Pattern Matching
Strings

- A string is a sequence of characters
- Examples of strings:
  - Java program
  - HTML document
  - DNA sequence
  - Digitized image
- An alphabet $\Sigma$ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - Unicode
  - $\{0, 1\}$
  - $\{A, C, G, T\}$

Let $P$ be a string of size $m$
- A substring $P[i..j]$ of $P$ is the subsequence of $P$ consisting of the characters with ranks between $i$ and $j$
- A prefix of $P$ is a substring of the type $P[0..i]$
- A suffix of $P$ is a substring of the type $P[i..m-1]$

Given strings $T$ (text) and $P$ (pattern), the pattern matching problem consists of finding a substring of $T$ equal to $P$

Applications:
- Text editors
- Search engines
- Biological research
Brute-Force Pattern Matching

- The brute-force pattern matching algorithm compares the pattern $P$ with the text $T$ for each possible shift of $P$ relative to $T$, until either
  - a match is found, or
  - all placements of the pattern have been tried

- Brute-force pattern matching runs in time $O(nm)$

- Example of worst case:
  - $T = \text{aaa} \ldots \text{ah}$
  - $P = \text{aaah}$
  - may occur in images and DNA sequences
  - unlikely in English text

Algorithm $\text{BruteForceMatch}(T, P)$

**Input** text $T$ of size $n$ and pattern $P$ of size $m$

**Output** starting index of a substring of $T$ equal to $P$ or $-1$ if no such substring exists

for $i \leftarrow 0$ to $n - m$

\{ test shift $i$ of the pattern \}

$j \leftarrow 0$

while $j < m \land T[i + j] = P[j]$

\[ j \leftarrow j + 1 \]

if $j = m$

\{ match at $i$ \}

return $i$

else

break while loop \{ mismatch \}

return $-1$ \{ no match anywhere \}
Boyer-Moore Heuristics

The Boyer-Moore’s pattern matching algorithm is based on two heuristics

**Looking-glass heuristic:** Compare $P$ with a subsequence of $T$ moving backwards

**Character-jump heuristic:** When a mismatch occurs at $T[i] = c$
- If $P$ contains $c$, shift $P$ to align the last occurrence of $c$ in $P$ with $T[i]$
- Else, shift $P$ to align $P[0]$ with $T[i + 1]$

**Example**

| a | p | a | t | t | e | r | n | m | a | t | c | h | i | n | g | a | l | g | o | r | i | t | h | m |
| r | i | t | h | m | 1 | r | i | t | h | m | 3 | r | i | t | h | m | 5 | 11 | 10 | 9 | 8 | 7 |
| r | i | t | h | m | 2 | r | i | t | h | m | 4 | r | i | t | h | m | 6 | r | i | t | h | m |
Last-Occurrence Function

Boyer-Moore’s algorithm preprocesses the pattern $P$ and the alphabet $\Sigma$ to build the last-occurrence function $L$ mapping $\Sigma$ to integers, where $L(c)$ is defined as:

- the largest index $i$ such that $P[i] = c$ or
- $-1$ if no such index exists

**Example:**

- $\Sigma = \{a, b, c, d\}$
- $P = abacab$

<table>
<thead>
<tr>
<th></th>
<th>$c$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

The last-occurrence function can be represented by an array indexed by the numeric codes of the characters.

The last-occurrence function can be computed in time $O(m + s)$, where $m$ is the size of $P$ and $s$ is the size of $\Sigma$.
The Boyer-Moore Algorithm

Algorithm BoyerMooreMatch(\(T, P, \Sigma\))
\[
L \leftarrow \text{lastOccurrenceFunction}(P, \Sigma)
\]
\[
i \leftarrow m - 1
\]
\[
j \leftarrow m - 1
\]
repeat
  if \(T[i] = P[j]\)
    if \(j = 0\)
      return \(i\) \{ match at \(i\) \}
    else
      \(i \leftarrow i - 1\)
      \(j \leftarrow j - 1\)
  else
    \{ character-jump \}
    \(l \leftarrow L[T[i]]\)
    \(i \leftarrow i + m - \min(j, 1 + l)\)
    \(j \leftarrow m - 1\)
until \(i > n - 1\)
return \(-1\) \{ no match \}

Case 1: \(j \leq 1 + l\)

Case 2: \(1 + l \leq j\)
Example
Analysis

- Boyer-Moore’s algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore’s algorithm is significantly faster than the brute-force algorithm on English text
The KMP Algorithm

- Knuth-Morris-Pratt’s algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$.

No need to repeat these comparisons
Resume comparing here
KMP Failure Function

Knuth-Morris-Pratt’s algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.

The failure function $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$.

Knuth-Morris-Pratt’s algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j - 1)$.
The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time.
- At each iteration of the while-loop, either
  - $i$ increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP’s algorithm runs in optimal time $O(m + n)$

Algorithm $\text{KMPMatch}(T, P)$

```plaintext
F \leftarrow \text{failureFunction}(P)
i \leftarrow 0
j \leftarrow 0

\text{while } i < n
  \text{if } T[i] = P[j]
    \text{if } j = m - 1
      \text{return } i - j \{ \text{ match } \}
    \text{else}
      i \leftarrow i + 1
      j \leftarrow j + 1
  \text{else}
    \text{if } j > 0
      j \leftarrow F[j - 1]
    \text{else}
      i \leftarrow i + 1
  \text{return } -1 \{ \text{ no match } \}
```
Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(m)$ time.
- The construction is similar to the KMP algorithm itself.
- At each iteration of the while-loop, either
  - $i$ increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$).
- Hence, there are no more than $2m$ iterations of the while-loop.

Algorithm $\text{failureFunction}(P)$

```plaintext
\begin{align*}
F[0] & \leftarrow 0 \\
i & \leftarrow 1 \\
j & \leftarrow 0 \\
\text{while } i < m \\
\quad & \text{if } P[i] = P[j] \\
\quad & \quad \{ \text{we have matched } j + 1 \text{ chars} \} \\
\quad & \quad F[i] \leftarrow j + 1 \\
\quad & \quad i \leftarrow i + 1 \\
\quad & \quad j \leftarrow j + 1 \\
\quad & \text{else if } j > 0 \text{ then} \\
\quad & \quad \{ \text{use failure function to shift } P \} \\
\quad & \quad j \leftarrow F[j - 1] \\
\quad & \text{else} \\
\quad & F[i] \leftarrow 0 \{ \text{ no match } \} \\
\quad & i \leftarrow i + 1
\end{align*}
```
Example

\[
\begin{array}{ccccccc}
 & a & b & a & c & a & a & b & a & c & c & a & b & a & c & a & b & a & a & b & b \\
1 & 2 & 3 & 4 & 5 & 6 \\
 & a & b & a & c & a & b \\
7 & a & b & a & c & a & b \\
 & 8 & 9 & 10 & 11 & 12 \\
 & a & b & a & c & a & b \\
 & 13 \\
 & a & b & a & c & a & b \\
 & 14 & 15 & 16 & 17 & 18 & 19 \\
 & a & b & a & c & a & b \\
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
j & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
P[j] & a & b & a & c & a & b \\
\hline
F(j) & 0 & 0 & 1 & 0 & 1 & 2 \\
\hline
\end{array}
\]