0. Introduction

Strings and sequences are one of the simplest, most natural and most used forms of storing information.

- natural language, biosequences, programming source code, XML, music, any data stored in a file
- Many algorithmic techniques are first developed for strings and later generalized for more complex types of data such as graphs.

The area of algorithm research focusing on strings is sometimes known as stringology. Characteristic features include

- Huge data sets (document databases, biosequence databases, web crawls, etc.) require efficiency. Linear time and space complexity is the norm.
- Strings come with no explicit structure, but many algorithms discover implicit structures that they can utilize.

Strings

An alphabet is the set of symbols or characters that may occur in a string. We will usually denote an alphabet with the symbol Σ and its size with σ.

We consider three types of alphabets:

- **General alphabet**: An arbitrary set Σ = {c₁, c₂, ..., cᵣ}. We assume that the alphabet is ordered: c₁ < c₂ < ... < cᵣ.
- **Integer alphabet**: Σ = {0, 1, 2, ..., σ − 1}.
- **Constant alphabet**: A general alphabet for a (small) constant σ.

**Example 0.1**: Suppose we want to count how many distinct characters occur in a given string. We can do this by maintaining a set of distinct characters. Each character of the string is added to the set if not already there. The type of alphabet determines the choice of the set data structure:

- For general alphabet, we can use a balanced binary search tree.
- For integer alphabet, we can use an array of size σ, which is much faster.
- If σ is large, an array of size σ might be too big. We could use a hash table instead.
- For constant alphabet, we could even use an unordered list without affecting the time complexity.

An alternative solution is to sort the characters, which groups identical characters together. The choice of the sorting algorithm depends on the alphabet.

---

**Contents**

0. Introduction

1. Sets of strings
   - Search trees, string sorting, binary search

2. Exact string matching
   - Finding a pattern (string) in a text (string)

3. Approximate string matching
   - Finding in the text something that is similar to the pattern

4. Suffix tree and suffix array
   - Preprocess a long text for fast string matching and all kinds of other tasks

---

**About this course**

On this course we will cover a few cornerstone problems in stringology. We will describe several algorithms for the same problems:

- the best algorithms in theory and/or in practice
- algorithms using a variety of different techniques

The goal is to learn a toolbox of basic algorithms and techniques.

On the lectures, we will focus on the clean, basic problem. Exercises may include some variations and extensions. We will mostly ignore any application specific issues.
There are many notations for strings.
When describing algorithms, we will typically use the array notation to emphasize that the string is stored in an array:
Note the half-open range notation \([0..n)\) which is often convenient.
In an abstract context, we often use other notations, for example:
- \( \alpha, \beta \in \Sigma^* \)
- \( x = a_1a_2 \ldots a_n \) where \( a_i \in \Sigma \) for all \( i \)
- \( w = w_v, u, v \in \Sigma^* \) (\( w \) is the concatenation of \( u \) and \( v \))
We will use \(|w|\) to denote the length of a string \( w \).

Some Interesting Strings

The Fibonacci strings are defined by the recurrence:
\[
F_0 = \varepsilon
F_1 = a
F_{i+2} = F_iF_{i+1}
\]
\( \sigma \)

- For all \( i > 1 \), we can obtain \( F_i \) from \( F_{i-1} \) by applying the substitutions
  \( a \mapsto ab \) and \( b \mapsto a \) to every character.

A De Bruijn sequence \( B_k \) of order \( k \) for an alphabet \( \Sigma \) of size \( \sigma \) is a cyclic string of length \( \sigma^k \) that contains every string of length \( k \) over the alphabet \( \Sigma \) as a factor exactly once. The cycle can be opened into a string of length \( \sigma^k + 1 \) with the same property.

Example 0.4: Let \( w = ab \). Then
- \( a, b, ab, aba, abab, ababa \) are the prefixes of \( w \)
- \( ab, aba, abab \) are the suffixes of \( w \)
- \( a, b \) are the borders of \( w \)
- \( \varepsilon, a, b, ab, aba, abab, ababa \) are the factors of \( w \).

Note that \( a \) and \( w \) are always suffixes, prefixes, and borders of \( w \).
A suffix/prefix/border of \( w \) is proper if it is not \( w \), and nontrivial if it is not \( \varepsilon \) or \( w \).

1. Sets of Strings

Basic operations on a set of objects include:
- Insert: Add an object to the set
- Delete: Remove an object from the set.
- Lookup: Find if a given object is in the set, and if it is, possibly return some data associated with the object.

There can also be more complex queries:
- Range query: Find all objects in a given range of values.

There are many other operations too but we will concentrate on these here.

The above time complexities assume that basic operations on the objects including comparisons can be performed in constant time. When the objects are strings, this is no more true:

- The worst case time for a string comparison is the length of the shorter string. Even the average case time for a random set of \( n \) strings is \( \Omega(\log n) \) in many cases, including for basic operations in a balanced binary search tree. We will show an even stronger result for sorting later. And sets of strings are rarely fully random.
- Computing a hash function is slower too. A good hash function depends on all characters and cannot be computed faster than the length of the string.

For a string set \( R \), there are also new types of queries:
- Prefix query: Find all strings in \( R \) that have the query string \( S \) as a prefix. This is a special type of range query.
- Lcp (longest common prefix) query: What is the length of the longest prefix of the query string \( S \) that is also a prefix of some string in \( R \).

Thus we need special set data structures and algorithms for strings.

1. Trie

A simple but powerful data structure for a set of strings is the trie. It is a rooted tree with the following properties:

- Edges are labelled with symbols from an alphabet \( \Sigma \).
- For every node \( v \), the edges from \( v \) to its children have different labels.

Each node represents the string obtained by concatenating the symbols on the path from the root to that node.

The trie for a string set \( R \), denoted by \( \text{trie}(R) \), is the smallest trie that has nodes representing all the strings in \( R \). The nodes representing strings in \( R \) may be marked.

Example 1.1: \( \text{trie}(R) \) for \( R = \{ \text{ali}, \text{alice}, \text{anna}, \text{elias}, \text{aliza} \} \).
The trie is conceptually simple but it is not simple to implement efficiently. The time and space complexity of a trie depends on the implementation of the child function:

For a node $v$ and a symbol $c \in \Sigma$, $\text{child}(v, c)$ is $u$ if $u$ is a child of $v$ and the edge $(v, u)$ is labelled with $c$, and $\text{child}(v, c) = \perp$ (null) if $v$ has no such child.

As an example, here is the insertion algorithm:

**Algorithm 1.2:** Insertion into trie

**Input:** $\text{trie}(R)$ and a string $S[0..m] \not\in R$

**Output:** $\text{trie}(R \cup \{S\})$

(1) $v \leftarrow \text{root}; j \leftarrow 0$

(2) while $\text{child}(v, S[j]) \neq \perp$ do

(3) $v \leftarrow \text{child}(v, S[j]); j \leftarrow j + 1$

(4) while $j < m$ do

(5) Create new node $u$ (initializes $\text{child}(u, c)$ to $\perp$ for all $c \in \Sigma$)

(6) $\text{child}(v, S[j]) \leftarrow u$

(7) $v \leftarrow u; j \leftarrow j + 1$

(8) Mark $v$ as representative of $S$

Prefix free string sets

Many data structures and algorithms for a string set $R$ become simpler if $R$ is prefix free.

**Definition 1.3:** A string set $R$ is prefix free if no string in $R$ is a prefix of another string in $R$.

There is a simple way to make any string set prefix free:

- Let $\$ \not\in \Sigma$ be an extra symbol satisfying $\$ < $c$ for all $c \in \Sigma$.
- Append $\$ to the end of every string in $R$.

This has little or no effect on most operations on the set. The length of each string increases by one only, and the additional symbol could be there only virtually.

**Example 1.4:** The set $\{ali$, alice, anna, elias, eliza$\}$ is not prefix free because $ali$ is a prefix of alice, but $\{ali\$, alice$, anna$, elias$, eliza$\}$ is prefix free.

Compact Trie

Tries suffer from a large number of nodes, close to $|R|$ in the worst case.

- For a string set $R$, we use $|R|$ to denote the number of strings in $R$ and $||R||$ to denote the total length of the strings in $R$.

The space requirement can be problematic, since typically each node needs much more space than a single symbol.

Compact tries reduce the number of nodes by replacing branchless path segments with a single edge.

**Example 1.6:** Compact trie for $\{ali$, alice$, anna$, elias$, eliza$\}$.

There is also an intermediate form of trie called leaf-path-compacted trie, where branchless path segments are compacted only when they end in a leaf.

- Typically (though not in the worst case) this achieves most of the advantages of a compact trie.
- For trie algorithms, this means stopping the normal search, when only one string is remaining in the subtree.

**Example 1.7:** Leaf-path-compacted trie for $\{ali$, alice$, anna$, elias$, eliza$\}$.
Example 1.8: Ternary tries for \{ali$, alice$, anna$, elias$, eliza$\}.

Ternary tries have the same asymptotic size as the corresponding (\(\sigma\)-ary) tries.

A ternary trie is balanced if each left and right subtree contains at most half of the strings in its parent tree.

- The balance can be maintained by rotations similarly to binary search trees.

- We can also get reasonably close to a balance by inserting the strings in the tree in a random order.

Note that there is no restriction on the size of the middle subtree.

In a balanced ternary trie each step down either
- moves the position forward (middle branch), or
- halves the number of strings remaining in the subtree (side branch).

Thus, in a balanced ternary trie storing \(n\) strings, any downward traversal following a string \(S\) passes at most \(|S|\) middle edges and at most \(\log n\) side edges.

Thus the time complexity of insertion, deletion, lookup and lcp query is \(O(|S| + \log n)\).

In comparison based tries, where the child function is implemented using binary search trees, the time complexities could be \(O(|S| \log \sigma)\), a multiplicative factor \(O(\log \sigma)\) instead of an additive factor \(O(\log n)\).

Prefix and range queries behave similarly (exercise).