These notes are adapted from [2], and are a better explanation of the Knuth-Morris-Pratt algorithm than found in CLRS3, section 32.4, pages 1002–1011.

The beautiful Knuth-Morris-Pratt string matching algorithm finds the leftmost occurrence of a character string \textit{pattern} in a character string \textit{text} in linear time, using character-to-character comparisons. However, the original presentation of the algorithm, and presentations based on it are more complicated and less clear than they could be, partly because of the intricate loops of the algorithm. The best presentation is CLRS3, chapter 32; it is highly polished, but still quite complex, especially the proof of correctness. The alternative version presented here, a recursive, memoized (CLRS3, chapter 15) version, is more readily understood and analyzed.

1 Notation

We write the algorithm in Java; in addition we use the following notation. A string variable \textit{s} contains \#s characters indexed 0 to \#s − 1. In other words, \textit{s} = \textit{s}[0], \textit{s}[1], \ldots, \textit{s}[\#s − 1]. A substring of \textit{s} consisting of \textit{s}[i], \ldots, \textit{s}[j] is denoted by \textit{s}[i..j]; \textit{s}[i..j] is the empty string if \( i = j + 1 \). The concatenation of string \textit{s} and character \textit{c} is written \textit{s} + \textit{c}. String \textit{s}[0..j], for \(-1 \leq j < \#s\), is a prefix of \textit{s}; if \( j < \#s − 1 \) it is a proper prefix. \textit{s}[0..−1] is the empty prefix. String \textit{s}[j..\#s − 1], \( 0 \leq j \leq \#s \), is a suffix of \textit{s}.

2 The Basic Ideas

The K-M-P string matching algorithm processes \textit{text} one character at a time, from beginning to end. When ready to process character \textit{text}[\textit{n}], it knows a value \textit{m} such that the suffix \textit{text}[n−m..\#text−1] contains the next possible occurrence of \textit{pattern}, and its first \textit{m} characters equal \textit{pattern}[0..\textit{m} − 1]:

\[
\begin{array}{ccc}
  & 0 & n - m \\
\text{text:} & \text{pattern does not begin here} & \text{} = \text{pattern}[0..m - 1] \\
\end{array}
\]

We describe this state of affairs by the assertion

\[
P : \quad 0 \leq m \leq \#\textit{pattern} \\
    \wedge \ 0 \leq n \leq \#\textit{text} \\
    \wedge \ \text{notbegin}(0, n − m − 1) \\
    \wedge \ \textit{pattern}[0..m - 1] = \textit{text}[n − m..n − 1],
\]
where \textit{notbegin}(t, u) is an abbreviation for “pattern does not occur in text beginning in text\text{[}t..u\text{]}”. Assertion \(P\) holds trivially for \(n = m = 0\).

The idea of a \textit{longest proper prefix that is a suffix} lies at the heart of K-M-P string matching, and the algorithm uses an array \(\text{Prefix}\) to record (“memoize”) the results of the function \(\text{prefix}\) that computes these prefixes. The values in \(\text{Prefix}\) are defined by the assertion

\[
Q: \quad \text{For all } j, 0 \leq j < \#\text{pattern}, \text{ either } \text{Prefix}[j] = -1 \text{ or } \text{Prefix}[j] \text{ is the length of the longest proper prefix of } \text{pattern}[0..j-1] \text{ that is a suffix of } \text{pattern}[0..j-1].
\]

\(\text{Prefix}[0]\) always equals \(-1\), since the empty string has no proper prefix. For example, from the K-M-P original paper [1] we have

\[
\begin{array}{cccccccccc}
   i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
   \text{pattern}[i] & a & b & c & a & b & c & a & c & a & b \\
   \text{Prefix}[i] & -1 & 0 & 0 & 0 & 1 & 2 & 3 & 4 & 0 & 1 \\
\end{array}
\]

### 3 The Algorithm

The algorithm, written in Java, consists of a driver function \(\text{KMPmatch}\),

```java
public int KMPmatch(String pattern, String text) {
    Prefix = new int[pattern.length()];
    for (int i = 0; i < pattern.length(); i++)
        Prefix[i] = -1;
    // Q holds and P holds for n = m = 0
    return match(pattern, 0, text, 0);
}
```

and a pair of mutually recursive functions \(\text{match}\) and \(\text{extend}\):

```java
private int match(String pattern, int m, String text, int n) {
    if (m == pattern.length()) // End of pattern ...
        return n-m; // ... a match
    if (n == text.length()) // End of text ...
        return -1; // ... no match
    // S holds
    return match(pattern, extend(pattern, m, text.charAt(n)),
                 text, n+1);
}
```

```java
private int extend(String pattern, int m, char n) {
    // Specification: Given that P and Q hold for parameters n and
    // m, return position in text of first occurrence of pattern
    // (or -1 if pattern does not occur in text).
    if (m == pattern.length()) // End of pattern ...
        return n-m;
    if (n == text.length()) // End of text ...
        return -1;
    // S holds
    return match(pattern, extend(pattern, m, text.charAt(n)),
                 text, n+1);
}
```
// Specification: Given Q and 0 <= j < #pattern, return
// length of longest prefix of pattern that is a suffix of
// pattern[0..j-1]+c.
private int extend(String pattern, int j, char c) {
    if (pattern.charAt(j) == c)
        return j+1;
    if (j == 0)
        return 0;
    return extend(pattern, prefix(pattern, j), c);
}

// Specification: Given Q and 0 < i < #pattern, return length
// of longest proper prefix of pattern[0..i-1] that is a suffix
// of pattern[0..i-1]. Also, store computed values in array
// Prefix, in order to maintain Q.
private int prefix(String pattern, int i) {
    if (Prefix[i] == -1)
        if (i == 1)
            Prefix[i] = 0;
        else
            Prefix[i] = extend(pattern, prefix(pattern, i-1),
            pattern.charAt(i-1));
    return Prefix[i];
}

The function prefix corresponds exactly to the function \( f \) in the original K-M-P article [1] and to the function \( \pi \) in CLRS3 at the bottom of page 1004; that is, prefix\((k)\) is \( \sigma(P_k) \) on page 996, with the additional requirement that the prefix be proper. The function extend is the computation of \( \delta(q,a) \) in equation (32.4) on page 998 of CLRS3 (which is computed by the procedure Compute-Transition-Function on page 1001).

For a given integer \( j \), the call prefix(pattern, \( j \)) can be evaluated many times; therefore, we memoize the value by storing it the first time prefix(pattern, \( j \)) is evaluated and use the stored value for subsequent calls. Writing prefix without memoization, that is, as

// Specification: Given 0 < i < #pattern, return length of longest
// proper prefix of pattern[0..i-1] that is a suffix of pattern[0..i-1].
private int prefix(String pattern, int i) {
    if (i == 1)
        return 0;
    else
        return extend(pattern, prefix(pattern, i-1),
        pattern.charAt(i-1));
}

does the string matching properly, but not generally in linear time.

4 Termination

The function KMPmatch clearly terminates provided match does. The recursion in match terminates because the sequence of fourth arguments of recursive calls forms a strictly increasing sequence that is bounded above by \#text; thus match terminates if extend and prefix do.
Because, by induction,

\[\text{prefix}(\text{pattern}, k) < k,\] (1)

and

\[\text{extend}(\text{pattern}, k, c) \leq k + 1,\] (2)

the mutual recursive calls to \text{prefix} and \text{extend} terminate by an inductive argument as follows:

Clearly, \text{prefix} terminates for \(i = 1\) and \text{extend} terminates for \(j = 0\). The termination of \text{prefix} for \(i < n\) implies the termination of \text{extend} for \(j < n\) which, in turn, implies the termination of \text{prefix} for \(i = n\). In CLRS3, (1) follows trivially from the definition of \(\pi\) at the bottom of page 1004; (2) is Lemma 32.2 on page 999.

5 Correctness

The function \text{KMPmatch} initializes array \text{Prefix} to insure that assertion \(Q\) holds, and then invokes \text{match}, which returns the position of the first occurrence of \text{pattern} in \text{text} (or \(-1\) if \text{pattern} does not occur in \text{text}). The correctness of \text{KMPmatch} follows directly from the specification of \text{match}.

We now consider the correctness of \text{match}. Analyses of the trivial cases \(m = \#\text{pattern}\) (line 6) and \(n = \#\text{text}\) (line 8) are left to the reader. Consider the case at line 10, where \(m < \#\text{pattern}\) and \(n < \#\text{text}\). Based on the specification of \text{match}, we see that the recursive call to \text{match} returns the correct value if \(P\) and \(Q\) hold just before the call—with the arguments of the call replacing the parameters. That is, just before the call, assertions \(P'\) and \(Q\) must hold, where \(P'\) is:

\[
P': \quad 0 \leq e \leq \#\text{pattern} \\
\land \quad 0 \leq n + 1 \leq \#\text{text} \\
\land \quad \text{notbegin}(0, n - e) \\
\land \quad \text{pattern}[0..e - 1] = \text{text}[n + 1 - e..n],
\]

where \(e = \text{extend}(\text{pattern}, m, \text{text}[n])\). We assume the precondition

\[
S: \quad m < \#\text{pattern} \\
\land \quad n < \#\text{text} \\
\land \quad P
\]

(which holds at line 10 of \text{match}) and prove \(P'\), thus proving that \text{match} works correctly in this case.

The first clause of \(P'\), \(0 \leq e \leq \#\text{pattern}\), follows from the specification of \(e = \text{extend}(\text{pattern}, m, \text{text}[n])\).

The second clause, \(0 \leq n + 1 \leq \#\text{text}\), follows from \(P\) and \(n < \#\text{text}\). Finally, we show that the third and fourth clauses are implied by \(S\) and \(e = \text{extend}(\text{pattern}, m, \text{text}[n])\). By the specification of \text{extend},

\[e = \text{length of longest prefix of } \text{pattern} \text{ that is a suffix of } \text{pattern}[0..m - 1] + \text{text}[n],\]
and by $S$, $P$ holds, so that $\text{pattern}[0..m-1] = \text{text}[n-m..n-1]$; hence
\[
e = \text{length of longest prefix of pattern that is a suffix of } \text{text}[n-m..n].
\]
By $S$ and the definitions of prefix and suffix this means
\[
\text{pattern}[0..e-1] = \text{text}[n+1-e..n]
\]
\[
\land \not\text{begin}(n-m,n-e)
\]
and in $P$ we have $\not\text{begin}(0,n-m-1)$ so that
\[
\text{pattern}[0..e-1] = \text{text}[n+1-e..n]
\]
\[
\land \not\text{begin}(0,n-e),
\]
which are the fourth and third clauses of $P'$, respectively. Note that in the argument above the replacement of $\text{text}[n-m..n]$ by $\text{pattern}[0..m-1] + \text{text}[n]$ allows $\text{extend}$ to deal just with $\text{pattern}$ and not with $\text{text}$.

We argue that $\text{extend}$ satisfies its specification. Clearly, $\text{extend}$ satisfies its specification in the cases $\text{pattern}[j] = c$ (line 6) and $\text{pattern}[j] \neq c$, $j = 0$ (line 8). Consider the case at line 9, where $\text{pattern}[j] \neq c$ and $j > 0$; here, the returned value $e$ satisfies $e < j + 1$, so
\[
e = \text{length of longest proper prefix of } \text{pattern}[0..j] \text{ that is a suffix of } \text{pattern}[0..j-1] + c.
\]
Let
\[
j' = \text{length of longest proper prefix of } \text{pattern}[0..j-1] \text{ that is a suffix of } \text{pattern}[0..j-1].
\]
Then, since $\text{pattern}[0..j'-1] = \text{pattern}[j-j'..j-1],
\[
e = \text{length of the longest prefix of } \text{pattern}[0..j'] \text{ that is a suffix of } \text{pattern}[0..j'-1] + c.
\]
Thus, $e = \text{extend}(\text{pattern}, j', c)$. The specification of $\text{prefix}$ tells us that $j' = \text{prefix}(\text{pattern}, j)$, so $e = \text{extend}(\text{pattern}, \text{prefix}(\text{pattern}, j), c)$. Hence, $\text{extend}$ satisfies its specification in this case too.

Finally, for the correctness of $\text{prefix}$, we leave to the reader the verification of the simple cases $\text{Prefix}[i] \neq -1$ (line 6) and $i = 1$ (line 8) and consider only the case at lines 10–11, where $\text{Prefix}[i] = -1$, $i > 1$; here, $\text{Prefix}[i]$ is calculated and returned. Using $h$ for the value to be stored in $\text{Prefix}[i]$, we have:
\[
h = \text{length of longest proper prefix of } \text{pattern}[0..i-1] \text{ that is a suffix of } \text{pattern}[0..i-1].
\]
Let
\[
t = \text{length of the longest proper prefix of } \text{pattern}[0..i-2] \text{ that is a suffix of } \text{pattern}[0..i-2].
Then, since \( \text{pattern}[0..t-1] \) is a suffix of \( \text{pattern}[0..i-2] \),
\[
h = \text{length of the longest prefix of } \text{pattern} \text{ that is a suffix of } \text{pattern}[0..t-1] + \text{pattern}[i-1].
\]
But then \( h = \text{extend}(\text{pattern}, t, \text{pattern}[i-1]) \). The specification of \( \text{prefix} \) tells us that \( t = \text{prefix}(\text{pattern}, i-1) \), so \( h = \text{extend}(\text{pattern}, \text{prefix}(\text{pattern}, i-1), \text{pattern}[i-1]) \). Hence, \( \text{prefix} \) satisfies its specification in this case.

6 Time Required

The initialization in lines 4–6 of \( \text{KMPmatch} \) is \( O(\#\text{pattern}) \). To analyze the time for the call of \( \text{match} \) in line 8 of \( \text{KMPmatch} \), recall from above that in the tail recursion in lines 10–11 of \( \text{match} \), the value of \( n \) increases every time—this can happen at most \( \#\text{text} \) times. So the worst case running time will be \( O(\#\text{text}) \), plus the time spent in \( \text{extend} \).

Before we consider the time spent in \( \text{extend} \) during calls from lines 10–11 of \( \text{match} \), we determine how much time is spent in \( \text{extend} \) during calls from lines 10–11 in \( \text{prefix} \). Because \( \text{prefix} \) is memoized, we can consider a call to it to take \( O(1) \) time, if we add separately the time spent in \( \text{extend} \) for all \( \#\text{pattern} \) possible values of \( i \) in the call from lines 10–11 of \( \text{prefix} \). Each such call spawns a linear sequence of recursive calls to \( \text{extend} \) from line 9 of \( \text{extend} \); the total time spent in \( \text{extend} \) is proportional to the number of calls in all these sequences. A simple induction using (1) proves that the number of calls in the sequence that originated with middle parameter \( j \), including the originating call, satisfies
\[
\text{number of calls} \leq 2 + j - \text{value returned}. \tag{3}
\]
Thus the total time spent in \( \text{extend} \) in calls from lines 10–11 of \( \text{prefix} \) is proportional to the sum of (3) over all values of \( j = \text{prefix}(\text{pattern}, i-1) \), where \( i = 0, 1, \ldots, \#\text{pattern} - 1 \). Of course, in such a call, the value returned is just \( \text{prefix}(\text{pattern}, i) \) and it will be put into \( \text{Prefix}[i] \). The sum of (3) thus telescopes to
\[
2 \times \#\text{pattern} + \text{Prefix}[0] - \text{Prefix}[\#\text{pattern} - 1],
\]
which is \( O(\#\text{pattern}) \).

We have already determined the total amount of time to compute entries in the array \( \text{Prefix} \), so we can assume that calls to \( \text{prefix} \) are \( O(1) \) time. Under this assumption, the cost of all calls to \( \text{extend} \) from lines 10–11 in \( \text{match} \) is a telescoping sum by (3) because the value returned by such a call is the value supplied for \( m \) on the subsequent call. The sum thus telescopes to
\[
2 \times \#\text{text} + \text{Prefix}[0] - \text{Prefix}[k],
\]
for some value of \( k \) depending on \( \text{text} \); this expression is \( O(\#\text{text}) \) for any value of \( k \).

We conclude that the total time for the algorithm is \( O(\#\text{pattern} + \#\text{text}) \).
7 Concluding Remarks

The implementation above would simplify ever so slightly if we used $prefix(j)$ (and hence also $Prefix[j]$) to be the length of the longest proper prefix of $pattern[0..j]$ that is a suffix of $pattern[0..j]$. However, the prefix function would then not be so closely akin to what is used generally in the literature.

Another minor simplification results from keeping track of the first unknown value in $Prefix$ instead of initializing the entire array to $-1$. This makes the code a bit shorter, but the proof of correctness is messier. This simplification would be important if we were to write K-M-P search incrementally with the use of the backspace character, much like the GNU Emacs incremental search because we would not know the length of the pattern a priori. The time required by an incremental form of the algorithm would be $O(#pattern + (B + 1) \times #text)$, where $B$ is the number of backspace characters in $pattern$.

References
