Strings and Pattern Matching
Outline

- Strings
- Pattern matching algorithms
  - Brute-force algorithm
  - Boyer-Moore algorithm
  - Knuth-Morris-Pratt algorithm
Strings

- A string is a sequence of characters
- Examples of strings:
  - C++ 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 (used by C and C++)
  - Unicode (used by Java)
  - $\{0, 1\}$
  - $\{A, C, G, T\}$

$P$ is 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 $P[i .. m - 1]$
Alphabets

<table>
<thead>
<tr>
<th>name</th>
<th>R()</th>
<th>lgR()</th>
<th>characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>2</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>OCTAL</td>
<td>8</td>
<td>3</td>
<td>01234567</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>10</td>
<td>4</td>
<td>0123456789</td>
</tr>
<tr>
<td>HEXADECIMAL</td>
<td>16</td>
<td>4</td>
<td>0123456789ABCDEF</td>
</tr>
<tr>
<td>DNA</td>
<td>4</td>
<td>2</td>
<td>ACTG</td>
</tr>
<tr>
<td>LOWERCASE</td>
<td>26</td>
<td>5</td>
<td>abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>26</td>
<td>5</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>PROTEIN</td>
<td>20</td>
<td>5</td>
<td>ACDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>BASE64</td>
<td>64</td>
<td>6</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZabcddefghijklmnopqrstuvwxyz0123456789+/</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td>7</td>
<td>ASCII characters</td>
</tr>
<tr>
<td>EXTENDED.ASCII</td>
<td>256</td>
<td>8</td>
<td>extended ASCII characters</td>
</tr>
<tr>
<td>UNICODE16</td>
<td>65536</td>
<td>16</td>
<td>Unicode characters</td>
</tr>
</tbody>
</table>
Interesting Fact

**IBM System/360 defined 8 bit byte (1964)**

- Instead of 4 or 6 bits which was cheaper
- Enabled the extended ASCII set, both upper and lowercase, as well as symbols
- Huge ramifications!
- Decided by Fred Brooks
  - His most important decision:
    "The most important single decision I ever made was to change the IBM 360 series from a 6-bit byte to an 8-bit byte, thereby enabling the use of lowercase letters. That change propagated everywhere."
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
- And many others...
Brute-Force Algorithm

- Compares the pattern \( P \) (of length \( m \)) with the text \( T \) (of length \( n \)) for each possible shift of \( P \) relative to \( T \), until
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in what time?
  - \( O(nm) \)
- What is an example 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 \( j = m \)

return \( i \) \{match at \( i \}\}

else

break while loop \{mismatch\}

return \(-1\) \{no match anywhere\}
Brute Force Algorithm 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
Brute Force Algorithm 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
Brute Force Algorithm 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 |
Brute Force Algorithm Example
Brute Force Algorithm Example
How can we do better? Ideas?

Pattern Matching
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:

```
pattern matching algorithm
```

```
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
```
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**

```
pattern matching algorithm
```

1. `r i t h m`
2. `r i t h m`

```
1
pattern matching
```

```
2
pattern matching
```
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:

```
 Pattern Matching algorithm

1 r i t h m
2 r i t h m
3 r i t h m
```
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**

```
1  r i t h m
2  r i t h m
3  r i t h m
4  r i t h m
```
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 | r | i | t | h | m | r | i | t | h | m | r | i | t | h | m | r | i | t | h | m |
| 1 | 3 | 5 |

| r | i | t | h | m |
| 2 | 4 |

Patterns Matching
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
```

1. $r  i  t  h  m$
2. $r  i  t  h  m$
3. $r  i  t  h  m$
4. $r  i  t  h  m$
5. $r  i  t  h  m$
6. $r  i  t  h  m$

Pattern Matching
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 |
| 2 |
| r | i | t | h | m |
| 3 |
| r | i | t | h | m |
| 4 |
| r | i | t | h | m |
| 5 |
| r | i | t | h | m |
| 6 |
```

Pattern Matching
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

```
atternmatchingalgorithm

1 2 3 4 5 6 7 8 9 10 11

1 2 3 4 5 6

1 2 3 4 5 6

```

Pattern Matching
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>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>
</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 $\Sigma$. 
Additional examples

- $P=ab, S=\{a,b,c,d\}$
  - $L=[a,0], [b,1], [c,-1], [d,-1]$

- $P=abab$
  - $L=[a,2], [b,3], [c,-1], [d,-1]$

- $P=dcba$
  - $L=[a,3], [b,2], [c,1], [d,0]$
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 \}

$l \leftarrow L[T[i]]$

$i \leftarrow i + m - \text{min}(j, 1 + l)$

$j \leftarrow m - 1$

until $i > n - 1$

return $-1$ \{ no match \}

Case 1: $j \leq 1 + l$

Case 2: $1 + l \leq j$
Example

Pattern Matching
Analysis

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

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

```plaintext
F \leftarrow \text{failureFunction}(P)
i \leftarrow 0
j \leftarrow 0
while i < n
  if $T[i] = P[j]$
    if $j = m - 1$
      return $i - j$ { match }
    else
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
  else
    if $j > 0$
      $j \leftarrow F[j - 1]$
    else
      $i \leftarrow i + 1$
return $-1$ { 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 $\text{failureFunction}(P)$

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

Pattern:
aba
123

Failure function f:
0 0 1
Additional Example

Pattern:
\[a \ a \ b \ a \ a \ b \ a \ b \ b \]
\[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9\]

Failure function \(f\):

??
Additional Example

Pattern:
\[ \text{a a b a a b a b b} \]
\[ \text{1 2 3 4 5 6 7 8 9} \]

Failure function \( f \):
\[ f(\text{a}) = 0 \text{ (always } = 0 \text{ for one letter)} \]
\[ f(\text{aa}) = 1 \text{ ('a' is both a prefix and suffix)} \]
\[ f(\text{aab}) = ? \]
Additional Example

Pattern:
\[ a \ a \ b \ a \ a \ b \ a \ b \ b \]
\[ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \]

Failure function \( f \):
\[ f(a) = 0 \] (always \( = 0 \) for one letter)
\[ f(aa) = 1 \] ('a' is both a prefix and suffix)
\[ f(aab) = 0 \] (no same suffixes and prefixes: \( a \neq b \), \( aa \neq ab \))
Additional Example

Pattern:

```
a a b a a b a b b
1 2 3 4 5 6 7 8 9
```

Failure function $f$:

$f(aaba) = ?$
Additional Example

Pattern:

a a b a a b a b b
1 2 3 4 5 6 7 8 9

Failure function $f$:

$f(aaba) = 1$ ('$a$' is the same in the beginning and the end, but if you take 2 letters, they won't be equal: $aa \neq ba$)
Additional Example

Pattern:
\[ a \ a \ b \ a \ a \ b \ a \ b \ b \]
\[ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \]

Failure function \( f \):
\[ f(aabaa) = ? \]
Additional Example

Pattern:

a a b a a b a b b
1 2 3 4 5 6 7 8 9

Failure function f:
f(aaba) = 2 (you can take 'aa' but no more: aab != baa)
Additional Example

Pattern:
\[ a \ a \ b \ a \ a \ b \ a \ b \ b \]
\[ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \]

Failure function \( f \):
\[ f(aabaab) = ? \]
Additional Example

Pattern:
```
a a b a a b a b b
1 2 3 4 5 6 7 8 9
```

Failure function $f$:

$f(aabaab) = 3$ (you can take 'aab')

$f(aabaaba) = ?$
Additional Example

Pattern:
```
a a b a a b a b b
1 2 3 4 5 6 7 8 9
```

Failure function $f$:
- $f(aabaab) = 3$ (you can take 'aab')
- $f(aabaaba) = 4$ (you can take 'aaba')
Additional Example

Pattern:
\[a \ a \ b \ a \ a \ b \ a \ b \ b \]
\[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \]

Failure function \( f \):
\[f(aabaab) = 3 \text{ ( you can take 'aab')}\]
\[f(aabaaba) = 4 \text{ ( you can take 'aaba')}\]
\[f(aabaabab) = ?\]
Additional Example

Pattern:
\[a \ a \ b \ a \ a \ b \ a \ b \ b\]
\[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9\]

Failure function \(f:\)
\[f(aabaab) = 3 \text{ (you can take 'aab')}\]
\[f(aabaaba) = 4 \text{ (you can take 'aaba')}\]
\[f(aabaabab) = 0 \text{ ('a' \neq 'b', 'aa' \neq 'ab', etc \& can't be = 5, 'aabaa' \neq 'aabab')}\]
Additional Example

Pattern:
a a b a a b a b b
1 2 3 4 5 6 7 8 9

Failure function f:
f(aabaab) = 3 (you can take 'aab')
f(aabaaba) = 4 (you can take 'aaba')
f(aabaabab) = 0
f(aabaababb) = ?
Additional Example

Pattern:
\[ a \ a \ b \ a \ a \ b \ a \ b \ b \]
\[ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \]

Failure function \( f \):
\[
\begin{align*}
f(aabaab) &= 3 \ (\text{you can take 'aab'}) \\
f(aabaaba) &= 4 \ (\text{you can take 'aaba'}) \\
f(aabaabab) &= 0 \\
f(aabaababb) &= 0
\end{align*}
\]
Additional Example

Pattern:
\[a \ a \ b \ a \ a \ b \ a \ b \ b\]
\[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9\]

Failure function \( f \):
\[0 \ 1 \ 0 \ 1 \ 2 \ 3 \ 4 \ 0 \ 0\]
Example

\[
\begin{array}{cccccccc}
& a & b & a & c & a & a & b & a & c & c & a & b & a & c & a & b & a & a & b & b & a & b & a & c & a & b & & \\
\end{array}
\]

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

\[
P[j] = \begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
ab & ab & ac & ab & ab & ab \\
P[j] & a & b & a & c & a & b \\
F(j) & 0 & 0 & 1 & 0 & 1 & 2
\end{array}
\]
Rabin-Karp

- Calculates a **hash value** for the pattern, and for each M-character subsequence of text.
- If hash values unequal, then calculate the hash value for next M-character sequence.
- If hash values equal, then do **Brute Force** comparison.
Rabin-Karp: Analysis

- For “good” hash functions, the hashed values of two different patterns will usually be distinct.

- Thus, average case $O(N)$, where $N$ is size of text.

- Worst case complexity $O(MN)$ but rare for good hash functions.