Chapter E1: String Matching Algorithms

The input to a string-matching problem is a short string \( P \) of length \( m \) called the pattern, and a long string \( T \) of length \( n \) called the text. The task is to find all occurrences of the pattern string in the text string.

**Example.** If the pattern is `ana` and the text is `bananaNirvana`, then there are 3 occurrences of the pattern.

A prefix is an initial segment of a string and a suffix is an end segment of a string. We use \( \varepsilon \) to denote the empty string. Note that \( \varepsilon \) is always both a prefix and a suffix. The alphabet is the legal characters in the two strings combined and is denoted by \( \Sigma \).

E1.1 The Naive Algorithm

The simplest idea is to consider the strings stored as arrays. A window is a subarray of the text of length equal to the pattern.

```
Naive Algorithm
for all windows (of length \( m \)) in \( T \) do
   check whether window matches \( P \) or not
```

The inner check can take time proportional to \( m \), in particular if there is actually a match; and the outer loop is executed nearly \( n \) times. So the naive algorithm takes \( O(mn) \) time. Indeed, it can be shown that the naive algorithm has both worst case and typical case \( O(mn) \). That is, time taken is proportional to the product of \( m \) and \( n \). The advantage is simplicity.

One can, however, hope to do better. The key observation is that we do not use the information gained in a check. For example, if the pattern is `abc` and a match is detected, then the next two windows cannot be matches because they start with `b` and `c`.

E1.2 Knuth–Morris–Pratt Algorithm

The algorithm given by Knuth, Morris, and Pratt (KMP) remembers the information that has been gained so far. That is, to do the check for the next window it uses a particular summary from the previous window. In particular, it processes the text one character at a time using a finite automaton: namely, the finite automaton that
accepts all strings ending with $P$. Thus the running time is $O(m + n)$, after some (significant) preprocessing.

For example, here is the automaton for $abbcab$. To avoid clutter, some transitions are omitted: if a state is missing a transition, it is assumed to go to state 0.

Now, there is a generic algorithm to produce a (deterministic) finite automaton from a given regular expression. (See, for example, any Introduction to the Theory of Computation.) This algorithm would produce the automaton in time potentially exponential in $m$.

However, under this scenario we can do better. Each state in the automaton corresponds to a prefix $s$ of the pattern; in particular there are $m + 1$ states. And:

the transition out of state $s$ on character $c$ goes to: the longest prefix (initial segment) of $P$ that is also a suffix (final segment) of the string $sc$.

**Example.** If the pattern is $ananarama$, then prefix $anana$ goes to $anan$ on the character $n$, and to $\varepsilon$ on the character $m$.

The calculation of each transition takes $O(m^2)$ time naively. Thus the automaton can be generated in time proportional to $m^3|\Sigma|$.

The main contribution of KMP is to show how to construct the automaton in time proportional to $m$. We leave that for another day.

### E1.3 Boyer-Moore Algorithm

The algorithm introduced by Boyer and Moore (BM) is more like the naive algorithm. The contribution is that one can move the window along more than one character at a time by using what are called *heuristics*. One also matches the window from *right-to-left*.

There are two heuristics. Each provides an amount by which the window can be advanced. The larger value is then used, and the process starts again.
1. **The bad character heuristic.** This heuristic considers the characters of the text as read. For example, if the rightmost character in the window does not appear in the pattern, then one can advance the window \( m \) places. In general, one can advance the window so that the character just read is aligned with the “next” occurrence of it in the pattern.

*Example.* Suppose the pattern is `bananaNirvana` and the corresponding window in the text ends `ia`. Then the mismatched character `i` means one can advance the window by 4 places.

For efficiency, this position is pre-calculated and stored. The cheapest procedure is just to calculate for each character in the alphabet its rightmost occurrence in the pattern. Unfortunately though, sometimes this rightmost occurrence is to the right of the current position and then the heuristic is not helpful. So another procedure is to precalculate for each character in the alphabet and for each position in the pattern, the closest occurrence of that character to the left of the current position.

2. **The good-match heuristic.** This heuristic considers the match so far and finds the first occurrence earlier in the pattern of this suffix.

*Example.* Suppose the pattern is `bananaNirvana` and the corresponding window in the text ends `dana`. Then one can shift the window forward so that the `ana` in the text is now aligned with the `ana` in positions 4 through 6 of the pattern.

Again, the re-alignment counts are pre-calculated and stored.

Clearly, further improvements are possible. But one has to show that the benefit achieved outweighs the extra calculations made each time it doesn’t help.

The overall running time of the algorithm is \( O(mn) \), which is no better than the naive algorithm in the worst case. But in practice, BM runs very fast.

### E1.4 Rabin-Karp Algorithm

A different approach was given by Rabin and Karp. In this algorithm, a value is calculated for each window (called its **signature**). (Essentially a hash function.) This is compared with the signature of the pattern. Different signatures guarantee a mismatch. However, the same signature does not guarantee a match and this must be checked (called a **spurious hit**).

The signatures must be found in linear time for all windows. So one uses a procedure that can, in constant time, determine the signature of the next window from the signature of the current one.
Method: Treat the window as a number and calculate the remainder modulo a fixed prime \( p \).

**Example.** The text is 247531189 and pattern is 4753 and working modulo 11. The signature of the pattern is 4753\( \mod 11 = 1 \). The signatures of the windows are:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

For example, to get the signature for the second window from the first:

\[
4753 = (2475 - 2000) \times 10 + 3 \\
\equiv (0 - 9) \times 10 + 3 \\
= -87 \\
\equiv 1
\]

where we use “\( \equiv \)” to mean “has the same remainder modulo 11 as”. This uses the fact that if one wants the answer modulo \( p \), then one can take remainders modulo \( p \) at each stage of the calculation. (To speed up things, the remainders of 1,000, 2,000, 3,000 etc. are pre-calculated.)

While the RK algorithm does no better than \( O(mn) \), the big advantage is that it is easily generalizable to other situations such as two-dimensional pattern matching, and looking for multiple patterns.

**Exercises**

1. Construct an example where the naive algorithm takes \( O(mn) \) time even if the check routine stops as soon as a mismatch is detected.

2. Apply the KMP algorithm with pattern `abbac` and text `cabbabbacababbac`.

3. Calculate the BM functions for both patterns `abbac` and `cabbabbacababbac`.

4. Apply the Rabin-Karp algorithm with pattern `1521` and text `9152152144` first for prime 7 and then for prime 13.

5. Consider a generalization of the problem where the pattern does not have to appear consecutively in the text but arbitrarily long gaps are allowed. Discuss algorithms for this problem.