Chapter 8

String Searching

8.1 The General Problem

The string searching problem can be defined as the following: given a string P and a string T, determine whether or not P appears as a substring anywhere in T. If so, return the offset of the start of the occurrence of P in T. If not, indicate that no such occurrence exists by returning -1.

Note that string searching algorithms can, in general, be extended to find *all* of the occurrence of P in T instead of only the first, but for the sake of simplicity we will initially only consider the problem of finding the *first* occurrence.

For example, let $P =$ "ring" and $T =$ "string searching". The string "ring" does appear in T, starting in position 2 (using C's zero-based indexing).

8.1.1 Terms and Definitions

- The **text** is the string that we are searching.
- The **pattern** is the string that we are searching for.
- A **shift** is an offset into a string.

In the previous example, the text is "string searching", the pattern is "ring", and the shift of the occurrence is 2.

An algorithm to solve the string searching problem immediately suggests itself– starting with a shift i of 0, simply compare the substring of T starting at i with the pattern P.

Continue testing until a match is found, or the end of T is reached. This algorithm is presented in C in algorithm 8.1.

Algorithm 8.1 (Brute-Force String Search) *Brute-Force string searching algorithm to determine whether there exists any occurrence of pattern P (of length M) in text T (of length N). If so, returns the index of the start of the occurrence; otherwise returns -1.*

```
long stringSearchBF (unsigned char *P, long M, unsigned char *T, long N)
{
	long i, j;

	for (i = 0; i <= N - M; i++) {
		for (j = 0; j < M; j++) {
			if (P[j] != T[i + j]) {
				break;
			}
		}
		if (j == M) {
			return (i);
		}
	}
	return (-1);
}
```

One important optimization that appears in this code is that the attempt to match P and each substring of T is abandoned as soon as any character in P fails to match the substring from T. Depending on the nature of P and T, this simple optimization can have a profound impact on the execution speed of the algorithm!

In the worst case, this algorithm performs $M(N - M)$ character comparisons (comparing each of the M characters in P to each of the characters in each of the $N - N$ possible substrings of T). If mismatches are recognized earlier, however, then substantially fewer comparisons are necessary. For example, if the first character of P does not appear anywhere in T, or only as part of a match, then the attempts to match each substring will all fail immediately, and this algorithm will only require $N - M$ character comparisons.

In the rest of this chapter, we will explore several ways to reduce the amount of work required in the worst case. We will begin with the Rabin-Karp algorithm, and then progress on to the Knuth-Morris-Pratt and Boyer-Moore algorithms.

It must be noted that for many purposes, the brute force string searching algorithm is sufficient. It is small and simple, and can be coded in an efficient manner that takes advantage of the underlying hardware. For short patterns, or even long patterns that begin with "rare" prefixes, the brute-force method is as fast as any.

8.2 The Rabin-Karp Algorithm

8.2.1 String Searching via Hashing

We can use the technique of hashing (explored earlier in the semester) as a way of testing whether two strings are the same– if two strings hash to the same value, then they *might* be the same string, but if they hash to different values then then are definitely *not* the same. This property can be used in a string searching algorithm as shown in algorithm 8.2.

Algorithm 8.2 (Hash Matching) *A string-searching algorithm that uses hashing to find probable matches.*

Given pattern S (of length M) and text T (of length N):

1. Compute $h_p = \mathbf{hash}(p)$.
2. For each substring t_i (of length M) of T, compute $h_i = \mathbf{hash}(t_i)$. If $h_i == h_p$, test whether $S == t$. If so, then return i.

However, at first analysis, this seems like a terrible idea: good hash functions need to look at every character in the string. If the length of the pattern is M, and the hash function looks at each character in the substring, then computing a hash of a substring of length M takes (at least) $O(M)$ to compute. If the hashing function looks at every character in the string, then this algorithm will may need to perform an $O(N)$ calculation at each of the $M - N$ possible positions in the text, giving an algorithm that is $O(M(M - N))$.

If you recall from section 8.1.1, $O(M(M - N))$ is the time complexity of the *worst-case* of the naive algorithm! This is certainly a bad sign– it would seem that by using hashing,

Figure 8.1: A hashing function to compute the hash of string $\alpha_0 \cdots \alpha_{n-1}$, where the size of the character set is b and p is relatively prime to b.

$$\mathbf{hash}_{b,p}(\alpha_0 \cdots \alpha_{n-1}) = \left(\sum_{i=0}^{n-1} \alpha_{n-i-1} b^i \right) \mathbf{mod}\ p$$

we're dooming ourselves to *always* do the amount of work that would otherwise be done by worst case of the brute-force algorithm.

As you might guess, however, there is a way around this problem. Although there is no general way to reduce the amount of work required to perform a *single* hash, there exist hashing functions that can be computed very efficiently for a *group* of strings that are related to each other. In this algorithm, each substring t_i is closely related to the next substring: the next substring simply has one character chopped off of one end and a new character added to the other end.

Can we devise a good hashing function that takes advantage of this fact? There are several, but one that is commonly used is shown in figure 8.1. This hashing function treats the string as a large number, where each character in the array represents a digit in a number in base b. The modulo of this number is then taken by p to give the hash value. In order to get good results, this p should be chosen so that it and b are relatively prime; this is easily accomplished by having b or p be prime.

Properties of Modulo

The hash function given in figure 8.1, as stated above, is clumsy (or impossible) to implement in C. The problem is that integers in C have a fixed number of bits (usually 32 or 64, though 16-bit machines are not unusual). Depending on the values of b and n, it may be inevitable that the $\sum \alpha_{n-i-1} b^i$ term will be too large to fit into an `int` (or *any* ordinary C type). For example, if $b = 256$, then n must be 4 or less in order for the result of the summation to fit into a 32-bit integer.

Luckily, we can exploit an important property of the modulo operation– taking the modulus of intermediate steps of an addition, multiplication, or subtraction calculation has no effect on the outcome of the calculation. For example, for any α and β:

$$(\alpha + \beta) \ \% \ m \ = \ ((\alpha \ \% \ m) + (\beta \ \% \ m)) \ \% \ m$$

$$
\begin{aligned}
(\alpha\beta) \% m &= ((\alpha \% m)(\beta \% m)) \% m \\
(\alpha - \beta) \% m &= ((\alpha \% m) - (\beta \% m)) \% m
\end{aligned}
$$

This means that by performing the modulo operation early and often, we can compute the modulus of the results of operations on very large numbers without actually ever worrying about the size of the numbers, as long as the modulus we are using is not very large.

For example, imagine that we want to compute $\alpha\beta \% b$, where α and β are very large numbers, but b is a small number. The product $\alpha\beta$ will be an extremely large number– but we need not ever deal with it, because $\alpha\beta \% b = (\alpha \% b)(\beta \% b) \% b$, and by definition, the result must always be somewhere in the range 0 and $b - 1$.

By aggressively taking the modulus during intermediate steps, we can avoid using very large numbers at all.

An Example

Imagine that we want to compute $2345_{10} \% 7$. It is useful to first compute several powers of 10, % 7:

$$
\begin{aligned}
1 \% 7 &= 1 \\
10 \% 7 &= (10 \times 1) \% 7 \\
&= ((10 \% 7) \times 1) \% 7 \\
&= (3 \times 1) \% 7 \\
&= 3 \\
100 \% 7 &= (10 \times 10) \% 7 \\
&= ((10 \% 7) \times (10 \% 7)) \% 7 \\
&= (3 \times 3) \% 7 \\
&= 2 \\
1000 \% 7 &= (100 \times 10) \% 7 \\
&= ((100 \% 7) \times (10 \% 7)) \% 7 \\
&= (2 \times 3) \% 7 \\
&= 6 \\
10000 \% 7 &= (1000 \times 10) \% 7 \\
&= ((1000 \% 7) \times (10 \% 7)) \% 7
\end{aligned}
$$

$$
\begin{aligned}
&= (6 \times 3) \ \% \ 7 \\
&= 4
\end{aligned}
$$

Given these values, it is easy to compute 2345 % 7:

$$
\begin{aligned}
2345 \ \% \ 7 &= (2000 + 300 + 40 + 5) \ \% \ 7 \\
&= ((2 \times 1000 \ \% \ 7) + (3 \times 100 \ \% \ 7) + (4 \times 10 \ \% \ 7) + 5) \ \% \ 7 \\
&= ((2 \times 6) \ \% \ 7 + (3 \times 2) \ \% \ 7 + (4 \times 3) \ \% \ 7 + 5) \ \% \ 7 \\
&= (5 + 6 + 5 + 5) \ \% \ 7 \\
&= 21 \ \% \ 7 \\
&= 0
\end{aligned}
$$

8.2.2 The Trick That Makes Rabin-Karp Fast

The trick that makes the Rabin-Karp string searching algorithm efficient is the fact that we have chosen a hash function which is easy to compute for adjacent substrings– once we know what the hash value of the first substring is, we can compute the hash value of the next in time $O(1)$. This is accomplished by taking advantage, once again, of the properties of the modulo operation.

For example, imagine we are that the text and pattern string are drawn from the alphabet consisting only the characters 0 through 9. The text is "234591", and we are searching for a pattern of length 4. We have already computed 2345 % 7. How can we use this to help us compute 3459 % 7?

Observe that 2345 can be transformed into 3459 by the following steps:

1. Multiply 2345 by 10, producing 23450.

2. Subtract 20000 from 23450, leaving 3450. This erases the effect of the "2" digit, which has "shifted" off to the right.

3. Add 9, giving 3459. This adds in the "9" digit.

Using this method results in the following calculation:

$$
\begin{aligned}
3459 \ \% \ 7 &= ((2345 \times 10) - (2 \times 10000) + 9 \ \% \ 7) \ \% \ 7 \\
&= (((2345 \ \% \ 7) \times 10) - (2 \times (10000 \ \% \ 7)) + 9 \ \% \ 7) \ \% \ 7 \\
&= ((0 \times 10) - (2 \times 4) \ \% \ 7 + 9 \ \% \ 7) \ \% \ 7 \\
&= (0 - 1 + 2) \ \% \ 7 \\
&= 1
\end{aligned}
$$

Similarly, once we know 3459 % 7, it is easy to use this knowledge to compute 4591 % 7, in precisely the same manner as above:

$$
\begin{aligned}
4591 \ \% \ 7 &= ((3459 \times 10) - (3 \times 10000) + 1) \ \% \ 7 \\
&= (((3459 \ \% \ 7) \times 10) - (3 \times (10000 \ \% \ 7)) + 1) \ \% \ 7 \\
&= ((1 \times 10) - (3 \times 4 \ \% \ 7) + 1) \ \% \ 7 \\
&= (3 - 5 + 1) \ \% \ 7 \\
&= 6
\end{aligned}
$$

8.2.3 Implementing the Rabin-Karp String Search

We will now explore how to implement this algorithm in C, for strings consisting of C characters.

A hashing function based on the function given in figure 8.1 is shown in figure 8.2. Also note that modulus m is not defined in the function; it is a parameter that can be specified at runtime.

The actual string matching algorithm is given in figure 8.3.

The first action taken is to compute b^M, which will be used later to figure out what amount to subtract from the hash value as characters are "shifted off" to the left.

To gracefully deal with the case where M is large, the divide-and-conquer algorithm for computing exponentials is used. This is the same as the algorithm you explored in the first assignment, except that the modulus is taken early and often to prevent the result from becoming too large. The code for the resulting function is shown in figure 8.4.

The next action is to laboriously hash the pattern and the substring of length M at shift zero of the text. This is the only time we'll hash a substring of the text using the hashing function; all subsequent hashes will be computed by the method described previously.

Figure 8.2: A hashing function.

```
long hash (unsigned char *str, long len, long b, long m)
{
        long i;
        long value = 0;
        long power = 1;

        for (i = len - 1; i >= 0; i--) {
                value += (power * str [i]);
                value %= m;
                power *= b;
                power %= m;
        }
        return (value);
}
```

Before we begin working our way down the string, however, we must check whether shift zero itself is a match. If it is, then we are finished. Otherwise, we loop through the rest of the string, computing each hash based on the previous (using the `nextHash` function, shown in figure 8.5).

8.2.4 Speeding Up Rabin-Karp

Avoiding the Modulo Operation

The implementation of the Rabin-Karp algorithm shown in the previous section is faithful to the original algorithm, but has a flaw that makes it somewhat suboptimal on many modern computers– it relies heavily on the modulo operation, which is usually relatively slow.[1]

However, the modulus operation can be eliminated from the algorithm by choosing a different hashing function that has the same property that it is easy to compute from the previous string, yet does not require so many modulo operations. One way to generate such a function is to keep the same basic function, but always choose a modulus that is a power of 2,

[1]Many modern compilers implement % via division on machines that do not have a `mod` instruction, and many RISC architectures do not have `mod`. Therefore, it seems that the faster your machine is, the slower % runs relative to everything else.

Figure 8.3: The Rabin-Karp String Searching Algorithm

```
long stringSearchRK (unsigned char *P, long P_len,
                unsigned char *T, long T_len,
                long b, long m)
{
        long top_one    = exp_mod (b, P_len, m);
        long P_hash     = hash (P, P_len, b, m);
        long T_hash     = hash (T, P_len, b, m);
        long i;

        if ((T_hash == P_hash) && (memcmp (T, P, P_len) == 0))
                return (0);
        for (i = 0; i <= T_len - P_len; i++) {
                T_hash = nextHash (T_hash, T[i + P_len], T[i],
                                top_one, b, m);
                if ((T_hash == P_hash) && (memcmp (T + i + 1, P, P_len) == 0))
                        return (i + 1);
        }
        return (-1);
}
```

Figure 8.4: An `exp` function that computes $x^n \% m$.

```
unsigned long exp_mod (unsigned long x, unsigned long n, unsigned long m)
{
        if (n == 0)
                return (1);
        else if (n == 1)
                return (x % m);
        else {
                unsigned long square = (x * x) % m;
                unsigned long exp = exp_mod (square, n / 2, m);

                if (n % 2 == 0)
                        return (exp % m);
                else
                        return ((exp * x) % m);
        }
}
```

Figure 8.5: The `nextHash` function.

```
/*
 * nextHash - Compute the next hash value, based on the current.  "in"
 * is the character being shifted in, "out" is the character being
 * shifted out, top_one is b^l (where l is the length of the pattern),
 * b is the base, and m the modulus.
 */
long nextHash (long curr, int in, int out, long top_one, long b, long m)
{
        curr *= b;
        curr -= ((out * top_one) % m);
        curr += in;
        curr %= m;
        if (curr < 0)
                curr += m;
        return (curr);
}
```

so that bit masks can be used instead of the more expensive % operation. Of course, using a modulus that is a power of 2 will have disastrous consequences for the hashing function if the base b used by the function is also a power of 2– which it often will be. However, we have some freedom here as well– all that we need to do is to choose a b that is relatively prime with respect to m– any number larger than the number of characters in the alphabet will do.[2] In this case, using a b of 257 is a safe bet.

A function that implements this change is shown in figure 8.6. Note that this function completely ignores its b and m parameters and instead alway uses $b = 257$ and $m = 1024$. (These parameters are retained only so this function can be used as a drop-in replacement for `stringSearchRK`.)

Avoiding False Matches Through Randomness

The Rabin-Karp algorithm, as presented here, can be very slow if the text contains a lot of "false matches": substrings that hash to the same number as the pattern and therefore cause an expensive `memcmp` to be performed. An intelligent (or lucky) adversary who knows b and m can create texts that make this algorithm perform terribly.

One solution is to choose m and b randomly at runtime, and whenever the number of false matches is high, randomly choose a new m and/or b.

[2]Using a b that is smaller than the number of characters in the alphabet is possible as well, if for some reason this is desirable, but doing so does require a bit more care.

Figure 8.6: A faster implementation of the Rabin-Karp algorithm, which avoids using the mod operation.

```
long stringSearchRKFast (unsigned char *P, long P_len,
                unsigned char *T, long T_len,
                long b, long m)
{
        long fast_b     = 257;
        long fast_m     = 1024;
        long top_one    = exp_mod (fast_b, P_len, fast_m);
        long P_hash     = hash (P, P_len, fast_b, fast_m);
        long T_hash     = hash (T, P_len, fast_b, fast_m);
        long i;

        if ((T_hash == P_hash) && (memcmp (T, P, P_len) == 0))
                return (0);
        for (i = 0; i <= T_len - P_len; i++) {
                T_hash *= fast_b;
                T_hash -= ((T[i] * top_one) & (fast_m - 1));
                T_hash += T[i + P_len];
                T_hash &= (fast_m - 1);
                if (T_hash < 0) {
                        T_hash += fast_m;
                }
                if ((T_hash == P_hash) && (memcmp (T + i + 1, P, P_len) == 0))
                        return (i + 1);
        }
        return (-1);
}
```

8.3 The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm ensures that a string search will never require more than N character comparisons (once some precomputation is performed).

Imagine that we are performing a string search, and the pattern is "AAAB" and the text is "AAAXAAAAA". If we use the brute-force algorithm, then the first test will fail when the "B" in the pattern fails to match the fourth character in the text, which is an "X". At this point, the brute-force algorithm will shift the pattern by one position and start over.

However, this seems silly. In our first attempt to match by brute-force, we learned some key information about text– because we managed to successfully match three characters in the string, *we know what those three characters are and can reuse this information.* More importantly, we know implicitly what the first two characters are of the next substring that we will try– so there is no need to explicitly check them.

The Knuth-Morris-Pratt string searching algorithm uses this information to reduce the number of times it compares each character in the text to a character in the pattern. If properly implemented, the Knuth-Morris-Pratt algorithm only looks at each character in the text once.

The Knuth-Morris-Pratt algorithm, stated in terms of the brute-force algorithm, is given in algorithm 8.3.

Algorithm 8.3 (Knuth-Morris-Pratt) *Knuth-Morris-Pratt string searching*

1. Set $i = 0$, $j = 0$.
2. Match as in the brute-force algorithm, but when a mismatch is detected:
   - If $j = 0$, increment i.
   - Otherwise, let len be the length of the longest suffix of the first j characters in the pattern that appears as a prefix of the first j characters in the pattern.

     Let the *skip* be $j - len$. Increment i by the skip, and decrement j by the skip.

The basic form of the matching algorithm is very similar to the brute-force algorithm– except that instead of always incrementing `i` and setting `j` back to zero at the end of every failed substring test, we set `j` back only as far as the beginning of the possible match, and increment `i` by the corresponding amount.

The difficulty is in figuring out how far to skip when a mismatch occurs. The trick is to precompute a table of skips for each prefix ahead of time, so that in the second step of the algorithm we know immediately how much to increment i and decrement j. Note that this precomputation is based *entirely* upon P, so the cost of this precomputation is not related to the length of the text.

Before writing the code, however, we must clarify exactly what this table of skips will contain. If we have successfully matched x characters in the text against x characters in the pattern, and a mismatch is detected at the next character, how far should we skip? The goal is to skip as far as possible *without missing any potential matches*.

This turns out to be the length of the longest *prefix* of fewer than x characters in the pattern that also appears as a suffix of the first x characters in the pattern. If there is no such prefix, then we know that we can skip over x characters.

It is extremely important to note that the text itself is not considered– if we have already matched x characters in the text, then we know what that text is in terms of the pattern.

If $x = 1$, then the pattern is immaterial– the skip must always be at least one (or else we won't ever make any progress), so we abandon the matched character, skip over it, and begin again at the mismatch.

A table of the skips for several short strings is shown in figure 8.7.

8.3.1 Implementation of Knuth-Morris-Pratt

It is sometimes difficult to think concretely about this sort of problem, because humans have an enormous innate pattern-matching ability, and therefore we have the ability to use this kind of algorithm without really understanding it! Algorithms that make a great deal of intuitive sense can sometimes be difficult to implement elegantly (or even correctly). Figure 8.8 shows code that computes the skip array for a pattern.

Once the `skipArray` is computed, however, the code to actually implement the algorithm is very straightforward. A C implementation of the code to implement algorithm 8.3 is shown in figure and 8.9.

8.3.2 Analysis of Knuth-Morris-Pratt

The careful reader will note that this algorithm does not fulfill the promise that we will look at each character in the text only once. Characters in the midst of a match will only be looked at once, and characters that fail immediately to match the first character in the pattern are looked at only once, but we will look at mismatched characters twice. When a match fails somewhere in the midst of a string, the character that caused the failure is tested

Figure 8.7: The `skipArray` for several patterns. The Prefix/Suffix column shows the substring that is both a prefix and suffix of the first j characters of the pattern. A Prefix/Suffix of **nil** denotes the empty string.

Pattern	**j**	**Skip**	**Prefix/Suffix**
AABAA	2	1	A
AABAA	3	3	**nil**
AABAA	4	3	A
ABABC	2	2	**nil**
ABABC	3	2	A
ABABC	4	2	AB
AAABA	2	1	A
AAABA	3	1	A
AAABA	4	4	**nil**
ABCABC	2	2	**nil**
ABCABC	3	3	**nil**
ABCABC	4	3	A
ABCABC	5	3	AB

Figure 8.8: Code to compute the prefix lengths for the Knuth-Morris-Pratt algorithm.

```
long *skipArrayComputeKMP (unsigned char *P, long M)
{
	long *skipArray;
	long matched, len;

	if (M < 2)
		M = 2;

	skipArray = malloc (M * sizeof (long));

	/*
	 * Indices 0 and 1 are unrelated to the pattern: skipArray[0]
	 * should never be used, and skipArray[1] must always be 1
	 * because if only one character matched before the failure,
	 * the only thing we can do is skip over it-- it can't play a
	 * role in a later match.
	 */

	skipArray[0] = 0;
	skipArray[1] = 1;

	for (matched = 2; matched < M; matched++) {
		for (len = matched - 1; len > 0; len--) {
			if (memcmp (P, &P[matched - len], len) == 0) {
				break;
			}
		}
		skipArray[matched] = matched - len;
	}
	return (skipArray);
}
```

Figure 8.9: The Knuth-Morris-Pratt algorithm, implemented in C.

```
long stringSearchKMP (unsigned char *P, long M, unsigned char *T, long N)
{
        long i, j;
        long *skipArray = skipArrayComputeKMP (P, M);
        long skip;

        j = 0;
        i = 0;
        while (i <= N - M) {
                if (j == M)
                        break;
                else if (P[j] == T[i + j])
                        j++;
                else if (j == 0)
                        i++;
                else {
                        skip = skipArray[j];
                        i += skip;
                        j -= skip;
                }
        }

        free (skipArray);

        if (i <= N - M)
                return (i);
        else
                return (-1);
}
```

again in the next attempt. Although we know what this character is and can make use of this information, the only information we are using is what it *isn't*– it isn't a match.

However, although we may have to look at some characters more than once, the algorithm looks at each character at most twice and so is $O(N)$ (see the exercises for more detail).

We can improve upon the performance of our implementation by extending our table to refine our information about the prefix lengths to include the character that caused the mismatch. This can result in a much larger table, since instead of a one-dimensional table indexed by the number of matching characters, we now need to have a two-dimensional array indexed by both the number of matching characters and the identity of the character that caused the mismatch. Although the table is often sparse, and can be represented in relatively little space, this optimization adds considerably to the memory requirements and/or complexity of the implementation. Whether the added performance is worth the extra memory and complexity is a design decision.

8.3.3 A Knuth-Morris-Pratt Example

This section illustrates the Knuth-Morris-Pratt algorithm with an example.

Let us begin with an example where the pattern is ACAC and the text is ABACABACAC. In figure 8.10, j and i are shown, along with the pattern shifted by i.

Figure 8.10: An illustration of the Knuth-Morris-Pratt algorithm.

```
Text    = ABACABACAC
Pattern = ACAC

          ABACABACAC            i = 0
          ACAC                  j = 0
          ^                     match
          ABACABACAC            i = 0
          ACAC                  j = 1
           ^                    mismatch! skip 1
          ABACABACAC            i = 1
           ACAC                 j = 0
           ^                    mismatch! skip 1
          ABACABACAC            i = 2
            ACAC                j = 0
            ^                   match
          ABACABACAC            i = 2
            ACAC                j = 1
             ^                  match
          ABACABACAC            i = 2
            ACAC                j = 2
              ^                 match
          ABACABACAC            i = 2
            ACAC                j = 3
               ^                mismatch! skip 2
          ABACABACAC            i = 4
              ACAC              j = 0
              ^                 match
          ABACABACAC            i = 4
              ACAC              j = 1
               ^                mismatch! skip 1
          ABACABACAC            i = 5
               ACAC             j = 0
               ^                mismatch! skip 1
          ABACABACAC            i = 6
                ACAC            j = 0
                ^               match
  (Final four matches omitted)  MATCH at i = 6
```

8.4 The Boyer-Moore Algorithm

Once again, imagine that we are performing a string search, and the pattern is "AAAB" and the text is "AAAXAAAB".

If we skip ahead and look at the fourth character of the text, we can immediately note that this character appears nowhere in the pattern. Therefore, if there is a match anywhere in this text, *it cannot possibly overlap with the fourth character in the text.* Since the pattern is of length 4, that means that the smallest possible shift that needs to be considered is 4. Therefore, just by checking a single character, we have completely eliminated the need to look at the first three characters in the text!

This kind of skip is the goal of the Boyer-Moore algorithm– unlike the Knuth-Morris-Pratt algorithm, which strives to look at each character in the text only once, the Boyer-Moore algorithm strives to completely ignore as many characters in the text as possible. The algorithm itself is given in algorithm 8.4.

Once again, the algorithm is very simple, but the computation of the skips is not. As we did in the Knuth-Morris-Pratt algorithm, we will precompute an array of skips, but unlike the Knuth-Morris-Pratt algorithm, this array will be two-dimensional– it will take into account both the number of characters successfully matched (j) and the identity of the character in the text (c) that caused the mismatch.

It is much easier to understand the computation of the skips from an intuitive perspective and then see how this algorithm obeys our intuition than it is to prove that this algorithm is correct and complete, so that is how I will proceed.

In the calculation of the skips, there are two basic cases that we need to consider in algorithm 8.4:

1. For $d \leq j$, then the new shift must overlap all of the pattern that we have already successfully matched. Therefore, it must match all of the pattern we have already matched (because otherwise, a mismatch is inevitable). In addition, the character that caused the mismatch in the current shift must match its counterpart in the pattern in the new shift (because, once again, otherwise we know that a mismatch is inevitable if this character does not match the new shift of the pattern).

2. If $d > j$, then we are shifting "past" the part of the pattern that we have already matched– we are shifting far enough so that the new shift moves past some or all of the characters already matched.

   We have determined that the mismatched character cannot appear anywhere among the characters we have already matched, because there is no shift that contains it and

Algorithm 8.4 (Boyer-Moore) *Boyer-Moore string searching.*

Let P be the pattern and M its length. Let T be the pattern and N its length.
Let i be the shift, which is initially zero. Execute the following loop, until a match has been found or the end of the text is reached (when $i = N - M$). (If the end of the text is reached, return -1 to indicate failure.)

1. Attempt to match the pattern to the text at shift i, starting from the end of the pattern and moving towards the front. If the pattern completely matches the text at this shift, return i. Otherwise, let j represent the index in the pattern where the mismatch occured. (The number of successfully matched characters is therefore $M - j$.)

2. Advance i by d positions. The skip d is chosen to be the smallest d that satisfies the following conditions:

   Let $\mathbf{start} = \mathbf{max}(j + 1, d)$, and let $\mathbf{len} = M - \mathbf{start}$.

   - $d \leq j$ and the substring of length **len** starting at offset $\mathbf{start} - d$, in P matches the substring starting at offset **start**, and $P[j - d] == c$.
   - $d > j$ and the substring of length d starting at offset 0 in P matches the substring of length **len** that starts at offset **start** in P.

would be consistent with the other characters we have already matched– if there was, we would have already found it when we were investigating the first case. Therefore, we must skip past the characters we have already matched in the text, looking for a way to match them up with the pattern.

The only question is how far to skip– and this is determined, in a similar manner as the Knuth-Morris-Pratt algorithm, by finding the length of the longest suffix of the matched characters that matches a prefix of the pattern.

Note that once again, we always choose the smallest skip that has not already been proven to lead to a mismatch. This prevents the mishap of skipping past a match.

8.4.1 Implementation of Boyer-Moore

The code that implements the heart of the Boyer-Moore algorithm is shown in figure 8.11, and the code that implements the computation of the skip array is shown in figure 8.12.

As in the implementation of the Knuth-Morris-Pratt algorithm, no special effort is taken to make sure that this implementation is efficient– in particular, the algorithm used to compute the skip array is quite inefficient. It is possible to compute the skip array in time $O(M)$ (rather than $O(M^3)$ as this code does), but the algorithm for doing so is quite complex.[3] This means that for long patterns or short texts, this implementation is quite a bit slower than any of the other algorithms shown in this chapter. However, once the text length grows large, the true speed of this algorithm becomes apparent– for texts of more than a few hundred kilobytes, this algorithm is roughly twice as fast as any of the others.[4] Clearly, when the text is long, and speed is important, an optimized Boyer-Moore function will give superior performance.

8.4.2 A Boyer-Moore Example

This section illustrates the Boyer-Moore algorithm with an example.

Let us begin with an example where the pattern is ACAC and the text is ABACABACAC. In figure 8.13, j and i are shown, along with the pattern shifted by i:

[3] It would make a **wonderful** (and ambitious) final project for someone to nail down the $O(M)$ algorithm for computing the skip array.

[4] Using randomly generated text and pattern strings.

Figure 8.11: Code that implements the Boyer-Moore algorithm.

```
long stringSearchBM (unsigned char *P, long M, unsigned char *T, long N)
{
        long i, j;

        skipArrayComputeBM (P, M);

        i = 0;
        while (i <= N - M) {
                j = M - 1;
                while ((j >= 0) && (P[j] == T[i + j]))
                        j--;
                if (j < 0)
                        break;
                i += skipArray[j][T[i + j]];
        }
        if (i <= N - M)
                return (i);
        else
                return (-1);
}
```

Figure 8.12: Code to compute the skipArray for the Boyer-Moore algorithm.

```
void skipArrayComputeBM (unsigned char *P, long M)
{
        long c, j;

        for (j = 0; j < M; j++)
                for (c = 0; c < ALPHA_SIZE; c++)
                        skipArray [j][c] = prefixComputeBM (P, M, j, c);
        return ;
}

long prefixComputeBM (unsigned char *P, long M, long j, long c)
{
        long d, start, len;

        for (d = 1; d < M; d++) {
                if (d <= j) {
                        start = j + 1;
                        len = M - start;

                        if ((memcmp (&P[start - d], &P[start], len) == 0) &&
                                        (P[j - d] == c)) {
                                return (d);
                        }
                }
                else {
                        len = M - d;
                        if (memcmp (P, &P[d], len) == 0)
                                return (d);
                }
        }
        return (d);
}
```

Figure 8.13: An illustration of the Boyer-Moore algorithm.

```
Text    = ABACABACAC
Pattern = ACAC

          ABACABACAC            i = 0
          ACAC                  j = 3
             ^                  match

          ABACABACAC            i = 0
          ACAC                  j = 2
            ^                   match

          ABACABACAC            i = 0
          ACAC                  j = 1
           ^                    mismatch! skip 2

          ABACABACAC            i = 2
            ACAC                j = 3
               ^                mismatch! skip 4

          ABACABACAC            i = 6
                ACAC            j = 3
                   ^            match

          ABACABACAC            i = 6
                ACAC            j = 2
                  ^             match

          ABACABACAC            i = 6
                ACAC            j = 1
                 ^              match

          ABACABACAC            i = 6
                ACAC            j = 0
                ^               match
                                MATCH at i = 6
```

8.5 Exercises

1. The Rabin-Karp algorithm can be used very effectively for 2-dimensional pattern matching– finding whether a “pattern” array of characters appears as a subarray of a “text” array of characters.

   Show how the algorithm could be changed for the two-dimensional case, and generalize your algorithm to work with higher numbers of dimensions.

2. Find at least one other hashing function that can be used with the Rabin-Karp string matching algorithm. Try to find one that is efficient to compute.

3. Modify the `stringSearchRK` function so that it randomly selects a new prime m whenever the number of mismatches exceeds $4/m$ of the total number of characters processed at any point in the text. You may assume the existence of an array of suitable primes.

4. What is the maximum number of times that any character in the text will be looked at by the `stringSearchKMP` function? Is it possible that adjacent characters in the text will be considered this many times?

5. The way that the `skipArray` is computed for the Knuth-Morris-Pratt string searching algorithm is quite inefficient. This means that searching for relatively long patterns in relatively short texts is inefficient with this function, because the time necessary to compute the `skipArray` is comparable to the time necessary to perform the search itself!

   Show how the computation of the `skipArray` can be significantly improved. (It can be done in time $O(M)$.) This is not an easy problem.