

# String Matching

# String Matching

What's the problem?

Given a string  $P$  (*the pattern*) and a longer string  $T$  (*the text*).  
Find all occurrences of pattern  $P$  in text  $T$ .

# String Matching

## What's the problem?

Given a string  $P$  (*the pattern*) and a longer string  $T$  (*the text*).  
Find all occurrences of pattern  $P$  in text  $T$ .

## Example

- $P = aba$  and  $T = bbabaxababay$ .
- $P$  occurs in  $T$  at positions 3, 7, and 9.
- Note: Occurrences may overlap!

# String Matching

## Terminology Confusion

- Substring: *aba* in *bb**ab**axababay*.
- Subsequence: *aba* in *bb**ab**x**a**ababay*.

# String Matching

## Terminology Confusion

- Substring: *aba* in *bb**ab**axababay*.
- Subsequence: *aba* in *bb**ab**x**a**ababay*.

## Literature

Dan Gusfield, *Algorithms on Strings, Trees, and Sequences*,  
New York, 1997

# String Matching Algorithms

## Classical comparison-based methods

- Naive Approach
- Boyer-Moore
- Knuth-Morris-Pratt (KMP)
- Rabin-Karp

Improvements result from preprocessing of the given pattern  $P$ .

## Methods based on special data structures

- Data Structures: Suffix-Tries, Suffix-Trees, Suffix-Arrays.
- Improvements result from preprocessing of the given text  $T$ .
- Good for finding many different patterns in the text.

# Naive Approach

## Naive Approach

```
n = text.size();
m = pattern.size();

for s = 0 to n - m {
    if (pattern[1 .. m] = text[s+1 .. s+m]) {
        add_result(s);
    }
}
```

For  $T = a^n$ ,  $P = a^m$  and  $m = n/2$  the worst case occurs, yielding a running time of  $\Theta(n^2)$ .

## Key Ideas

- Right-to-left scan.
  - Bad character rule.
  - The good suffix rule.
- 
- Possible that some text characters are never compared.
  - Preprocessing stage needed for efficient use of the bad character rule and good suffix rule.
  - In practice one of the best known algorithms for string matching.
  - Good explanation in *Dan Gusfield, Algorithms on strings trees and sequences*.



## Bad Character Rule

T: prstaqst**u**abvqxrst

|

P: qcabdae**d**ab

## Bad Character Rule

T: prstaqst**u**abvqxrst

|

P: qcabdaedab

## Bad Character Rule

T: prstaqst**u**babvqxrst

|

P:       qcab**d**aedab

## The Good Suffix Rule

T: prstabst**u**abvqxrst

|

P: qcabdab**d**ab

## The Good Suffix Rule

T: prstabst**u**abvqxrst

|

P: qcabdab

## The Good Suffix Rule

T: prstabst**u**abvqxrst

|

P: qcab**d**ab**d**ab

## The Good Suffix Rule

T: prstabst**u**abvqxrst

|

P: qcabdab**d**ab

## The Good Suffix Rule

T: prstabst**u**abvqxrst

|

P:           q**c**abdabdab

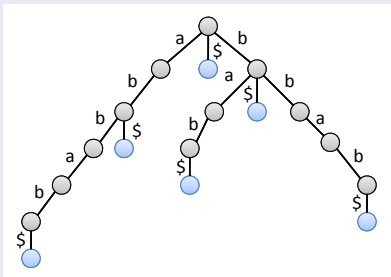


## General

- Representation of one (or many) string(s) in a tree-structure.
- Usually used for preprocessing the text not the pattern.
- Searching fast for any substring.

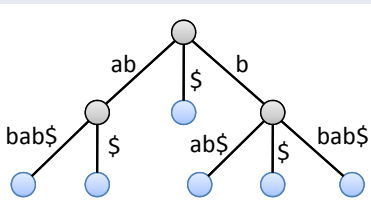
## Suffix-Tries (from retrieval)

- Each leaf corresponds to a suffix of some string (contains the starting-position).
- Each edge corresponds to a character.
- Example trie for  $T = abbab$ :



## Suffix-Trees

- Trie→Tree: One edge for paths without branches.
- There are efficient algorithms for constructing suffix trees, e.g. Ukkonen.



# Suffix-Arrays

## Suffix-Arrays

- Array of length  $|T| = n$  listing the suffixes of  $T$  in ascending order.
- Search for each  $P$  ( $|P| = m$ ) in  $m \log n$  time.
- Simple construction in  $\mathcal{O}(n^2 \log n)$ .

## Example for $T = abbab$

ab

abbab

b

bab

bbab