Strings (§ 11.1)

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 (§ 11.2.1)

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:
- $T = aaa \ldots ah$
- $P = aaah$

may occur in images and DNA sequences
unlikely in English text

Algorithm 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 $n - m$
  
  for $j \leftarrow 0$ to $m - 1$
    
    if $T[i + j] = P[j]$
      
      $j \leftarrow j + 1$
    
    else
      
      if $j = m$
        
        return $i$ (match at $i$)
      
      else
        
        break while loop (mismatch)

return $-1$ (no match anywhere)

Boyer-Moore Heuristics (§ 11.2.2)

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 f t c r n m a t c h i n g a l g o r i t h m
  1 2 3 4 5 6 7 8 9 10 11

  r i t h m
  1 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
```
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:

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

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

Algorithm $\text{BoyerMooreMatch}(T, P, \Sigma)$

1. $L \leftarrow \text{lastOccurrenceFunction}(P, \Sigma)$
2. $i \leftarrow m - 1$
3. $j \leftarrow m - 1$
4. 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 \}
     - $l \leftarrow L[T[i]]$
     - $i \leftarrow i + m - \min(j, 1 + l)$
     - $j \leftarrow m - 1$
5. until $i > n - 1$
6. return $-1$ \{ no match \}

Case 1: $j \leq 1 + l$

Case 2: $1 + l \leq j$

Example

```
  a b a c a a b a d c a b a c a b a a b b
  1
  a b a c a b
  4 3 2
  a b a c a b
  13 12 11 10 9 8
  a b a c a b
  5
  a b a c a b
  6
  a b a c a b
```

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 (§ 11.2.3)

- 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 $P[1..j]$

**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 $j \leftarrow F(j-1)$.

**Computing the Failure Function**

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** 

```
Algorithm KMPMatch(T, P)
F ← failureFunction(P)
i ← 0
j ← 0
while i < n
    if T[i] = P[j]
        if j = m - 1
            return i - j {match}
        else
            j ← i + 1
            j ← j + 1
    else if j > 0
        j ← F[j - 1]
    else
        i ← i + 1
    return -1 {no match}
```

**KMP Failure Function**

```
Algorithm failureFunction(P)
F[0] ← 0
i ← 0
while i < m
    if P[i] = P[j]
        if j = m - 1
            F[i] ← i - j
        else
            F[i] ← F[i - 1]
    else if j > 0
        F[i] ← F[j - 1]
    else if F[i] = -1
        F[i] ← 0
    i ← i + 1

F[4] = 3
```

When a mismatch occurs, what is the shift amount?

- $i$ 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.
Example

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
7 & a & b & a & c & a & b \\
8 & 9 & 10 & 11 & 12 \\
13 & a & b & a & c & a & b \\
\end{array}
\]

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