Pattern Matching

Strings

- A string is a sequence of characters
- Examples of strings:
  - Python 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:
  - $T = \text{aaa ... ah}$
  - $P = \text{aaah}$
  - may occur in images and DNA sequences
  - unlikely in English text

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

**Input**
- text $T$ of size $n$
- 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 $j < m$ ∧ $T[i + j] = P[j]$

$j \leftarrow j + 1$

if $j = m$

return $i$ {match at $i$}

else

break while loop {mismatch}

return $-1$ {no match anywhere}

---

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 3 5 11 10 9 8 7
```

© 2014 Goodrich, Tamassia, Goldwasser
**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:**
  - $\Sigma = \{a, b, c, d\}$
  - $P = abacab$
  - 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**

```plaintext
Algorithm BoyerMooreMatch(T, P, \Sigma)

1. $L \leftarrow$ lastOccurrenceFunction($P$, $\Sigma$)
2. $i \leftarrow m - 1$
3. $j \leftarrow m - 1$
4. repeat
   1. if $T[i] = P[j]$
      1. if $j = 0$
         1. return $i$ { match at $i$ }
      2. else
         1. $i \leftarrow i - 1$
         2. $j \leftarrow j - 1$
   2. else { character-jump }
      1. $l \leftarrow L[T[i]]$
      2. $i \leftarrow i + m - \min(j, 1 + l)$
      3. $j \leftarrow m - 1$
6. until $i > n - 1$
7. return $-1$ { no match }
```

---

**Case 1:** $j \leq 1 + l$

```
... a ... i ...
... b a ...
... j I ...
... m - j ...
... b a ...
```

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

```
... a ... i ...
... b ...
... j I ...
... m - (1 + l) ...
... a ... b ...
```

© 2014 Goodrich, Tamassia, Goldwasser  Pattern Matching
Example

```
   string S = a b a c a a b a d c a b a c a b a a b b
   pattern P = a b a c a
```

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 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 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 $\text{failureFunction}(P)$

```
F[0] ← 0
i ← 1
j ← 0
while $i < m$
    if $P[i] = P[j]$
        {we have matched $j + 1$ chars}
        $F[i] ← j + 1$
        $i ← i + 1$
        $j ← j + 1$
    else if $j > 0$
        {use failure function to shift $P$}
        $j ← F[j - 1]$
    else
        $F[i] ← 0$ { no match }
        $i ← i + 1$
```

Example

```
Array P:
a b a c a a b a c a b a a b b

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[j]</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>F(j)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
Java Implementation

```java
/** Returns the lowest index at which substring pattern begins in text (or else -1). */
public static int findKMP(char[] text, char[] pattern) {
    int n = text.length;
    int m = pattern.length;
    int fail[] = computeFailKMP(pattern); // computed by private utility
    int j = 0;
    int k = 0;
    while (j < n) {
        if (text[j] == pattern[k]) { // pattern[0..k] matched thus far
            j++; k++;
            if (k == m - 1) return j - m + 1; // match is complete
        } else if (k > 0) k = fail[k-1]; // otherwise, try to extend match
    }
    return -1; // reached end without match
}
```

---

Java Implementation, 2

```java
private static int[] computeFailKMP(char[] pattern) {
    int m = pattern.length;
    int fail[] = new int[m]; // by default, all overlaps are zero
    int j = 1;
    int k = 0;
    while (j < m) {
        if (pattern[j] == pattern[k]) { // k + 1 characters match thus far
            fail[j] = k + 1;
            j++; k++;
        } else if (k > 0) k = fail[k-1]; // k follows a matching prefix
    }
    return fail; // no match found starting at j
}
```