Exact String Matching

The Knuth-Morris-Pratt Algorithm
Outline for Today

• The Exact Matching Problem
• A simple algorithm
• Motivation for better algorithms
• The Knuth-Morris-Pratt algorithm
The Exact Matching Problem

• Consider a text $T[1,\ldots,m]$ with $m$ characters and a pattern $P[1,\ldots,n]$ with $n$ characters, both over the same alphabet

• We say that $P$ occurs in $T$ with shift $s \geq 0$ if
  \[ P[1,\ldots,n] = T[s+1,\ldots,s+n] \]

• Problem: Find all shifts of $P$ in $T$
Motivation

- Word processing, information retrieval, keyword searching, e.g., in digital libraries or the internet, searching genomics databases or matching a DNA fragment to a reference genome, ...
A Simple Algorithm

Simple-ExMa(P[1,\ldots,n], T[1,\ldots,m]) \ // \ 1 \leq n \leq m

for s \leftarrow 0 \text{ to } m - n

\quad \text{if } ( P[1,\ldots,n] = T[s+1,\ldots,s+n] )

\quad \text{output } s
A Simple Algorithm

Simple-ExMa(P[1,...,n], T[1,...,m]) // 1 ≤ n ≤ m

for s ← 0 to m − n
    if ( P[1,...,n] = T[s+1,...,s+n] )
        output s

Running time is Θ(mn)
A Simple Algorithm

Simple-ExMa(P[1,...,n], T[1,...,m]) // 1 ≤ n ≤ m

for s ← 0 to m – n
    if ( P[1,...,n] = T[s+1,...,s+n] )
        output s

Exercise: If on each iteration, characters are compared from left to right ONLY until a mismatch is found, will the algorithm still take Θ(mn) time in the worst case?
A Simple Algorithm

Simple-ExMa(P[1,...,n], T[1,...,m]) // 1 ≤ n ≤ m

for s ← 0 to m − n
    if ( P[1,...,n] = T[s+1,...,s+n] )
        output s

Exercise: If the pattern and text are chosen uniformly at random over an alphabet of size k, what is the expected time for the algorithm to finish?
Knuth-Morris-Pratt (KMP) Algorithm

• The KMP algorithm is similar to the naive algorithm: it considers shifts in order from 0 to m−n, and determines if the pattern matches at that shift.

• The difference is that the KMP algorithm uses information gleaned from partial matches of the pattern and text to “slide” over shifts that are guaranteed not to result in a match.
Knuth-Morris-Pratt (KMP) Algorithm

Example:


\[ T = \text{xyx} \quad \ldots \ldots \]
\[ P = \text{xyx} \quad \ldots \]

We can "slide" the pattern from shift 0 to shift 2 and continue comparing to \( T[4] \):

\[ T = \text{xyx} \quad \ldots \ldots \]
\[ P = \text{xyx} \quad \ldots \]

next comparison: \( P[2] = T[4]? \)
Another example:

\[ T = \text{xxxxx} \ldots \]
\[ P = \text{xxxxx} \ldots \]


How far can we slide the pattern, and continue comparing to \( T[6] \)?

\[ T = \text{xxxxx} \ldots \]
Knuth-Morris-Pratt (KMP) Algorithm

Another example:

\[ T = \text{x} \text{x} \text{x} \text{x} \text{x} \text{x} \ldots \]
\[ P = \text{x} \text{x} \text{x} \text{x} \text{x} \text{x} \ldots \]


We can "slide" the pattern from shift 0 to shift 1 and continue comparing to \( T[6] \):

\[ T = \text{x} \text{x} \text{x} \text{x} \text{x} \text{x} \ldots \]
\[ P = \text{x} \text{x} \text{x} \text{x} \text{x} \text{x} \ldots \]
Knuth-Morris-Pratt (KMP) Algorithm

Useful notation

Let $S = S[1,...,q]$ be a string
Knuth-Morris-Pratt (KMP) Algorithm

Useful notation

Let $S = S[1,...,q]$ be a string

- **Prefix**: string $S[1,...,i]$, for any $i$ with $0 \leq i \leq q$
- **Proper prefix**: a prefix with $i < q$

- **Suffix**: string $S[j,...,q]$ for any $j$ with $1 \leq j \leq q+1$
- **Proper suffix**: suffix with $1 < j$
Knuth-Morris-Pratt (KMP) Algorithm

Useful notation

Let $S = S[1,...,q]$ be a string
Knuth-Morris-Pratt (KMP) Algorithm

**Useful notation**

Let $S = S[1,...,q]$ be a string

Define $\pi[q]$ so that $S[1,..., \pi[q]]$ is the longest string that is both a proper prefix and suffix of $S[1,...,q]$

Let’s call $S[1,..., \pi[q]]$ the *KMP prefix* of $S[1,...,q]$
Knuth-Morris-Pratt (KMP) Algorithm

Useful notation

A KMP “slide” of the pattern P in text T:
• Before the slide, $P[1,...,q]$ is matched with $T[,...,i]$
• After the slide, $P[1,...,\pi(q)]$ is matched with $T[,...,i]$

(In the algorithm, a “slide” simply updates $q$ to $\pi(q)$)
Knuth-Morris-Pratt (KMP) Algorithm

Sliding rule

Mismatch: Suppose that $P[1,\ldots,q] = T[\ldots,i]$ and a mismatch occurs: $P[q+1] \neq T[i+1]$

If $q = 0$

If $q > 0$
**Knuth-Morris-Pratt (KMP) Algorithm**

*Sliding rule*

*Mismatch*: Suppose that $P[1,\ldots,q] = T[\ldots,i]$ and a mismatch occurs: $P[q+1] \neq T[i+1]$

If $q = 0$

*slide $P$ one to the right (by incrementing $i$)*

If $q > 0$
Knuth-Morris-Pratt (KMP) Algorithm

Sliding rule

Mismatch: Suppose that $P[1,...,q] = T[,...,i]$ and a mismatch occurs: $P[q+1] \neq T[i+1]$

If $q = 0$
   
   *slide $P$ one to the right (by incrementing $i$)*

If $q > 0$
   
   *do a KMP slide of $P$ (by updating $q$ to $\pi(q)$)*
Knuth-Morris-Pratt (KMP) Algorithm

Sliding rule

Match: Suppose that $P[1,...,q] = T[,...,i]$ and a match occurs: $P[q+1] = T[i+1]$
Knuth-Morris-Pratt (KMP) Algorithm

*Sliding rule*

**Match:** Suppose that $P[1, \ldots, q] = T[\ldots, i]$ and a match occurs: $P[q+1] = T[i+1]

*increment i and q*

If $q = n$

we’ve found an exact match of $P$ at shift $i-n$!

*do a KMP slide of $P$ (by updating $q$ to $\pi(q)$)*
Knuth-Morris-Pratt (KMP) Algorithm

algorithm KMP(P[1,...,n],T[1,...,m]) // 1 ≤ n ≤ m
output: list of all numbers s such that P occurs with shift s in T
q ← 0; i ← 0;

repeat // invariant: P[1,...,q] = T[i−q+1,...,i]
    if (P[q+1] ≠ T[i+1]) //mismatch
        if (q = 0) i ← i + 1
        else q ← π(q)
    else // match
        q ← q + 1; i ← i + 1;
        if q = n
            output i−n // there's an exact match of P at shift i-n
        q ← π(q); // slide the pattern to the right
until (i = m) // end of text has been reached
Knuth-Morris-Pratt (KMP) Algorithm

Runtime: $O(n)$ time to pre-process the pattern $P$; $O(m)$ post-processing time to search $T$

Useful when searching for a given pattern in many texts, e.g., searching a large and often-changing distributed collection of online resources for the same pattern
Knuth-Morris-Pratt (KMP) Algorithm

Pre-processing the pattern: calculating KMP indices

Recall: $P[1,\ldots, \pi[q]]$ is the longest string that is both a proper prefix \textit{and} suffix of $P[1,\ldots,q]$
Knuth-Morris-Pratt (KMP) Algorithm

Pre-processing the pattern: calculating KMP indices

Recall: P[1,..., π[q]] is the longest string that is both a proper prefix and suffix of P[1,...,q]

<table>
<thead>
<tr>
<th>P:</th>
<th>x</th>
<th>y</th>
<th>x</th>
<th>y</th>
<th>y</th>
<th>y</th>
<th>x</th>
<th>y</th>
<th>x</th>
<th>y</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>q:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>π(q):</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Knuth-Morris-Pratt (KMP) Algorithm

Pre-processing the pattern: calculating KMP indices

Compute-π-values(P [1, . . . , n])

π(1) ← 0;

for i ← 1 to n − 1 // calculate π[i + 1], given π[1, . . . , i]

output π[1, . . . , n]
Knuth-Morris-Pratt (KMP) Algorithm

Pre-processing the pattern: calculating KMP indices

Compute-\(\pi\)-values\((P[1, \ldots, n])\)
\[
\pi(1) \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } n - 1 \quad \text{// calculate } \pi[i+1], \text{ given } \pi[1, \ldots, i] \\
\quad q \leftarrow \pi[i] \\
\quad \text{while } (P[q+1] \neq P[i+1]) \text{ and } (q > 0)) \\
\quad \quad q \leftarrow \pi[i] \\
\quad \quad \text{if } (P[q+1] = P[i+1]) \\
\quad \quad \quad \pi(i+1) \leftarrow q+1 \\
\quad \quad \text{else} \\
\quad \quad \quad \pi(i+1) \leftarrow 0 \\
\text{output } \pi[1, \ldots, n]
Summary

• KMP solves the Exact Matching Problem of finding all occurrences of P[1,..,n] in T[1,..,m] using O(n) pre-processing time and O(m) post-processing time

• KMP works even when the text T is presented in “real time”

• KMP has been generalized to handle sets of patterns (the Aho-Corasick algorithm)

• In other situations, it’s useful to have an algorithm that takes O(m) pre-processing time but then can find all occurrences of P in T in time O(n + k), where there are k occurrences. We’ll look at algorithms to do this next time