

# String processing

Robin Jadoul



30 januari 2016

# Table of Contents

Ad hoc

Tries

String matching

- Naive

- Rabin-Karp

- Z-algorithm

- Knuth-Morris-Pratt

Exercises

# Ad hoc

- ▶ Straightforward solution

# Ad hoc

- ▶ Straightforward solution
- ▶ See CP3, section 6.3 (pages 236 - 240)

# Ad hoc

- ▶ Straightforward solution
- ▶ See CP3, section 6.3 (pages 236 - 240)
- ▶ If you know regular expressions, C++ 11 has those as well

# Table of Contents

Ad hoc

Tries

String matching

Naive

Rabin-Karp

Z-algorithm

Knuth-Morris-Pratt

Exercises

# Trie

## Properties

- ▶ <Retrieval (but can be pronounce as either *tree* or *try*)

# Trie

## Properties

- ▶ <Retrieval (but can be pronounce as either *tree* or *try*)
- ▶ Store a set of words (with or without associated values)



# Trie

## Properties

- ▶ <Retrieval (but can be pronounce as either *tree* or *try*)
- ▶ Store a set of words (with or without associated values)
- ▶ insert/retrieve in  $O(S)$ , with  $S$  the length of the string

# Trie

## Properties

- ▶  $\leq$  Retrieval (but can be pronounce as either *tree* or *try*)
- ▶ Store a set of words (with or without associated values)
- ▶ insert/retrieve in  $O(S)$ , with  $S$  the length of the string
- ▶ Allows for non-exact matches ( $\leq$  set/map)

# Trie

## Structure

- ▶ Tree structure

# Trie

## Structure

- ▶ Tree structure
- ▶ Stores the *path* for the string instead of the string

# Trie

## Structure

- ▶ Tree structure
- ▶ Stores the *path* for the string instead of the string
- ▶ Edges labeled with single characters

# Trie

## Structure

- ▶ Tree structure
- ▶ Stores the *path* for the string instead of the string
- ▶ Edges labeled with single characters
- ▶ If the last character of a stored word, marked (+ associated value)

# Trie

## Structure

- ▶ Tree structure
- ▶ Stores the *path* for the string instead of the string
- ▶ Edges labeled with single characters
- ▶ If the last character of a stored word, marked (+ associated value)
- ▶ Can vary in type of *character* (bits/ints/...)

# Trie

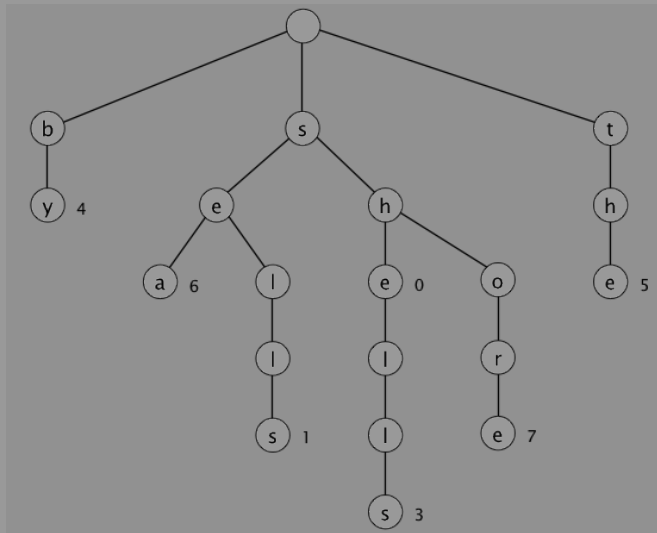
## Structure

- ▶ Tree structure
- ▶ Stores the *path* for the string instead of the string
- ▶ Edges labeled with single characters
- ▶ If the last character of a stored word, marked (+ associated value)
- ▶ Can vary in type of *character* (bits/ints/...)
- ▶ Can be compressed by eliminating successive single-edge nodes



# Trie

## Structure



# Trie

## Usage

- ▶ Spelling suggestions

# Trie

## Usage

- ▶ Spelling suggestions
- ▶ Autocompletion

# Trie

## Usage

- ▶ Spelling suggestions
- ▶ Autocompletion
- ▶ Bioinformatics (DNA/RNA)

# Trie

## Usage

- ▶ Spelling suggestions
- ▶ Autocompletion
- ▶ Bioinformatics (DNA/RNA)
- ▶ Alphabetical ordering / sorting

# Trie

## Usage

- ▶ Spelling suggestions
- ▶ Autocompletion
- ▶ Bioinformatics (DNA/RNA)
- ▶ Alphabetical ordering / sorting
- ▶ (Similar to structure for Aho-Corasick)

# Trie

## Usage

- ▶ Spelling suggestions
- ▶ Autocompletion
- ▶ Bioinformatics (DNA/RNA)
- ▶ Alphabetical ordering / sorting
- ▶ (Similar to structure for Aho-Corasick)
- ▶ (Basis for *suffix tree*)

# Trie

## Code

```
#include <map>
using namespace std;

struct Trie
{
    //Can be map/unordered_map/direct addressing table/implicit edge/...
    map<char, Trie*> children;
    bool marked;
};
```



# Trie

## Code

```
void insert(Trie* t, string s)
{
    for (int index = 0; index < s.length(); index++)
    {
        if (t->children.find(s[index]) == t->children.end())
        {
            t->children[s[index]] = new Trie();
        }
        t = t->children[s[index]];
    }
    t->marked = true;
}
```

# Trie

## Code

```
bool contains(Trie* t, string s)
{
    for (auto c : s)
    {
        if (t->children.find(c) == t->children.end())
            return false;
        t = t->children[c];
    }
    return t->marked;
}
```

# Table of Contents

Ad hoc

Tries

String matching

- Naive

- Rabin-Karp

- Z-algorithm

- Knuth-Morris-Pratt

Exercises

# Table of Contents

Ad hoc

Tries

String matching

**Naive**

Rabin-Karp

Z-algorithm

Knuth-Morris-Pratt

Exercises

# Naive matching

principle

- ▶ Straightforward

# Naive matching

## principle

- ▶ Straightforward
- ▶ Check for match at each index

# Naive matching

## principle

- ▶ Straightforward
- ▶ Check for match at each index
- ▶ Usually, use the one in the standard library, don't write your own

# Naive matching

## principle

- ▶ Straightforward
- ▶ Check for match at each index
- ▶ Usually, use the one in the standard library, don't write your own
- ▶  $O(s * p)$  ( $s$  = length of string,  $p$  = length of pattern)



# Naive matching

## Code

```
#include <string>
using namespace std;

int match(string s, string pat)
{
    if (s.length() < pat.length())
        return -1;
    for (int i = 0; i <= s.length() - pat.length(); i++)
    {
        bool found = true;
        for (int j = 0; j < pat.length(); j++)
        {
            if (s[i+j] != pat[j])
            {
                found = false;
                break;
            }
        }
        if (found) return i;
    }
    return -1;
}
```

# Table of Contents

Ad hoc

Tries

String matching

Naive

**Rabin-Karp**

Z-algorithm

Knuth-Morris-Pratt

Exercises

# Rabin-Karp

## Idea

- ▶ Checking for a match with the pattern:  $O(n)$

# Rabin-Karp

## Idea

- ▶ Checking for a match with the pattern:  $O(n)$
- ▶ Faster possible?

# Rabin-Karp

## Idea

- ▶ Checking for a match with the pattern:  $O(n)$
- ▶ Faster possible?
- ▶ What about hashes, integer comparison =  $O(n)$

# Rabin-Karp

## Idea

- ▶ Checking for a match with the pattern:  $O(n)$
- ▶ Faster possible?
- ▶ What about hashes, integer comparison =  $O(1)$
- ▶ We still need a  $O(1)$  way to generate the hashes.

# Rabin-Karp

## Idea

- ▶ Checking for a match with the pattern:  $O(n)$
- ▶ Faster possible?
- ▶ What about hashes, integer comparison =  $O(n)$
- ▶ We still need a  $O(1)$  way to generate the hashes.
- ▶ Useful for multiple same-length patterns (check all hashes)

# Rabin-Karp

## Polynomial hashing

- ▶ Generate successive hashes of the same length as the pattern (and hash the pattern)



# Rabin-Karp

## Polynomial hashing

- ▶ Generate successive hashes of the same length as the pattern (and hash the pattern)
- ▶ Polynomial hashing: the string is an integer in some base  $B$  (usually prime)

# Rabin-Karp

## Polynomial hashing

- ▶ Generate successive hashes of the same length as the pattern (and hash the pattern)
- ▶ Polynomial hashing: the string is an integer in some base  $B$  (usually prime)
- ▶  $s_i, s_{i+1}, \dots, s_{i+k-1} = s_i \times B^{k-1} + s_{i+1} \times B^{k-2} + \dots + s_{i+k-1} \times B^0$

# Rabin-Karp

## Polynomial hashing

- ▶ Generate successive hashes of the same length as the pattern (and hash the pattern)
- ▶ Polynomial hashing: the string is an integer in some base  $B$  (usually prime)
- ▶  $s_i, s_{i+1}, \dots, s_{i+k-1} = s_i \times B^{k-1} + s_{i+1} \times B^{k-2} + \dots + s_{i+k-1} \times B^0$
- ▶ Too big  $\Rightarrow$  modulo  $H$  (usually prime)

# Rabin-Karp

## Polynomial hashing

- ▶ Generate successive hashes of the same length as the pattern (and hash the pattern)
- ▶ Polynomial hashing: the string is an integer in some base  $B$  (usually prime)
- ▶  $s_i, s_{i+1}, \dots, s_{i+k-1} = s_i \times B^{k-1} + s_{i+1} \times B^{k-2} + \dots + s_{i+k-1} \times B^0$
- ▶ Too big  $\Rightarrow$  modulo  $H$  (usually prime)
- ▶ Watch out for false positives

# Rabin-Karp

## Rolling hashes

- ▶ Once we have a hash, it's easy to compute the next

# Rabin-Karp

## Rolling hashes

- ▶ Once we have a hash, it's easy to compute the next
- ▶  $s_{i+1}, s_{i+2}, \dots, s_{i+k} = ((s_i, \dots, s_{i+k-1}) - s_i \times B^{k-1}) \times B + s_{i+k}$

# Rabin-Karp

## Rolling hashes

- ▶ Once we have a hash, it's easy to compute the next
- ▶  $s_{i+1}, s_{i+2}, \dots, s_{i+k} = ((s_i, \dots, s_{i+k-1}) - s_i \times B^{k-1}) \times B + s_{i+k}$
- ▶ A rolling hash *frame*

# Rabin-Karp

## Rolling hashes

- ▶ Once we have a hash, it's easy to compute the next
- ▶  $s_{i+1}, s_{i+2}, \dots, s_{i+k} = ((s_i, \dots, s_{i+k-1}) - s_i \times B^{k-1}) \times B + s_{i+k}$
- ▶ A rolling hash *frame*
- ▶  $O(1)$



# Rabin-Karp

## Collision strategies

- ▶ If equal hashes  $\Rightarrow$  compare the strings explicitly

# Rabin-Karp

## Collision strategies

- ▶ If equal hashes  $\Rightarrow$  compare the strings explicitly
- ▶ Worst case, still  $O(n^2)$

# Rabin-Karp

## Collision strategies

- ▶ If equal hashes  $\Rightarrow$  compare the strings explicitly
- ▶ Worst case, still  $O(n^2)$
- ▶ *Gambling*: keep 2 hashes (with distinct  $B$  and  $H$ )

# Rabin-Karp

## Collision strategies

- ▶ If equal hashes  $\Rightarrow$  compare the strings explicitly
- ▶ Worst case, still  $O(n^2)$
- ▶ *Gambling*: keep 2 hashes (with distinct  $B$  and  $H$ )
- ▶ Collision chance is low, if the two hashes match, guess it's correct

# Rabin-Karp

## Collision strategies

- ▶ If equal hashes  $\Rightarrow$  compare the strings explicitly
- ▶ Worst case, still  $O(n^2)$
- ▶ *Gambling*: keep 2 hashes (with distinct  $B$  and  $H$ )
- ▶ Collision chance is low, if the two hashes match, guess it's correct
- ▶  $\Rightarrow$  triple hashing, ...

# Rabin-Karp

code

```
const int B = 17;
const int H = 12632251;

int hash_pattern(string pat, int start, int end)
{
    int h = 0;
    for (int i = start; i <= end; i++)
    {
        h = ((h * B) % H + pat[i]) % H;
    }
    return h;
}
```

# Rabin-Karp

code

```
bool check(string s, string pat, int start)
{
    for (int i = 0; i < pat.length(); i++)
    {
        if (s[start + i] != pat[i])
            return false;
    }
    return true;
}

int modpow(int exp) { //This can be done in O(log N)
    int result = 1;
    for (int i = 0; i < exp; i++)
    {
        result = (result * B) % H;
    }
    return result;
}
```

# Rabin-Karp

code

```
int match(string s, string pat)
{
    if (pat.length() > s.length()) return -1;
    int k = pat.length();
    int Hp = hash_pattern(pat, 0, k - 1);
    int Hs = hash_pattern(s, 0, k - 1);
    int Bk = modpow(k-1);
    for (int i = 0; i <= s.length() - k; i++)
    {
        if (Hs == Hp && check(s, pat, i))
        {
            return i;
        }
        Hs = ((B * (Hs - (s[i] * Bk) % H)) % H + s[i+k]) % H;
        if (Hs < 0) Hs += H;
    }
    return -1;
}
```



# Table of Contents

Ad hoc

Tries

String matching

Naive

Rabin-Karp

**Z-algorithm**

Knuth-Morris-Pratt

Exercises

# Z-algorithm

## terminology

- ▶ Z-box = substring that matches with a prefix from the string

# Z-algorithm

## terminology

- ▶ Z-box = substring that matches with a prefix from the string
- ▶ Z-score  $Z_i(S)$  = length of Z-box starting at index  $i$

# Z-algorithm

## terminology

- ▶ Z-box = substring that matches with a prefix from the string
- ▶ Z-score  $Z_i(S)$  = length of Z-box starting at index  $i$

A A B A A A B



# Z-algorithm

## terminology

- ▶ Z-box = substring that matches with a prefix from the string
- ▶ Z-score  $Z_i(S)$  = length of Z-box starting at index  $i$

A A B A A A B

letter	A	A	B	A	A	A	B
Z-score	7	1	0	2	3	1	0

# Z-algorithm

## Matching

- ▶  $P$  = pattern

# Z-algorithm

## Matching

- ▶  $P$  = pattern
- ▶  $S$  = search string

# Z-algorithm

## Matching

- ▶  $P$  = pattern
- ▶  $S$  = search string
- ▶  $\$$  = sentinel (not part of alphabet)



# Z-algorithm

## Matching

- ▶  $P$  = pattern
- ▶  $S$  = search string
- ▶  $\$$  = sentinel (not part of alphabet)
- ▶ return  $i$  for each  $i > 0$  where  $Z_i(P\$S) = |P|$

# Z-algorithm

Calculating Z-scores

- ▶ Naive  $\Rightarrow O(n^2)$ , possible in  $O(n)$

# Z-algorithm

## Calculating Z-scores

- ▶ Naive  $\Rightarrow O(n^2)$ , possible in  $O(n)$
- ▶ Keep track of the Z-box with right end furthest to the right (bounds:  $[l, r]$ )

# Z-algorithm

## Calculating Z-scores

- ▶ Naive  $\Rightarrow O(n^2)$ , possible in  $O(n)$
- ▶ Keep track of the Z-box with right end furthest to the right (bounds:  $[l, r]$ )
- ▶ if current character in  $[l, r]$ : look at corresponding character in prefix (computed previously)

# Z-algorithm

## Calculating Z-scores

- ▶ Naive  $\Rightarrow O(n^2)$ , possible in  $O(n)$
- ▶ Keep track of the Z-box with right end furthest to the right (bounds:  $[l, r]$ )
- ▶ if current character in  $[l, r]$ : look at corresponding character in prefix (computed previously)
- ▶ expand if grows beyond  $r$ , update  $[l, r]$

# Z-algorithm

## Calculating Z-scores

- ▶ Naive  $\Rightarrow O(n^2)$ , possible in  $O(n)$
- ▶ Keep track of the Z-box with right end furthest to the right (bounds:  $[l, r]$ )
- ▶ if current character in  $[l, r]$ : look at corresponding character in prefix (computed previously)
- ▶ expand if grows beyond  $r$ , update  $[l, r]$
- ▶ else: calculate explicitly, update  $[l, r]$

# Z-algorithm

## Calculating Z-scores

- ▶ Naive  $\Rightarrow O(n^2)$ , possible in  $O(n)$
- ▶ Keep track of the Z-box with right end furthest to the right (bounds:  $[l, r]$ )
- ▶ if current character in  $[l, r]$ : look at corresponding character in prefix (computed previously)
- ▶ expand if grows beyond  $r$ , update  $[l, r]$
- ▶ else: calculate explicitly, update  $[l, r]$
- ▶ (Nicely illustrated:  
<https://www.cs.umd.edu/class/fall2011/cmsc858s/Lec02-zalg.pdf>)

# Z-algorithm

code

```
int match(string s, string pat)
{
    string S = pat + "$" + s;
    vector<int> Z(S.length());
    int l = -1, r = -1;

    for (int i = 1; i < S.length(); i++)
        if (i > r) //Outside furthest Z-box
        {
            int len = 0;
            for (int j = i; j < S.length() && S[j] == S[j-i]; j++)
            {
                len++;
            }
            Z[i] = len;
            if (len > 0)
            {
                l = i;
                r = i + len - 1;
            }
        }
    else
```



# Z-algorithm

code

```
{
    int inside = r - i + 1;
    int corresponding = i - l;
    if (Z[corresponding] < inside)
    {
        Z[i] = Z[corresponding];
    }
    else //Need to grow beyond r
    {
        int len = 0;
        for (int j = r + 1; j < S.length() && S[j] == S[j - i]; j++)
        {
            len++;
        }
        Z[i] = inside + len;
        if (i + len - 1 > r) //Only update if better
        {
            l = i;
            r = i + len - 1;
        }
    }
}
for (int i = 1; i < S.length(); i++)
    if (Z[i] == pat.length())
        return i - pat.length() - 1; //Don't forget to subtract the sentinel
return -1;
}
```

# Table of Contents

Ad hoc

Tries

String matching

Naive

Rabin-Karp

Z-algorithm

**Knuth-Morris-Pratt**

Exercises

# Knuth-Morris-Pratt

## Idea

- ▶ Don't inspect substring starting at each index

# Knuth-Morris-Pratt

## Idea

- ▶ Don't inspect substring starting at each index
- ▶ Fail smart

# Knuth-Morris-Pratt

## Idea

- ▶ Don't inspect substring starting at each index
- ▶ Fail smart
- ▶ Jump ahead to the next possible index, given what we've seen so far

# Knuth-Morris-Pratt

## Idea

- ▶ Don't inspect substring starting at each index
- ▶ Fail smart
- ▶ Jump ahead to the next possible index, given what we've seen so far
- ▶ Precompute the failure jumps

# Knuth-Morris-Pratt

Idea

- ▶ How to choose the jump?

# Knuth-Morris-Pratt

## Idea

- ▶ How to choose the jump?
- ▶ Next possible partially matched pattern = longest proper suffix (of the partial match) that is a prefix



# Knuth-Morris-Pratt

## Idea

- ▶ How to choose the jump?
- ▶ Next possible partially matched pattern = longest proper suffix (of the partial match) that is a prefix
- ▶ Precompute and keep the length of this suffix/prefix in an array (call this  $L$ )

# Knuth-Morris-Pratt

## Idea

- ▶ How to choose the jump?
- ▶ Next possible partially matched pattern = longest proper suffix (of the partial match) that is a prefix
- ▶ Precompute and keep the length of this suffix/prefix in an array (call this  $L$ )
- ▶  $L[i] = \text{length of that prefix for } S[0..i-1] \text{ (inclusive)}$

# Knuth-Morris-Pratt

## Precomputation

- ▶  $L[0] = L[1] = 0$  (it should be a *proper* suffix)

# Knuth-Morris-Pratt

## Precomputation

- ▶  $L[0] = L[1] = 0$  (it should be a *proper* suffix)
- ▶ Search for the next *parent* in  $L$  that can be expanded with the current character

# Knuth-Morris-Pratt

## Precomputation

- ▶  $L[0] = L[1] = 0$  (it should be a *proper* suffix)
- ▶ Search for the next *parent* in  $L$  that can be expanded with the current character
- ▶  $L[i] = j + 1$  ( $j$  is the length of the *parent's* match)

# Knuth-Morris-Pratt

## Precomputation

- ▶  $L[0] = L[1] = 0$  (it should be a *proper* suffix)
- ▶ Search for the next *parent* in  $L$  that can be expanded with the current character
- ▶  $L[i] = j + 1$  ( $j$  is the length of the *parent's* match)
- ▶ If none can be found:  $L[i] = 0$

# Knuth-Morris-Pratt

## Matching

- ▶ Precompute the failure jumps ( $L$ ) of the pattern

# Knuth-Morris-Pratt

## Matching

- ▶ Precompute the failure jumps ( $L$ ) of the pattern
- ▶ Skip characters using  $L$  while matching



# Knuth-Morris-Pratt

## Matching

- ▶ Precompute the failure jumps ( $L$ ) of the pattern
- ▶ Skip characters using  $L$  while matching
- ▶ Very similar to the actual precomputation

# Knuth-Morris-Pratt

## Matching

- ▶ Precompute the failure jumps ( $L$ ) of the pattern
- ▶ Skip characters using  $L$  while matching
- ▶ Very similar to the actual precomputation
- ▶  $O(S + P)$

# Knuth-Morris-Pratt

code

```
vector<int> precompute(string pat)
{
    vector<int> L(pat.length() + 1);
    L[0] = L[1] = 0;
    for (int i = 2; i <= pat.length(); i++)
    {
        for (int j = L[i-1]; ; j = L[j])
        {
            if (pat[j] == pat[i - 1])
            {
                L[i] = j + 1;
                break;
            }
            if (j == 0)
            {
                L[i] = 0;
                break;
            }
        }
    }
    return L;
}
```

# Knuth-Morris-Pratt

code

```
int match(string s, string pat)
{
    vector<int> L = precompute(pat);
    int j = 0; //The current index for L
    for (int i = 0; i < s.length(); i++)
    {
        while (j != 0 && s[i] != pat[j])
        {
            j = L[j];
        }
        if (s[i] == pat[j])
            j++;
        if (j == pat.length())
            //Found a match, reconstruct the beginning of the substring
            return i + 1 - j;
    }
    return -1;
}
```

# Table of Contents

Ad hoc

Tries

String matching

Naive

Rabin-Karp

Z-algorithm

Knuth-Morris-Pratt

Exercises

# Exercises

Ad hoc

- ▶ UVa 10115 (*Automatic Editing*)
- ▶ UVa 10361 (*Automatic Poetry*)
- ▶ UVa 10082 (*WERTYU*)
- ▶ UVa 1368 (*DNA Consensus String*)

# Exercises

## Tries

- ▶ UVa 902 (*Password Search*)
- ▶ UVa 755 (*487–3279*)
- ▶ Codechef Remember the recipe  
(<https://www.codechef.com/problems/TWSTR/>)

# Exercises

## String matching

- ▶ UVa 363 (*Approximate matches*)
- ▶ UVa 455 (*Periodic strings*)
- ▶ UVa 1223 (*Editor*)
- ▶ Codeforces 126, problem B  
(<http://codeforces.com/contest/126/problem/B>)
- ▶ UVa 11151 (*Longest Palindrome*)
- ▶ UVa 11475 (*Extend to Palindrome*)