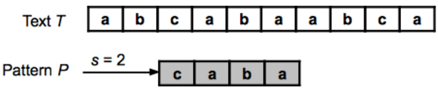
Thuật toán trên Chuỗi

Khoa Công Nghệ Thông Tin,
Trường Đại Học Thủy Lợi.

Ngày 9 tháng 3 năm 2017

Tìm kiếm chuỗi

- Bài toán trùng khớp chuỗi: tìm một hoặc tất cả sự xuất hiện của một mẫu trong một văn bản cho trước
- Ứng dụng
  - Thu hồi thông tin (information retrieval)
  - Trình soạn thảo văn bản
  - Sinh học tính toán (chuỗi DNA)
- Công thức chính thức
  - Một văn bản là một mảng $T[1..n]$ và một mẫu là một mảng $P[1..m]$ $(m \neq n)$
  - $T[i], P[j] \in$ một bảng chữ cái giới hạn $\sum$ (ví dụ, $\sum = \{0, 1\}$ hoặc $\sum = \{a, \ldots, z\}$)
  - Chúng ta nói rằng mẫu P **xuất hiện với độ dịch chuyển** s trong T nếu $0 \leq s \leq n - m$ và $T[s+1..s+m]{=}P[1..m]$



Thuật toán tìm kiếm chuỗi

- Giản đơn
- Boyer-Moore
- Rabin-Karp
- Knuth-Morris-Pratt (KMP)

Thuật toán giản đơn

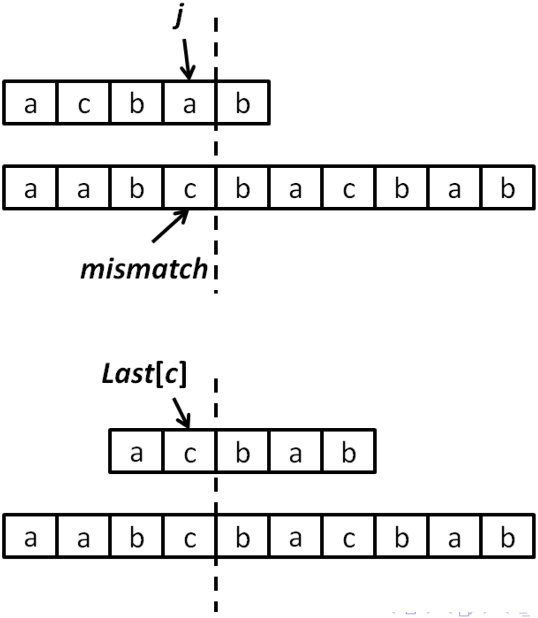
Algorithm 1: NaiveSM(P, T)

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);

Thuật toán Boyer-Moore

- Dịch chuyển từ trái sang phải
- Quyét từ phải qua trái
- Sử dụng thông tin nhận được bằng cách tiền xử lý P để bỏ qua nhiều sự sắp hàng (alignment) nhất có thể
- Quy tắc dịch chuyển ký tự tồi
  - $last[c]$: sự xuất hiện bên phía phải nhất của c trong P
  - Khi không trùng khớp: dịch P sang phải bởi $max\{j - last[c], 1\}$ với j là vị trí của kí tự không trùng khớp của P

Thuật toán Boyer-Moore



Thuật toán Boyer-Moore

```
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]];
      s = s + max(j-k,1);
    }
```

Thuật toán Rabin-Karp

- Chuyển mẫu $P[1..m]$ thành một số

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

với mỗi kí tự $P[i]$ được xem như một số nguyên không âm $< d$, và d là kích thước của bảng chữ cái

- Sử dụng luật Horner:

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

- Chuyển $T[s+1..s+m]$ thành số nguyên

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

- **Lưu ý**: t_{s+1} có thể được tính toán dễ dàng từ t_s như sau: $t_{s+1} = (t_s - T[s+1] * d^{m-1}) * d + T[s+m+1]$

Thuật toán Rabin-Karp

- Điểm yếu: khi m lớn, thì sự tính toán của p và t_s không theo thời gian liên tục
- Giải pháp: Tính p và t_s mô dun một số thích hợp q
  - Vấn đề còn lại: $p \equiv t_s (mod\ q)$ không có nghĩa rằng $p = t_s$, chúng ta phải kiểm tra $P[1..m]$ và $T[s+1..s+m]$ theo từng kí tự để xem liệu chúng có thực sự giống nhau
- Thời gian trong trường hợp xấu nhất là $\mathcal{O}(mn)$ với $P = a^m$ và $T = a^n$

Thuật toán Knuth-Morris-Pratt (KMP)

- So sánh: từ trái qua phải
- Dịch chuyển: hơn một vị trí
- Tiền xử lý mẫu
  - Mẫu $P[1..m]$
  - $\pi[q]$ là độ dài của tiền tố dài nhất của $P[1..q]$ mà cũng **chính** là hậu tố của $P[1..q]$

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

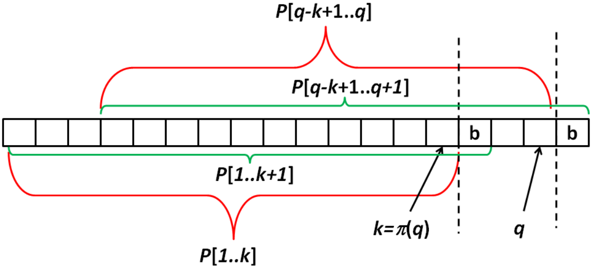
Thuật toán Knuth-Morris-Pratt (KMP) - Tiền xử lý

```
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;
  }
}
```

Thuật toán Knuth-Morris-Pratt (KMP) - Tiền xử lý

Chỉ rõ $k = \pi[q]$

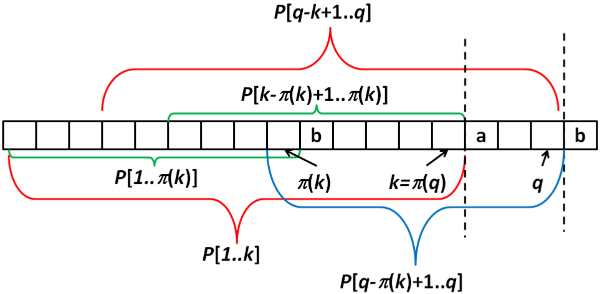
- Nếu $P[q+1] = P[k+1]$, thì $\pi[q+1] = \pi[q] + 1$



Thuật toán Knuth-Morris-Pratt (KMP) - Tiền xử lý

Chỉ rõ $k = \pi[q]$

- nếu $P[q+1] \neq P[k+1]$ và $P[q+1] = P[\pi[k]+1] = b$:
  - $P[1..k] = P[q-k+1..q] \Rightarrow P[k-\pi[k]+1..k] = P[q-\pi[k]+1..q]$
  - Hơn nữa, $P[k-\pi[k]+1] = P[1..\pi[k]$, vì vậy $P[1..\pi[k]] = P[q-\pi[k]+1..q]$,
  - Do đó $P[1..\pi[k]+1] = P[q-\pi[k]+1..q+1]$, điều này nghĩa là $\pi[q+1] = \pi[k]+1$

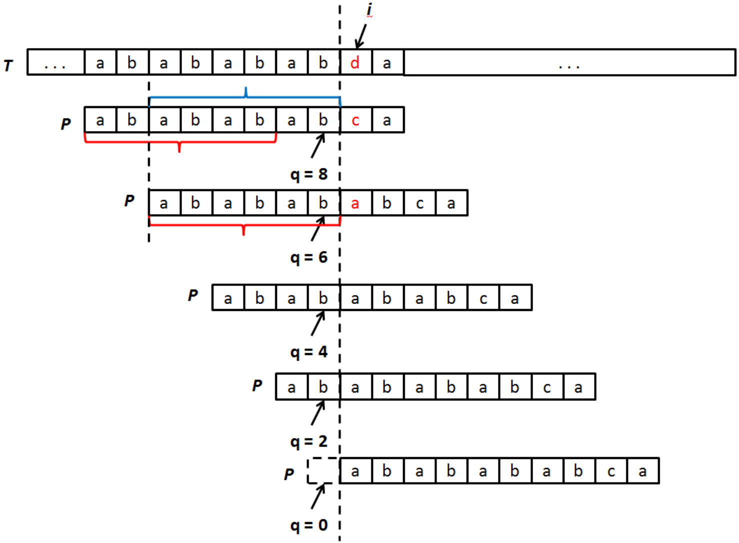


Thuật toán Knuth-Morris-Pratt (KMP)

SAMSUNG

```
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];
    }
  }
}
```

Thuật toán Knuth-Morris-Pratt (KMP)

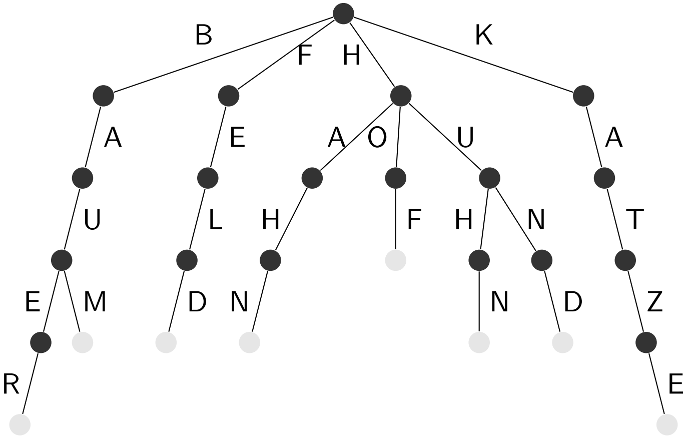


1. Tìm kiếm chuỗi
2. Tries
3. Tries hậu tố
4. Cây/Mảng Hậu tố

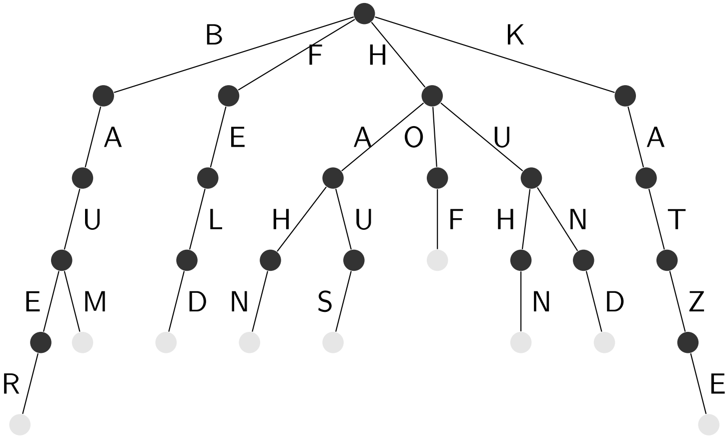
Tập các chuỗi

- Chúng ta thường có một tập (hoặc bản đồ) các chuỗi
- Phép chèn và tra cứu thường đảm bảo $O(\log n)$ phép so sánh
- Nhưng so sánh chuỗi thực sự khá đắt ...
- Có những cấu trúc dữ liệu khác, giống như **trie**, mà làm việc này theo một cách thông minh hơn

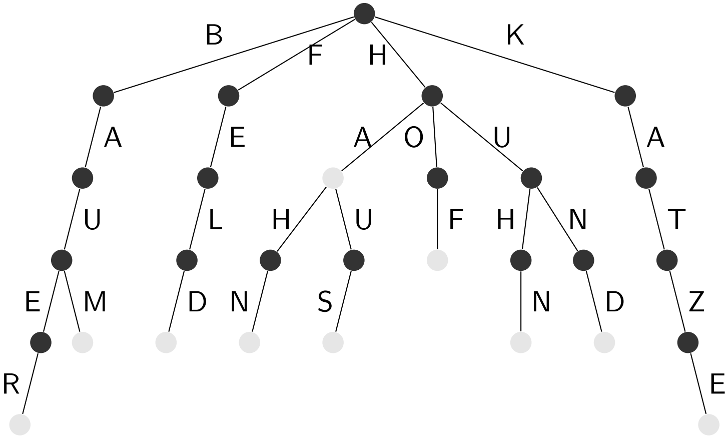
Tries



Tries



Tries



Tries

```
struct node {
    node* children[26];
    bool is_end;

    node() {
        memset(children, 0, sizeof(children));
        is_end = false;
    }
};
```

Tries

```
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;
    }
}
```

Tries

```
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")) {
    // ...
}
```

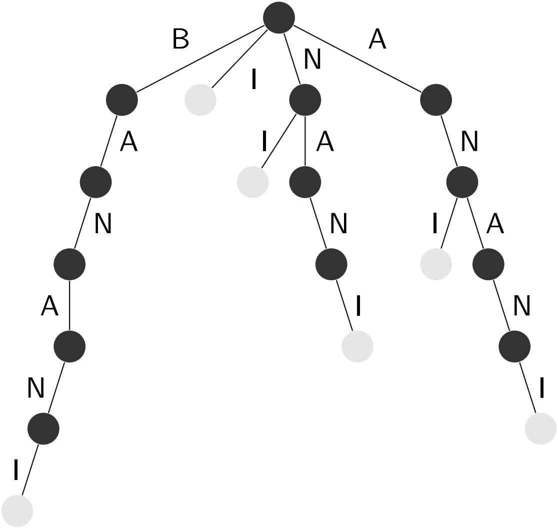
Tries

- Độ phức tạp thời gian?
- Đặt k là chiều dài chuỗi chúng ta đang thực hiện chèn hoặc tìm kiếm
- Phép tra cứu và chèn đều tốn $O(k)$
- Không gian cũng rất hiệu quả

Tries hậu tố

- Giả sử chúng ta đang làm việc với một vài chuỗi S có độ dài n
- Hãy chèn tất cả hậu tố của S vào trong một trie
- $S = \texttt{banani}$
  - `insert(trie, "banani");`
  - `insert(trie, "anani");`
  - `insert(trie, "nani");`
  - `insert(trie, "ani");`
  - `insert(trie, "ni");`
  - `insert(trie, "i");`

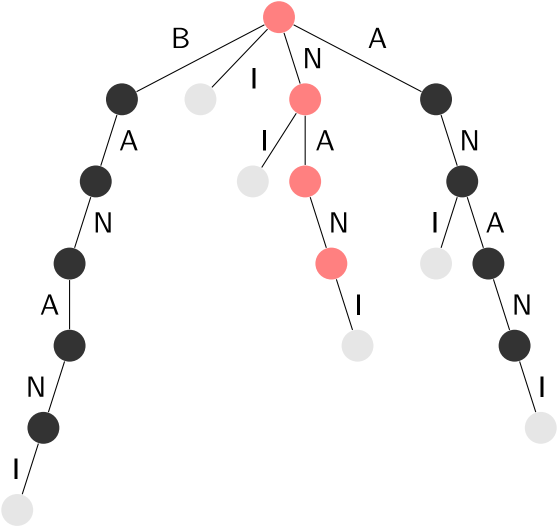
Tries hậu tố



Tries hậu tố

- Có rất nhiều những thứ thú vị mà chúng ta có thể làm với trie hậu tố
- Ví dụ: Trùng khớp chuỗi
- Nếu một chuỗi T là một chuỗi con trong S, thì (một cách hiển nhiên) nó phải bắt đầu tại một vài hậu tố của S
- Vì vậy chúng ta có thể đơn giản tìm kiếm T trong trie hậu tố của S, bỏ qua liệu nút cuối cùng có phải là nút kết thúc hay không
- Điều này chỉ tốn $O(m)$...

Tries hậu tố



Tries hậu tố

- Trùng khớp chuỗi nhanh nếu chúng ta có trie hậu tố cho S
- Nhưng độ phức tạp thời gian của việc xây dựng trie hậu tố là bao nhiêu?
- Có n hậu tố, và nó tốn $O(n)$ để chèn một trong số chúng
- Vì vậy $O(n^2)$, là khá là chậm
- Chúng ta có thể làm tốt hơn?
- Có thể lên đến n^2 nút trong đồ thị, vì vậy đây thực sự là tối ưu...

1. Tìm kiếm chuỗi
2. Tries
3. Tries hậu tố
4. Cây/Mảng Hậu tố

Cây hậu tố

- Tồn tại một phiên bản gọn/nén của một trie hậu tố, được gọi là cây hậu tố
- Nó có thể được xây dựng trong $O(n)$, và có tất cả đặc điểm mà trie hậu tố có
- Nhưng thuật toán xây dựng $O(n)$ là rất phức tạp, một bất lợi lớn cho chúng ta

Mảng hậu tố

- Một biến thể của những cấu trúc ở trước
- Nó có thể làm mọi thứ mà những cấu trúc khác có thể làm, với chi phí nhỏ
- Nó có thể được xây dựng rất nhanh với đoạn mã đơn giản tương đối

Mảng hậu tố

- Lấy tất cả hậu tố của S

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

- và sắp xếp chúng

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

Mảng hậu tố

- Chúng ta có thể dùng mảng này để làm mọi việc mà trie hậu tố có thể làm
- Giống như trùng khớp chuỗi

Mảng hậu tố

- Hãy tìm kiếm `nan`

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

Mảng hậu tố

- Hãy tìm kiếm `nan`
- Kí tự đầu tiên trong chuỗi phải là `n`, vì vậy chúng ta có thể tìm kiếm nhị phân cho khoảng của những chuỗi bắt đầu với `n`

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

Mảng hậu tố

- Hãy tìm kiếm `nan`
- Kí tự đầu tiên trong chuỗi phải là `n`, vì vậy chúng ta có thể tìm kiếm nhị phân cho khoảng của những chuỗi bắt đầu với `n`

```
nani
ni
```

Mảng hậu tố

- Hãy tìm kiếm `nan`
- Kí tự thứ hai trong chuỗi phải là `a`, vì vậy chúng ta có thể tìm kiếm nhị phân cho khoảng của những chuỗi có `a` là kí tự thứ hai

```
nani
ni
```

Mảng hậu tố

- Hãy tìm kiếm `nan`
- Kí tự thứ hai trong chuỗi phải là a, vì vậy chúng ta có thể tìm kiếm nhị phân cho khoảng của những chuỗi có a là kí tự thứ hai

```
nani
```

Mảng hậu tố

- Hãy tìm kiếm `nan`
- Kí tự thứ ba trong chuỗi phải là `n`, vì vậy chúng ta có thể tìm kiếm nhị phân cho khoảng của những chuỗi có `n` là kí tự thứ ba

```
nani
```

Mảng hậu tố

- Hãy tìm kiếm `nan`
- Kí tự thứ ba trong chuỗi phải là `n`, vì vậy chúng ta có thể tìm kiếm nhị phân cho khoảng của những chuỗi có `n` là kí tự thứ ba

```
nani
```

Mảng hậu tố

- Hãy tìm kiếm `nan`
- Kí tự thứ ba trong chuỗi phải là `n`, vì vậy chúng ta có thể tìm kiếm nhị phân cho khoảng của những chuỗi có `n` là kí tự thứ ba

```
nani
```

- Nếu chỉ còn ít nhất một chuỗi, chúng ta có một trùng khớp

Mảng hậu tố

- Độ phức tạp thời gian?
- Với mỗi ký tự trong T, chúng ta thực hiện hai tìm kiếm nhị phân trên n hậu tố để tìm khoảng mới
- Độ phức tạp thời gian là $O(m \times \log n)$
- Hơi chậm hơn khi làm việc này với trie hậu tố, nhưng vẫn không tồi

Mảng hậu tố

- Nhưng bằng cách nào chúng ta xây dựng một mảng hậu tố cho một chuỗi?

- Một `sort(suffixes)` đơn giản là $O(n^2 \log(n))$, bởi vì so sánh hai hậu tố là $O(n)$
- Và chúng ta vẫn có cùng một vấn đề như với trie hậu tố, có tối đa n^2 kí tự nếu chúng ta lưu trữ tất cả hậu tố

Mảng hậu tố

- Vấn đề thứ hai là dễ sửa lỗi
- Chỉ lưu trữ chỉ số của hậu tố

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

- trở thành

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

Mảng hậu tố

- Còn về việc xây dựng thì sao?
- Tóm lại, chúng ta
  - sắp xếp tất cả hậu tố bằng cách chỉ xét kí tự đầu tiên
  - sắp xếp tất cả hậu tố bằng cách chỉ xét 2 kí tự đầu tiên
  - sắp xếp tất cả hậu tố bằng cách chỉ xét 4 kí tự đầu tiên
  - sắp xếp tất cả hậu tố bằng cách chỉ xét 8 kí tự đầu tiên
  - ...
  - sắp xếp tất cả hậu tố bằng cách chỉ xét 2^i kí tự đầu tiên
  - ...
- Nếu chúng ta sử dụng một thuật toán sắp xếp $O(n \log n)$, đây là $O(n \log^2 n)$
- Chúng ta cũng sử dụng một thuật toán sắp xếp $O(n)$, do tất cả giá trị được sắp xếp ở giữa 0 và n, khiến nó giảm xuống $O(n \log n)$

Mảng hậu tố

```
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
};
```

Mảng hậu tố

```
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;
}
```

Mảng hậu tố

- Cũng có một thao tác hữu ích khác trên mảng hậu tố
- Tìm kiếm tiền tố chung lớn nhất (lcp) của hai hậu tố của S

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

- `lcp(1,3) = 2`
- `lcp(2,1) = 0`

- Hàm này được cài đặt trong $O(\log n)$ bằng cách sử dụng những kết quả trung gian từ việc xây dựng mảng hậu tố

Mảng hậu tố

```
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;
}
```

Thực hành

Chuỗi con chung lớn nhất

- Cho hai chuỗi S và T, tìm chuỗi con chung lớn nhất của chúng

- `S = banani`
- `T = kanina`

- Chuỗi con chung lớn nhất của chúng là `ani`

GATTACA

UVa 11512