Basic String Algorithms

Mike Mirzayanov
Codeforces

Definitions

A **string** is a finite (possibly empty) sequence of characters such as letters, digits, spaces or numbers. Examples, s_1=”010101”, s_2=”abacaba”, s_3=[31,34,41], s_4=””. We use $|s|$ as a length of s.

A **character** is an element of alphabet.

An **alphabet** is non-empty set of characters. Usually alphabet is finite, but it is not important in this lecture. Examples, Σ_1={‘0’,’1’}, Σ_2={‘a’, ‘b’, …, ‘z’}.

Substring of *s* is consecutive subsequence of characters from *s*. The list of all substrings of the string "apple" would be "apple", "appl", "pple", "app", "ppl", "ple", "ap", "pp", "pl", "le", "a", "p", "l", "e", "".

Definitions

An **occurrence** of a substring t in s is such pair of indices (l,r) that $t=s_l s_{l+1}...s_r$.

A **prefix** of s is a such substring $s_0 s_1...s_i$. A **proper prefix** of a string is not equal to the string itself.

A **suffix** of s is a such substring $s_i s_{i+1}...s_{|s|-1}$. A **proper suffix** of a string is not equal to the string itself.

A **border** is proper suffix and proper prefix of the same string, e.g. "bab", “b” and “” are borders of "babab".

Example: for s=”aabaaba” borders are: “aaba”, “a” and “”.

String Searching (Matching) Problem

You are given string *t* called text and string *p* called pattern. Find all occurrences of *p* in *t*.

Example: t=”abacababa”, p=”aba”. There are three occurrences:

- <u>aba</u>cababa: (0, 2)
- abac<u>aba</u>ba: (4, 6)
- abacab<u>aba</u>: (6, 8)

Let’s n=|t| and m=|p|. The naive algorithm works in $O(nm)$.

String Searching (Matching) Problem

There are two main ways how deal with problem:

- Preprocess text (z-function, prefix-function)
- Preprocess pattern (suffix tree/array)

Z-function

For given string $s=s_0s_1...s_{n-1}$ the z-function is array indexed by indices of the string. So it is $z[0]$, $z[1]$, ..., $z[n-1]$, where z[i] is the length of the longest common prefix of s and $s[i..n-1]$. Usually $z[0] = 0$ or $z[0] = n$.

Examples

- s=”abacaba”, z=[0, 0, 1, 0, 3, 0,1]
- s=”aaaaaaaa”, z=[0, 7, 6, 5, 4, 3, 2, 1]
- s=”abababab”, z=[0, 0, 6, 0, 4, 0, 2, 0]

Z-function

Exercise

1) Find z-function of s=”abaababa”.
2) Find z-function of s=”baababaab”.

Z-function

Answers:

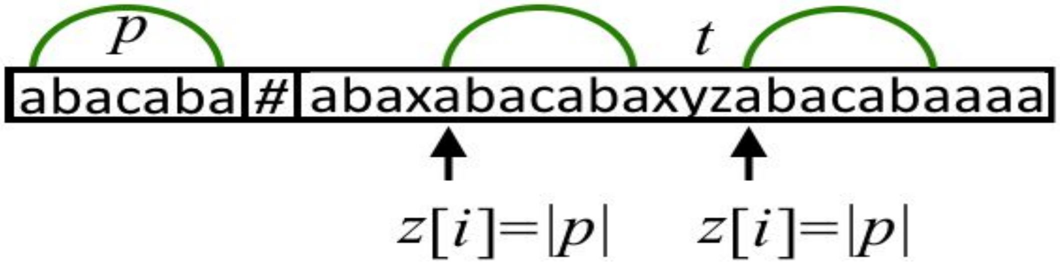
1) [0, 0, 1, 3, 0, 3, 0, 1]
2) [0, 0, 0, 2, 0, 4, 0, 0, 1]

The naive $O(n^2)$ algorithm:

```
for i = 1..n-1
    while z[i] + i < n && s[z[i] + i] == s[z[i]]:
        z[i]++
```

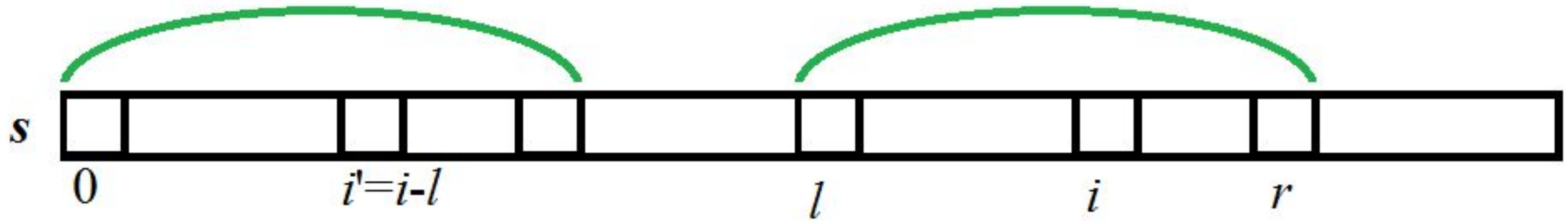
Z-function (motivation)

- $s := p + \text{“\#”} + t$



- Positions i ($i>|p|$) where $z[i] = |p|$ correspond to beginnings of occurrences.
- So if there is linear $O(|s|)$ time algorithm to find z-function then linear solution for string matching problem exists.

Z-algorithm (fast calculation)



Maintain z-block [*l*,*r*] containing *i*, such that s[0..*r*-*l*] = s[*l*..*r*]. The value *r* is maximal possible among all such blocks.

On each step:

* If i<=*r* then initialize *z*[i]=min(z[i - l], *r* - i + 1)
* After it do naive z[i] growth
* Update l and *r*?

Z-algorithm (fast calculation)

```
l = r = 0
for i = 1..n-1:
    if r >= i:
        z[i] = min(z[i - l], r - i + 1)
    while z[i] + i < n && s[z[i]] == s[z[i] + i]:
        z[i]++
    if i + z[i] - 1 > r:
        l = i, r = i + z[i] - 1
```

- Runs in $O(n)$ because on each iteration of internal loop r moves right.

Z-algorithm (applications)

- String Searching Problem.
- Number of different substrings in a string in O(n^2).
- String Period: s=tttttt. Find such smallest i that $i + z[i] = n$ and n is divisible by i.
- Matching with one mistake in $O(n+m)$.

Prefix-function

A **border** of a string is such proper prefix which is its proper suffix at the same time.

Examples

- s="abacaba", borders: {"", "a", "aba"}
- s="aaaaa", borders: {"", "a", "aa", "aaa", "aaaa"}

For the given string $s=s_0s_1...s_{n-1}$ the **prefix function** is array $b[0..n-1]$, where $b[i]$ is the length of longest border of $s[0..i]$ (i.e. of the prefix of length $i+1$).

Prefix-function

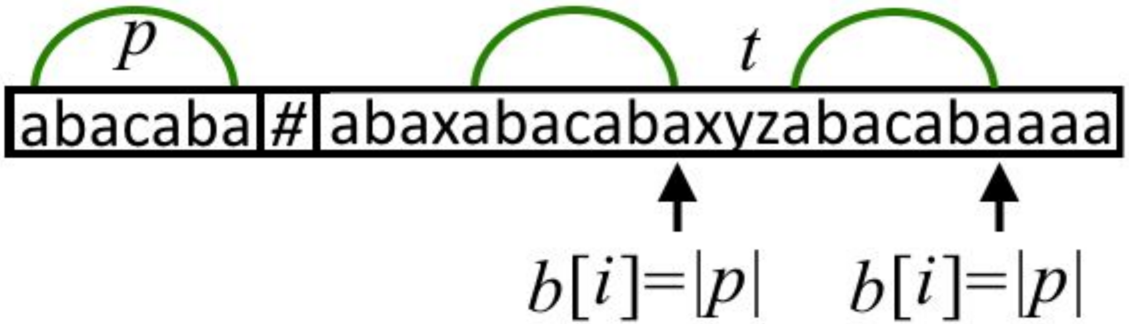
For the given string $s=s_0s_1\ldots s_{n-1}$ the **prefix function** is array $b[1..n]$, where $b[i]$ is the length of longest border of $s[0..i]$ (i.e. of the prefix of length $i+1$).

Examples

- s="abacaba", b=[0, 0, 1, 0, 1, 2, 3]
- s="aaaaaaaa", b=[0, 1, 2, 3, 4, 5, 6, 7]
- s="abababab", b=[0, 0, 1, 2, 3, 4, 5, 6]

Prefix-function (motivation)

- $s := p + \text{“\#”} + t$



- Positions i (i>|p|) where $b[i]$ = |p| correspond to ends of occurrences.
- So if there is linear $O(|s|)$ time algorithm to find prefix-function then linear solution for string matching problem exists.

Prefix-function (properties)

Properties

- Grows for at most 1: $b[i + 1] <= b[i] + 1$

- $b[i]$ is length of border, $b[b[i]-1]$ is also length of border, $b[b[b[i]-1]-1]$ is also length of border and so on.

Prefix-function (fast computation)


```
for i=1..n-1:
    k = b[i - 1]
    while k > 0 && s[k] != s[i]:
        k = b[k - 1]
    if s[k] == s[i]:
        b[i] = k + 1
```

Prefix-function (Knuth-Morris-Pratt Algorithm)

- Precompute *b* - prefix-function of *p*
- Maintain *k* - length of longest suffix of *t* which is also a proper prefix of *p*
- Needs only *O*(|*p*|) additional memory

```
for c in t:
    while k > 0 && p[k] != c:
        k = b[k - 1]
    if p[k] == c:
        k = k + 1
        if k == |p|:
            an occurrence ends in c
            k = b[k - 1]
```

Prefix-function (applications)

- Knuth-Morris-Pratt Algorithm
- Number of different substrings in *s* (almost the same as for z-function)
- String Period (If n-$b[n$-$1]$ divides n, it is the answer)
- Longest Palindromic Prefix
  - To find such longest prefix which is palindrome.
  - Calculate last value of prefix function for s+#+reverse(s).

Prefix-function (finite state machine)

Examples

p = aba$

A(p):

Length	**If append ‘a’**	**If append ‘b’**
0	1	0
1	1	2
2	3	0
3	1	2

- A[0]['a'] = 1
- A[1]['b'] = 2
- A[1]['a'] = 1
- A[3]['b'] = 2
- ...

Prefix-function (finite state machine)

Exercise

- Find finite state machine A for p=aaba$

Prefix-function (finite state machine)

Exercise

- Find finite state machine A for p=aaba$

Answer

Length	**If append ‘a’**	**If append ‘b’**
0	1	0
1	2	0
2	2	3
3	4	0
4	2	0

Prefix-function (finite state machine)

Exercise

- Find finite state machine A for p=ababca$

Prefix-function (finite state machine)

Exercise

- Find finite state machine A for p=ababca$

Answer:

Length	If append ‘a’	If append ‘b’	If append ‘c’
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	3	0	5
5	6	0	0
6	1	2	0

Prefix-function (finite state machine)

```
for c = 'a'..'z':
    if p[0] == c:
        A[0][c] = 1
    else:
        A[0][c] = 0
for i = 1..|p|-1:
    for c = 'a'..'z':
        if p[i] == c:
            A[i][c] = i + 1
        else:
            A[i][c] = A[b[i - 1]][c]
```

Prefix-function (finite state machine)

Problem

g[0] = “”, *g*[1] =”a”, *g*[2] = “aba”, *g*[3] = “abacaba”, *g*[4] = “abacabadabacaba”, …

Number of occurrences of *s* in *g*[*k*]?

F[*i*][*k*] = state if initial state is *i* and *g*[*k*] is input.

```
if k=0:
    F[i][k]=i
else:
    x = F[i][k-1]
    y = A[x][‘a’+k-1]
    F[i][k] = F[y][k-1]
```

Prefix-function (finite state automata)

Problem

g[0] = “”, *g*[1] =”a”, *g*[2] = “aba”, *g*[3] = “abacaba”, *g*[4] = “abacabadabacaba”, …

Number of occurrences of *s* in *g*[*k*]?

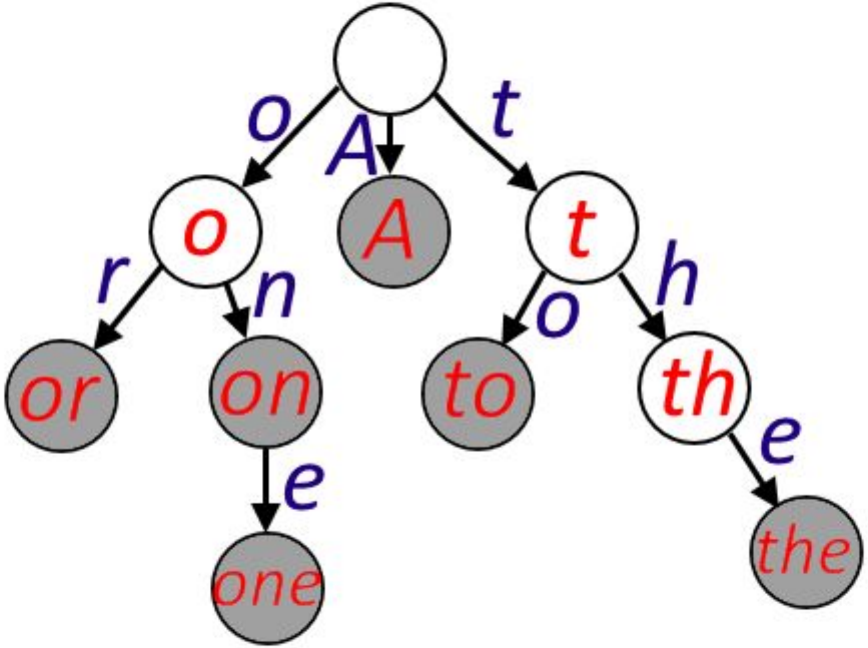
R[*i*][*k*] = number of additional matches if current state is *i* and we append *g*[*k*]:

```
R[i][k] = R[i][k - 1]
x = F[i][k-1]
y = A[x][‘a’+k-1]
if y==|p|:
    R[i][k] = R[i][k] + 1
F[i][k] = F[y][k-1]
R[i][k] += R[y][k-1]
```

Trie

Given a set of strings *S*. Trie is a rooted outgoing tree with:

- edges marked with chars
- for each vertex outgoing edges are marked with different chars
- one can pronounce all prefixes of string from *S* (and only them) moving from the root



The trie for {“A”, “or”, “on”, “one”, “to”, “the”}.

Trie

Property

- Consider all distinct prefixes of strings from S. Each prefix is exactly one node of a trie.

Applications:

- Test if w is in S in $O(|w|)$
- Find the longest prefix of w and some word in S in $O(k)$, where k is the length of the longest prefix
- Used in DP problems, node is a state in DP

Trie

```
struct node {
    node* nxt[26] = {0};
    bool end = false;
    int c = 0;
    node* p = nullptr;
};
```

```
trie root = new node();

function add(s):
    trie t = root
    for i = 0..|s|-1:
        int c = s[i] - 'a'
        if t->nxt[v] == nullptr:
            trie child = new node()
            child->c = c
            child->p = t
            t->nxt[v] = child
        t = t->nxt[v]
        if i + 1 == |s|:
            t->end = true
```

Thank you

Questions?