String Algorithms

Chow Kwan Ting Jeremy

30/3/2019

Content

- Trie
- KMP Algorithm
- Hashing
- Suffix Array
- Aho-Corasick algorithm
- Something else...

Prerequisite

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

Prerequisite

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”
- string $x = x_1x_2\ldots\, x_{|x|}$ is lexicographically larger than string $y = y_1y_2\ldots\, y_{|y|}$ if either $|x| > |y|$ and $x_1 = y_1, x_2 = y_2, \ldots, x_{|y|} = y_{|y|}$, or there exists such number r ($r < |x|, r < |y|$) that $x_1 = y_1, x_2 = y_2, \ldots, x_r = y_r$ and $x_{r+1} > y_{r+1}$

Prerequisite

Substring

- contiguous sequence of characters within a string
- ABCDE ABCDE ABCDE

Prefix

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

Prerequisite

Suffix

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

Subsequence

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

Prerequisite

Palindrome

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

Trie

- Not a formal word
- come from the word retrieval

- Pronounce as “try”
- Tree
- Dictationary of a set of strings
- support quick insert and lookup

Trie

- Each node represents a character
  - except root
- Also represents a word / prefix of a string
  - obtained by concatenating characters from root to this node

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

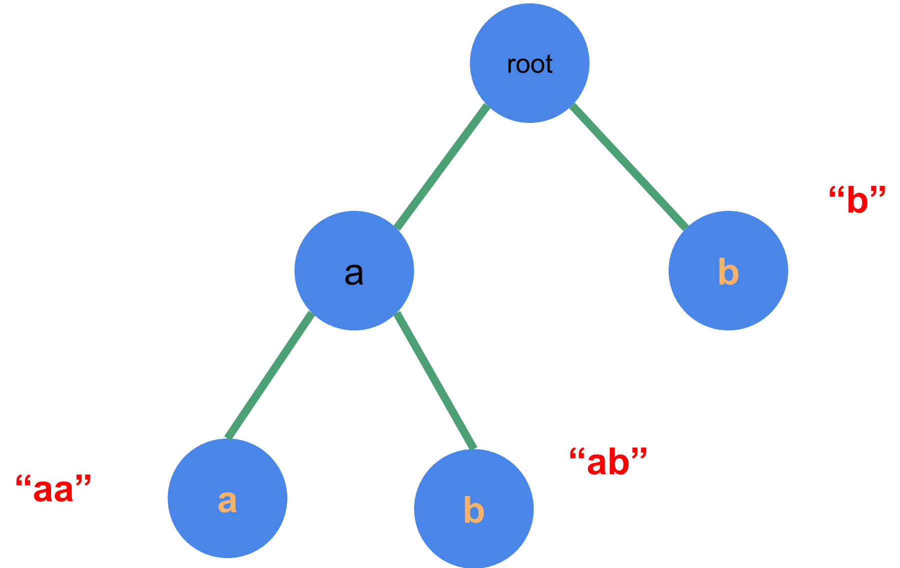
Trie

```
struct TrieNode {
    int children[ALPHABET_SIZE];      //edges
    bool isword;           // true if the node is the end of a word
}Trie[SIZE];
```

- APLHABET_SIZE depends of your input
  - lowercase letter -> 26
  - lowercase + uppercase letter -> 52
  - ascii -> 128
  - we assume the input is all lowercase letter in the following slides

Trie

- Trie of {“aa”, “ab”, “b”}



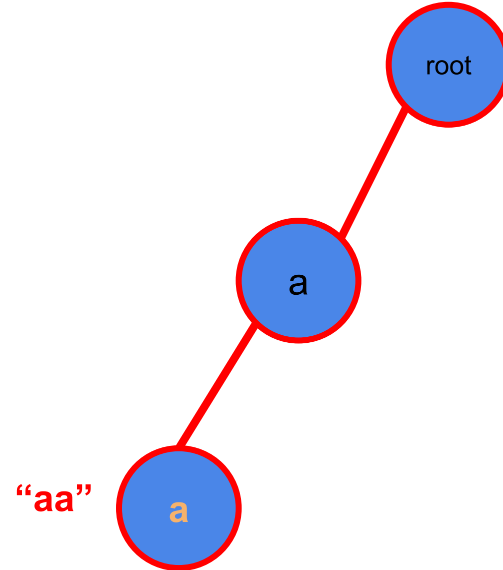
Trie

```
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

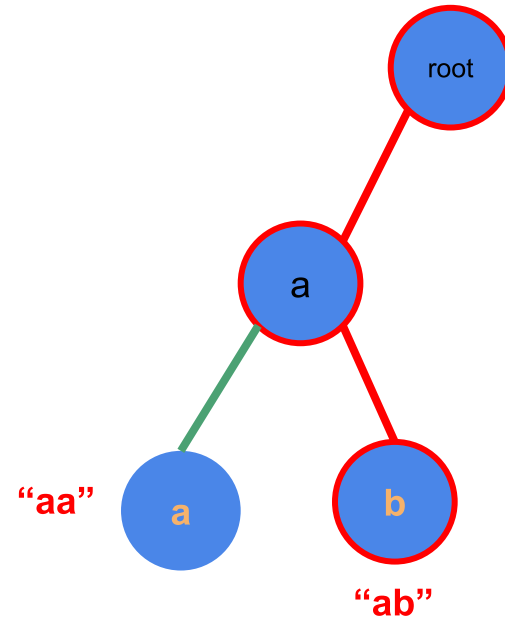
Trie

Insert “aa”



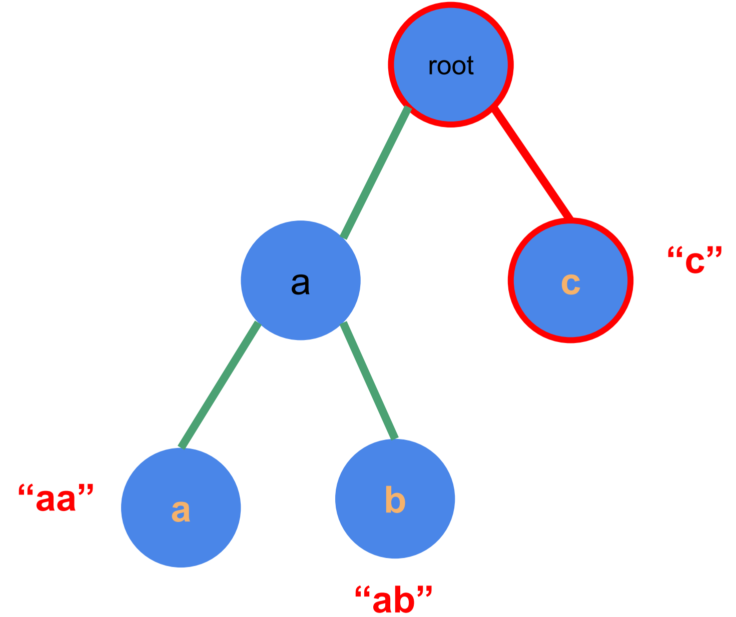
Trie

Insert “ab”



Trie

Insert “c”

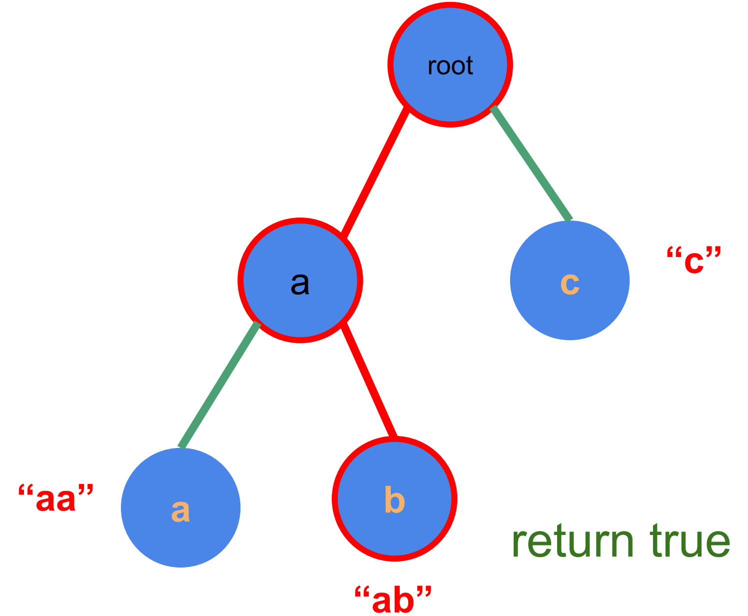


Trie

```
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;
}
```

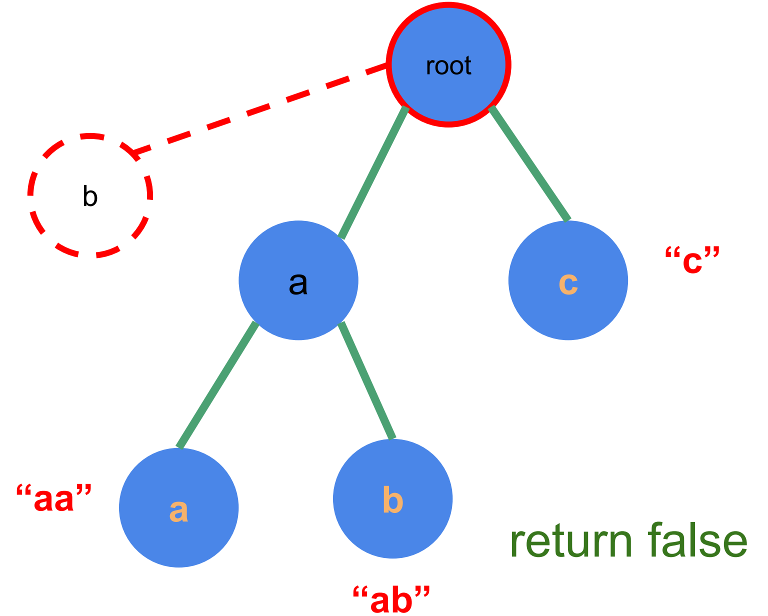
Trie

Lookup “ab”



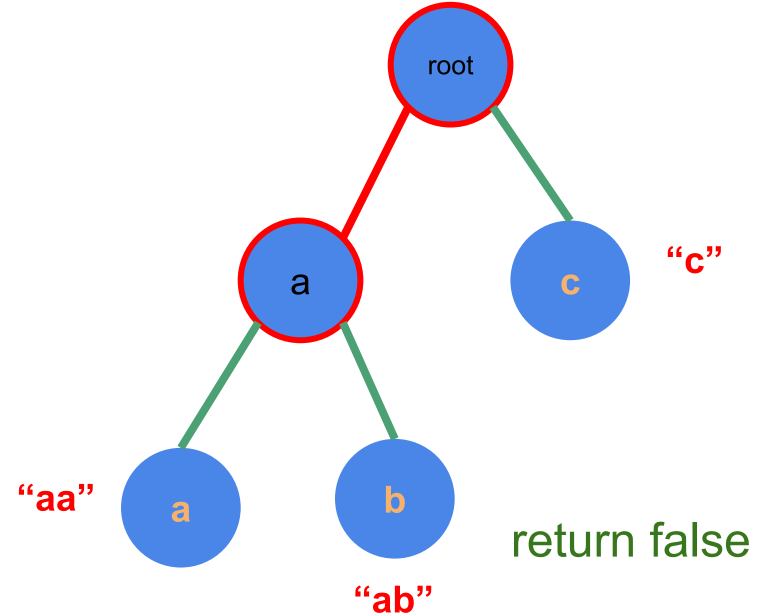
Trie

Lookup “b”



Trie

Lookup “a”



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(ALPHABETSIZE * N)

Trie

- You can tranform an integer into a binary string with length of 31
- Store into trie
- Act as an Binary Search Tree

- Maybe useful to solve problem about xor
- E.g. Maximum xor-sum subarray

Hashing

- Hashing in number
    - number -> number
    - taught in ds(ii) hash table
- Hashing in string
    - string -> number
    - compare number instead of string

Hashing

- Rolling hash
  - common hash function

$$H = (c_0a^{k-1}+c_1a^{k-2}+c_2a^{k-3}+\ldots+c_{k-1}a^0) \bmod p$$

- H = hash value
- c_i = ascii value of character
- a = some constant
- p = some constant, usually a prime
  - use unsigned long long -> $p = 2^{64} - 1$

Hashing

"HKOI"

c = 7, 10, 14, 8 // - 'A'

a = 26

p = 64997

$H = (7*26^3+10*26^2+14*26+8) \bmod 64997$

$= 130164 \bmod 64997$

$= 170$

Hashing

"GO"

c = 6, 14 // - ‘A’

a = 26

p = 64997

H = (6*26+14) mod 64997

= 170 mod 64997

= 170

Hashing

- H(“HKOI”) = 170 = H(“GO”)
- “HKOI” <> “GO”
- collision occur

- Hash function is not an injective function
- One hash value may represents multiple strings

Hashing

- Solution 1: pick a better constant to reduce the probability of collision
  - p -> prime number
- Solution 2 : Double hashing
- Solution 3 : Give up hashing
  - Use exact algorithm e.g. KMP algorithm

Hashing

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

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

Hashing

- Get hash value of a substring in O(1)
- Let $H_i = (c_0a^i+c_1a^{i-1}+c_2a^{i-2}+...+c_ia^0) \bmod p$
  - partial sum array

- Hash(l, r) = $(H_r - H_{l-1} * pow(a, r - l + 1)) \bmod p$
- String matching

KMP

- String matching problems
  - deteremine whether one or serveral strings are found within a large string

- Knuth–Morris–Pratt algorithm
- Solve single pattern matching problem

KMP

- HKOJ 01002 A Counting Problem

- Given 2 string S, T
- |S| <= 1000, |T| <= 200
- Find the occurences of T in S

KMP

S = abcdefabcghiabcabcjklmnlabcw

T = abc

no. of occurences = 5

abcdefabcghiabcabcjklmnlabcw

p.s. allow overlap

KMP

- Naive Solution
- for each position i in S
  - check whether S[i..i + T.length() - 1] == T

- Time complexity = O(|S| * |T|)
- Enough to get AC on HKOJ

KMP

Occurence = 1

abcdefabcghiabcabcjklmnlabcw

abc

KMP

Occurence = 1

abcdefabcghiabcabcjklmnlabcw

abc

KMP

Occurence = 1

abcdefabcghiabcabcjklmnlabcw

abc

KMP

Occurence = 1

abcdefabcghiabcabcjklmnlabcw

def

2000 YEARS
LATER

KMP

Occurence = 5

abcdefabcghiabcabcjklmnlabcw

def

KMP

- What if the constraint is bigger
  - $|S|, |T| <= 10^5$
- Too slow
- TLE

KMP

- Compare whole string is too slow
    - $O(|T|)$
- Compare hash value of string instead
    - $O(1)$
- Maintain hash value with sliding window

- Time complexity = $O(|S| + |T|)$
- Collisions may result in WA or TLE

KMP

- Need to recalcuate everything after a mismatch
- make use of the information in the previous match
- Avoid rematching

KMP

- Match T with S
- Do the naive matching first

- We get a mismatch
- S[5] != T[5]

ABABABABACB
ABABACB

KMP

ABABABABACB
ABABACB

ABABA

- Already match “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”

KMP

- We already know S[0..4] = T[0..4] = “ABABA”
- Also, T[0..2] = T[2..4]
- Therefore, S[2..4] = T[0..2]
- Don’t need to rematch this part

ABABABABACB
ABABACB

KMP

- Why skip checking S[1..1+T.length()-1] = T?
- Because S[0..4] = T[0..4] = “ABABA”
- T[0..3] != T[1..4]
- T[0..3] != S[1..4]

ABABABABACB
  ABABACB

- We can know that we won’t be able to match T starting at position 1 of S

KMP

- How to find the longest prefix suffix of a string?
  - By dynamic programming

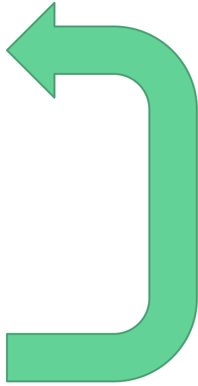
- next[i] = length(longest prefix suffix of T[0..i]) - 1
  - where T[0..next[i]] = T[i - next[i]..i] and next[i] < i

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

KMP

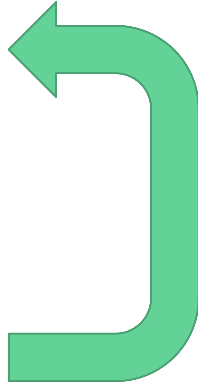
- We know next[i - 1] and want to calculate next[i]

- let 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

- 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]

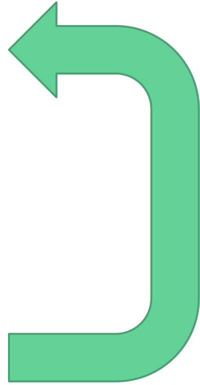
need to find smaller j such that T[0..j] = T[4-j..4]

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

- 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[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

- 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

ABABABABACB
ABABACB

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

KMP

ABABABABACB
ABABACB

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

KMP

ABABABABACB

ABABACB

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

KMP

ABABABABACB
ABABACB

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

KMP

ABABABABBACB

ABABACB

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

KMP

ABABABABACB
ABABACB

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

KMP

```
bool 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

- In every iteration of whole loop
    - j will decrease
    - >= -1
- In every iteration of for loop
- j will increase at most 1
- So there are at most |S|/|T| iteration of whole loop
- Time complexity = O(|S| + |T|)

KMP

- 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]]

KMP

- No improvement on time complexity
- Ignore it and just code MP algorithm

- Other application : P002 Power String

Suffix array

- Sorted list of suffixes of string
  - Store the indices instead of the string

- Do the string matching in O(|T|log|S|)
  - Binary search
- Perform many different query on the string

Suffix array

SA 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(NlogN) comparison
- O(N) pre comparison
- Time complexity = $O(N^2 \log N)$

- sort vector<pair<string, int> >

Suffix array

- O(N) per comparison
- 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 ith suffix

len = 1, 2, 4, 8, ...

Suffix array

1. compute rank[] by the first 1 character of the ith 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.......

Suffix array

- Compare elements by rank[]
  - Assume we computed rank[] when len = k
- Len = 2k, compare i and j (suffix start at i and j)

- Compare pair(rank[i], rank[i + k]) and pair(rank[j], rank[j + k])
  - compare rank[i] and rank[j] = comparing S[i..i+k-1] and S[j..j+k-1]
  - compare rank[i+k] and rank[j+k] = comparing S[i+k..i+2k-1] and S[j+k..j+2k-1]

Suffix array

```
bool cmpSA(int u, int v) { //comparing uth and vth 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(logN)
- No. of comparison per sorting = O(NlogN)
- Time complexity per comparison = O(1)

- Overall time complexity = O(NlogNlogN)
- Space complexity = O(N)

Suffix array

- Observe that range of rank[] < N
- replace std :: sort with radix sort
- O(NlogN) -> O(N)

- Overall time complexity O(NlogNlogN) -> O(NlogN)

- O(N) build Suffix array
  - http://gagguy.blogspot.com/2012/08/linear-time-suffix-array-dc3.html

Suffix array

- Not really useful for just building the SA
- Calcuate 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

- Calculate lcp[] in the order of rank[0], rank[1], rank[2], ….
  - Position that contains 0, 1, 2, …. in the SA[] array

- Observation
  - If lcp[rank[i]] = h
  - lcp[rank[i + 1]] >= h - 1

Suffix array

E.g. S = “BCEABCDABCEB”

SA[rank[i]] = ABCEB

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

SA[rank[i + 1]] = BCEB

lcp[rank[i + 1]] >= 2

p.s. lcp[rank[i + 1]] may not be 2, in this case lcp[rank[i + 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 any two substring's LCP

- lcp(suffix(i), suffix(j)) = min{lcp[rank[i] + 1], lcp[rank[i] + 2], … , lcp[rank[j]]}
  - Assume rank[i] <= rank[j]
- = minmum value in [rank[i] + 1..rank[j]] in lcp[]
- = 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

- Other application
    - longest repeated substring
        - overlap / non-overlap
    - longest common substring
    - longest palindromic substring
    - number of distinct substring
- Binary search on answer
- Group indices by lcp

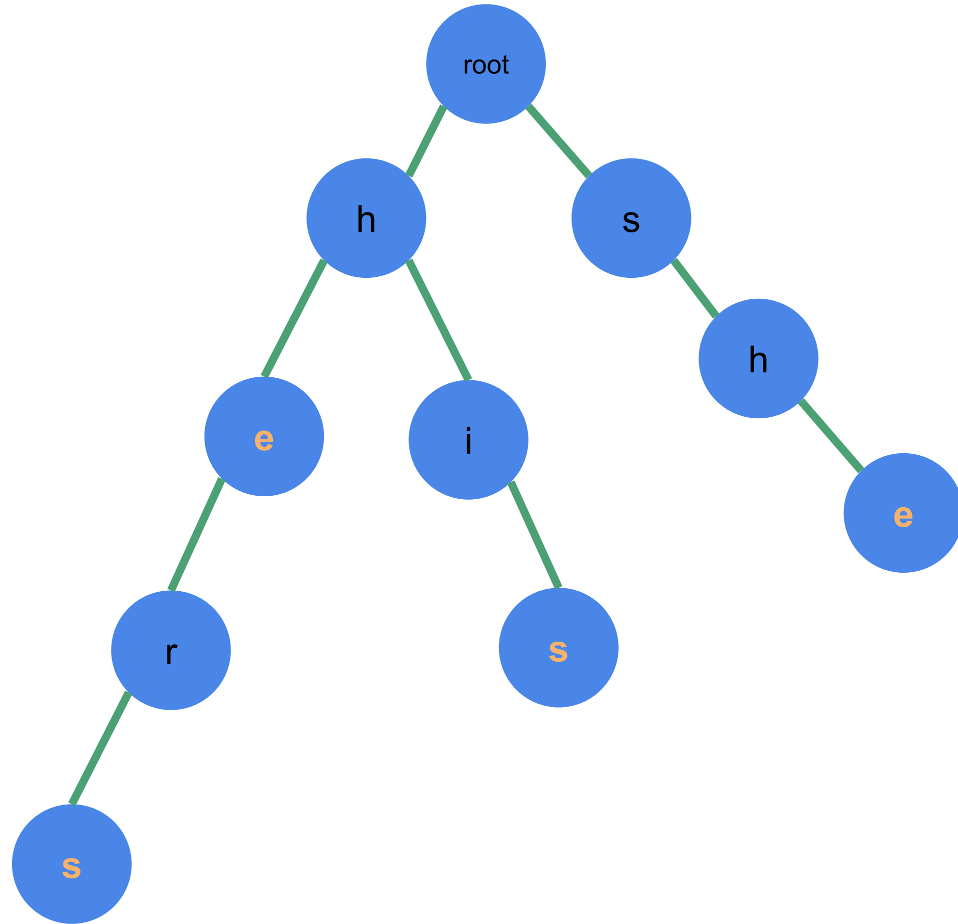
Aho-Corasick algorithm

- ~ Trie + KMP
- Multi-pattern
  - M = no. of pattern
  - L = average length

- KMP : O(M * (L + |S|))
- AC automation : O(M * L + |S|)

Aho-Corasick algorithm

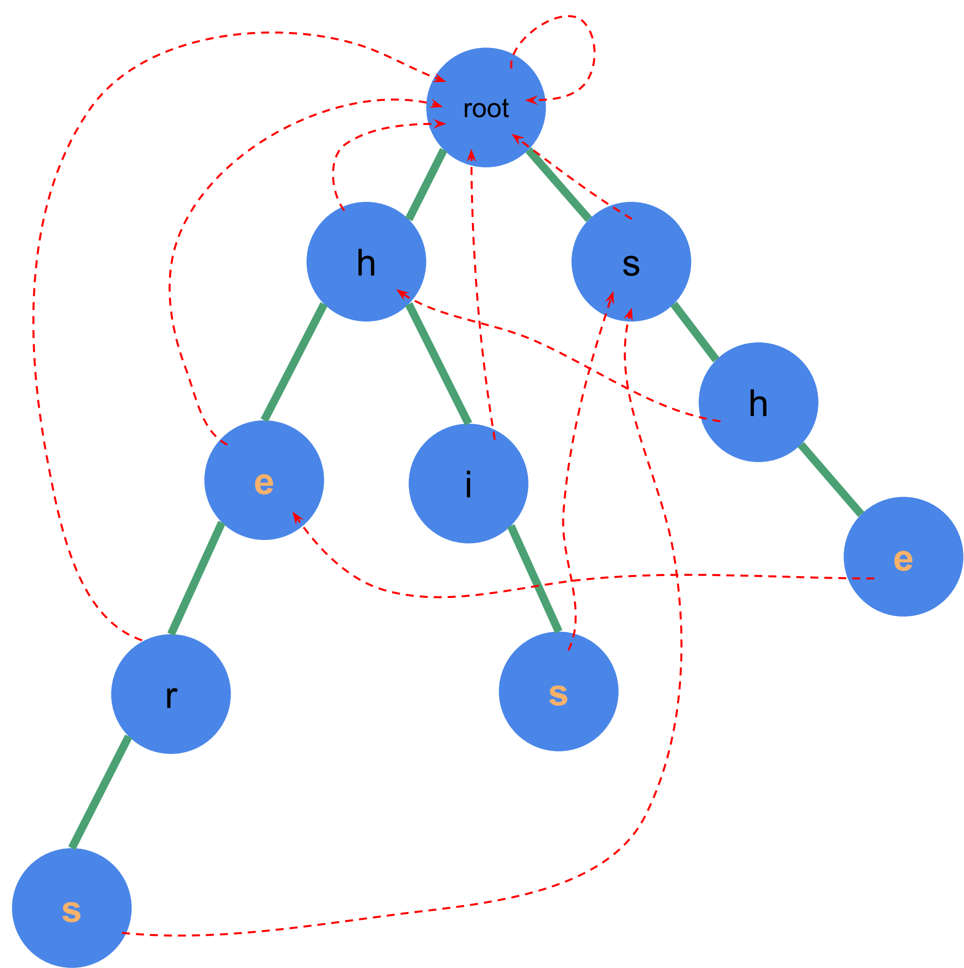
- Build a trie with the patterns
- Patterns = “he”, “she”, “his”, “hers”



Aho-Corasick algorithm

- Build a next[] (failure link)
- Patterns = “he”, “she”, “his”, “hers”

- Higher depth to lower depth
- Constructed by BFS
- Just like what we do in KMP



Aho-Corasick algorithm

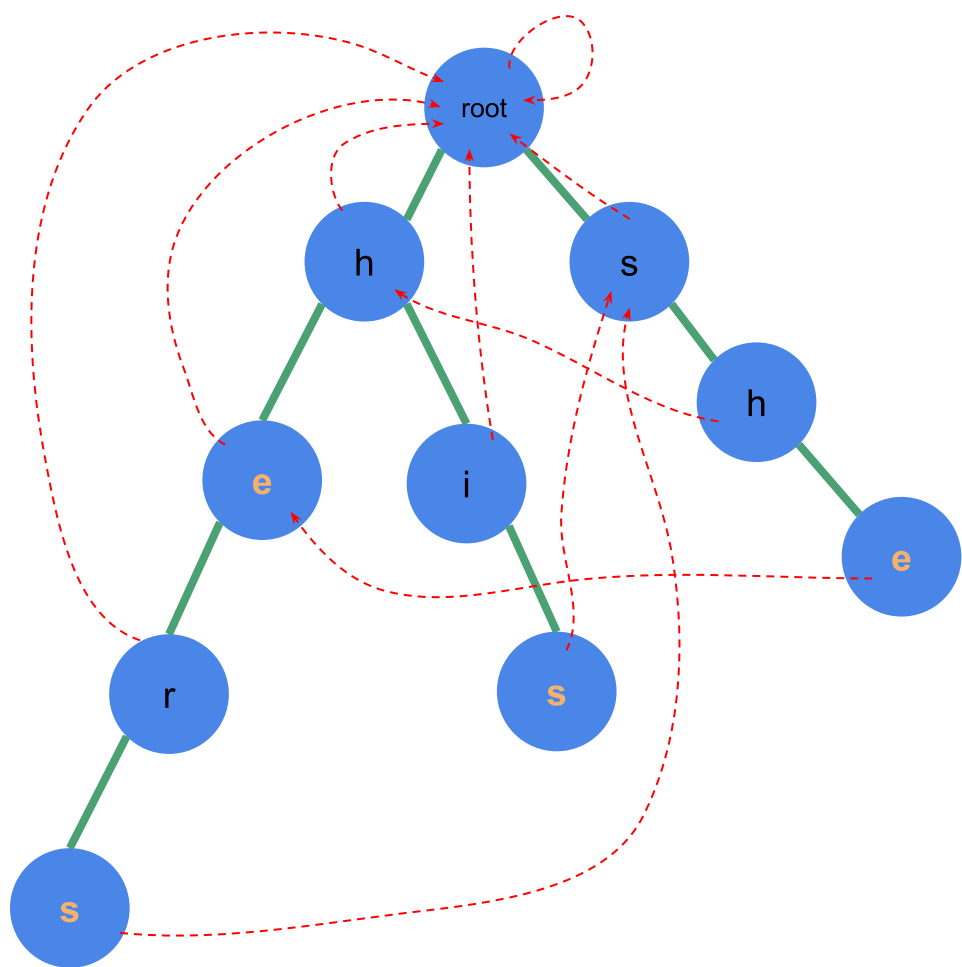
- last[] = previous node along failure link s.t. it is a word
- Add extra edge
  - No need to go back many time
  - Delete while loop

```
void getFail() {
	queue <int> q;
	f[0] = 0;
	for (int c = 0; c < sigma_size; c++) {
		int u = ch[0][c];
		if (u) { f[u] = 0; q.push(u); last[u] = 0; }
	}

	while (q.size()) {
		int r = q.front(); q.pop();
		for (int c = 0; c < sigma_size; c++) {
			int u = ch[r][c];
			//if (!u) continue;
			if (!u) { ch[r][c] = ch[f[r]][c]; continue; }
			q.push(u);
			int v = f[r];
			//while (v && !ch[v][c]) v = f[v]
			f[u] = ch[v][c];
			last[u] = val[f[u]] ? f[u] : last[f[u]];
		}
	}
}
```

Aho-Corasick algorithm

- Reach a node <> match 1 pattern
- Reach “she”
  - Match “she”
  - Match “he”
- Need to go back via last[]



Aho-Corasick algorithm

- Set value to 0 after reach the node
  - Avoid double count
  - Number of matched pattern

- Number of occurence
  - res += cnt[i]
  - cnt[last[i]] += cnt[i]

```
int find(char* s) {
    //number of matched pattern in string S


    int n = strlen(s);
    int j = 0;
    int res = 0;

    for (int i = 0; i < n; i++) {
        int c = idx(s[i]);
        j = ch[j][c];

        int pos = j;
        if (!val[j]) pos = last[j];

        while (val[pos]) {
            res += val[pos];
            val[pos] = 0;
            pos = last[pos];
        }
    }

    return res;
}
```

Aho-Corasick algorithm

- Often combine with dp
- node on AC automation = state in dp
- edge on AC automation = transition in dp

- `for (int k = 0; k < 26; j++) dp[i + 1][ch[j][k]] += dp[i][j]`

Aho-Corasick algorithm

POJ 2778 DNA Sequence

- Given m pattern
- Find the number of length-k string such that it doesn’t contain any those pattern
- last[i] != 0 || val[i] != 0   => Dangerous node
    - Visit = not valid string
- = Number of length-k path start at root s.t. it doesn’t pass any dangerous node
- Use matrix to speed up

Something else

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

Aho-Corasick algorithm

- HDU 2222
- HDU 3065
- POJ 2778