13 String Matching

String matching is important for virtually each computer user. When editing a text, the user processes it, organizes it into paragraphs and sections, reorders it, and, very often, searches for some subtext or pattern in the text to locate the pattern or replace it with something else. The larger the text that is being searched for, the more important is the efficiency of the searching algorithm. The algorithm cannot usually rely on, say, alphabetical ordering of words, as would be the case with a dictionary. For example, string searching algorithms are increasingly important in molecular biology, where they are used to extract information from DNA sequences by locating some pattern in them and comparing the sequences for common subsequences. Such processing has to be done frequently, under the assumption that an exact match cannot be expected. Problems of that type are addressed by what is often called *stringology*, whose major area of interest is *pattern matching*. Some stringological problems are discussed in this chapter.

This chapter uses the following notation: For a text T, which is a sequence of symbols, characters, or letters, $|T|$ signifies the length of T, T_j is the character at position j of T, and $T(i \ldots j)$ is a substring of T that begins at position i and ends at j. The first characters in pattern P and text T are in position 0. Also, a *regular expression* a^n stands for a string $a \ldots a$ of n as.

13.1 Exact String Matching

Exact string matching consists of finding an exact copy of pattern P in text T. It is an all-or-nothing approach; if there is a very close similarity between P and a substring of T, the partial match is rejected.

13.1.1 Straightforward Algorithms

A simple approach to string matching is starting the comparison of P and T from the first letter of T and the first letter of P. If a mismatch occurs, the matching begins from the second character of T, and so on. Any information that can be useful in subsequent tries is not retained. The algorithm is given in this pseudocode:

```
bruteForceStringMatching(pattern P, text T)
    i = 0;
    while i ≤ |T| - |P|
        j = 0;
        while T_i == P_j and j < |P|
            i++;                              // try to match all characters in P;
            j++;
        if j == |P|
            return match at i - |P|;          // success if the end of P is reached;
                                              // if there is a mismatch,
        i = i - j + 1;                        // shift P to the right by one position;
    return no match;                          // failure if fewer characters left in T than |P|;
```

In the worst case the algorithm executes in $O(|T||P|)$ time. For example, if $P = a^{m-1}b$ and $T = a^n$, then the algorithm makes $(n - (m - 1))m = nm - m^2 + m$ comparisons, which is approximately nm for a large n and small m.

The average performance depends on the probability distribution of the characters in both the pattern and the text. As an example, assume that only two characters are used, and the probability of using any of the two characters equals 1/2. In this case, for a particular scan i, the probability equals 1/2 that only one comparison is made, the probability $1/2 \cdot 1/2 = 1/4$ that two comparisons are made, . . . , and the probability $1/2 \cdot \ldots \cdot 1/2 = 2^{-|P|}$ that m comparisons are performed; that is, on average, for a given i, the number of comparisons equals

$$\sum_{k=1}^{|P|} \frac{k}{2^k} < 2$$

so that the average number of comparisons for all the scans equals $2(|T| - (|P| - 1)) < 2|T|$ for a large $|T|$. A far better estimate, $2^{|P|+1} - 2$, is found using the theory of absorbing Markov chains, and, more generally, for an alphabet A, the average number of comparisons is $(|A|^{|P|+1} - |A|) / (|A| - 1)$ (Barth, 1985).

Here is an example of execution of the brute force algorithm for T = *ababcdabbabababad* and P = *abababa*:

```
  ababcdabbabababad
1 abababa
2  abababa
3   abababa
4    abababa
5     abababa                                  (13-1)
6      abababa
7       abababa
8        abababa
9         abababa
10         abababa
```

The corresponding characters in P and T are compared—which is marked by underlining characters in P—starting at the position where P is currently aligned with T.

After a mismatch is found, the scan through P and T is aborted and restarted after P is shifted to the right by one position. In the first iteration, the matching process begins at the first characters of P and T, and a mismatching occurs at the fifth character of $T(T_4 = c)$ and the fifth character of $P(P_4 = a)$. The next round begins from the first character of P, but this time, from the second character of T, which immediately leads to a mismatch. The third iteration reaches the third character of P,*a*, and the fifth character of T,*c*. The match of the entire pattern P is found in the tenth iteration.

Note that no actual shifting takes place; the shifting is accomplished by updating index *i*.

An improvement is accomplished by a not-so-naïve algorithm proposed by Hancart (1992). It begins comparisons from the second character of P, goes to the end, and ends comparisons with the first character. So the order of characters involved in comparisons is $P_1, P_2, \ldots, P_{|P|-1}, P_0$.

The information about equality of the first two characters of P is recorded and used in the matching process. Two cases are distinguished: $P_0 = P_1$ and $P_0 \neq P_1$. In the first case, if $P_1 \neq T_{i+1}$, text index *i* is incremented by 2, because $P_0 \neq T_{i+1}$; otherwise, *i* is incremented by 1. It is similar in the second case, if $P_1 = T_{i+1}$. In this way, a shift by two positions is possible. Here is the algorithm:

```
Hancart(pattern P, text T)
    if P0 == P1
         sEqual = 1;
         sDiff = 2;
    else sEqual = 2;
         sDiff = 1;
    i = 0;
    while i ≤ |T| - |P|
        if Ti+1 ≠ P1
             i = i + sDiff;
        else j = 1;
             while j < |P| and Ti+j == Pj
                  j++;
             if j == |P| and P0 == Ti
                  return match at i;
             i = i + sEqual;
    return no match;
```

Matching begins from the second pattern character. If there is a mismatch between P_1 and T_{i+1}, then P can be shifted by two positions before beginning the next round, as long as the first two characters of P are the same, because this mismatch means that P_0 and T_{i+1} are also different:

```
  i
  ↓
  acaaca
1 aab
2   aab
```

However, after a mismatch occurs in the inner `while` loop, the pattern is shifted by only one position:

```
      i
      ↓
   acaaca
 2   aab
 3    aab
```

On the other hand, if the first two characters of P are different, then after noticing in the `if` statement that P_1 and T_{i+1} are different, P is shifted by one position only:

```
    i
    ↓
   aabaca
 1 abb
 2  abb
```

so that a possible occurrence of P is not missed. However, after a mismatch is found in any other position, P is shifted by two places:

```
      i
      ↓
   aabaca
 2  abb
 3    aab
```

This can be done safely, because P_1 and T_{i+1} have just been determined as equal and because P_0 and P_1 are different, P_0 and T_{i+1} must be also different, thus there is no need to check this in the third iteration. Here is another example:

```
   ababcdabbabababad
 1 abababa
 2   abababa
 3     abababa
 4      abababa
 5       abababa
 6         abababa
 7          abababa
```

In the worst case, the algorithm executes in $O(|T||P|)$ time, but, as Hancart shows, it performs on average better than some of the more developed algorithms to be discussed in the next section.

13.1.2 The Knuth-Morris-Pratt Algorithm

The brute force algorithm is inefficient in that it shifts pattern P by one position after a mismatch is found. To speed up the process, Hancart's algorithm allows for a shift by two characters. However, a method is needed to shift P by as many positions to the right as possible but so that no match is missed.

The source of inefficiency of the brute force algorithm lies in performing redundant comparisons. The redundancy can be avoided by observing that pattern P includes identical substrings at the beginning of P and before the mismatched character. This fact can be used to shift P to the right by more than one position before beginning the next scan. Consider line 1 of the following diagram. The mismatch occurs at the fifth character, but until that point, both the prefix ab of P and the substring $P(2 \ldots 3)$, which is also ab, have been successfully processed. P can now be moved to the right to align its substring ab with the substring $T(2 \ldots 3)$ and the matching process can start from character P_2 and from the mismatched character in T, T_4. Because characters in the substring $P(2 \ldots 3)$ have just been successfully matched with $T(2 \ldots 3)$, it is as though the characters in the prefix $P(0 \ldots 1)$ were matched with $T(2 \ldots 3)$. In this way, the two redundant comparisons in line 2 can be omitted. After the mismatch

```
          i
          ↓
  ababcdabbababababad
1 abababa
      ↑
      j
```

the matching process continues as in

```
          i
          ↓
  ababcdabbababababad
2   abababa
      ↑
      j
```

thereby skipping $ab = P(0 \ldots 1)$. The two identical parts relevant to this shift are the prefix of P and suffix of this part of P that is currently successfully matched, which is the prefix $P(0 \ldots 1)$ and the suffix $P(2 \ldots 3)$ of the matched part of P, $P(0 \ldots 3)$.

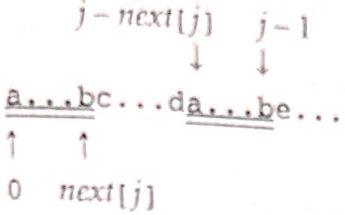
Generally, to perform a shift, we first need to match a prefix of P with a suffix of $P(0 \ldots j)$, where P_{j+1} is a mismatched character. This matching prefix should be the longest possible so that no potential match is passed after shifting P; that is, if the match is of length *len* and the current scan starts at position k of T, then no occurrence of P should begin in any position between k and $k + len$, but it may begin at position $k + len$, so that shifting P by *len* positions is safe.

This information will be used many times during the matching process; therefore, P should be preprocessed. Importantly, in this approach only the information about P is used; the configuration of characters in T is irrelevant.

Define the table *next*:

$$next[j] = \begin{cases} -1 & \text{for } j = 0 \\ \max\{k{:}0 < k < j \text{ and } P[0 \ldots k-1] = P[j-k \ldots j-1]\} & \text{if such a } k \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

that is, the number $next[j]$ indicates the length of the longest suffix of substring $P(0 \ldots j-1)$ equal to a prefix of the same substring:



The condition $k < j$ indicates that the prefix is also a proper suffix. Without this condition, $next[2]$ for $P(0 \ldots 2) = aab$ would be 2, because aa is at the same time a prefix and suffix of aa, but with the additional condition, $next[2] = 1$, not 2.

For example, for $P = abababa$,

P	a	b	a	b	a	b	a
j	0	1	2	3	4	5	6
$next[j]$	-1	0	0	1	2	3	4

Note that because of the condition requiring that the matching suffix be the longest, $next[5] = 3$ for $P(1 \ldots 6) = ababab$, because aba is the longest suffix of $ababa$ matching its prefix (they overlap), not 1, although a is also both a prefix and a suffix of $ababa$.

The Knuth-Morris-Pratt algorithm can be obtained relatively easily from `bruteForceStringMatching()`:

```
KnuthMorrisPratt(pattern P, text T)
    findNext(P,next);
    i = j = 0;
    while i ≤ |T| - |P|
        while j == -1 or j < |P| and T_i == P_j
            i++;                    // increment i only for matched characters;
            j++;
        if j == |P|
            return a match at i - |P|;
        j = next[j]                 // in the case of a mismatch, i does not change;
    return no match;
```

The algorithm `findNext()` to determine the table *next* will be defined shortly. For example, for $P = abababa$, $next = [-1\ 0\ 0\ 1\ 2\ 3\ 4]$, and $T = ababcdabbabababad$, the algorithm executes as follows:

```
  ababcdabbabababad
1 abababa
2   abababa
3     abababa
4      abababa
5       abababa
6         abababa
7          abababa
```

The diagram indicates that –1 in *next* means that the entire pattern P should be shifted past the mismatched text character; see the shift from line 4 to 5 and from line 6 to 7. One major difference between `bruteForceStringMatching()` and `KnuthMorrisPratt()` is that i is never decremented in the latter algorithm. It is incremented in the case of a match; in the case of a mismatch i stays the same so that the mismatched character in T is compared to another character in P in the next iteration of the outer `while` loop. The only case when i is incremented in the case of mismatch is when the first character in P is a mismatched character; to this end the subcondition `j == -1` is needed in the inner loop. After finding a mismatch at position $j \neq 0$ of P, P is shifted by $j - next[j]$ positions; when the mismatch occurs at the first position of P, the pattern is shifted by one position.

To assess the computational complexity of `KnuthMorrisPratt()`, note that the outer loop executes $O(|T|)$ times. The inner loop executes at most $|T| - |P|$ times, because i is incremented in each iteration of the loop, and by the condition on the outer loop, $|T| - |P|$ is the maximum value for i. But for a mismatched character T_i, j can be assigned a new value $k \leq |P|$ times. When this happens, the first character in P, for which the mismatch occurs, is aligned with the character T_{i+k}. Consider $P = aaab$ and $T = aaacaaadaaab$. In this case, $next = [-1\ 0\ 1\ 2]$, and the trace of the execution of the algorithm is as follows:

```
   aaacaaadaaab
1  aaab
2   aaab
3    aaab
4     aaab
5      aaab
6       aaab
7        aaab
8         aaab
9          aaab
```

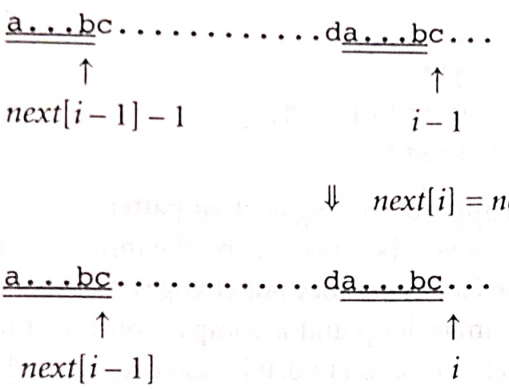
The mismatched c in T is compared to four characters in P in lines 1 through 4 because b is the fourth character in P for which the mismatch occurs for the first time and because b is aligned with c; that is, all the preceding characters have already been matched successfully. The next time, such a situation occurs for d in T and again for b in P on line 5, and by this time all the preceding characters in P are successfully matched. This means, that for some i, $|P|$ comparisons can be performed, but this can not happen for every i, but only for every $|P|^{\text{th}}$ i, so that the number of unsuccessful comparisons can be up to $|P|(|T|/|P|) = |T|$. Up to $|T| - |P|$ successful comparisons have to be added to this number to obtain the running time $O(|T|)$.

The table *next* still remains to be determined. We can use the brute force algorithm to that end, which is not necessarily inefficient for short patterns. But we can also adapt the Knuth-Morris-Pratt algorithm to improve the efficiency of determining *next*.

Remember that *next* contains the lengths of the longest suffixes matching prefixes of P; that is, parts of P are being matched with other parts of P. But the problem of matching is solved already by the Knuth-Morris-Pratt algorithm. In this case, P is matched against itself. However, the Knuth-Morris-Pratt algorithm uses *next*, which is

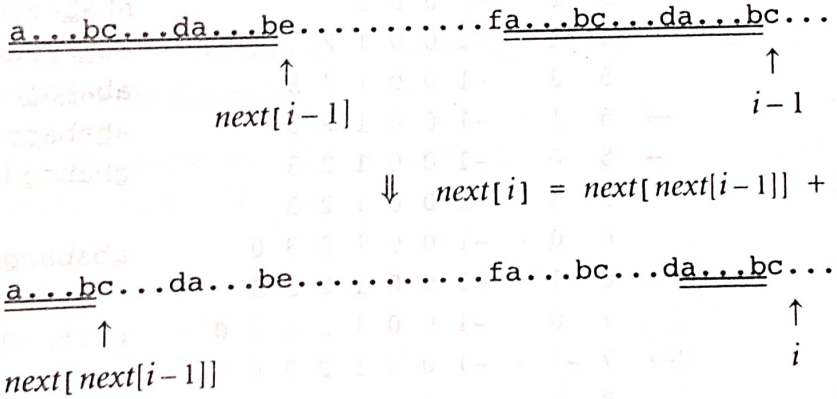
still unknown. Therefore, the Knuth-Morris-Pratt algorithm has to be modified so that it determines the values of the table *next* by using values already found. Let $next[0] = -1$. Assuming that values $next[0], \ldots, next[i-1]$ have already been determined, we want to find the value $next[i]$. There are two cases to consider.

In the first case, the longest suffix matching a prefix is found by simply attaching the character P_{i-1} to the suffix corresponding to position $next[i-1]$, which is true when $P_{i-1} = P_{next[i-1]}$:



In this case, the current suffix is longer by one character than the previously found suffix so that $next[i] = next[i-1] + 1$.

In the second case, $P_{i-1} \neq P_{next[i-1]}$. But this is simply a mismatch, and a mismatch can be handled with the table *next*, which is why it is being determined. Because $P_{next[i-1]}$ is a mismatched character, we need to go to $next[next[i-1]]$ to check whether P_{i-1} matches $P_{next[next[i-1]]}$. If they match, $next[i]$ is assigned $next[next[i-1]] + 1$:



otherwise P_{i-1} is compared to $P_{next[next[next[i-1]]]}$ to have $next[i] = next[next[next[i-1]]] + 1$ if the characters match; otherwise, the search continues until a match is found or the beginning of P is reached.

Note that in the previous diagram, the first prefix $a \ldots bc \ldots da \ldots b$ of $P(0 \ldots i-1)$ has a prefix $a \ldots b$ identical to its suffix. This is not an accident. The reason for $a \ldots b$ being both prefix and suffix of $a \ldots bc \ldots da \ldots b$ when $a \ldots b$ is about to be found as the longest prefix and suffix of $P(0 \ldots i-1)$ is as follows. The prefix $P(0 \ldots j-1) = a \ldots bc \ldots da \ldots b$ of $P(0 \ldots i-1)$ indicated by $next[i-1]$ is, by definition, equal to the suffix $P(i-j-1 \ldots i-2)$, which means that the suffix $P(j - next[j] \ldots j-1) = a \ldots b$ is also a suffix of $P(i-j-1 \ldots i-2)$. Therefore, to determine the value of $next[i]$ we refer to the already determined value $next[j]$ that specifies the length of this shorter suffix of $P(0 \ldots j-1)$ matching a prefix of P, and thus the length of the suffix $a \ldots b$ of $P(0 \ldots i-1)$ matching the same prefix.

The algorithm to find the table *next* is as follows:

```
findNext(pattern P, table next)
    next[0] = -1;
    i = 0;
    j = -1;
    while i < |P|
        while j == 0 or i < |P| and P_i == P_j
            i++;
            j++;
            next[i] = j;
        j = next[j];
```

Here is an example of finding *next* for pattern $P = ababacdd$. The values of indices i and j and the table *next* before entering the inner `while` loop are indicated with an arrow (and by the fact that i does not change); the remaining lines show these values at the end of the inner loop and a comparison that follows. For example, in line 2, after incrementing i to 1 and j to 0, 0 is assigned to *next*[1], and then the first and second characters of P are compared, which leads to exiting the loop.

```
      i  j      next[]          P
→     0 -1    -1                ababacdd
      1  0    -1 0              ababacdd
→     1 -1    -1 0
      2  0    -1 0 0            ababacdd
      3  1    -1 0 0 1          ababacdd
      4  2    -1 0 0 1 2        ababacdd
      5  3    -1 0 0 1 2 3      ababacdd
→     5  1    -1 0 0 1 2 3      ababacdd
→     5  0    -1 0 0 1 2 3      ababacdd
→     5 -1    -1 0 0 1 2 3
      6  0    -1 0 0 1 2 3 0    ababacdd
→     6 -1    -1 0 0 1 2 3 0
      7  0    -1 0 0 1 2 3 0 0  ababacdd
→     7 -1    -1 0 0 1 2 3 0 0
      8  0    -1 0 0 1 2 3 0 0
```

Because of the similarity of this algorithm and the Knuth-Morris-Pratt algorithm, we conclude that *next* can be determined in $O(|P|)$ time.

The outer `while` loop in `KnuthMorrisPratt()` executes in $O(|T|)$ time, so the Knuth-Morris-Pratt algorithm, including `findNext()`, executes in $O(|T| + |P|)$ time. Note that in the analysis of the complexity of the algorithm, no mention was made about the alphabet underlying the text T and pattern P; that is, the complexity is independent of the number of different characters constituting P and T.

The algorithm requires no backtracking in text T; that is, the variable i is never decremented during execution of the algorithm. This means that T can be processed one character at a time, which is very convenient for online processing.

The Knuth-Morris-Pratt algorithm can be improved if we eliminate unpromising comparisons. If the mismatch occurs for characters T_i and P_j, then the next match

is attempted for the same character T_i and character $P_{next[j]+1}$. But if $P_j = P_{next[j]+1}$ then the same mismatch takes place, which means a redundant comparison is made. Consider P = *abababa* and T = *ababcdabbabababad*, analyzed earlier, for which *next* = [−1 0 0 1 2 3 4] and the Knuth-Morris-Pratt algorithm begins with:

```
  ababcdabbabababad
1 abababa
2   abababa
```

The first mismatch occurs for *a* at the fifth position of *P* and for *c* in *T*. The table *next* indicates that in the case of the mismatch of the fifth character of *P*, *P* should be shifted by two positions to the right, because 4 − *next*[4] = 2; that is, the two-character prefix of *P* should be aligned with the two-character suffix of *P*(0 : . . 3). The situation is illustrated on the second line of the diagram. However, this means that the next comparison is made between *c* that just caused a mismatch and *a* at the third position of *P*. But this is a comparison that has just been made on line 1 of the diagram, where *a* in the fifth position of *P* was also compared to *c*. Therefore, if we knew that the prefix *ab* of *P* is followed by *a*, which is also a character following suffix *ab* of *P*(0 . . . 3), then the situation of the second line of the diagram could be avoided. To accomplish it, the table *next* has to be redesigned to exclude such redundant comparisons. This is done by extending the definition of *next* by one more condition, which leads to the following definition of a stronger *next*:

$$nextS[j] = \begin{cases} -1 & \text{for } j = 0 \\ \max\{k{:}0 < k < j \text{ and } P[0 \ldots k-1] = P[j-k \ldots j-1] \text{ and } P_{k+1} \neq P_j\} & \text{If such a } k \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

To compute *nextS*, the algorithm `findNext()` needs to be modified slightly to account for the additional condition, as in

```
findNextS(pattern P, table nextS)
    nextS[0] = -1;
    i = 0;
    j = -1;
    while i < |P|
        while j == -1 or i < |P| and P_i == P_j
            i++;
            j++;
            if P_i ≠ P_j
                nextS[i] = j;
            else nextS[i] = nextS[j];
        j = nextS[j];
```

The rationale is as follows. If $P_i \neq P_j$—that is, the new subcondition defining *nextS* is satisfied—then clearly *next*[*i*] and *nextS*[*i*] are equal and so *nextS*[*i*] in `findNextS()` is assigned the same value as *next*[*i*] in `findNext()`. If the characters P_i and P_j are equal,

```
a...bc...da...be...fa...bc...da...be...
                  ↑   ↑               ↑
                  j  i-j              i
```

then the subcondition is violated, thus $nextS[i] < next[i]$, and the situation is as follows

```
a...bc...da...be...fa...bc...da...be...
          ↑    ↑              ↑    ↑
  j – nextS[j]  j          i – nextS[i]  i
```

The underlined substrings are the proper prefix and suffix of $P(0 \ldots i - 1)$ indicated by $nextS[i]$ that are shorter than $next[i]$ (they can be empty). But the prefix $P(0 \ldots j-1) = a \ldots bc \ldots da \ldots b$ of $P(0 \ldots i-1)$ indicated by $next[i]$ is, by definition, equal to the suffix $P(i-j \ldots i-1)$, which means that the suffix $P(j - nextS[j] \ldots j - 1) = a \ldots b$ shown in italic is also a suffix of $P(i-j \ldots i-1)$. Therefore, to determine the value of $nextS[i]$ we refer to the already determined value $nextS[j]$ that specifies the length of the italicized suffix of $P(0 \ldots j-1)$ matching a prefix of P, and thus the length of the suffix of $P(0 \ldots i-1)$ matching the same prefix. If the prefix is followed by the character P_j, then $nextS[j]$ contains the length of a shorter prefix determined by the same process. For example, when processing position 11 in the string

```
P     = abcabdabcabdfabcabdabcabd
nextS = .....2.....2............
```

number 2 is copied to $nextS[11]$ from $nextS[5]$, and the same number from position 11—that is, indirectly, from position 5—to $nextS[24]$:

```
P     = abcabdabcabdfabcabdabcabd
nextS = .....2.....2...........2
```

The Knuth-Morris-Pratt algorithm is modified by replacing `findNext()` with `findNextS()`. The execution of this algorithm for $P = abababa$ generates $nextS = [-1\ 0\ -1\ 0\ -1\ 0\ -1]$ and then continues with comparisons as summarized in this diagram:

```
  ababcdabbabababad
1 abababa
2      abababa
3       abababa
4          abababa
```

The Knuth-Morris-Pratt algorithm exhibits the worst case performance for Fibonacci words defined recursively as follows:

$$F_1 = b, F_2 = a, F_n = F_{n-1}F_{n-2} \text{ for } n > 2$$

The words are: *b, a, ab, aba, abaab, abaababa, ...*

In the case of mismatch, a Fibonacci word F_n can be shifted $\log_{\varphi}|F_n|$ times, where $\varphi = (1 + \sqrt{5})/2$ is the golden ratio. If the pattern $P = F_7 = abaababaabaab$, the Knuth-Morris-Pratt algorithm executes as follows:

```
  abaababaabaca...
1 abaababaabaa
2      abaabab
3         abaa
4           ab
5            a
6             a...
```

13.1.3 The Boyer-Moore Algorithm

In the Knuth-Morris-Pratt algorithm, each of the first $|T| - |P| + 1$ characters is used at least once in a comparison for an unsuccessful search. The source of this algorithm's better efficiency over the brute force approach lies in not starting the matching process from the beginning of pattern P when a mismatch is detected if possible. So the Knuth-Morris-Pratt algorithm goes through almost all characters in T from left to right and tries to minimize the number of characters in P involved in matching. It is not possible to skip any characters in T itself to avoid unpromising comparisons. To accomplish such skipping, the Boyer-Moore algorithm tries to match P with T by comparing them from right to left, not from left to right. In the case of a mismatch, it shifts P to the right and always begins the next matching from the end of P, but it shifts P to the right so that many characters in T are not involved in the comparisons. Thus, the Boyer-Moore algorithm attempts to gain speed by skipping characters in T rather than, as the Knuth-Morris-Pratt algorithm does, skipping them in P, which is more prudent because the length of P is usually negligible in comparison to the length of T.

The basic idea is very simple. In the case of detecting a mismatch at character T_i, P is shifted to the right to align T_i with the first encountered character equal to T_i, if such a character exists. For example, for T = *aaaaebdaabadbda* and P = *dabacbd*, first, characters $T_6 = d$ and $P_6 = d$, then characters $T_5 = b$ and $P_5 = b$ are compared, and then the first mismatch is found at $T_4 = e$ and $P_4 = c$. But there is no occurrence of e in P. This means that there is no character in P to be aligned with e in T; that is, no character can be successfully matched with e. Therefore, P can be shifted to the right past the mismatched character:

```
  aaaaebdaabadbda
1 dabacbd
2     dabacbd
```

In this way, the first four characters of the text are excluded from later comparisons. Now, matching starts from the end of P and from position 11 = 4 + 7 = (the position of the mismatched character T_4) + $|P|$. A mismatch is found at $T_{10} = a$ and $P_5 = b$, and then the mismatched a is aligned with the first a to the left of mismatched P_5:

```
  aaaaebdaabadbda
2     dabacbd
3       dabacbd
```

that is, the position in T from which the matching process starts in the third line is 13 = 10 + 3 = (the position of the mismatched character $T_{10} = a$) + ($|P|$ – position of the rightmost a in P). After matching characters T_{13} with P_6 and T_{12} with P_5, a mismatch is found at $T_{11} = d$ and $P_4 = c$. If we aligned the mismatched d in text with the rightmost d in P, P would be moved backwards. Therefore, if there is a character in P equal to the mismatched character in T to the left of the mismatched character in P, the pattern P is shifted to the right by one position only:

```
  aaaaebdaabadbda
3       dabacbd
4        dabacbd
```

To sum up, the three rules can be termed character occurrence rules:

1. *No occurrence rule.* If the mismatched character T_i appears nowhere in P, align P_0 with T_{i+1}.
2. *Right side occurrence rule.* If there is a mismatch at T_i and P_j, and if there is an occurrence of character *ch* equal to T_i to the *right* of P_j, shift P by one position.
3. *Left side occurrence rule.* If there is an occurrence of character *ch* equal to T_i *only* to the *left* of P_j, align T_i with $P_k = ch$ closest to P_j.

To implement the algorithm, a table *delta1* specifies, for each character in the alphabet, by how much to increment *i* after a mismatch is detected. The table is indexed with characters and is defined as follows:

$$delta1[ch] = \begin{cases} |P| & \text{if } ch \text{ is not in } P \\ \min\{|P| - i - 1 : P_i = ch\} & \text{otherwise} \end{cases}$$

For the pattern P = *dabacbd*, *delta1*['a'] = 3, *delta1*['b'] = 1, *delta1*['c'] = 2, *delta1*['d'] = 0, and for remaining characters *ch*, *delta1*[*ch*] = 7.

The algorithm itself can be summarized as follows:

```
BoyerMooreSimple(pattern P, text T)
    initialize all cells of delta1 to |P|;
    for j = 0 to |P| - 1
        delta1[Pj] = |P| - j - 1;
    i = |P| - 1;
    while i < |T|
        j = |P| - 1;
        while j ≥ 0 and Pj == Ti
            i--;
            j--;
        if j == -1
            return match at i+1;
        i = i + max(delta1[Ti],|P|-j);
    return no match;
```

In the algorithm, *i* is incremented by *delta1*[T_i] if the character T_i that caused a mismatch has in P an equivalent to the left of character P_j that caused the same mismatch and none to its right, which means shifting P to the right by *delta1*[T_i] – ($|P| - j$) positions; otherwise, *i* is incremented by $|P| - j$ which is tantamount to shifting P by one position to the right. Without the latter provision, P would be shifted backwards to align the two characters.

In the worst case the algorithm executes in $O(|T||P|)$ time; for example, if $P = ba^{m-1}$ and $T = a^n$. Note that in this case, the algorithm rechecks characters in T that have already been checked.

The algorithm can be improved if we take into account the entire substring that follows a mismatched character P_j. Consider the following shift

```
  aaabcabcbabbaecabcab
1 abdabcabcab
2  abdabcabcab
```

which shifts P by one position in accordance with the left side occurrence rule. But a longer shift can result from aligning the substring of T equal to the already matched suffix that *directly* follows the mismatched character $P_8 = b$ with an equal substring in P that begins to the left of P_8.

```
  aaabcabcbabbaecabcab
1 abdabcabcab
2    abdabcabcab
```

However, note that after the shift, the mismatched character b in T is again aligned with $c = P_5$, which just caused a mismatch. Therefore, if matching ever reaches c after restarting from the end of P, the mismatch is guaranteed to reoccur. To avoid this mismatch, it is better to align the suffix ab of P that directly follows $P_8 = c$ with an equal substring of P that is preceded by a different character than c. In our example, substring ab in P that follows the mismatched character P_8 should be aligned with ab preceded by d because it is different from c:

```
  aaabcabcbabbaecabcab
1 abdabcabcab
2       abdabcabcab
```

after which the matching process restarts from the end of P.

What should be the shift if no substring begins to the left of a mismatched character P_j and is equal to the suffix that directly follows P_j? For example, what should be the shift after a mismatch is found in line 2? In this case, we align the longest suffix of P that follows the mismatched character P_j with an equal prefix of P:

```
  aaabcabcbabbaecabcab...
2       abdabcabcab
3             abdabcabcab
```

To sum up, there are two cases to consider:

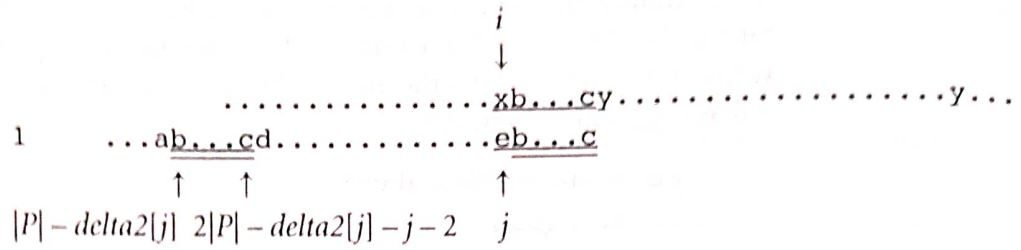
1. *A full suffix rule.* If a mismatched character P_j is directly followed by a suffix that is equal to a substring of P that begins anywhere to the left of P_j, align the suffix with the substring.
2. *A partial suffix rule.* If there is a prefix of P equal to the longest suffix anywhere to the right of the mismatched character P_j, align the suffix with the prefix.

To accomplish these shifts, a table *delta2* is created that for each position in P holds a number by which the index i that scans T has to be incremented to restart the matching process; that is, if a mismatched character is P_j, then i is incremented by *delta2*[j] (and j is set to $|P| - 1$). Formally, *delta2* is defined as follows:

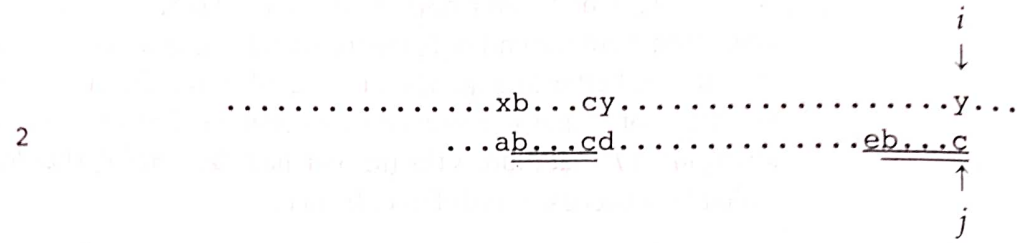
$$delta2[j] = \min\{s + |P| - j - 1 : 1 \le s \text{ and } (j \le s \text{ or } P_{j-s} \ne P_j) \text{ and for } j < k < |P| : (k \le s \text{ or } P_{k-s} = P_k)\}$$

and $delta2[|P| - 1] = delta2[|P| - 2]$ if the last two characters in P are the same (if they are different, then $delta2[|P| - 1] = 1$, because the third subcondition in the definition, "for . . . ", is vacuously true).

There are, as already indicated, two cases. In the first case, the suffix directly following the mismatched character P_j has a matching substring in P, so the situation after detecting a mismatch

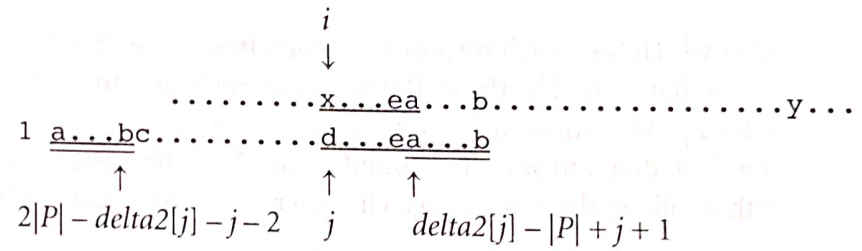


when suffix $P(j+1 \ldots |P|-1)$ equals to the substring $P(|P| - delta2[j] \ldots 2|P| - delta2[j] - j - 2)$ changes to:

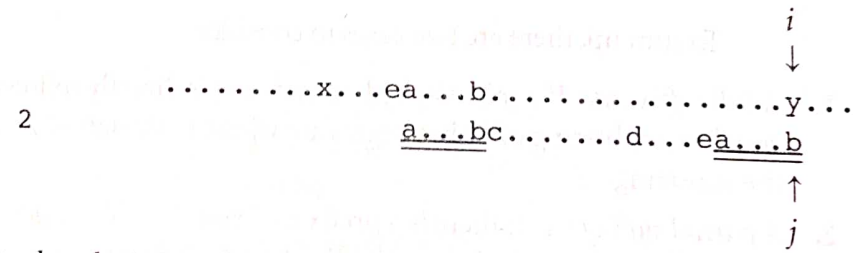


before resuming the matching process.

In the second case, the situation



when suffix $P(delta2[j] - |P| + j + 1 \ldots |P| - 1)$ equals the prefix $P(0 \ldots 2|P| - delta2[j] - j - 2)$ changes to:



Note that the inequalities in the *or* clauses in the definition of *delta2* are indispensable for the second case.

To compute *delta2*, a brute force algorithm can be used, as follows:

```
computeDelta2ByBruteForce(pattern P, table delta2)
    for k = 0 to |P|-1
        delta2[k] = 2*|P|-k-1;
    // partial suffix phase:
    for k = 0 to |P|-2                          // k is a mismatch position;
        for (i = 0, s = j = k+1; j < |P|; s++, j = s, i = 0)
            while j < |P| and P_i == P_j
                i++;
                j++;
            if j == |P|                         // a suffix to the right of k is detected
```

```
            delta2[k] = |P|-(k+1) + |P|-i;      // that is equal to a prefix of P,
            break;                              // P(0... i-1) equals P(|P|-i... |P|-1);
// full suffix phase:
for k = |P|-2 downto 0                          // k is a mismatch position;
    for (i = |P|-1, s = j = |P|-2; j ≥ |P|-k-2; s--, j = s, i = |P|-1)
        if P_i == P_j
            while i > k and P_i == P_j
                i--;
                j--;
            if j == -1 or i == k and P_i ≠ P_j   // a substring in P is detected
                delta2[k] = |P|-j-1;        // that is equal to the suffix directly following k,
                break;                      // P(j+1... j+|P|-k-1) equals P(k+1... |P|-1);
if P_|P|-1 == P_|P|-2
    delta2[|P|-1] = delta2[|P|-2];
else delta2[|P|-1] = 1;
```

The algorithm has three phases: initialization, partial suffix phase, and full suffix phase. Initialization prepares the pattern for the longest shift; after a mismatch, the pattern is shifted all the way past the mismatched character. The only exception is the mismatch at the last character in P, after which P is shifted by only one position. The partial suffix phase looks for the longest suffixes after a mismatch point by matching prefixes. The full suffix phase updates those values in *delta2* that correspond to a mismatch being followed by a suffix that has a matching substring in P. For P = *abdabcabcab*, the values in *delta2* after each phase are as follows:

```
                a  b  d  a  b  c  a  b  c  a  b
delta2 = 21 20 19 18 17 16 15 14 13 12  *   after initialization
delta2 = 19 18 17 16 15 14 13 12 11 12  *   after partial suffix phase
delta2 = 19 18 17 16 15  8 13 12  8 12  1   after full suffix phase
```

The algorithm can be applied to short patterns only, because it is quadratic in the best case and cubic in the worst case. For $P = a^m$, the full suffix phase executes in total,

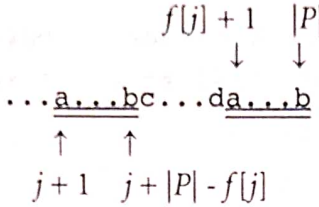
$$\sum_{k=0}^{m-2} \sum_{j=m-k-2}^{m-2} (m-k) = \frac{(m-1)m(m+4)}{6}$$

comparisons, because in one iteration of the inner `for` loop, $m-k-1$ comparisons are performed in the `while` loop and one in the `if` statement. $P = a^{m-1}b$ is an example of the worst case for the partial suffix phase. Clearly, for longer patterns a faster algorithm is needed.

The algorithm can be significantly improved by using an auxiliary table f that is a counterpart of *next* for the reverse of P. The table f is defined as follows:

$$f[j] = \begin{cases} |P| & \text{if } j = |P| - 1 \\ \min\{k : j < k < |P| - 1 \text{ and } P(j+1 \ldots j+|P|-k) = P(k+1 \ldots |P|-1)\} & \text{if } 0 \le j < |P| - 1 \end{cases}$$

That is, $f[j]$ is the position preceding the starting position of the longest suffix of P of length $|P|-f[j]$ that is equal to the substring of P that begins at position $j+1$:



For example, for $P = aaabaaaba$, $f[0] = 4$, because substring $P(1 \ldots 4) = aaba$ is the same as the suffix $P(5 \ldots 8)$; $f[1] = 5$, because substring $P(2 \ldots 4)$ is equal to the suffix $P(6 \ldots 8)$; that is, the underlined substrings in $P = aa\underline{aba}a\underline{aba}$ are equal. The entire table $f = [4\ 5\ 6\ 7\ 7\ 7\ 8\ 8\ 9]$. Note that a substring of P equal to a suffix of P can overlap, as in $P = baaabaaaba$.

The table f allows us to go from a certain substring of P to a matching suffix of P. But the matching process during execution of the Boyer-Moore algorithm proceeds from right to left, so that after a mismatch is found, a suffix is known and we need to know a matching substring of P to align the substring with the suffix. In other words, we need to go from the suffix to the matching substring, which is the opposite direction with respect to information provided by f. That is why *delta2* is created to have direct access to the needed information. This can be accomplished with the following algorithm, which is obtained from `computeDelta2ByBruteForce()`.

```
computeDelta2UsingNext(pattern P, table delta2)
    findNext2(reverse(P),next);
    for i = 0 to |P|-1
        f[i] = |P| - next[|P|-i-1] - 1;
        delta2[i] = 2*|P| - i - 1;
    // full suffix phase:
    for i = 0 to |P|-2
        j = f[i];
        while j < |P|-1 and P_i ≠ P_j
            delta2[j] = |P| - i - 1;
            j = f[j];
    // partial suffix phase:
    for (i = 0; i < |P|-1 and P_0 == P_f[i]; i = f[i])
        for j = i to f[i]-1
            if delta2[j] == 2*|P| - j - 1 // if not updated during full suffix phase,
                delta2[j] = delta2[j] - (|P| - f[i]); // update it now;
    if P_|P|-1 == P_|P|-2
        delta2[|P|-1] = delta2[|P|-2];
    else delta2[|P|-1] = 1;
```

First, table *next* is created and used to initialize table f. Also, table *delta2* is initialized to values that indicate shifting P past the mismatched character in T. For $P =$ *dabcabeeeabcab* it is

```
P      = a  b  c  a  b  d  a  b  c  a  b  e  e  e  a  b  c  a  b
f      = 14 15 16 17 18 13 14 15 16 17 18 18 18 16 17 18 18 18 19
delta2 = 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19
```

As in the brute force algorithm, the full suffix phase addresses the first case when a mismatched character P_j is directly followed by the suffix that is equal to a substring of P that begins anywhere to the left of P_j. In that phase, positions directly accessible from f are processed. For example, in the sixth iteration of the `for` loop, $i = 5$ and $f[5] = 13$, which means that there is a suffix beginning at position 14, *abcab*, for which there is a substring beginning at position $|P| - f[5] = 6$, which is equal to the suffix. Because also $P_5 \neq P_{13}$, *delta2*[13] can be assigned a proper value:

```
                0                 5                          13                 18
P       =  a  b  c  a  b  d  a  b  c  a  b  e  e  e  a  b  c  a  b
f       = 14 15 16 17 18 13 14 15 16 17 18 18 18 16 17 18 18 18 19
delta2 = 37 36 35 34 33 32 31 30 29 28 27 26 25 13 23 22 21 20 19
```

But the substring $P(6 \ldots 10) = abcab$ has a prefix *ab* that is the same as a suffix of the suffix *abcab*, and both the prefix *ab* and suffix *ab* are preceded by different characters. The position right before this shorter suffix is directly accessible only from the position $f[13] = f[f[5]] = 16$ because i is still 5. Therefore, after the second iteration of the `while` loop—still during the sixth iteration of the `for` loop—the situation changes to

```
                0                 5                          13          16    18
P       =  a  b  c  a  b  d  a  b  c  a  b  e  e  e  a  b  c  a  b
f       = 14 15 16 17 18 13 14 15 16 17 18 18 18 16 17 18 18 18 19
delta2 = 37 36 35 34 33 32 31 30 29 28 27 26 25 13 23 22 13 20 19
```

Number 13 is put in cell *delta2*[16], which means that when the matching process stops at P_{16}, index i that scans T is incremented by 13; that is, when scanning stops after detecting a mismatch with P_{16}, the situation is as in

```
                  i
                  ↓
...abcabdabcabeeeeabcab.......x...
abcabdabcabeeeeabcab
```

so that scanning is resumed after updating i as in

```
                                  i
                                  ↓
...abcabdabcabeeeeabcab.......x...
                abcabdabcabeeeeabcab
```

Note that the increment is the same as for the mismatch with P_{14}. This is because in both cases character $T_{i+1} = a$ (before updating i) is aligned with P_6, because both suffixes *ab* and *abcab* have a match that starts at P_6. However, as shown in the previous diagram, a match would be missed if i were incremented by 13. But the algorithm continues and for $i = 13$ and $j = f[13] = 16$ the `while` loop is entered to modify *delta2*:

```
                0                                          13          16    18
P       =  a  b  c  a  b  d  a  b  c  a  b  e  e  e  a  b  c  a  b
f       = 14 15 16 17 18 13 14 15 16 17 18 18 18 16 17 18 18 18 19
delta2 = 37 36 35 34 33 32 31 30 29 28 27 26 25 13 23 22  5 20 19
```

which prevents i from missing a match.

After finishing the first outer `for` loop, the second outer `for` loop is executed to decrease, if possible, other values in *delta2*. For $i = 0$ the inner `for` loop is entered and executed for j from $i = 0$ to $f[0] - 1 = 13$, so that *delta2* becomes

```
          0  1  2  3  4  5  6  7  8  9 10 11 12 13                18
P      =  a  b  c  a  b  d  a  b  c  a  b  e  e  e  a  b  c  a  b
f      = 14 15 16 17 18 13 14 15 16 17 18 18 18 16 17 18 18 18 19
delta2 = 32 31 30 29 28 27 26 25 24 23 22 21 20 13 23 22  5 20 19
```

In this way, the first 13 values in *delta2* are decremented by 5. This corresponds to the situation when a mismatch occurs for any of the first 13 characters in P. When this happens, the suffix *abcab* is aligned with the prefix *abcab* because the suffix is to the right of any of these positions.

In the second iteration of the outer loop, when $i = 14$ and $f[14] - 1 = 16$, the inner loop updates *delta2*[14] and *delta2*[15] by decrementing them by 2 because this is the length of suffix *ab* to the right of P_{14} and P_{15} that have a matching prefix:

```
          0                                          14 15 16    18
P      =  a  b  c  a  b  d  a  b  c  a  b  e  e  e  a  b  c  a  b
f      = 14 15 16 17 18 13 14 15 16 17 18 18 18 16 17 18 18 18 19
delta2 = 32 31 30 29 28 27 26 25 24 23 22 21 20 13 21 20  5 20 19
```

After exiting the loops, the last value is changed so that finally the situation is

```
          0                                                   18
P      =  a  b  c  a  b  d  a  b  c  a  b  e  e  e  a  b  c  a  b
f      = 14 15 16 17 18 13 14 15 16 17 18 18 18 16 17 18 18 18 19
delta2 = 32 31 30 29 28 27 26 25 24 23 22 21 20 13 21 20  5 20  1
```

To find the complexity of the algorithm we claim that the `while` loop is executed at most $|P| - 1$ times in total. This is because after entering the `while` loop for an i and executing k iterations, the `while` loop is not entered for the next $|P| - f[i] - 1$ iterations of the outer `for` loop, where $|P| - f[i] - 1$ is the length of a matching suffix that begins at $P_{f[i]+1}$ that matches the substring that begins at P_1 and for which $k \leq |P| - f[i] - 1$. It is because for each character P_r in the substring $P(i + 2 \ldots i + |P| - f[i])$, P_r and the corresponding characters $P_{r+f[i]}$ are preceded by the same character so that the condition in the `while` loop is false. Consider $P = badaacadaa$ for which $f = [5\ 6\ 7\ 8\ 9\ 8\ 9\ 8\ 9\ 10]$. Iterations of the `while` loop can be activated either for P_0 or for an s for which $f[s-1] > f[s]$, (e.g., for $s = 5$, $f[4] = 9$ and $f[5] = 8$). Increasing numbers in f indicate substrings that are extensions of following substrings and corresponding suffixes of P. For example, for $f[1] = 6$ and $f[2] = 7$, numbers 1 and 6 indicate that the substring $P(2 \ldots 4)$ equals suffix $P(7 \ldots 9)$ and numbers 2 and 7 indicate that the substring $P(3 \ldots 4)$ equals suffix $P(8 \ldots 9)$; that is, $P(2 \ldots 4)$ is an extension of $P(3 \ldots 4)$ and so is suffix $P(7 \ldots 9)$ a forward extension of suffix $P(8 \ldots 9)$. This means that $P_2 = P_7$, and for $j = 2$ the `while` loop cannot be entered.

In the worst case, the first half of P can lead to $|P|/2$ iterations of the `while` loop followed by $|P|/2$ iterations of the outer `for` loop without entering the `while` loop. Next, the first half of the second half of P can lead to $|P|/4$ iterations of the `while` loop followed by the same number of iterations of the outer `for` loop without entering the `while` loop, and so on, which gives

$$\sum_{k=1}^{\lfloor |P| \rfloor} \frac{|P|}{2^k} = |P| - 1$$

iterations of the `while` loop in total. Because the outer `for` loop can iterate $|P| - 1$ times, this gives $2(|P| - 1)$ as the maximum number of values assigned to j, and hence the complexity of the outer `for` loop.

The last nested `for` loop is executed at most $|P| - 1$ times: For each i, it is executed for j from i to $f[i] - 1$ and then i is updated to $f[i]$; therefore, j refers to one position at most once. We can conclude that the algorithm is linear in the length of P.

To use *delta2*, algorithm `BoyerMooreSimple()` is modified by replacing the line that updates i

```
else i = i + max(delta1[T_i],|P|-j);
```

with the line

```
else i = i + max(delta1[T_i],delta2[j]);
```

In an involved proof, Knuth shows that the Boyer-Moore algorithm that utilizes tables *delta1* and *delta2* performs at most $7|T|$ comparisons if the text does not contain any occurrence of the pattern (Knuth, Morris, and Pratt, 1977). Guibas and Odlyzko (1980) improved the bound to $4|T|$ and Cole (1994) improved it to $3|T|$.

The Sunday Algorithms

Daniel Sunday (1990) begins his analyses with an observation that in the case of a mismatch with a text character T_i, the pattern shifts to the right by at least one position so that the character $T_{i+|P|}$ is included in the next iteration. The Boyer-Moore algorithm shifts the pattern according to the value in the *delta1* table (we leave the table *delta2* aside, for now) and this table includes shifts with respect to the mismatched character T_i. It would be more advantageous, Sunday submits, to build *delta1* with respect to character $T_{i+|P|}$. In this way, *delta1*[*ch*] is the position of character *ch* in P counted from the left. This is closely related to Boyer-Moore's *delta1* because by incrementing by one the values in the latter, we obtain Sunday's *delta1*.

One advantage of this solution is that the set of three rules used in the Boyer-Moore algorithm can be simplified. The no occurrence rule is slightly modified: If the character $T_{i+|P|}$ appears nowhere in P, align P_0 with $T_{i+|P|+1}$. The right side occurrence rule is no longer needed, because all characters in P are to the left of $T_{i+|P|}$. Finally, the left side occurrence rule can be simplified to the occurrence rule: If there is an occurrence in P of character *ch* equal to $T_{i+|P|}$, align $T_{i+|P|}$ with the closest (the rightmost) *ch* in P.

Although the definition of *delta1* depends on right-to-left scan of pattern, the matching process can be performed in any order, not only left to right or right to left. Sunday's `quickSearch()` performs the scan left to right. Here is its pseudocode:

```
quickSearch(pattern P, text T)
    initialize all cells of deltal to |P| + 1;
    for i = 0 to |P|-1
        deltal[P_i] = |P| + 1 - i;
```

```
i = 0
while i ≤ |T|-|P|
    j = 0;
    while j < |P| and i < |T| and P_j == T_i
        i++;
        j++;
    if j > |P|
        return success at i-|P|;
    i = i + delta1[T_{i+|P|}];
return failure;
```

For example, for P = *cababa*, *delta1*['a'] = 1, *delta1*['b'] = 2, *delta1*['c'] = 6, and for remaining characters *ch*, *delta1*[*ch*] = 7. Here is an example:

```
   ffffaabcfacababafa
1  cababa
2    cababa
3            cababa
4              cababa
```

Line 1 has a mismatch right at the beginning, so that in line 2, character $T_{i+|P|} = T_{0+6} = b$ is aligned with the rightmost b in P and $i = 0$ is incremented by $delta1[T_{i+|P|}] = delta1['b'] = 2$, so that $i = 2$. Again there is a mismatch at the beginning of P; $i = 2$ is incremented by $delta1[T_{i+|P|}] = delta1['f'] = 6$, so that $i = 9$; that is, in effect, P is shifted past letter $T_{i+|P|} = f$. The fourth iteration is successful. Compare this trace with the trace for `BoyerMooreSimple()`, for which *delta1*['a'] = 0, *delta1*['b'] = 1, *delta1*['c'] = 5, and for remaining characters *ch*, *delta1*[*ch*] = 6:

```
   ffffaabcfacababafa
1  cababa
2   cababa
3    cababa
4          cababa
5           cababa
6             cababa
```

Sunday introduces two more algorithms, both based on a generalized *delta2* table. Sunday's *delta2* may be the same as Knuth-Morris-Pratt's *next* table if the *delta2* table is initialized by scanning P left to right. If it is scanned in the reverse order, then *delta2* is the same as Boyer-Moore's *delta2*. However, the matching process can be done in any order. Sunday's second algorithm, the maximal shift algorithm, uses *delta2* such that *delta2*[0] is associated with a character in P whose next leftward occurrence in P is maximum; *delta*[1] refers to a character in P for which the next leftward occurrence in P is not less than *delta2*[0], and so on. In the third algorithm, the optimal mismatch algorithm, characters are ordered in ascending order of frequency of occurrence. This is motivated by the fact that in English, 20 percent of words end with the letter *e* and 10 percent of the letters used in English are also *e*. Thus, it is very likely to match first characters tested by using Boyer-Moore's backward scanning. The testing of the least probable characters first improves the likelihood of early mis-

match. However, Sunday's own tests show that although his three algorithms fare much better on searching for short English words than the Boyer-Moore algorithm, there is little difference between the three algorithms, and for all practical purposes `quickSearch()` is sufficient. This is particularly true when the overhead to find *delta2* is taken into account (see Pirklbauer, 1992). To address the problem of frequency of character occurrence, an adaptive technique can be used, as in Smith (1991).

Sunday points out that his *delta1* table usually allows for shifts one position greater than shifts based on Boyer-Moore's *delta1*. However, after pointing out that this is not always the case, Smith (1991) indicates that the larger of the two values should be used.

13.1.4 Multiple Searches

The algorithms presented in the preceding sections are designed to find an occurrence of a pattern in a text. Even if there are many occurrences, the algorithms are discontinued after finding the first. Many times, however, we are interested in finding all occurrences in the text. One way to accomplish this is to continue the search after an occurrence is detected, after shifting the pattern by one position. For example, the Boyer-Moore algorithm can be quickly modified to accommodate multiple searches in the following fashion:

```
BoyerMooreAllOccurrences(pattern P, text T)
    initialize all cells of delta1 to |P|;
    for i = 0 to |P| - 1
        delta1[P_i] = |P| - i - 1;
    compute delta2;
    i = |P| - 1;
    while i < |T|
        j = |P| - 1;
        while j ≥ 0 and P_j == T_i
            i--;
            j--;
        if j == -1
            output: match at i+1;
            i = i + |P| + 1; // shift P by one position to the right;
        else i = i + max(delta1[T_i],delta2[j]);
```

But consider the process of finding all occurrences of P = *abababa* in T = *abababababa* . . . :

```
  abababababa...
1 abababa
2  abababa
3   abababa
4    abababa
```

In every second iteration, the entire pattern is compared to the text only after shifting by two positions. For this reason, the algorithm requires $|P|(|T| - |P| + 1)/2$, or more

generally, $O(|T||P|)$ steps. To reduce the number of comparisons it should be recognized that the pattern includes consecutive repetitive substrings, called *periods*, which should not be reexamined after they were matched with substrings in the text with which they are about to be matched.

The Boyer-Moore-Galil algorithm works the same as the Boyer-Moore algorithm until the first occurrence of the pattern is detected (Galil, 1979). After that, the pattern is shifted by p = |the period of the pattern|, and only the last p characters of the pattern need to be compared to the corresponding characters in the text to know whether the entire pattern matches a substring in the text. In this way, the part overlapping a previous occurrence does not have to be rechecked. For example, for P = *abababa* with the period *ab*, the new algorithm is executed as follows:

```
  abababababa...
1 abababa
2   abababa
3     abababa
```

However, if a mismatch is found, then the Boyer-Moore-Galil algorithm resumes its executions in the same way as the Boyer-Moore algorithm. The algorithm is as follows:

```
BoyerMooreGalil(pattern P, text T)
    p = period(P);
    compute delta1 and delta2;
    skip = -1;
    i = |P|-1;
    while i < |T|
        j = |P|-1;
        while j > skip and P_j == T_i
            i--;
            j--;
        if j == skip
            output: a match at i-skip;
            if p == 0
                i = i + |P|+1;
            else if skip == -1
                    i = i + |P|+p;
                else i = i + 2*p;
                skip = |P|-p-1;
        else skip = -1;
            i = i + max(delta1[T_i],delta2[j]);
```

It is clear that the algorithm achieves better performance only for patterns with periods and only if the text contains a high number of overlapping occurrences of the pattern. For patterns with no periods, the two algorithms work the same way. For a pattern with periods but with no overlapping occurrence, the Boyer-Moore-Galil algorithm performs better shifts than the Boyer-Moore algorithm, but only when an occurrence is found.