Tên bài giảng
Xâu ký tự

Môn học: **Thuật toán và ứng dụng**

Chương: 5

Hệ: Đại học

Giảng viên: TS. Phạm Đình Phong

Email: phongpd@utc.edu.vn

Nội dung bài học

1. **Sắp xếp xâu**
2. **Cấu trúc dữ liệu mảng tiền tố và cây tiền tố**
3. **Xâu con**
4. **Biểu thức chính quy và otomat hữu hạn**

Sắp xếp xâu

- Khái niệm
  - **Thứ tự từ điển**: Các kí tự được sắp xếp theo một trật tự nhất định, được quy định rõ ràng trong bảng chữ cái a < b < c < d < e < f < ... < y < z. Kí hiệu a < b có nghĩa là a có thứ tự từ điển nhỏ hơn b.
  - Cho 2 xâu *A* = a[0]a[1]a[2]a[3]...a[n-1] và B = b[0]b[1]b[2]b[3]...b[m-1]. Ta nói *A* có thứ tự từ điển nhỏ hơn *B* khi và chỉ khi tồn tại 0 ≤ k ≤ n-1 và 0 ≤ k ≤ m-1 thỏa mãn: *A*[0...(k-1)] = *B*[0...(k-1)] và a[k] < b[k].
  - Ví dụ:

    A = “abcdef” và *B* = “abdefg”

    → ta thấy k = 2, *A*[0..1] = *B*[0..1] = “ab” và a[2] = “c” < b[2] = “d” suy ra A < B

Sắp xếp xâu

- Khái niệm
  - **Thứ tự từ điển**:
    - Cho xâu *X* có độ dài *N* và xâu *Y* có độ dài *M*, *x* có độ dài *n* là xâu con của *X* và y cũng có độ dài *n* là xâu con của *Y*. Giả sử *x* xuất hiện tại vị trí *k* của *X* và *y* cũng xuất hiện tại vị trí *k* của *Y* (*k* < *N* và *k* < *M*) và *X*[0..(k-1)] = Y[0..(k-1)], tức là *k* kí tự đầu tiên của *X* và *Y* trùng nhau tương ứng đôi một.

    Ta có tính chất sau:

    + nếu *x* < *y* thì *X* < *Y*

    + nếu *x* > *y* thì *X* > *Y*

    Ví dụ:

    X = "abcdefg", *x* = "def"

    Y = "abcdeeg", *y* = "dee"

    rõ ràng: k=3 vì,

    X[0..2] = *Y*[0..2] = "abc"

    x > y ("def" > "dee") → *X* > *Y*

Sắp xếp xâu

- Phương pháp LSD (least significant-digit first)
  - Đặc điểm:
    - Thuật ngữ *digit* được sử dụng thay cho *character* do xâu ký tự được mã hóa dựa trên bảng mã số (bảng mã ASCII có 256 số)
    - Xem xét các ký tự từ phải qua trái (xem xét các ký tự ít quan trọng trước)
    - Chỉ áp dụng để sắp xếp các xâu ký tự có độ dài bằng nhau

Sắp xếp xâu

- Phương pháp LSD (least significant-digit first)
  - Các bước: sắp xếp lần lượt từng ký tự từ phải qua trái sử dụng phương pháp sắp xếp đếm phân phối (counting sort):
    - Tính tần xuất của các ký tự cùng ở vị trí *d* trong các xâu và lưu vào mảng *count*[]
    - Chuyển các tần xuất sang chỉ số mảng bằng cách cộng dồn các tần xuất từ trái qua phải của mảng *count*[]
    - Phân phối lại các xâu ký tự ban đầu vào mảng trung gian theo chỉ số mảng *count*[] thu được ở bước trên
    - Sao chép lại mảng trung gian vào mảng ban đầu

Sắp xếp xâu

- Phương pháp LSD (least significant-digit first)

```
vector<string> sort(vector<string> a, int W) {
        // Sort a[] on leading W characters.
        int N = a.size();
        int R = 256;
        vector<string> aux(N);
        for (int d = W-1; d >= 0; d--) {   // Sắp xếp đếm ký tự thứ d
            vector<int> count(R+1);  // Mảng tần xuất
            for (int i = 0; i < N; i++)     // Tính tần xuất các ký tự thứ d
                count[a[i].at(d) + 1]++;
            for (int r = 0; r < R; r++)  // Chuyển sang chỉ số
                count[r+1] += count[r];
            for (int i = 0; i < N; i++)   // Phân phối lại
                aux[count[a[i].at(d)]++] = a[i];
            for (int i = 0; i < N; i++)   // Sao chép lại về mảng gốc
               a[i] = aux[i];
        }
        return a;
}
```

Sắp xếp xâu

- Phương pháp LSD (least significant-digit first)
  - Phân tích:

input ($W = 7$)	$d = 6$	$d = 5$	$d = 4$	$d = 3$	$d = 2$	$d = 1$	$d = 0$	output
4PGC938	2IYE230	3CIO720	2IYE230	2RLA629	1ICK750	3ATW723	1ICK750	1ICK750
2IYE230	3CIO720	3CIO720	4JZY524	2RLA629	1ICK750	3CIO720	1ICK750	1ICK750
3CIO720	1ICK750	3ATW723	2RLA629	4PGC938	4PGC938	3CIO720	1OHV845	1OHV845
1ICK750	1ICK750	4JZY524	2RLA629	2IYE230	1OHV845	1ICK750	1OHV845	1OHV845
1OHV845	3CIO720	2RLA629	3CIO720	1ICK750	1OHV845	1ICK750	1OHV845	1OHV845
4JZY524	3ATW723	2RLA629	3CIO720	1ICK750	1OHV845	2IYE230	2IYE230	2IYE230
1ICK750	4JZY524	2IYE230	3ATW723	3CIO720	3CIO720	4JZY524	2RLA629	2RLA629
3CIO720	1OHV845	4PGC938	1ICK750	3CIO720	3CIO720	1OHV845	2RLA629	2RLA629
1OHV845	1OHV845	1OHV845	1ICK750	1OHV845	2RLA629	1OHV845	3ATW723	3ATW723
1OHV845	1OHV845	1OHV845	1OHV845	1OHV845	2RLA629	1OHV845	3CIO720	3CIO720
2RLA629	4PGC938	1OHV845	1OHV845	1OHV845	3ATW723	4PGC938	3CIO720	3CIO720
2RLA629	2RLA629	1ICK750	1OHV845	3ATW723	2IYE230	2RLA629	4JZY524	4JZY524
3ATW723	2RLA629	1ICK750	4PGC938	4JZY524	4JZY524	2RLA629	4PGC938	4PGC938

Sắp xếp xâu

- Phương pháp LSD (least significant-digit first)
  - Phân tích:
    - Sử dụng ~7*WN* + 3*WR* lần truy cập mảng, trong đó *W* là độ dài của một xâu, *N* là số xâu, *R* là kích thước bảng mã ASCII
    - Không gian sử dụng thêm là *N* + *R*
    - Độ phức tạp về thời gian: *NW*

Sắp xếp xâu

- Phương pháp MSD (most significant-digit first)
  - Sắp xếp các xâu ký tự có độ dài khác nhau
  - Các ký tự được xem xét từ trái qua phải
  - Ý tưởng:
    - Xâu bắt đầu với ký tự a sẽ đứng trước xâu bắt bầu bằng ký tự b, …
      → Sử dụng phương pháp sắp xếp đếm phân phối (counting sort) để sắp xếp **ký tự đầu tiên** của các xâu, sau đó sắp xếp đệ quy trên các mảng con ứng với mỗi ký tự đầu tiên (các xâu của mảng con đã loạt bỏ ký tự đầu tiên)
    - MSD phân hoạch mảng thành các mảng con để có thể sắp xếp chúng độc lập

Sắp xếp xâu

- Phương pháp MSD (most significant-digit first)

input			d			
she	are		are	are	are	are
sells	by	lo	by	by	by	by
seashells	she		sells	seashells	sea	sea
by	sells		seashells	sea	seashells	seashells
the	seashells		sea	seashells	seashells	seashells
sea	sea		sells	sells	sells	sells
shore	shore		seashells	sells	sells	sells
the	shells		she	she	she	she
shells	she		shore	shore	shore	shore
she	sells		shells	shells	shells	shells
sells	surely		she	she	she	she
are	seashells		surely	surely	surely	surely
surely	the	hi	the	the	the	the
seashells	the		the	the	the	the

Sắp xếp xâu

- Phương pháp MSD (most significant-digit first)

```
void sort (vector<string> &a, int lo, int hi, int d)
{     // Sort from a[lo] to a[hi], starting at the dth character.
        /*if (hi <= lo + M)
        { insertionSort(a, lo, hi, d); return; }   */
        vector<int> count(R+2);      // Compute frequency counts.
        for (int i = lo; i <= hi; i++)
                count[charAt(a[i], d) + 2]++;
        for (int r = 0; r < R+1; r++) // Transform counts to indices.
                count[r+1] += count[r];
        for (int i = lo; i <= hi; i++) // Distribute.
                aux[count[charAt(a[i], d) + 1]++] = a[i];
        for (int i = lo; i <= hi; i++) // Copy back.
                a[i] = aux[i - lo];
        // Recursively sort for each character value.
        for (int r = 0; r < R; r++) {  //R = 256;
                if (lo + count[r+1] - 1 >= lo + count[r])
                sort (a, lo + count[r], lo + count[r+1] - 1, d+1);
        }
}
```

Sắp xếp xâu

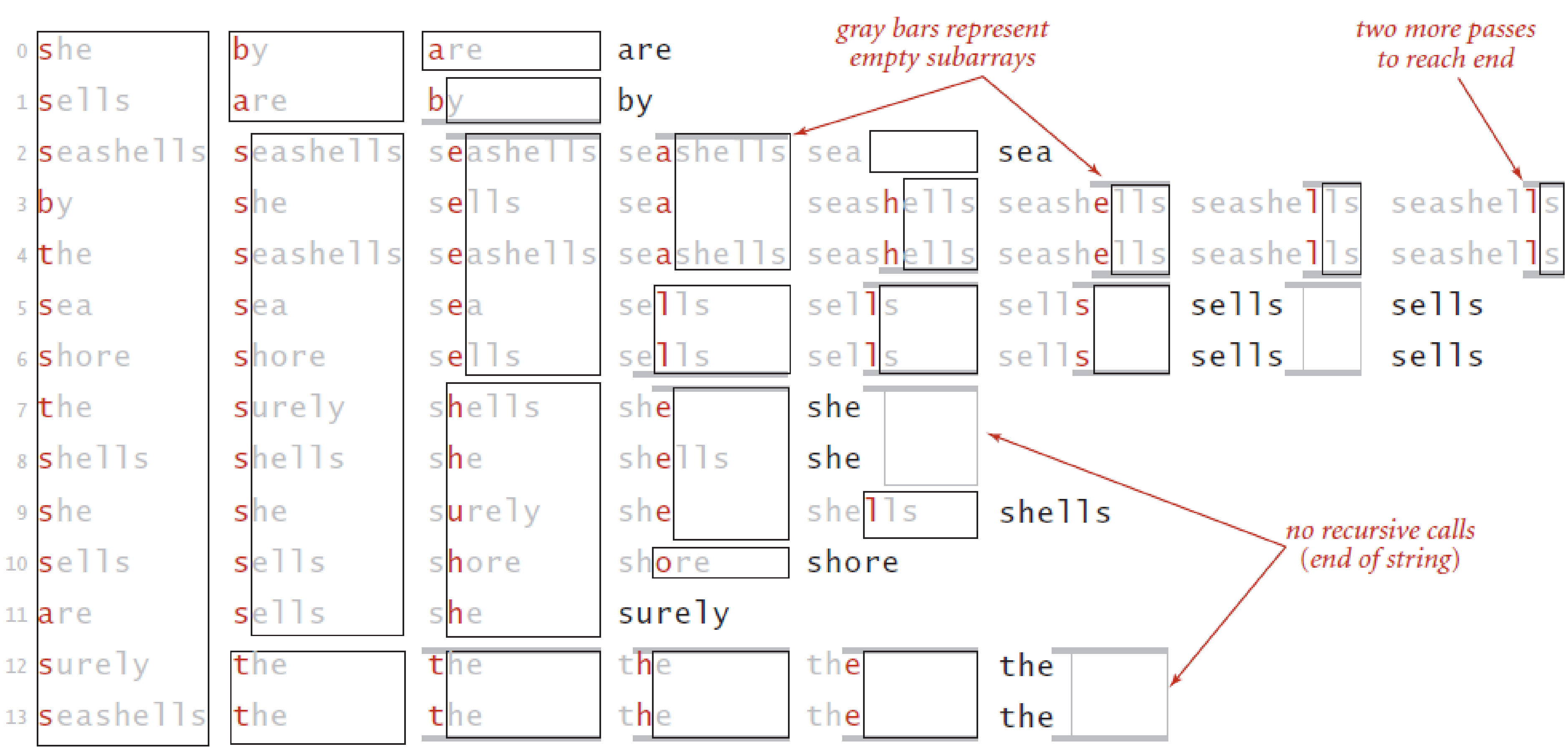
- Phương pháp MSD (most significant-digit first)
  - Phân tích:
    - Thuật toán MSD phụ thuộc vào số ký tự trong bảng ký tự, ta đang xem xét bảng ASCII với 256 ký tự. Nếu xem xét bảng ký tự Unicode thì có 65536 ký tự → MSD ngốn nhiều thời gian và không bộ nhớ.
    - Việc chia nhỏ mảng thành các mảng con đem lại hiệu quả vì chỉ thực hiện sắp xếp trên các mảng con đó. Tuy nhiên, khi các mảng con quá nhỏ sẽ ảnh hưởng đến hiệu suất của thuật toán MSD
    - Các xâu con giống nhau (equal keys): số lượng lớn sẽ ảnh hưởng đến hiệu suất của MSD do phải gọi đệ quy không cần thiết cho mọi ký tự của các xâu con giống nhau này. Trường hợp xấu nhất đối với MSD là tất các cả các xâu giống nhau

Sắp xếp xâu

- Sắp xếp nhanh (3-way string quick sort)
  - Phân hoạch ba đường dựa trên ký tự đầu tiên của các chuỗi trong mảng
  - Chỉ chuyển đến ký tự tiếp theo trên mảng con ở giữa (ký tự đầu tiên bằng với ký tự dùng để phân hoạch)

Sắp xếp xâu

- Sắp xếp nhanh (3-way string quick sort)



Sắp xếp xâu

- Sắp xếp nhanh (3-way string quick sort)

```
void sort(vector<string> &a, int lo, int hi, int d) {
        if (hi <= lo) return;
        int lt = lo, gt = hi;
        int v = charAt(a[lo], d);
        int i = lo + 1;
        while (i <= gt) {
                int t = charAt(a[i], d);
                if (t < v) exch(a, lt++, i++);     //Đổi chỗ hai xâu a[lt] và a[i]
                else if (t > v) exch(a, i, gt--);  //Đổi chỗ hai xâu a[i] và a[gt]
                else i++;
        }
        // a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi]
        sort(a, lo, lt-1, d);
        if (v >= 0) sort(a, lt, gt, d+1);
        sort(a, gt+1, hi, d);
}
```

Sắp xếp xâu

- Sắp xếp nhanh (3-way string quick sort)
  - Đặc điểm
    - Chia mảng thành ba phần nên hạn chế được số lượng phân hoạch rỗng
    - Xử lý tốt trường hợp có nhiều xâu bằng nhau hoặc có tiền tố chung dài hoặc trên vùng có nhiều mảng con
    - Không sử dụng không gian dư thừa cho bảng mã hóa ký tự

Sắp xếp xâu

- Sắp xếp nhanh (3-way string quick sort)

algorithm	stable?	inplace?	running time	extra space	
insertion sort for strings	yes	yes	between N and N^2	1	small arrays, arrays in order
quicksort	no	yes	$N \log^2 N$	$\log N$	general-purpose when space is tight
mergesort	yes	no	$N \log^2 N$	N	general-purpose stable sort
3-way quicksort	no	yes	between N and $N \log N$	$\log N$	large numbers of equal keys
LSD string sort	yes	no	NW	N	short fixed-length strings
MSD string sort	yes	no	between N and Nw	$N + WR$	random strings
3-way string quicksort	no	yes	between N and Nw	$W + \log N$	general-purpose, strings with long prefix matches

Sắp xếp xâu

- Sắp xếp xâu tiếng Việt
  - Đặc điểm:
    - Quy tắc sắp xếp các dấu theo thứ tự:

      không dấu < huyền < hỏi < sắc < ngã < nặng

      Ví dụ 1: Hoa < Hòa < Hỏa < Hóa < Hõa < Họa
    - Các âm: a < ă < â, e < ê, ô < ơ, u < ư, d < đ
    - **Lưu ý**: quy tắc bỏ dấu sẽ ảnh hưởng đến thứ tự sắp xếp các từ tiếng Việt

      Ví dụ 2:

      - Theo quy tắc bỏ dấu truyền thống thì dấu của các từ ở ví dụ 1 sẽ được bỏ vào nguyên âm giữa. Như vậy, ta sẽ thấy Hoàn sẽ đứng trước Hòa.
      - Nếu bỏ dấu vào nguyên âm thứ hai thì ta sẽ có Hoà sẽ đứng trước Hoàn

Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Tiền tố, hậu tố
  - Cho xâu *X* có độ dài *N* kí tự, đánh số từ 0 đến *N*-1 từ trái sang phải. Xâu *X* chỉ gồm các kí tự in thường trong bảng chữ cái latin (*a, b, c, d, ..., z*)
  - Xâu *P* (độ dài của *P* là *M* <= *N*) được gọi là tiền tố (prefix) của xâu *X* khi và chỉ khi *P* trùng với *M* kí tự đầu tiên của *X*, tức là *P* = *X*[0...(*M*-1)]. Hoặc, tồn tại một xâu *Y* thỏa mãn *X* = *PY* thì *P* là tiền tố của *X*
  - Ví dụ:

X = **abc**def; *P* = **abc** => *Y* = def và P là tiền tố của X

X = ab**c**def; *P* = ab**b** => *P* **không** là tiền tố của X

Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Tiền tố, hậu tố
  - Xâu *S* (độ dài là *m* <= *N*) được gọi là hậu tố (suffix) của xâu *X* khi và chỉ khi *S* trùng với *m* kí tự cuối cùng của *X*, tức là *S* = *X*[(*N*-*m*)...(*N*-1)]. Hoặc, tồn tại một xâu *Z* thỏa mãn *X* = *ZS* thì *S* là hậu tố của *X*
  - Ví dụ:

X = abc<u>**def**</u>; *S* = <u>**def**</u> => *Z* = abc và S là hậu tố của *X*

X = abcde<u>**f**</u>; *S* = de => *S* <u>**không**</u> là hậu tố của *X*

Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Tiền tố, hậu tố
  - Tính chất 1 (bắc cầu):

    1) *a* là tiền tố của *b* và *b* là tiền tố của *c* thì *a* cũng là tiền tố của *c*

    2) *a* là hậu tố của *b* và *b* là hậu tố của *c* thì *a* cũng là hậu tố của *c*

  - Tính chất 2: (ký hiệu |*a*| là độ dài của xâu *a*)

    1) *a* là tiền tố của *c* và *b* là tiền tố của *c* và $|a| \le |b|$ thì *a* là tiền tố của *b*

    2) *a* là hậu tố của *c* và *b* là hậu tố của *c* và $|a| \le |b|$ thì *a* là hậu tố của *b*

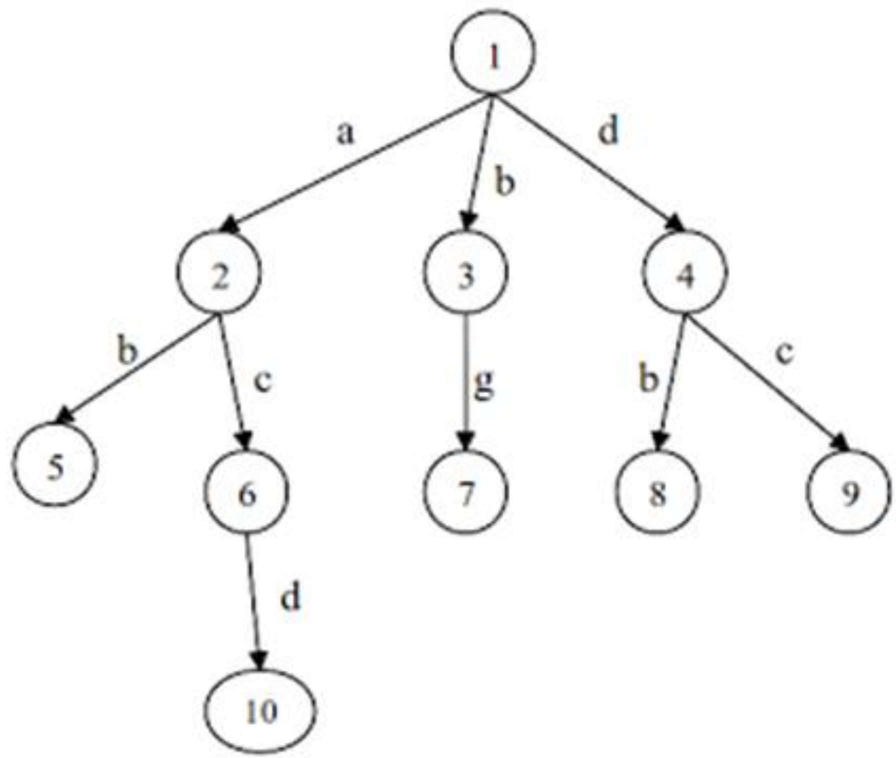
Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Cấu trúc cây tiền tố (Prefix Tree/Trie)
  - Xử lý tiền tố trong một tập các xâu là một trong những vấn đề thường được đề cập trong các bài toán xử lý xâu
  - Trong nhiều trường hợp, cấu trúc cây tiền tố (Trie) được coi là một giải pháp hiệu quả
  - Bản chất của cấu trúc Trie là một cây có gốc
  - Nút gốc không chứa thông tin, nhưng mỗi cạnh nối hai nút trên cây tương ứng với một ký tự

Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Cấu trúc cây tiền tố (Prefix Tree/Trie)
  - Đường đi từ nút gốc tới một nút bất kỳ trên cây này cho biết tập đang xét có chứa xâu tiền tố biểu diễn bởi tập các cạnh thể hiện đường đi từ nút gốc tới nút đang đề cập đến

Đường đi từ nút 1 tới nút 7 đại diện cho xâu tiền tố “bg”, đường đi từ nút 1 đến nút 8 đại diện cho xâu tiền tố là “db”, đường đi từ nút 1 đến nút 10 đại diện cho xâu tiền tố “acd”



Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

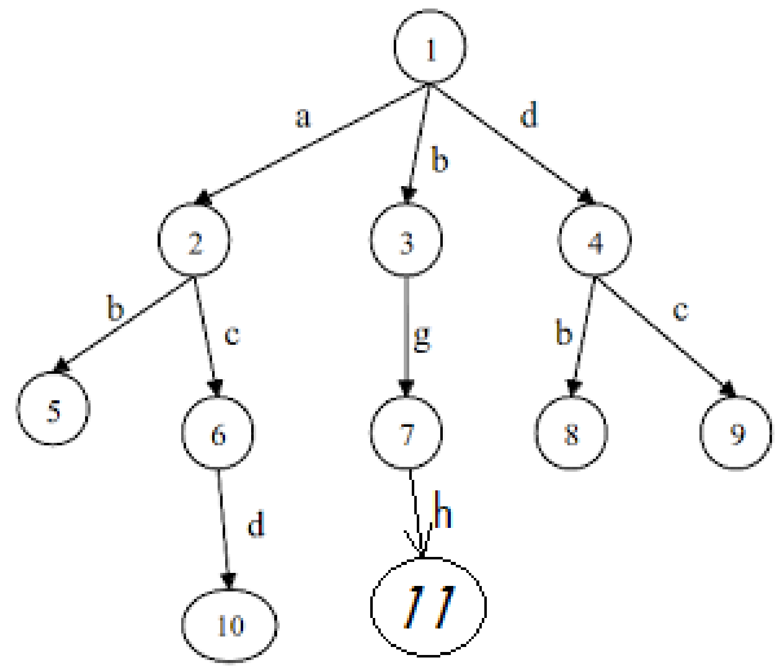
- Cấu trúc cây tiền tố (Prefix Tree/Trie)

Từ mỗi nút chỉ có tối đa 1 cạnh tỏa tới một nút khác gắn ký tự *c* bất kỳ

Muốn lưu thêm 1 xâu "bgh" thì khi xuất phát từ nút 1 ta sẽ sử dụng cạnh chứa ký tự 'b' đã có tỏa ra từ nút 1 và đi tới nút 3 (không vẽ thêm cạnh mới)

Tương tự, sử dụng tiếp cạnh 3-7 để tới nút 7, chứa ký tự 'g'

Từ nút 7 không có cạnh nào tỏa ra mà mang theo ký tự 'h', sẽ vẽ thêm cạnh này và đích đến của nó sẽ là 1 nút mới (đánh số thứ tự nút mới là 11 cho liền mạch)



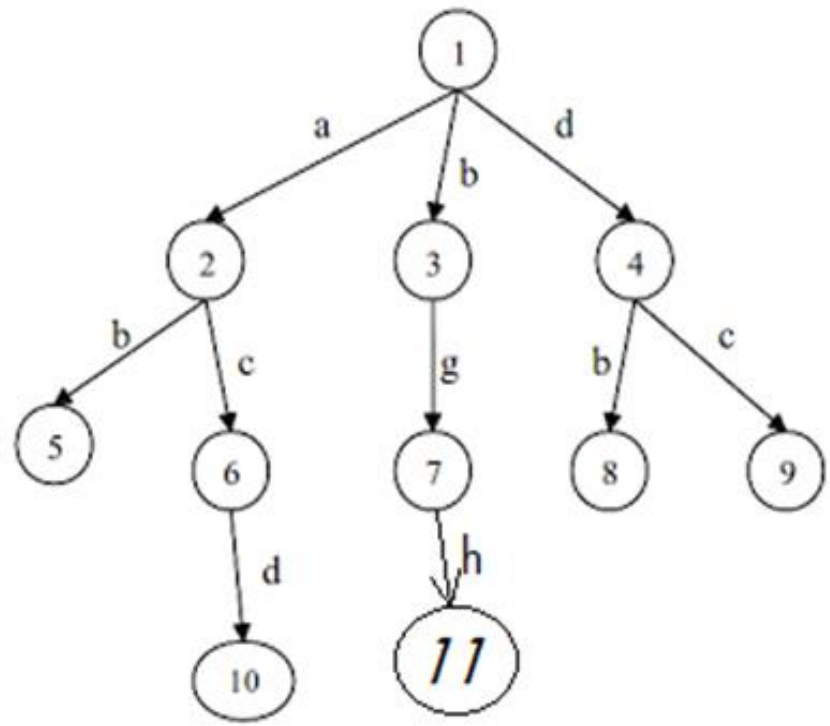
Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Cấu trúc cây tiền tố (Prefix Tree/Trie)

Trong một số trường hợp không thể biết được đâu là điểm kết thúc xâu.

Ví dụ như với cây bên phải, ta không thể biết được liệu có xâu "bg" trong tập hợp không, hay "bg" chỉ đơn giản là tiền tố của xâu "bgh" vừa thêm vào

→ mỗi nút có thêm 1 thuộc tính boolean nữa, cho biết nút tương ứng có phải điểm kết thúc của một xâu hay không.



Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Cấu trúc cây tiền tố (Prefix Tree/Trie)
  - Có ba thao tác chính
    - Thêm một xâu S vào cây. Độ phức tạp là O(|S|)
    - Xóa một xâu S khỏi cây. Độ phức tạp là O(|S|)
    - Kiểm tra xem một xâu S có tồn tại trong tập hợp dưới dạng một xâu hoàn chỉnh hoặc một tiền tố hay không. Độ phức tạp O(|S|)

Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Cấu trúc cây tiền tố (Prefix Tree/Trie)
  - Ưu điểm
    - Cài đặt đơn giản, dễ nhớ
    - Tiết kiệm bộ nhớ: Khi số lượng khóa lớn và các khóa có độ dài nhỏ thì trie tiết kiệm bộ nhớ hơn do các phần đầu giống nhau của các khoá chỉ được lưu một lần. Ưu điểm này có ứng dụng rất lớn, chẳng hạn trong từ điển
    - Dựa vào tính chất của cây trie, có thể thực hiện một số thao tác liên quan đến thứ tự từ điển như sắp xếp, tìm một khóa có thứ tự từ điển nhỏ nhất và lớn hơn một khóa cho trước, …

Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Tiền tố chung lớn nhất (longest common prefix – LCP)
  - Ví dụ: “apple”, “ape”, “april” → tiền tố chung lớn nhất là “ap”
  - **Sử dụng cấu trúc cây tiền tố**

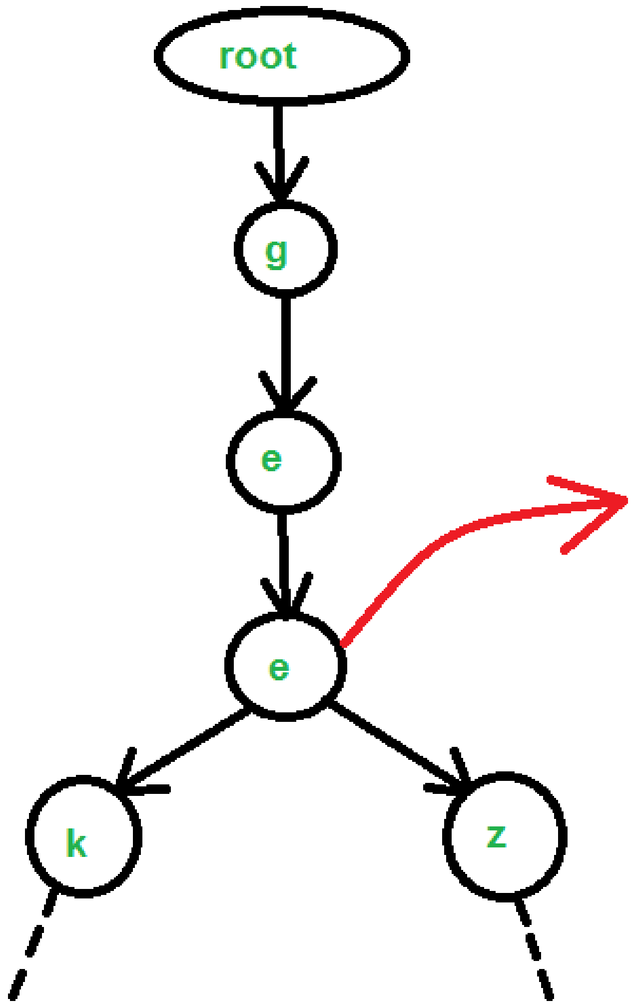
Bước 1. Chèn lần lượt các ký tự vào cây tiền tố.

Bước 2. Tiến hành duyệt cây. Duyệt cho tới khi tìm thấy một nút có nhiều hơn một con (xuất hiện nhánh) hoặc không có con nào (kết thúc xâu)

(*các ký tự xuất hiện trong tiền tố dài nhất phải là con đơn của bố mẹ, tức là bố mẹ phải không có nhánh tại bất kỳ nút nào*)

Cấu trúc dữ liệu mảng tiền tố và cây tiền tố

- Tiền tố chung lớn nhất (LCP)



Một cây tiền tố cho các

geeksforgeeks, geeks, geek, geezer

Tìm nút xuất hiện nhánh

Tất cả các ký tự phía trên nút này đều nằm trong tiền tố dài nhất

Do đó, "gee" là xâu con chung dài nhất

Xâu con

- Mảng hậu tố
  - Cho xâu T = t[0]t[1]t[2]t[3]t[4]...t[n-1] có độ dài là *n* kí tự chỉ gồm các kí tự trong bảng chữ cái latin
  - Ta thấy T có *n* hậu tố, các hậu tố này có dạng:

    t[k]t[k+1]...t[n-1] với k=0...n-1, các hậu tố có đặc điểm chung là các xâu con của xâu T mà kí tự cuối luôn là t[n-1]

    Ví dụ: xâu banana có các hậu tố:

    k = 0 → banana
    k = 1 → anana
    k = 2 → nana
    k = 3 → ana
    k = 4 → na
    k = 5 → a

Xâu con

- Mảng hậu tố
  - Mỗi hậu tố được đồng nhất với *k* là vị trí bắt đầu của nó, xâu T có n hậu tố, như vậy ta sẽ sắp xếp các hậu tố này theo thứ tự từ điển tăng dần và lưu vào một mảng, mảng này chính là mảng hậu tố (suffix array)
  - Ví dụ: T = “banana”

Các hậu tố:	Sắp xếp các hậu tố theo thứ tự từ điển:
0 - banana	5 - a
1 - anana	3 - ana
2 - nana	1 - anana
3 - ana	0 - banana
4 - na	4 - na
5 - a	2 - nana

Mảng hậu tố là Suffix array = {5, 3, 1, 0, 4, 2}

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp $O(n^2 \mathrm{Log} n)$, n là độ dài của xâu
    - Sinh tất cả các hậu tố và sắp xếp chúng bằng thuật toán Quick Sort hoặc Merge Sort, trong khi sắp xếp thì duy trì chỉ số gốc của chúng
      - ✓ Sắp xếp: $O(n \mathrm{Log} n)$
      - ✓ So sánh chuỗi: $O(n)$

      → $O(n^2 \mathrm{Log} n)$

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp $O(n \times \text{Log}n \times \text{Log}n)$
    - Ý tưởng xuất phát từ việc các xâu được sắp xếp là các hậu tố của một xâu đơn
    - Xét bài toán so sánh hai xâu *X* và *Y* có cùng độ dài là 4. Giả sử, ta có *px* = *X*[0…1], *py* = *Y*[0…1], *sx* = *X* [2…3], *sy* = *Y*[2...3], ta suy ra các trường hợp:
      - *px* < *py* => *X* < *Y*
      - *px* > *py* => *X* > *Y*
      - *px* = *py* và *sx* < *sy* => *X* < *Y*
      - *px* = *py* và *sx* > *sy* => *X* > *Y*

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp $O(n \times \mathrm{Log}n \times \mathrm{Log}n)$
    - Đầu tiên sắp xếp các hậu tố theo thứ tự tăng dần của **kí tự đầu tiên** của mỗi hậu tố
    - Sau đó sắp xếp các hậu tố theo thứ tự tăng dần của **hai** kí tự đầu tiên
    - Tiếp theo sắp xếp các hậu tố theo thứ tự tăng dần của **bốn** kí tự đầu tiên bằng cách so sánh thứ tự tiền tố độ dài 2 và so sánh thứ tự xâu con độ dài 2, xuất hiện ở vị trí 2 của 2 hậu tố cần so sánh
    - Cứ như vậy, tăng số ký tự đầu tiên được sắp xếp lên gấp đôi trong khi số ký tự được xem xét nhỏ hơn $2n$

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp $O(n \times \text{Log}n \times \text{Log}n)$
    - Điểm quan trọng ở đây là nếu chúng ta đã sắp xếp 2^i ký tự đầu tiên của các hậu tố thì chúng ta có thể sắp xếp 2^{i+1} ký tự đầu tiên của các hậu tố trong thời gian $O(n\text{Log}n)$ sử dụng thuật toán có độ phức tạp thời gian là $O(n\text{Log}n)$ như Quick Sort hoặc Merge Sort vì hai hậu tố có thể được so sánh với thời gian O(1)
    - Hàm sắp xếp được gọi $O(\text{Log}n)$ lần, do đó độ phức tạp thời gian tổng thể là $O(n \times \text{Log}n \times \text{Log}n)$

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp $O(n \times \mathrm{Log}n \times \mathrm{Log}n)$

Bước 1. Sắp xếp các hậu tố theo thứ tự tăng dần của hai ký tự đầu tiên của mỗi hậu tố
Gán hạng (rank) cho tất cả các hậu tố dựa trên mã ASCII của ký tự đầu tiên (str[i] – ‘a' đối với hậu tố thứ i)

Index	Suffix	Rank
0	banana	1
1	anana	0
2	nana	13
3	ana	0
4	na	13
5	a	0

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp $O(n \times \mathrm{Log}n \times \mathrm{Log}n)$

Bước 1 (tiếp). Với mỗi ký tự, ta cũng lưu hạng của ký tự kế tiếp (là hạng của ký tự tại str[i + 1]) dùng để sắp xếp hai ký tự đầu tiên. Nếu str[i] là ký tự cuối xâu thì hạng của ký tự kế tiếp nó (next rank) là -1

Index	**Suffix**	**Rank**	**Next Rank**
0	banana	1	0
1	anana	0	13
2	nana	13	0
3	ana	0	13
4	na	13	0
5	a	0	-1

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp $O(n \times \mathrm{Log}n \times \mathrm{Log}n)$

Bước 1 (tiếp). Sắp xếp tất cả các hậu tố theo rank và next rank. Rank được xem là số đầu tiên và next rank được xem là số thứ hai.

Index	Suffix	Rank	Next Rank
5	a	0	-1
1	anana	0	13
3	ana	0	13
0	banana	1	0
2	nana	13	0
4	na	13	0

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp $O(n \times \text{Log}n \times \text{Log}n)$

Bước 2. Gán hạng mới cho các hậu tố → xem xét từng hậu tố một. Gán 0 cho hậu tố đầu tiên. Tiếp theo, xem xét cặp hạng (rank và next rank) của hậu tố liền trước hậu tố hiện tại. Nếu hai cặp hạng giống nhau thì gán cùng một hạng, ngược lại thì tăng hạng lên 1.

Index	**Suffix**	**Rank**	
5	a	0	[Gán 0 cho hậu tố đầu tiên]
1	anana	1	(0, 13) khác trước đó (0, -1)
3	ana	1	(0, 13) giống trước đó (0, 13)
0	banana	2	(1, 0) khác trước đó (0, 13)
2	nana	3	(13, 0) khác trước đó (1, 0)
4	na	3	(13, 0) giống trước đó (0, 13)

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp $O(n \times \mathrm{Log}n \times \mathrm{Log}n)$

Bước 2 (tiếp). Tính lại next rank.

Với mọi hậu tố str[i], lưu hạng của hậu tố tiếp sau (next rank) tại str[i + 2]. Nếu str[i] không có hậu tố tiếp sau tại vị trí i + 2 thì gán next rank bằng -1.

Index	Suffix	Rank	Next Rank
5	a	0	-1
1	anana	1	1
3	ana	1	0
0	banana	2	3
2	nana	3	3
4	na	3	-1

Xâu con

- Mảng hậu tố
  - **Xây dựng mảng hậu tố**
  - Thuật toán với độ phức tạp O($n \times$ Log$n \times$ Log$n$)

Bước 2 (tiếp). Sắp xếp tất cả các hậu tố theo rank và next rank.

Index	Suffix	Rank	Next Rank
5	a	0	-1
3	ana	1	0
1	anana	1	1
0	banana	2	3
4	na	3	-1
2	nana	3	3

Xâu con

- Khái niệm
  - Cho hai xâu *X* (độ dài |*X*|) và *Y* (độ dài |*Y*|). Nếu xâu *X* xuất hiện tại một vị trí nào đó trong xâu *Y* thì *X* là xâu con (substring) của xâu *Y* (hay *X* là một dãy các kí tự liên tiếp bất kì trong *Y*). Tức là *X*=*Y*[*k*...(*k*+|*X*|)], *k* là vị trí mà *X* xuất hiện trong *Y*
  - *X* là xâu con của *Y* khi và chỉ khi *X* là tiền tố của một hậu tố của *Y* (*X* là hậu tố của một tiền tố của Y)
  - Ví dụ:

    Y = "BANANA", X = "ANA" → *X* xuất hiện tại vị trí *k* = 1, đồng thời *X* là tiền tố của hậu tố "ANANA" và *X* cũng là hậu tố của tiền tố "BANA"

Xâu con

- Giải bài toán so khớp chuỗi
  - Bài toán: Cho 2 xâu *A* và *B*. Hãy tìm các vị trí trên xâu *A* mà nó khớp với xâu *B*.
  - Ví dụ: với *A* = “aaaaa”, *B* = “aa” thì các vị trí tìm được là 0, 1, 2, 3 (với hệ đếm từ 0)

Xâu con

- Giải bài toán so khớp chuỗi bằng phương pháp khờ khạo (ngây ngô)
  - Với từng vị trí của xâu *A*, ta xét xem xâu con liên tục của *A* có độ dài |*B*|, có trùng khớp với xâu *B* hay không
  - Phương pháp này có độ phức tạp $O(|A|*|B|)$
  - Khi *A* và *B* trở nên rất lớn, thuật toán này trở nên rất chậm chạp

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Cách tiếp cận của thuật toán KMP được coi là cải tiến từ phương pháp khờ khạo có độ phức tạp $O(|A|*|B|)$
  - Phương pháp khờ khạo chậm chạp là do phải xét lại các vị trí đã được duyệt qua rồi
  - KMP khắc phục nhược điểm này bằng cách khởi tạo sẵn thông các thông tin về xâu *B* và sử dụng kỹ thuật 2-con-trỏ (two pointers)

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Nhờ bảng thông tin khởi tạo sẵn, ta biết được những vị trí nào, hoặc là không thể khớp với xâu *B*, hoặc là chắc chắn khớp với xâu *B* tới một mức độ nào đó → đủ để việc duyệt xâu *A* là một quá trình tuyến tính (đi và không quay đầu nhìn lại)

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Ví dụ: xem xét hai xâu

    A = “I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN”

    B = “SEVENTY SEVEN”

    → cần thông tin khởi tạo của xâu *B*. Đó là thông tin gì?
  - Với phương pháp khờ khạo, khi bắt đầu có một vị trí trên xâu *A* mà khớp với *B*, ta phải duyệt lại từng vị trí phía sau vị trí mới tìm được lúc đầu, trong khi có nhiều trường hợp ta có thể bỏ qua được

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Ví dụ:

    A = "**AAAA**ABAAABA"

    B = "**AAAA**“

    Ở trên, xâu B khớp với 4 ký tự đầu của xâu A

    A = "**AAAAA**BAAABA"

    B = "**AAAA**“

    Khi dịch chuyển mẫu sang phải một ký tự, ta có thể chỉ cần so sánh ký tự thứ 4 của xâu B với ký tự thứ 5 của xâu B vì ta biết rằng ba ký tự đầu sẽ khớp → bỏ qua việc so khớp 3 ký tự đầu

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - **Công việc cần làm là:**
  - Với mỗi vị trí thứ i trên xâu *B*, ta cần phải biết được giá trị *k* (0 ≤ k ≤ i) thỏa mãn:
    - Tiền tố độ dài *k* của *B* trùng với hậu tố độ dài *k* của *B*
    - Giá trị của *k* phải lớn nhất có thể
  - Mảng lưu các giá trị này là LPS[] (Longest Proper Suffix – hậu tố đúng dài nhất trùng với tiền tố)

    → LPS[i] = hậu tố đúng dài nhất của *B*[0..i] cũng là tiền tố của *B*[0..i]
  - Các tiền tố đúng là các tiền tố không bao gồm toàn bộ xâu. Ví dụ: tiền tố đúng của “ABC” là “A”, “AB” và không phải là “ABC”. Tương tự với các hậu tố đúng.

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Ví dụ với các xâu *B*:

    B = "AAAA" => LPS[] = {0, 1, 2, 3}

    B = "ABCDE" => LPS[] = {0, 0, 0, 0, 0}

    B = "AABAACAABAA" => LPS[] = {0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5}

  - Vậy việc xây dựng mảng LPS[] như thế nào?

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Trước hết, ta thấy rằng LPS[0] = 0 vì không có hậu tố đúng nào có độ dài nhỏ hơn 1 cả
  - Với các phần tử LPS[i] (i > 0), ta xử lý như sau:

    1) Đặt một biến tạm *tmp* = LPS[i-1]. Đó là độ dài của hậu tố đúng lớn nhất đang khớp ngay trước đó.

    2) Chừng nào *B*[*tmp*] và *B*[i] chưa khớp nhau và *tmp* còn mang giá trị dương thì rút ngắn hậu tố đúng cần đối chứng lại. Công thức rút ngắn sẽ là: *tmp* = LPS[*tmp*-1].

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Do lệnh lặp nên sẽ xảy ra 2 khả năng:

    1) Nếu *B*[*tmp*] đã khớp với *B*[i], tức là ta đã tìm được hậu tố đúng thỏa mãn ở vị trí i → tăng giá trị *tmp* thêm 1 đơn vị và gán giá trị mới này vào LPS[i].

    2) Nếu không, tức là *tmp* = 0 và không có hậu tố đúng thỏa mãn cho vị trí i → gán LPS[i] = 0.

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Xét xâu *B* = “SEVENTY SEVEN”, ta thấy:

Khởi tạo: LPS[0] = 0, *tmp* = 0.

i = 1, *tmp* = 0 → B[0] = ‘S’ ≠ ‘E’ = B[1], nên LPS[1] = 0.

Tương tự, ta có LPS[2] = LPS[3] = … = LPS[7] = 0.

i = 8, *tmp* = 0. Do B[0] = B[8] nên LPS[8] = ++*tmp* = 1 (ta có hậu tố đúng độ dài 1 thỏa mãn cho xâu B{8} vì “S” = “S”).

i = 9, *tmp* = 1. Ta lại có B[1] = B[9] nên LPS[9] = ++*tmp* = 2 (“SE” = “SE”).

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Xét xâu *B* = “SEVENTY SEVEN”, ta thấy:

    Tiếp tục như vậy:

    i = 10, *tmp* = 2 → LPS[10] = ++*tmp* = 3 (“SEV” = “SEV”).

    i = 11, *tmp* = 3 → LPS[11] = ++*tmp* = 4 (“SEVE” = “SEVE”).

    i = 12, *tmp* = 4 → LPS[12] = ++*tmp* = 5 (“SEVEN” = “SEVEN”).

  - Phương pháp duyệt này sử dụng kỹ thuật hai con trỏ, con trỏ trước (dùng để đánh dấu độ dài của hậu tố đúng còn đang khớp) không bao giờ chèn qua con trỏ sau, con trỏ sau thì đi tuần tự từ đầu tới cuối xâu *B*. Độ phức tạp của quá trình này là $O(|B|)$.

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Duyệt xâu *A* → áp dụng kỹ thuật hai con trỏ tương tự như duyệt xâu *B*
    - Con trỏ trước dùng để đánh dấu vị trí ướm xâu *B* vào xâu *A* và không bao giờ chèn qua con trỏ sau
    - Con trỏ sau thì đi tuần tự từ đầu tới cuối xâu *A* → Độ phức tạp của quá trình này là $O(|A|)$.

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Quá trình duyệt xâu *A* để so sánh như sau:
    - Ta đặt lại biến tạm tmp = 0 rồi bắt đầu duyệt từ đầu xâu
    - Chừng nào *B*[*tmp*] và *A*[i] chưa khớp nhau và *tmp* còn mang giá trị dương, ta giảm giá trị *tmp* xuống. Công thức rút ngắn sẽ là: *tmp* = LPS[*tmp*-1]

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - **Ý nghĩa của việc giảm giá trị *tmp*:**
    - Giả định là ta đang đặt song song 2 xâu *A* và *B*, và phía trên hai xâu là một cây kim dùng để đánh dấu vị trí đang xem xét
    - Nếu ở vị trí cây kim, hai ký tự tương ứng của *A* và *B* trùng nhau, ta dịch cây kim sang phải và tiếp tục xét tiếp. Nếu không, ta đẩy xâu *B* sang phải cho tới khi nào tất cả các ký tự trước cây kim của hai xâu *A* và *B* là trùng nhau hoặc cây kim đặt ở ký tự đầu của xâu *B*, tức là không có vị trí nào trùng ở trước

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - **Do lệnh lặp nên sẽ xảy ra hai khả năng:**
    - Nếu *B*[*tmp*] đã khớp với *A*[i], tức là đã tìm được hậu tố đúng thỏa mãn ở vị trí i → tăng giá trị *tmp* thêm 1 đơn vị (đẩy kim sang phải 1 ký tự).
      - Nếu giả sử cây kim vẫn còn khớp với *A* nhưng đã nằm bên phải B? Có nghĩa là hậu tố độ dài |*B*| của *A*{i} đã khớp hoàn toàn với xâu *B*.
        - *Lưu lại vị trí đúng*: i đang biểu diễn vị trí cuối cùng của xâu khớp nên để lưu vị trí đầu ta sẽ lấy (i - |*B*| + 2) (với hệ đếm từ 1) hoặc (i - |*B*| + 1) (với hệ đếm từ 0).
        - Đồng thời đẩy xâu *B* sao cho một phần của *B* nằm trên hoặc bên phải cây kim, tức là tìm tiền tố đúng của *B* còn khớp với hậu tố tương ứng của *A*{i}. Giảm *tmp* bằng công thức: *tmp* = LPS[*tmp*-1].

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - **Do lệnh lặp ở trên nên sẽ xảy ra 2 khả năng:**
    - Nếu không (*B*[*tmp*] không khớp với *A*[i]), tức là *tmp* = 0 và không có tiền tố nào của *B* khớp với hậu tố tương ứng cùng độ dài của xâu *A*{i}. Ta bỏ qua và duyệt tiếp tới i tiếp theo (không cần thay đổi *tmp* vì *tmp* = 0 có nghĩa là ta vẫn đang so sánh *A*[i] mới với vị trí đầu tiên của *B*).

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Xét xâu A = "I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN"
  - Hiển nhiên không thể khớp *A* và *B* với 0 ≤ i ≤ 13

```
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
A = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
B = SEVENTY SEVEN
    0123456789012
```

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Với i = 14 cho tới i = 24, *A* và *B* khớp nhau liên tục. Sau khi xử lý xong i = 24 thì *tmp* = 11

```
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
A = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
B =               SEVENTY SEVEN
                  0123456789012
                            1
```

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Xét i = 25. Ta thấy *A*[25] ≠ *B*[11] (‘ ‘ ≠ ‘E’), nên ta sẽ hạ biến *tmp*:

    → *tmp* = LPS[*tmp*-1] = LPS[11-1] = 3.

```
                 1         2         3         4         5
    012345678901234567890123456789012345678901234567890
A = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
B =                       SEVENTY SEVEN
                          0123456789012
```

A[25] ≠ *B*[3] (‘ ‘≠ ‘E’), nên ta tiếp tục hạ biến *tmp*:

→ *tmp* = LPS[*tmp*-1] = LPS[3-1] = 0.

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - *A*[25] ≠ *B*[0] nên bỏ qua vị trí này và duyệt tiếp. Quá trình bỏ qua tiếp tục diễn ra cho tới i = 30.
  - Duyệt liên tục cho tới i = 42, *tmp* = 12. Tới đây *A*[42] = *B*[12] → là một trường hợp khớp (bắt đầu khớp từ vị trí số 30), *tmp* được đặt là 13 (13 >= |*B*|).

```
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
A = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
B =                               SEVENTY SEVEN
                                  0123456789012
                                            1
```

Xâu con

- Giải bài toán so khớp chuỗi bằng thuật toán Knuth-Morris-Pratt (KMP)
  - Tiếp theo, đẩy xâu *B* tới vị trí gần nhất tiếp theo còn khớp

    → Ta có: *tmp* = LPS[*tmp*-1] = LPS[13-1] = 5.
  - Duyệt liên tục cho tới i = 50, *tmp* = 12. *A*[50] = *B*[12] nên một trường hợp khớp nữa được ghi nhận (bắt đầu khớp từ vị trí số 38). *tmp* lúc này là 13.

```
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
A = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
B =                                       SEVENTY SEVEN
                                          0123456789012
                                                    1
```

Xâu con

- Một số thuật toán so khớp chuỗi khác
  - Thuật toán Rabin-Karp
  - Thuật toán Z
  - Thuật toán Boyer Moore

Sắp xếp xâu

- Bài tập

TỔNG KẾT

1. **Tổng quan**