Strings

A string is a sequence of characters.

An alphabet $\Sigma$ is the set of possible characters in strings.

Examples of strings:
- C program
- HTML document
- DNA sequence
- Digitized image

Examples of alphabets:
- ASCII
- Unicode
- ($\{0,1\}$)
- ($\{A,C,G,T\}$)

Notation:
- $\text{length}(P)$ ... #characters in $P$
- $\lambda$ ... empty string ($\text{length}(\lambda) = 0$)
- $\Sigma^m$ ... set of all strings of length $m$ over alphabet $\Sigma$
- $\Sigma^*$ ... set of all strings over alphabet $\Sigma$

$\nu\omega$ denotes the concatenation of strings $\nu$ and $\omega$

Note: $\text{length}(\nu\omega) = \text{length}(\nu) + \text{length}(\omega)$

Exercise #1: Strings

The string $a/a$ of length 3 over the ASCII alphabet has

- how many prefixes?
- how many suffixes?
- how many substrings?

Note:
- 4 prefixes: "", "a", "a/", "a/a"
- 4 suffixes: "a/a", "a/", "a", ""
- 6 substrings: "", "a", "a/", "a/a", "/a", "a/a"

Note:
- "" means the same as $\lambda$ (empty string)

ASCII (American Standard Code for Information Interchange)

- Specifies mapping of 128 characters to integers 0..127
- The characters encoded include:
  - upper and lower case English letters: A-Z and a-z
  - digits: 0-9
  - common punctuation symbols
  - special non-printing characters: e.g. newline and space

Reminder:
In C a string is an array of chars containing ASCII codes

- these arrays have an extra element containing a 0
- the extra 0 can also be written '\0' (null character or null-terminator)
- convenient because don't have to track the length of the string

Because strings are so common, C provides convenient syntax:

```c
char str[] = "hello";  // same as char str[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

Note: `str[]` will have 6 elements
C provides a number of string manipulation functions via \#include <string.h>, e.g.

- `strlen()` // length of string
- `strncpy()` // copy one string to another
- `strcat()` // concatenate two strings
- `strstr()` // find substring inside string

Example:

```c
char *strncat(char *dest, char *src, int n)
```

appends string `src` to the end of `dest` overwriting the `\0` at the end of `dest` and adds terminating `\0` returns start of string `dest` will never add more than `n` characters (If `src` is less than `n` characters long, the remainder of `dest` is filled with `\0` characters. Otherwise, `dest` is not null-terminated.)

### Pattern Matching

**Pattern Matching**

**Pattern Matching**

Example (pattern checked backwards):

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>
```

- **Text** ... abacaab
- **Pattern** ... abacab

### Analysis of Brute-force Pattern Matching

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

Examples of worst case (forward checking):

- $T = \text{aaa...ah}$
- $P = \text{aaah}$
  - may occur in DNA sequences
  - unlikely in English text

### Boyer-Moore Algorithm

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

- **Looking-glass heuristic**: Compare $P$ with subsequence of $T$ moving backwards
- **Character-jump heuristic**: When a mismatch occurs at $T[i] = c$
  - if $P$ contains $c$ => shift $P$ so as to align the last occurrence of $c$ in $P$ with $T[i]$
  - otherwise => shift $P$ so as to align $P[0]$ with $T[i+1]$ (a.k.a. "big jump")

```
j=0
while j<m ∧ T[i+j]=P[j] do // check from left to right
    j=j+1
    if j=m then // test ith shift of pattern
        return i // entire pattern checked
    end if
end while
end for
return -1 // no match found
```

**Boyer-Moore Algorithm**

Brute-force pattern matching algorithm preproceses pattern $P$ and alphabet $\Sigma$ to build

- last-occurrence function $L$
  - $L$ maps $\Sigma$ to integers such that $L(c)$ is defined as
    - the largest index $i$ such that $P[i]=c$, or
    - -1 if no such index exists

Example: $\Sigma = \{a,b,c,d\}$, $P = \text{acab}$

```
c a b c d
```
L(c)

- L can be represented by an array indexed by the numeric codes of the characters
- L can be computed in \(O(m+s)\) time (\(m\) ... length of pattern, \(s\) ... size of \(\Sigma\))

... Boyer-Moore Algorithm

BoyerMooreMatch(T, P, \(\Sigma\)):

Input text T of length n, pattern P of length m, alphabet \(\Sigma\)
Output starting index of a substring of T equal to P
-1 if no such substring exists

L = lastOccurrenceFunction(P, \(\Sigma\)) \hspace{1cm} // start at end of pattern
i = m-1, j = m-1 \hspace{1cm} // start at end of pattern
repeat
  if T[i] = P[j] then
    if j = 0 then
      return i \hspace{1cm} // match found at i
    else
      i = i-1, j = j-1
    end if
  else
    i = i + m - min(j, 1 + L[T[i]]) \hspace{1cm} // character-jump
    j = m-1
  end if
until i ≤ n
return -1 \hspace{1cm} // no match

- Biggest jump (\(m\) characters ahead) occurs when \(L[T[i]] = -1\)

... Boyer-Moore Algorithm

Case 1: \(j ≤ 1 + L[c]\)

Case 2: \(1 + L[c] < j\)

Exercise #2: Boyer-Moore algorithm

For the alphabet \(\Sigma = \{a, b, c, d\}\)

1. compute last-occurrence function \(L\) for pattern \(P = abacab\)
2. trace Boyer-More on \(P\) and text \(T = abacababcdabacabaabb\)
   - how many comparisons are needed?

Analysis of Boyer-Moore algorithm:

- Runs in \(O(nm+s)\) time
- \(m\) ... length of pattern \(n\) ... length of text \(s\) ... size of alphabet
- Example of worst case:
  - \(T = \text{aaa} \_ \text{a}\)
  - \(P = \text{baaa}\)
- Worst case may occur in images and DNA sequences but unlikely in English texts
  \(\Rightarrow\) Boyer-Moore significantly faster than brute-force on English text

Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm ...

- compares the pattern to the text left-to-right
- but shifts the pattern more intelligently than the brute-force algorithm

Reminder:

- \(Q\) is a prefix of \(P\) ... \(P = Q\omega\), for some \(\omega \in \Sigma^*\)
- \(Q\) is a suffix of \(P\) ... \(P = \omega Q\), for some \(\omega \in \Sigma^*\)

... Knuth-Morris-Pratt Algorithm

When a mismatch occurs ...

- what is the most we can shift the pattern to avoid redundant comparisons?
• Answer: the largest \textit{prefix} of \(P[0..j]\) that is a \textit{suffix} of \(P[1..j]\)

... Knuth-Morris-Pratt Algorithm

KMP preprocesses the pattern to find matches of its prefixes with itself

• \textit{Failure function} \(F(j)\) defined as
  o the size of the largest prefix of \(P[0..j]\) that is also a suffix of \(P[1..j]\)
  o if mismatch occurs at \(P_j \Rightarrow \text{advance } j \text{ to } F[j-1]\)

Example: \(P = \text{abaaba}\)

| \(j\) | 0 | 1 | 2 | 3 | 4 | 5 |
| \(P_j\) | a | b | a | a | b | a |
| \(F(j)\) | 0 | 0 | 1 | 1 | 2 | 3 |

\[ j \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \]
\[ P_j \quad a \quad b \quad a \quad a \quad b \quad a \]
\[ F(j) \quad 0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 3 \]

... Knuth-Morris-Pratt Algorithm

Construction of the failure function is similar to the KMP algorithm itself:

\textit{failureFunction} \((P)\):

\begin{itemize}
  \item \textbf{Input} pattern \(P\) of length \(m\)
  \item \textbf{Output} failure function for \(P\)
\end{itemize}

\begin{itemize}
  \item \(F[0]=0\)
  \item \(i=1, j=0\)
  \item \textbf{while} \(i < m\) \textbf{do}
    \begin{itemize}
      \item \textbf{if} \(P[i]=P[j]\) \textbf{then}
        \begin{itemize}
          \item \(j=j+1\)
        \end{itemize}
      \item \textbf{else if} \(j > 0\) \textbf{then}
        \begin{itemize}
          \item \(F[i]=F[j-1]\)
        \end{itemize}
      \item \textbf{else}
        \begin{itemize}
          \item \(F[i]=0\)
        \end{itemize}
    \end{itemize}
  \item \textbf{end while}\n\end{itemize}

KMPMatch \((T, P)\):

\begin{itemize}
  \item \textbf{Input} text \(T\) of length \(n\), pattern \(P\) of length \(m\)
  \item \textbf{Output} starting index of a substring of \(T\) equal to \(P\)
    \begin{itemize}
      \item \(-1\) if no such substring exists
    \end{itemize}
\end{itemize}

\begin{itemize}
  \item \(P=\text{failureFunction}(P)\)
  \item \(i=0, j=0\)
  \item \textbf{while} \(i < n\) \textbf{do}
    \begin{itemize}
      \item \textbf{if} \(T[i]=P[j]\) \textbf{then}
        \begin{itemize}
          \item \(F[i]=F[j-1]\)
        \end{itemize}
      \item \textbf{else if} \(j > 0\) \textbf{then}
        \begin{itemize}
          \item \(i=i+1, j=j+1\)
        \end{itemize}
      \item \textbf{else}
        \begin{itemize}
          \item \(i=i+1\)
        \end{itemize}
    \end{itemize}
  \item \textbf{end while}\n\end{itemize}
Knuth-Morris-Pratt Algorithm

Analysis of failure function computation:
- 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

$\Rightarrow$ failure function can be computed in $O(m)$ time

Boyer-Moore vs KMP

Boyer-Moore algorithm
- decides how far to jump ahead based on the mismatched character in the text
- works best on large alphabets and natural language texts (e.g., English)

Knuth-Morris-Pratt algorithm
- uses information embodied in the pattern to determine where the next match could begin
- works best on small alphabets (e.g., A, C, G, T)

Tries

Tries are trees organised using parts of keys (rather than whole keys)

Preprocessing Strings

Preprocessing the pattern speeds up pattern matching queries
- After preprocessing $P$, KMP algorithm performs pattern matching in time proportional to the text length
If the text is large, immutable and searched for often (e.g., works by Shakespeare)
- we can preprocess the text instead of the pattern

A trie …
- is a compact data structure for representing a set of strings

Depth $d$ of trie = length of longest key value
Cost of searching $O(d)$ (independent of $n$)

Tries

Possible trie representation:
```c
#define ALPHABET_SIZE 26

typedef struct Node *Trie;

typedef struct Node {
    bool finish; // last char in key?
    Item data; // no Item if !finish
    Trie child[ALPHABET_SIZE];
} Node;

typedef char *Key;
```

Note: Can also use BST-like nodes for more space-efficient implementation of tries
Basic operations on tries:

1. search for a key
2. insert a key

Traversing a path, using char-by-char from Key:

```plaintext
find(trie, key):
    Input  trie, key
    Output pointer to element in trie if key found
         NULL otherwise

    node=trie
    for each char in key do
        if node.child[char] exists then
            node=node.child[char]  // move down one level
        else
            return NULL
        end if
    end for
    if node.finish then  // “finishing” node reached?
        return node
    else
        return NULL
    end if
```

Insertion into Trie:

```plaintext
insert(trie, item, key):
    Input trie, item with key of length m
    Output trie with item inserted

    if trie is empty then
        t=new trie node
    end if
    if m=0 then
        t.finish=true, t.data=item
    else
        t.child[key[0]]=insert(trie, item, key[1..m-1])
    end if
    return t
```

Analysis of standard tries:

- \(O(n)\) space
- \(O(d \cdot m)\) time for insertion and search in text

\(n\) ... total size of text (e.g. sum of lengths of all strings in a given dictionary)

\(m\) ... size of the string parameter of the operation (the “key”) (e.g. 26)

\(d\) ... size of the underlying alphabet (e.g. 26)

Word Matching With Tries

Preprocessing the text:

1. Insert all searchable words of a text into a trie
2. Each leaf stores the occurrence(s) of the associated word in the text

Example text and corresponding trie of searchable words:
Compressed Tries

Compressed tries …

- have internal nodes of degree ≥ 2
- are obtained from standard tries by compressing "redundant" chains of nodes

Example:

Possible compact representation of a compressed trie to encode an array $S$ of strings:

- nodes store ranges of indices instead of substrings
- use triple $(i,j,k)$ to represent substring $S[i][j..k]$
- requires $O(s)$ space ($s$ = #strings in array $S$)

Example:

Pattern Matching With Suffix Tries

The suffix trie of a text $T$ is the compressed trie of all the suffixes of $T$

Example:

Compact representation:

Input:

compact suffix trie for text $T$

pattern $P$

Goal:

find starting index of a substring of $T$ equal to $P$

Example:

```
suffixTrieMatch(trie, P):
    Input compact suffix trie for text $T$, pattern $P$ of length $m$
    Output starting index of a substring of $T$ equal to $P
    -1 if no such substring exists
    j=0, v=root of trie
    repeat
      // we have matched $j+1$ characters
```
if \( \exists w \in \text{children}(v) \) such that \( P[j] = T[\text{start}(w)] \) then

\[
\begin{align*}
  i &= \text{start}(w) & // \text{start}(w) \text{ is the start index of } w \\
  x &= \text{end}(w) - i + 1 & // \text{end}(w) \text{ is the end index of } w \\
  \text{if } m \leq x & & // \text{length of suffix } \leq \text{length of the node label} \\
    \text{if } P[j..j+m-1] = T[i..i+m-1] & & \text{then} \\
      \text{return } i - j & & // \text{match at } i - j \\
    \text{else} & & \text{return } -1 & & // \text{no match} \\
  \text{else} & & \text{return } -1 & & \text{if } P[j..j+x-1] = T[i..i+x-1] \text{ then} \\
    \text{if } \text{match} & & \text{update suffix start index and length} \\
      v &= w & & // \text{move down one level} \\
    \text{else} & & \text{return } -1 & & // \text{no match} \\
  \text{else} & & \text{return } -1 & & \text{end if} \\
\end{align*}
\]

until \( v \) is leaf node

\text{return } -1 & // \text{no match}

... Pattern Matching With Suffix Tries

Analysis of pattern matching using suffix tries:

Suffix trie for a text of size \( n \) ...

- can be constructed in \( O(n) \) time
- uses \( O(n) \) space
- supports pattern matching queries in \( O(s \cdot m) \) time
  - \( m \) ... length of the pattern
  - \( s \) ... size of the alphabet

Text Compression

Problem: Efficiently encode a given string \( X \) by a smaller string \( Y \)

Applications:
- Save memory and/or bandwidth

Huffman's algorithm
- computes frequency \( f(c) \) for each character \( c \)
- encodes high-frequency characters with short code
- no code word is a prefix of another code word
- uses optimal encoding tree to determine the code words

Example: \( T = \text{abracadabra} \)

Huffman's algorithm
- computes frequency \( f(c) \) for each character
- successively combines pairs of lowest-frequency characters to build encoding tree "bottom-up"

Example: \( \text{abracadabra} \)
Huffman Code

Huffman's algorithm using priority queue:

\[
\text{HuffmanCode}(T): \\
\quad \text{Input: string } T \text{ of size } n \\
\quad \text{Output: optimal encoding tree for } T \\
\end{align*}
\]

- compute frequency array
- \(Q=\text{new priority queue}\)
  - for all characters \(c\) do
    - \(T=\text{new single-node tree storing } c\)
    - \(\text{join}(Q,T) \text{ with frequency}(c) \text{ as key}\)
  - end for
- while \(|Q|>2\) do
  - \(f_1=Q.\text{minKey}(), T_1=\text{leave}(Q)\)
  - \(f_2=Q.\text{minKey}(), T_2=\text{leave}(Q)\)
  - \(T=\text{new tree node with subtrees } T_1 \text{ and } T_2\)
  - \(\text{join}(Q,T) \text{ with } f_1+f_2 \text{ as key}\)
- end while
- return \(\text{leave}(Q)\)

Analysis of Huffman's algorithm:
- \(O(n+d \cdot \log d)\) time
  - \(n\) … length of the input text \(T\)
  - \(d\) … number of distinct characters in \(T\)

Summary

- Alphabets and words
- Pattern matching
  - Boyer-Moore, Knuth-Morris-Pratt
- Tries
- Text compression
  - Huffman code

Suggested reading:
- Tries … Sedgewick, Ch.15.2

Produced: 17 Oct 2017