String algorithms *

(Knuth-Morris-Pratt algorithm, data structures for strings)

In this and the next two sections we will talk about strings, which are really just arrays. The elements of the array come from a constant size set \( \Sigma \) called the alphabet; the elements themselves are called characters. Common examples are English text whose alphabet consists of 26 letters plus special characters, strands of DNA constructed from an alphabet of four nucleotides, and proteins constructed from an alphabet of twenty amino acids.

Text and pattern. The problem we want to solve is the following. Given two strings, a text \( T[1..n] \) and a pattern \( P[1..m] \), find the first substring of the text that is the same as the pattern. Here a substring is just a contiguous subarray. For any shift \( 0 \leq s \leq n-m \) let \( T_s \) denote the substring \( T[s+1..s+m] \). More formally, we want to find the smallest shift \( s \) such that \( T_s = P \), or report that there is no match.

The straightforward approach to solving this problem uses two nested for-loops. The outer loop enumerates the \( T_s \) and the inner loop compares \( T_s \) with \( P \). We improve this algorithm by exiting the inner loop as soon as we find the first mismatch.

\begin{verbatim}
for s = 0 to n - m do j = 1;
    while j <= m and T[s + j] = P[j] do
        j++
    endwhile;
    if j = m + 1 then
        print "T_s = P"; stop
    endif
endfor; print "there is no match".
\end{verbatim}

In the worst case, we test \( P \) against \( n - m + 1 \) substrings comparing \( m \) pairs of characters in each test, which takes time \( O(nm) \). We can in fact create an input where the running time is as high as this pessimistic estimate: take a text consisting of \( n \) A's and search with a pattern AA...AB of \( m-1 \) A's followed by one B. On the other hand, breaking out of the inner loop at the first mismatch makes the algorithm quite practical. Certainly for random strings, the probability of having long common substrings is rather small. But then again, text is typically not random.

*These notes are extracted from the lecture notes on “Design and Analysis of Algorithms”, Fall 2005 by Herbert Edelsbrunner.
Redundant comparisons. Suppose we are looking for the pattern ABRACADABRA in some longer text using the above straightforward algorithm. Consider the case shown in Figure 1 in which, for shift $s = 10$, the substring comparison fails at the fifth position. At this point the algorithm increments the shift $s$ and starts the substring!

HOCUSPOCUSABRA | BRACADABRA . . .
ABRA | CADABRA
ABR | ACADABRA
AB | RACADABRA
A | BRACADABRA

Figure 1: Action of the straightforward algorithm. The box marks the leftmost mismatch for each shift.

comparison from scratch. Note, however, that there is no point in looking at the shift $s = 11$. We already know that $T[12] = B$ because it matched $P[2]$ during the previous comparison. Likewise, we already know that the next shift $s = 12$ also fail, so why bother looking there? Finally, when we get to $s = 13$ we cannot immediately rule out a match based on earlier considerations. However, since we already know that $T[14] = P[4] = A$, we should not start the substring comparison from scratch. Instead, we should start the substring comparison at the second character of the pattern, since we do not yet know whether or not it matches the corresponding text character.

Notice that with these improvements the character comparison should always advance through the text. More precisely, once we have found a match for a text character, we never need to do another comparison with that character again. In other words, we should improve the straightforward algorithm so that it always advances through the text. We also need a good rule for finding the next shift. Remember that a prefix of a string is a substring that includes the first character. Symmetrically, a suffix is a substring that includes the last character. A prefix or suffix is proper if it is not the entire string. Suppose that we have just discovered that $T[i] \neq P[j]$. The next reasonable shift is the smallest value of $s$ such that $T[i+1..i-s]$, which is a suffix of the previously read text, is also a proper prefix of the pattern; see Figure 2.

$$
\begin{array}{cccc}
 & T & & \\
old & A & B & R & ACADABRA \\
new & & & & \\
& & & & \\
\ne & & & & \\
\end{array}
$$

Figure 2: The new position, $s$, precedes the first character of the largest suffix that is a proper prefix of the pattern.

Finite state machines. If we have a string matching algorithm that always advances through the text, we can interpret it as feeding the text through a finite state machine,
which is a directed graph with labeled vertices. Each vertex is called a state and is labeled with a character from the pattern, except for two special states which are labeled $!$.

Figure 3: The finite state machine for ABRACADABRA. The (thick) success edges connect the characters in sequence while the (thin) failure edges return to earlier positions in the string.

$!$. Success edges connect the characters in the sequence of the pattern beginning at $!$ and ending at $. Failure edges point back to earlier characters. Figure 3 illustrates the idea for the pattern ABRACADABRA which we considered earlier.

We use the finite state machine to search for the pattern as follows. At all times, we have a current text character $T[i]$ and a current state, which is usually labeled by some pattern character $P[j]$. Initially, $i = 0$ and the current state is the one labeled $!$. We iterate the following two rules:

- If $T[i] = P[j]$ or the current label is $!, then we follow the success edge and increment $i$;
- If $T[i] \neq P[j]$, then we follow the failure edge back to an earlier state and we keep $i$ unchanged.

The finite state machine is a convenient metaphor for a simple type of algorithm.

**Knuth-Morris-Pratt algorithm.** In a real implementation, we would of course not construct the entire graph. Since the success edges go through the pattern in order, we only have to remember the failure edges. Each state has one failure edge (except for states labeled with the two special characters, which have none) and we encode them in an array $Fail[1..m]$ so that for each $j$ there is a failure edge from state $j$ to state $Fail[j]$. Following a failure edge back to an earlier state corresponds to shifting the pattern forward. For now, we assume that the failure edges are correctly computed and stored in the array $Fail$. The algorithm implementing the finite state machine then looks as follows.
\[ j = 1; \]
\[
\text{for } i = 1 \text{ to } n \text{ do}
\]
\[
\text{while } j > 0 \text{ and } T[i] \neq P[j] \text{ do}
\]
\[
\text{ } j = \text{Fail}[j]
\]
\[
\text{endwhile; } j++; \]
\[
\text{if } j = m + 1 \text{ then}
\]
\[
\text{print } "T_{i-m} = P"; \text{ stop}
\]
\[
\text{endif}
\]
\[
\text{endfor}; \text{ print } "\text{there is no match}."
\]

It is fairly easy to analyze the running time of the algorithm. At each character comparison, either we increase \( i \) and \( j \) by one each, or we decrease \( j \) and leave \( i \) unchanged. We can increment \( i \) at most \( n \) times before we run out of text, which implies that there are at most that many successful comparisons. Similarly there can be at most \( n - 1 \) failed comparisons, since the number of times we decrease \( j \) cannot exceed the number of times we increment \( j \). In other words, we amortize character mismatches over earlier character matches. The number of character comparisons performed by the Knuth-Morris-Pratt algorithm is less than \( 2n \), hence the running time is in \( O(n) \).

**Failure function.** We now rephrase our rule about how to choose a reasonable shift after a character mismatch \( T[i] \neq P[j] \): \( P[1..\text{Fail}[j] - 1] \) is the longest proper prefix of \( P[1..j - 1] \) that is also a suffix of \( T[1..i - 1] \). Notice, however, that if we compare \( T[i] \) with \( P[j] \), then we must have already matched the first \( j - 1 \) characters of the pattern. In other words, we already know that \( P[1..j - 1] \) is a suffix of \( T[1..i - 1] \). We can therefore substitute \( P[1..j - 1] \) for \( T[1..i - 1] \) in the above rule. We arrived at the definition of the Knuth-Morris-Pratt failure function \( \text{Fail}[j] \) for all \( j > 1 \). By convention, we set \( \text{Fail}[1] = 0 \), which tells the algorithm that if the first pattern character does not match it should give up and try the next text character. Table 1 shows the failure function for our standard pattern example. The way we compute this failure

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 1:** Failure function of the string ABRACADABRA.

function is essentially to use the Knuth-Morris-Pratt algorithm to look for the pattern inside itself. The variable \( k \) identifies \( P[1..k - 1] \) as the longest prefix of \( P[1..j - 1] \) that is also a suffix of \( P[1..j - 1] \). If \( P[j] \) matches \( P[k] \) we increment \( k \), else we try the previously computed next smaller prefix:

\[
k = 0; \]
\[
\text{for } j = 1 \text{ to } m \text{ do}
\]
\[
\text{ } \text{Fail}[j] = k;
\]
\[
\text{while } k > 0 \text{ and } P[j] \neq P[k] \text{ do}
\]
\[
\text{ } k = \text{Fail}[k]
\]
\[
\text{endwhile; } k++
\]
\[
\text{endfor.}
\]
Just as we did for the Knuth-Morris-Pratt algorithm, we can analyze the construction of
the failure function by amortizing character mismatches over earlier character matches.
Since there are at most $m$ character matches, the running time is in $O(m)$.

**Improvement.** We can speed up the algorithm by making one small change to the
failure function. Recall that after comparing $T[i]$ with $P[j]$ and finding a mismatch,
the algorithm compares $T[i]$ with $P[Fail[j]]$. With the current definition, it is possible
that $P[j]$ and $P[Fail[j]]$ are the same character, in which case the next character com-
parison will automatically fail. We can improve the failure function by short-cutting
these redundant comparisons using some simple post-processing.

```plaintext
for j = 2 to m do
  if P[j] = P[Fail[j]] then
    Fail[j] = Fail[Fail[j]]
  endif
endfor.
```

Alternatively, we can compute the improved failure function directly by substituting
lines 2, 3, 4 for the line 1 in the earlier algorithm.

```plaintext
2  if P[j] = P[k] then Fail[j] = Fail[k]
3    else Fail[j] = k
4  endif;
```

The improved failure function is shown in Table 2 and the corresponding finite state
machine is shown in Figure 4.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>K</th>
<th>C</th>
<th>A</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The improved failure function of ABRACADABRA. The changed failure pointers are
bold.

![Figure 4: The improved finite state machine for the pattern ABRACADABRA. The changed failure edges are dotted.](image-url)
We consider three generalizations of the string matching problem: searching with a collection of patterns instead of just one, allowing wild-cards in the pattern, and searching for common substrings. We present two data structures: the keyword tree, which solves the first two generalizations, and the suffix tree, which solves the first generalization and the third.

**String matching with a collection of patterns.** We generalize string matching to a collection of patterns \( P_1, P_2, \ldots, P_k \). For each pattern, we ask whether or not it occurs as a substring of a given text \( T[1..n] \). It is convenient to assume that the collection is prefix-free, that is, no pattern is a prefix of any other pattern, although this is not necessary but leads to complications we would like to avoid. With \( m_j \) the length of pattern \( P_j \), we let \( m = \sum_j m_j \) be the total length of the patterns. Assuming the alphabet has constant size, \( c \), we can store the patterns in a tree in which every node has degree at most \( c \). Each edge is labeled with a character, and the paths from the root to the leaves spell out the patterns. We refer to this as the *keyword tree* of the patterns, assuming the outgoing edges of a node are sorted according to an ordering of the alphabet, as in Figure 5. It is easy to construct the tree in time \( O(m) \). Each node \( \mu \) corresponds to a prefix \( L(\mu) \) of a pattern. If \( \mu \) is a leaf, \( L(\mu) \) is an entire pattern and we store its index \( j(\mu) \) at \( \mu \). To search with the text, we traverse a longest common prefix for each suffix \( T[s+1..n] \), which takes time \( O(nm) \).

**Failure function for keyword trees.** We improve the running time by adapting the idea of a failure function to keyword trees. For each node \( \mu \), let \( \ell(\mu) \) be the length of the longest suffix of \( L(\mu) \) that is a prefix of some pattern. We store a link \( \text{Fail}[\mu] \) from \( \mu \) to the node \( \nu \) for which \( L(\nu) \) is this prefix. This is illustrated in Figure 6. For convenience, we let \( \text{Fail}[\mu] \) point to the root if \( \ell(\mu) = 0 \). The search algorithm maintains a current node \( \mu \), which it initializes to the root, \( \sigma \), of the tree. It also uses two pointers into the text array: \( s \) precedes the starting position of the substring currently matched with the pattern, and \( i \) is the position currently tested.
Figure 6: The keyword tree of Figure 5 with failure links speeding up the search. The number next to each link is the depth of the target node. The links back to the root are not shown.

\[
s = 0; \ i = 1; \ \mu = g;
\]

repeat
  while \( \mu \) has child \( \nu \) with edge-label \( T[i] \) do
    if \( \nu \) is leaf then print “\( P_j(\nu) = T[s..i] \)”
    else \( \mu = \nu; \ i = i + 1 \)
  endif
endwhile;

\[ s = i - \ell(\mu); \ \mu = \text{Fail}[\mu] \]
until \( s > n \).

It is convenient to assume \( T[n+1] \) stores a special end-symbol that avoids patterns are compared with entries beyond the end of the text array. We omit the construction of the lengths \( \ell(\mu) \) and the failure links \( \text{Fail}[\mu] \), which takes time \( O(m) \). The running time of the search algorithm is \( O(n + m) \) because every step either advances the position in the text and simultaneously increases the depth of the node in the tree, or it reduces the depth, which cannot be done more often than increasing the depth.

**Sting matching with wild-cards.** We can use keyword trees to solve the string matching problem in which we allow for wild-cards, \(^*\), in the pattern. The wild-card is a special character that matches any (single) character in the text. For example, the pattern \( \text{ABRA}^{* * * * * * * \text{A}} \) occurs twice in \( \text{HOCUSPUSABRA} \text{BRACADABRA} \), starting after positions 10 and 13. Instead of concurrently following different branches for each wild-card, we search for each maximal substring without wild-cards, and we record each match at the text position preceding the first (full) pattern position. We use an integer array \( Q[0..n - 1] \) to record the matches. Initially, \( Q \) is all zero. Let \( P_1, P_2, \ldots, P_k \) be the maximal substrings of the pattern that do not contain any wild-cards. For each \( P_j \), we let \( p_j \) be the position preceding \( P_j \) in the pattern. For example, for \( P = \text{ABRA}^{* * * * * * * \text{A}} \) we have \( P_1 = \text{ABRA} \) with \( p_1 = 0 \) and \( P_2 = \text{A} \) with \( p_2 = 10 \). We store the \( P_j \) in a keyword tree with added failure links.
Step 1. For each $1 \leq j \leq k$, find all starting positions $s + 1$ of $P_j$ in $T$ and increment $Q[s - p_j]$ by one, provided $s \geq p_j$.

Step 2. Scan $Q$ and report every position $\ell$ with $Q[\ell] = k$.

In our example, $P_1 = \text{ABRA}$ increments $Q$ for indices 10, 13, and 20, and $P_2 = A$ increments $Q$ at indices 0, 3, 6, 8, 10, and 13. There are therefore two occurrences of the pattern in the text, the first at $T[11..21] = \text{ABRACADA}$ and the second at $T[14..24] = \text{ABRACADABRA}$. The total running time is $O(n + m)$.

**Suffix trees.** The string matching algorithms we have discussed so far preprocess the pattern, which is the smaller of the two strings, and use the obtained structure to find matches in time proportional to the length of the text. We now turn things around and preprocess the text. Specifically, we take the collection of suffixes, $T_s = T[s + 1..n]$, and construct the keyword tree for $T_0; T_1; \ldots; T_n$. To avoid complications, we enforce the prefix-free property by appending the special character $\$ to each $T_s$. We thus get a bijection between the leaves of the tree and the suffixes of the text. In each leaf, we record the position preceding the starting position of the corresponding suffix. Finally, we remove each non-branching internal node, merging its two edges into one and concatenating their labels. We thus obtain the suffix tree of $T$, illustrated in Figure 7. We note that because every internal node has two or more children, the size of the tree is linear in the length of the text. Indeed, there are $n + 1$ leaves and therefore at most $n$ internal nodes and at most $2n$ edges. An edge-label can be an arbitrarily long string, but we can store it using only two integers giving the first and last positions in the text. We also note that any two edges connecting a node with its children are labeled by strings that begin with different characters.

It is straightforward to construct the suffix tree in time $O(n^2)$, simply by adding the suffixes one at a time. It is more difficult but possible to construct it in time $O(n)$. One strategy is to read the text from front to back, and for each new character to expand all suffixed by one and start one new suffix. The details are complicated and omitted.

**String matching with suffix trees.** Given the suffix tree for text $T$, we can determine whether or not the pattern $P$ is a substring of $T$ by traversing a single path.

**Case 1.** We exhaust $P$ and thus find a suffix of $T$ that has $P$ as a prefix.
**Case 2.** We could not exhaust \( P \) implying that \( P \) is not a substring of \( T \).

The time to search in the tree is \( O(m) \). In Case 1, we have an internal node whose path from the root spells out \( P \), plus possibly a few additional characters at the end, if we used only a portion of the last edge's label. To find all occurrences of \( P \), we can traverse the subtree of this node and report the starting positions of the suffixed stored at the leaves. Since each internal node has two or more children, this takes time linear in the number of occurrences found.

Using the linear-time algorithm for constructing the suffix tree as a preprocessing step, we can solve the string matching problem in time \( O(n + m) \), which is the same as for the Knuth-Morris-Pratt algorithm. An advantage of using the suffix tree is that we can search with many patterns, without paying again for the length of the text. The performance is the same as that of the keyword tree, but now we do not have to know the patterns in advance. A disadvantage of using the suffix tree is the extra memory we need for storing the tree.

**Longest common substrings.** A classic problem in string analysis is to find the longest substring common to two given strings, \( T_1 \) and \( T_2 \). For example, if \( T_1 = \text{austrialife} \) and \( T_2 = \text{australialife} \) then the longest common substring is \( \text{ialife} \). To find it, we construct the suffix tree for both texts, representing the suffixes of \( T_1 \) and of \( T_2 \), each by a path from the root to a leaf. Each leaf represents a suffix of one string or of both. We mark each internal node \( \mu \) with “1” if at least one of the leaves in its subtree represents a suffix of \( T_1 \). Similarly, we mark \( \mu \) with “2” if at least one of the leaves in its subtree represents a suffix of \( T_2 \). If an internal node has both marks, then its path spells out the prefix of a suffix of \( T_1 \) as well as a suffix of \( T_2 \). In other words, it spells out a substring of both. Call the length of this substring the **string-depth** of \( \mu \). To find the longest common substring, we just need to find the internal node with maximum string-depth that has both marks. We summarize the algorithm:

**Step 1.** Construct the suffix tree for \( T_1 \) and \( T_2 \).

**Step 2.** Mark internal nodes and determine their string-depths.

**Step 3.** Return the node with maximum string-depth that has marks for both strings.

Letting \( n_i \) be the length of \( T_i \), Step 1 constructs the suffix tree in time \( O(n_1 + n_2) \). The linear-time algorithm mentioned above extends to the case of two (or more) strings, so we can construct the suffix tree of \( T_1 \) and \( T_2 \) in time \( O(n_1 + n_2) \). The size of the tree is \( O(n_1 + n_2) \), and Step 2 marks the internal nodes and computes their string-depths in the same amount of time. Finally, Step 3 find the right node in the same amount of time. In summary, we have an algorithm that determines the longest common substring in time linear in the total length of the two strings.

**Suffix arrays.** Given a text \( T[1..n] \), the **suffix array**, \( \text{Pos}[0..n] \), records the lexicographic order of the \( n+1 \) suffixes. Specifically, the suffix \( T_{\text{Pos}[i]} = T[\text{Pos}[i]+1..n] \) is lexicographically smaller than \( T_{\text{Pos}[i+1]} \). As an example, consider \( T = \text{mississippi} \)
and order its suffixes lexicographically:

11 : 
10 : i 
7 :ippi 
4 :issippi 
1 :ississippi 
0 :mississippi 
9 :pi 
8 :ppi 
6 :sippi 
3 :sissippi 
5 :ssippi 
2 :ssississippi

The suffix array stores 12 integers, one more than the length of the text, which for $T = \text{mississippi}$ are 11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2. We can construct the suffix array from the suffix tree by in-order traversal, but for that we have to interpret $\$ \text{lexicographically smaller than all other characters in the alphabet. A key property}$ 
of the array is that it groups suffixes with common prefixes together in contiguous positions. We can therefore use binary search to find all suffixes that contain a given pattern $P$. This takes time $O(m \log n)$, which is not quite as fast as the suffix tree itself, but the array takes less memory. Also, there are methods that can speed up the search to time $O(m + \log n)$, which as fast as the suffix tree unless the size of the text exceeds 2 to the size of the pattern.