment; i.e., $p[0]$ is aligned with $t[l]$ and, in general, $p[i]$, $0 \leq i < m$, with $t[l+i]$. Whenever there is a mismatch, the pattern is *shifted* to the right, i.e., l is increased, as explained in the following sections.

7.4.1 Algorithm Outline

The overall structure of the program is a loop that has the invariant:

Q1: Every occurrence of p in t that starts before l has been recorded.

The following loop records every occurrence of p in t eventually.

$l := 0$;
{ Q1 }
loop
  { Q1 }
  "increase l while preserving Q1"
  { Q1 }
endloop

Next, we show how to increase l while preserving Q1. To do so, we need to match certain symbols of the pattern against the text. We introduce variable j, $0 \leq j < m$, with the meaning that the suffix of p starting at position j matches the corresponding portion of the alignment; i.e.,

Q2: $0 \leq j \leq m$, $p[j..m] = t[l+j..l+m]$

Thus, the whole pattern is matched when $j = 0$, and no part has been matched when $j = m$.

We establish Q2 by setting j to m. Then, we match the symbols from *right to left* of the pattern (against the corresponding symbols in the alignment) until we find a mismatch or the whole pattern is matched.

$j := m$;
{ Q2 }
while $j > 0 \ \wedge \ p[j-1] = t[l+j-1]$ **do** $j := j-1$ **endwhile**
  { Q1 $\wedge$ Q2 $\wedge$ $(j = 0 \ \vee \ p[j-1] \neq t[l+j-1])$ }
  if $j = 0$
    then { Q1 $\wedge$ Q2 $\wedge$ $j = 0$ } record a match at l; $l := l'$ { Q1 }
    else { Q1 $\wedge$ Q2 $\wedge$ $j > 0 \ \wedge \ p[j-1] \neq t[l+j-1]$ } $l := l''$ { Q1 }
  endif
{ Q1 }

Next, we show how to compute l and l', $l' > l$ and $l'' > l$, so that Q1 is satisfied. For better performance, l should be increased as much as possible in each case. We take up the computation of l'' next; computation of l' is a special case of this.

The precondition for the computation of l'' is,

$$\text{Q1} \wedge \text{Q2} \wedge j > 0 \ \wedge \ p[j-1] \neq t[l+j-1].$$

We consider two heuristics, each of which can be used to calculate a value of l''; the greater value is assigned to l. The first heuristic, called the *bad symbol heuristic*, exploits the fact that we have a mismatch at position $j-1$ of the pattern. The second heuristic, called the *good suffix heuristic*, uses the fact that we have matched a suffix of p with the suffix of the alignment, i.e., $p[j..m] = t[l+j..l+m]$ (though the suffix may be empty).

7.4.2 The Bad Symbol Heuristic

Suppose we have the pattern "attendance" that we have aligned against a portion of the text whose suffix is "hce", as shown in Table 7.5.

text	-	-	-	-	-	-	-	h	c	e										
pattern	a	t	t	e	n	d	a	n	c	e										
align									a	t	t	e	n	d	a	n	c	e		

Table 7.5: The bad symbol heuristic

The suffix "ce" has been matched; the symbols 'h' and 'n' do not match. We now reason as follows. If symbol 'h' of the text is part of a full match, that symbol has to be aligned with an 'h' of the pattern. There is no 'h' in the pattern; so, no match can include this 'h' of the text. Hence, the pattern

may be shifted to the symbol following 'h' in the text, as shown under *align* in Table 7.5. Since the index of 'h' in the text is $l + j - 1$ (that is where the mismatch occurred), we have to align $p[0]$ with $t[l+j]$, i.e., l should be increased to $l + j$. Observe that we have shifted the alignment several positions to the right without scanning the text symbols shown by dashes, '-', in the text; this is how the algorithm achieves sublinear running time in many cases.

Next, suppose the mismatched symbol in the text is 't', as shown in Table 7.6.

text	-	-	-	-	-	-	-	t	c	e
pattern	a	t	t	e	n	d	a	n	c	e

Table 7.6: The bad symbol heuristic

Unlike 'h', symbol 't' appears in the pattern. We align some occurrence of 't' in the pattern with that in the text. There are two possible alignments, which we show in Table 7.7.

text	-	-	t	c	e	-	-	-	-	-	-
align1	a	t	t	e	n	d	a	n	c	e	
align2		a	t	t	e	n	d	a	n	c	e

Table 7.7: New alignment in the bad symbol heuristic

Which alignment should we choose? The same question also comes up in the good suffix heuristic. We have several possible shifts each of which matches a portion of the alignment. We adopt the following rule for shift:

Minimum shift rule: Shift the pattern by the minimum allowable amount.

According to this rule, in Table 7.8 we would shift the pattern to get *align1*.

Justification for the rule: This rule preserves Q1; we never skip over a possible match following this rule, because no smaller shift yields a match at the given position, and, hence no full match.

Conversely, consider the situation shown in Table 7.8. The first pattern line shows an alignment where there is a mismatch at the rightmost symbol in the alignment. The next two lines show two possible alignments that correct the mismatch. Since the only text symbol we have examined is 'x', each dash in Table 7.8 could be any symbol at all; so, in particular, the text could be such that the pattern matches against the first alignment, *align1*. Then, we will violate invariant Q1 if we shift the pattern as shown in *align2*. □

For each symbol in the alphabet, we precalculate its rightmost position in the pattern. The rightmost 't' in "attendance" is at position 2. To align the mismatched 't' in the text in Table 7.7 that is at position $t[l+j-1]$, we align $p[2]$ with $t[l + j - 1]$, that is, $p[0]$ with $t[l - 2 + j - 1]$. In general, if the mismatched

text	-	-	x	-	-
pattern	x	x	y		
align1		x	x	y	
align2			x	x	y

Table 7.8: Realignment in the good suffix heuristic

symbol's rightmost occurrence in the pattern is at $p[k]$, then $p[0]$ is aligned with $t[l-k+j-1]$, or l is increased by $-k+j-1$. For a nonexistent symbol in the pattern, like 'h', we set its rightmost occurrence to -1 so that l is increased to $l+j$, as required.

The quantity $-k+j-1$ is negative if $k > j-1$. That is, the rightmost occurrence of the mismatched symbol in the pattern is to the right of the mismatch. Fortunately, the good suffix heuristic, which we discuss in Section 7.4.3, always yields a positive increment for l; so, we ignore this heuristic if it yields a negative increment.

Computing the rightmost positions of the symbols in the pattern For a given alphabet, we compute an array rt, indexed by the symbols of the alphabet, so that for any symbol 'a',

$$rt(\text{'a'}) = \begin{cases} \text{position of the rightmost 'a' in } p, & \text{if 'a' is in } p \\ -1 & \text{otherwise} \end{cases}$$

The following simple loop computes rt.

let rt['a'] := -1, for every symbol 'a' in the alphabet;
for $j = 0$ **to** $m-1$ **do**
  $rt[p[j]] := j$
endfor

7.4.3 The Good Suffix Heuristic

Suppose we have a pattern "abxabyab" of which we have already matched the suffix "ab", but there is a mismatch with the preceding symbol 'y', as shown in Table 7.9.

text	-	-	-	-	-	z	a	b	-	-
pattern	a	b	x	a	b	y	a	b		

Table 7.9: A good suffix heuristic scenario

Then, we shift the pattern to the right so that the matched part is occupied by the same symbols, "ab"; this is possible only if there is another occurrence

of "ab" in the pattern. For the pattern of Table 7.9, we can form the new alignment in two possible ways, as shown in Table 7.10.

text	-	-	z	a	b	-	-	-	-	-	-
align1	a	b	x	a	b	y	a	b			
align2				a	b	x	a	b	y	a	b

Table 7.10: Realignment in the good suffix heuristic

No complete match of the suffix s is possible if s does not occur elsewhere in p. This possibility is shown in Table 7.11, where s is "xab". In this case, the best that can be done is to match with a suffix of "xab", as shown in Table 7.11. Note that the matching portion "ab" is a prefix of p. Also, it is a suffix of p, being a suffix of "xab", that is a suffix of p.

text	-	-	x	a	b	-	-	-
pattern	a	b	x	a	b			
align				a	b	x	a	b

Table 7.11: The matched suffix is nowhere else in p

As shown in the preceding examples, in all cases we shift the pattern to align the right end of a proper prefix r with the right end of the previous alignment. Also, r is a suffix of s or s is a suffix of r. In the example in Table 7.10, s is "ab" and there are two possible r, "abxab" and "ab", for which s is a suffix. Additionally, ϵ is a suffix of s. In Table 7.11, s is "xab" and there is exactly one nonempty r, "ab", which is a suffix of s. Let

$$R = \{r \text{ is a proper prefix of } p \wedge \\ (r \text{ is a suffix of } s \vee s \text{ is a suffix of } r)\}$$

The good suffix heuristic aligns an r in R with the end of the previous alignment, i.e., the pattern is shifted to the right by $m - \overline{r}$. Let $b(s)$ be the amount by which the pattern should be shifted for a suffix s. According to the minimum shift rule,

$$b(s) = \min\{m - \overline{r} \mid r \in R\}$$

In the rest of this section, we develop an efficient algorithm for computing $b(s)$.

7.4.3.1 Shifting the pattern in the algorithm of Section 7.4.1

In the algorithm outlined in Section 7.4.1, we have two assignments to l, the assignment

$l := l'$, when the whole pattern has matched, and
$l := l''$, when $p[j..m] = t[l + j..l + m]$ and $p[j - 1] \neq t[l + j - 1]$

$l := l'$ is implemented by
  $l := l + b(p)$, and
$l := l''$ is implemented by
  $l := l + \max(b(s), j - 1 - rt(h))$,
    where $s = p[j..m]$ and $h = t[l + j - 1]$

7.4.3.2 Properties of $b(s)$, the shift amount for suffix s

We repeat the definition of $b(s)$.

$$b(s) = \min\{m - \overline{r} \mid r \in R\}$$

Notation We abbreviate $\min\{m - \overline{r} \mid r \in R\}$ to $\min(m - R)$. In general, let S be a set of strings and $e(S)$ an expression that includes S as a term. Then, $\min(e(S)) = \min\{e(\overline{i}) \mid i \in S\}$, where $e(\overline{i})$ is obtained from e by replacing S by $\overline{i}$. □

Rewrite R as $R' \cup R''$, where

$$\begin{aligned} R' &= \{r \text{ is a proper prefix of } p \ \wedge \ r \text{ is a suffix of } s\} \\ R'' &= \{r \text{ is a proper prefix of } p \ \wedge \ s \text{ is a suffix of } r\} \end{aligned}$$

Then,

$$b(s) = \min(\min(m - R'),\ \min(m - R''))$$

where minimum over empty set is ∞.

P1: R is nonempty and $b(s)$ is well-defined.
Proof: Note that $\epsilon \in R'$ and $R = R' \cup R''$. Then, from its definition, $b(s)$ is well-defined. □

P2: $c(p) \in R$
Proof: From the definition of core, $c(p) \prec p$. Hence, $c(p)$ is a proper prefix of p. Also, $c(p)$ is a suffix of p, and, since s is a suffix of p, they are totally ordered. So, either $c(p)$ is a suffix of s or s is a suffix of $c(p)$. Hence, $c(p) \in R$. □

P3: $\min(m - R') \ \geq \ m - \overline{c(p)}$
Proof: Consider any r in R'. Since r is a suffix of s and s is a suffix of p, r is a suffix of p. Also, r is a proper prefix of p. So, $r \prec p$. From the definition of core, $r \preceq c(p)$. Hence, $m - \overline{r} \geq \ m - \overline{c(p)}$ for every r in R'. □

P4: Let $V = \{v \mid v \text{ is a suffix of } \ p \ \wedge \ c(v) = s\}$.
Then, $\min(m - R'') \ = \ \min(V - \overline{s})$
Proof: Note that R'' may be empty. In that case, V will be empty too and both will have the same minimum, ∞. This causes no problem in computing $b(s)$ because, from (P1), R is nonempty.

Consider any r in R''. Note that r is a prefix of p and has s as a suffix; so $p = xsy$, for some x and y, where $r = xs$ and y is the remaining portion of p. Also, $y \neq \epsilon$ because r is a proper prefix. Let u stand for sy; then u is a suffix of p. And, $u \neq \epsilon$ because $y \neq \epsilon$, though u may be equal to p, because x could be empty. Also, s is a prefix of u ($u = sy$) and a suffix of u (s and u are both suffixes of p and u is longer than s). Therefore, $s \prec u$. Define

$$U = \{u \mid u \text{ is a suffix of } \ p \ \wedge \ s \prec u\}$$

We have thus shown that there is a one-to-one correspondence between the elements of R'' and U. Given that $p = xsy$, $r = xs$, and $u = sy$, we have $m - \overline{r} = \overline{y} = \overline{u} - \overline{s}$. Hence,

$$\min(m - R'') \ = \ \min(U - \overline{s}).$$

For a fixed s, the minimum value of $\overline{u} - \overline{s}$ over all u in U is achieved for the shortest string v of U. We show that $c(v) = s$. This proves the result in (P4).

$$\begin{array}{ll} & v \text{ is the shortest string in } U \\ \Rightarrow & \{\overline{c(v)} < \overline{v}\} \\ & v \in U \ \wedge \ c(v) \notin U \\ \Rightarrow & \{\text{definition of } U\} \\ & s \prec v \ \wedge \ (c(v) \text{ is not a suffix of } \ p \ \vee \ \neg(s \prec c(v)) \\ \Rightarrow & \{c(v) \text{ is a suffix of } v \text{ and } v \text{ is a suffix of } p\text{; so, } c(v) \text{ is a suffix of } p\} \\ & s \prec v \ \wedge \ \neg(s \prec c(v)) \\ \Rightarrow & \{\text{definition of core}\} \\ & (s = c(v) \ \vee \ s \prec c(v)) \ \wedge \ (\neg(s \prec c(v))) \\ \Rightarrow & \{\text{predicate calculus}\} \\ & s = c(v) \end{array}$$ □

Note: The converse of this result is not true. There may be several u in U for which $c(u) = s$. For example, consider "sxs" and "sxyx", where the symbols 'x' and 'y' do not appear in "s". Cores for both of these strings are "s". □

7.4.3.3 An abstract program for computing $b(s)$

We derive a formula for $b(s)$, and use that to develop an abstract program.

$$\begin{array}{ll} & b(s) \\ = & \{\text{definition of } b(s) \text{ from Section 7.4.3.2}\} \\ & \min(m - R) \\ = & \{\text{from (P2): } c(p) \in R\} \\ & \min(m - \overline{c(p)}, \ \min(m - R)) \\ = & \{R = R' \cup R'', \text{ from Section 7.4.3.2}\} \\ & \min(m - \overline{c(p)}, \ \min(m - R'), \ \min(m - R'')) \\ = & \{\text{from (P3): } \min(m - R') \ \geq \ m - \overline{c(p)}\} \\ & \min(m - \overline{c(p)}, \ \min(m - R'')) \\ = & \{\text{from (P4): } \min(m - R'') \ = \ \min(V - \overline{s})\} \\ & \min(m - \overline{c(p)}, \ \min(V - \overline{s})) \end{array}$$

Recall that

$$V = \{v \mid v \text{ is a suffix of } p \ \wedge \ c(v) = s\}$$

Now, we propose an abstract program to compute $b(s)$, for *all* suffixes s of p. We employ an array b where $b[s]$ ultimately holds the value of $b(s)$, though it is assigned different values during the computation. Initially, set $b[s]$ to $m - \overline{c(p)}$. Next, scan the suffixes v of p: let $s = c(v)$; update $b[s]$ to $\overline{v} - \overline{s}$ provided this value is lower than the current value of $b[s]$.

The program

assign $m - \overline{c(p)}$ to all elements of b;
for all suffixes v of p **do**
  $s := c(v)$;
  if $b[s] > \overline{v} - \overline{s}$ **then** $b[s] := \overline{v} - \overline{s}$ **endif**
endfor

7.4.3.4 A concrete program for computing $b(s)$

The goal of the concrete program is to compute an array e, where $e[j]$ is the amount by which the pattern is to be shifted when the matched suffix is $p[j..m]$, $0 \leq j \leq m$. That is,

$e[j] = b[s]$, where $j + \overline{s} = m$, or
$e[m - \overline{s}] = b[s]$, for any suffix s of p

We have no need to keep explicit prefixes and suffixes; instead, we keep their lengths, $\overline{s}$ in i and $\overline{v}$ in j. Let array f hold the lengths of the cores of all suffixes of p. Summarizing, for suffixes s and v of p,

$i = \overline{s}$,
$j = \overline{v}$,
$e[m - i] = b[s]$, using $i = \overline{s}$,
$f[\overline{v}] = \overline{c(v)}$, i.e., $f[j] = \overline{c(v)}$

The abstract program, given earlier, is transformed to the following concrete program.

assign $m - \overline{c(p)}$ to all elements of e;
for j, $0 \leq j \leq m$, **do**
  $i := f[j]$;
  if $e[m - i] > j - i$ **then** $e[m - i] := j - i$ **endif**
endfor

Computation of f The given program is complete except for the computation of f, the lengths of the cores of the suffixes of p. We have already developed a program to compute the cores of the prefixes of a string; we employ that program to compute f, as described next.

For any string r, let $\hat{r}$ be its reverse. Now, v is a suffix of p iff $\hat{v}$ is a prefix of $\hat{p}$. Moreover for any r (see exercise below)

$$c(\hat{r}) = \widehat{c(r)}$$

Therefore, for any suffix v of p and $u = \hat{v}$,

$$\begin{array}{ll} c(u) = \widehat{c(v)}, & \text{replace } r \text{ by } v, \text{ above; note: } \hat{r} = \hat{v} = u \\ \overline{c(v)} = \widehat{c(v)}, & \overline{r} = \hat{\overline{r}}, \text{ for any } r; \text{ let } r = c(v) \\ \overline{c(u)} = \overline{c(v)}, & \text{from the above two} \end{array}$$

Since our goal is to compute the lengths of the cores, $\overline{c(v)}$, we compute $\overline{c(u)}$ instead, i.e., the lengths of the cores of the prefixes of $\hat{p}$, and store them in f.

Exercise 92
Show that

1. $r \preceq s \;\equiv\; \hat{r} \preceq \hat{s}$
2. $r \prec s \;\equiv\; \hat{r} \prec \hat{s}$
3. $c(\hat{r}) = \widehat{c(r)}$

Solution

1. $$\begin{array}{ll} & r \preceq s \\ \equiv & \{\text{definition of } \preceq\} \\ & r \text{ is a prefix of } s \text{ and } r \text{ is a suffix of } s \\ \equiv & \{\text{properties of prefix, suffix and reverse}\} \\ & \hat{r} \text{ is a suffix of } \hat{s} \text{ and } \hat{r} \text{ is a prefix of } \hat{s} \\ \equiv & \{\text{definition of } \preceq\} \\ & \hat{r} \preceq \hat{s} \end{array}$$

2. Similarly.

3. Indirect proof of equality is a powerful method for proving equality. This can be applied to elements in a set which has a reflexive and antisymmetric relation like $\preceq$. To prove $y = z$ for specific elements y and z, show that for every element x,

   $$x \preceq y \;\equiv\; x \preceq z.$$

   Then, set x to y to get $y \preceq y \;\equiv\; y \preceq z$, or $y \preceq z$ since $\preceq$ is reflexive. Similarly, get $z \preceq y$. Next, use antisymmetry of $\preceq$ to get $y = z$.

   We apply this method to prove the given equality: we show that for any s, $s \preceq c(\hat{r}) \;\equiv\; s \preceq \widehat{c(r)}$.

$$
\begin{array}{ll}
 & s \preceq c(\hat{r}) \\
\equiv & \{\text{definition of core}\} \\
 & s \prec \hat{r} \\
\equiv & \{\text{second part of this exercise}\} \\
 & \hat{s} \prec r \\
\equiv & \{\text{definition of core}\} \\
 & \hat{s} \preceq c(r) \\
\equiv & \{\text{first part of this exercise}\} \\
 & s \preceq \widehat{c(r)}
\end{array}
$$

□

Execution time for the computation of b The computation of $b(s)$, for all suffixes s of p, requires (1) computing $c(p)$, (2) computing array f, and (3) executing the concrete program of this section. Note that (1) can be computed from array f; so, the steps (1,2) can be combined. The execution times of (1), (2) and (3) are linear in m, the length of p, from the text of the concrete program. So, array b can be computed in time that is linear in the length of the pattern.