

CHUYÊN ĐỀ: CÂY TIỀN TỐ - TRIE

MỤC LỤC

I. Mở đầu	4
II. Các kiến thức cơ bản về cây Trie	4
III. Cách cài đặt cây Trie.....	7
3.1 Khai báo cây Trie ban đầu.....	8
3.2 Hàm chèn một xâu vào cây Trie	9
IV. Ứng dụng của cấu trúc dữ liệu Trie trong các bài tập lập trình thi đấu	10
4.1 Dùng Trie để biến đổi từ một bài toán xử lí xâu thành một bài toán trên đồ thị dạng cây.....	10
____ 4.1.1 Bài NPREFIX	10
____ 4.1.1.1 Đề bài.....	10
____ 4.1.1.2 Ý tưởng thuật toán	12
____ 4.1.1.3 Cách cài đặt cụ thể.....	12
____ 4.1.1.4 Nhận xét chung về bài toán :.....	14
____ 4.1.2 Bài PREFIX.....	14
____ 4.1.2.1 Đề bài.....	14
____ 4.1.2.2 Ý tưởng thuật toán	14
____ 4.1.2.3 Cách cài đặt chi tiết.....	15
____ 4.1.2.4 Nhận xét chung về bài toán	16
____ 4.1.3 Bài Printer.....	16
____ 4.1.3.1 Đề bài :.....	16
____ 4.1.3.2 Ý tưởng thuật toán	17

4.1.3.3 Cách cài đặt cụ thể	19
4.1.3.4 Nhận xét chung về bài toán	21
4.1.4 Bài Sr2n	21
4.1.4.1 Đề bài :	21
4.1.4.2 Ý tưởng thuật toán	22
4.1.4.3 Cách cài đặt cụ thể	23
4.1.4.4 Nhận xét chung về bài toán	26
4.1.5 Bài tập vận dụng	26
4.2 Trie được dùng để quản lý tập hợp số	26
4.2.1 Bài ORDSET	26
4.2.1.1 Đề bài :	26
4.2.1.2 Ý tưởng thuật toán	28
4.2.1.3 Cách cài đặt chi tiết	29
4.2.1.4 Nhận xét chung về bài toán	33
4.2.2 Bài XOR	33
4.2.2.1 Đề bài	33
4.2.2.2 Ý tưởng thuật toán	34
4.2.2.3 Cách cài đặt chi tiết	34
4.2.2.4 Nhận xét chung về bài toán	35
4.2.3 Bài tập vận dụng	36
4.3 Trie hỗ trợ cho các thuật toán quy hoạch động	36
4.3.1 Bài Chia Dãy	36
4.3.1.1 Đề bài	36

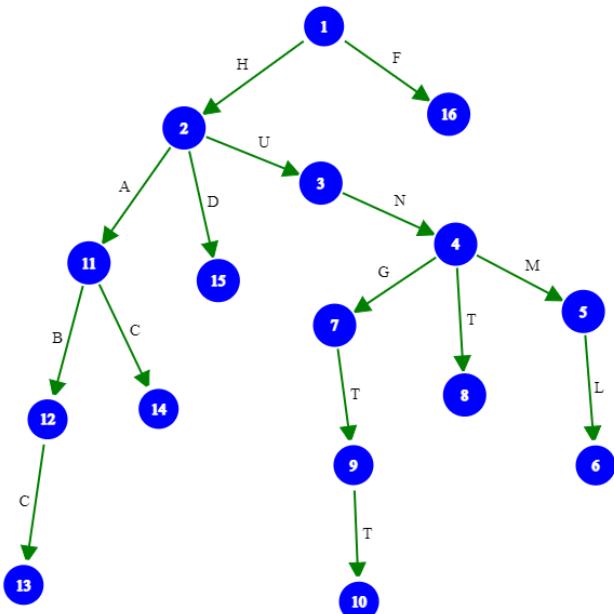
4.3.1.2 Ý tưởng thuật toán	38
4.3.1.3 Cách cài đặt chi tiết.....	40
4.3.1.4 Nhận xét về bài toán.....	43
4.3.2 Bài SEPARATE.....	43
4.3.2.1 Đề bài.....	43
4.3.2.2 Ý tưởng thuật toán	44
4.3.2.3 Cách cài đặt chi tiết.....	45
4.3.2.4 Nhận xét chung về bài toán	46
4.3.3 Bài SEED	47
4.3.3.1 Đề bài.....	47
4.3.3.2 Ý tưởng thuật toán	48
4.3.3.3 Cách cài đặt chi tiết.....	49
4.3.3.4 Nhận xét chung về bài toán	52
4.3.4 Bài tập vận dụng.....	51
V. KẾT LUẬN	52
VI. TÀI LIỆU THAM KHẢO.....	53

I. MỞ ĐẦU

Trie là một cấu trúc dữ liệu quan trọng , xuất hiện khá nhiều trong các bài tập hay các kì thi HSG môn Tin học. Đây là một cấu trúc dữ liệu giúp chúng ta quản lý một tập hợp các xâu kí tự. Chúng ta có thể thực hiện các thao tác như thêm một xâu vào tập hợp , xoá một xâu khỏi tập hợp hay kiểm tra xem một xâu nào đó có thuộc tập hợp hay không..... Đây là một cấu trúc rất hữu ích trong các bài toán về xử lí xâu , quy hoạch động,.....

II. CÁC KIẾN THỨC CƠ BẢN VỀ CÂY TRIE

Có thể hình dung cây Trie cơ bản có hình dạng giống bất kì một loại cây nào đó trong đời thực. Nó cũng có các quan hệ cha con , tổ tiên và có một nút gốc duy nhất. Mỗi cạnh trên cây Trie sẽ lưu 1 kí tự duy nhất , mỗi nút trên cây Trie sẽ thể hiện cho một xâu kí tự, xâu này được ghép bởi các kí tự nằm trên đường đi từ nút gốc tới nút đó. Để dễ hiểu chúng ta có thể nhìn vào ví dụ mẫu cây Trie dưới đây :



Ta thấy: nút 1 là nút gốc của cây , trên cạnh (1,2) lưu kí tự duy nhất “H” , trên cạnh (7,9) lưu kí tự “T” . Với nút 12 xâu kí tự ở đây sẽ là “HAB” , trong khi đó nút 15 xâu kí tự sẽ là “HD” , nút 16 sẽ biểu diễn cho xâu “F” và nút 1 sẽ biểu diễn cho xâu rỗng.

Các xâu có chung một phần tiền tố thì phần tiền tố chung nhau đó sẽ được biểu diễn giống nhau trên cây Trie. Ví dụ với hình vẽ trên, xâu “HUNT” và xâu “HUNM” có chung phần tiền tố “HUN” chính là phần chung nhau trên đường đi từ nút gốc tới nút 7 với đường đi từ nút gốc tới nút 5. Nói cách khác phần tiền tố chung nhau này chính là xâu được biểu diễn bởi LCA của 2 nút này (LCA là cha chung gần nhất của 2 nút trên cây) . Ví dụ nút 7 và nút 5 là $LCA(7,5)=4$ và xâu được biểu diễn bởi nút 4 chính là xâu “HUN”.

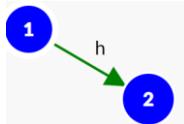
Tuỳ vào từng bài mà có các hàm xử lí khác nhau trên cây Trie , tuy nhiên thường sẽ có 2 hàm chính là thêm 1 xâu kí tự vào cây và tìm vị trí biểu diễn của một xâu bất kì trên cây Trie. Một cây Trie khi chưa được chèn thêm bất kì chuỗi kí tự nào sẽ chỉ có đúng 1 nút duy nhất là nút gốc .

a. Thao tác chèn 1 xâu kí tự vào cây

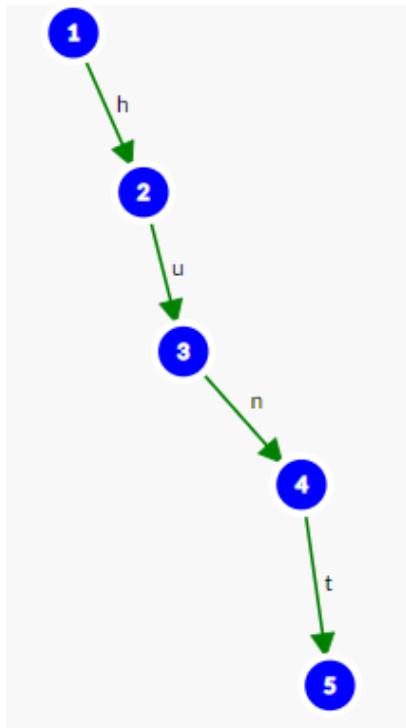
Với hàm thêm 1 xâu kí tự **S** vào cây ý tưởng chính là chúng ta sẽ xuất phát tại nút gốc và đồng thời duyệt mỗi kí tự của **S**. Tại nút trên cây hiện tại là **x** và kí tự đang duyệt là **S[i]** ,ta xét xem đã có nhánh mang nhãn là kí tự **S[i]** hay chưa , nếu chưa có ta sẽ tạo nhánh mới mang nhãn là kí tự **S[i]** ; sau đó đi xuống con của **x** theo nhánh con mang nhãn là kí tự **S[i]** . Để dễ tưởng tượng hãy xem ví dụ dưới đây :

Cây Trie ban đầu rỗng chỉ có đúng một nút là nút gốc :

Xét cây Trie khi chèn thêm xâu “hunt” , tại nút 1 ta xét xem đã tồn tại nhánh con nào của 1 mang kí tự “h” hay chưa. Do chưa có nên ta sẽ tạo thêm nhánh mới và một đỉnh mới (đánh chỉ số đỉnh này là 2)



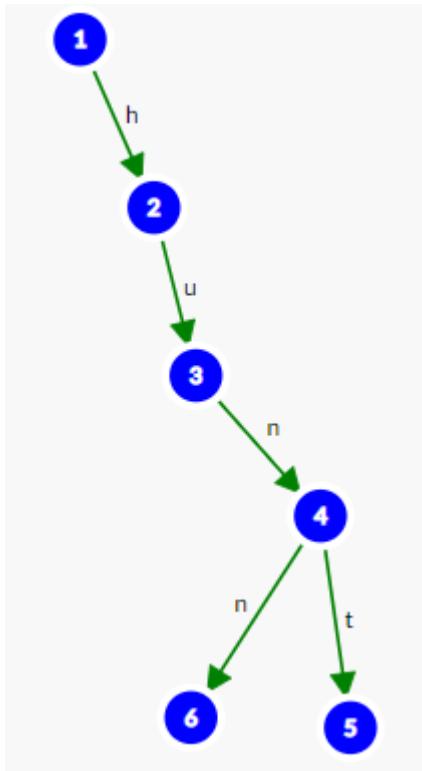
Sau đó ta sẽ nhảy xuống đỉnh 2 theo cạnh (1,2) vì cạnh (1,2) lúc này mang kí tự “h”. Tại đỉnh 2 chưa có cạnh nào mang kí tự “u” nên tiếp tục phải tạo một đỉnh mới và cạnh mới . Tiếp tục làm vậy sau khi chèn xong xâu kí tự “hunt” vào cây Trie ta sẽ xâu được cây có hình dạng như sau :



Xét tiếp với cây Trie trên khi chèn thêm xâu “hunn” . Ta tiếp tục xuất phát tại nút gốc 1 khi đó đã tìm thấy 1 nhánh con mang kí tự “h” ta nhảy xuống qua nhánh này tức là nhảy từ đỉnh 1 xuống đỉnh 2. Tại đỉnh 2 tiếp tục tìm thấy nhánh con mang kí tự “u” do đó nhảy xuống đỉnh 3. Tại đỉnh 3 tiếp tục tìm thấy nhánh con mang kí tự

“n” ta lại nhảy xuống đỉnh 4. Tại đỉnh 4 không tìm thấy nhánh con mang kí tự “n” ta sẽ tạo một nhánh và một đỉnh mới .

Sau 2 lần chèn 2 xâu “hunt” và “hunn” cây Trie sẽ biến thành :



Như vậy ta sẽ có đỉnh 6 biểu diễn cho xâu “hunn” và đỉnh 5 biểu diễn cho xâu “hunt”.

(Cách cài đặt chi tiết sẽ được trình bày ở phần cài đặt cây bên dưới)

b. Thao tác tìm nút biểu diễn trên cây Trie cho xâu kí tự cho trước

Khá giống với thao tác khi chèn xâu kí tự mới, ta chỉ việc nhảy từ nút gốc xuống đi theo các cạnh mang các kí tự đang xét , kết thúc ở đâu thì nút đó sẽ biểu diễn cho xâu kí tự đó. Nếu kết thúc tại một nút rỗng thì chứng tỏ xâu đó không tồn tại trong Trie.

III. CÁCH CÀI ĐẶT CÂY TRIE BAN ĐẦU

Để biểu diễn Trie trong các bài toán lập trình thi đấu, thông thường có 2 cách biểu diễn chính là dùng mảng thông thường hoặc con trỏ.

Mảng là cấu trúc rất đơn giản và thông dụng nên hầu hết các bạn học sinh đều biết sử dụng mảng trong khi con trỏ lại là một kiến thức mà khá nhiều bạn học sinh bỏ qua. Chính vì thế có khá nhiều bạn biểu diễn Trie bằng mảng thông thường tuy nhiên dùng con trỏ ở đây lại có lợi hơn rất nhiều so với việc dùng mảng. Cả 2 cách biểu diễn bằng mảng hay bằng con trỏ đều có tư tưởng thuật toán giống nhau nhưng khi biểu diễn Trie bằng con trỏ sẽ tối ưu hơn khá nhiều về mặt bộ nhớ, cũng như giúp bạn có một đoạn code đẹp hơn, gọn gàng hơn rất nhiều so với dùng mảng . Hơn nữa khi biểu diễn Trie bằng con trỏ , chúng ta cũng chỉ cần biết sơ qua về cách sử dụng con trỏ trong lập trình thi đấu chứ không nhất thiết phải nắm quá rõ về con trỏ (để có một cái nhìn toàn diện về con trỏ trong các ngôn ngữ lập trình thì cần có kiến thức về kiến trúc máy tính, hệ điều hành, cách lưu trữ biến trong bộ nhớ). Do vậy ở đây tôi sẽ trình bày cách biểu diễn Trie bằng con trỏ .

(Để tìm hiểu nhiều hơn về cách sử dụng con trỏ trong lập trình thi đấu có thể tìm hiểu trong bài viết ở link sau : <https://vnoi.info/wiki/languages/cpp/pointers.md>)

3.1. Khai báo cây Trie ban đầu

Trước hết chúng ta sẽ khai báo một struct Node để miêu tả cho một nút trên cây Trie như sau :

```
struct Node
```

```
{
```

```
    Node *child[256] ; // tại một nút trên cây Trie sẽ có 256 nút con với trường hợp các xâu có đủ 256 ký tự trong bảng mã ASCII , hoặc có thể là 2 với trường hợp ta chỉ cần lưu các xâu nhị phân , 26 với trường hợp các xâu cần lưu chỉ có các ký tự 'a,b,c,...,z' .
```

```

    // các nút con của nút hiện tại ta đang xét cũng được khai báo dưới dạng con
    // trỏ , và các nút này cũng có thể rỗng.

    // trên một nút ta có thể lưu tuỳ ý các thông tin tuỳ thuộc vào các bài toán khác
    // nhau, ta có thể lưu độ sâu của nút hay biến kiểm tra xem có xâu nào kết thúc tại nút
    // này hay không.....
```

```

Node() {
    memset(child, NULL, sizeof child);

    // đây là hàm tạo ra một nút mới nên với tất cả các nút con của nút này
    // chưa tồn tại trong cây nên ta sẽ biểu diễn chúng bằng một con trỏ rỗng
}
```

Ban đầu cây Trie chỉ có đúng một nút là nút gốc , ta gọi nút này là nút gốc và khai báo đó như sau :

```
Node *root= new Node();
```

Để gọi ra 1 hàm hay một biến nằm trong một con trỏ p bắt kì ta sẽ dùng dấu mũi tên “->” , ví dụ root->child[k] .

3.2. Hàm chèn một xâu vào cây Trie

Đúng với tư tưởng thuật toán đã được trình bày ở mục 2 , chúng ta có một đoạn code chèn một xâu vào cây Trie như sau :

```

void addword ( const string & s)
{
    Node *p= root; // khai báo một con trỏ p và gán nó bằng root
```

```

For(int i=0 ; i< (int) s.size() ; ++i)

{

    If( p -> child[s[i]] == NULL) p->child[s[i]] = new Node();

    // tức là khi đó chưa có nhánh từ p trỏ xuống con mang kí tự s[i] , khi
    // đó ta sẽ tạo một nhánh con mới cho p mang kí tự s[i]

    p=p->child[s[i]];

    // từ p ta sẽ đi theo nhánh mang kí tự s[i] xuống con của nó

}

}

```

Sau khi thực hiện các hàm addword chúng ta sẽ có một cây Trie hoàn chỉnh trong đó các nút trên cây được biểu diễn bằng một con trỏ , nút cha trỏ xuống các nút con , $p->child[x]$ tức là con của nút p theo nhánh mang kí tự x .

Cần được nhắc lại rằng bên trên chỉ là cách tạo cây Trie cơ bản , còn phụ thuộc vào các bài toán để lưu thêm các thông tin tại mỗi nút trên cây .

IV. ỨNG DỤNG CỦA CẤU TRÚC DỮ LIỆU TRIE.

4.1. Dùng Trie để biến đổi từ một bài toán xử lí xâu thành một bài toán trên đồ thị dạng cây

4.1.1. Bài NPREFIX

4.1.1.1. Đề bài

Cho tập S gồm N xâu và M truy vấn. Với mỗi truy vấn là một xâu X , ta cần xác định số lượng xâu trong tập S nhận X là tiền tố.

Ta nói X là tiền tố của xâu Y nếu như: $|X| \leq |Y|$ và $X_i = Y_i \forall i \in [1, |X|]$.

Hãy lập trình giải bài toán trên.

Input: Dòng đầu chứa ba số nguyên dương N ($N \leq 10^4$).

- N dòng tiếp, dòng thứ i chứa xâu S_i .
- Dòng tiếp theo ghi số nguyên dương M ($M \leq 10^4$)
- M dòng tiếp, mỗi dòng chứa một xâu X thể hiện truy vấn

Output: M dòng tương ứng là kết quả từng truy vấn.

NPREFIX.INP	NPREFIX.OUT
6	4
abc	3
abdh	1
acfī	2
bgjab	0
abe	
bcf	
5	
a	
ab	
abc	
b	
bj	

Các xâu chỉ gồm các chữ cái in thường trong tập “abcdefghijkl”. Tổng số ký tự trong tất cả các xâu không quá 10^6 .

4.1.1.2 Ý tưởng thuật toán

Ta sẽ dùng một cây Trie để quản lý tập hợp xâu S .

Nhận xét : để một xâu s là tiền tố của xâu t thì nút biểu diễn trên cây Trie phải là tổ tiên của nút t trên cây Trie. Từ đây để xác định có bao nhiêu xâu trong tập S nhận X là tiền tố, ta chỉ cần đếm số xâu có nút biểu diễn là con của nút biểu diễn xâu X hay nói cách khác là đếm số xâu trong tập S đi qua nút biểu diễn xâu X trên cây Trie.

Độ phức tạp của thuật toán này chính bằng tổng độ dài của tất cả các xâu trong tập S và trong các truy vấn.

4.1.1.3 Cài đặt

Khai báo một nút trên cây Trie và thêm thông tin cnt để đếm số xâu đi qua nút này

Struct Node

{

 Node *child[10] ; // mỗi nút chỉ có tối đa 10 nút con do các xâu trong tập S chỉ có các ký tự thuộc tập hợp “abcdefghijkl”

 int cnt ; // đếm số lượng số xâu đi qua nút này

 Node()

{

 Memset(child,NULL, sizeof child) ;

 cnt=0 ;

}

}

Khi thêm xâu trong tập S vào trong cây Trie ta làm tương tự với cách chèn xâu bình thường và tại mỗi nút p trên đường đi của xâu đó sẽ tăng *cnt* lên 1.

(tại một nút p bất kì muốn truy cập vào giá trị *cnt* của nó ta sẽ gọi *p->cnt*)

Hàm tính kết quả rất đơn giản chỉ cần đi theo đường đi của xâu X trên cây Trie và in ra *cnt* của nút cuối cùng :

```
int get(const string & X)

{
    Node *p=root ;

    For(int i=0;i<(int)X.size();++i)

    {
        Int k= X[i]-‘a’ ;

        If(p->child[k]==NULL) return 0 ; // do không có nút nào biểu diễn cho
        xâu X trên cây Trie nên ta sẽ trả ra kết quả 0

        p=p->child[k] ;

    }

    return p->cnt ;
}
```

Code mẫu <https://ideone.com/tGIR60>

Bô test

<https://www.dropbox.com/sh/hlu4yn7matuzt69/AAAKp0F2S6FAqHZj43VShB-Va?dl=0>

4.1.1.4. Nhận xét chung về bài toán :

Đây là một bài cơ bản về cách xử lí xâu kết hợp với cây Trie , áp dụng cho rất nhiều các bài toán về xâu khác. Các mối quan hệ về tiền tố, hậu tố trên cây Trie thành các mối quan hệ cha con trên bài toán đồ thị.

4.1.2. Bài PREFIX

4.1.2.1. Đề bài

Một xâu được gọi là xâu tiền tố của một xâu khác nếu nó xuất hiện ở vị trí đầu tiên của xâu này. Ví dụ xâu 'ab' là tiền tố của xâu 'abcd'; 'aa' là tiền tố của 'aa'

Yêu cầu: Cho n xâu ký tự, hãy đếm số cặp xâu mà xâu này là tiền tố của xâu còn lại

Input:

- Dòng đầu tiên ghi số nguyên dương n ($1 \leq n \leq 10^6$)
- n dòng tiếp theo, mỗi dòng ghi một xâu ký tự chỉ gồm các chữ cái tiếng Anh in thường với độ dài của mỗi xâu không vượt quá 10

Output: In ra một số nguyên duy nhất là số lượng xâu tìm được

PREFIX.inp	PREFIX.out
4	
abc	
aa	
aab	
aa	

4.1.2.2 Ý tưởng thuật toán

Ta tiếp tục với nhận xét từ bài NPREFIX, bài PREFIX này có thể chuyển thành bài toán đếm số cặp đỉnh mà đỉnh này là tổ tiên của đỉnh kia trên cây Trie . Do đó tại

nút p trên cây Trie có *last* xâu kết thúc tại đây (hay có last xâu được biểu diễn trên cây Trie bởi nút p) và *cnt* là số xâu đi qua nút p trên cây Trie thì ta sẽ cập nhật vào kết quả một lượng *last*cnt* .

Độ phức tạp của thuật toán này chính bằng tổng độ dài của *n* xâu kí tự đã cho

4.1.2.3 Cài đặt

Trong hàm thêm một xâu kí tự vào cây Trie ta sẽ cập nhật tất cả *cnt* của các nút trên đường đi của xâu này lên 1 , và tại nút cuối cùng ta sẽ cập nhật *last* của nó lên 1.

Sau khi thêm hết tất cả các xâu kí tự vào bên trong cây Trie ta sẽ thực hiện *dfs* trên cây và tính kết quả của bài toán

```
Void dfs(Node *p)
{
    For(int i=0;i<=25;++i)
        If(p->child[i]!=NULL)
            dfs(p->child[i]) ;
    ans+=p->last*p->cnt ; // cập nhật vào kết quả
}
```

Code mẫu tham khảo

<https://ideone.com/kqayZc>

Bô test

<https://www.dropbox.com/sh/o1l4ydgpz0g6a3g/AACz7R-zW3XkEfVvx7YnIGBwa?dl=0>

4.1.2.4 Nhận xét chung về bài toán

Giống như bài NPREFIX ở trên , bài PREFIX này tiếp tục nhấn mạnh cho chúng ta về tư duy các bài toán xử lí xâu trên cây Trie. Mỗi nút trên cây Trie của bài này lưu thêm những thông tin cần thiết, và chúng ta có thể dfs trên cây Trie giống với tư tưởng dfs trên cây của đồ thị thông thường.

4.1.3 Bài Printer

4.1.3.1 Đề bài :

Bạn có một máy in và cần in n từ để ghép thành một khâu hiệu. Máy in có một khay để xếp các miếng kim loại (mỗi miếng chứa một kí tự) để tạo thành từ. Ban đầu khay rỗng, mỗi lượt bạn được thao tác một trong ba loại sau:

- Xếp thêm một miếng kim loại chứa một kí tự vào cuối khay;
- Loại bỏ một miếng kim loại chứa một kí tự ở cuối khay;
- In ra từ được tạo bởi các kí tự trên khay.

Vào cuối quá trình in, bạn được phép để lại các miếng kim loại trên khay, ngoài ra bạn được phép in các từ theo bất kì thứ tự nào.

Yêu cầu: Cho n từ, hãy tính số thao tác ít nhất để in được n từ đó.

Input :

- Dòng đầu chứa số nguyên dương n ;
- Tiếp theo là n dòng, mỗi dòng chứa một từ cần in, các từ chỉ gồm các kí tự ‘a’ đến ‘z’ và tổng số các kí tự không vượt quá 1000000

Output :

- Gồm một dòng, chứa số thao tác ít nhất cần thực hiện để in n từ trên.

Subtask :

- Subtask 1 : $n \leq 20$
- Subtask 2: Không có ràng buộc gì thêm

Printer.inp	Printer.out
<pre>3 print the poem</pre>	20

4.1.3.2 Ý tưởng thuật toán

Nhận xét chung : do số thao tác 3 luôn bằng n nên ta chỉ cần quan tâm đến các thao tác loại 1 và thao tác loại 2

Subtask 1 :

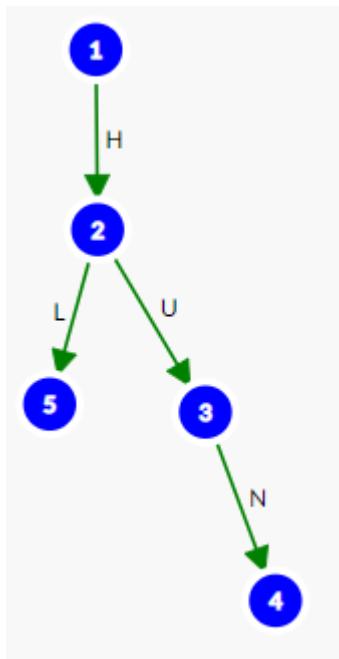
- Nhận xét : nếu xâu p được in ra ngay sau xâu s thì ta chỉ cần xoá xâu s cho đến khi xâu trong khay là một tiền tố của xâu p và thêm tiếp các kí tự ở phần sau của xâu p . Tức là khi chuyển từ xâu s sang p trong khay tổng số thao tác chèn, xoá ít nhất mà chúng ta cần thực hiện chính là tổng độ dài 2 xâu s và p trừ đi 2 lần độ dài tiền tố chung dài nhất của s và p . Do đó Ta hoàn toàn có thể chuẩn bị trước một mảng $trans[i][j]$ là số thao tác chèn,xoá cần thực hiện khi chuyển từ xâu s sang xâu p .
 - Từ nhận xét trên kết hợp với ràng buộc $n \leq 20$, ta sẽ có một thuật toán quy hoạch động trạng thái rất cơ bản như sau : gọi $f[mask][i]$ là số thao tác chèn,xoá ít nhất sao cho đã in được các xâu j nếu bit thứ j của $mask$ bằng 1, và xâu đang còn lại trong khay hiện tại là xâu thứ i .
 - Công thức chuyển trạng thái sẽ là :
- $$f[mask \wedge (1 << j)][j] = \min(f[mask \wedge (1 << j)][j], f[mask][j] + trans[i][j])$$

thoả mãn bit thứ j của *mask* bằng 0 .

- Khi đó kết quả của bài toán là $\min(f[(1 << n) - 1][i])$ với mọi i chạy từ 0 đến $n-1$ (do ta đánh chỉ số các xâu từ 0 đến $n-1$)

Subtask 2:

- Nhận xét quan trọng : các thao tác chèn hoặc xoá các kí tự có thể được hình dung như một phép nhảy trên cây Trie lưu n từ này . Ta sẽ xem một xâu đang có trong khay được biểu diễn thành một nút trên cây . Một thao tác chèn thêm một kí tự vào trong khay giống như một bước nhảy từ nút cha tới nút con trên cây Trie , ngược lại thao tác xoá một kí tự trong khay lại giống như một phép nhảy từ nút con tới nút cha . Để dễ hình dung chúng ta xét ví dụ với một cây Trie như sau:



- Giả sử trong khay hiện tại là xâu “HU” khi đó chúng ta đang đứng ở nút thứ 3 trên cây Trie , nếu muốn chèn thêm kí tự “N” nó sẽ giống với thao tác di chuyển từ nút 3 xuống nút 4 để thu được xâu “HUN” , nếu muốn xoá kí tự ở cuối đi (kí tự “N”) sẽ là phép nhảy từ con lên cha hay từ nút 4 quay về nút 3.

- Do ta cần in hết cả N ký tự nên ta cần di chuyển tới mọi nút lá trên cây Trie , nói cách khác từ đây bài toán về xâu chuyển thành 1 bài toán về đồ thị được phát biểu lại như sau: cho một đơn đồ thị vô hướng dạng cây , đếm số bước di chuyển ít nhất từ nút gốc đến tất cả các đỉnh còn lại , mỗi đỉnh có thể được đi qua nhiều lần. Và kết quả của bài toán này chính là $2^*(\text{số cạnh trên cây}) - (\text{đường đi dài nhất trên cây xuất phát tại nút gốc})$. Chú ý đồ thị trong hình vẽ ví dụ có định chiều các cạnh nhưng trên thực tế ta có thể đi ngược lại chiều đó trong các thao tác xoá kí tự nên có thể coi cây Trie ta tạo ra là 1 đơn đồ thị vô hướng dạng cây.
- Độ phức tạp của thuật toán này chính bằng tổng độ dài của n xâu kí tự đã cho

4.1.3.3 Cách cài đặt cụ thể

Trước hết chúng ta cần tạo ra một cây Trie chứa các xâu của đề bài . Trong quá trình thêm các xâu kí tự vào cây , ta hoàn toàn có thể đếm được số cạnh của cây Trie này và tính đường đi dài nhất trên cây xuất phát tại nút gốc (đường đi dài nhất trên cây xuất phát tại nút gốc chính là độ sâu dài nhất của mọi nút trên cây).

Để hiểu rõ hơn hãy đọc đoạn mã giả tham khảo dưới đây :

```
Struct Node
{
    Node *child[26] ;
    int dep ; // lưu độ xâu của nút này trên cây trie
    Node()
    {
        Memset(child, NULL , sizeof child)
    }
}
```

```

    dep=0 ;

}

}

Node * root= new Node() ;

int cntEdge=0 ; // đếm số cạnh của cây Trie

int maxDep=0 ; // tính độ sâu lớn nhất của cây Trie

Void addword (const string & s)

{

    Node *p = root ;

    For(int i=0;i< (int) s.size(); ++i)

    {

        If(p->child[i]==NULL)

        {

            p->child[i]=new Node() ;

            ++cntEdge ; // do lúc này cây Trie được tạo thêm một cạnh mới

            p->child[i]->dep=p->dep+1 ; // tính độ sâu cho nút mới

            maxDep=max(maxDep,p->child[i]->dep) ;

        }

        p= p->child[i] ;

    }

}

```

Như vậy chúng ta sẽ thu được đáp án là $2 * \text{cntEdge} - \text{maxDep} + n$

Code mẫu : <https://ideone.com/pxIYj8>

Bộ test :

https://www.dropbox.com/sh/wwcjx8vn8vhuj88/AACprGxSIpLdiz5_MYNia3AZa?dl=0

4.1.3.4 Nhận xét chung về bài toán

Bài toán cho ta thấy có thể nhìn các thao tác xoá một kí tự ở cuối, chèn thêm một kí tự vào cuối giống như các phép di chuyển giữa các nút trên cây Trie. Từ đó chúng ta có thể chuyển đổi hoàn toàn một bài toán về xâu thành một bài toán trên đồ thị dạng cây đơn thuần. Lúc này sẽ cần các tư duy giải quyết bài toán trên đồ thị dạng cây để giải bài toán này và không liên quan tới các tư duy xử lý xâu. Độ khó của bài toán lúc này phụ thuộc vào độ khó của bài toán con trên cây.

4.1.4. Bài Sr2n

4.1.4.1 Đề bài :

Khi học về xâu kí tự, để luyện tập thêm về nội dung này, An và Bình cùng nhau chơi một trò chơi với các xâu kí tự như sau:

- An tạo ra n xâu kí tự ngẫu nhiên, sau đó, mỗi xâu ban đầu tạo ra một xâu mới bằng cách sao chép một đoạn đầu (hoặc toàn bộ) của xâu đó để tạo thêm được n xâu.
- Với $2n$ xâu mà An tạo ra và được đánh số theo thứ ngẫu nhiên từ 1 đến n , Bình cần đưa ra một phương án để giải thích cách tạo xâu của An.

Yêu cầu: Cho $2n$ xâu, hãy chia $2n$ xâu thành n nhóm, mỗi nhóm gồm hai xâu mà xâu này là đoạn đầu (hoặc toàn bộ) của xâu kia.

Input :

- Dòng đầu chứa số nguyên dương n ;
- Tiếp theo là $2n$ dòng, mỗi dòng là một xâu chỉ gồm các kí tự ‘a’ đến ‘z’.
Tổng số kí tự trong file không vượt quá 10^6 .

Output :

- Gồm n dòng, mỗi dòng chứa hai số là chỉ số xâu gốc và chỉ số xâu được tạo ra.

Str2n.inp	Str2n.out
2	1 3
ab	4 2
adc	
a	
adce	

Subtask :

- Subtask 1 : $n \leq 10$
- Subtask 2 : không có ràng buộc gì thêm .

4.1.4.2 Ý tưởng thuật toán

Subtask 1 :

Ta có mảng $f[]$ như sau : gọi tập hợp các vị trí bit 1 trong mask là (x_1, x_2, \dots, x_k) , $f[mask] \geq 0$ nếu ta có thể xếp (x_1, x_2, \dots, x_k) vào $k/2$ nhóm và k chẵn , $f[mask] = -1$ nếu không thể xếp .

Cách tính mảng $f[]$: thực hiện giống như các bài quy hoạch động trạng thái , ban đầu gán toàn bộ mảng $f[]$ mang giá trị -1 và chỉ có $f[0] = 0$. Với 1 cấu hình mask và $f[mask] \neq -1$ đang xét ta thực hiện duyệt các vị trí bit 0 trong mask với 2 biến i, j .

Nếu i có thể ghép cùng nhóm với j ta sẽ gán $f[mask \wedge (1 << i) \wedge (1 << j)] = mask$ có nghĩa là ta thực hiện ghép i, j vào chung 1 nhóm kết hợp với những nhóm đã ghép ở trước.

Sau khi tính được mảng $f[]$ rất dễ dàng để thực hiện in ra các nhóm theo yêu cầu đề bài bằng cách truy vết từ $(1 << n) - 1$ về 0.

Độ phức tạp của thuật toán này sấp sỉ $O((n * 2)^2 * 2^{n^2})$

Subtask 2:

Dựng cây Trie lưu $2 * n$ xâu kí tự , tại mỗi nút trên cây ta dùng 1 vector để lưu các vị trí có xâu kết thúc tại nút đó . Từ đây bài toán trên sẽ quy về 1 bài toán trên đồ thị dạng cây và được phát biểu lại như sau : Cho 1 đơn đồ thị dạng cây , tại mỗi đỉnh có 1 tập hợp các số khác nhau , ta cần tạo ra n cặp ghép sao cho x, y được ghép với nhau khi và chỉ khi nếu x thuộc đỉnh u , y thuộc đỉnh v thì u và v có quan hệ cha con trên cây hoặc $u = v$. Ta sẽ thực hiện dfs trên cây và sử dụng 1 $stack$, khi thăm đỉnh u ta sẽ $push$ tất cả các số vào bên trong $stack$. Khi đã thăm xong đỉnh u , ta sẽ thực hiện ghép cặp cho tất cả các số trong cây con gốc u vẫn còn đang nằm trong $stack$ với nhau, các số này chính là các số đang nằm ở đầu $stack$ (vì ta $push$ chúng vào $stack$ theo thứ tự dfs) , khi đã được ghép thì loại nó khỏi $stack$. Do ta push các số theo thứ tự dfs và khi dfs xong cây con gốc u ta đã ghép cặp cho tất cả các số bên trong cây con gốc u chưa được ghép rồi bỏ nó ra khỏi $stack$ nên trong mọi thời điểm các nút trong $stack$ luôn có quan hệ cha con (có thể là cha không trực tiếp) .

Độ phức tạp của thuật toán này chính bằng tổng độ dài của $2 * n$ xâu kí tự đã cho

4.1.4.3 Cách cài đặt cụ thể

Tại mỗi nút trên cây Trie ta sẽ lưu thêm một tập hợp chứa các vị trí có xâu được biểu diễn trên cây bởi nút này

Struct Node

```
{  
  
    Node *child[26] ;  
  
    Vector<int> last ; // lưu các chỉ số i mà xâu s[i] có kết thúc tại đây , hay  
    xâu s[i] được biểu diễn bởi nút này trên cây  
  
    Node ()  
  
    {  
  
        last=vector<int> () ; // gán last ban đầu bằng 1 vector rỗng  
        memset(child,NULL, sizeof child) ;  
  
    }  
  
}
```

Tại hàm chèn xâu ta chỉ việc push_back vào last của nút cuối cùng như sau:

```
Void addword(const string &s,int pos) // truyền vào xâu s tại vị trí pos  
  
{  
  
    Node *p=root ;  
  
    For(int i=0;i<(int)s.size();++i)  
  
    {  
  
        Int k=s[i]-‘a’ ;  
  
        If(p->child[k]==NULL) p->child[k]=new Node() ;  
  
        p=p->child[k] ;  
  
    }
```

```

    p->last.push_back(pos) ; // nút p hiện tại chính là nút biểu diễn cho xâu s ,
khi đó ta sẽ cần thêm pos vào trong vector<int> last của nút này

}

```

Dưới đây sẽ là hàm dfs và thực hiện ghép cặp các nút trên cây :

```

Pair<int,int> st[]; int top ;

Void dfs(Node *p,int level) // level là tầng của nút p, ví dụ nút gốc có tầng là 0 ,
nút con trực tiếp của nút gốc sẽ có tầng là 1

{
    For(auto x: p->last)

        st[++top]=x ; // đây tất cả chỉ số xâu kết thúc tại nút này vào trong
stack

    For(i,0,25)

        If(p->child[i]!=NULL)

            dfs(p->child[i],level+1) ; // dfs tới tất cả các con của nút p hiện
tại

    // sau khi đã duyệt sang tất cả các nút trong cây con gốc p hiện tại ta sẽ thực hiện
ghép cặp cho các cặp số hiện tại

    While(top>=2 && st[top].second>=level) // ghép đỉnh cho đến khi nào
trong stack vẫn còn 2 phần tử và st[top].second>=level tức là st[top].first vẫn đang
nằm trong cây con gốc u

    {
        Ans.push_back({st[top].first,st[top-1].first}) ; // thêm vào kết
quả cặp này
    }
}

```

```
    top-=2 ; // loại cắp này ra khỏi stack  
}  
}
```

Code mẫu : <https://ideone.com/puYMNr>

Bộ test : https://www.dropbox.com/sh/mqtlcejxzbm9ynq/AACPhZqVf_qjYCI-tnyWL7mfa?dl=0

4.1.4.4. Nhận xét chung về bài toán

Giống với bài Printer , bài toán này tiếp tục biến đổi hoàn toàn một bài toán xử lí xâu về một bài toán trên đồ thị dạng cây. Tuy nhiên bài này có mức độ khó cao hơn bài Printer do bài toán con trên đồ thị khó hơn , yêu cầu phải có các kiến thức, kỹ năng giải quyết trên đồ thị dạng cây.

4.1.5. Bài tập vận dụng

<https://codeforces.com/problemset/problem/965/E>

<https://oj.vnoi.info/problem/sec>

<https://oj.vnoi.info/problem/npr>

<https://codeforces.com/problemset/problem/888/G>

4.2. Trie được dùng để quản lý tập hợp số

4.2.1. Bài ORDSET

4.2.1.1 Đề bài :

Bạn cần quản lý một tập hợp động S hỗ trợ hai thao tác cơ bản:

- $INSERT(S, x)$: Nếu x không thuộc S , thêm x vào S
- $DELETE(S, x)$: Nếu x thuộc S , xóa x khỏi S

và hai loại truy vấn:

- $K\text{-TH}(S)$: Trả về số bé thứ k của S
- $COUNT(S, x)$: Đếm số lượng số thuộc S bé hơn x

Input:

- Dòng đầu tiên ghi Q ($1 \leq Q \leq 200000$) - số thao tác
- Q dòng sau, đầu mỗi dòng chứa ký tự I , D , K hoặc C cho biết thao tác tương ứng là $INSERT$, $DELETE$, $K\text{-TH}$ hay $COUNT$. Tiếp theo là một khoảng trắng và một số nguyên là tham số cho thao tác đó.

Nếu tham số là x , dữ liệu đảm bảo $|x| \leq 10^9$. Nếu tham số là chỉ số k dữ liệu đảm bảo $1 \leq k \leq 10^9$

Output: Với mỗi truy vấn in kết quả trên một dòng. Với truy vấn $K\text{-TH}$, nếu k lớn hơn số phần tử của S , in ra “Invalid”

ORDSET.inp	ORDSET.out
8	1
I -1	2
I -1	2
I 2	Invalid
C 0	
K 2	
D -1	
K 1	
K 2	

4.2.1.2 Ý tưởng thuật toán

Trước hết chúng ta cần biết tới cách lưu các số trong cây Trie , có thể lưu bằng nhiều cách khác nhau như lưu một xâu ở dạng thập phân , tam phân... Cách làm thường được sử dụng chính là lưu các số dưới dạng xâu nhị phân. Tuỳ thuộc vào giá trị các số cần lưu mà chúng ta có thể tính được số tầng của cây Trie . Vì trong bài toán trên tập hợp có lưu cả số nguyên âm lẫn nguyên dương nên chúng ta cần cộng vào mỗi số lên một lượng 10^9 để biến tất cả các số thành số nguyên dương tiện lợi cho việc lưu vào cây Trie và khi in ra kết quả của các truy vấn chỉ cần trừ đi 10^9 để quay lại số chính xác ban đầu. Do vậy các số lưu trong cây Trie của chúng ta sẽ bé hơn hoặc bằng $2*10^9$ nên ta có thể biểu diễn các số này dưới dạng một xâu nhị phân có độ dài 31(ngoài ra ta có thể sử dụng phương pháp nén số). Các xâu nhị phân ở đây sẽ được viết từ bit cao nhất tới bit thấp nhất. Ví dụ số 1 sẽ được biểu diễn bởi xâu “0000....01”, hay số 3 được biểu diễn bởi xâu “0000.....011” . Như vậy các cạnh của cây Trie lúc này sẽ mang kí tự “0” hoặc “1” .

Tại mỗi nút trên cây Trie chúng ta sẽ lưu 1 biến *cnt* thể hiện số các số trong tập hợp hiện tại đi qua nút đó trên cây Trie. Như vậy khi thêm một số *x* vào tập hợp , ta sẽ tăng *cnt* của tất cả các nút trên đường đi của xâu nhị phân biểu diễn cho số *x* trên cây Trie . Khi xoá một số *x* ta sẽ làm ngược lại , giảm *cnt* của tất cả các nút trên đường đi của xâu nhị phân biểu diễn cho số *x* trên cây.

Nhận xét : khi so sánh một số *a* với một số *b* bất kì dưới dạng thập phân cũng giống hệt với việc so sánh các xâu nhị phân biểu diễn cho *a* với xâu biểu diễn cho *b*. Ví dụ *a*<*b* thì xâu nhị phân biểu diễn cho *a* cũng bé hơn xâu biểu diễn cho *b*. Từ đây ta sẽ rút ra một nhận xét khác trên cây Trie : khi ta đang đứng tại một nút bất kì trên cây , mọi số đi qua nhánh con mang kí tự 0 của nó luôn nhỏ hơn các số đi qua nhánh con mang kí tự 1. Từ nhận xét này ta sẽ rút ra cách làm cho 2 truy vấn như sau :

- Với truy vấn tìm số bé thứ k : ta sẽ xuất phát từ nút gốc của cây , gọi t là số các số đang có trong tập hợp đi qua nhánh con mang kí tự “0” . Nếu $t \geq k$ chứng tỏ số bé thứ k sẽ phải đi qua nhánh con mang kí tự “0” và ta sẽ nhảy xuống con qua nhánh này . Ngược lại ta sẽ nhảy xuống qua nhánh con mang kí tự “1” và tìm số lớn thứ $k-t$ trong tập hợp này. Khi tới nút lá (nút không có các nút con) , ta sẽ lấy kết quả chính bằng số của xâu nhị phân được biểu diễn bởi nút này.
- Với truy vấn đếm số lượng số bé hơn x : ta sẽ xuất phát từ nút gốc của cây và đi theo đường đi của xâu nhị phân biểu diễn cho số x này. Khi chúng ta nhảy xuống nhánh mang kí tự “1” cũng có nghĩa các xâu đi qua nhánh mang kí tự “0” nhỏ hơn x . Do đó ta sẽ cộng vào kết quả số lượng các số đi qua nhánh mang kí tự “0” .

Độ phức tạp của thuật toán này là $O(n^*31)$

4.2.1.3 Cách cài đặt chi tiết

Trước hết sẽ khai báo kiểu dữ liệu cho các nút trên cây Trie :

Struct Node

{

 Int cnt ; // lưu số lượng số hiện tại trong tập hợp đi qua nút này

 Node *child[2] ;

 Node()

{

 Memset(child, NULL , sizeof child) ;

 cnt=0 ;

```
    }  
}
```

Trước khi đi vào các hàm xử lí sau , cần nhắc lại các phép xử lí bitmask

```
#define BIT(x,i) ( ((x)>>(i))&1 ) // lấy ra bit thứ i của x  
  
#define MASK(i) (1<<(i)) // một xâu nhị phân chứa đúng 1 bit 1 ở vị trí thứ i , đây  
cũng chính bằng  $2^i$ 
```

Hàm chèn một số vào tập hợp :

```
void add(int x)  
{  
    Node *p=root ;  
    for(int i=31; i>=0 ;--i)  
    {  
        int k=BIT(x,i) ;  
        if(p->child[k]==NULL) p->child[k]=new Node() ;  
        p=p->child[k] ;  
        p->cnt++ ; // cập nhập số lượng số trong tập hợp đi qua nút p  
        // tương tự với hàm xoá số x ta sẽ thay đổi ở đây thành p->cnt – để giảm  
đi số lượng số trong tập hợp đi qua nút p  
    }  
}
```

```
}
```

Chú ý khi add thêm một số cần kiểm tra trước xem số đó đã có trong tập hợp hay chưa

Hàm tìm số bé thứ k :

```
int Find(int k)

{
    Node *p=root;

    Int res =0 ; // biến lưu kết quả

    For(int i=31;i>=0 ;--i)

    {
        if(p->child[0]!=NULL && p->child[0]>=k)

            p=p->child[0] ;

        else

        {
            if(p->child[0]!=NULL) k-=p->child[0]->cnt ;

            p=p->child[1] ;

            res+=MASK(i) ; // do ta đi theo nhánh 1 nên kết quả sẽ được
            tăng thêm một lượng MASK(i) hay bit thứ i của kết quả phải bằng 1
        }
    }
}
```

```
    return res ;  
}  
}
```

Hàm đếm số lượng số bé hơn x :

```
int get(int x)  
{  
    Node *p=root ;  
    Int res=0 ; // biến lưu kết quả  
    For(int i=31;i>=0 ; i--)  
    {  
        Int k= BIT(x,i) ;  
        If(p==NULL) return res;  
        if(k==1 && p->child[0]!=NULL) res+=p->child[0]->cnt ;  
        p=p->child[k] ;  
    }  
    return res ;  
}
```

Code mẫu : <https://ideone.com/57HAd8>

Bộ test :

<https://www.dropbox.com/sh/022xrj3fphhu825/AACDnfx7uBPXOR0hg89KjG9pa?dl=0>

4.2.1.4 Nhận xét chung về bài toán

Đây là một bài toán cơ bản về việc quản lý tập hợp số với cây Trie . Cách quản lý tập hợp số này được áp dụng cho rất nhiều các bài tập khác . Qua đó cho ta thấy không chỉ để xử lí xâu , Trie còn rất hữu ích trong việc quản lí các tập hợp số giống như các cấu trúc dữ liệu khác như BIT, IT..

4.2.2 Bài XOR

Đây là một bài cơ bản về sự kết hợp giữa Trie với các tính chất đặc biệt của phép XOR

4.2.2.1 Đề bài

Cho một dãy số nguyên không âm n phần tử a_1, a_2, \dots, a_n . Gọi giá trị hoà hợp của cặp số (a_i, a_j) được tính bằng $a_i \text{ XOR } a_j$.

Yêu cầu : Hãy tìm giá trị hoà hợp lớn nhất trong tất cả các cặp .

Input :

- Dòng đầu tiên chứa số nguyên dương T ($T \leq 10$) là số bộ dữ liệu
- Tiếp theo là T dòng ứng với mỗi bộ dữ liệu ,với số nguyên dương đầu tiên là n ($n \leq 10^5$) tiếp theo là n số nguyên dương a_1, a_2, \dots, a_n ($0 \leq a_i \leq 10^9$)

Output :

- Gồm T dòng , mỗi dòng ghi câu trả lời cho một bộ dữ liệu tương ứng .

XOR.inp	XOR.out
2	3
3 1 2 3	6
3 2 4 6	

4.2.2.2 Ý tưởng thuật toán

Ta tiếp tục sử dụng ý tưởng lưu các số vào cây Trie thông qua các xâu nhị phân biểu diễn như bài ORDSET. Ta duyệt tất cả số $a[i]$ và tìm một số x trong tập n số sao cho $a[i] \text{ xor } x$ lớn nhất. Sau khi đã lưu các số vào Trie ta sẽ rút ra được một thuật toán tham lam như sau : xuất phát từ nút gốc , với bit thứ j của $a[i]$ là k và đang ở nút p trên cây. Ta sẽ xét xem nút p có nhánh con mang kí tự (k^1) hay không . Nếu có thì chúng ta sẽ đi vào nhánh này và kết quả sẽ có bit thứ j bằng 1, ngược lại ta sẽ đi vào nhánh k , và kết quả sẽ có bit thứ j bằng 0. Sở dĩ có được thuật toán tham lam như vậy nhờ các tính chất đặc biệt của phép xor ta có $k^{(k^1)} = 1$ cùng với $k^k=0$. (nếu $k=0$ thì $k^1=1$ và ngược lại $k=1$ thì $k^1=0$) . Do các số được lưu từ bit cao nhất xuống bit thấp nhất nên rõ ràng ta sẽ luôn ưu tiên đi vào nhánh giúp ta bắt được bit trong kết quả.

Độ phức tạp của thuật toán này là $O(n*31)$

4.2.2.3 Cách cài đặt chi tiết

Với hàm chèn một số vào cây Trie có thể tham khảo trong bài ORDSET.

Hàm tìm kết quả lớn nhất nếu xor số x với một số bất kì nào đang có trong cây Trie:

```
Int get(int x)
```

```
{
```

```
    Node *p=root;
```

```

Int res=0 ;

For(int i=31; i>=0 ; i--)

{

    Int k= BIT(x,i) ;

    If(p->child[k^1]!=NULL)

    {

        res+=MASK(i) ;

        p=p->child[k^1];

    }

    else p=p->child[k] ;

}

}

```

Ta sẽ duyệt mọi số $a[i]$ và lấy max các hàm $\text{get}(a[i])$

Code mẫu: <https://ideone.com/iSCy0U>

Bộ test: <https://www.dropbox.com/sh/dzpusobofitw67x/AAC117KWcj6j2-Hxsizih6uua?dl=0>

4.2.2.6 Nhận xét chung về bài toán

Tiếp tục sử dụng cách quản lý số như bài ORDSET nhưng ở bài này ta cần biết thêm các kỹ năng về xử lý bitmask mà cụ thể là các tính chất đặc biệt của phép xor. Việc kết hợp sử dụng Trie với các tính chất đặc biệt của phép xor này được áp dụng cho rất nhiều các bài tập khác. Các cấu trúc khác như IT hay BIT không thể thay thế Trie trong trường hợp này.

4.2.3 Bài tập vận dụng

<https://codeforces.com/problemset/problem/706/D>

<https://codeforces.com/problemset/problem/665/E>

<https://codeforces.com/problemset/problem/282/E>

<https://codeforces.com/problemset/problem/979/D>

<https://vn.spoj.com/problems/MEDIAN/>

<https://vn.spoj.com/problems/VOXOR/>

<https://codeforces.com/problemset/problem/948/D>

<https://codeforces.com/problemset/problem/1416/C>

4.3. Trie hỗ trợ cho các thuật toán quy hoạch động

4.3.1 Bài Chia Dãy

4.3.1.1 Đề bài

Đạt rất yêu thích môn Tin học, ngay từ những buổi học đầu tiên Đạt đã bộc lộ rõ niềm đam mê của mình. Hôm nay, thầy giáo dạy về phép xor bit. Đạt biết được rằng, giá trị của phép

xor được định nghĩa như sau:

X	Y	$X \wedge Y$
0	0	0
0	1	1
1	0	1

1		1		0	
---	--	---	--	---	--

Với 2 số nguyên 32-bit A và B , giá trị tổng xor của 2 số là số nguyên $C = A \wedge B$ với các giá trị bit $C_i = A_i \wedge B_i$ (Ký hiệu P_i là giá trị bit thứ i của số nguyên P ; $0 \leq i \leq 31$).

Cuối buổi, thầy giáo giao cả lớp một dãy số D_1, D_2, \dots, D_n và yêu cầu Đạt tính toán tổng

xor của một số đoạn các số. Đạt tính nhầm vô cùng nhanh và chính xác. Thầy vậy, thầy giáo giao thêm một bài toán * riêng cho Đạt. Thầy bổ sung thêm k cặp số nguyên không âm L_i và R_i . Thầy yêu cầu Đạt cần tách dãy số ra thành k đoạn liên tiếp mà tổng xor của đoạn thứ i không nhỏ hơn L_i và không lớn hơn R_i . Đạt vất và tìm được một vài cách và đang thắc mắc có bao nhiêu cách chia các đoạn như vậy.

Yêu cầu: cho dãy số D_1, D_2, \dots, D_n và k cặp L_i, R_i . Hãy giúp Đạt xác định số cách chia dãy số thành k đoạn, mỗi đoạn có tổng xor có giá trị không nhỏ hơn L_i và không lớn hơn R_i .

Input : vào từ file CHIADAY.INP

- Dòng đầu tiên chứa 2 số nguyên dương n, k .
- Dòng thứ 2 chứa n số nguyên 32 bit, không âm D_1, D_2, \dots, D_n
- k dòng tiếp theo, dòng thứ i chứa 2 số nguyên L_i và R_i ($0 \leq L_i \leq R_i \leq 10^9$).

Output : Ghi ra file CHIADAY.OUT một số nguyên duy nhất là số cách chia dãy tìm được. Kết quả đưa ra theo module trong phép chia cho 1000000007 .

CHIADAY.INP	CHIADAY.OUT
4 2 1 1 1	2

0 0	
1 1	

Subtask :

- 20% test tương ứng 20% số điểm có $n \leq 100, k \leq 4$
- 40% test khác tương ứng 40% số điểm có $n \leq 500, k \leq 100$
- 40% test còn lại tương ứng 40% số điểm có $n \times k \leq 10^5, k \leq n$

4.3.1.2 Ý tưởng thuật toán

Thuật toán quy hoạch động : gọi $F(i,j)$ là số cách chia dãy thành j phần gồm i phần tử đầu tiên . Ta sẽ có công thức như sau :

$F(i,j) = \text{tổng } F(x-1,j-1) \text{ với } x \text{ trong khoảng từ } 1 \rightarrow i \text{ sao cho } D_x \text{ xor } D_{x+1} \text{ xor } \dots \text{ xor } D_i \text{ nằm trong đoạn } [L_j, R_j] .$

Nhờ các tính chất đặc biệt của phép xor ta có thể viết gọn lại tổng xor $D_x \text{ xor } D_{x+1} \text{ xor } \dots \text{ xor } D_i$ thành $\text{sum}[i] \text{ xor } \text{sum}[x-1]$ với $\text{sum}[i]$ là tổng xor của đoạn từ $1 \rightarrow i$ hay $\text{sum}[i] = D_1 \text{ xor } D_2 \text{ xor } \dots \text{ xor } D_i$.

Kết quả của bài toán trên sẽ là $\mathbf{F[n][k]}$.

Với ý tưởng thuật toán quy hoạch động trên ta có thể thực hiện duyệt mọi trạng thái quy hoạch động $F(i,j)$ và duyệt mọi x trong khoảng từ $1 \rightarrow i$ sau đó kiểm tra xem $\text{sum}[i] \text{ xor } \text{sum}[x-1]$ có nằm trong đoạn $[L_j, R_j]$ hay không, nếu có thì ta sẽ cộng $F(x-1,j-1)$ vào $F(i,j)$. Như vậy thuật toán sẽ có độ phức tạp $O(n*n*k)$.

Để cải thiện thuật toán trên ta nhận thấy số trạng thái quy hoạch động là $n*k$ và không thể giảm bớt được . Phép chuyển trạng thái quy hoạch động của thuật toán

trên đang có độ phức tạp là $O(n)$ cho mỗi trạng thái nên ta dễ dàng nhận ra nên rút gọn thời gian cho phép chuyển trạng thái quy hoạch động. Ta sẽ giảm độ phức tạp cho các phép chuyển trạng thái này xuống độ phức tạp $O(\log_2 a_i)$ cho tất cả các trạng thái quy hoạch động với cấu trúc dữ liệu để hỗ trợ ở đây là cây Trie.

Trước hết ta sẽ dùng k cây Trie , cây Trie thứ j sẽ biểu diễn cho các trạng thái quy hoạch động $F(i,j)$ với mọi i trong khoảng $0->n$. Ta sẽ lưu tất cả các số $sum[i]$ với mọi i trong khoảng $0->n$ vào cây Trie. Các lưu số tương tự giống với các bài ORDSET hay XOR ở phía trên . Mỗi nút trên cây Trie thứ j sẽ có 1 giá trị *value* mang ý nghĩa là tổng của các $F(i,j)$ sao cho $sum[i]$ đi qua nút này trên đường đi của cây Trie . Khi duyệt đến trạng thái quy hoạch động $F(i,j)$ ta có thể tính nhanh được tổng của tất cả $F(x,j-1)$ với $sum[x] xor sum[i] \leq C$ với mọi số nguyên dương C bằng cây Trie thứ $j-1$. Như vậy dễ dàng có thể tính được giá trị của $F(i,j)$ bằng cách gọi hàm tính tổng của tất cả $F(x,j-1)$ với $sum[x] xor sum[i] \leq R_j$ trừ đi hàm tính tổng của tất cả $F(x,j-1)$ với $sum[x] xor sum[i] \leq L_{j-1}$, sau đó ta sẽ cập nhật $F(i,j)$ vào cây Trie thứ j theo đường đi của $sum[i]$ trên cây Trie . Để tính được tổng của tất cả $F(x,j-1)$ với $sum[x] xor sum[i] \leq C$ ta sẽ thực hiện bằng cây Trie như sau : đi theo đường đi của $sum[i]$ trên cây Trie thứ $j-1$, xuất phát từ nút gốc , khi đứng tại một nút p trên cây , gọi k là bit tiếp theo của $sum[i]$. Nếu bit tiếp theo của C là bit 0 thì ta sẽ phải bắt buộc đi theo nhánh k do nếu ta đi theo nhánh k^1 thì $sum[i] xor$ với các số trong nhánh này đều lớn hơn C . Nếu bit tiếp theo của C là bit 1 thì ta sẽ cộng giá trị $p->child[k]->value$, do lúc này $sum[i] xor$ với các số trong nhánh k đều bé hơn C ; sau đó ta sẽ đi theo nhánh k^1 . Tiếp tục làm như vậy cho tới khi ta nhảy tới một nút rỗng. Để hiểu rõ hơn hãy xem các đoạn mã giả trong phần cài đặt chi tiết.

Độ phức tạp của thuật toán này là $O(n*k*31)$

4.3.1.3 Cách cài đặt chi tiết

Trước hết ta sẽ viết 1 struct cho các k cây Trie

Struct Trie

{

// struct dưới đây sẽ biểu diễn cho các nút trên cây Trie

Struct Node

{

int value ; // tổng giá trị các $F(i,j)$ với $sum[i]$ đi qua nút này

Node * child[2] ;

Node()

{

value=0 ;

memset(child,0, sizeof child) ;

}

}

Node *root = new Node() ;

// hàm tính tổng giá trị $F(x,j-1)$ sao cho $sum[x] \text{ xor } sum[i]$ bé hơn C

Int get(int i,int C)

{

Int res=0 ;

For(int j=31;j>=0;--j)

```

    {

        Int k= BIT(sum[i],j) ;

        If(p==NULL) return res ; // nếu nút p hiện tại rỗng ta sẽ return luôn
        kết quả

        If(BIT(C,j) ==0)

            p=p->child[k] ; // nếu bit thứ j của C bằng 0 thì ta phải
            nhảy theo nhánh k

        else

            {

                res+=p->child[k]->value ;

                p=p->child[k^1] ; // nếu bit thứ j của C bằng 1 thì ta sẽ
                nhảy theo nhánh k^1 và cộng vào kết quả tổng value của nhánh k

            }

        }

// hàm cập nhật giá trị F(i,j) vào cây Trie thứ j

Void update(int i,int val) // với val=F(i,j)

{

    For(int j=31; j>=0 ;j--)

    {

```

```

        Int k= BIT(sum[i],j) ;

        If(p ->child[k]==NULL) p->child[k]=new Node() ;

        p=p->child[k] ;

        p->value+=val ; // cập nhật giá trị value của nút này

    }

};

Trie T[100005] ;

```

Dưới hàm chính ta sẽ thực hiện duyệt các trạng thái quy hoạch động và tính cho các trạng thái quy hoạch động này

```

F[0][0]=1 ; // ban đầu tất cả các F[i][j] mang giá trị 0 chỉ có F[0][0] mang giá trị 1
T[0].update(0,F[0][0]) ; // cập nhật giá trị F[0][0] vào cây Trie thứ 0

```

```
For(int i=1;i<=n ; ++i)
```

```
{
```

```
    For(int j=1;j<=k;++j)
```

```
{
```

```
        F[i][j] = T[j-1].get(i,R) - T[j-1].get(i,L-1) ;
```

```
        // tính giá trị F[i][j]
```

```
}
```

```
    For(int j=1;j<=k ;++j)
```

```
{
```

```

        T[j].update(i,F[i][j]) ;

    // cập nhật giá trị F[i][j] vào cây Trie thứ j

    }

}

```

Sau khi tính xong ta sẽ in ra $F[n][k]$, dĩ nhiên ta sẽ phải thực hiện các phép toán mod cho các giá trị $F[i][j]$.

Code mẫu: <https://ideone.com/tn9Xtu>

Bộ test :

<https://www.dropbox.com/sh/q6ygxrkfb9t1bwn/AACXskNl3Sy0DFr3ZDe5RB6Oa?dl=0>

4.3.1.4 Nhận xét về bài toán

Trong bài toán này ta đã áp dụng cách kết hợp Trie quản lý tập hợp số với các tính chất đặc biệt của phép xor giống như bài XOR để hỗ trợ cho thuật toán quy hoạch động. Các cấu trúc khác như IT hay BIT không thể kết hợp cùng các tính chất đặc biệt của phép xor nên lúc này chỉ có Trie giúp tăng tốc thời gian chạy và giảm đi rất nhiều độ phức tạp của thuật toán.

4.3.2 Bài SEPARATE

4.3.2.1 Đề bài

Có một xâu ký tự cần phải chia ra thành các đoạn con (gồm các ký tự liên tiếp của xâu) sao cho mỗi đoạn con này là một từ trong tập danh sách các từ cho trước.

Yêu cầu: Viết chương trình đếm số cách chia khác nhau có thể có. Hai cách chia được gọi là khác nhau nếu có ít nhất một vị trí cắt khác nhau. Vì con số này có thể rất lớn nên bạn chỉ cần in phần dư của nó khi chia cho 1337377.

Input:

- Dòng 1: Chứa xâu ký tự S cần chia có độ dài không quá $3 \cdot 10^5$ ký tự.
- Dòng 2: Chứa số nguyên dương n ($1 \leq n \leq 4000$) - số từ có trong danh sách
- Dòng $3 \dots n + 2$: Mỗi dòng chứa một từ trong danh sách. Tất cả các từ có độ dài không vượt quá 100 và chỉ chứa các chữ cái tiếng Anh in thường.

Output: Một số nguyên duy nhất - kết quả tìm được.

SEPARATE.INP	SEPARATE.OUT
Abcd 4 a b cd ab	2

4.3.2.2 Ý tưởng thuật toán

Thuật toán quy hoạch động : gọi $F[i]$ là số cách chia đoạn con từ 1 đến i của xâu ký tự S cần chia . Ta dễ dàng có được công thức $F[i] = \text{tổng}(F[j])$ nếu đoạn con $j+1$ đến i của xâu S là một xâu nằm trong tập hợp cho trước. Do các từ trong tập hợp cho trước có độ dài không quá 100 nên ta chỉ cần xét tối đa 100 vị trí j . Như vậy vấn đề còn lại của chúng ta là tìm cách xác định nhanh xem đoạn con từ $j+1$ đến i

của xâu S có phải là một từ trong tập hợp hay không . Để kiểm tra xem một xâu con từ l đến r của S có nằm trong tập hợp cho trước hay không ta có 2 cách kiểm tra :

Cách 1: ta sẽ sử dụng Hash để mã hoá các xâu trong tập hợp cũng như xâu S , sau đó nén số hoặc dùng map đánh dấu mã của các từ xuất hiện trong tập hợp. Để kiểm tra xâu con từ l đến r của S có nằm trong tập hợp hay không ta chỉ việc kiểm tra xem mã này có xuất hiện trong tập hợp hay không . Tuy bước kiểm tra này có thể được thực hiện trong độ phức tạp $O(1)$ nhưng thời gian chạy khá lâu do việc dùng hash phải thực hiện cái phép toán module .

Cách 2: ta sử dụng một cây Trie lưu các xâu đảo ngược của các xâu nằm trong tập hợp. Khi duyệt ta duyệt ngược vị trí j từ i về đầu , tại mỗi bước xâu con của S ta cần xét sẽ được thêm 1 kí tự vào đầu , nếu ta lật ngược lại xâu này thì sẽ là thêm 1 kí tự vào cuối. Việc thêm 1 kí tự vào cuối này giống với một phép nhảy từ nút cha tới nút con trực tiếp trên cây Trie. Như vậy khi duyệt vị trí j ta sẽ thực hiện các phép nhảy trên cây Trie. Tại mỗi nút trên cây Trie ta sẽ lưu thêm 1 biến có giá trị =0/1 để kiểm tra xem có xâu nào kết thúc tại nút này hay không. Nếu có xâu kết thúc tại nút biểu diễn cho xâu con từ $j+1$ đến i của S trên cây Trie thì ta sẽ cộng thêm $F[j]$ vào $F[i]$.

4.3.2.3 Cách cài đặt chi tiết

Phần cài đặt cây Trie ở đây hoàn toàn tương tự các bài ở trên , ta chỉ việc lưu thêm 1 biến bool $last$; để kiểm tra xem có xâu nào kết thúc tại đây hay không. Sau đó ta sẽ thực hiện lưu các xâu đảo ngược của các xâu trong tập hợp vào cây Trie. Đoạn mã giả quy hoạch động sẽ như sau :

```
S=' '+S; // đánh chỉ số các vị trí trong xâu S từ 1 để thuận tiện cho việc quy hoạch động
```

```
F[0]=1;
```

```

For(int i=1;i<(int)S.size() ; ++i)

{

    Node *p=root ; // xuất phát từ nút gốc của cây Trie

    For(int j=i;j>=1;j--)

    {

        int k=s[j]-‘a’ ;

        p=p->child[k] ;



        if(p==NULL) break; // nếu là nút rỗng thì ta sẽ dừng lại vì không có

xâu nào đi qua nút này cả

        if(p->last==true) dp[i]+=dp[j-1] ; // nếu có xâu kết thúc tại nút p ta sẽ

cộng thêm 1 lượng dp[j-1] do xâu con từ j đến i là một từ trong tập hợp cho trước

    }

}

}

```

Code mẫu : <https://ideone.com/JMzK3V>

Bộ test : https://www.dropbox.com/sh/y7i3uqnhsg5ri6i/AABozBuST-PEhxYrPhl_i5Qaa?dl=0

4.3.2.4 Nhận xét chung về bài toán

Cây Trie trong bài toán này tiếp tục giúp ta có thể tăng tốc thời gian chạy và giảm bớt đi rất nhiều độ phức tạp của thuật toán quy hoạch động. Đây là một bài cơ bản về việc dùng Trie quản lý xâu để hỗ trợ tăng tốc các thuật toán quy hoạch động. Tư tưởng dùng Trie hỗ trợ các bài quy hoạch động với xâu có trong rất nhiều các bài tập khác phải kể đến như bài 4 đề thi HSG QG môn Tin học năm 2021...

4.3.3 Bài SEED

4.3.3.1 Đề bài

Một SEED là một xâu chỉ gồm 2 loại ký tự “1” hoặc “*” thỏa điều kiện bắt đầu và kết thúc của SEED là “1”. Một SEED s được gọi là “hit” được xâu nhị phân x độ dài N nếu tồn tại một vị trí i trên xâu x thỏa mãn: Nếu ký tự thứ k của xâu s bằng “1” thì ký tự thứ $i+k-1$ của x cũng bằng “1”.

Ví dụ: $1*1$ có thể “hit” được các xâu **0101100**, **1110000**, **1010111** nhưng không “hit” được xâu **0100010**.

Yêu cầu: Cho N và một SEED S , đếm số lượng xâu nhị phân độ dài N mà s “hit” được.

Input :

- Dòng 1: chứa số N ($N \leq 50$)
- Dòng 2: ghi SEED S là một xâu chỉ gồm 2 loại ký tự “1” và “*”

Output :

- Gồm một dòng, chứa một số là số lượng xâu nhị phân mà s có thể “hit” được.

Subtask :

- Subtask 1 : $N \leq 20$
- Subtask 2 : Độ dài của xâu $S \leq 30$ và có số kí tự “*” không nhiều hơn kí tự loại “1”

SEED.inp	SEED.out
3 1*1	2

4.3.3.2 Ý tưởng thuật toán

Với subtask 1 ta dễ dàng sinh ra toàn bộ $2^{20}-1$ khả năng có thể của xâu N sau đó kiểm tra xem có đoạn con nào của xâu đó mà S “hit” được hay không.

Với subtask 2 ta nhận thấy có không quá 2^{15} xâu mà xâu S có thể “hit” được ta sẽ đưa tất cả 2^{15} vào trong một cây Trie. Gọi $F(i,j)$ là số xâu có i kí tự và đoạn hậu tố của xâu đang xây dựng này đang đứng tại nút j trên cây Trie và những xâu này chưa bị xâu S “hit”. Do cả 2^{15} xâu trong cây Trie có chiều dài bằng nhau nên ta chỉ cần lưu nút j xâu nhất trên cây Trie, nếu j là kết thúc của một xâu trên cây Trie hay nó có độ sâu là độ dài của xâu S thì đó là một xâu bị xâu S “hit” và ta sẽ loại bỏ các vị trí j này. Như vậy biến j sẽ giúp ta kiểm soát được xâu ta đang xây dựng có bị xâu S “hit” hay không. Sau khi tính được mảng F , kết quả của bài toán chính là $2^N - \text{tổng } F(n,j)$ với mọi j không phải là nút lá trên cây Trie. Do tổng $F(N,j)$ với mọi j không phải là nút lá trên cây Trie chính là số lượng xâu độ dài n không bị xâu S “hit”, còn 2^N chính là số lượng xâu nhị phân độ dài N . Như vậy công thức chuyển trạng thái quy hoạch động sẽ như sau $F(i+1,x) = \text{tổng } F(i,j)$ sao cho từ nút j khi thêm kí tự ‘0’ hoặc ‘1’ vào thì từ nút j trên cây Trie ta sẽ phải nhảy sang nút x , thỏa mãn x và j đều không phải nút lá trên cây Trie. Từ trạng thái (i,j) ta xét khi thêm vào xâu đang xây dựng các kí tự ‘0’ hoặc ‘1’. Nếu tồn tại nhánh con mang kí tự đang xét trên cây Trie ta sẽ nhảy xuống nhánh con này. Nếu không tồn tại ta sẽ lấy xâu được biểu diễn bởi nút j trên cây Trie và thêm kí tự đang xét vào cuối. Duyệt mọi hậu tố của xâu này và chọn ra hậu tố dài nhất có thể có nút biểu diễn trên xâu S . Các bước này ta có thể chuẩn bị trước tạo mảng $\text{nxt}(j,c)$ là nút trên cây Trie khi thêm kí tự c vào xâu được biểu diễn bởi nút j trên cây Trie, $\text{nxt}(j,c)=-1$ nếu nút tiếp theo là một nút lá trên cây Trie. Như vậy chúng ta sẽ thu được thuật toán với độ phức tạp $O(N * \text{số nút trên cây Trie} \cdot 2^{15})$ xâu nhị phân có độ dài không quá 30).

4.3.3.3 Cách cài đặt chi tiết

Trước hết ta sẽ tạo ra tất cả các xâu mà xâu S có thẻ “hit” được và chèn nó vào cây Trie, trên mỗi nút của cây Trie sẽ có 1 biến *last=0/1* để kiểm tra xem có xâu nào kết thúc tại nút này trên cây Trie hay không. Tại mỗi nút không phải nút lá ta sẽ lưu thêm một biến *lab* để đánh chỉ số trên cây Trie tiện cho việc đưa vào mảng quy hoạch động.

Ta sẽ *dfs* trên cây Trie để thực hiện việc tính mảng *nxt* như sau :

```
Void dfs(Node *p, string s) // sử dụng string s để lưu lại các xâu trên đường đi từ gốc đến nút này

{
    For(int c=0;c<=1;++c)

    {
        If(p->child[c]!=NULL)

        {
            If(p->child[c]->last!=false)

            {
                Dfs(p->child[c]->last,s+char(c+'0')) ;

                Nxt[p->lab][c]=p->child[c]->lab ;

            }

            Else Nxt[p->lab][c]=-1 ; // trường hợp nút tiếp theo là nút lá trên cây Trie
        }
    }
}
```

```

Else

{

String t= s+char(c+'0') ;

For(int i=0;i<t.size();++i)

{

// xét hậu tố bắt đầu từ vị trí i của xâu t

Node *q=root ; // xuất phát từ nút gốc cây Trie

bool ok=false ; // biến xem kiểm tra đã tìm được hậu tố

nào thoả mãn hay chưa

For(int j=0;j<t.size();++j)

{

Int k=t[j]-'0' ;

If(q->child[k]==NULL) break ;

q=q->child[k] ;

if(q->last==true) break ; // q đang là một nút lá

trên cây

if(j==t.size()-1) ok=true ; // đã tìm được hậu tố

dài nhất xuất hiện trên cây Trie và nút q không phải nút lá trên cây Trie

}

If(ok==true) nxt[p->lab][c]=q->lab,break ;

}

```

```
    }  
  
}  
  
}
```

Sau khi tính xong mảng nxt bằng cách gọi $dfs(root, "")$ ta dễ dàng thực hiện thuật toán quy hoạch động xây dựng xâu như sau :

```
F[0][0]=1 ; // nút gốc sẽ có lab=0  
  
For(int i=1;i<=N; ++i)  
  
{  
  
    For(int j=0;j<=numNode;++) // numNode là số nút không phải nút là trên  
    cây Trie  
  
        For(int c=0;c<=1;++c)  
  
        {  
  
            Int x= nxt[j][c] ;  
  
            If(x!= -1) F[i+1][x]+=F[i][j] ;  
  
        }  
  
}
```

Sau đó ta sẽ in ra tổng của các $F[n][j]$ với mọi j chạy từ 0 đến $numNode$

Code mẫu : <https://ideone.com/h7yv2C>

Bộ test :

<https://www.dropbox.com/sh/v9mr0d9y1enx8lx/AAAZrAtOY5XXzkjrwCaAFugna?dl=0>

4.3.3.4 Nhận xét chung về bài toán

Khác với 2 bài toán trước sử dụng Trie để quản lí và lấy các kết quả cho các trạng thái quy hoạch động , bài SEED này lại sử dụng các nút trên cây Trie để xác định các trạng thái quy hoạch động, cây Trie lúc này không chỉ là một cấu trúc dữ liệu để hỗ trợ thuật toán quy hoạch động nữa. Đây là một bài toán khá khó , lạ và độc đáo. Ý tưởng này được áp dụng cho các bài toán quy hoạch động xây dựng xâu tránh các mâu xâu cho trước.

4.3.4 Bài tập vận dụng

<https://vnoi.info/problems/NKSEV/>

<https://oj.vnoi.info/problem/chain2>

<https://oj.vnoi.info/problem/bubba2>

https://oj.vnoi.info/problem/voi21_bonus

<https://codeforces.com/problemset/problem/856/B>

KẾT LUẬN

Sau khi áp dụng chuyên đề cây tiền tố Trie vào thực tế dạy học sinh lớp chuyên Tin, đặc biệt là các em học sinh trong đội tuyển dự thi chọn học sinh giỏi Tỉnh và Quốc gia, tôi thấy nó mang lại hiệu quả rất rõ rệt. Nó thay đổi cách tiếp cận, phương pháp làm các dạng bài toán liên quan đến xâu có kích thước lớn, chúng ta đã có thêm một phương pháp giúp cải tiến thuật toán một cách tối ưu nhất để thực hiện. Hầu hết giáo viên và học sinh đều đánh giá đây là một phương pháp hay, cần

phổ biến rộng và củng cố, luyện tập để nâng cao được chất lượng giảng dạy, học tập và thi cử của học sinh trong các kì thi có các bài toán liên quan.

Với chuyên đề này, tôi đã cố gắng phân chia các bài toán về thành lớp các bài toán cùng dạng để áp dụng Trie, mỗi dạng tôi đều phân tích các bài toán cụ thể và đưa ra các bài toán tương tự. Hi vọng học sinh và giáo viên các trường bạn sẽ có cách nhìn tổng thể về cấu trúc dữ liệu Trie, và việc áp dụng các bài toán sẽ đạt hiệu quả cao nhất.

Với thời gian nghiên cứu có hạn, chuyên đề này chắc chắn không tránh khỏi những khiếm khuyết. Tôi xin chân thành cảm ơn những nhân xét, đánh giá và góp ý của các đồng nghiệp để tôi bổ sung và hoàn thiện cho đề tài.

Tôi xin trân trọng cảm ơn !

TÀI LIỆU THAM KHẢO

<https://vnoi.info/wiki/algo/data-structures/trie.md>

<https://vi.wikipedia.org/wiki/Trie>

<https://www.geeksforgeeks.org/trie-insert-and-search/>

<https://leduythuccs.github.io/2018-05-23-c-u-tr-c-d-li-u-trie/>

[Các bài tập ôn luyện của thầy Đỗ Đức Đông](#)