String Algorithms

Jeremy Chow {jeremy624}

2021-05-15

Table of Contents

1. Trie
2. Hashing
3. KMP Algorithm
4. Suffix Array

Definitions

String

- a sequence of characters
  - char s[]
  - string s
- “Hello world!”

ASCII

- American Standard Code for Information Interchange
- 7 bits / 8 bits (1 byte)
- 0 ~ 127 / 0 ~ 255
- “0” = 48, “A” = 65, “a” = 97

Definitions

Concatenation

- addition in string
- 1 + 2 = 3
- “1” + “2” = “12”
- “ab” + “c” = “abc”

Lexicographical order

- order in string
- “2” < “3”
- “23” < “3”
- “bc” < “bcf”
- “bd” > “bcf”

Definitions

Substring

- **contiguous** sequence of characters within a string
- prefix of suffix / suffix of prefix
- ABCDE
- ABCDE
- ABCDE

Definitions

Prefix

- substring that start with the **beginning** of the string
- ABCDE
- ABCDE
- ABCDE

Suffix

- substring that end with the **end** of the string
- ABCDE
- ABCDE
- ABCDE

Definitions

Subsequence

- obtained by **deleting some or no characters** of a string
- order is kept
- ABCDE
- ABCDE
- ABCDE

Definitions

Palindrome

- string that **remains same when reversed**
- A
- ABA
- ABBA
- ABCBA

Trie

Trie

Not a formal word

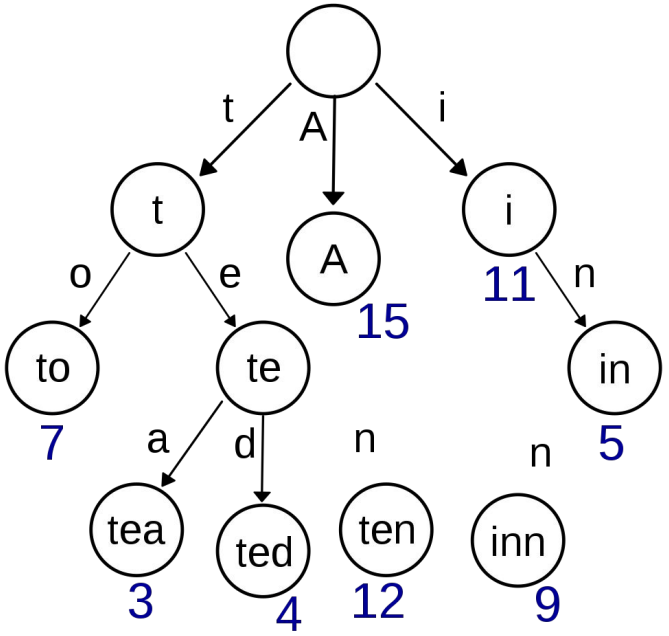
- come from the word retrieval

Pronounce as “try”

Tree data structure

Dictionary of a set of strings

- support quick insert and lookup

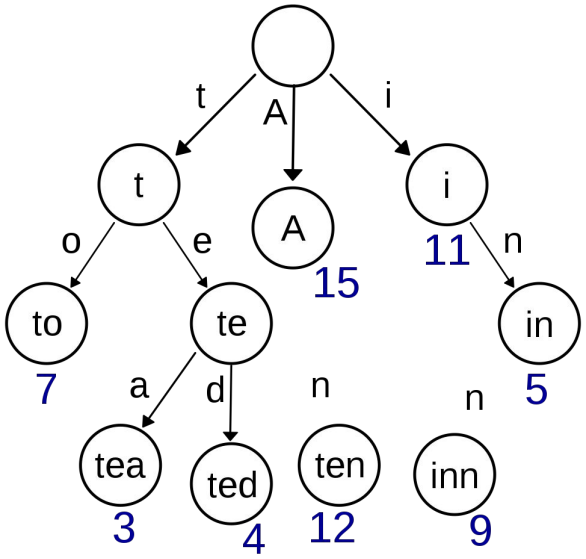


Trie

Edge: addition of a character

Node: a word / prefix of a string

- obtained by concatenating characters from root to this node

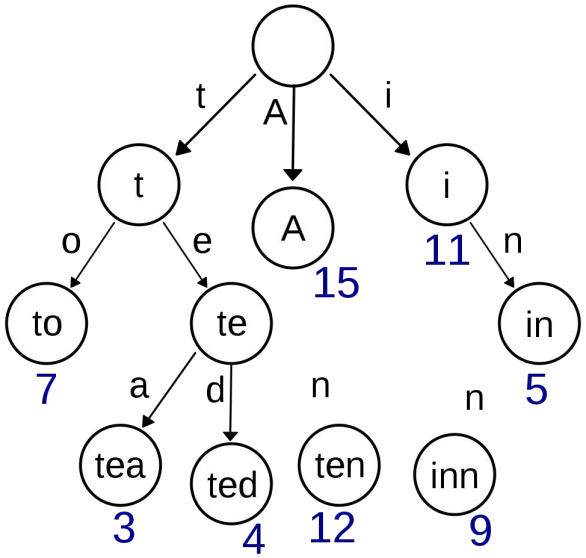


Usually, we

- Store a boolean, indicating whether this node represent **the end of a word**
- Store a integer to represent **frequency** is duplicated string is allowed

Trie

```
struct TrieNode {
    int children[ALPHABET_SIZE];
    bool isword;
}Trie[SIZE];
```



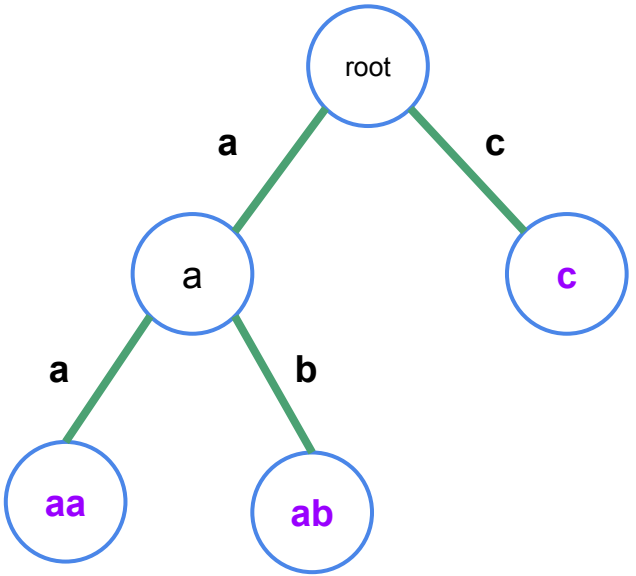
APLHABET_SIZE depends of your input

- lowercase letters -> 26
- ascii -> 128

we assume the input is all lowercase letters in the following slides

Trie

Trie of {“aa”, “ab”, “c”}



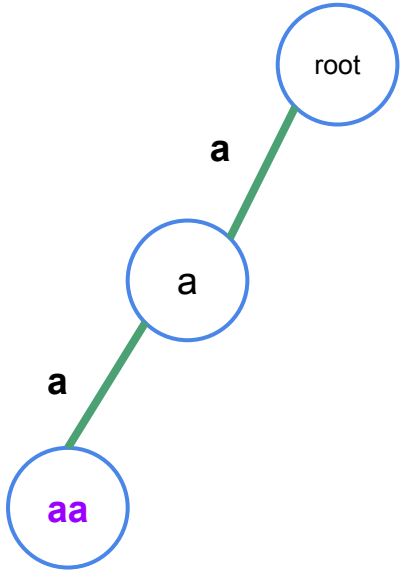
Trie

```
void insert(string s) {
    int cur = 0;
    for (int i = 0; i < s.size(); i++) {
        if (!Trie[cur].children[s[i] - 'a'])
            Trie[cur].children[s[i] - 'a'] = siz++;
        cur = Trie[cur].children[s[i] - 'a']];
    }
    Trie[cur].isword = 1;
}
```

- Similar implementation for delete operation

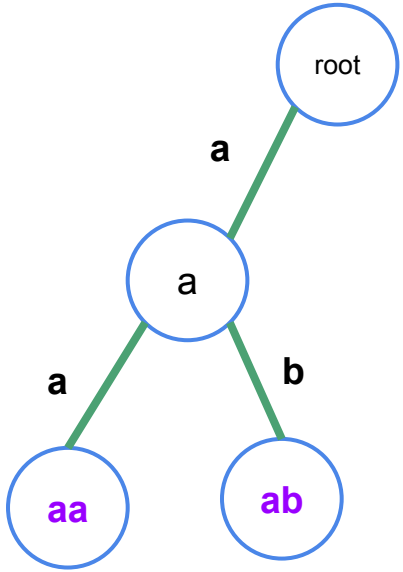
Trie

```
insert("aa")
```



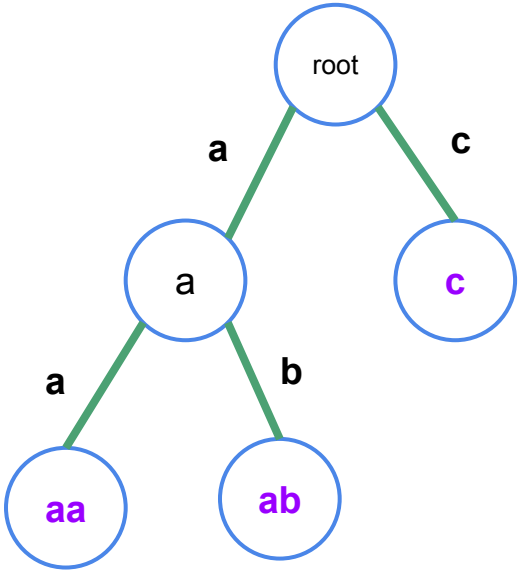
Trie

```
insert("ab")
```



Trie

```
insert("c")
```

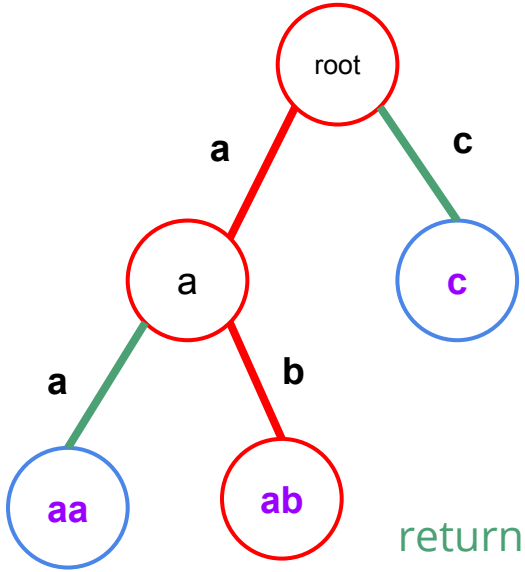


Trie

```
bool lookup(string s) {
        int cur = 0;
        for (int i = 0; i < s.size(); i++) {
                if (!Trie[cur].children[s[i] - 'a'])
                        return 0;
                cur = Trie[cur].children[s[i] - 'a'];
        }
        return Trie[cur].isword;
}
```

Trie

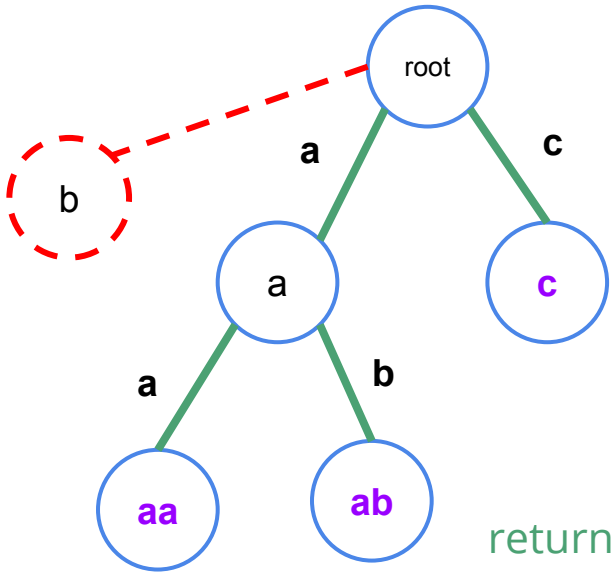
```
lookup("ab")
```



return true

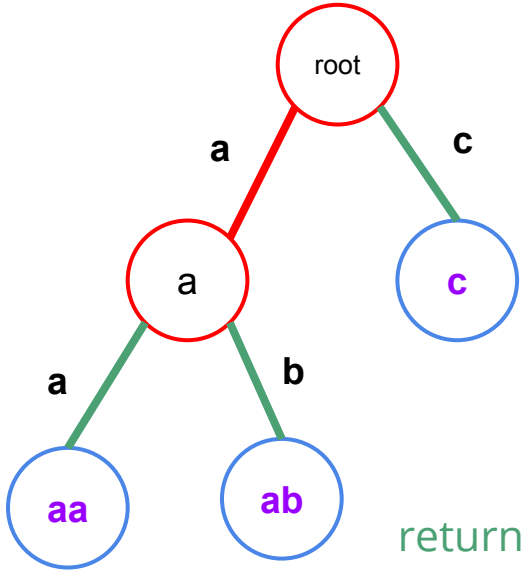
Trie

```
lookup("b")
```



Trie

`lookup("a")`



return false

Trie

N: total length M: length of the target string

Time complexity

- Insert = O(M)
- Delete = O(M)
- Lookup = O(M)
- All of them are dfs

Memory Complexity = O(`APLHABET_SIZE` × N)

Trie

Useful in solving string problems

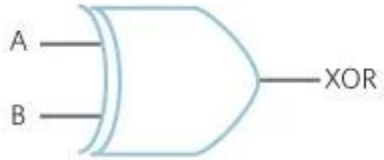
Special applications

- Represent 32-bit signed integer as binary string with length of 31
- Act as an Binary Search Tree

Useful to solve problems about xor

- e.g. Maximum xor-sum subarray

$$X = A \oplus B$$



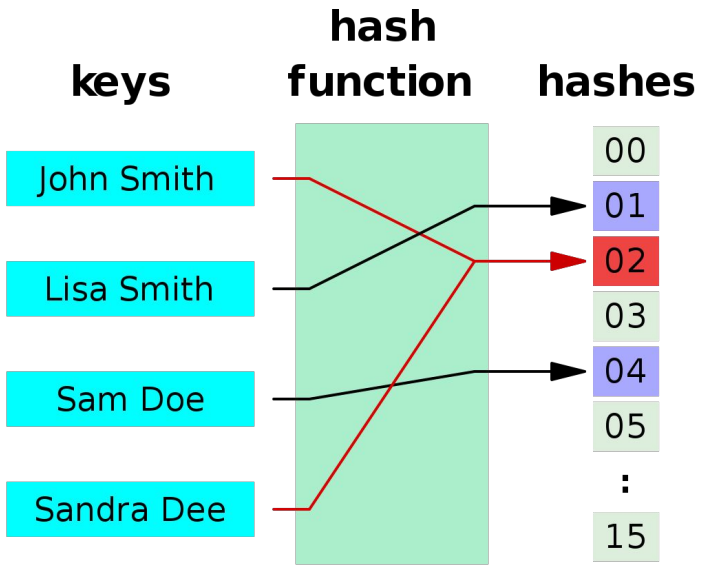
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Hashing

Hashing

Hashing in number

- number → number
- Hash table
- Data structure (II)



Hashing in string

- string → number
- compare number instead of string

Hashing

Polynomial rolling hash

- common hash function in CP
- only uses multiplications and additions

$$H = (c_0 a^{k-1} + c_1 a^{k-2} + c_2 a^{k-3} + \cdots + c_{k-1} a^0) \mod p$$

H = hash value

c_i = ascii value of i^{th} character

a = some constant

p = some constant, usually a prime

use unsigned long long → p = 2^{64}

Hashing

$$H = (c_0 a^{k-1} + c_1 a^{k-2} + c_2 a^{k-3} + \cdots + c_{k-1} a^0) \mod p$$

"HKOI"

a = 26, p = 64997

c_0 = 7, c_1 = 10, c_2 = 14, c_3 = 8

- Subtract 'A'

H = (7×26^3 + 10×26^2 + 14×26 + 8) mod 64997

= 130164 mod 64997

= 170

Hashing

$$H = (c_0 a^{k-1} + c_1 a^{k-2} + c_2 a^{k-3} + \cdots + c_{k-1} a^0) \mod p$$

"GO"

a = 26, p = 64997

c_0 = 6, c_1 = 14

- Subtract 'A'

H = (6*26+14) mod 64997
  = 170 mod 64997
  = 170

Hashing

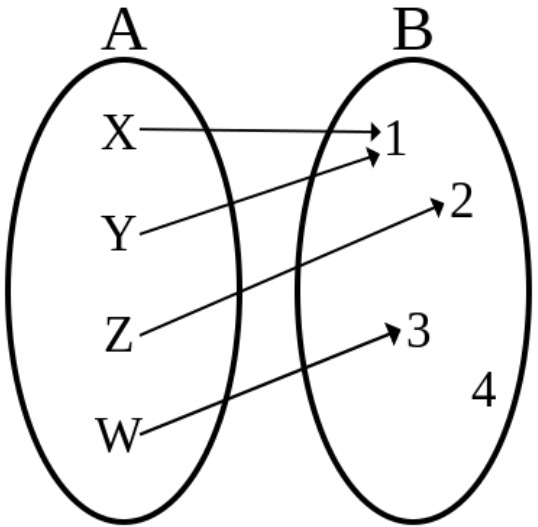
`H("HKOI") = 170 = H("GO")`

`"HKOI" ≠ "GO"`

collision occurs

Hash function is not an injective function

One hash value may represents multiple strings



Hashing

Solution 1: pick better constants `a` and `p` to reduce the probability of collision

- `p` → prime number
- `a` > `max(`c_i`)`

Solution 2 : Double hashing

- Use two pair of `a` and `p`, compare two hash values

Solution 3 : Give up hashing

- Use exact algorithm e.g. KMP algorithm

Hashing

BCDE

$a = 26,\ p = 64997$

$H = (1 \times 26^2 + 2 \times 26 + 3) \bmod 64997$

BCDE

$H' = (2 \times 26^2 + 3 \times 26 + 4) \bmod 64997$

$H' = ((H - 1 \times 26^2) \times 26 + 4) \bmod 64997$

$H = ((H' - 4) \times 26^{-1} + 1 \times 26^2) \bmod 64997$

Hashing

Sliding window

- $H_i = (c_i a^{k-1} + c_{i+1} a^{k-2} + c_{i+2} a^{k-3} + ... + c_{i+k-1} a^0) \bmod p$
- $H_{i+1} = (c_{i+1} a^{k-1} + c_{i+2} a^{k-2} + c_{i+3} a^{k-3} + ... + c_{i+k} a^0) \bmod p$

$H_{i+1} = ((H_i - c_i a^{k-1}) * a + c_{i+k}) \bmod p$

$H_i = ((H_{i+1} - c_{i+k}) * a^{-1} + c_i a^{k-1}) \bmod p$

Handle negative number $\rightarrow$ $(x - y + p) \bmod p$

Hashing

Similar idea to get hash values of substring in O(1)

- Let $H_i = (c_0a^i+c_1a^{i-1}+c_2a^{i-2}+...+c_ia^0) \bmod p$
  - partial hash sum array
- $\text{Hash}(l, r) = (H_r - H_{l-1} * a^{r-l+1}) \bmod p$

Useful technique in string matching problems

Hashing

HKOJ 01002 A Counting Problem

Given string `s` and `t`, find number of occurrences of string `t` in string `s`

- `Allow overlap`

```
abcdefabcghiabcabcjklmnlabcw
abc
```

```
Ans = 5
```

Hashing

HKOJ 01002 A Counting Problem

Idea: compare T to each substring of S with length |T|

Compare character by character

- O(|S|×|T|)

Compare by hash values

- O(|S| + |T|)

Hashing

a = 131, p = 2^64 (unsigned long long)

```
int res = 0;
if (m - 1 < n && prefixHash[m - 1] == targetHash) res++;

for (int i = m; i < n; i++) {
        unsigned long long h = prefixHash[i] - prefixHash[i - m] * pw[m];
        if (h == targetHash) res++;
}
```

Note: use gets() instead of scanf() to pass test case 10

Hashing

HKOJ M0932 String Rotation

Given string S and T, find the number of rotation required so that string S' = T

|S| = |T|

ABCDE → BCDEA → CDEAB → DEABC → EABCD

EABCD

Ans = 4

Hashing

HKOJ M0932 String Rotation

Idea: compare T to each rotation of S

Compare character by character

- $O(|S|^2)$

Compare by hash values using sliding windows

- $O(|S|)$

Hashing

In contest, problem setter may prepare some anti hash test

- https://codeforces.com/blog/entry/4898
- receive WA
- depends on the value of p (modules)

Use different choices of p

Double hashing

KMP Algorithm

KMP Algorithm

String matching problems

- determine whether one or several strings are found within a large string

Knuth–Morris–Pratt algorithm

Solve single pattern matching problem

- determine whether a short string are found within a large string

KMP Algorithm

HKOJ 01002 A Counting Problem

Given string `s` and `t`, find number of occurrences of string `t` in string `s`

- `Allow overlap`

```
abcdefabcghiabcabcjklmnlabcw
abc

Ans = 5
```

KMP Algorithm

Compare character by character

```
for each position i in S
    check whether S[i..i+|T|-1] == T
```

Time complexity = $O(|S| * |T|)$

Enough to get AC on HKOJ

KMP Algorithm

Occurence = 1

```
abcdefabcghiabcabcjklmnlabcw

abc
```

KMP Algorithm

Occurence = 1

```
abcdefabcghiabcabcjklmnlabcw

 abc
```

KMP Algorithm

Occurence = 1

```
abcdefabcghiabcabcjklmnlabcw

   abc
```

KMP Algorithm

Occurence = 1

```
abcdefabcghiabcabcjklmnlabcw

   abc
```


2000 YEARS
LATER

KMP Algorithm

Occurence = 5

```
abcdefabcghiabcabcjklmnlabcw
                        abc
```

KMP Algorithm

What if the constraint is bigger

- $|S|, |T| \leq 10^5$

Too slow

TLE

Find a faster algorithm!

KMP Algorithm

Comparing whole string is too slow

- $O(|T|)$

Comparing hash value of string instead

- $O(1)$
- Maintain hash value with sliding window
- Time complexity = $O(|S| + |T|)$
- Collisions may result in WA

KMP Algorithm

Comparing whole string is too slow

- $O(|T|)$
- Can we do better..?

Need to recalculate everything after a mismatch

Idea: make use of the information in the previous match

avoid rematching

KMP Algorithm

Match `T` with `S`

Compare character by character

There is a mismatch in index 5 (0-based)

- `S[5] != T[5]`

ABABABABACB
ABABACB

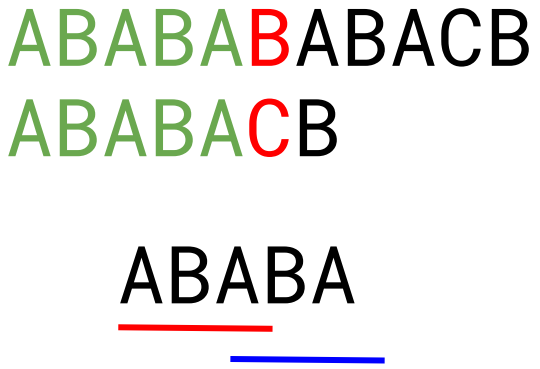
KMP Algorithm

Already matched "ABABA" in the first iteration

- T[0..2] = T[2..4]
- "ABA"

"ABA" is longest string s.t.

- it is not the whole string of "ABABA"
- it is the prefix suffix of "ABABA"
  - prefix and suffix at the same time



KMP Algorithm

We already know `S[0..4] = T[0..4] = "ABABA"`

Also, `T[0..2] = T[2..4]`

Therefore, `S[2..4] = T[0..2]`

ABABABABACB
ABABACB

Don't need to rematch this part

KMP Algorithm

Why skip checking `S[1..7] = T`?

Because `S[0..4] = T[0..4] = “ABABA”`

`“ABAB” is not a longest prefix suffix`

- `T[0..3] != T[1..4]`
- `T[0..3] != S[1..4]`

We already know that `T` will not match `S` starting at position 1

KMP Algorithm

How to find the longest prefix suffix of a string?

- By dynamic programming

`next[i] = length(longest prefix suffix of T[0..i]) - 1`

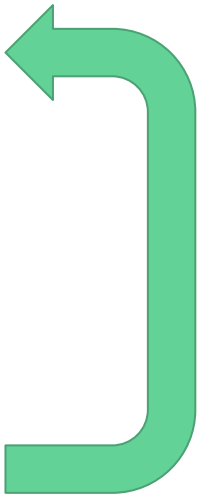
- `T[0..next[i]] = T[i - next[i]..i] and next[i] < i`

T	A	B	A	B	A	C	B
next[i]	-1	-1	0	1	2	-1	-1

KMP Algorithm

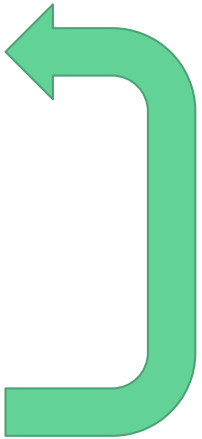
We know `next[i - 1]` and want to calculate `next[i]`

```
j = next[i - 1]
if T[j + 1] == T[i]
  • next[i] = j + 1;
else if j == -1
  • next[i] = -1;
else
  • j = next[j]
  • repeat
```



KMP Algorithm

```
if T[j + 1] == T[i]
  • next[i] = j + 1;
else if j == -1
  • next[i] = -1;
else
  • j = next[j]
  • repeat
```



`T[2] != T[5]`
`T[0..2] != T[3..5]`

need to find smaller `j` such that `T[0..j] = T[5-j..5]`

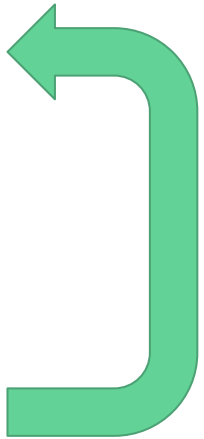
smaller `j` = longest prefix of `T[0..j]` = suffix of `T[4-j..4]`

= longest prefix suffix of `T[0..j]`

T	A	A	B	A	A	A
next[i]	-1	0	-1	0	1	?

KMP Algorithm

```
if T[j + 1] == T[i]
  • next[i] = j + 1;
else if j == -1
  • next[i] = -1;
else
  • j = next[j]
  • repeat
```



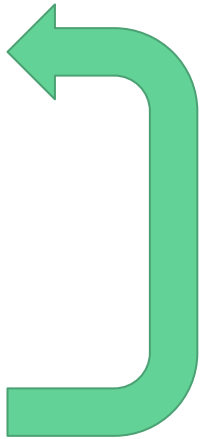
```
j = 0
T[1] = T[5]
T[0..1] = T[4..5]

next[5] = j + 1 = 1
```

T	A	A	B	A	A	A
next[i]	-1	0	-1	0	1	1

KMP Algorithm

```
if T[j + 1] == T[i]
  • next[i] = j + 1;
else if j == -1
  • next[i] = -1;
else
  • j = next[j]
  • repeat
```



```
for(i = 1, j = next[0] = -1; i < l1; i++){
    while(j >= 0 && t[j + 1] != t[i]) j = next[j];

    if(t[j + 1] == t[i]) j++;

    next[i] = j;
}
```

Time Complexity: O(N)

KMP Algorithm

We can speed up the matching with the help of `next[]`

`j = length of (matched character in position i - 1) - 1`

- `T[0..j] = S[i-1-j..i-1]`

`if T[j + 1] == S[i]`

- `j++`

`else if j != -1`

- `j = next[j]`
- `repeat`

`if j == |T| - 1`

- `found T in S, res++`
- `j = next[j]`

KMP Algorithm

ABABABABACB
ABABACB

T	A	B	A	B	A	C	B
next[i]	-1	-1	0	1	**2**	-1	-1

KMP Algorithm

ABABABABACB
ABABACB

T	A	B	A	B	A	C	B
next[i]	-1	-1	0	1	2	-1	-1

KMP Algorithm

ABABABABACB
ABABACB

T	A	B	A	B	A	C	B
next[i]	-1	-1	0	1	2	-1	-1

KMP Algorithm

ABABABABACB
ABABACB

T	A	B	A	B	A	C	B
next[i]	-1	-1	0	1	**2**	-1	-1

KMP Algorithm

ABABABABACB
ABABACB

T	A	B	A	B	A	C	B
next[i]	-1	-1	0	1	2	-1	-1

KMP Algorithm

ABABABABACB
ABABACB

T	A	B	A	B	A	C	B
next[i]	-1	-1	0	1	2	-1	-1

KMP Algorithm

j = length of (matched character in position i - 1) - 1

- T[0..j] = S[i-1-j..i-1]

if T[j + 1] = S[i]

- j++

else if j != -1

- j = next[j]
- repeat

if j == |T| - 1

- found T in S, res++
- j = next[j]

```
int match(string s, string t) {
    int res = 0;

    for (int i = 0, j = -1; i < s.size(); i++) {
        while (j >= 0 && t[j + 1] != s[i])
        j = next[j];

        if(t[j + 1] == s[i]) j++;

        if(j == t.size() - 1) {
            res++;
            j = next[j];
        }
    }

    return res;
}
```

KMP Algorithm

In every iteration of while loop

- j will decrease
- ≥ -1

In every iteration of for loop, `j` will increase at most 1

So there are at most `O(|S|)` iterations of while loop

Time complexity = `O(|S| + |T|)`

KMP Algorithm

Actually what we go through is just MP algorithm

Knuth improve MP algorithm's next array, become KMP algorithm

```
if T[next[i] + 1] != T[i + 1]
```

- `next[i] = next[next[i]]`
- `skip extra steps`

KMP Algorithm

Constant optimization

No improvement on time complexity

Usually we just code MP algorithm

Other application : HKOJ P002 Power String

- Think how longest prefix suffix is related to this question

Suffix Array

Suffix Array

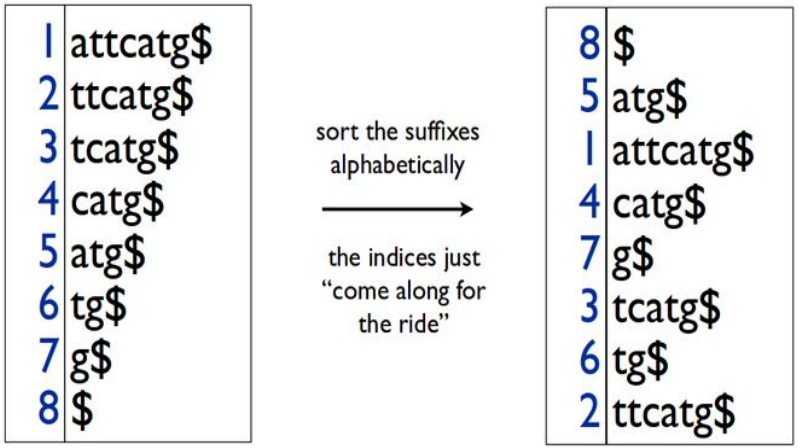
Sorted list of suffixes of string

- Store the indices instead of the actual suffix

Can do string matching in O(|T|log|S|)

Binary search on suffix array

Perform many different query on the string



Suffix Array

Suffix array of “ABRACADABRA”

i	sa[i]	suffix(sa[i])
0	10	A
1	7	ABRA
2	0	ABRACADABRA
3	3	ACADABRA
4	5	ADABRA
5	8	BRA
6	1	BRACADABRA
7	4	CADABRA
8	6	DABRA
9	9	RA
10	2	RACADABRA

Suffix Array

Naive construction

- Sort every suffix
- $O(N \log N)$ comparison
- $O(N)$ per comparison
- Time complexity = $O(N^2 \log N)$

sort `vector<pair<string, int>>`

Suffix Array

O(N) per comparison

- Too slow

We can use the technique of doubling (倍增) to speed up the construction

Suffix Array

Instead of comparing the whole suffix

Comparing the length of power of two

Sort `O(logN)` times

Compare strings by the rank of previous sorting

`rank[i]` = the rank of the first `len` characters of the i^{th} suffix

`len = 1, 2, 4, 8, ….`

Suffix Array

1. Compute `rank[]` by the first `1` character of the i^{th} suffix (ascii)
2. Sort `SA[]` by the first `2` character
3. Compare elements by `rank[]`
4. Update the `rank[]`
5. Sort `SA[]` by the first `4` character
6. So on.....

Finally, we sorted `SA[]` by the whole suffix (`len ≥ N`)

Suffix Array

Compare elements by `rank[]`

- Assume we computed `rank[]` when `len = k`

`len = 2k`, compare `i` and `j` (suffixes start at `i` and `j`)

Compare `(rank[i], rank[i+k])` and `(rank[j], rank[j+k])`

Suffix Array

`len = 2k`, compare `i` and `j` (suffixes start at `i` and `j`)

Compare `(rank[i], rank[i+k])` and `(rank[j], rank[j+k])`

compare `rank[i]` and `rank[j]` = compare `S[i..i+k-1]` and `S[j..j+k-1]`

compare `rank[i+k]` and `rank[j+k]` = compare `S[i+k..i+2k-1]` and `S[j+k..j+2k-1]`

Suffix Array

```
bool cmpSA(int u, int v) {                  //comparing u^th and v^th suffix when len = 2k
    if(RANK[u] != RANK[v]) return RANK[u] < RANK[v];
    else {
        int x, y;
        x = u + k < N ? RANK[u + k] : -1;    //special handle with u + k >= N
        y = v + k < N ? RANK[v + k] : -1;
        return x < y;
    }
}
```

Suffix Array

SA of “ABRACADABRA”

len = 1

sa[i]	S[sa[i]]	rank[sa[i],1]
0	A	0
3	A	0
5	A	0
7	A	0
10	A	0
1	B	1
8	B	1
4	C	2
6	D	3
2	R	4
9	R	4

Suffix Array

SA of “ABRACADABRA”

len = 2

sa[i]	S[sa[i]..sa[i]+1]	rank[sa[i],1]	rank[sa[i]+1,1]
10	A	0	-1
0	AB	0	1
7	AB	0	1
3	AC	0	2
5	AD	0	3
1	BR	1	4
8	BR	1	4
4	CA	2	0
6	DA	3	0
2	RA	4	0
9	RA	4	0

Suffix Array

SA of “ABRACADABRA”

len = 2

sa[i]	S[sa[i]..sa[i]+1]	rank[sa[i], 2]
10	A	0
0	AB	1
7	AB	1
3	AC	2
5	AD	3
1	BR	4
8	BR	4
4	CA	5
6	DA	6
2	RA	7
9	RA	7

Suffix Array

SA of “ABRACADABRA”

len = 4

sa[i]	S[sa[i]..sa[i]+3]	rank[sa[i],2]	rank[sa[i]+2,2]
10	A	0	-1
0	ABRA	1	7
7	ABRA	1	7
3	ACAD	2	3
5	ADAB	3	1
8	BRA	4	0
1	BRAC	4	2
4	CADA	5	6
6	DABR	6	4
9	RA	7	-1
2	RACA	7	5

Suffix Array

SA of “ABRACADABRA”

len = 4

sa[i]	S[sa[i]..sa[i]+3]	rank[sa[i], 4]
10	A	0
0	ABRA	1
7	ABRA	1
3	ACAD	2
5	ADAB	3
8	BRA	4
1	BRAC	5
4	CADA	6
6	DABR	7
9	RA	8
2	RACA	9

Suffix Array

SA of “ABRACADABRA”

len = 16

i	sa[i]	suffix(sa[i])
0	10	A
1	7	ABRA
2	0	ABRACADABRA
3	3	ACADABRA
4	5	ADABRA
5	8	BRA
6	1	BRACADABRA
7	4	CADABRA
8	6	DABRA
9	9	RA
10	2	RACADABRA

Suffix Array

```
void build_SA(string s) {
     N = s.size();
     for (i = 0; i < N; i++) {
          SA[i] = i;
          RANK[i] = s[i];
     }
     for (i = 1; i < N; i *= 2) {
          sort(SA, SA + N, cmpSA);

          TMP[SA[0]] = 0;
          for (j = 1; j < N; j++)
          TMP[SA[j]] = TMP[SA[j - 1]] + cmpSA(SA[j - 1], SA[j]);
          for (j = 0; j < N; j++)
          RANK[j] = TMP[j]; // updating the rank[]
     }
}
```

Suffix Array

No. of sorting = $O(\log N)$

No. of comparison per sorting = $O(N \log N)$

Time complexity per comparison = $O(1)$

Overall time complexity = $O(N \log^2 N)$

Space complexity = $O(N)$

Suffix Array

Observe that range of `rank[] < N`

replace std :: sort with radix sort

`O(N log N)` → `O(N)`

Overall time complexity `O(N log`^{2} `N)` → `O(N log N)`

O(N) build Suffix array

- http://gagguy.blogspot.com/2012/08/linear-time-suffix-array-dc3.html

Suffix Array

Only having the SA is not really helpful

Calculate another array `lcp[]`

- `lcp[i]` = longest common prefix of `suffix(sa[i])` and `suffix(sa[i-1])`

longest common prefix

e.g. ABCDE ABEDC, lcp = 2

Suffix Array

Observation

- If `lcp[rank[i]]=h` → `lcp[rank[i + 1]] ≥ h-1`

Calculate `lcp[]` in the order of `rank[0]`, `rank[1]`, `rank[2]`, ...

- Position that contains 0, 1, 2, ... in the SA[] array

Suffix Array

E.g. S = "BCEABCDABCEB"

`SA[rank[7]]` = ABCEB

`lcp[rank[7]]` = 3, i.e. there exist suffix = ABCD……

`SA[rank[7 + 1]]` = BCEB

`lcp[rank[7 + 1]]` ≥ 2

p.s. `lcp[rank[7+1]]` may not be 2, in this case `lcp[rank[7+1]]`=3

Suffix Array

```
for (i = 0; i < N; i++) {
        if (RANK[i] == 0) continue;
        int t = SA[RANK[i] - 1];
        h = max(0, h - 1);
        while (i + h < N && t + h < N) {
            if (s[i + h] != s[t + h]) break;
            h++;
        }
        LCP[RANK[i]] = h;
}
```

Suffix Array

For each loop

- h will decrease 1
- h won't exceed N

Time complexity = O(N)

```
for (i = 0; i < N; i++) {
    if (RANK[i] == 0) continue;
    int t = SA[RANK[i] - 1];
    h = max(0, h - 1);
    while (i + h < N && t + h < N) {
        if (s[i + h] != s[t + h]) break;
        h++;
    }
    LCP[RANK[i]] = h;
}
```

Suffix Array

We can calculate `lcp(suffix(i),suffix(j))` with `lcp[]`

- or LCP of any two substring

`lcp(suffix(i),suffix(j)) = min{lcp[rank[i]+1],lcp[rank[i]+2],…,lcp[rank[j]]}`

- Assume `rank[i] ≤ rank[j]`
- = minimum value in lcp[rank[i] + 1..rank[j]]

Use data structure for RMQ

- Segment tree, Sparse Table

Suffix Array

lcp("ABRA", "ADABRA")

= query(2, 4)

= 1

i	sa[i]	suffix(sa[i])	LCP[i]
0	10	A	0
1	7	ABRA	1
2	0	ABRACADABRA	4
3	3	ACADABRA	1
4	5	ADABRA	1
5	8	BRA	0
6	1	BRACADABRA	3
7	4	CADABRA	0
8	6	DABRA	0
9	9	RA	0
10	2	RACADABRA	2

Suffix Array

Given a string S, find the number of distinct substring

ababa

Ans = 9 {"a","b","ab","ba","aba","bab","abab","baba","ababa"}

Exhaust all substrings, count the number of distinct substrings

Time complexity: $O(|S|^2)$

Suffix Array

Consider prefix of each suffix

ababa

{'a'}

ans = 1

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Suffix Array

Consider prefix of each suffix

ababa

{‘a’,‘ab’,‘aba’}

LCP[i] = 1

- don’t add ‘a’

ans = 3

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Suffix Array

Consider prefix of each suffix

ababa

{‘a’,‘ab’,‘aba’,’abab’,’ababa’}

LCP[i] = 3

- don’t add ‘a’,’ab’,’aba’

ans = 5

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Suffix Array

Consider prefix of each suffix

ababa

{‘a’,‘ab’,‘aba’,’abab’,’ababa’,
‘b’,‘ba’}
LCP[i] = 0
ans = 7

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Suffix Array

Consider prefix of each suffix

ababa

{‘a’,‘ab’,‘aba’,’abab’,’ababa’,
‘b’,‘ba’,’bab’,’baba’}

LCP[i] = 2

- don’t add ‘b’,’ba’

ans = 9

i	sa[i]	suffix(sa[i])	LCP[i]
0	4	a	0
1	2	aba	1
2	0	ababa	3
3	3	ba	0
4	1	baba	2

Suffix Array

```
int res = n - sa[0];
for (int i = 1; i < n; i++) res += (n - sa[i]) - LCP[i];
return res;
```

Time complexity: $O(N)$

Problem: https://www.spoj.com/problems/DISUBSTR/

Suffix Array

Other application

- longest repeated substring
  - overlap / non-overlap
- longest common substring
- longest palindromic substring

Useful application: binary search on answer

- Group indices by lcp

Something else

- Z algorithm
  - String matching in O(N)
- Manacher algorithm
  - Longest palindromic substring in O(N)
- Suffix tree
- Palindromic Tree
- Aho-Corasick algorithm
- Suffix Automation

Exercise

Trie

- IOI08 Type Printer

KMP

- HKOJ 01002
- HKOJ M0932
- HKOJ P002
- NOI14 動物園

Suffix array

- HKOI M1762
- POJ 2774
- NOI15 品酒大會

Q & A