

# Algorithms on Strings

Pham Quang Dung and Do Phan Thuan

Computer Science Department, SoICT,  
Hanoi University of Science and Technology.

February 23, 2017

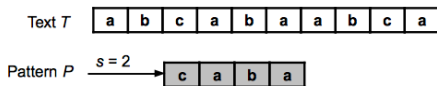
1 String searching

2 Tries

3 Suffix Tries

4 Suffix Trees/Arrays

- String matching problem: find one or all occurrences of a pattern in a given text
- Applications
  - ▶ information retrieval
  - ▶ Text editors
  - ▶ computational biology (DNA sequences)
- Formal formulation
  - ▶ A text is an array  $T[1..n]$  and a pattern is an array  $P[1..m]$  ( $m \neq n$ )
  - ▶  $T[i], P[j] \in$  a finite alphabet  $\Sigma$  (e.g.,  $\Sigma = \{0, 1\}$  or  $\Sigma = \{a, \dots, z\}$ )
  - ▶ We say that pattern  $P$  **occurs with shift  $s$**  in  $T$  if  $0 \leq s \leq n - m$  and  $T[s + 1..s + m] = P[1..m]$



# String searching algorithms

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

---

**Algorithm 1:** NaiveSM( $P, T$ )

---

```
foreach  $s = 0..n-m$  do  
     $i \leftarrow 1$ ;  
    while  $i \leq m$  and  $P[i] = T[i + s]$  do  
         $i \leftarrow i + 1$ ;  
    if  $i \geq m$  then  
        Output( $s$ );
```

---

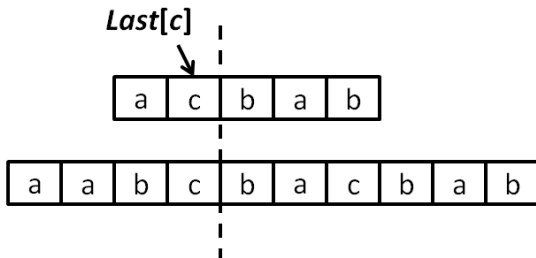
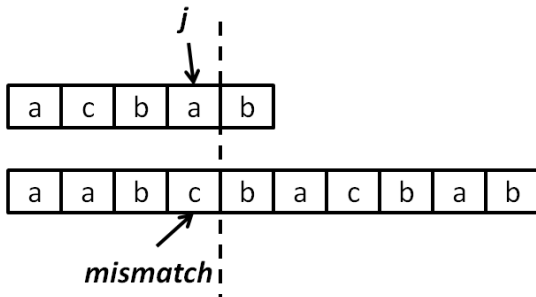
# Boyer-Moore algorithm

- Left to right shift
- Right to left scan
- Use information gained by preprocessing  $P$  in order to skip as many alignment as possible
- Bad character shift rule
  - ▶  $last[c]$ : the right-most occurrence of  $c$  in  $P$
  - ▶ When mismatch: shift  $P$  right by  $\max\{j - last[c], 1\}$  where  $j$  is the position of mismatch character of  $P$

# Boyer-Moore algorithm



SAMSUNG



# Boyer-Moore algorithm

```
void computeLast(){
    for(int c = 0; c < 256; c++){
        last[c] = 0;
    }
    for(int i = m; i >= 1; i--){
        if(last[P[i]] == 0)
            last[P[i]] = i;
    }
}

void BoyerMoore(){
    int s = 0;
    while(s <= n-m){
        int j = m;
        while(j > 0 && T[j+s] == P[j])    j--;
        if(j == 0){
            Output(s);
            s = s + 1;
        }else{
            int k = last[T[j+s]];

```



- Convert the pattern  $P[1..m]$  to a number:

$$p = P[1] * d^{m-1} + P[2] * d^{m-2} + \dots + P[m] * d^0$$

where each character  $P[i]$  is viewed as a nonnegative integer  $< d$ , and  $d$  is the size of the alphabet

- Using Horner's rule:

$$p = P[m] + d * (P[m-1] + d * (\dots + d * P[1]) + \dots)$$

- Convert  $T[s+1..s+m]$  to the integer

$$t_s = T[s+1] * d^{m-1} + \dots + T[s+m]$$

- **Note:**  $t_{s+1}$  can easily be computed from  $t_s$  as follows:

$$t_{s+1} = (t_s - T[s+1] * d^{m-1}) * d + T[s+m+1]$$

- Drawback: when  $m$  is large, then the computation of  $p$  and  $t_s$  does not take constant time
- Solution: Compute  $p$  and  $t_s$  modulo a suitable number  $q$ 
  - ▶ Still problem:  $p \equiv t_s \pmod{q}$  does not mean that  $p = t_s$ , we have to check  $P[1..m]$  and  $T[s + 1..s + m]$  character by character to see if they are really identical
- Worst-case time is  $\mathcal{O}(mn)$  where  $P = a^m$  and  $T = a^n$

# Knuth-Morris-Pratt (KMP) algorithm

- Comparison: from left to right
- Shift: more than one position
- Preprocessing the pattern
  - ▶ Pattern  $P[1..m]$
  - ▶  $\pi[q]$  is the length of the longest prefix of  $P[1..q]$  which is also the **strictly** suffix of  $P[1..q]$

## Example

$q$	1	2	3	4	5	6	7	8	9	10
$P[q]$	a	b	a	b	a	b	a	b	c	a
$\pi[q]$	0	0	1	2	3	4	5	6	0	1

# Knuth-Morris-Pratt (KMP) algorithm - preprocessing



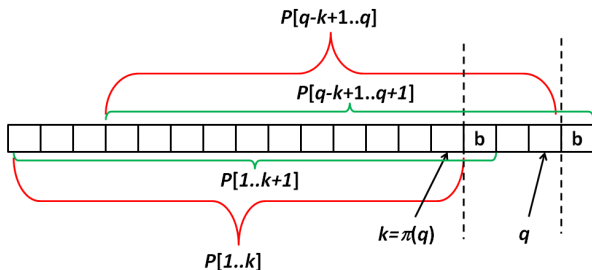
SAMSUNG

```
void computePI(){
    pi[1] = 0;
    int k = 0;
    for(int q = 2; q <= m; q++){
        while(k > 0 && P[k+1] != P[q])
            k = pi[k];
        if(P[k+1] == P[q])
            k = k + 1;
        pi[q] = k;
    }
}
```

# Knuth-Morris-Pratt (KMP) algorithm - preprocessing

Denote  $k = \pi[q]$

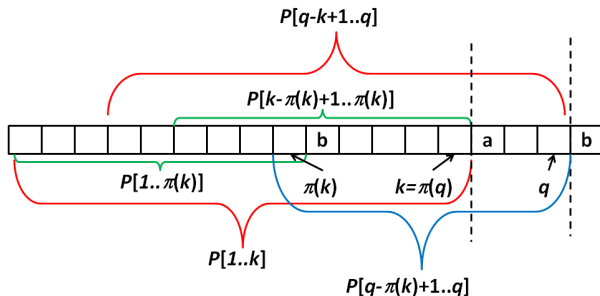
- If  $P[q+1] = P[k+1]$ , then  $\pi[q+1] = \pi[q] + 1$



# Knuth-Morris-Pratt (KMP) algorithm - preprocessing

Denote  $k = \pi[q]$

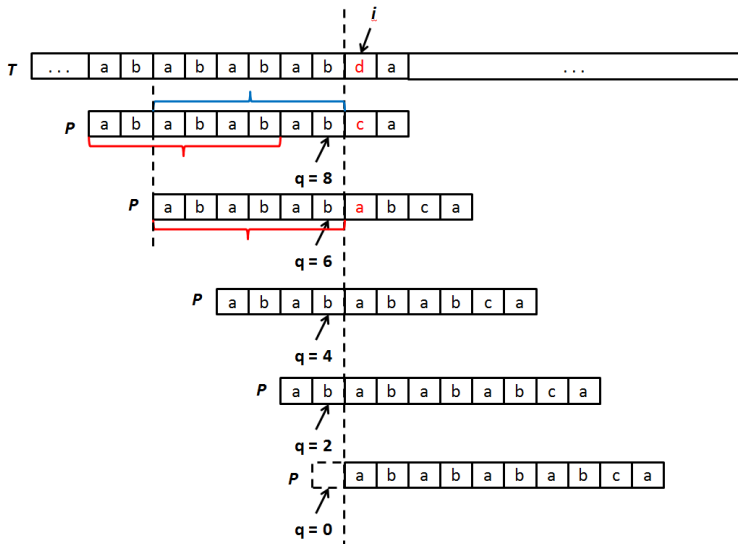
- if  $P[q + 1] \neq P[k + 1]$  and  $P[q + 1] = P[\pi[k] + 1] = b$ :
  - ▶  $P[1..k] = P[q - k + 1..q] \Rightarrow P[k - \pi[k] + 1..k] = P[q - \pi[k] + 1..q]$
  - ▶ Moreover,  $P[k - \pi[k] + 1] = P[1.. \pi[k]]$ , so  
 $P[1.. \pi[k]] = P[q - \pi[k] + 1..q]$ ,
  - ▶ Hence  $P[1.. \pi[k] + 1] = P[q - \pi[k] + 1..q + 1]$ , this means  
 $\pi[q + 1] = \pi[k] + 1$



# Knuth-Morris-Pratt (KMP) algorithm

```
void kmp(){
    int q = 0;
    for(int i = 1; i <= n; i++){
        while(q > 0 && P[q+1] != T[i]){
            q = pi[q];
        }
        if(P[q+1] == T[i])
            q++;
        if(q == m){
            cout << "match at position " << i-m+1 << endl;
            q = pi[q];
        }
    }
}
```

# Knuth-Morris-Pratt (KMP) algorithm





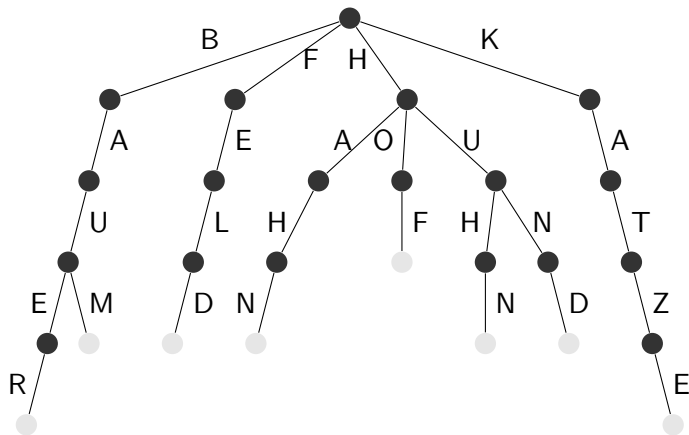
- 1 String searching
- 2 Tries
- 3 Suffix Tries
- 4 Suffix Trees/Arrays

- We often have sets (or maps) of strings
- Insertions and lookups usually guarantee  $O(\log n)$  comparisons
- But string comparisons are actually pretty expensive...
- There are other data structures, like tries, which do this in a more clever way

# Tries



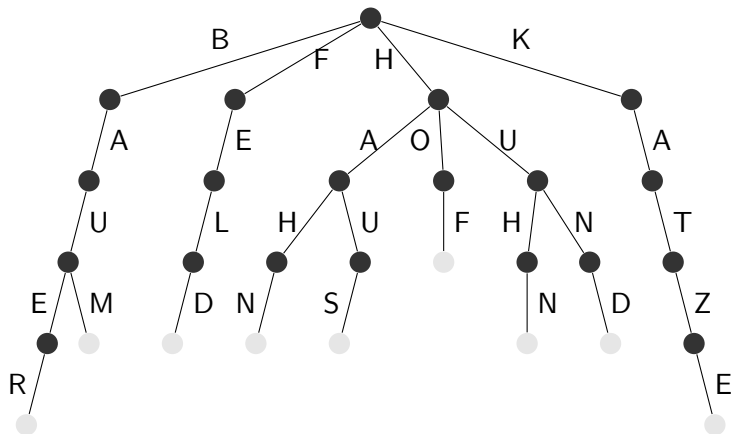
**SAMSUNG**



# Tries



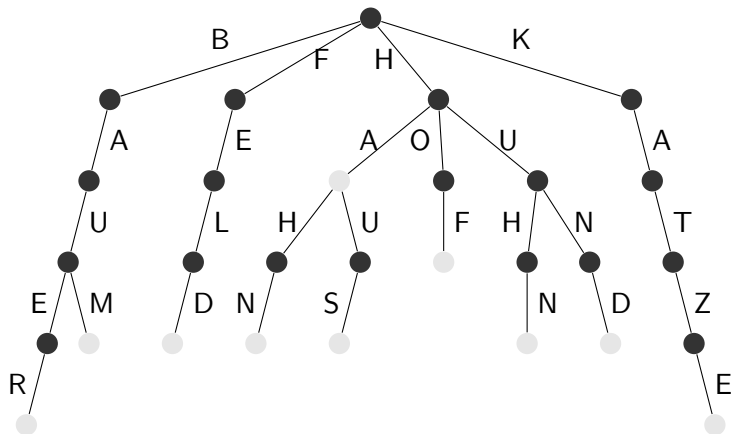
SAMSUNG



# Tries



SAMSUNG



```
struct node {  
    node* children[26];  
    bool is_end;  
  
    node() {  
        memset(children, 0, sizeof(children));  
        is_end = false;  
    }  
};
```

```
void insert(node* nd, char *s) {  
    if (*s) {  
        if (!nd->children[*s - 'a'])  
            nd->children[*s - 'a'] = new node();  
  
        insert(nd->children[*s - 'a'], s + 1);  
    } else {  
        nd->is_end = true;  
    }  
}
```

```
bool contains(node* nd, char *s) {  
    if (*s) {  
        if (!nd->children[*s - 'a'])  
            return false;  
  
        return contains(nd->children[*s - 'a'], s + 1);  
    } else {  
        return nd->is_end;  
    }  
}
```



# Tries

```
node *trie = new node();  
  
insert(trie, "banani");  
  
if (contains(trie, "banani")) {  
    // ...  
}
```

- Time complexity?
- Let  $k$  be the length of the string we're inserting/looking for
- Lookup and insertion are both  $O(k)$
- Also very space efficient...

1 String searching

2 Tries

3 Suffix Tries

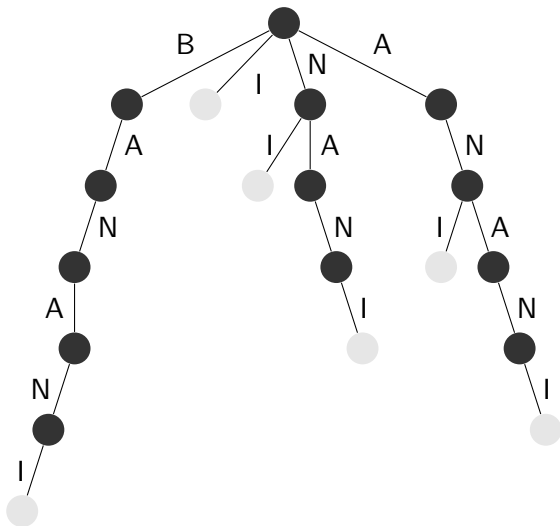
4 Suffix Trees/Arrays

- Say we're dealing with some string  $S$  of length  $n$
- Let's insert all suffixes of  $S$  into a trie
- $S = \text{banani}$ 
  - ▶ `insert(trie, "banani");`
  - ▶ `insert(trie, "anani");`
  - ▶ `insert(trie, "nani");`
  - ▶ `insert(trie, "ani");`
  - ▶ `insert(trie, "ni");`
  - ▶ `insert(trie, "i");`

# Suffix tries



SAMSUNG

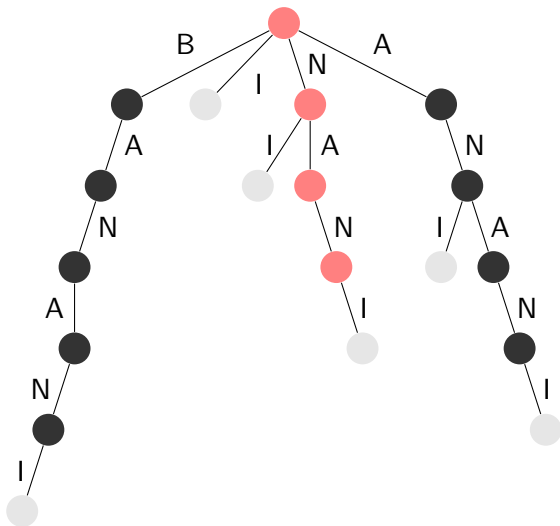


- There are a lot of cool things we can do with suffix tries
- Example: String matching
- If a string  $T$  is a substring in  $S$ , then (obviously) it has to start at some suffix of  $S$
- So we can simply look for  $T$  in the suffix trie of  $S$ , ignoring whether the last node is an end node or not
- This is just  $O(m)$ ...

# Suffix tries



SAMSUNG



- String matching is fast if we have the suffix trie for  $S$
- But what is the time complexity of suffix trie construction?
- There are  $n$  suffixes, and it takes  $O(n)$  to insert each of them
- So  $O(n^2)$ , which is pretty slow
- Can we do better?
- There can be up to  $n^2$  nodes in the graph, so this is actually optimal...



- 1 String searching
- 2 Tries
- 3 Suffix Tries
- 4 Suffix Trees/Arrays

- There exists a compressed version of a suffix trie, called a suffix tree
- It can be constructed in  $O(n)$ , and has all the features that suffix tries have
- But the  $O(n)$  construction algorithm is pretty complex, a big disadvantage for us

- A variation of the previous structures
- Can do everything the other structures can do, with a small overhead
- Can be constructed pretty quickly with relatively simple code

# Suffix arrays

- Take all the suffixes of  $S$

banani

anani

nani

ani

ni

i

- and sort them

anani

ani

banani

i

nani

ni

# Suffix arrays

- We can use this array to do everything that suffix tries can do
- Like string matching

# Suffix arrays

- Let's look for nan

anani  
ani  
banani  
i  
nani  
ni

# Suffix arrays

- Let's look for `nan`
- The first letter in the string has to be `n`, so we can binary search for the range of strings starting with `n`

```
anani  
ani  
banani  
i  
nani  
ni
```

# Suffix arrays

- Let's look for `nan`
- The first letter in the string has to be `n`, so we can binary search for the range of strings starting with `n`

```
nani  
ni
```



# Suffix arrays

- Let's look for `nan`
- The second letter in the string has to be `a`, so we can binary search for the range of strings that have `a` as the second letter

```
nani  
ni
```

- Let's look for `nan`
- The second letter in the string has to be `a`, so we can binary search for the range of strings that have `a` as the second letter

`nani`

- Let's look for `nan`
- The third letter in the string has to be `n`, so we can binary search for the range of strings that have `n` as the third letter

`nani`

- Let's look for `nan`
- The third letter in the string has to be `n`, so we can binary search for the range of strings that have `n` as the third letter

`nan`  
`i`

- Let's look for `nan`
- The third letter in the string has to be `n`, so we can binary search for the range of strings that have `n` as the third letter

`nan`  
`i`

- If there is at least one string left, we have a match

- Time complexity?
- For each letter in  $T$ , we do two binary searches on the  $n$  suffixes to find the new range
- Time complexity is  $O(m \times \log n)$
- A bit slower than doing it with a suffix trie, but still not bad

- But how do we construct a suffix array for a string?
- A simple `sort(suffixes)` is  $O(n^2 \log(n))$ , because comparing two suffixes is  $O(n)$
- And we still have the same problem as with suffix tries, there are almost  $n^2$  characters if we store all suffixes

# Suffix arrays

- The second problem is easy to fix
- Just store the indices of the suffixes

anani

ani

banani

i

nani

ni

- becomes

1: anani

3: ani

0: banani

5: i

2: nani

4: ni



- What about the construction?
- In short, we
  - ▶ sort all suffixes by only looking at the first letter
  - ▶ sort all suffixes by only looking at the first 2 letters
  - ▶ sort all suffixes by only looking at the first 4 letters
  - ▶ sort all suffixes by only looking at the first 8 letters
  - ▶ ...
  - ▶ sort all suffixes by only looking at the first  $2^i$  letters
  - ▶ ...
- If we use an  $O(n \log n)$  sorting algorithm, this is  $O(n \log^2 n)$
- We can also use an  $O(n)$  sorting algorithm, since all sorted values are between 0 and  $n$ , bringing it down to  $O(n \log n)$

# Suffix arrays

```
struct suffix_array {  
    struct entry {  
        pair<int, int> nr;  
        int p;  
  
        bool operator <(const entry &other) {  
            return nr < other.nr;  
        }  
    };  
  
    string s;  
    int n;  
    vector<vector<int> > P;  
    vector<entry> L;  
    vi idx;  
  
    // constructor  
};
```

# Suffix arrays

```
suffix_array(string _s) : s(_s), n(s.size()) {
    L = vector<entry>(n);
    P.push_back(vi(n));
    idx = vi(n);

    for (int i = 0; i < n; i++)
        P[0][i] = s[i];

    for (int stp = 1, cnt = 1; (cnt >> 1) < n; stp++, cnt <= 1) {
        P.push_back(vi(n));
        for (int i = 0; i < n; i++) {
            L[i].p = i;
            L[i].nr = make_pair(P[stp - 1][i],
                                i + cnt < n ? P[stp - 1][i + cnt] : -1);
        }
        sort(L.begin(), L.end());
        for (int i = 0; i < n; i++) {
            if (i > 0 && L[i].nr == L[i - 1].nr)
                P[stp][L[i].p] = P[stp][L[i - 1].p];
            else
                P[stp][L[i].p] = i;
        }
    }
    for (int i = 0; i < n; i++)
        idx[P[P.size() - 1][i]] = i;
}
```

- There is also one other useful operation on suffix arrays
- Finding the longest common prefix (lcp) of two suffixes of  $S$

```
1: anani
3: ani
0: banani
5: i
2: nani
4: ni
```

- $\text{lcp}(1,3) = 2$
- $\text{lcp}(2,1) = 0$
- This function can be implemented in  $O(\log n)$  by using intermediate results from the suffix array construction

# Suffix arrays

```
int lcp(int x, int y) {  
    int res = 0;  
    if (x == y) return n - x;  
    for (int k = P.size() - 1; k >= 0 && x < n && y < n; k--)  
        if (P[k][x] == P[k][y]) {  
            x += 1 << k;  
            y += 1 << k;  
            res += 1 << k;  
        }  
    }  
    return res;  
}
```

## Longest common substring

- Given two strings  $S$  and  $T$ , find their longest common substring
- $S = \text{banani}$
- $T = \text{kanina}$
- Their longest common substring is `ani`

GATTACA

UVa 11512