

Mục lục Athena IX

| | |
|---|----|
| SỐ HỌC | 7 |
| Tính nhanh lũy thừa | 7 |
| Bài toán | 7 |
| Giải thuật | 7 |
| Chương trình | 8 |
| Nhận xét | 8 |
| Tính nhanh số Fibonacci | 10 |
| Bài toán | 10 |
| Giải thuật | 10 |
| Chương trình | 11 |
| Hàm Euler | 12 |
| Định nghĩa | 12 |
| Tính chất | 12 |
| Giải thuật | 12 |
| Ứng dụng | 13 |
| Tìm GCD và LCM | 14 |
| Tìm ước số chung lớn nhất | 14 |
| Tìm bội số chung nhỏ nhất | 14 |
| Sàng Eratosthenes lọc số nguyên tố | 15 |
| Lọc dãy số | 15 |
| Phương pháp lọc theo khối | 15 |
| Phản tử nghịch đảo theo mô đun | 17 |
| Định nghĩa | 17 |
| Tìm nghịch đảo theo giải thuật Euclid mở rộng | 17 |
| Giải thuật Euclid mở rộng | 17 |
| Tìm nghịch đảo theo mô đun | 18 |
| Tìm nghịch đảo bằng phương pháp tính nhanh lũy thừa | 19 |
| Tìm các nghịch đảo theo mô đun nguyên tố | 19 |
| Lô ga rít rời rạc | 21 |
| Bài toán | 21 |

| | |
|--|----|
| Giải thuật | 21 |
| Chương trình | 23 |
| Phương trình tuyến tính Diophantine hai ẩn số | 24 |
| Bài toán | 24 |
| Tìm một nghiệm | 24 |
| Giải thuật | 24 |
| Chương trình | 25 |
| Công thức tính tất cả các nghiệm | 25 |
| Xác định số lượng nghiệm và dẫn xuất tất cả nghiệm thuộc đoạn cho trước | 26 |
| Giải thuật | 26 |
| Hàm get_segment | 26 |
| Tìm nghiệm có tổng nhỏ nhất trên đoạn cho trước | 27 |
| Giải thuật | 27 |
| Chương trình: | 28 |
| Phương trình mô đun tuyến tính bậc nhất | 30 |
| Bài toán | 30 |
| Giải thuật dựa trên cơ sở tìm phần tử nghịch đảo | 30 |
| Chương trình | 31 |
| Dẫn xuất về phương trình Diophantine | 31 |
| Tìm bậc của ước trong giải thừa | 32 |
| Bài toán | 32 |
| Trường hợp k - nguyên tố | 32 |
| Tính phần trong giải thừa rút gọn | 35 |
| Đặt vấn đề | 35 |
| Giải thuật | 35 |
| Hàm tính giải thừa rút gọn theo mô đun p | 36 |
| Căn gốc theo mô đun | 37 |
| Định nghĩa | 37 |
| Tồn tại | 37 |
| Mối quan hệ với hàm Euler | 37 |
| Giải thuật tìm căn gốc | 37 |

| | |
|--|----|
| Đánh giá độ phức tạp | 38 |
| Chương trình: | 38 |
| Tìm căn giá trị nguyên | 40 |
| Bài toán | 40 |
| Giải thuật tìm một nghiệm | 40 |
| Tìm tất cả các nghiệm khi biết một nghiệm | 40 |
| Chương trình | 41 |
| Xử lý số lớn | 43 |
| Biểu diễn số | 43 |
| Đổi cơ số | 44 |
| Tổng hai số | 44 |
| Tính hiệu | 44 |
| Tính tích | 45 |
| Tính số dư | 45 |
| Chương trình minh họa | 46 |
| Tính giá trị biểu thức theo mô đun | 48 |
| Giải thuật Leman | 50 |
| Phân số liên tục (Continued fractions) | 50 |
| Giải thuật Leman phân tích số nguyên ra thừa số | 52 |
| THỦY TRIỀU ĐỎ Tên chương trình: HAB.CPP | 55 |
| HÌNH HỌC TÍNH TOÁN | 59 |
| Hợp nhất các đoạn thẳng | 59 |
| Bài toán | 59 |
| Giải thuật | 59 |
| Chương trình | 60 |
| Diện tích đa giác cạnh không tự cắt | 61 |
| Bài toán | 61 |
| Giải thuật | 61 |
| Chương trình | 62 |
| Định lý Pick | 63 |
| Định lý | 63 |
| Chứng minh | 63 |

| | |
|---|-----|
| Tập điểm phủ các đoạn thẳng | 65 |
| Bài toán 1 | 65 |
| Giải thuật | 65 |
| Chương trình | 66 |
| Bài toán 2 | 67 |
| Giải thuật | 67 |
| Chương trình | 73 |
| Bao lồi | 75 |
| Bài toán | 75 |
| Giải thuật Graham | 75 |
| Chương trình | 77 |
| Kiểm tra điểm trong | 78 |
| Bài toán | 78 |
| Giải thuật | 78 |
| Chương trình | 81 |
| Tìm cặp điểm gần nhất | 82 |
| Bài toán | 82 |
| Giải thuật | 82 |
| Chương trình | 84 |
| Đường thẳng quét | 85 |
| Bài toán | 85 |
| Giải thuật | 86 |
| Chương trình: | 91 |
| XỬ LÝ XÂU | 94 |
| HÀM Z | 94 |
| Ví dụ ứng dụng | 99 |
| VR16. THUẦN CHỦNG <i>Tên chương trình: PURE.???</i> | 99 |
| Chương trình | 101 |
| Kỹ thuật hàm băm | 106 |
| Ví dụ ứng dụng | 108 |
| MẬT KHẨU <i>Tên chương trình: PAROLE.???</i> | 108 |

| | | |
|---|--------------------------------------|-----|
| <i>Giải thuật</i> | 109 | |
| VS05. CON ĐƯỜNG GÓM SỨ | <i>Tên chương trình: CERAMIC.CPP</i> | 112 |
| Cây tiền tố | 115 | |
| MẢNG HẬU TỐ | 126 | |
| Xây dựng mảng hậu tố | 128 | |
| Ứng dụng: | 135 | |
| Tìm xâu con dài nhất xuất hiện 2 lần và không giao nhau trong xâu cho trước | 135 | |
| Tìm xâu đầy vòng nhỏ nhất | 136 | |
| Tìm xâu con | 138 | |
| Thực hiện các truy vấn so sánh hai xâu con | 141 | |
| Tìm tiền tố chung dài nhất của 2 xâu con (Phương pháp sử dụng bộ nhớ trung gian) | 144 | |
| Tìm tiền tố chung dài nhất của 2 xâu con (Phương pháp không sử dụng bộ nhớ trung gian), Tìm tiền tố chung dài nhất của 2 hậu tố liên tiếp, | 147 | |
| Số lượng xâu con khác nhau | 148 | |
| Tìm LCP của các hậu tố một xâu. Giải thuật Kasai. | 149 | |
| Ký hiệu | 151 | |
| Tính chất của LCP | 151 | |
| Giải thuật | 153 | |
| Ô TÔ MÁT HẬU TỐ | 155 | |
| Định nghĩa | 155 | |
| Ví dụ về ô tô mát hậu tố | 156 | |
| Giải thuật xây dựng ô tô mát hậu tố với chi phí thời gian tuyến tính | 156 | |
| Một số tính chất của ô tô mát hậu tố | 164 | |
| Ứng dụng ô tô mát hậu tố | 168 | |
| Cấu trúc ROPE | 175 | |
| Mô tả cấu trúc | 175 | |
| Cộng xâu – merge | 175 | |
| Xác định ký tự theo chỉ số - hàm get | 176 | |
| Tách xâu – hàm split | 176 | |
| Các phép xóa, bổ sung – delete và insert | 177 | |

| | |
|--|-----|
| Cân bằng cây..... | 178 |
| Các giải pháp nâng cao hiệu quả | 178 |
| Quy hoạch động..... | 179 |
| <i>Nguyên lý Bellman</i> | 179 |
| Bài 1a. VỀ ĐÍCH <i>Tên chương trình: FINISH.CPP</i> | 179 |
| Các bước xử lý..... | 182 |
| Bài 1b,..... | 186 |
| Các bài tập ứng dụng giải thuật quy hoạch động..... | 188 |
| Bài 2. NÓI ĐIỂM <i>Tên chương trình: JOIN.CPP</i> | 188 |
| Bài 3. DÃY KÝ TỰ CON CHUNG DÀI NHẤT <i>Tên chương trình: LCS.CPP</i> | 191 |
| Quy hoạch động đơn giản..... | 194 |
| SỐ CATALAN <i>Tên chương trình: CATALAN.CPP</i> | 195 |
| LÁT ĐƯỜNG VIỀN <i>Tên chương trình: PAVE.CPP</i> | 198 |
| HAI SỐ 1 <i>Tên chương trình: TWOONE.CPP</i> | 201 |
| MÁY ATM <i>Tên chương trình: ATM.CPP</i> | 204 |

Tính nhanh lũy thừa

Bài toán

Cho 3 số nguyên dương a , n và p . Hãy tính $a^n \bmod p$.

Dữ liệu: Vào từ file văn bản FASTPOWER.INP gồm một dòng chứa 3 số nguyên dương a , n và p ($1 \leq a \leq 10^9$, $1 \leq n \leq 10^{18}$, $1 \leq p \leq 10^9$).

Kết quả: Đưa ra file văn bản FASTPOWER.OUT một số nguyên – kết quả tìm được.

Ví dụ:

| FASTPOWER.INP |
|---------------|
| 2 9 10 |

| FASTPOWER.OUT |
|---------------|
| 2 |

Giải thuật

Việc yêu cầu đưa ra theo mô đun p là để đảm bảo các kết quả trung gian và cuối cùng là đủ nhỏ, *không cần xử lý số lớn*,

Bản chất của vấn đề là phải tính nhanh x^y , trong đó x, y là các số nguyên dương.

Giả thiết y có dạng biểu diễn nhị phân là

$$y = (b_m, b_{m-1}, \dots, b_1, b_0),$$

trong đó b_i là 0 hoặc 1, $i = 0 \div m-1$, $b_m = 1$.

Ví dụ:

$$y = 9_{10} = 1001_2 = 2^3 + 2^0$$

```

graph TD
    y[9<sub>10</sub>] --> 1001[1001<sub>2</sub>]
    1001 --> b3[b<sub>3</sub>]
    1001 --> b2[b<sub>2</sub>]
    1001 --> b1[b<sub>1</sub>]
    1001 --> b0[b<sub>0</sub>]
    b3 --> 23["2<sup>3</sup>"]
    b0 --> 20["2<sup>0</sup>"]
  
```

Ta có

$$x^y = \prod_{i=0}^m x^{b_i 2^i} = \prod_{i=0, b_i=1}^m x^{2^i} \quad (*)$$

Như vậy ta phải tính $x^1, x^2, x^4, x^8, \dots, x^{2^i}$

Để tính x^{2^i} ta chỉ cần thực hiện i phép nhân:

$$x \rightarrow x \times x = x^2 \rightarrow x^2 \times x^2 = x^4 \rightarrow x^4 \times x^4 = x^8 \rightarrow \dots$$

Như vậy để tích lũy tích ($*$) ta cần thực hiện không quá $2 \times m$ phép nhân, tức là với chi phí thời gian $O(\log_2 y)$.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "fastpower."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t a,n,p,ans=1;

int main()
{
    fi>>a>>n>>p;
    a%=p;
    while (n)
    {
        if (n&1) ans=(ans*a)%p;
        a=(a*a)%p;
        n>>=1;
    }
    fo<<ans;
}
```

Nhận xét

- Có thể lập luận theo sơ đồ xử lý đệ quy, chặt chẽ hơn về mặt lý thuyết nhưng chỉ phù hợp với kỹ thuật lập trình đệ quy, chương trình chẳng những không ngắn gọn mà còn tốn bộ nhớ và thời gian thực hiện hơn sơ đồ lặp đã nêu ở trên: xét việc tính $r = x^y$, (x, y – nguyên dương), nếu y chẵn ta có $r = x^{y/2} \times x^{y/2}$, còn với y lẻ ta có:

$$r = x^{(y-1)/2} \times x^{(y-1)/2} \times x,$$

tức là với một hoặc 2 phép nhân ta đã giảm độ lớn của số mũ xuống một nửa. Như vậy với không quá $2 \times \log_2 y$ ta có số mũ của lũy thừa là 0.

```
int64_t FastPow(int64_t a, int64_t n)
{
    if (n==0) return 1;
    if (n&1) return FastPow(a, n-1) * a;
    else
    {
        int64_t b = FastPow(a, n/2);
        return b*b;
    }
}
```

- Giải thuật tính nhanh lũy thừa được phát triển từ *Sơ đồ Nhân Ai Cập* kinh điển: Tính $a \times b$ và đưa kết quả theo mô đun p ($1 \leq a, b \leq 10^{18}$, $1 \leq p \leq 10^9$).

- Trong các tài liệu, sơ đồ nhân Ai Cập còn được gọi với nhiều tên khác.
- Việc chứng minh tính đúng đắn và hiệu quả của Sơ đồ nhân Ai Cập cũng tương tự như đối với việc tính nhanh lũy thừa.
- Nói chung trong phần lớn các trường hợp sơ đồ đệ quy sẽ cho chương trình ngắn gọn hơn sơ đồ lặp, nhưng trong thực hiện – tốn bộ nhớ hơn và thời gian tính cũng lớn hơn,

Chương trình Nhân Ai Cập:

```
#include <bits/stdc++.h>
#define NAME "egypmul."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t a,b,p,ans=0;

int main()
{
    fi>>a>>b>>p;
    a%=p;
    while(b)
    {
        if(b&1) ans=(ans+a)%p;
        a=(a<<1)%p;
        b>>=1;
    }
    fo<<ans;
}
```



Tính nhanh số Fibonacci

Bài toán

Dãy số $\mathbf{F}_0 = \mathbf{F}_1 = 1$, $\mathbf{F}_i = \mathbf{F}_{i-1} + \mathbf{F}_{i-2}$ với $i > 1$ được gọi là dãy số Fibonacci. Cho 2 số nguyên dương n và p . Hãy tính và đưa ra \mathbf{F}_n theo mô đun p .

Dữ liệu: Vào từ file văn bản FIB.INP gồm một dòng chứa 2 số nguyên n và p ($1 \leq n \leq 10^{18}$, $0 < p \leq 10^9$).

Kết quả: Đưa ra file văn bản FIB.OUT một số nguyên số \mathbf{F}_n theo mô đun p .

Ví dụ:

| FIB.INP |
|---------|
| 11 100 |

| FIB.OUT |
|---------|
| 44 |

Giải thuật

Dãy số Fibonacci là dãy số tăng rất nhanh. Do dãy số này có rất nhiều ứng dụng cả trong lý thuyết lẫn thực tế nên nó đã được khảo sát rất kỹ. Có nhiều công thức tính số Fibonacci đã được xác định.

Với tin học công thức tính phù hợp nhất là công thức ma trận, đảm bảo độ phức tạp O(logn) và không phải giải quyết vấn đề tích lũy sai số làm tròn.

Xét 2 ma trận vuông kích thước 2×2 :

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, \quad B = \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}, \text{ ta có } A \times B = C, \text{ trong đó}$$

$$C = \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} \times b_{00} + a_{01} \times b_{10} & a_{00} \times b_{01} + a_{01} \times b_{11} \\ a_{10} \times b_{00} + a_{11} \times b_{10} & a_{10} \times b_{01} + a_{11} \times b_{11} \end{pmatrix}$$

Để thuận tiện tính toán ta tuyến tính hóa ma trận 2 chiều thành một chiều:

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} = \begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \end{pmatrix},$$

$$B = \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \end{pmatrix},$$

$$C = \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} z_0 & z_1 \\ z_2 & z_3 \end{pmatrix}, \text{ trong đó}$$

$$z_0 = x_0 \times y_0 + x_1 \times y_2,$$

$$z_1 = x_0 \times y_1 + x_1 \times y_3,$$

$$z_2 = x_2 \times y_0 + x_3 \times y_2,$$

$$z_3 = x_2 \times y_1 + x_3 \times y_3.$$

Công thức ma trận tính số Fibonacci:

$$(F_{n-2} \ F_{n-1}) \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_n \ F_{n+1})$$

Từ đây ta có

$$(F_0 \ F_1) \times P^n = (F_n \ F_{n+1}),$$

Trong đó $P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$.

Bằng kỹ thuật tính nhanh lũy thừa có thể nhận được P^n với độ phức tạp $O(\log n)$.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "fib."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
uint64_t n,a,ans,fib,t[4]={0,1,1,1},r[4]={0,1,1,1},tg[4],p;

void mp2(uint64_t x[],uint64_t y[],uint64_t z[])
{
    tg[0]=(x[0]*y[0]+x[1]*y[2])%p;
    tg[1]=(x[0]*y[1]+x[1]*y[3])%p;
    tg[2]=(x[2]*y[0]+x[3]*y[2])%p;
    tg[3]=(x[2]*y[1]+x[3]*y[3])%p;
    z[0]=tg[0];z[1]=tg[1];z[2]=tg[2];z[3]=tg[3];
}
int main()
{
    fi>>n>>p;
    while(n)
    {
        if(n&1) mp2(r,t,r);
        mp2(t,t,t);
        n>=1;
    }
    ans=r[2]%p;
    fo<<ans;

    fo<<"\nTime: "<<clock()/(double)1000<<" sec";
}
```



Hàm Euler

Định nghĩa

Hàm Euler $\Phi(n)$ (hay còn ký hiệu là $\phi(n)$) là số lượng các số trong phạm vi từ 1 đến n nguyên tố cùng nhau với n . Nói một cách khác, đó là số lượng các số trong đoạn $[1, n]$ có ước số chung lớn nhất với n là 1.

Các giá trị đầu tiên của hàm này:

$$\Phi(1) = 1,$$

$$\Phi(2) = 1,$$

$$\Phi(3) = 2,$$

$$\Phi(4) = 2,$$

$$\Phi(5) = 4.$$

Tính chất

Ba tính chất đơn giản sau đây cho phép tính hàm với n bất kỳ:

$$\boxed{\text{Nếu } p \text{ là số nguyên tố thì } \Phi(p) = p-1}$$

Đó là điều hiển nhiên vì mọi số nguyên tố đều là nguyên tố cùng nhau với các số nhỏ hơn nó.

$$\boxed{\text{Nếu } p \text{ là số nguyên tố và } a \text{ là số tự nhiên thì } \Phi(p^a) = p^a - p^{a-1}}$$

Thật vậy, p^a chỉ không nguyên tố cùng nhau với các số dạng $p \times k$, trong đó k là số nguyên và trong đoạn đang xét có tất cả $p^a/p = p^{a-1}$ số có thể đóng vai trò của k .

$$\boxed{\text{Nếu } a \text{ và } b \text{ là nguyên tố cùng nhau thì } \Phi(a \times b) = \Phi(a) \times \Phi(b)}$$

Xét z – số nguyên bất kỳ thỏa mãn điều kiện $z \leq a \times b$. Gọi x và y là số dư của phép chia z tương ứng cho a và cho b . z nguyên tố cùng nhau với $a \times b$ khi và chỉ khi z nguyên tố cùng nhau với a và nguyên tố cùng nhau với b , điều này dẫn đến việc x nguyên tố cùng nhau với a và y nguyên tố cùng nhau với b . Theo [định lý Số dư Trung Hoa](#), cặp số (x, y) như vậy ($x \leq a$, $y \leq b$) sẽ đơn trị xác định $z \leq a \times b$. Đó là điều phải chứng minh.

Giải thuật

Hàm Euler được tính dựa trên việc phân tích số n ra thừa số nguyên tố.

Nếu có

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdot \dots \cdot p_k^{a_k}$$

trong đó \mathbf{p}_i là các số nguyên tố, $i = 1 \div k$, thì

$$\begin{aligned}\phi(n) &= \phi(p_1^{a_1}) \cdot \phi(p_2^{a_2}) \cdot \dots \cdot \phi(p_k^{a_k}) = \\ &= (p_1^{a_1} - p_1^{a_1-1}) \cdot (p_2^{a_2} - p_2^{a_2-1}) \cdot \dots \cdot (p_k^{a_k} - p_k^{a_k-1}) = \\ &= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdot \dots \cdot \left(1 - \frac{1}{p_k}\right).\end{aligned}$$

Hiệu quả của giải thuật phụ thuộc vào phương pháp phân tích n ra thừa số nguyên tố. Các phương pháp phân tích n ra thừa số nguyên tố với độ phức tạp thấp sẽ được xét sau. Dưới đây là phương pháp tính $\Phi(n)$ đơn giản nhất với độ phức tạp $O(\sqrt{n})$.

```
int phi (int n) {
    int result = n;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    if (n > 1)
        result -= result / n;
    return result;
}
```

Ứng dụng

- Tính chất quan trọng và thường được sử dụng là *Định lý Euler*:

$$a^{\phi(m)} \equiv 1 \pmod{m},$$

Trong đó a và m – nguyên tố cùng nhau.

- Trong trường hợp riêng, khi m là số nguyên tố, ta có *Định lý nhỏ Fermat*:

$$a^{m-1} \equiv 1 \pmod{m}$$

- Hàm Euler được ứng dụng trong việc giải nhiều bài toán thực tế cũng như các bài toán Olympic.



Tìm GCD và LCM

Tìm ước số chung lớn nhất

Ước số chung lớn nhất (*Greatest Common Divisor – GCD*) của 2 số nguyên không âm **a** và **b** là số nguyên lớn nhất mà **a** và **b** cùng chia hết cho số đó.

Giải thuật Euclid cho phép tìm GCD với số lượng phép chia phải thực hiện là $O(\log(\min\{a, b\}))$.

Giải thuật theo sơ đồ lặp:

```
int gcd (int a, int b) {
    while (b) {
        a %= b;
        swap (a, b);
    }
    return a;
}
```

Giải thuật cũng có thể triển khai theo sơ đồ đệ quy:

```
int gcd (int a, int b) {
    return b ? gcd (b, a % b) : a;
}
```

Do tính phổ biến của nhu cầu tìm GCD trong hệ thống lập trình C++ cung cấp **hàm __gcd(a, b)** trong thư viện của hệ thống.

Tìm bội số chung nhỏ nhất

Bội số chung nhỏ nhất (*Least Common Multiplier – LCM*) là số nguyên nhỏ nhất cùng chia hết cho **a** và **b**.

Công thức tính LCM:

$$\text{lcm}(a, b) = \frac{a \cdot b}{\text{gcd}(a, b)}$$

Hàm xác định LCM:

```
int lcm(int a, int b)
{
    return a / __gcd(a, b) * b;
}
```



Sàng Eratosthenes lọc số nguyên tố

Sàng Eratosthenes là giải thuật tìm các số nguyên tố trong khoảng $[1, n]$ với độ phức tạp $O(n \log \log n)$.

Lọc dãy số

Tư tưởng của giải thuật đơn giản:

B1 – Đặt $p = 2$,

B2 – Gạch tất cả các số chia hết cho p ngoại trừ bản thân số p ,

B3 – Nếu không có số mới bị gạch – kết thúc xử lý, trong trường hợp ngược lại – gán cho p số nhỏ nhất chưa bị gạch và lớn hơn p ở bước B2, chuyển sang B2.

```
int n;
vector<char> prime(n+1, true);
prime[0] = prime[1] = false;
for (int i=2; i<=n; ++i)
    if (prime[i])
        if (i * i * i <= n)
            for (int j=i*i; j<=n; j+=i)
                prime[j] = false;
```

Nhược điểm chính của giải thuật trên là tốn bộ nhớ. Có thể sử dụng bít để đánh dấu. Bộ nhớ dành cho dữ liệu giảm 8 lần, nhưng giá phải trả là chi phí thời gian đánh dấu.

Phương pháp lọc theo khối

Bằng phương pháp lọc đơn giản đã nêu tìm và đánh dấu các số nguyên tố trong phạm vi từ 1 đến \sqrt{n} , lưu kết quả vào bảng số nguyên tố cơ sở.

Toàn bộ dữ liệu được lọc theo khối, ở mỗi thời điểm chỉ giữ và xử lý một khối.

Gọi s là kích thước khối. Ta có số lượng khối là $(n+s-1)/s$. Khối thứ k chứa các số trong đoạn $[k \times s, k \times s + s - 1]$, $k = 0, 1, 2, \dots, n/s$. Với mỗi khối dùng bảng nguyên tố cơ sở để lọc tương tự như trong giải thuật đơn giản đã nêu.

Cần đặc biệt lưu ý việc xử lý khối đầu tiên và khối cuối cùng:

- ⊕ Khối đầu tiên:
 - ▲ Các số 0 và 1 không được đánh dấu là nguyên tố,
 - ▲ Không được loại bỏ các số nguyên tố trong bảng nguyên tố cơ sở,
- ⊕ Khối cuối cùng: Có thể không chứa đủ s số.

Lựa chọn s :

- ⊕ Nếu s quá nhỏ hiệu quả giải thuật sẽ thấp vì có nhiều khối phải xử lý và với mỗi khối – phải một lần làm việc với bảng nguyên tố cơ sở,
- ⊕ Nếu s quá lớn – tốn bộ nhớ,

- ➡ Các kết quả thực nghiệm cho thấy nên lựa chọn **s** trong khoảng $10^4 - 10^5$.

```

const int SQRT_MAXN = 100000;
const int S = 10000;
bool nprime[SQRT_MAXN], bl[S];
int primes[SQRT_MAXN], cnt;

int main() {
    int n;
    cin >> n;
    int nsqrt = (int) sqrt (n + .0);
    for (int i=2; i<=nsqrt; ++i)
        if (!nprime[i]) {
            primes[cnt++] = i;
            if (i * 111 * i <= nsqrt)
                for (int j=i*i; j<=nsqrt; j+=i)
                    nprime[j] = true;
        }

    int result = 0;
    for (int k=0, maxk=n/S; k<=maxk; ++k) {
        memset (bl, 0, sizeof bl);
        int start = k * S;
        for (int i=0; i<cnt; ++i) {
            int start_idx = (start + primes[i] - 1) / primes[i];
            int j = max(start_idx, 2) * primes[i] - start;
            for (; j<S; j+=primes[i])
                bl[j] = true;
        }
        if (k == 0)
            bl[0] = bl[1] = true;
        for (int i=0; i<S && start+i<=n; ++i)
            if (!bl[i])
                ++result;
    }
    cout << result;
}

```

Tìm bảng nguyên tố cơ sở

Địa chỉ đầu số bội của số nguyên tố cơ sở thứ j

Thống kê số lượng



Phần tử nghịch đảo theo mô đun

Định nghĩa

Với số nguyên dương m cho trước, tập các số nguyên $0, 1, 2, \dots, m-1$ được gọi là vành tạo bởi mô đun m .

Số nguyên b được gọi là nghịch đảo của số nguyên a theo mô đun m nếu

$$a \times b \equiv 1 \pmod{m}$$

b còn thường được ký hiệu là a^{-1} .

Dễ dàng thấy rằng số 0 không có nghịch đảo. Với m cho trước mỗi số nguyên khác 0 có thể có hoặc không có nghịch đảo. Nghịch đảo chỉ tồn tại với các số nguyên a *nguyên tố cùng nhau* với m .

Tìm nghịch đảo theo giải thuật Euclid mở rộng

Giải thuật Euclid mở rộng

Ngoài việc tìm ước số chung lớn nhất (gcd) của 2 số a và b , nhiều khi ta còn phải biết các hệ số thể hiện gcd qua a và b , tức là phải tìm x và y thỏa mãn

$$a \times x + b \times y = \text{gcd}(a, b)$$

Việc tính các hệ số này không phức tạp, chỉ cần dẫn xuất công thức biến đổi từ cặp (a, b) sang $(b \% a, a)$.

Giả thiết ta đã biết nghiệm (x_1, y_1) đối với cặp $(b \% a, a)$:

$$(b \% a) \times x_1 + a \times y_1 = g$$

Và muốn tìm cặp (x, y) đối với (a, b) .

Ta có $b \% a = b - (a/b) \times a$.

Thế vào phương trình trên, có:

$$\begin{aligned} g &= b \% a \times x_1 + a \times y_1 \\ &= (b - a/b) \times x_1 + a \times y_1 \\ &= b \times x_1 + a \times (y_1 - (a/b) \times x_1) \end{aligned}$$

Từ đây suy ra:

$$x = y_1 - (a/b) \times x_1,$$

$$y = x_1.$$

Chương trình hiện thực hóa giải thuật Euclid mở rộng:

```
#include <bits/stdc++.h>
#define NAME "gcd_ex."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t a,b,x,y,gcd;
int64_t gcd_ex(int64_t a, int64_t b, int64_t &x,int64_t &y)
{
    if(a==0) {x=0; y=1; return b;}
    int64_t x1, y1;
    int64_t d = gcd_ex(b%a,a,x1,y1);
    x=y1-(b/a)*x1;
    y=x1;
    return d;
}

int main()
{
    fi>>a>>b;
    gcd=gcd_ex(a,b,x,y);
    fo<<gcd<<' '<<x<<' '<<y;
}
```

Tìm nghịch đảo theo mô đun

Xét phương trình

$$ax + my = 1$$

Đây là phương trình Diophantine tuyến tính bậc 2. Phương trình này có nghiệm khi và chỉ khi $\text{gcd}(a, m) = 1$.

Lấy mô đun m cả 2 vế phương trình trên, ta có:

$$ax \equiv 1 \pmod{m}$$

Bằng giải thuật Euclid mở rộng dễ dàng tìm được x :

```

#include <bits/stdc++.h>
#define NAME "inv_ex."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t a,m,x,y,g;
int64_t gcd_ex(int64_t a, int64_t b, int64_t &x,int64_t &y)
{
    if(a==0) {x=0; y=1; return b;}
    int64_t x1, y1;
    int64_t d = gcd_ex(b%a,a,x1,y1);
    x=y1-(b/a)*x1;
    y=x1;
    return d;
}

int main()
{
    fi>>a>>m;
    g=gcd_ex(a,m,x,y);
    if(g!=1) fo<<"No solution";
    else
    {
        x=(x%m+m)%m;
        fo<<x;
    }
}

```

Tìm nghịch đảo bằng phương pháp tính nhanh lũy thừa

Với a và m nguyên tố cùng nhau, theo định lý Euler ta có

$$a^{\phi(m)} \equiv 1 \pmod{m},$$

Nếu m là nguyên tố thì tình huống còn đơn giản hơn, theo định lý Ferma nhỏ ta có

$$a^{m-1} \equiv 1 \pmod{m}$$

Nhân cả 2 vế với a^{-1} ta có

$$a^{m-2} \equiv a^{-1} \pmod{m}$$

Trong trường hợp m là nguyên tố vấn đề cần giải quyết đơn thuần chỉ là tính nhanh lũy thừa với độ phức tạp $O(\log m)$.

Với m không phải là nguyên tố, cần xác định $\Phi(n)$ trước khi tính nhanh lũy thừa. Điều này đòi hỏi phải phân tích m ra thừa số nguyên tố - một việc không đơn giản.

Tìm các nghịch đảo theo mô đun nguyên tố

Bài toán: Cho số nguyên tố m . Hãy tìm nghịch đảo của mỗi số trong đoạn $[1..m-1]$.

Giải thuật:

Theo các giải thuật đã nêu, bài toán có thể được giải quyết với độ phức tạp $O(m \log m)$.

Có một phương pháp tiếp cận khác cho phép giải với độ phức tạp $O(m)$.

Ký hiệu $r[i]$ là nghịch đảo của i theo mô đun m . Ta có:

- $r[1] = 1$,
- $r[i] = -(m/i) \times r[m \% i] \% m$ với $i > 1$.

Chứng minh:

$$m \% i = m - (m/i) \times i$$

Lấy mô đun m cả 2 vế, ta có

$$m \% i = - (m/i) \times i \pmod{m}$$

Nhân cả 2 vế với nghịch đảo của i và của $(m \% i)$ ta có

$$r[i] = - (m/i) \times r[m \% i] \pmod{m}.$$

Đó là điều phải chứng minh.

Chương trình xử lý hết sức ngắn gọn.

Đoạn chương trình tính r có dạng:

```
r[1] = 1;
for (int i=2; i<m; ++i)
    r[i] = (m - (m/i) * r[m \% i] \% m) \% m;
```



Lô ga rit rời rạc

Bài toán

Cho 3 số nguyên dương a , b và m , trong đó a và m – nguyên tố cùng nhau. Hãy tìm số nguyên x thỏa mãn điều kiện

$$a^x \equiv b \pmod{m}.$$

Dữ liệu: Vào từ file văn bản M_I_M.INP gồm một dòng chứa 3 số nguyên dương a , b và m ($1 \leq a, b, m \leq 10^9$).

Kết quả: Đưa ra file văn bản M_I_M.OUT một số nguyên – kết quả tìm được. Nếu không tồn tại x thì đưa ra số -1.

Ví dụ:

| M_I_M.INP | M_I_M.OUT |
|-----------|-----------|
| 3 43 100 | 5 |

Giải thuật

Giải thuật cho bài toán trên có tên thường gọi là *Meet-in-the-Middle* (*Gặp nhau ở giữa*) do Shanks đề xuất năm 1971. Bản thân Shanks gọi nó là “*Giải thuật bước nhỏ – bước lớn*” (*Baby-step-giant-step Algorithm*).

Ta có phương trình cần giải

$$a^x \equiv b \pmod{m}$$

trong đó a và m nguyên tố cùng nhau.

Đặt $x = np-q$ với n là một hằng nguyên dương chọn trước.

Khi đó phương trình cần giải có dạng:

$$a^{np-q} \equiv b \pmod{m}$$

$$\begin{aligned} a^{np} &= b \times a^q \pmod{m} \\ f1(p) &= f2(q) \pmod{m} \end{aligned}$$

trong đó p và q là các số cần tìm. p được gọi là bước lớn vì khi p thay đổi một đơn vị x sẽ thay đổi n đơn vị, còn q – bước nhỏ.

Dễ dàng thấy rằng chỉ cần tìm x trong khoảng $[0, m]$.

Khi n được chọn, ta có $p \in [1, (m+n-1)/n]$, $q \in [0, n]$.

Bằng phương pháp nâng nhanh lũy thừa, các hàm **f1 (p)** và **f2 (q)** có thể tính với độ phức tạp O(log m).

Giải thuật bao gồm 2 bước:

Bước 1: Tính tất cả các giá trị của **f1 (p)** trong miền xác định và sắp xếp kết quả,

Bước 2: Với mỗi giá trị của **q** trong miền xác định tính giá trị **f2 (q)** và kiểm tra sự có mặt của nó trong tập giá trị đã tính ở bước 1.

Đặt $k = \lceil \frac{m}{n} \rceil = (m+n-1)/n$.

Độ phức tạp của bước 1 là O($k \log m$),

Độ phức tạp của bước 2 bao gồm tính giá trị hàm và tìm kiếm nhị phân: O($n \log m$).

Độ phức tạp của toàn giải thuật sẽ thấp nhất khi $n \approx m/n$, tức là $n \approx \sqrt{m}$.

Trên thực tế để tính **f1 (p)** ta chỉ cần dùng sơ đồ nâng nhanh lũy thừa một lần khi tính $a^n = a^n$, các giá trị còn lại nhận được bằng cách nhân tiếp với a^n . Do n không quá lớn, độ phức tạp của giải thuật sẽ giảm không đáng kể ngay cả khi không dùng sơ đồ tính nhanh lũy thừa.

Với **f2 (q)**, xuất phát từ **b**, lần lượt nhân với **a** ta sẽ được tất cả các giá trị cần kiểm tra.

Thay vì việc dùng mảng lưu giá trị ở bước 1 và sau đó sắp xếp ta có thể dùng cấu trúc **map** (*Cây Đỏ - Đen*) để lưu trữ. Điều này sẽ làm cho việc lập trình ở bước 2 trở nên đơn giản hơn.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "m_i_m."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t a,b,m,an,ta,ans;
int n,tn;
map<int,int> vals;

int main()
{
    fi>>a>>b>>m;
    n=(int)sqrt(m+.0);
    an=1;
    ta=a%m; tn=n;
    while(tn)
    {
        if(tn&1) an=(an*ta)%m;
        ta=(ta*ta)%m;
        tn>>=1;
    }
    for(int i=1, cur=an;i<=n;++i)
    {
        if(!vals.count(cur)) vals[cur]=i;
        cur=(cur*an)%m;
    }
    for(int i=0, cur=b;i<=n;++i)
    {
        if(vals.count(cur))
        {
            ans=vals[cur]*n-i;
            if(ans<m) {fo<<ans; return 0;}
        }
        cur=(cur*a)%m;
    }

    fo<<"-1";
}
```



Phương trình tuyến tính Diophantine hai ẩn số

Bài toán

Cho phương trình tuyến tính 2 ẩn số

$$ax + by = c,$$

trong đó a, b, c – các số nguyên.

Các yêu cầu thực hiện có thể là:

- ✚ Một cặp số nguyên (x, y) là nghiệm của phương trình trên,
- ✚ Tìm tất cả các nghiệm thuộc miền cho trước,
- ✚ Tính số lượng nghiệm thuộc miền cho trước,
- ✚ Tìm nghiệm có tổng các ẩn số là nhỏ nhất.

Trong trường hợp bài toán vô nghiệm – đưa ra số -1.

Dưới đây ta chỉ xét trường hợp phương trình không suy biến, tức là $a^2+b^2 \neq 0$.

Tìm một nghiệm

Giải thuật

Đầu tiên xét trường hợp a và b không âm.

Giải thuật Euclid mở rộng cho phép tính được g , x_g và y_g , trong đó g – ước số chung lớn nhất của a, b và $ax_g + by_g = g$.

Về trái của phương trình ban đầu chia hết cho g , vì vậy về phải, tức là c cũng phải chia hết cho g và việc dẫn xuất tiếp theo dưới đây cho thấy bài toán có nghiệm. Nếu c không chia hết cho g – bài toán vô nghiệm.

Nhân cả 2 vế của biểu thức đánh dấu ở trên với c/g , ta có:

$$ax_g \times (c/g) + by_g \times (c/g) = c$$

Nghiệm của phương trình sẽ là:

- ✚ $x_0 = x_g \times (c/g)$,
- ✚ $y_0 = y_g \times (c/g)$.

Trường hợp có hệ số âm, ta tìm x_0, y_0 với giá trị tuyệt đối của hệ số, sau đó đổi dấu nghiệm nếu hệ số tương ứng nhỏ hơn 0.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "d_eq_1."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t a,b,c,x,y,x0,y0,g,t;
int64_t gcd_ex(int64_t a, int64_t b, int64_t
&x, int64_t &y)
{
    if(a==0) {x=0; y=1; return b;}
    int64_t x1, y1;
    int64_t d = gcd_ex(b%a,a,x1,y1);
    x=y1-(b/a)*x1;
    y=x1;
    return d;
}

int main()
{
    fi>>a>>b>>c;
    g=gcd_ex(abs(a),abs(b),x,y);
    if(c%g) {fo<<"-1"; return 0;}
    t=c/g;
    x0=x*t; y0=y*t;
    if(a<0) x0=-x0; if(b<0) y0=-y0;
    fo<<x0<<' '<<y0;
}
```

Công thức tính tất cả các nghiệm

Nếu phương trình Diophantine có nghiệm thì nó sẽ có vô số nghiệm.

Ký hiệu $g = \gcd(a, b)$ và cặp số nguyên (x_0, y_0) thỏa mãn

$$ax_0 + by_0 = c,$$

ta sẽ dẫn xuất công thức tính tất cả các nghiệm của phương trình ban đầu.

Nếu ta thêm vào x_0 một lượng bằng b/g và bớt ở y_0 một lượng a/g thì đẳng thức trên vẫn đúng:

$$a(x_0+b/g)+b(y_0-a/g)=ax_0+by_0+a/b/g-a/b/g=ax_0+by_0=c$$

Quá trình thêm/bớt nói trên có thể thực hiện bao nhiêu lần cũng được, như vậy ta có công thức tổng quát dẫn xuất nghiệm:

$$\begin{cases} x = x_0 + kb/g, \\ y = y_0 - ka/g \end{cases} \quad k - số nguyên bất kỳ$$

Xác định số lượng nghiệm và dẫn xuất tất cả nghiệm thuộc đoạn cho trước

Cho hai đoạn $[\min_x, \max_x]$ và $[\min_y, \max_y]$. Cần xác định số lượng nghiệm nằm trong đoạn đã cho và dẫn xuất chính các nghiệm đó.

Giải thuật

Nếu một trong 2 hệ số **a** hoặc **b** bằng 0 phương trình đã cho sẽ có không quá một nghiệm và do đó dưới đây ta sẽ không xét trường hợp này.

Đầu tiên ta tìm **x** nhỏ nhất nằm trong khoảng đã cho, tức là $x \geq \min_x$. Gọi phần tử đó là **lx1**. **lx1** dễ dàng xác định được với chi phí O(1) theo công thức dẫn xuất nghiệm đã nêu và ta có **k11** – hệ số **k** tương ứng với **lx1**.

Tương tự như vậy có thể tìm được $x = rx1 \leq \max_x - x$ lớn nhất thuộc khoảng đã cho và **kr1** – hệ số **k** tương ứng với **rx1**.

Hoán đổi **k11** và **kr1** nếu $k11 > kr1$. Như vậy ta có đoạn $[k11, kr1]$ giá trị **k** đảm bảo **x** nằm trong khoảng đã cho.

Bây giờ ta bắt đầu xét đến ràng buộc đối với **y**. Trước hết tìm **y** nhỏ nhất lớn hơn hoặc bằng \min_y . Ký hiệu **k12** là **k** tương ứng với **y** tìm được. Tiếp sau – tìm **y** lớn nhất thỏa mãn điều kiện $y \leq \max_y$ và ký hiệu **kr2** là **k** tương ứng với **y** tìm được.

Hoán đổi **k12** và **kr2** nếu $k12 > kr2$. Như vậy ta có đoạn $[k12, kr2]$ giá trị **k** đảm bảo **y** nằm trong khoảng đã cho.

Giao của 2 đoạn $[k11, kr1]$ và $[k12, kr2]$ là tập các giá trị **k** cho nghiệm trong đoạn đã cho.

Nếu giao là rỗng – không có nghiệm nào thỏa mãn yêu cầu. Trường hợp giao khác rỗng – ta có đoạn chung là $[k1, kr]$. Số lượng nghiệm thỏa mãn sẽ là **kr - k1 + 1**.

Việc dẫn xuất nghiệm không phải là vấn đề lớn: lần lượt cho **k** nhận các giá trị trong đoạn $[k1, kr]$ và tính nghiệm theo công thức đã tìm được ở phần trước.

Giải thuật tìm giao của 2 đoạn thẳng:

```
k1 = max(k11, k12);  
kr = min(kr1, kr2);  
if(k1>kr) /* vô nghiệm */; else /* xử lý trường hợp có nghiệm */;
```

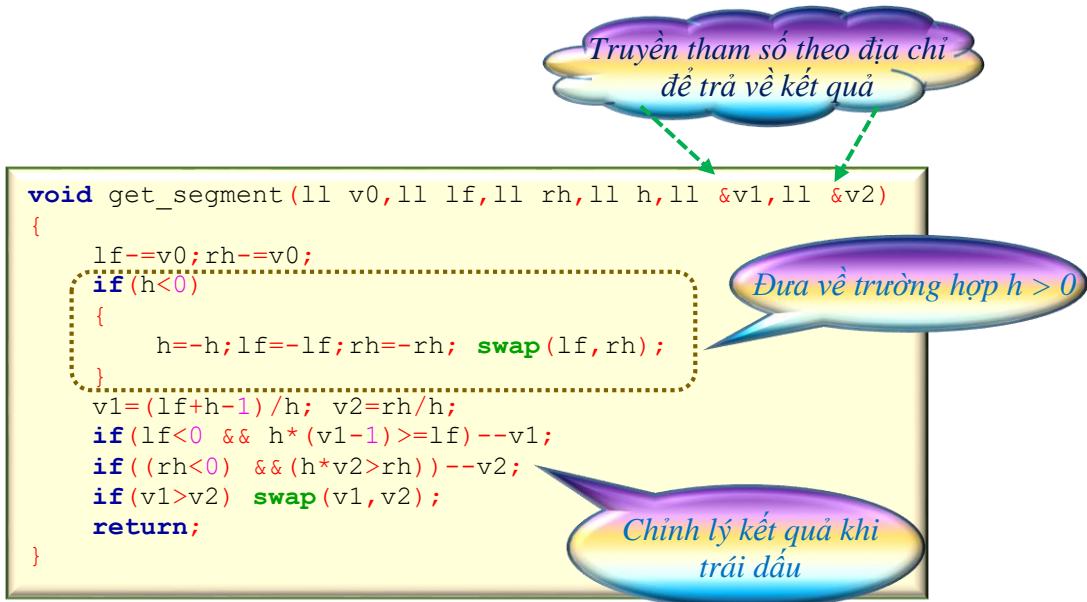
Hàm get_segment

Hàm **get_segment** tính đoạn $[v1, v2]$ chứa các giá trị **v** thỏa mãn điều kiện $lf \leq z \leq rh$, trong đó $z = v0 + v \times h$ với $h \neq 0$,

Hàm này phục vụ tính các đoạn $[k11, kr1]$ và $[k12, kr2]$ đã nêu ở trên.

Với $h < 0$: biến đổi dữ liệu vào, đưa về trường hợp $h > 0$.

Phép chia nguyên x/h được thực hiện theo trình tự chia các giá trị tuyệt đối, sau đó mới gán dấu tương ứng vào kết quả, vì vậy cần lưu ý kiểm tra và chỉnh lý kết quả khi $x < 0$.



Tìm nghiệm có tổng nhỏ nhất trên đoạn cho trước

Cho hai đoạn $[min_x, max_x]$ và $[min_y, max_y]$. Cần xác định nghiệm nằm trong đoạn đã cho có tổng $x+y$ là nhỏ nhất.

Giai thuật

Theo giải thuật ở phần trên ta có thể tìm đoạn $[kl, kr]$ để dẫn xuất các nghiệm nằm trong các đoạn đã cho.

Giả thiết (x, y) là một nghiệm của phương trình đang xét. Với một k cụ thể nào đó ta có nghiệm mới (xn, yn) :

- $xn = x + k \times (b/g)$,
- $yn = y - k \times (a/g)$.

Từ đây có:

$$xn + yn = x + y + k \times (b - a) / g$$

Như vậy, để $xn + yn$ là nhỏ nhất cần chọn k nhỏ nhất có thể khi $b \geq a$ và k lớn nhất khi $b < a$.

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "d_eq_2."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
typedef int64_t ll;
int64_t a,b,c,x,y,x0,y0,g,t;
int64_t minx,maxx,miny,maxy;
int64_t klx,krx,kly,kry,kl,kr;
int flag=0;

int64_t gcd_ex(ll a, ll b, ll &x, ll &y)
{
    if(a==0) {x=0; y=1; return b;}
    int64_t x1, y1;
    int64_t d = gcd_ex(b%a,a,x1,y1);
    x=y1-(b/a)*x1;
    y=x1;
    return d;
}

void get_segment(ll v0,ll lf,ll rh,ll h,ll &v1,ll &v2)
{
    lf-=v0; rh-=v0;
    if(h<0)
    {
        h=-h; lf=-lf; rh=-rh; swap(lf,rh);
    }
    v1=(lf+h-1)/h; v2=rh/h;
    if(lf<0 && h*(v1-1)>=lf)--v1;
    if((rh<0) && (h*v2>rh))--v2;
    if(v1>v2) swap(v1,v2);
    return;
}

void sol_segment()
{
    fi>>minx>>maxx>>miny>>maxy;
    get_segment(x0,minx,maxx,b/g,klx,krx);
    get_segment(y0,miny,maxy,-a/g,kly,kry);
    kl=max(klx,kly); kr=min(krx,kry);
    if(kl>kr) fo<<"\n-1\n";
    else
    {
        int64_t ta=b/g, tb=-a/g;
        fo<<'\n'<<kr-kl+1<<'\n';
        for(int i = kl; i<=kr; ++i)
            fo<<x0+i*ta<< ' '<<y0+i*tb<<'\n';
    }
}
```

```

void sol_min_sum()
{
    int64_t ta=b/g, tb=-a/g, k;
    k= (a<b) ? k1:kr;
    fo<<' \n'<<x0+k*ta<< ' <<y0+k*tb;
}

int main()
{
    fi>>a>>b>>c;
    fi>>minx>>maxx>>miny>>maxy;
    g=gcd_ex (abs (a) , abs (b) , x, y) ;
    if (c%g) {fo<<"-1\n"; return 0; }
    t=c/g; flag=1;
    x0=x*t; y0=y*t;
    if (a<0)x0=-x0; if (b<0)y0=-y0;
    fo<<x0<< ' '<<y0<< '\n';
    if (flag) sol_segment ();
    if (flag) sol_min_sum ();
}

```



Phương trình mô đun tuyến tính bậc nhất

Bài toán

Cho các số nguyên a, b, n ($n > 0$). Hãy tìm các số nguyên x trong đoạn $[0, n-1]$ thỏa mãn điều kiện $a \times x = b \pmod{n}$.

Dữ liệu: Vào từ file văn bản M_EQ.INP gồm một dòng chứa 3 số nguyên a, b và n ($0 \leq a, b \leq 10^9, 0 < n \leq 2 \times 10^9$).

Kết quả: Đưa ra file văn bản M_EQ.OUT:

- ⊕ Dòng đầu tiên chứa số nguyên k – số lượng số tìm được,
- ⊕ Nếu $k > 0$: Dòng thứ 2 chứa k số thỏa mãn điều kiện bài toán.

Ví dụ:

| M_EQ.INP |
|----------|
| 5 9 13 |

| M_EQ.OUT |
|----------|
| 1 |
| 7 |

Giải thuật dựa trên cơ sở tìm phần tử nghịch đảo

Đầu tiên xét trường hợp a và n *nguyên tố cùng nhau*. Khi đó có thể tìm *phần tử* a^{-1} nghịch đảo với a theo mô đun n . Nhân cả 2 vế phương trình với phần tử nghịch đảo này, ta nhận được nghiệm của phương trình và là *nghiệm duy nhất*:

$$x = b \times a^{-1} \pmod{n}$$

Xét trường hợp a và n *không nguyên tố cùng nhau*. Phương trình đã cho có thể vô nghiệm, ví dụ $2 \times x = 1 \pmod{4}$.

Ký hiệu $g = \gcd(a, n)$. Vé trái của phương trình luôn luôn chia hết cho g , vì vậy nếu b không chia hết cho g – bài toán vô nghiệm.

Chia a, b và n cho g , ta có phương trình

$$a' \times x = b' \pmod{n'}$$

a' và n' nguyên tố cùng nhau và theo cách đã xét, ta tìm được nghiệm x' duy nhất. Để dàng thấy rằng nó cũng là nghiệm của phương trình ban đầu, nhưng *không phải là nghiệm duy nhất*.

Trong đoạn cần xét, phương trình ban đầu có đúng g nghiệm và có dạng:

$$x_i = x' + i \times n', \quad i = 0, 1, \dots, g-1$$

Như vậy, trong mọi trường hợp, phương trình ban đầu hoặc vô nghiệm hặc có đúng $g = \gcd(a, n)$ nghiệm trong đoạn $[0, n-1]$.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "M_eq."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t a,b,n,g,x,y,k;

int64_t gcd_ex(int64_t a, int64_t b, int64_t
&x,int64_t &y)
{
    if(a==0) {x=0; y=1; return b;}
    int64_t x1, y1;
    int64_t d = gcd_ex(b%a,a,x1,y1);
    x=y1-(b/a)*x1;
    y=x1;
    return d;
}
int main()
{
    fi>>a>>b>>n;
    g=__gcd(a,n);
    if(b%g!=0) {fo<<'0'; return 0;}
    a/=g; b/=g; n/=g;
    k=gcd_ex(a,n,x,y);
    x=(x%n+n)%n; x=(x*b)%n;
    fo<<g<<'\n';
    for(int i = 0;i<g;++i) fo<<x+i*n<<'\n';
}
```

Dẫn xuất về phương trình Diophantine

Việc tìm nghiệm bài toán đã cho tương đương với việc tìm tất cả các nghiệm của phương trình Diophantine tuyến tính bậc 2

$$ax + ny = b$$

trong đoạn $[0, n-1]$, ở đây x và y là ẩn số.

Sơ đồ xử lý có rắc rối hơn đôi chút, nhưng bản chất giải thuật và độ phức tạp không thay đổi.



Tìm bậc của ước trong giai thừa

Bài toán

Cho các số nguyên dương n và k . Hãy tìm số nguyên x lớn nhất thỏa mãn điều kiện $n!$ chia hết cho k^x .

Dữ liệu: Vào từ file văn bản GRADE.INP gồm một dòng chứa 2 số nguyên n và k ($1 \leq n \leq 10^9$, $1 < k \leq 10^{15}$).

Kết quả: Đưa ra file văn bản GRADE.OUT số nguyên x tìm được.

Ví dụ:

| GRADE.INP |
|-----------|
| 6 3 |

| GRADE.OUT |
|-----------|
| 2 |

Trường hợp k – nguyên tố

Biểu thức tính giai thừa: $n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n$.

Trong tích trên các thừa số ở vị trí (bắt đầu từ 1) là bội của k chia hết cho k , mỗi thừa số đó sẽ làm tăng số mũ cần tìm lên 1 và có tất cả n/k thừa số như vậy.

Mỗi thừa số ở vị trí là bội của k^2 sẽ làm tăng số mũ cần tìm lên 1 (*vì đã tính một lần ở bước trước*). Số lượng thừa số loại này là n/k^2 .

Tương tự như vậy, mỗi thừa số ở vị trí là bội của k^i sẽ làm tăng số mũ cần tìm lên 1 (*vì đã tính i-1 một lần ở các bước trước*). Số lượng thừa số loại này là n/k^i .

Như số cần tìm là $n/k + n/k^2 + \dots + n/k^i + \dots$. Số lượng số hạng trong tổng là $O(\log_k n)$ và độ phức tạp của giải thuật cũng sẽ là như vậy.

Hàm tính x :

```
int fact_pow (int n, int k) {
    int res = 0;
    while (n) {
        n /= k;
        res += n;
    }
    return res;
}
```

Trường hợp k – hợp số

Phân tích k ra thừa số nguyên tố. Giả thiết k_i là thừa số thứ i với số mũ p_i . Áp dụng giải thuật đã xét với k_i ta có số mũ là ans_i với độ phức tạp tính toán $O(\log n)$.

Số mũ cần tìm sẽ là $\min\{\text{ans}_i/p_i\}$ với mọi i .

Tồn tại các phương pháp phân tích nhanh ra thừa số nguyên tố (sẽ xét sau). Ở đây ta xét phương pháp đơn giản trong lập trình và có độ phức tạp $O(\sqrt{k})$.

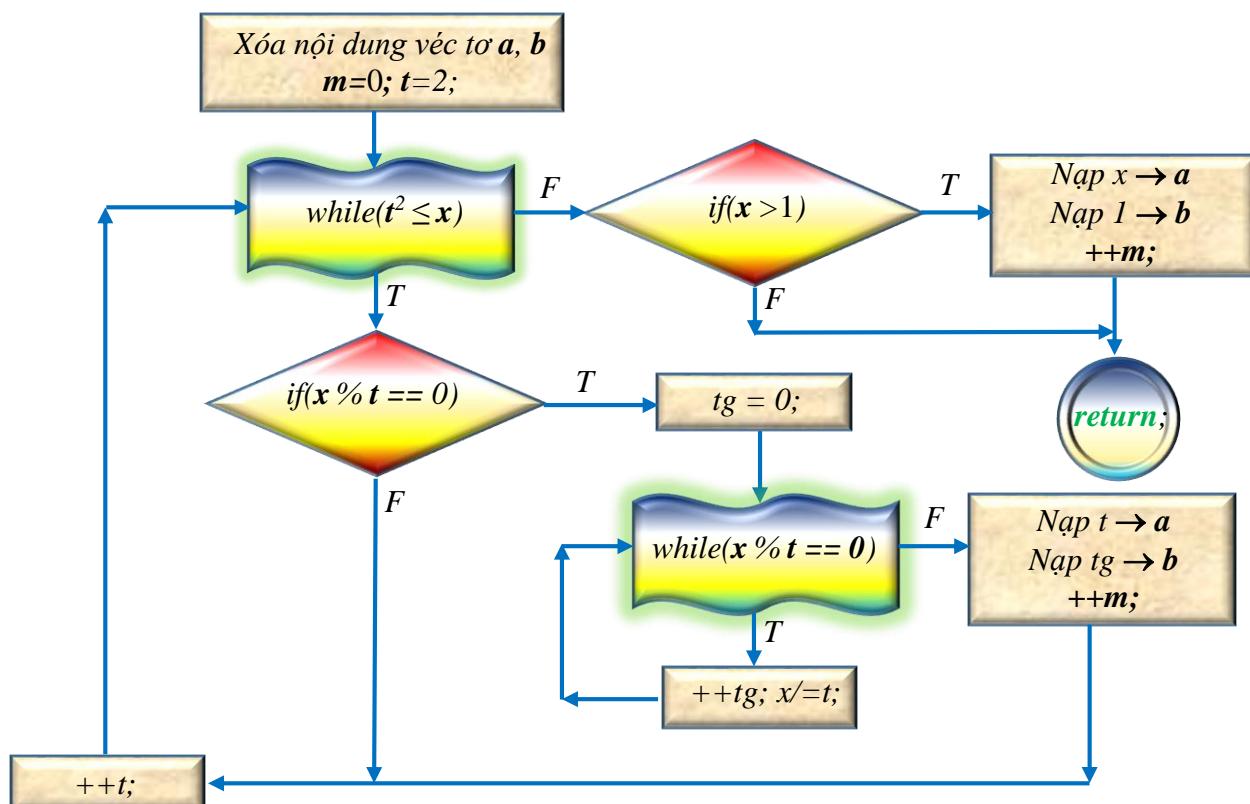
Giải thuật:

Nhận xét:

- ⊕ Do có yêu cầu đưa ra số lượng số nguyên tố khác nhau trong phép phân rã \mathbf{k} ra tích các thừa số nguyên tố nên cần lưu trữ các giá trị a_i, b_i nhận được trong quá trình phân tích,
- ⊕ Vì m không biết trước, để tiện xử lý a_i được tổ chức lưu giữ trong `vector<int64_t> a, bi` – lưu giữ trong `vector<int> b`.

Xử lý:

Giải thuật xử lý với mỗi số nguyên $\mathbf{k} = \mathbf{x}$:



Chương trình

```
#include <bits/stdc++.h>
#define NAME "grade."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n,m;
int64_t k,res,ans=1e17;
vector<int64_t>a;
vector<int>b;

void calc_p(int64_t x)
{
    int64_t t,tg;
    m=0; t=2;
    while(t*t<=x)
    {
        if(x%t==0)
        {
            tg=0;
            while(x%t==0) x/=t, ++tg;
            a.push_back(t); b.push_back(tg);
            ++m;
        }
        ++t;
    }
    if(x>1)
    {
        a.push_back(x); b.push_back(1); ++m;
    }
}

void calc_x(int64_t x)
{
    res=0;
    int t=n;
    while(t)
    {
        t/=x;
        res+=t;
    }
}

int main()
{
    fi>>n>>k;
    calc_p(k);
    //fo<<k<<' ' <<m<<'\n';
    for(int i=0;i<m;++i)
    {
        calc_x(a[i]);
        if(ans>res/b[i]) ans=res/b[i];
    }
    fo<<ans;

    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Tính phần dư trong giải thừa rút gọn

Đặt vấn đề

Có nhiều bài toán tổ hợp, đếm cấu hình, ... đòi hỏi phải làm việc với $n!$, trong đó giản ước các thừa số chia hết cho p ($1 < p < n$, $p -$ nguyên tố), tức là thừa số dạng $x = a \times p^k$ ($1 \leq a < p$) được thay bằng a . Ta gọi kết quả tính là *gai thừa rút gọn*.

Yêu cầu: Tính số dư của phép chia giải thừa rút gọn cho p .

Việc tính nhanh số dư của phép chia giải thừa rút gọn cho p sẽ hỗ trợ rất nhiều trong việc xử lý công thức ở các bài toán tổ hợp.

Ta sẽ xét với trường hợp p không quá lớn.

Giải thuật

Xét giải thừa rút gọn theo p ở dạng tƣờng minh:

$$\begin{aligned} n! \% p &= 1 \cdot 2 \cdot 3 \cdots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot (p+1) \cdot (p+2) \cdots \cdot (2p-1) \cdot \underbrace{2}_{2p} \cdot (2p+1) \cdots \\ &\quad \cdot (p^2 - 1) \cdot \underbrace{1}_{p^2} \cdot (p^2 + 1) \cdots \cdot n = \\ &= 1 \cdot 2 \cdot 3 \cdots \cdot (p-2) \cdot (p-1) \cdots \underbrace{1}_{p} \cdot 1 \cdot 2 \cdots \cdot (p-1) \cdot \underbrace{2}_{2p} \cdot 1 \cdot 2 \cdots \cdot (p-1) \cdot \underbrace{1}_{p^2} \cdots \\ &\quad \cdot 1 \cdot 2 \cdots \cdot (n \% p) \pmod{p}. \end{aligned}$$

Dãy các thừa số có thể chia thành các nhóm độ dài $p-1$, trừ nhóm cuối cùng có thể ngắn hơn:

$$\begin{aligned} n! \% p &= \underbrace{1 \cdot 2 \cdots \cdot (p-2) \cdot (p-1)}_{1st} \cdot \underbrace{1 \cdot 1 \cdot 2 \cdots \cdot (p-1) \cdot 2}_{2nd} \cdots \underbrace{1 \cdot 2 \cdots \cdot (p-1) \cdot 1}_{p-th} \cdots \\ &\quad \cdot \underbrace{1 \cdot 2 \cdots \cdot (n \% p)}_{tail} \pmod{p}. \end{aligned}$$

Các khối độ dài giống nhau có một phần chung. Mỗi phần chung tương đương với giá trị $(p-1)! \bmod p$.

Giá trị của phần chung có thể tính trực tiếp hoặc áp dụng *định lý Wilson*:

$$(p-1)! \bmod p = p-1$$

Việc nhân các phần chung sẽ cho ta kết quả đan xen $(p-1)$ và 1 phụ thuộc vào số lần nhân là lẻ hay chẵn. Khối không đầy đủ (khối cuối cùng) – tính riêng với chi phí $O(p)$.

Biểu thức dãy xuất hiện trong bài toán rút gọn theo mô đun p sẽ có dạng:

$$n! \% p = \underbrace{\dots \cdot 1 \cdot \dots \cdot 2 \cdot \dots \cdot 3 \cdot \dots \cdot \dots}_{\text{khối}} \cdot \underbrace{(p-1) \cdot \dots \cdot 1 \cdot \dots \cdot 1 \cdot \dots \cdot 2 \cdot \dots}_{\text{khối}}$$

Ta lại có giai thừa rút gọn, nhưng với kích thước nhỏ hơn – bằng số lượng các khối giống nhau và là n/p . Như vậy, với chi phí $O(p)$ ta đã đưa bài toán ban đầu về bài toán tính $(n/p)! \% p$. Suy diễn đệ quy này có chiều sâu $O(\log_p n)$ và như vậy, độ phức tạp của toàn giải thuật sẽ là $O(p \log_p n)$.

Hàm tính giai thừa rút gọn theo mô đun p

Việc tính toán có thể thực hiện theo sơ đồ lặp sau:

```
int factmod (int n, int p) {
    int res = 1;
    while (n > 1) {
        res = (res * ((n/p) % 2 ? p-1 : 1)) % p;
        for (int i=2; i<=n%p; ++i)
            res = (res * i) % p;
        n /= p;
    }
    return res % p;
}
```



Căn gốc theo mô đun

Định nghĩa

Căn gốc theo mô đun n (*Primitive root modulo n*) là số nguyên g mà tất cả các số nguyên tố cùng nhau với n đều là lũy thừa của g tính theo mô đun n , nói một cách khác, nếu a là số nguyên tố cùng nhau với n thì tồn tại k để $g^k \equiv a \pmod{n}$.

Trong trường hợp riêng, khi n là nguyên tố thì lũy thừa của g chạy hết các số từ 1 đến $n-1$.

Tồn tại

Căn gốc theo mô đun n tồn tại khi và chỉ khi n hoặc là lũy thừa của một số nguyên tố lẻ, hoặc là 2 lần của một lũy thừa nguyên tố, hoặc bằng 1, 2 hay 4.

Định lý này được Gauss chứng minh năm 1801.

Mối quan hệ với hàm Euler

Xét g – căn gốc theo mô đun n . Khi đó số k nhỏ nhất thỏa mãn $g^k \equiv 1 \pmod{n}$ là $\Phi(n)$ và ngược lại.

Mối quan hệ này sẽ được sử dụng trong giải thuật tìm căn gốc.

Ngoài ra ta còn có mối quan hệ sau: nếu với n tồn tại một căn gốc, thì số lượng căn gốc khác nhau sẽ là $\Phi(\Phi(n))$.

Giải thuật tìm căn gốc

Từ mối quan hệ của căn gốc với hàm Euler với mỗi g nếu số k nhỏ nhất thỏa mãn $g^k \equiv 1 \pmod{n}$ là $\Phi(n)$ thì g là căn gốc. Do với mọi số a đều có $a^{\Phi(n)} \equiv 1 \pmod{n}$ nên để biết g có phải là căn gốc hay không cần kiểm tra điều kiện $g^d \equiv 1 \pmod{n}$ với mọi $d < \Phi(n)$. Nếu không có d nào thỏa mãn thì g là căn gốc. Tuy vậy, mặc dù đã thu hẹp đáng kể phạm vi xét cạn, giải thuật này vẫn còn hoạt động quá chậm.

Theo định lý Lagrange, lũy thừa của số bất kỳ theo mô đun n đều là ước của $\Phi(n)$. Như vậy ta chỉ cần kiểm tra điều kiện $g^d \not\equiv 1 \pmod{n}$ với các d là ước của $\Phi(n)$. Tốc độ của giải thuật đã tăng một cách đáng kể. Nhưng ta sẽ còn đi xa hơn.

Phân tích $\Phi(n)$ ra thừa số nguyên tố:

$$\phi(n) = p_1^{a_1} \cdots p_s^{a_s}$$

Nếu d là một ước thực của $\Phi(n)$ thì tồn tại j và k thỏa mãn $d \times k = \Phi(n) / p_j$.

Trong giải thuật tìm kiếm trên ta chỉ cần kiểm tra với các d dạng $\Phi(n) / p_j$.

Nếu $g^d \equiv 1 \pmod{n}$ thì

$$g^{\frac{\phi(n)}{p_j}} \equiv g^{d \cdot k} \equiv (g^d)^k \equiv 1^k \equiv 1 \pmod{n},$$

tức là cũng tìm thấy một số mũ $\Phi(n)/p_j$ cho kết quả lũy thừa của \mathbf{g} bằng 1 theo mô đun n .

Như vậy, giải thuật tìm \mathbf{g} sẽ bao gồm các bước sau:

- ⊕ Tìm $\Phi(n)$ và phân tích nó ra thừa số nguyên tố,
- ⊕ Duyệt các $\mathbf{g} = 1, 2, \dots, n-1$, và với mỗi \mathbf{g} tính $g^{\frac{\Phi(n)}{p_i}} \pmod{n}$,

Nếu với một \mathbf{g} nào đó mọi giá trị tính được đều khác 1 thì \mathbf{g} là số cần tìm.

Đánh giá độ phức tạp

- ♣ Số $\Phi(n)$ có số lượng ước là $O(\log(\Phi(n)))$,
- ♣ Thời gian tính lũy thừa là $O(\log(n))$,
- ♣ Gọi **ans** – kết quả căn gốc tìm được,
- ♣ Độ phức tạp của toàn bộ giải thuật là $O(\text{ans} \times \log \Phi(n) \times \log n)$

Tốc độ tăng của căn gốc so với tốc độ tăng của n là khá chậm và chỉ được đánh giá gần đúng, chỉ biết rằng căn gốc là một số không lớn. Một trong các đánh giá được sử dụng rộng rãi là của Shoup dựa trên giả thuyết Riman. Theo đánh giá này, căn gốc có giá trị cỡ $O(\log^6 n)$.

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "prim_root."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t a,n,p,ans;

int64_t binpow(int64_t a,int64_t b,int64_t p)
{int64_t ans=1;
 a%=p;
 while(b)
 {
     if(b&1) ans=(ans*a)%p, --b;
     a=(a*a)%p;
     b>>=1;
 }
 return ans;
}

int64_t f_phi(int64_t n)
{int64_t result = n;
 for(int i = 2; i*i<= n; ++i)
```

```

    if (n%i==0)
    {
        while (n%i==0) n/=i;
        result-=result/i;
    }
    if (n>1) result-=result/n;
    return result;
}

int64_t generator(int64_t p)
{
    vector<int64_t> fact;
    int64_t phi=f_phi(p),m;
    m=phi;
    for(int64_t i=2; i*i<=m; ++i)
        if (m%i==0)
        {
            fact.push_back(i);
            while (m%i==0) m/=i;
        }
    if (m>1) fact.push_back(m);
    for(int64_t res=2; res<=p; ++res)
    {
        bool ok=true;
        if (__gcd(res,p)!=1) continue;
        for(int64_t i=0; i<fact.size() && ok; ++i)
            ok&=binpow(res,phi/fact[i],p)!=1;
        if (ok) return res;
    }
    return -1;
}

int main()
{
    fi>>n;
    ans=generator(n);
    fo<<ans;
}

```



Tìm căn giá trị nguyên

Bài toán

Cho số nguyên tố n và các số nguyên a, k . Hãy tìm tất cả các x thỏa mãn điều kiện:

$$x^k \equiv a \pmod{n}$$

Dữ liệu: Vào từ file văn bản DISCROOT.INP gồm một dòng chứa 3 số nguyên a, k và n ($0 \leq a, k \leq 10^9$, $1 < n < 2 \times 10^9$, n – nguyên tố).

Kết quả: Đưa ra file văn bản DISCROOT.OUT dòng thứ nhất chứa số nguyên m – số lượng nghiệm tìm được, dòng thứ 2 chứa m số nguyên – các nghiệm thỏa mãn phương trình đã cho.

Ví dụ:

| DISCROOT.INP |
|--------------|
| 6 3 7 |

| DISCROOT.OUT |
|--------------|
| |

Giải thuật tìm một nghiệm

Trường hợp $a = 0$ ta có ngay $x = 0$.

Với $a > 0$ bài toán ban đầu được dẫn xuất về bài toán lô ga rít rời rạc.

Vì n là số nguyên tố nên nó có căn gốc g và số nguyên bất kỳ trong đoạn từ 1 đến $n-1$ có thể biểu diễn dưới dạng lũy thừa của g , phương trình ban đầu có thể đưa về dạng $(g^y)^k \equiv a \pmod{n}$, trong đó $x = g^y$. Để dàng thấy rằng $(g^y)^k = (g^k)^y$ và ta có phương trình

$$(g^k)^y \equiv a \pmod{n}.$$

Đây là phương trình lô ga rít rời rạc và có thể giải bằng phương pháp Meet-in-the-Middle của Shank với độ phức tạp $O(\sqrt{n} \log n)$, tức là tìm được một lời giải y_0 hay xác định phương trình vô nghiệm.

Như vậy, nếu phương trình có nghiệm, ta đã xác định được một nghiệm của phương trình ban đầu là $x_0 = g^{y_0} \pmod{n}$.

Tìm tất cả các nghiệm khi biết một nghiệm

Ta đã biết căn gốc luôn có bậc là $\Phi(n)$, hay nói cách khác – lũy thừa nhỏ nhất của g cho giá trị 1 là $\Phi(n)$. Vì vậy, nếu bổ sung vào số mũ một số nguyên lần $\Phi(n)$ – kết quả không thay đổi. Với $l = 0, 1, 2, \dots$ ($l \in \mathbb{Z}$).

$$x^k \equiv g^{y_0 \cdot k + l \cdot \phi(n)} \equiv a \pmod{n}$$

Như vậy tất cả các lời giải sẽ có dạng:

$$x = g^{y_0 + \frac{l \cdot \phi(n)}{k}} \pmod{n} \quad \forall l \in \mathbb{Z}$$

l cần được lựa chọn sao cho $\frac{l \cdot \phi(n)}{k}$ nhận giá trị nguyên. Như vậy tử số phải là bội của bội số chung nhỏ nhất của $\Phi(\mathbf{n})$ và \mathbf{k} .

Như vậy công thức dẫn xuất tất cả các nghiệm sẽ là

$$x = g^{y_0 + i \frac{\phi(n)}{\gcd(k, \phi(n))}} \pmod{n} \quad \forall i \in \mathbb{Z}$$

Chương trình

```
#include <bits/stdc++.h>
#define NAME "disroot."
using namespace std;
typedef int64_t ll;
typedef pair<ll, ll> pll;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t a, n, k, p, ans;

int64_t binpow(int64_t a, int64_t b, int64_t p)
{
    int64_t ans=1;
    a%=p;
    while (b)
    {
        if (b&1) ans=(ans*a)%p, --b;
        a=(a*a)%p;
        b>>=1;
    }
    return ans;
}

int64_t generator(int64_t p)
{
    vector<int64_t> fact;
    int64_t phi=p-1, m=phi;
    m=phi;
    for(int64_t i=2; i*i<=m; ++i)
        if (m%i==0)
        {
            fact.push_back(i);
            while (m%i==0) m/=i;
        }
    if (m>1) fact.push_back(m);
    for(int64_t res=2; res<=p; ++res)
    {
        bool ok=true;
```

```

//if(__gcd(res,p)!=1)continue;
for(int64_t i=0;i<fact.size() && ok; ++i)
    ok&=binpow(res,phi/fact[i],p)!=1;
if(ok) return res;

}

return -1;
}

int main()
{
    fi>>a>>k>>n;
    if(a==0){fo<<"1\n0"; return 0;}
    ll g=generator(n);
    ll sq=(ll)sqrt(n+.0)+1;
    vector<pair<ll,ll> >dec(sq);
    for(int i=1;i<=sq;++i)dec[i-1]={binpow(g,i*sq*k%(n-1),n),i};
    sort(dec.begin(),dec.end());
    ll any_ans=-1;
    for(ll i=0;i<sq;++i)
    {
        ll my=binpow(g,i*k%(n-1),n)*a%n;

        auto it=lower_bound(dec.begin(),dec.end(),make_pair(my,0),
                           [&](pll a,pll b){return a.first<b.first || (a.first==b.first
&& (a.second < b.second)); });
        if(it!=dec.end() && it->first==my)
        {
            any_ans=it->second*sq-i;
            break;
        }
    }
    if(any_ans==-1){fo<<"-1"; return 0;}
    ll delta = (n-1)/__gcd(k,n-1);
    vector<ll>ans;
    for(ll cur=any_ans%delta;cur<n-1;cur+=delta)
        ans.push_back(binpow(g,cur,n));
    sort(ans.begin(),ans.end());
    fo<<ans.size()<<'\n';
    for(ll &i:ans) fo<<i<<' ';
    fo<<"\nTime: "<<clock() / (double) 1000<<" sec";
}

```



Xử lý số lớn

Nhu cầu thực hiện các phép tính số học với các số nguyên rất lớn là không cao, đặc biệt trong các bài toán olympic.

Tuy vậy vẫn cần thiết thực hiện các biểu thức số học mà kết quả trung gian có thể vượt quá 64 bit, ví dụ tính $a * b \% p$, trong đó a , b và p là các số nguyên 64 bit.

Ở đây ta sẽ xem xét việc thực hiện các phép cộng, trừ nhân và lấy mô đun với *các số 64 bit không âm*.

Không có vấn đề mới với việc nhập dữ liệu.

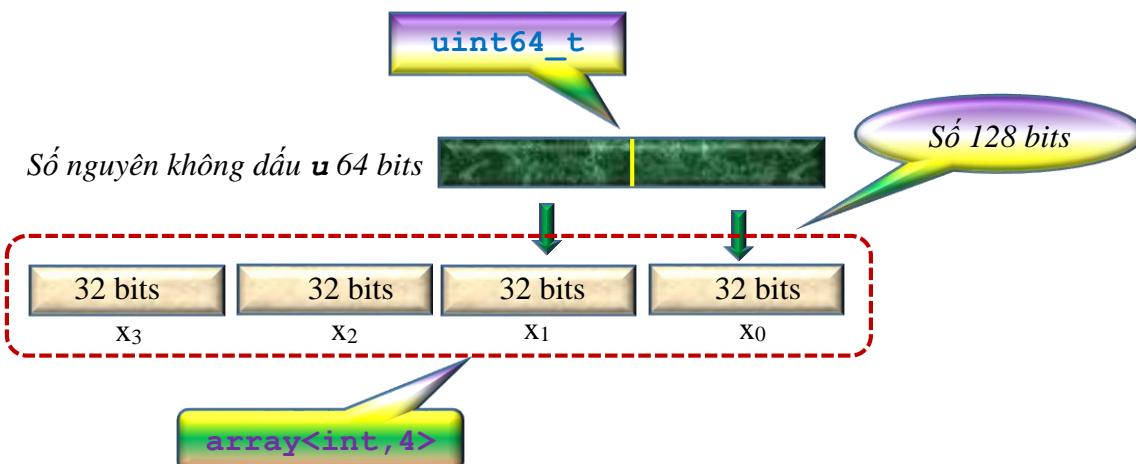
Các vấn đề cần xem xét giải quyết là:

- ✚ Biểu diễn số,
- ✚ Thực hiện các phép tính số học,
- ✚ Xuất kết quả.

Biểu diễn số

Kết quả các phép cộng, trừ, nhân đều không vượt quá 128 bit trong dạng biểu diễn nhị phân. Tư tưởng chung là biểu diễn các số cần xử lý trong cơ số mới BASE đủ lớn và mô phỏng các phép tính cần thực hiện trong hệ cơ số mới.

Cơ sở BASE lớn nhất có thể lựa chọn là $\text{BASE} = 2^{32}$. Trong trường hợp này, mọi giá trị trung gian và kết quả cuối cùng sẽ có không quá 4 chữ số trong hệ cơ số BASE, mỗi chữ số có giá trị nhỏ hơn BASE:



Số 128 bit tối đa có 39 chữ số trong hệ cơ số 10. Trong nhiều trường hợp các kết quả trung gian không vượt quá 10^{36} và vì vậy chỉ cần chọn $\text{BASE} = 10^9$ là đủ. Cơ số này đảm bảo dễ dàng theo dõi hoặc dẫn xuất các giá trị trung gian cũng như kết quả cuối cùng với các số lớn hơn 2^{64} .

Nếu các kết quả trung gian còn được dùng để xử lý tiếp thì sử dụng $\text{BASE} = 2^{32}$ sẽ thuận tiện hơn.

Đổi cơ số

Chuyển số **u** ở dạng 64 bit sang biến **x** dạng 128 bit.

Biến **x** có thể khai báo dưới các dạng **int x[4]** hoặc **array<int, 4> x**. Dạng thứ 2 sẽ thuận tiện hơn trong việc khai báo thuộc tính ở các lời gọi của hàm.

```
Int128ToInt128(uint64_t u)
{
    Int128 v={0};
    v[0]= u%BASE;
    v[1]= u/BASE;
    return v;
}
```

Cần thiết

Việc gán giá trị đầu bằng 0 cho biến trung gian là cần thiết. Hệ thống lập trình không xóa bộ nhớ cho các biến được phân phối trong giai đoạn thực hiện chương trình.

Kiểu **Int128** được xác định bởi khai báo

```
typedef array<uint32_t, 4> Int128;
```

Tổng hai số

Yêu cầu tính tổng **a+b**, trong đó:

- + **a** số nguyên 64 bit, được lưu trữ trong biến **x** kiểu **Int128**,
- + **b** số nguyên 64 bit, được lưu trữ trong biến **y** kiểu **Int128**.

Kết quả lưu ở biến **z** kiểu **int128**.

```
int128 Plus(Int128 u, Int128 v)
{
    Int128 w = {0};
    int debt=0; uint64_t t;
    for(int i=0;i<4;++i)
    {
        t=u[i]+v[i]+debt;
        debt=t/BASE;
        w[i]=t%BASE;
    }
    return w;
}
```

Cần thiết

Tính hiệu

Yêu cầu tính hiệu **a-b**, trong đó:

- + **a** số nguyên 64 bit, được lưu trữ trong biến **x** kiểu **Int128**,
- + **b** số nguyên 64 bit, được lưu trữ trong biến **y** kiểu **Int128**.

Xét trường hợp **a ≥ b**.

Trong trường hợp **a < b**, cần đổi vai trò 2 số và dùng biến riêng để đánh dấu kết quả âm (nếu cần).

Kết quả lưu ở biến **z** kiểu **Int128**.

```
Int128 Minus(Int128 u, Int128 v)
{Int128 w = {0};
 int debt=0;
 for(int i=0;i<4;++i)
 {
     if(u[i]<v[i]+debt)
         w[i]=u[i]+BASE-v[i]-debt, debt=1;
     else w[i]=u[i]-v[i]-debt, debt=0;
 }
 return w;
}
```

Cần thiết

Tính tích

Yêu cầu tính tích **a*b**, trong đó:

- ✚ **a** số nguyên 64 bit, được lưu trữ trong biến **x** kiểu **Int128**,
- ✚ **b** số nguyên 64 bit, được lưu trữ trong biến **y** kiểu **Int128**.

Kết quả lưu ở biến **z** kiểu **Int128**.

```
int128 Mult(int128 u, int128 v)
{int128 w={0};
 uint64_t z[4] = {0};
 for (int i = 0; i < 2; ++i)
     for (int j = 0; j < 2; ++j)
         z[i + j] += (uint64_t)u[i] * v[j];
 for (int i = 0; i < 3; ++i)
 {
     w[i] = z[i] % BASE;
     z[i + 1] += z[i] / BASE;
 }
 return w;
}
```

Tính số dư

Yêu cầu tính phần dư **z%p**, trong đó:

- ✚ **z** số nguyên kiểu **Int128**,
- ✚ **p** số nguyên kiểu **int**.

Kết quả: trả về giá trị kiểu **int**.

```
int Mod(Int128 u, int q)
{int64_t t=0;
 for(int i=3;i>=0;--i)t=(t*BASE+u[i])%q;
 return t;
}
```

Chương trình minh họa

```
#include <bits/stdc++.h>
#define NAME "Big_Num."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
typedef array<uint32_t,4> Int128;
//const uint64_t BASE = (uint64_t)1<<32;
const uint64_t BASE = 1000000000;

uint64_t a,b,p,r;
Int128 x,y,z;

Int128ToInt128(uint64_t u)
{Int128 v={0};
    v[0]= u%BASE;
    v[1]= u/BASE;
    return v;
}

Int128 Plus(Int128 u, Int128 v)
{Int128 w = {0};
    int debt=0; uint64_t t;
    for(int i=0;i<4;++i)
    {
        t=u[i]+v[i]+debt;
        debt=t/BASE;
        w[i]=t%BASE;
    }
    return w;
}

Int128 Minus(Int128 u, Int128 v)
{Int128 w = {0};
    int debt=0;
    for(int i=0;i<4;++i)
    {
        if(u[i]<v[i]+debt)
            w[i]=u[i]+BASE-v[i]-debt, debt=1;
        else w[i]=u[i]-v[i]-debt, debt=0;
    }
    return w;
}

Int128 Mult(Int128 u, Int128 v)
{Int128 w={0};
    uint64_t z[4] = {0};
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            z[i + j] += (uint64_t)u[i] * v[j];
    for (int i = 0; i < 3; ++i)
    {
        w[i] = z[i] % BASE;
        z[i + 1] += z[i] / BASE;
    }
    return w;
}
```

```

int64_t Mod(Int128 u, int q)
{int64_t t=0;
 for(int i=3;i>=0;--i)t=(t*BASE+u[i])%q;
 return t;
}

void Print(Int128 u)
{bool flg=false;
 for(int i=3;i>=0;--i)
 {
    flg|=u[i]>0;
    if(flg) fo<<u[i];
 }
 fo<<'\\n';
}

int main()
{
 fi>>a>>b>>p;
 x =ToInt128(a);

 y =ToInt128(b);
 z = Plus(x,y); Print(z);

 z = Minus(x,y);
 Print(z);

 z = Mult(x,y); Print(z);

 r = Mod(x,p); fo<<"r = "<<r<<'\\n';
}

```



Tính giá trị biểu thức theo mô đun

Cho biểu thức $f(x)$ chỉ chứa các phép tính cộng, trừ, nhân, chia và lũy thừa. Cho các số nguyên dương a và p . Yêu cầu tính $f(a) \% p$. Trong $f(a)$, nếu tồn tại phép chia kết quả phép chia là nguyên.

Dễ dàng thấy rằng:

Nếu $f(a) = a + b$, thì $f(a) \% p = (a \% p + b \% p) \% p$,

Nếu $f(a) = a - b$, thì $f(a) \% p = (a \% p - b \% p + p \% p) \% p$,

Nếu $f(a) = a \times b$, thì $f(a) \% p = (a \% p \times b \% p) \% p$,

Nếu $f(a) = a^n$, bằng phương pháp tính nhanh lũy thừa ta có thể tính $a^n \% p$ với độ phức tạp $O(\log n)$.

Xét trường hợp $f(a) = \frac{g(a)}{b}$, trong đó $g(a)$ chia hết cho b .

Nếu p và b đủ nhỏ để $b \times p$ có thể biểu diễn bằng số nguyên 64 bit, ta có

$$g(a) = k \times (p \times b) + x \times b,$$

trong đó $0 \leq x < p$.

Như vậy công thức tính phần dư sẽ là

$$\frac{g(a)}{b} \% p = (g(a) \% (p \times b)) / b$$

Ví dụ $\frac{180}{15} \% 10 = (180 \% (10 \times 15)) / 15 = 2$.

Trường hợp $p \times b \geq 2^{64}$:

Nếu b và p nguyên tố cùng nhau thì tồn tại phần tử nghịch đảo b^{-1} để $(b \times b^{-1}) \% p = 1$.

Do $g(a)$ chia hết cho b nên $\exists c \geq 0$, nguyên: $g(a) = c \times b$.

Ta có

$$\frac{g(a)}{b} = \frac{g(a) \times b^{-1}}{b \times b^{-1}} = c$$

Mặt khác, ta có

$$(g(a) \times b^{-1}) \% p = (c \times b \times b^{-1}) \% p = (c \% p \times (b \times b^{-1}) \% p) \% p = c \% p.$$

Như vậy

$$\frac{g(a)}{b} \% p = (g(a) \times b^{-1}) \% p$$

Ví dụ: $g(a) = 27 \times 9$, $b = 3$, $p = 11$.

Nghịch đảo của 3 theo mô đun 11 là 4.

Ta có

$$\frac{27 \times 9}{3} \% 11 = (27 \times 9 \times 4) \% 11 = (5 \times 9 \times 4) \% 11 = 1 \times 4 = 4$$

Việc tìm số nghịch đảo theo mô đun đã được trình bày ở các phần trên.



Giải thuật Leman

Phân số liên tục (Continued fractions)

Phân số liên tục là biểu thức hữu hạn hoặc vô hạn dạng:

$$[a_0; a_1, a_2, a_3, \dots] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \dots}}}$$

trong đó:

- ✚ a_0 – số nguyên,
- ✚ a_i , $i = 1, 2, 3, \dots$ – số nguyên dương.

a_i , $i = 0, 1, 2, \dots$ được gọi là *phần tử* của phân số liên tục.

Mọi số thực đều có thể biểu diễn dưới dạng phân số liên tục (hữu hạn hoặc vô hạn). Một số thực được biểu diễn dưới dạng phân số liên tục hữu hạn khi và chỉ khi là số hữu tỷ.

Một trong những ứng dụng quan trọng của phân số liên tục là để tìm biểu diễn gần đúng số thực dưới dạng phân số thường với độ chính xác bất kỳ cho trước.

Phân số liên tục được sử dụng rộng rãi trong lý thuyết số và trong tính toán khoa học, trong vật lý, cơ học không gian, trong kỹ thuật, ... và là một công cụ hết sức hữu hiệu trong việc phân tích toán học cho các toán thực tế hoặc lý thuyết thuộc mọi lĩnh vực.

Xét số thực x . Ký hiệu $\lfloor x \rfloor$ là phần nguyên của x . Nếu biểu diễn x dưới dạng phân số liên tục, ta có:

$$a_0 = \lfloor x \rfloor, x_0 = x - a_0,$$

$$a_1 = \left\lfloor \frac{1}{x_0} \right\rfloor, x_1 = \frac{1}{x_0} - a_1,$$

.....

$$a_n = \left\lfloor \frac{1}{x_{n-1}} \right\rfloor, x_n = \frac{1}{x_{n-1}} - a_n,$$

.....

Nếu x là số hữu tỷ thì việc phân tích sẽ khi nhận được $x_n = 0$. Giải thuật hữu hiệu để phân tích một phân số thường ra phân số liên tục là *Giải thuật Euclidean*.

Với x là số vô tỷ việc phân tích sẽ kéo dài vô hạn và ta có một phân số liên tục vô hạn

$$x = [a_0; a_1, a_2, a_3, \dots]$$

Dãy số $[a_0; a_1, a_2, a_3, \dots]$ có thể chứa chu kỳ (dãy con các số lặp lại), khi đó ta có một **phân số liên tục tuần hoàn**.

Một số vô tỷ được biểu diễn dưới dạng phân số liên tục tuần hoàn khi và chỉ khi nó là nghiệm vô tỷ của một phương trình bậc 2 với các hệ số nguyên.

Nếu ngắt ra $n+1$ phần tr đầu tiên của $[a_0; a_1, a_2, a_3, \dots]$ ta có một **phân số thích hợp** và nó $[a_0; a_1, \dots, a_n]$ tương ứng với số hữu tỷ $\frac{p_n}{q_n}$.

Các phân số thích hợp với **n chẵn** tạo thành dãy số **tăng dần** $\frac{q_n}{p_n}$ có giới hạn tiến tới x. Các phân số thích hợp với **n lẻ** tạo thành dãy số **giảm dần** có giới hạn tiến tới x. Như vậy x luôn nằm ở khoảng giữa 2 phân số thích hợp liên tục.

Euler đã dẫn xuất công thức truy hồi tinhstwr số và mẫu số của các phân số thích hợp:

$$p_{-1} = 1, \quad p_0 = a_0, \quad p_n = a_n p_{n-1} + p_{n-2};$$

$$q_{-1} = 0, \quad q_0 = 1, \quad q_n = a_n q_{n-1} + q_{n-2}.$$

$\{p_n\}$ và $\{q_n\}$ là các dãy số tăng ngặt.

Tử số và mẫu số của 2 phân số thích hợp liên tiếp nhau có mối quan hệ:

$$p_n q_{n-1} - q_n p_{n-1} = (-1)^{n-1},$$

Các phân số thích hợp đều là **tối giản**.

Mối quan hệ nêu trên có thể viết dưới dạng:

$$\frac{p_n}{q_n} - \frac{p_{n-1}}{q_{n-1}} = \frac{(-1)^{n-1}}{q_{n-1} q_n}.$$

Từ đó ta có:

$$\left| x - \frac{p_{n-1}}{q_{n-1}} \right| < \frac{1}{q_{n-1} q_n} < \frac{1}{q_{n-1}^2}.$$

Như vậy phân số thích hợp $\frac{p_n}{q_n}$ là biểu diễn x gần đúng tốt nhất trong số các phân số có mẫu số không vượt quá q_n .

Ví dụ:

➡ Số $\pi=3,14159265\dots$ có dãy phân số thích hợp

$$3, 22/7, 333/106, 355/113, 103993/33102, \dots$$

➡ Số $\sqrt{2} = [1; 2, 2, 2, 2, \dots]$

➡ Lát cắt vàng $\varphi = [1; 1, 1, 1, \dots]$

➡ Số hữu tỷ $9/4 = [2; 3, 1] = [2; 4]$

Mọi số hữu tỷ đều có 2 cách khác nhau biểu diễn bằng phân số liên tục.

Giải thuật Leman phân tích số nguyên ra thừa số

Xét số nguyên n lẻ và lớn hơn 8.

B1. Với $a = 2, 3, 4, \dots, \lfloor n^{1/3} \rfloor$ kiểm tra điều kiện n chia hết cho a , nếu tìm được các thừa số thì ghi nhận và giảm n một cách tương ứng.

Ký hiệu $m = \lfloor n^{1/3} \rfloor + 1$, nếu $m^2 \leq n$ – việc phân tích ra thừa số chưa kết thúc, chuyển tới bước B2.

B2. Có 2 khả năng:

✚ n – số nguyên tố,

✚ n – hợp số, khi đó $n = p \times q$, với $n^{1/3} < p \leq q < n^{2/3}$, p và q – nguyên tố, $d = 0, \dots, \left\lfloor \frac{n^{1/6}}{4\sqrt{k}} \right\rfloor + 1$

Xét với mọi $k = 1, 2, \dots, \lfloor n^{1/3} \rfloor$ và mọi kiểm tra số

$$(\lfloor \sqrt{4kn} \rfloor + d)^2 - 4kn$$

có phải là một số chính phương hay không. Nếu đó là một số chính phương thì với $A = \lfloor \sqrt{4kn} \rfloor + d$ và $B = \sqrt{A^2 - 4kn}$ kiểm tra điều kiện:

$$\begin{aligned} A^2 &\equiv B^2 \pmod{n} \\ \text{hoặc } (A-B)(A+B) &\equiv 0 \pmod{n} \end{aligned}$$

Nếu điều kiện trên thỏa mãn thì $d^* = \gcd(A - B, n)$ tính và kiểm tra điều kiện $1 < d^* < n$.

Nếu tìm thấy d^* thỏa mãn điều kiện đã nêu thì ta có $n = d^* \cdot (n/d^*)$,

Trong trường hợp ngược lại – có n là số nguyên tố.

Giải thuật Leman có thể được sử dụng để kiểm tra n có phải là số nguyên tố hay không.

Bước 2 của giải thuật Leman là sự phát triển của giải thuật Ferma và tìm ước của n dựa trên đẳng thức $x^2 - y^2 = 4 \times k \times n$. Giải thuật đặt nền móng trên cơ sở của định lý sau:

Định lý: Nếu n là một hợp số và $n = p \times q$, trong đó p, q – lẻ, là các số nguyên tố cùng nhau và thỏa mãn điều kiện $n^{1/3} < p < q < n^{2/3}$, khi đó tồn tại các số nguyên $x, y, k \geq 1$ thỏa mãn các điều kiện:

$$\blacksquare \quad x^2 - y^2 = 4kn \text{ với } k < n^{1/3},$$

$$\blacksquare \quad 0 \leq x - \lfloor \sqrt{4kn} \rfloor < \frac{n^{1/6}}{4\sqrt{k}} + 1$$

Bổ đề:

Nếu các điều kiện của định lý được thực hiện thì tồn tại các số nguyên \mathbf{r} và \mathbf{s} thỏa mãn các điều kiện $\mathbf{r} \times \mathbf{s} < \mathbf{n}^{1/3}$ và $|\mathbf{p} \times \mathbf{r} - \mathbf{q} \times \mathbf{s}| < \mathbf{n}^{1/3}$.

Chứng minh bối đề:

Nếu $\mathbf{p} = \mathbf{q}$, tức là $\mathbf{n} = \mathbf{p}^2$, thì ta có $\mathbf{r} = \mathbf{s} = 1$.

Xét trường hợp $\mathbf{p} < \mathbf{q}$. Phân tích \mathbf{q}/\mathbf{p} ra phân số liên tục. Gọi $\mathbf{p}_j/\mathbf{q}_j$ là phân số thích hợp thứ j .

Ta có $\mathbf{p}_0 = \lfloor \mathbf{q}/\mathbf{p} \rfloor$, $\mathbf{q}_0 = 1$ và $0 < \mathbf{p}_0 \times \mathbf{q}_0 < \mathbf{n}^{1/3}$ vì $\mathbf{q}/\mathbf{p} = \mathbf{n}^{2/3}/\mathbf{n}^{1/3} = \mathbf{n}^{1/3}$.

Chọn \mathbf{m} nhỏ nhất thỏa mãn điều kiện

$$\begin{cases} p_m q_m < n^{1/3} \\ p_{m+1} q_{m+1} > n^{1/3} \end{cases}$$

Số \mathbf{m} tồn tại bởi vì ở phân số thích hợp cuối cùng mẫu số $\mathbf{q}_N = \mathbf{p} > \mathbf{n}^{1/3}$. Ta sẽ chứng minh rằng $\mathbf{r} = \mathbf{p}_m$ và $\mathbf{s} = \mathbf{q}_m$ thỏa mãn điều khẳng định của bối đề.

Rõ ràng là $\mathbf{r} \times \mathbf{s} < \mathbf{n}^{1/3}$.

Tùy tính chất của các phân số thích hợp có:

$$\left| \frac{r}{s} - \frac{q}{p} \right| \leqslant \left| \frac{r}{s} - \frac{p_{m+1}}{q_{m+1}} \right| = \frac{1}{sq_{m+1}}$$

Xét trường hợp

$$\frac{p_{m+1}}{q_{m+1}} \leqslant \frac{q}{p}$$

Ta có

$$|pr - qs| = ps \left| \frac{r}{s} - \frac{q}{p} \right| \leqslant \frac{ps}{sq_{m+1}} = \frac{p}{q_{m+1}} = \sqrt{\frac{p}{q_{m+1}}} \sqrt{\frac{p}{q_{m+1}}} \leqslant \sqrt{\frac{p}{q_{m+1}}} \sqrt{\frac{q}{p_{m+1}}} = \sqrt{\frac{n}{p_{m+1}q_{m+1}}} < \frac{n^{1/2}}{n^{1/6}} = n^{1/3}$$

Và đó là điều cần chứng minh.

Trường hợp

$$\frac{p_{m+1}}{q_{m+1}} > \frac{q}{p}$$

Ta có

$$\frac{p_{m+1}}{q_{m+1}} > \frac{q}{p} > \frac{p_m}{q_m}$$

Từ đó suy ra

$$\frac{q_m}{p_m} > \frac{p}{q} > \frac{q_{m+1}}{p_{m+1}}$$

Theo tính chất của phân số liên tục, tồn tại bất đẳng thức:

$$\frac{1}{rq} \leqslant \left| \frac{s}{r} - \frac{p}{q} \right| \leqslant \left| \frac{s}{r} - \frac{q_{m+1}}{p_{m+1}} \right| = \frac{1}{rp_{m+1}}$$

Từ đó suy ra

$$1 \leqslant |sq - pr| = rq \left| \frac{s}{r} - \frac{p}{q} \right| \leqslant \frac{rq}{rp_{m+1}} = \frac{q}{p_{m+1}} = \sqrt{\frac{q}{p_{m+1}} \frac{q}{p_{m+1}}} \leqslant \sqrt{\frac{q}{p_{m+1}}} \sqrt{\frac{p}{q_{m+1}}} = \sqrt{\frac{n}{p_{m+1} q_{m+1}}} < \frac{n^{1/2}}{n^{1/6}} = n^{1/3}$$

Đó là điều cần chứng minh.

Chứng minh định lý:

Gọi \mathbf{p} và \mathbf{q} là 2 ước lẻ của n .

Đặt $\mathbf{x} = \mathbf{p} \times \mathbf{r} + \mathbf{q} \times \mathbf{s}$ và $\mathbf{y} = \mathbf{p} \times \mathbf{r} - \mathbf{q} \times \mathbf{s}$, trong đó \mathbf{r} và \mathbf{s} – các số thỏa mãn bởđề.

Khi đó:

$$x^2 - y^2 = (pr + qs)^2 - (pr - qs)^2 = 4rspq = 4kn$$

trong đó $\mathbf{k} = \mathbf{r} \times \mathbf{s}$.

Theo bởđề, \mathbf{k} thỏa mãn điều kiện $\mathbf{k} < n^{1/3}$. Kết luận thứ nhất của định lý được chứng minh.

Chứng minh kết luận thứ 2:

$$\text{Đặt } z = x - \lfloor \sqrt{4kn} \rfloor = pr + qs - \lfloor \sqrt{4kn} \rfloor.$$

Từ $x^2 = 4kn + y^2$ suy ra $x \geq \sqrt{4kn}$ và $z \geq 0$.

Xuất phát từ đánh giá cận trên của y ta có:

$$n^{2/3} > y^2 = x^2 - 4kn = (pr + qs + \sqrt{4kn})(pr + qs - \sqrt{4kn}) \geq 2\sqrt{4kn}(pr + qs - \sqrt{4kn}) \geq 2\sqrt{4kn}(z - 1)$$

Từ đó suy ra

$$z < \frac{n^{2/3}}{2\sqrt{4kn}} + 1 = \frac{n^{1/6}}{4\sqrt{k}} + 1$$

Định lý được chứng minh.

Dánh giá độ phức tạp của giải thuật:

Ký hiệu $\lceil \mathbf{y} \rceil$ là số nguyên nhỏ nhất lớn hơn hoặc bằng \mathbf{y} .

Ở giai đoạn một cần thực $\lceil n^{1/3} \rceil$ hiện phép chia để tìm các ước nhỏ của n .

Độ phức tạp của bước 2 được xác định bởi số phép tính cần thiết để kiểm tra xem số có $(\lfloor \sqrt{4kn} \rfloor + d)^2 - 4kn$ phải là một chính phương hay không.

Ta nhận thấy với chính phương chỉ có thể $k < \frac{n^{1/6}}{4}$ đạt được một giá trị $d = 1$.

Như vậy độ phức tạp của giai đoạn 2 bị chặn trên bởi đại lượng

$$\frac{n^{1/6}}{4} \sum_{k=1}^{\lfloor n^{1/6} \rfloor} \frac{1}{\sqrt{k}} + 2(\lceil n^{1/3} \rceil - \lfloor n^{1/6} \rfloor) < 3\lceil n^{1/3} \rceil$$

Tổng kết chung, giải thuật có độ phức tạp $O(n^{1/3})$.

Ví dụ:

THỦY TRIỀU ĐỎ

Thủy triều đỏ là một thuật ngữ thông dụng được dùng để chỉ một trong một loạt các hiện tượng tự nhiên được gọi là tảo nở hoa gây hại hay HABs (viết tắt của cụm từ tiếng anh *Harmful Algal Blooms*). Thuật ngữ *Thủy triều đỏ* được sử dụng đặc biệt để đề cập đến sự nở hoa của một loài tảo có tên là *Karenia brevis*. Thủy triều đỏ gây tác hại lớn cho môi trường vì nó dẫn đến cái chết hàng loạt của nhiều loại sinh vật biển.

Cứ mỗi chu kỳ sinh sản mỗi cá thể tảo phát triển thành a cá thể. Phụ thuộc vào nhiệt độ và nồng độ các chất hữu cơ hòa tan trong nước, tảo *Karenia brevis* có thể phát triển nhanh hay chậm, vì vậy hệ số phát triển a không cố định mà thay đổi theo từng chu kỳ phát triển.

Ở m vùng biển có hiện tượng thủy triều đỏ các nhà khoa học lấy mẫu nước và ước lượng số cá thể tảo độc trong một đơn vị thể tích của từng mẫu, mẫu nước thứ i có b_i cá thể, $i = 1 \dots m$. Ở mức vô hại, ban đầu trong mỗi đơn vị thể tích nước chỉ có thể có một cá thể. Người ta muốn biết nhiều nhất có bao nhiêu chu kỳ sinh sản đã xảy ra, từ đó liên hệ với các yếu tố môi trường để tìm ra nguyên nhân chính gây ra thảm họa môi trường.

Ví dụ, với $b_1 = 20$, nhiều nhất có 3 chu kỳ phát triển với các hệ số phát triển là 2, 2 và 5: $20 = 1 \times 2 \times 2 \times 5$.

Với mỗi mẫu nước hãy đưa ra số chu kỳ phát triển nhiều nhất có thể và hệ số phát của từng chu kỳ, các hệ số đưa ra theo thứ tự không giảm.

Dữ liệu: Vào từ file văn bản HAB.INP:

- ✚ Dòng đầu tiên chứa một số nguyên m ($1 \leq m \leq 1000$),
- ✚ Dòng thứ 2 chứa m số nguyên b_1, b_2, \dots, b_m ($1 < b_i \leq 2 \times 10^9$, $i = 1 \dots m$).

Kết quả: Đưa ra file văn bản HAB.OUT, thông tin về mỗi mẫu nước đưa ra trên 2 dòng, dòng đầu tiên là một số nguyên k – số chu kỳ phát triển nhiều nhất có thể, dòng thứ 2 chứa k số nguyên (theo thứ tự không giảm) xác định hệ số phát triển của tảo trong từng chu kỳ.

Ví dụ:

| HAB.INP |
|----------------------|
| 4 |
| 20 108 896045556 231 |



| HAB.OUT |
|--------------------|
| 3 |
| 2 2 5 |
| 5 |
| 2 2 3 3 3 |
| 7 |
| 2 2 3 7 7 163 9349 |
| 3 |
| 3 7 11 |

Giải thuật: *Ứng dụng giải thuật Leman.*

Nhận xét:

Trong giải thuật cần tính tích 2 số nguyên vì vậy cần khai báo dữ liệu kiểu **int64_t** hoặc ép kiểu dữ liệu trước khi nhân,

Giải thuật đòi hỏi tính căn bậc 3 hoặc bậc 6, kết quả phép lấy căn sẽ là một số thực, để chống sai số làm tròn cần thực hiện phép lấy căn với số có độ chính xác cao (ví dụ - **long double**).

Tổ chức dữ liệu:

Với mỗi số liệu xử lý, số chu kỳ sẽ không vượt quá 31, vì vậy có thể khai báo mảng chứa kết quả với kích thước tĩnh và tổ chức quản lý số ước tìm được hoặc dùng vector để lưu trữ kết quả.

Độ phức tạp của giải thuật: bậc $O(m \ln n^{1/3})$.

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "hab."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");

vector<uint64_t> find_prime_divisors_leman(uint64_t n) {
    vector<uint64_t> divisors;
    uint64_t sqrt3_n_up = ceil(pow((long double)n, (long double)1.0 / 3))+2;

    // finding divisors <= n ^ (1 / 3)
    uint64_t m = 2;
    while (m * m <= n && m < sqrt3_n_up) {
        while (n % m == 0) {
            divisors.push_back(m);
            n /= m;
        }
        ++m;
    }
    if (m * m > n)
    {
        if (n != 1)
            divisors.push_back(n);
        return divisors;
    }

    // either n is prime or n = p * q, n ^ (1 / 3) < p <= q < n ^ (2 / 3)
    long double sqrt6_n = pow((long double)n, (long double)1.0 / 6);
    long double sqrt_n = sqrt((long double)n);
    for(int k = 1; k < sqrt3_n_up; ++k) {
        long double sqrt_k = sqrt((long double)k);
        uint64_t sqrt_4nk = ceil(2 * sqrt_k * sqrt_n) + 2;

        // diff = ([sqrt(4nk)]^2 - 4nk)
        uint64_t diff = sqrt_4nk*sqrt_4nk-4 * k*n;
        while (diff >= 2 * sqrt_4nk - 1) {
            diff = diff - 2 * sqrt_4nk + 1;
            --sqrt_4nk;
        }
        int max_d = ceil(sqrt6_n / (4 * sqrt_k)) + 3;
        for (int d = 0; d < max_d; ++d) {
            uint64_t a = sqrt_4nk + d;
            uint64_t b = round(sqrt((long double)diff));
            if (b * b == diff) {
                // diff is a perfect square, a ^ 2 = b ^ 2 mod n
                uint64_t d1 = __gcd(a + b, n);
                uint64_t d2 = __gcd(a - b, n);
                if (1 < d1 && d1 < n || 1 < d2 && d2 < n) {
                    uint64_t p = (1 < d1 && d1 < n) ? d1 : d2;
                    divisors.push_back(p);
                    divisors.push_back(n / p);
                    return divisors;
                }
            }
            diff += 2 * a + 1;
        }
    }
}
```

```

// n is prime
divisors.push_back(n);
return divisors;
}

int main() {
    uint64_t n,m;
    fi>>m;
    for(int i=0;i<m;++i)
    {
        fi >> n;
        vector<uint64_t> divisors = find_prime_divisors_leman(n);
        fo << divisors.size() << endl;
        for (int i = 0; i < divisors.size(); ++i)
            fo << ((i == 0) ? "" : " ") << divisors[i];
        fo << endl;
    }
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}

```



HÌNH HỌC TÍNH TOÁN

Hợp nhất các đoạn thẳng

Bài toán

Cho n đoạn thẳng trên đường thẳng, đoạn thứ i có tọa độ các điểm đầu và cuối là a_i, b_i . Hãy tính độ dài của hợp các đoạn đã cho.

Dữ liệu: Vào từ file văn bản MERGE SEG.INP:

Dòng đầu tiên chứa một số nguyên n ($1 \leq n \leq 10^5$),

Dòng thứ i trong n dòng sau chứa 2 số nguyên a_i và b_i ($-10^9 \leq a_i \leq b_i \leq 10^9$).

Kết quả: Đưa ra file văn bản MERGE SEG.OUT một số nguyên – độ dài tính được.

Ví dụ:

| MERGE_SEG.INP |
|---------------|
| 4 |
| 1 5 |
| 3 8 |
| 14 16 |
| 10 20 |

| MERGE_SEG.OUT |
|---------------|
| 17 |

Giải thuật giải bài toán đã nêu được Klee công bố năm 1977 với độ phức tạp $O(n \log n)$ và là giải thuật nhanh nhất hiện nay.

Giải thuật

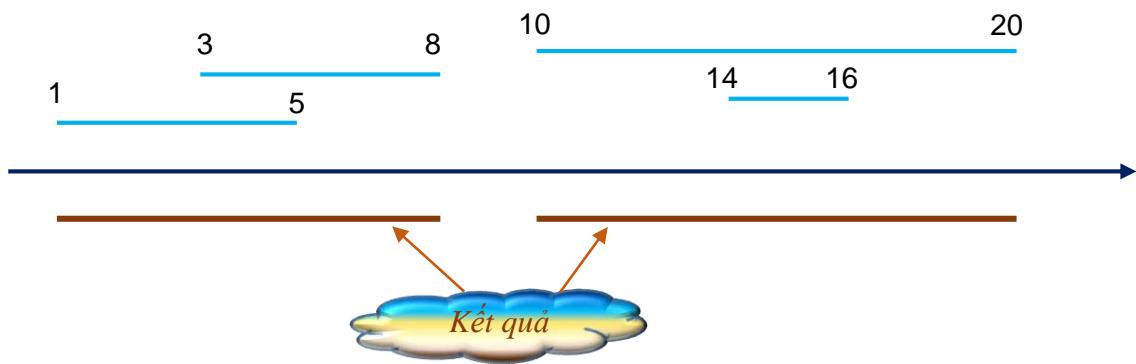
Lưu tất cả các tọa độ vào mảng \mathbf{x} ,

Với mỗi phần tử của \mathbf{x} : đánh dấu là điểm đầu hay điểm cuối của đoạn thẳng,

Sắp xếp \mathbf{x} theo thứ tự tăng dần, với các điểm cùng tọa độ – ưu tiên điểm đầu,

Sử dụng bộ đếm c thống kê số lượng đoạn thẳng giao nhau,

Duyệt mảng \mathbf{x} , nếu $c \neq 0$: cộng thêm vào kết quả hiệu $\mathbf{x}_i - \mathbf{x}_{i-1}$, nếu gặp điểm đầu – tăng c lên 1, gặp điểm cuối – giảm c .



Chương trình

```
#include <bits/stdc++.h>
#define NAME "merge_seg."
using namespace std;
typedef pair<int,bool> pib;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n,a,b,c=1,ans=0;

int main()
{
    fi>>n;
    vector<pib> x(2*n);
    for(int i=0;i<n;++i)
    {
        fi>>a>>b;
        x[2*i] = {a,0};
        x[2*i+1] = {b,1};
    }
    sort(x.begin(),x.end());
    for(int i=1;i<2*n;++i)
    {
        if(c) ans+=x[i].first-x[i-1].first;
        if(x[i].second) --c; else ++c;
    }
    fo<<ans;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Diện tích đa giác cạnh không tự cắt

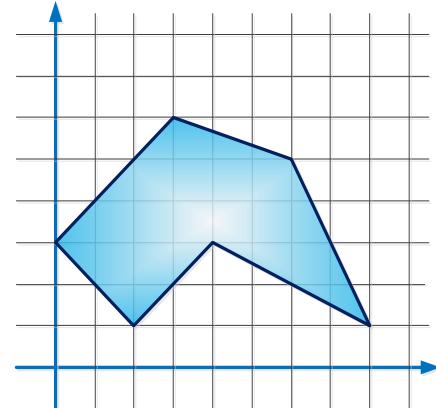
Bài toán

Cho đường gấp khúc khép kín n đỉnh không tự cắt, đỉnh thứ i có tọa độ thực (x_i, y_i) ($|x_i|, |y_i| \leq 10^6$, $i = 1 \div n$). Các đỉnh được liệt kê theo một trình tự nào đó.

Hãy tính diện tích của đa giác tạo bởi đường gấp khúc này.

Dữ liệu: vào từ file POLYG.INP:

- Dòng đầu tiên chữ số nguyên n ($2 < n \leq 10^5$),
- Dòng thứ i trong n dòng sau chứa 2 số thực x_i và y_i .



Kết quả: đưa ra file POLYG.OUT với độ chính xác 6 chữ số sau dấu chấm thập phân.

Ví dụ:

| POLYGON.INP |
|-------------|
| 6 |
| 2 1 |
| 0 3 |
| 3 6 |
| 6 5 |
| 8 1 |
| 4 3 |

| POLYGON.OUT |
|-------------|
| 20.000000 |

Giải thuật

Công thức tính diện tích đa giác:

$$S = \frac{1}{2} \left| \sum_{i=1}^n (x_i y_{i+1} - x_{i+1} y_i) \right|$$

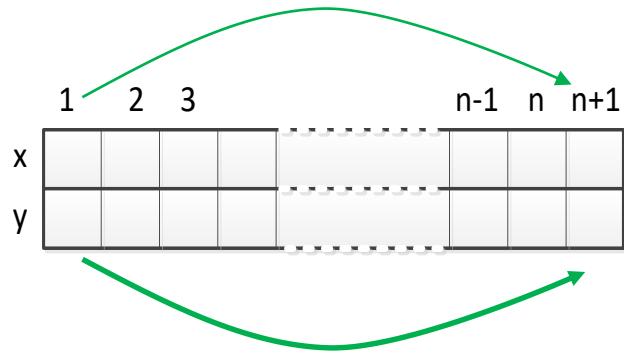
Trong đó $x_{n+1} \equiv x_1$, $y_{n+1} \equiv y_1$.

Lưu ý: biểu thức $\sum_{i=1}^n (x_i y_{i+1} - x_{i+1} y_i)$ cho giá trị dương khi các đỉnh được liệt kê theo chiều ngược kim đồng hồ và giá trị âm trong trường hợp ngược lại.

Dữ liệu vòng tròn:

Tạo $x_{n+1} = x_1$,

$y_{n+1} = y_1$.



Chương trình

```
#include <bits/stdc++.h>
#define NAME "polyg."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n;
double s=0,a,b;

int main()
{
    fi>>n;
    vector<double>x(n+1), y(n+1);
    for(int i=0;i<n;++i) fi>>x[i]>>y[i];
    x[n]=x[0]; y[n]=y[0];
    for(int i=0;i<n;++i)
        s+=x[i]*y[i+1]-x[i+1]*y[i];
    s=abs(s)/2;
    fo<<fixed<<setprecision(6)<<s;
}
```



Định lý Pick

Xét đa giác diện tích khác 0, cạnh không tự cắt và trong mặt phẳng với hệ tọa độ Đè các tọa độ các đỉnh đa giác đều nguyên.

Định lý

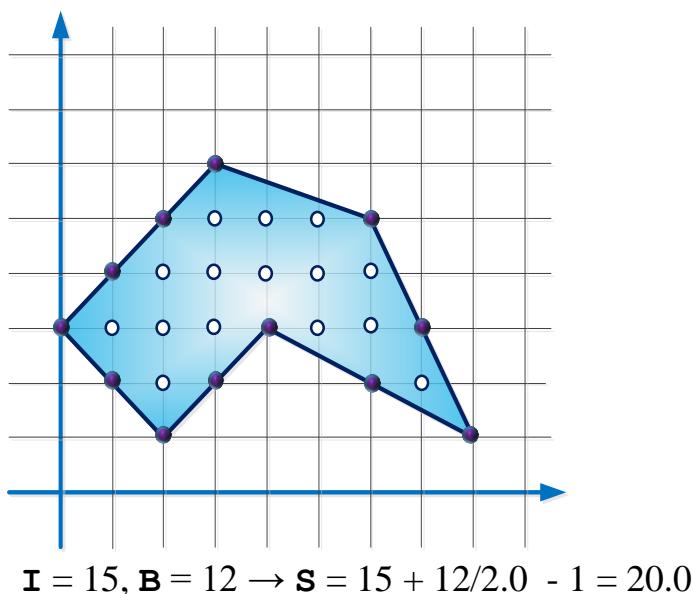
Gọi S là diện tích của đa giác, B – số điểm có tọa độ nguyên nằm trên cạnh của đa giác, I – số điểm trong có tọa độ nguyên. Khi đó tồn tại quan hệ

$$S = I + \frac{B}{2.0} - 1$$

Định này do nhà toán học Áo G. A. Pick phát biểu và chứng minh năm 1899.

Dựa vào định lý này, nếu biết B và I ta có thể tính diện tích đa giác mà không cần quan tâm đến tọa độ cụ thể của các đỉnh.

Ví dụ, với đa giác của hình dưới đây ta có :



Chứng minh

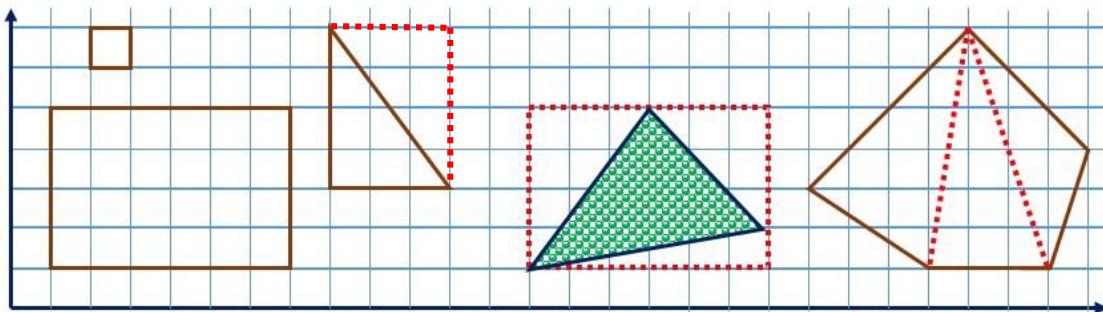
Định lý đúng với trường hợp hình vuông đơn vị,

Nếu đa giác là hình chữ nhật cạnh độ dài a, b song song với trục tọa độ, số điểm trong là $I = (a-1) \times (b-1)$, số điểm trên cạnh là $B = 2 \times (a+b)$. Để dàng thấy được là diện tích $S = a \times b = I + B/2.0 - 1$.

Trường hợp đa giác là tam giác vuông có các cạnh góc vuông song song với trục tọa độ: đa giác là một nửa hình chữ nhật nếu ta cắt hình chữ nhật theo đường chéo. Gọi c là số điểm có tọa độ nguyên trên đường chéo này. Để dàng chứng minh diện tích hình chữ nhật (và từ đó – diện tích tam giác vuông) không phụ thuộc vào c .

Trường hợp tam giác bất kỳ: bằng cách ghép thêm tối đa 3 tam giác vuông có các cạnh góc vuông song song với trực tọa độ ta đưa tam giác về hình chữ nhật cạnh song song với trực tọa độ. Lưu ý là diện tích các tam giác bổ sung không phụ thuộc vào số điểm nguyên trên cạnh huyền ta có điều cần chứng minh.

Với đa giác bất kỳ: chia thành các tam giác.



Lưu ý: Định lý Pick không áp dụng với trường hợp 3 hay nhiều chiều.



Tập điểm phủ các đoạn thẳng

Bài toán 1

Cho n đoạn thẳng trên đường thẳng, đoạn thứ i có tọa độ các điểm đầu và cuối là a_i, b_i . Hãy xác định số lượng ít nhất các điểm cần chọn để mỗi đoạn thẳng đã cho chứa ít nhất một điểm trong số đã chọn.

Dữ liệu: Vào từ file văn bản COVERING.INP:

- ⊕ Dòng đầu tiên chứa một số nguyên n ($1 < n \leq 10^5$),
- ⊕ Dòng thứ i trong n dòng sau chứa 2 số nguyên a_i và b_i ($-10^9 \leq a_i \leq b_i \leq 10^9$).

Kết quả: Đưa ra file văn bản COVERING.OUT một số nguyên – số điểm ít nhất cần chọn.

Ví dụ:

| COVERING.INP |
|--------------|
| 7 |
| 1 5 |
| 3 8 |
| 30 32 |
| 25 35 |
| 33 33 |
| 10 20 |
| 14 16 |

| COVERING.OUT |
|--------------|
| 4 |

Đây là mô hình toán học của một dạng bài toán lập lịch.

Giải thuật

Dữ liệu được lưu trữ vào vector \mathbf{x} , mỗi tọa độ được lưu dưới dạng nhóm 3 số

(tọa độ, dấu hiệu cuối đoạn, số thứ tự của đoạn)

a_i hoặc b_i

True nếu tọa độ là b_i

i

Dùng mảng $f1g[]$ để đánh dấu, $f1g[i] = true$ nếu đã có điểm đại diện và bằng $false$ trong trường hợp ngược lại,

Sắp xếp \mathbf{x} theo thứ tự tăng dần,

Duyệt \mathbf{x} từ đầu đến cuối:

- Gặp điểm đầu: nạp số thứ tự vào vector \mathbf{v} ,

- Gặp điểm cuối: kiểm tra **flg** tương ứng, nếu đoạn tương ứng chưa có đại diện thì tăng số lượng đại diện lên 1, đánh dấu có đại diện cho tất cả các đoạn có số thứ tự lưu trữ trong **v** và xóa **v**.

Độ phức tạp của giải thuật: $O(n \log n)$.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "covering."
using namespace std;
typedef tuple<int,bool,int> tibi;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n,ans=0,m,a,b,k,pv=0;
bool c;

int main()
{
    fi>>n;
    vector<tibi> x(2*n);
    vector<bool> flg(n,0);
    for(int i=0;i<n;++i)
    {
        fi>>a>>b;
        x[2*i]= make_tuple(a,0,i);
        x[2*i+1] = make_tuple(b,1,i);
    }
    sort(x.begin(),x.end());
    vector<int> v(n);
    for(int i=0;i<2*n;++i)
    {
        tie(a,c,m)=x[i];
        if(c==0) v[pv++]=m;
        if(c && (flg[m]==0))
        {
            for(int j=0;j<pv;++j) flg[v[j]]=true;
            flg[m]=true; ++ans; pv=0;
        }
    }
    fo<<ans;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Bài toán 2

Cho n đoạn thẳng trên đường thẳng, đoạn thứ i có tọa độ các điểm đầu và cuối là a_i, b_i . Mỗi đoạn thẳng thuộc một trong 2 loại 0 hoặc 1. Hãy xác định số lượng ít nhất các điểm cần chọn để mỗi đoạn thẳng loại 1 đã cho chứa ít nhất một điểm trong số đã chọn và không có điểm chọn nào thuộc đoạn thẳng loại 0, chỉ ra một phương án tọa độ các điểm được chọn.

Dữ liệu: Vào từ file văn bản COVR_2.INP:

- ✚ Dòng đầu tiên chứa một số nguyên n ($1 < n \leq 10^5$),
- ✚ Dòng thứ i trong n dòng sau chứa 3 số nguyên a_i, b_i và t_i ($-10^9 \leq a_i \leq b_i \leq 10^9, 0 \leq t_i \leq 1$).

Kết quả: Đưa ra file văn bản COVR_2.OUT, dòng thứ nhất chứa một số nguyên – số điểm ít nhất cần chọn, dòng thứ hai – tọa độ các điểm được chọn.

Ví dụ:

| COVR_2.INP |
|------------|
| 7 |
| 1 5 1 |
| 3 8 0 |
| 30 32 0 |
| 25 35 1 |
| 33 33 0 |
| 10 20 1 |
| 14 16 1 |

| COVR_2.OUT |
|------------|
| 3 |
| 2 16 35 |

Giải thuật

Bài toán *có thể vô nghiệm* vì có các đoạn không được phép lấy đại diện,

Việc chọn điểm đại diện *không có định ở điểm cuối* của khoảng (vì có thể thuộc vùng cấm),

Cần phân biệt 2 loại đánh giá:

- ✚ Độ phức tạp của *giải thuật*,
- ✚ Độ phức tạp *lập trình*.

Độ phức tạp của giải thuật:

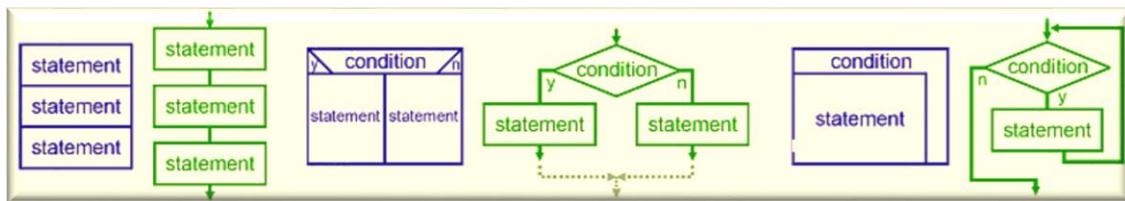
- Mối quan hệ giữa *thời gian thực hiện* chương trình với các tham số của bài toán,
- Sản phẩm lập trình được *nghiệm thu theo độ phức tạp của giải thuật*,
- Đây là *tiêu chuẩn định lượng*.

Độ phức tạp lập trình:

- Chương trình có cấu trúc tuyến tính là đơn giản nhất, dễ hiểu, dễ hiệu chỉnh và bảo trì (nâng cấp, cải tiến giải thuật, sửa đổi và mở rộng chương trình khi yêu cầu xử lý và cấu trúc dữ liệu vào thay đổi),
- Chương trình có độ phức tạp lập trình thấp khi nó có cấu trúc gần với tuyến tính nhất,
- Thời gian lập trình nhỏ nhất,
- Đây là **tiêu chuẩn định tính**.

Lý thuyết Lập trình cấu trúc (*Structured Programming*) do Niklaus Wirth, Dijkstra, Robert W. Floyd, Tony Hoare, Ole-Johan Dahl và David Gries đề xuất được truyền bá rộng rãi trên thế giới và những năm của thập kỷ 70 – 80 của thế kỷ XX với tham vọng biến lập trình từ nghệ thuật (*Art*) thành khoa học (*Discipline*).

Lập trình cấu trúc dựa trên 3 loại cấu trúc: tuyến tính, rẽ nhánh và chu trình, trong đó rẽ nhánh có thể là loại rẽ 2 nhánh (**if**) hoặc nhiều nhánh (**case/switch**), cấu trúc chu trình có thể là loại có số lần lặp biết trước (**for**) hoặc không biết trước (**while, repeat-until/do-while**).



Tuy vậy, dần dần người ta cũng nhận thấy lập trình vẫn mang trong mình **yếu tố nghệ thuật: nghệ thuật tổ chức dữ liệu** và **nghệ thuật thể hiện giải thuật**.

Ở bài toán đang xét có hai loại đoạn cho trước: loại bị cấm chọn điểm đại diện và loại cần chọn điểm đại diện. Hai loại đoạn này tác động lên việc tìm lời giải theo các nguyên tắc khác nhau: một điểm đã bị cấm thì bị cấm một lần hay k lần cũng như nhau. Nhưng điểm đại diện phải có mặt trong từng đoạn đã cho.

Chương trình sẽ đơn giản hơn nếu:

- ✓ Chọn đơn vị xử lý: điểm và các thuộc tính của nó,
- ✓ Phân loại và xử lý sơ bộ các đoạn cấm để đơn giản hóa yêu cầu cấm chọn,
- ✓ Lưu trữ dữ liệu xử lý theo cùng một quy cách.

Dữ liệu về các đoạn cần chọn được lưu trữ dưới dạng nhóm 4 số:



Thuộc tính điểm:

- 0 – điểm đầu,
- 1 – điểm cuối đoạn cần chọn điểm đại diện,
- 2 – điểm cuối đoạn bị cấm.

Thuộc tính đoạn: 0 hoặc 1 như đã cho. Thuộc tính có thể lưu trữ kiểu **bool**.

Số thứ tự của đoạn:

Với đoạn cần chọn: lưu đúng số thứ tự của nó trong dữ liệu vào,

Với các đoạn cấm chọn: không cần phân biệt, vì vậy có chứa giá trị bất kỳ!

Với cách lưu trữ dữ liệu như vậy, khi sắp xếp lại, nếu có các điểm *trùng tọa độ* thì điểm đầu của đoạn cấm sẽ đứng trái nhất, sau đó là các điểm đầu của những đoạn cần chọn, tiếp sau – các điểm cuối của những đoạn cần chọn và cuối cùng – điểm cuối đoạn cấm.

Xử lý sơ bộ: tổng hợp thông tin về các đoạn cấm. Đây là vấn đề đơn giản, có nhiều cách xử lý tương đương với độ phức tạp giải thuật O(n). Dưới đây là một trong số các cách đó, phục vụ việc *rèn luyện kỹ năng tổng quát* xử lý các vấn đề về đoạn thẳng.

Thông tin về các đoạn cấm được lưu trữ dưới dạng cặp dữ liệu (*tọa độ*, *thuộc tính*). Thuộc tính nhận giá trị 0 nếu là điểm đầu và 1 – cho điểm cuối.

Dữ liệu được sắp xếp theo thứ tự tăng dần. Như vậy, nếu trùng tọa độ thì các điểm đầu sẽ đứng trước các điểm cuối.

Dùng biến **cnt** tính số lượng đoạn cấm để lên một điểm. Bắt đầu từ **cnt = 0**, khi gặp điểm đầu – tăng cnt, gặp điểm cuối – giảm **cnt**. Khi **cnt** trở lại 0 – ta có điểm cuối của đoạn hợp nhất của một số đoạn cấm.

Lưu ý: Cần hợp nhất 2 đoạn cấm đi liên tiếp nhau, tức là nếu có các đoạn cấm **[a, b]** và **[b+1, c]** thì kết quả hợp nhất sẽ là **[a, c]**.

Sau khi hợp nhất các đoạn cấm, thông tin tổng hợp sẽ được ghi nhận vào mảng chứa thông tin về các đoạn cần chọn đại diện theo quy cách đã nêu.

Tại đây, quá trình nhập, ghi nhận và chuẩn bị dữ liệu kết thúc.

Tìm các điểm đại diện: tìm điểm tự do cuối cùng phải nhất. Khi gặp điểm cuối đoạn cần chọn, ghi nhận điểm tự do cuối cùng tìm được làm điểm đại diện và đánh dấu tất cả các đoạn đã có đại diện. Nếu gặp điểm cuối một đoạn mà điểm tự do nhỏ hơn điểm đầu – bài toán vô nghiệm.

Tổ chức dữ liệu

- Mảng **flg[]** để đánh dấu, **flg[i] = true** nếu đã có điểm đại diện và bằng **false** trong trường hợp ngược lại,
- Mảng **y[]** kiểu **pair<int, bool>** lưu thông tin về các đoạn cấm,
- Mảng **x[]** kiểu **tuple<int, int, bool, int>** lưu thông tin các điểm,
- Biến **int free_p** lưu tọa độ điểm tự do cuối cùng,
- Mảng **res[]** kiểu **int** lưu tọa độ các điểm được chọn,
- Mảng **vt[]** lưu số hiệu đoạn của các đoạn chắc chắn đã có đại diện nhưng chưa được ghi nhận,
- Cấu trúc **map<int, int> mp** lưu điểm đầu các đoạn đã gấp và chưa có đại diện.

Xử lý

Đọc và phân loại dữ liệu:

```
fi>>n;
vector<tiibi> x;
vector<pib>y;
vector<bool> flg(n, 0);
for (int i=0; i<n; ++i)
{
    fi>>a>>b>>t;
    if (t==0) {y.push_back({a, 0}); y.push_back({b, 1});}
    else {x.push_back(make_tuple(a, 0, t, i));
          x.push_back(make_tuple(b, 1, t, i));}
}
```

Hợp nhất các đoạn bị cấm:

Chỉ xử lý khi trong dữ liệu có đoạn cấm,

- Sắp xếp theo tọa độ tăng dần,
- Thông tin về đoạn hợp nhất phụ thuộc vào điểm đầu của đoạn tiếp theo vì vậy có thể dùng kỹ thuật hàng rào để tiện xử lý: thêm một điểm đầu với tọa độ vô cực,

- Thông tin về đoạn cấm được ghi nhận với số thứ tự đoạn là $n+1$.

```

if (!y.empty())
{
    sort(y.begin(), y.end());
    t0=y.back(); k=get<0>(t0); y.push_back({k+INF,0});
    int u=INF, v;

    for (auto &t0:y)
    {
        a=t0.first; c=t0.second;
        if (!c)
        {
            ++cnt;
            if (cnt==1)
            {
                if (i0==0) {u=a; i0=1; continue;}
                if (v+1==a) continue;
                x.push_back(make_tuple(u,0,0,n+1));
                x.push_back(make_tuple(v,2,0,n+1));
                u=a;
                continue;
            }
            if (c)--cnt, v=a;
        }
    }
}

```

Tìm điểm đại diện:

Xác định điểm tự do **free_p** phải nhất:

$$\text{free_p} = \begin{cases} \mathbf{a}, & \text{nếu } \mathbf{a} \text{ là điểm đầu của đoạn loại 1 và không bị phủ bởi đoạn loại 0,} \\ & \text{bởi đoạn loại 0,} \\ \mathbf{a}-1, & \text{nếu } \mathbf{a} \text{ là điểm đầu của đoạn loại} \\ & \text{a+1, nếu } \mathbf{a} \text{ là điểm cuối của đoạn loại 0,} \end{cases}$$

Xử lý điểm đầu đoạn loại 1:

- ❖ **a** – điểm đầu đoạn số thứ tự m ,
- ❖ Trường hợp không bị phủ: nạp m vào vector **vt**, ghi nhận **free_p**,
- ❖ Trường hợp bị phủ: nạp **a** vào **mp**: $\text{mp}[m]=a$;

Xử lý điểm cuối đoạn loại 1:

- ♣ **a** – điểm cuối đoạn số thứ tự m ,

- Trường hợp vô nghiệm: đoạn **m** chưa được đánh dấu, điểm cuối bị phủ (**cnt==1**) và **free_p < mp[m]**,
- Nếu không vô nghiêm: chỉnh lý **free_p** khi cần thiết, nạp **free_p** vào mảng kết quả **res**, nạp **m** vào **vt** (nếu cần), đánh dấu đã có đại diện các đoạn lưu trong **vt** và xóa rỗng **vt**.

Đưa ra kết quả:

- Số lượng điểm: kích thước của **res**,
- Các điểm đại diện lưu trong **res**.

```

cnt=0;
for(auto & z:x)
{
    tie(a,c,t,m)=z;
    if(c==0)
    {
        if(t==0) {cnt=1; free_p=a-1; continue;}
        if(t==1 && cnt==0) {free_p=a; vt.push_back(m); continue;}
        if(t==1 && cnt==1 && flg[m]==0) {mp[m]=a; continue;}
    }
    else
    {
        if(c==2) {cnt=0; free_p =a+1; continue;}
        if(cnt==1 && flg[m]==0)
            if(free_p<mp[m]) {fo<<'0'; return 0;}
        if(cnt==0 && flg[m]==0) {free_p=a;
                                    flg[m]=1; res.push_back(free_p);}
        if(flg[m]==0) res.push_back(free_p); vt.push_back(m);
        for(auto & iv:vt) flg[iv]=1; vt.clear();
    }
}

```

Độ phức tạp của giải thuật: $O(n \log n)$.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "covr_2."
using namespace std;
typedef tuple<int,int,bool,int> tiibi;
typedef pair<int,bool> pib;
typedef pair<int, int> pii;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int INF = 2000000000;
int n,ans=0,cnt=0,a,b,c,m,k,free_p =INF,i0=0,i1=0;
bool t;
pib t0;
pii tq;

int main()
{
    fi>>n;
    vector<tiibi> x;
    vector<pib>y;
    vector<bool> flg(n,0);
    for(int i=0;i<n;++i)
    {
        fi>>a>>b>>t;
        if(t==0) {y.push_back({a,0}); y.push_back({b,1});}
        else {x.push_back(make_tuple(a,0,t,i));
              x.push_back(make_tuple(b,1,t,i));}
    }
    if(!y.empty())
    {
        sort(y.begin(),y.end());
        t0=y.back(); k=get<0>(t0); y.push_back({k+INF,0});
        int u=INF,v;

        for(auto &t0:y)
        {
            a=t0.first; c=t0.second;
            if(!c)
            {
                ++cnt;
                if(cnt==1)
                {
                    if(i0==0){u=a; i0=1; continue;}
                    if(v+1==a)continue;
                    x.push_back(make_tuple(u,0,0,n+1));
                    x.push_back(make_tuple(v,2,0,n+1));
                    u=a;
                    continue;
                }
                if(c)--cnt, v=a;
            }
        }
        sort(x.begin(),x.end());
    }
    vector<int> vt;
    vector<int>res;
```

```

map<int,int> mp;
cnt=0;
for(auto & z:x)
{
    tie(a,c,t,m)=z;
    if(c==0)
    {
        if(t==0) {cnt=1;free_p=a-1; continue;}
        if(t==1 && cnt==0){free_p=a;vt.push_back(m); continue;}
        if(t==1 && cnt==1 && flg[m]==0){mp[m]=a; continue;}
    }
    else
    {
        if(c==2){cnt=0; free_p =a+1; continue;}
        if(cnt==1 && flg[m]==0)
            if(free_p<mp[m]) {fo<<'0';return 0;}
        if(cnt==0 && flg[m]==0){free_p=a;
                                    flg[m]=1;res.push_back(free_p);}
        if(flg[m]==0)res.push_back(free_p); vt.push_back(m);
        for(auto & iv:vt) flg[iv]=1; vt.clear();
    }
}
fo<<res.size()<<'\n';
for(int i:res) fo<<i<<' ';
fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}

```



Bao lồi

Bài toán

Cho n điểm trên mặt phẳng, điểm thứ i có tọa độ $(\mathbf{x}_i, \mathbf{y}_i)$. Hãy xác định bao lồi (hay nói cách khác – xác định đa giác lồi nhỏ nhất) chứa các điểm đã cho.

Dữ liệu: Vào từ file văn bản CONVEX.INP:

- ⊕ Dòng đầu tiên chứa một số nguyên n ($3 \leq n \leq 10^5$),
- ⊕ Dòng thứ i trong n dòng sau chứa 2 số thực \mathbf{x}_i và \mathbf{y}_i ($0 \leq |\mathbf{x}_i|, |\mathbf{y}_i| \leq 10^9$).

Kết quả: Đưa ra file văn bản CONVEX.OUT, dòng đầu tiên chứa số nguyên m – số đỉnh của bao lồi, dòng thứ j trong m dòng sau chứa 2 số thực \mathbf{u}_j và \mathbf{v}_j xác định đỉnh của bao lồi với độ chính xác 6 chữ số sau dấu chấm thập phân. Các đỉnh liệt kê theo chiều kim đồng hồ.

Ví dụ:

| CONVEX.INP |
|------------|
| 9 |
| 3 3 |
| 7 2 |
| 5 3 |
| 4 0 |
| 3 5 |
| 5 2 |
| 1 1 |
| 6 5 |
| 5 1 |

| CONVEX.OUT |
|-------------------|
| 5 |
| 1.000000 1.000000 |
| 3.000000 5.000000 |
| 6.000000 5.000000 |
| 7.000000 2.000000 |
| 4.000000 0.000000 |

Giải thuật Graham

Giải thuật tìm bao lồi được Graham công bố năm 1972 và được Andrew cải tiến năm 1979. Bao lồi được tìm với độ phức tạp $O(n \log n)$ và chỉ cần dùng các phép cộng, trừ, nhân và so sánh.

Đây là giải thuật tìm bao lồi có độ phức tạp tốt nhất. Tuy nhiên giải thuật này chỉ thích hợp với việc xử lý dữ liệu offline.

Tọa độ các điểm được lưu trữ dưới dạng mảng cặp tọa độ. Mảng được sắp xếp theo trình tự tăng dần. Gọi **P0** và **P1** tương ứng là điểm đầu và cuối trong mảng đã sắp xếp. **P0** sẽ là điểm có tọa độ \mathbf{x} nhỏ nhất và có tọa độ \mathbf{y} nhỏ nhất trong số các điểm có cùng tọa độ \mathbf{x} nhỏ nhất. **P1** sẽ là điểm có tọa độ \mathbf{x} lớn nhất và có tọa độ \mathbf{y} lớn nhất trong số các điểm có cùng tọa độ \mathbf{x} lớn nhất. Rõ ràng bao lồi cần tìm phải chứa các điểm **P0** và **P1**.

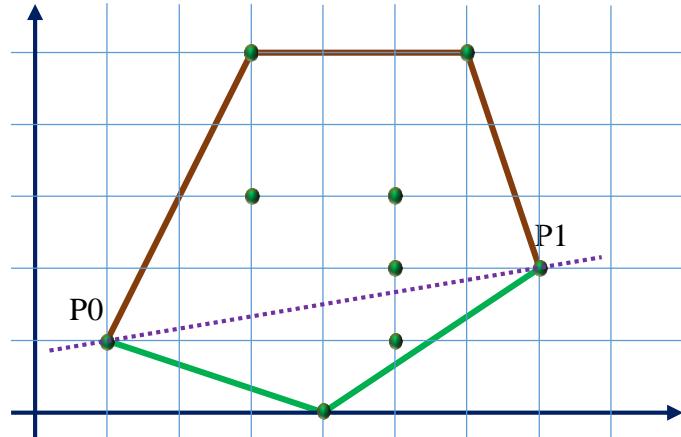
Đường thẳng d nối 2 điểm P_0 và P_1 sẽ chia các điểm đã cho thành 2 tập S_{up} và S_{down} . Tập S_{up} chứa các điểm nằm bên trái đường thẳng d , tập S_{down} chứa các điểm nằm bên phải đường thẳng d . Các điểm P_0 và P_1 tham gia vào cả 2 tập. Các điểm nằm trên đường thẳng có thể cho vào tập bất kỳ, dù sao chúng cũng không phải là những điểm tạo nên bao lồi.

Xây dựng phần trên của bao lồi từ các điểm thuộc S_{up} và phần dưới của bao lồi từ các điểm thuộc S_{down} .

Xét việc xây dựng phần trên của bao lồi (phần dưới – xây dựng tương tự).

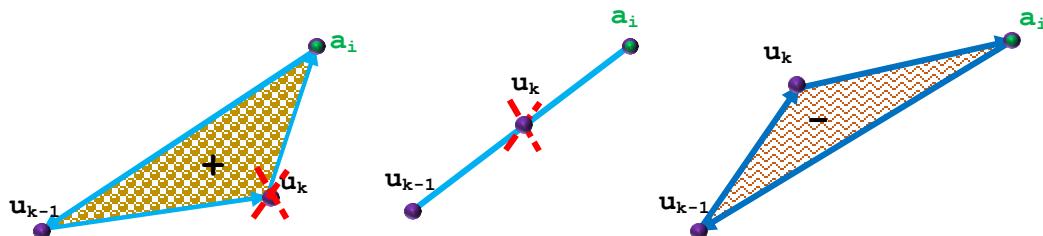
Từ tập up rỗng kết nạp P_0 và sau đó P_1 vào up .

Lần lượt duyệt các điểm còn lại của S_{up} theo trình tự đã sắp xếp chung.



Với mỗi điểm mới tính diện tích có dấu của tam giác có các đỉnh là điểm gần cuối cùng u_{pk-1} , điểm cuối cùng u_{pk} trong up và điểm mới theo đúng trình tự đã liệt kê. Nếu diện tích là dương – điểm mới nằm bên trái đường nối u_{pk-1} và u_{pk} , như vậy điểm u_{pk} sẽ không tham gia vào bao lồi và sẽ bị loại. Nếu diện tích bằng 0 – ba điểm nằm trên đường thẳng nhưng điểm u_{pk} nằm ở giữa và cũng sẽ bị loại.

Cuối cùng bổ sung điểm mới vào cuối up .



Sau khi xử lý tất cả các điểm trong S_{up} ta có phần trên của bao lồi. Xử lý tương tự - có phần dưới bao lồi. Kết quả cần tìm sẽ là sự liên kết 2 phần nhận được.

Lưu ý:

Ta không cần xây dựng tường minh các tập S_{up} và S_{down} ,

Gọi S là diện tích có dấu của tam giác ABC theo trình tự liệt kê A → B → C, có

$$2 \times S = x_a \times (y_b - y_c) + x_b \times (y_c - y_a) + x_c \times (y_a - y_b)$$

Độ phức tạp của giải thuật: $O(n \log n)$.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "convex."
using namespace std;
typedef pair<double,double> pdd;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n;

double area (pdd a,pdd b,pdd c)
{
    return (a.first*(b.second-c.second) +
            b.first*(c.second-a.second) +
            c.first*(a.second-b.second))
        ;
}

int main()
{
    fi>>n;
    vector<pdd> a(n);
    vector<pdd> up, down;
    for(int i=0;i<n;++i) fi>>a[i].first>>a[i].second;
    sort(a.begin(),a.end());
    pdd p1=a[0], p2=a[n-1];
    up.push_back(p1); down.push_back(p1);
    for(int i=1;i<a.size();++i)
    {
        if(i==a.size()-1 || area(p1,a[i],p2)<0)
        {
            while(up.size()>=2 &&
                  area(up[up.size()-2],up[up.size()-1],a[i])>=0) up.pop_back();
            up.push_back(a[i]);
        }
        if(i==a.size()-1 || area(p1,a[i],p2)>0)
        {
            while(down.size()>=2 &&
                  area(down[down.size()-2],down[down.size()-1],a[i])<0)
                down.pop_back();
            down.push_back(a[i]);
        }
    }
    a.clear();
    for(int i=0;i<up.size();++i)a.push_back(up[i]);
    for(int i=down.size()-2;i>0;--i)a.push_back(down[i]);

    fo<<a.size()<<'\n';
    for(int i=0;i<a.size();++i) fo<<fixed<<a[i].first<<' '<<a[i].second<<'\n';

    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Kiểm tra điểm trong

Bài toán

Cho đa giác lồi n đỉnh, đỉnh thứ i có tọa độ nguyên (a_i, b_i) , $i = 1 \div n$. Các đỉnh được liệt kê theo trình tự ngược kim đồng hồ.

Với mỗi điểm trong số m điểm cho trước, điểm thứ j có tọa độ nguyên (x_j, y_j) , $j = 1 \div m$, hãy xác định điểm nằm trong hay ngoài đa giác lồi đã cho và đưa ra thông báo tương ứng “**Inside**” hoặc “**Outside**”. Một điểm được gọi là nằm trong đa giác nếu nó nằm hoàn toàn trong miền trong của đa giác hay nằm trên cạnh hoặc đỉnh của đa giác.

Dữ liệu: Vào từ file văn bản INSIDE.INP:

- ✚ Dòng đầu tiên chứa một số nguyên n ($3 \leq n \leq 10^5$),
- ✚ Dòng thứ thứ i trong n dòng tiếp theo chứa 2 số nguyên a_i và b_i ,
- ✚ Dòng tiếp theo chứa số nguyên m ,
- ✚ Dòng thứ thứ j trong m dòng tiếp theo chứa 2 số nguyên x_j và y_j .

Các tọa độ đều nguyên và có giá trị tuyệt đối không vượt quá 10^9 .

Kết quả: Đưa ra file văn bản INSIDE.OUT các thông báo tìm được, mỗi thông báo đưa ra trên một dòng.

Ví dụ:

| INSIDE.INP | INSIDE OUT |
|------------|------------|
| 5 | Inside |
| 6 5 | Inside |
| 3 5 | Outside |
| 1 1 | Inside |
| 4 0 | Inside |
| 7 2 | Outside |
| 7 | Inside |
| 4 2 | Inside |
| 2 3 | Inside |
| 1 3 | Inside |
| 4 5 | Inside |
| 2 2 | Inside |
| 11 8 | Inside |
| 5 2 | Inside |

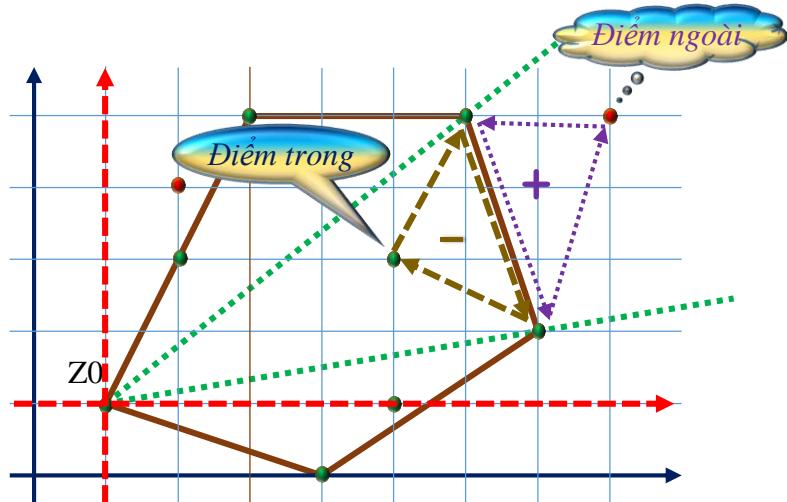
Giải thuật

Việc xác định một điểm có nằm trong đa giác lồi đã cho hay không được thực hiện bằng phương pháp tìm kiếm nhị phân theo góc.

Tìm đỉnh z_0 trái nhất và thấp nhất của đa giác, đánh số lại các đỉnh đa giác bắt đầu từ z_0 ,

Tịnh tiến trực tọa độ, đưa z_0 về gốc tọa độ,

Như vậy các đỉnh đa giác đều nằm ở nửa phải của mặt phẳng tọa độ,



Các góc giữa tia nối đỉnh đa giác với gốc tọa độ và trục Ox được gọi là góc tọa độ cực, *tăng dần* khi di chuyển từ một đỉnh sang đỉnh khác theo chiều ngược kim đồng hồ,

Các góc có giá trị trong đoạn $[-\frac{\pi}{2}, \frac{\pi}{2}]$, có thể tính bằng hàm **atan2**, tuy nhiên hàm này cho giá trị thực và việc tích lũy sai số làm tròn có thể dẫn đến kết quả sai lệch.

Ở đây ta không cần bắn thân giá trị góc mà chỉ cần so sánh 2 góc, vì vậy có thể hoàn toàn xác định được quan hệ so sánh góc với các tọa độ nguyên: với 2 điểm tọa độ (x_1, y_1) và (x_2, y_2) cùng ở nửa phải mặt phẳng tọa độ tang của góc tọa độ cực tương ứng sẽ là y_1/x_1 và y_2/x_2 , như vậy, chỉ cần *xử lý 2 phân số* này là có được quan hệ so sánh.

Với điểm **A** tọa độ (x, y) , bằng phương pháp tìm kiếm nhị phân, ta có thể xác định hai đỉnh liên tiếp nhau **P1** và **P2** của đa giác, **P1** có góc tọa độ cực nhỏ hơn hoặc bằng góc tọa độ cực của **A**, **P2** có góc tọa độ cực lớn hơn góc tọa độ cực của **A**.

Xét tam giác với các đỉnh **P1**, **P2**, **A** theo đúng trình tự đỉnh đã liệt kê ta có thể dễ dàng xác định **A** nằm bên trái hay phải của đoạn **P1** → **P2**. Nếu **A** nằm bên trái hoặc trên cạnh – **A** thuộc đa giác, trong trường hợp ngược lại – ngoài đa giác. Điều này có thể thực hiện bằng cách tính diện tích có dấu của tam giác.

Việc xác định một đỉnh có nằm trong đa giác hay không có độ phức tạp $O(\log n)$. Các công việc chuẩn bị chung cho mọi lần tìm kiếm có độ phức tạp $O(n)$.

Xử lý

Nhập dữ liệu về đa giác và đánh số lại:

The diagram shows a code snippet for finding the index of vertex z_0 and rotating the polygon. A callout bubble labeled "Tim vị trí z_0 " points to the code. Another callout bubble labeled "Dãy vòng tròn, đưa z_0 về đầu" points to the rotation part of the code. A third callout bubble labeled " z_0 không tham gia tính góc" points to the vertex z_0 itself.

```
fi>>n;
vector<pll>p(n); iz0=0;
for(int i=0;i<n;++i)
{
    fi>>p[i].first>>p[i].second;
    if(p[i]<p[iz0]) iz0=i;
}
pll z0=p[iz0];
rotate(p.begin(), p.begin() + iz0, p.end());
p.erase(p.begin()); --n;
```

Tính tiền tọa độ và xác định tham số tang các góc:

The diagram shows a code snippet for calculating intermediate coordinates and determining the sign of angles. A callout bubble labeled "Trường hợp điểm trên trực Oy" points to the angle calculation part.

```
vector<pll> a(n);
for(int i=0;i<n;++i)
{
    a[i].first = p[i].second - z0.second;
    a[i].second = p[i].first - z0.first;
    if(a[i].first==0)
        a[i].second=(a[i].second<0) ? -1 : 1;
}
```

Xử lý truy vấn:

The diagram shows a code snippet for point query processing. It includes parts for calculating intermediate coordinates, determining the sign of angles, and performing intersection detection. Callout bubbles include "Tính tiền và tính góc", "Tiêu chuẩn tìm kiếm nhị phân", "Xác định P1", and "Kiểm tra điểm trong".

```
fi>>x>>y; pll q={x,y};
bool in=false;
if(q==z0) in=true;
else
{
    pll mq={y-z0.second, x-z0.first};
    if(mq.first==0) mq.second=(mq.second<0) ? -1 : 1;

    auto it=upper_bound(a.begin(), a.end(), mq,
                         [&](pll a, pll b)
                         { return (a.first*b.second <= a.second*b.first); });

    if(it==a.end() && mq==a[n-1]) it=a.end()-1;
    if(it!=a.end() && it!=a.begin())
    {
        int p1=int(it-a.begin());
        in=(area(p[p1], p[p1-1], q)<=0);
    }
}
fo<<((in) ? "Inside\n" : "Outside\n");
```

Chương trình

```
#include <bits/stdc++.h>
#define NAME "inside."
using namespace std;
typedef int64_t ll;
typedef pair<ll, ll> pll;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n, m, iz0;
ll x, y;
ll area(pll a, pll b, pll c)
{
    return (a.first*(b.second-c.second) +
            b.first*(c.second-a.second) +
            c.first*(a.second-b.second));
}
int main()
{
    fi>>n;
    vector<pll>p(n); iz0=0;
    for(int i=0;i<n;++i)
    {
        fi>>p[i].first>>p[i].second;
        if(p[i]<p[iz0]) iz0=i;
    }
    pll z0=p[iz0];
    rotate(p.begin(), p.begin() + iz0, p.end());
    p.erase(p.begin()); --n;
    vector<pll> a(n);
    for(int i=0;i<n;++i)
    {
        a[i].first = p[i].second - z0.second;
        a[i].second = p[i].first - z0.first;
        if(a[i].first==0) a[i].second=(a[i].second<0)? -1: 1;
    }
    fi>>m;
    for(int i=0;i<m;++i)
    {
        fi>>x>>y; pll q={x, y}; bool in=false;
        if(q==z0) in=true;
        else
        {
            pll mq={y-z0.second, x-z0.first};
            if(mq.first==0) mq.second=(mq.second<0)? -1: 1;
            auto it=upper_bound(a.begin(), a.end(), mq,
                [&](pll a, pll b)
                { return (a.first*b.second <= a.second*b.first); });
            if(it==a.end() && mq==a[n-1]) it=a.end()-1;
            if(it!=a.end() && it!=a.begin())
            {
                int p1=int(it-a.begin());
                in=(area(p[p1], p[p1-1], q)<=0);
            }
        }
        fo<<((in)? "Inside\n" : "Outside\n");
    }
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Tìm cặp điểm gần nhất

Bài toán

Cho n điểm trên mặt phẳng, điểm thứ i có tọa độ (x_i, y_i) , $i = 1 \div n$. Khoảng cách d giữa 2 điểm i và j được tính theo công thức $d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ (*Khoảng cách Euclide*).

Hãy xác định khoảng cách ngắn nhất giữa 2 điểm trong số các điểm đã cho và chỉ ra một cặp điểm có khoảng cách ngắn nhất.

Dữ liệu: Vào từ file văn bản DISTANCE.INP:

- ➡ Dòng đầu tiên chứa một số nguyên n ($2 \leq n \leq 10^5$),
- ➡ Dòng thứ i trong n dòng sau chứa 2 số nguyên x_i và y_i ($|x_i|, |y_i| \leq 10^9$).

Kết quả: Đưa ra file văn bản DISTANCE.OUT, dòng đầu tiên là khoảng cách ngắn nhất tìm được với độ chính xác 6 chữ số sau dấu chấm thập phân, dòng thứ 2 chứa 2 số nguyên là số thứ tự của một cặp điểm có khoảng cách nhỏ nhất.

Ví dụ:

| DISTANCE.INP | DISTANCE .OUT |
|--------------|---------------|
| 8 | 2.000000 |
| 1 1 | 3 7 |
| 4 0 | |
| 5 2 | |
| 3 5 | |
| 8 5 | |
| 0 5 | |
| 3 2 | |
| 8 2 | |

Giải thuật

Phương pháp giải: Chia và trị,

Kỹ thuật lập trình: đệ quy.

Sơ đồ xử lý là chia tập các điểm thành 2 phần, giải bài toán tương tự ở mỗi phần,

Xác định một dãi đệm kết nối 2 phần, giải bài toán tương tự ở lớp đệm và tổng hợp kết quả từ các kết quả nhận được ở ba phần đã nêu.

Việc phân chia tập ban đầu thành 2 phần không quá khó khăn:

- ➡ Sắp xếp các điểm đã cho theo trình tự tăng dần của x,
- ➡ Lấy các điểm ở nửa đầu của dãy cho vào tập **A1**, phần còn lại – vào tập **A2**.

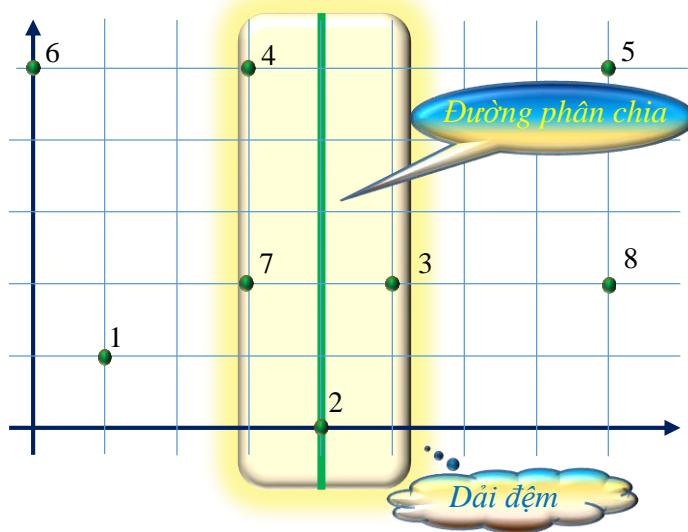
Giả thiết ta đã giải được bài toán ở các tập **A1** và **A2** với khoảng cách nhỏ nhất tìm được tương ứng là **d1** và **d2**.

Gọi $d = \min\{d_1, d_2\}$.

Rõ ràng tập các điểm \mathbf{B} ở dải đệm chỉ chứa các điểm thỏa mãn điều kiện khoảng cách tới đường phân chia nhỏ hơn d

$$|\mathbf{x}_v - \mathbf{x}_B| < d,$$

trong đó \mathbf{x}_B – tọa độ xác định đường phân chia.



Tập \mathbf{C} các điểm cần xét trong dải đệm còn hẹp hơn, với mỗi điểm \mathbf{p}_i trong \mathbf{B} chỉ cần xét các điểm có chênh lệch theo y tới \mathbf{p}_i không quá d . Gọi tập điểm cần xét với điểm \mathbf{p}_i là $\mathbf{C}(\mathbf{p}_i)$. $\mathbf{C}(\mathbf{p}_i)$ dễ dàng xác định được nếu các điểm trong \mathbf{B} được sắp xếp theo giá trị tọa độ y .

Xử lý đệ quy:

Hàm **rec(int l, int r)** tìm khoảng cách nhỏ nhất trong số các điểm từ **l** đến **r** của dãy đã sắp xếp:

- Nếu **r-l** là đủ bé – tiến hành vét cạn các cặp điểm, sắp xếp dữ liệu trong khoảng theo y,
- Trong trường hợp ngược lại:
 - Chia đôi khoảng và lần lượt gọi rec với các khoảng nhận được,
 - Hợp nhất khoảng và sắp xếp theo y,
 - Xử lý dãy đệm.

Lưu ý: dùng mảng dữ liệu trung gian kiểu static để tránh việc phải phân nhiều lần, điều có thể dẫn đến hiện tượng thiếu bộ nhớ làm việc.

Độ phức tạp của giải thuật: $O(n \log n)$.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "distance."
using namespace std;
typedef pair<double, double> pdd;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int MAXN=100001;
int n;
struct pt {int x, y, id; } a[MAXN];
inline bool cmp_x (const pt & a, const pt & b)
{ return a.x < b.x || a.x == b.x && a.y < b.y; }
inline bool cmp_y (const pt & a, const pt & b) { return a.y < b.y; }
double mindist; int ansa, ansb;
inline void upd_ans (const pt & a, const pt & b)
{double dist = sqrt ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y) + .0);
 if (dist < mindist) mindist = dist, ansa = a.id, ansb = b.id; }

void rec (int l, int r)
{if (r - l <= 3)
{
    for (int i=l; i<=r; ++i)
    for (int j=i+1; j<=r; ++j)
    upd_ans (a[i], a[j]);
    sort (a+l, a+r+1, &cmp_y);
    return;
}
int m = (l + r) >> 1;
int midx = a[m].x;
rec (l, m), rec (m+1, r);
static pt t[MAXN];
merge (a+l, a+m+1, a+m+1, a+r+1, t, &cmp_y);
copy (t, t+r-l+1, a+l);
int tsz = 0;
for (int i=l; i<=r; ++i)
if (abs (a[i].x - midx) < mindist)
{
    for (int j=tsz-1; j>=0 && a[i].y - t[j].y < mindist; --j)
        upd_ans (a[i], t[j]);
    t[tsz++] = a[i];
}
}

int main()
{
    fi>>n;
    for(int i=0; i<n; ++i)
    {
        fi>>a[i].x>>a[i].y; a[i].id=i;
    }
    sort(a, a+n, &cmp_x);
    mindist=1e19;
    rec(0, n-1);
    fo<<fixed<<setprecision(6)<<mindist<<'\n'<<ansa+1<< ' '<<ansb+1;
    fo<<"\nTime: "<<clock () / (double) 1000<<" sec";
}
```



Đường thẳng quét

Bài toán

Cho n đoạn thẳng trên mặt phẳng, đoạn thứ i được xác định bởi điểm đầu tọa độ (x_i, y_i) và điểm cuối tọa độ (u_i, v_i) , $i = 1 \dots n$.

Hãy xác định có tồn tại một cặp đoạn thẳng nào đó giao nhau hay không và đưa ra số thứ tự của các đoạn đó. Nếu không tồn tại cặp đoạn thẳng giao nhau thì đưa ra 2 số -1 và -1 . Hai đoạn thẳng được gọi là giao nhau nếu có ít nhất một điểm chung.

Dữ liệu: Vào từ file văn bản INTERSECT.INP:

- ✚ Dòng đầu tiên chứa một số nguyên n ($2 \leq n \leq 10^5$),
- ✚ Dòng thứ i trong n dòng sau chứa 4 số nguyên x_i, y_i, u_i, v_i , các tọa độ có giá trị tuyệt đối không vượt quá 10^9 .

Kết quả: Đưa ra file văn bản INTERSECT.OUT 2 số nguyên – số thứ tự của 2 đoạn thẳng giao nhau. Nếu tồn tại nhiều cặp đoạn thẳng giao nhau thì đưa ra một kết quả tùy chọn. Nếu không tồn tại cặp đoạn thẳng giao nhau thì đưa ra -1 và -1 .

Ví dụ:

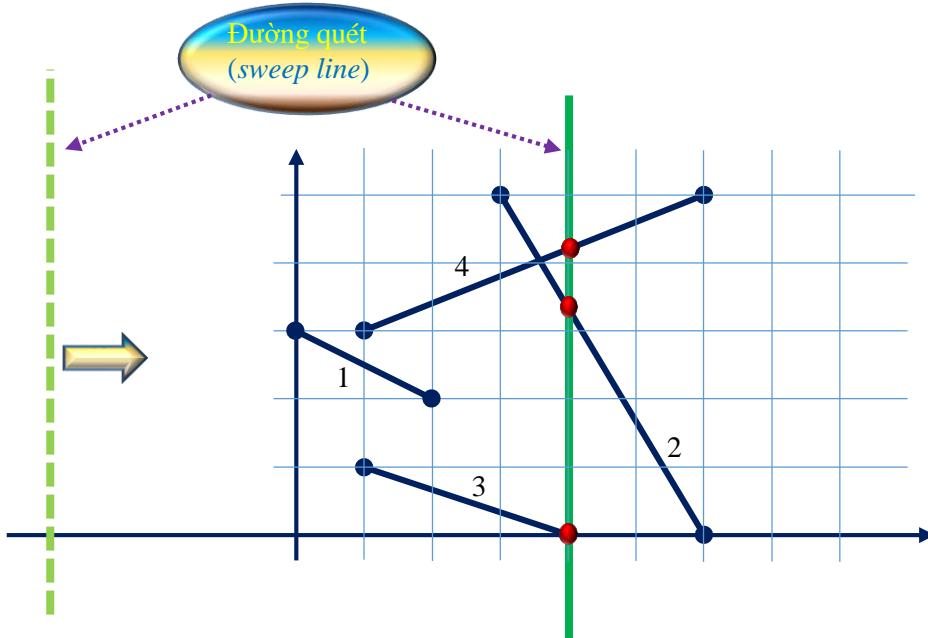
| INTERSECT.INP |
|---------------|
| 4 |
| 0 3 2 2 |
| 3 5 6 0 |
| 1 1 4 0 |
| 1 3 6 5 |

| INTERSECT .OUT |
|----------------|
| 4 2 |

Giải thuật

Phương pháp: giải thuật *đường thẳng quét* (*Sweep line*).

Đầu tiên xét trường hợp không có đoạn thẳng nào song song với trục Oy. Hình dung ta có đường thẳng $x = -\infty$ (đường thẳng vuông góc với trục Ox). Tịnh tiến đường thẳng này sang phải. Đến một lúc nào đó nó sẽ cắt một hoặc một số đoạn thẳng đã



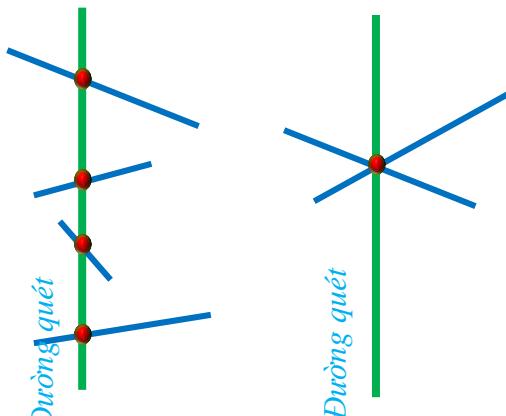
cho, tức là có với mỗi đoạn thẳng đó một điểm chung và đến một lúc nào đó – không cắt các đoạn thẳng đó nữa!

Điều mà ta quan tâm là vị trí tương đối của các đoạn thẳng bị đường quét cắt, tức là tọa độ y của giao điểm với đường quét.

Nếu 2 đoạn thẳng giao nhau thì sớm hay muộn giao điểm đó sẽ nằm trên đường quét.

Có thể rút ra một số kết luận:

- Để tìm cặp đoạn thẳng cắt nhau chỉ cần *kiểm tra cặp đoạn thẳng liên tiếp nhau* theo vị trí cắt trên đường quét,
- Không cần xét mọi vị trí của đường quét, chỉ cần quan tâm tới các vị trí khi nó *bắt đầu có điểm chung với đoạn mới* hoặc *khi một điểm chung bị mất*,
- Khi có đoạn mới cắt đường quét, cần nạp nó vào *danh sách quản lý các đoạn bị cắt* theo *trình tự từ dưới lên trên*,
- Chỉ cần kiểm tra cắt nhau giữa đoạn mới với *hai đoạn trên* và *dưới* nó,



- + Khi loại một đoạn thẳng khỏi danh sách quản lý, cần *kiểm tra* khả năng cắt nhau của 2 đoạn *trên* và *dưới* đoạn bị loại,
- + Không cần lưu tọa độ điểm cắt với đường quét cũng như thực hiện các loại kiểm tra nào khác.

Các kết luận trên được rút ra từ các nhận xét:

Với hai đoạn không cắt nhau *vị trí tương đối* của chúng trong tập quản lý *không thay đổi*, thật vậy, nếu một đoạn lúc đầu đứng trước đoạn kia, tức là giao với đường quét cao hơn, nhưng nếu sau đó – lại đứng thấp hơn thì giao của 2 đoạn với đường quét đã có lúc gặp nhau trên đường quét, tức là 2 đoạn giao nhau, đó là điều trái với giả thiết ban đầu!

Hệ quả từ nhận xét trên: Trong tập các đoạn đang được quản lý không tồn tại đồng thời 2 điểm giao cùng độ cao y trên đường quét với 2 đoạn không giao nhau,

Như vậy với các đoạn không giao nhau, sau khi đã ghi nhận trình tự xuất hiện trên đường quét ta chỉ phải chờ đợi khi nó ra khỏi đường quét,

Với các đoạn giao nhau: ở thời điểm giao nhau chúng *đứng cạnh nhau trong dòng xếp hàng quản lý* các đoạn giao với đường quét,

Như vậy ta chỉ cần *kiểm tra 2 đoạn liên tiếp* nhau trong dòng xếp hàng và việc kiểm tra chỉ cần thực hiện *khi xuất hiện đoạn mới* hay *một đoạn rời khỏi dòng xếp hàng* (thay đổi láng giềng).

Lưu ý:

- Nếu đồng thời xuất hiện đoạn mới và loại bỏ đoạn cũ thì phải *thực hiện các phép xử lý liên quan tới bỏ sung trước*, sau đó mới xử lý sự kiện loại bỏ. Quy trình này cho phép *xử lý cả các đoạn song song với trực Oy*. Với những đoạn loại này thời điểm bỏ sung trùng với thời điểm loại bỏ, nhưng nó vẫn được kết nạp vào dòng xếp hàng và *xử lý trước khi loại bỏ*. Đoạn song song với Oy có nhiều điểm giao với đường quét và có thể có nhiều hơn 2 láng giềng, nhưng ta chỉ cần làm việc với 2 trong số đó là đủ!
- Các đại lượng thực trung gian trong giải thuật được dẫn xuất trực tiếp từ các dữ liệu nguyên, vì vậy không sẽ không có hiện tượng tích lũy sai số làm tròn. Tuy nhiên, do đại lượng thực được lưu trữ gần đúng nên 2 đại lượng được coi là bằng nhau nếu nó chênh lệch không quá EPS đủ nhỏ.

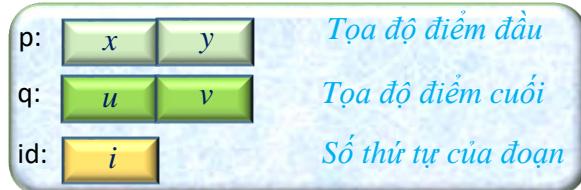
Đánh giá độ phức tạp của giải thuật:

- + Độ phức tạp O(1) để ghi nhận hoặc loại bỏ một đoạn vào dòng quản lý,
- + Tối đa phải thực hiện **n** lần ghi nhận/loại bỏ,

- Thực hiện kiểm tra giao nhau không quá $2n$ lần, mỗi lần – $O(\log n)$,
- Tổng hợp: Độ phức tạp của giải thuật: $O(n \log n)$.

Tổ chức dữ liệu và xử lý:

Thông tin về các đoạn thẳng lưu dưới dạng mảng bản ghi:



C++ là ngôn ngữ lập trình hướng đối tượng (*OOP – Object Oriented Programming*), nó cho phép gắn phép xử lý với đối tượng.

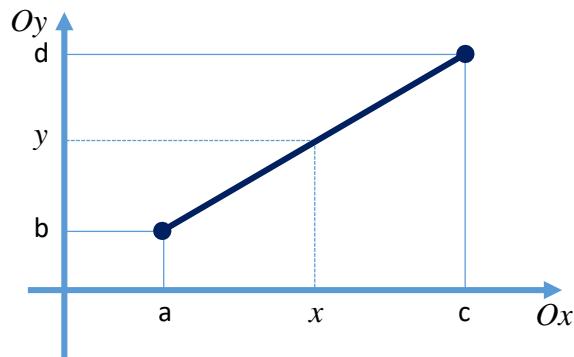
Với đoạn thẳng điểm đầu (a, b), điểm cuối (u, v), phương trình đường thẳng chứa đoạn thẳng sẽ có dạng:

$$\frac{x - a}{c - a} = \frac{y - b}{d - b}$$

Từ đây có

$$y = b + (d - b) \times \frac{x - a}{c - a}$$

Với đường quét đi qua điểm $(x, 0)$ ta cần tính tung độ giao điểm với đoạn thẳng, vì vậy khai báo kiểu dữ liệu có dạng:



```

struct pt {
    double x, y;
};

struct seg
{
    pt p, q;           Phép xử lý
    int id;
    double get_y(double x) const
    {
        if (abs(p.x - q.x) < EPS) return p.y;
        return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
    }
};

```

Đoạn thẳng là một điểm

Trả về tọa độ y

Kiểm tra 2 đoạn thẳng cắt nhau:

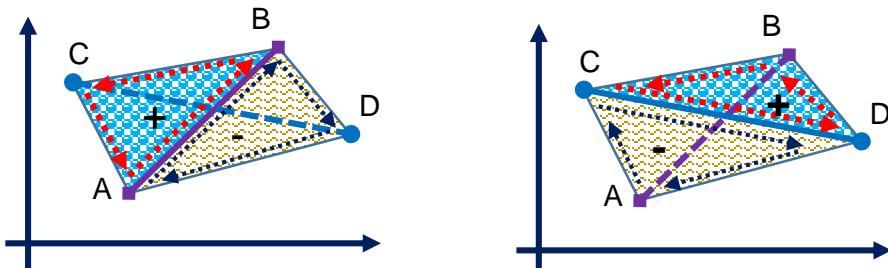
Hai đoạn thẳng AB và CD cắt nhau ở *điểm trong* của mỗi đoạn khi và chỉ khi đồng thời thỏa mãn 2 điều kiện: A và B nằm ở 2 phía của đường thẳng chứa CD và C, D nằm ở 2 phía của đường thẳng chứa AB.

Tuy nhiên, ta còn phải lưu ý trường hợp điểm cắt nhau là điểm đầu hay cuối của một đoạn thẳng nào đó.

Điều kiện cần của việc 2 đoạn thẳng có điểm chung là các *hình chiếu* của chúng trên trục tọa độ phải *có điểm chung*.

Việc giải **hệ phương trình tuyến tính** 2 ẩn số tìm điểm chung đòi hỏi phải xử lý trường hợp hệ phương trình **vô nghiệm** hay có **vô số nghiệm**.

Để tránh các phép kiểm tra phức tạp ta tính diện tích có dấu ΔABC theo trình tự duyệt $A \rightarrow B \rightarrow C$ và diện tích có dấu ΔABD theo trình tự duyệt $A \rightarrow B \rightarrow D$. Nếu 2 diện tích này trái dấu – C và D nằm ở 2 phía của AB. Tương tự như vậy đối với đoạn thẳng CD và các điểm A, B.



Trường hợp tồn tại diện tích bằng 0 – kết quả kiểm tra hình chiếu sẽ hỗ trợ việc rút ra kết luận.

Nhóm hàm sau thực hiện các công việc kiểm tra nêu trên:

*Macro kiểm tra
hình chiếu*

```
inline bool intersect1d (double l1, double r1, double l2, double r2)
{
    if (l1 > r1)    swap (l1, r1);
    if (l2 > r2)    swap (l2, r2);
    return max (l1, l2) <= min (r1, r2) + EPS;
}

inline int vec (const pt & a, const pt & b, const pt & c)
{
    double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    return abs(s)<EPS ? 0 : s>0 ? +1 : -1;
}

bool intersect (const seg & a, const seg & b)
{
    return intersect1d (a.p.x, a.q.x, b.p.x, b.q.x)
        && intersect1d (a.p.y, a.q.y, b.p.y, b.q.y)
        && vec (a.p, a.q, b.p) * vec (a.p, a.q, b.q) <= 0
        && vec (b.p, b.q, a.p) * vec (b.p, b.q, a.q) <= 0;
}
```

*Macro tính
diện tích*

Macro tương tự như hàm, nhưng sẽ được triển khai ngay tại nơi gọi.

Mảng `vector<event>` `e` ghi nhận sự kiện đường quét có còn cắt đoạn thẳng thứ `i` hay không và nếu còn thì cắt bắt đầu và kết thúc ở các hoành độ nào.

Cấu trúc `event` được kết gắn với phép xử lý của [2 cấu trúc event](#), khác với cách kết gắn xử lý ở cấu trúc `seg` – xử lý [dữ liệu nội bộ](#) một cấu trúc.

```
struct event
{
    double x;
    int tp, id;
    event() { }
    event (double x, int tp, int id)
        : x(x), tp(tp), id(id)
    { }

    bool operator< (const event & e) const
    {
        if (abs (x - e.x) > EPS)  return x < e.x;
        return tp > e.tp;
    }
};
```

Tập hợp `set<seg>` `s` quản lý các đoạn bị đường quét cắt. Với tập hợp này có 2 macro hỗ trợ `next` và `prev` xác định 2 đoạn láng giềng của đoạn đang xét.

Mảng `vector < set<seg>::iterator >` `where` lưu các con trỏ chỉ tới đoạn tương ứng trong `s`.

Tìm đoạn giao nhau:

- Khi đoạn mới xuất hiện: kiểm tra đoạn mới với đoạn trên và với đoạn dưới trong `s`,
- Khi một đoạn bị loại: kiểm tra 2 láng giềng của nó,
- Lưu ý trường hợp khi chỉ có một láng giềng.

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "intersect."
using namespace std;
typedef pair<int,int> pii;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n;
pii ans;
const double EPS = 1E-9;
struct pt {
    double x, y;
};
struct seg
{
    pt p, q;
    int id;
    double get_y (double x) const
    {
        if (abs (p.x - q.x) < EPS)  return p.y;
        return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
    }
};

vector<seg> a;

inline bool intersect1d (double l1, double r1, double l2, double r2)
{
    if (l1 > r1)  swap (l1, r1);
    if (l2 > r2)  swap (l2, r2);
    return max (l1, l2) <= min (r1, r2) + EPS;
}

inline int vec (const pt & a, const pt & b, const pt & c)
{
    double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    return abs(s)<EPS ? 0 : s>0 ? +1 : -1;
}

bool intersect (const seg & a, const seg & b)
{
    return intersect1d (a.p.x, a.q.x, b.p.x, b.q.x)
        && intersect1d (a.p.y, a.q.y, b.p.y, b.q.y)
        && vec (a.p, a.q, b.p) * vec (a.p, a.q, b.q) <= 0
        && vec (b.p, b.q, a.p) * vec (b.p, b.q, a.q) <= 0;
}

bool operator< (const seg & a, const seg & b)
{
    double x = max (min (a.p.x, a.q.x), min (b.p.x, b.q.x));
    return a.get_y(x) < b.get_y(x) - EPS;
}

struct event
{
    double x;
    int tp, id;
    event() { }
}
```

```

event (double x, int tp, int id)
    : x(x), tp(tp), id(id)
{ }

bool operator< (const event & e) const
{
    if (abs (x - e.x) > EPS) return x < e.x;
    return tp > e.tp;
}
};

set<seg> s;
vector < set<seg>::iterator > where;

inline set<seg>::iterator prev (set<seg>::iterator it)
{
    return it == s.begin() ? s.end() : --it;
}
inline set<seg>::iterator next (set<seg>::iterator it)
{
    return ++it;
}

pair<int,int> solve (const vector<seg> & a)
{
    int n = (int) a.size();
    vector<event> e;
    for (int i=0; i<n; ++i)
    {
        e.push_back (event (min (a[i].p.x, a[i].q.x), +1, i));
        e.push_back (event (max (a[i].p.x, a[i].q.x), -1, i));
    }
    sort (e.begin(), e.end());
    s.clear();
    where.resize (a.size());
    for (size_t i=0; i<e.size(); ++i)
    {
        int id = e[i].id;
        if (e[i].tp == +1)
        {
            set<seg>::iterator nxt = s.lower_bound (a[id]), prv=prev (nxt);
            if (nxt != s.end() && intersect (*nxt, a[id]))
                return make_pair (nxt->id, id);
            if (prv != s.end() && intersect (*prv, a[id]))
                return make_pair (prv->id, id);
            where[id] = s.insert (nxt, a[id]);
        }
        else
        {
            set<seg>::iterator nxt=next (where[id]), prv=prev (where[id]);
            if (nxt != s.end() && prv != s.end() && intersect (*nxt, *prv))
                return make_pair (prv->id, nxt->id);
            s.erase (where[id]);
        }
    }
    return make_pair (-1, -1);
}

int main()

```

```
{  
    fi>>n;  
    a.resize(n);  
    for(int i=0;i<n;++i)  
    {  
        fi>>a[i].p.x>>a[i].p.y>>a[i].q.x>>a[i].q.y;  
        a[i].id=i;  
    }  
    ans=solve(a);  
    fo<<ans.first+1<<' ' <<ans.second+1;  
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";  
}
```



XỬ LÝ XÂU

HÀM Z

Hàm Z của xâu **S** là mảng **Z**, trong đó Z_i là độ dài lớn nhất của tiền tố xâu con **U** các ký tự liên tiếp nhau của **S** bắt đầu từ vị trí i đồng thời cũng là tiền tố của xâu **S**.

Z_0 thường được gán giá trị bằng 0 hoặc bằng độ dài của xâu **S**.

Ví dụ: **S** = '**abcdabscabcdabia**'

$$Z(S)=[16,0,0,0,2,0,0,0,6,0,0,0,2,0,0,1].$$

Hàm Z được sử dụng trong nhiều bài toán xử lý xâu, đặc biệt có hiệu quả khi tìm kiếm theo mẫu.

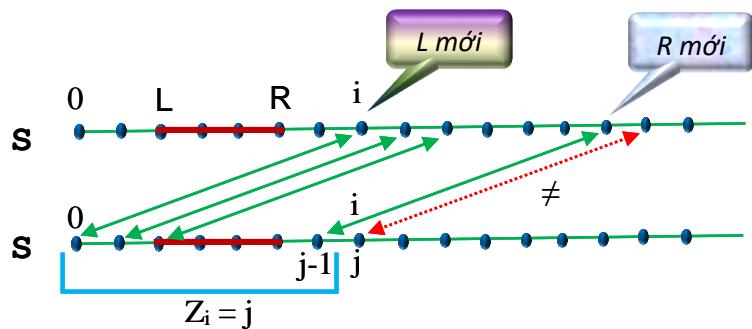
Giải thuật xác định Z:

- ✚ Các ký tự của xâu **S** được đánh số bắt đầu từ 0,
- ✚ Ký hiệu **L** và **R** là điểm đầu và cuối của tiền tố ứng với giá trị **R** lớn nhất đã tính được, ban đầu **L** = **R** = 0,
- ✚ $Z_0 = s.size()$,
- ✚ Giả thiết đã tính được Z_1, Z_2, \dots, Z_{i-1} ,
- ✚ Tính Z_i :
 - ❖ Nếu $i \in [L, R]$:
 - Đặt $j = i - L$,
 - Nếu $i + Z_j \leq R \rightarrow Z_i = Z_j$,
 - Nếu $i + Z_j > R$: Duyệt tiếp các ký tự sau **R**, kiểm tra sự trùng nhau các tiền tố (của **S** và của xâu con bắt đầu từ i), gán độ dài tìm được cho Z_i , điểm cuối mới – cho **R**, gán **L** = **i**.
 - ❖ Nếu $i \notin [L, R]$: So sánh trực tiếp các ký tự của xâu con (bắt đầu từ i) và của xâu **S**, tìm tiền tố dài nhất để xác định Z_i , cập nhật lại **L** và **R** (đầu và cuối tiền tố ở xâu con).

Độ phức tạp của giải thuật: Mỗi ký tự của xâu **S** được xét không quá 2 lần, vì vậy ta có độ phức tạp của giải thuật là $O(n)$, trong đó **n** – độ dài xâu **S**.

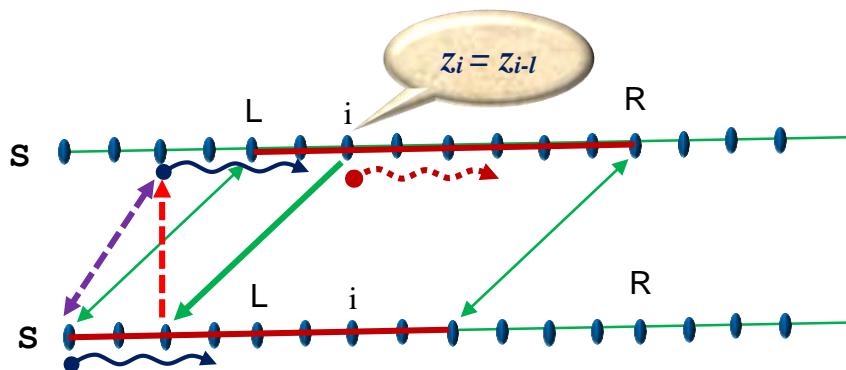
Sơ đồ hoạt động của giải thuật:

Khi i nằm ngoài khoảng $[L, R]$: ta có $i > R$,

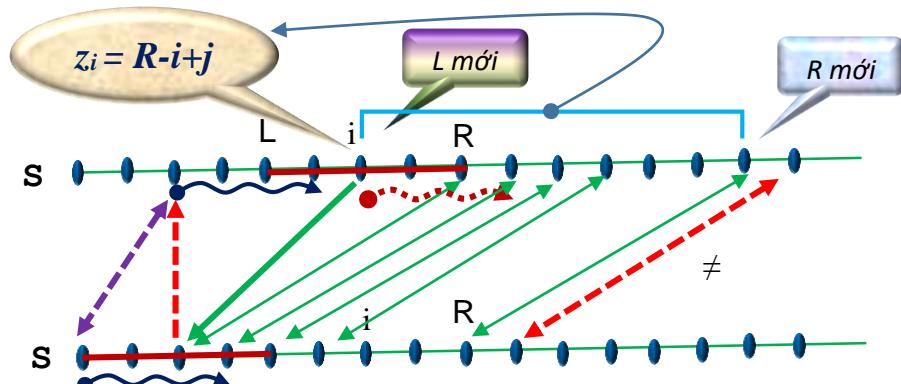


Khi i thuộc khoảng $[L, R]$: Hai trường hợp:

Trường hợp $Z_{i-L} < R-i+1$



Trường hợp $Z_{i-L} \geq R-i+1$

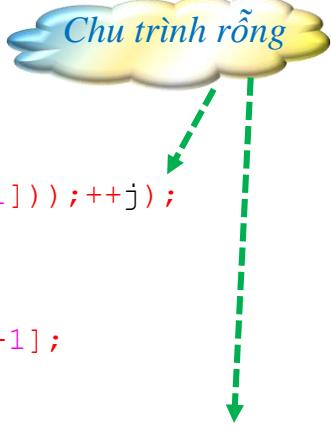


Hàm C++ tính Z:

```
vector<int> calc_z (string s) {
    vector<int> z;
    int len = s.size();
    z.resize(len);
    z[0] = len;
    int l = 0, r = 0;
    int j;
    for (int i = 1; i < len; i++)
        if (i > r) {
            for (j = 0; ((j + i) < len) && (s[i+j]==s[j]) ; j++)
                z[i] = j;
            l = i;
            r = i + j - 1;
        }
        else
            if (z[i - l] < r - i + 1)
                z[i] = z[i - l];
            else{
    for (j = 1; ((j + r) < len) && (s[r+j] == s[r-i+j]); j++)
        z[i] = r - i + j;
        l = i;
        r = r + j - 1;
    }
    return z;
}
```

Tương tự như vậy, ta có thể xây dựng hàm *tính trực tiếp* hậu tố (suffix):

```
void calc_zr(string s)
{int j,l,r;
zr[n-1]=n; l=n; r=n;
for(int i=n-2;i>=0;--i)
{
    if(i<l)
    {
        for(j=0; ((i-j)>=0) && (s[i-j]==s[n-j-1]); ++j);
        zr[i]=j; r=i; l=i-j+1;
    }
    else
        if(zr[n-r+i-1]<i-l+1) zr[i]=zr[n-r+i-1];
        else
        {
            for(j=0; ((l-j)>=0) && (s[l-j]==s[i-l-j]); ++j);
            zr[i]=i-l+j; r=i; l=(j-1);
        }
}
}
```



Giải thuật 2 tính hàm Z

Sơ đồ trên cho phép tính hàm Z một cách nhanh nhất nhưng *độ phức tạp chương trình* hơi cao. Tồn tại sơ đồ tính toán có *độ phức tạp của giải thuật* tăng thêm đôi chút nhưng độ phức tạp của chương trình giảm đáng kể.

Ta nhận thấy ở vị trí i bất kỳ việc tính xâu tiền tố có thể xuất phát từ độ dài xâu tiền tố đã tính ở vị trí $i-L$. Ban đầu ta có z_i không vượt quá z_{i-L} và phần còn lại $R-i+1$ của tiền tố đã tính.

Để xác định giá trị thực của z_i ta chỉ cần kiểm tra xem có thể tiếp tục kéo dài tiền tố hay không bằng cách so sánh trực tiếp các ký tự tiếp theo.

Hàm sau cho phép tính Z bắt đầu từ một vị trí x của xâu s và ghi kết quả vào mảng z (để thuận tiện cho các xử lý tiếp theo nên gán giá trị đầu $z_x = 0$).

```
void calc_z(string s, int x, int * z)
{int l, r;
 z[x] = 0;
 for (int i = x+1, l = 0, r = 0; i<n; ++i) {
    z[i] = min(z[i - l], max(0, r - i + 1));
    while (s[z[i]] == s[i + z[i]])
        z[i]++;
    l = i, r = i + z[i] - 1;
 }
}
```

Ví dụ ứng dụng

VR16. THUẦN CHỦNG

Gene là một đoạn kết nối các cặp AND, mỗi cặp AND được đặc trưng bằng một chữ cái trong tập $\{A, C, G, T\}$. Gene thuần chủng là gene hình thành từ một đoạn AND cơ sở độ dài không quá m , được kết nối lặp đi lặp lại nhiều lần và ở lần lặp cuối cùng có thể chỉ chứa phần đầu của đoạn cơ sở. Gene được mô tả dưới dạng xâu S chỉ chứa các ký tự trong tập nêu trên. Như vậy gene thuần chủng là xâu có thể biểu diễn như tổng của k đoạn cơ sở ($k \geq 0$) và có thể có thêm một đoạn đầu của cơ sở.

Ví dụ, với $m = 10$, $S = "ACATAGACATAGACATAGACA"$ là một gene thuần chủng vì có đoạn cơ sở là "**ACATAG**" và $S = "ACATAG" + "ACATAG" + "ACATAG" + "ACA"$, nhưng với $m = 5$ thì S không phải là gene thuần chủng.

Cho gene S độ dài n và giá trị m . Hãy xác định S có phải là gene thuần chủng hay không và đưa ra *đoạn cơ sở ngắn nhất* nếu S là gene thuần chủng hoặc đưa ra thông báo "**NO**" trong trường hợp ngược lại.

Dữ liệu: Vào từ file văn bản PURE.INP:

- ✚ Dòng đầu tiên chứa số nguyên m ($1 \leq m \leq 10^6$),
- ✚ Dòng thứ 2 chứa xâu S độ dài n chỉ chứa các ký tự trong tập đã nêu.

Kết quả: Đưa ra file văn bản PURE.OUT đoạn cơ sở ngắn nhất tìm được hoặc thông báo **NO**.

Ví dụ:

| PURE.INP | PURE.OUT |
|-----------------------|---------------|
| 10 | |
| ACATAGACATAGACATAGACA | ACATAG |



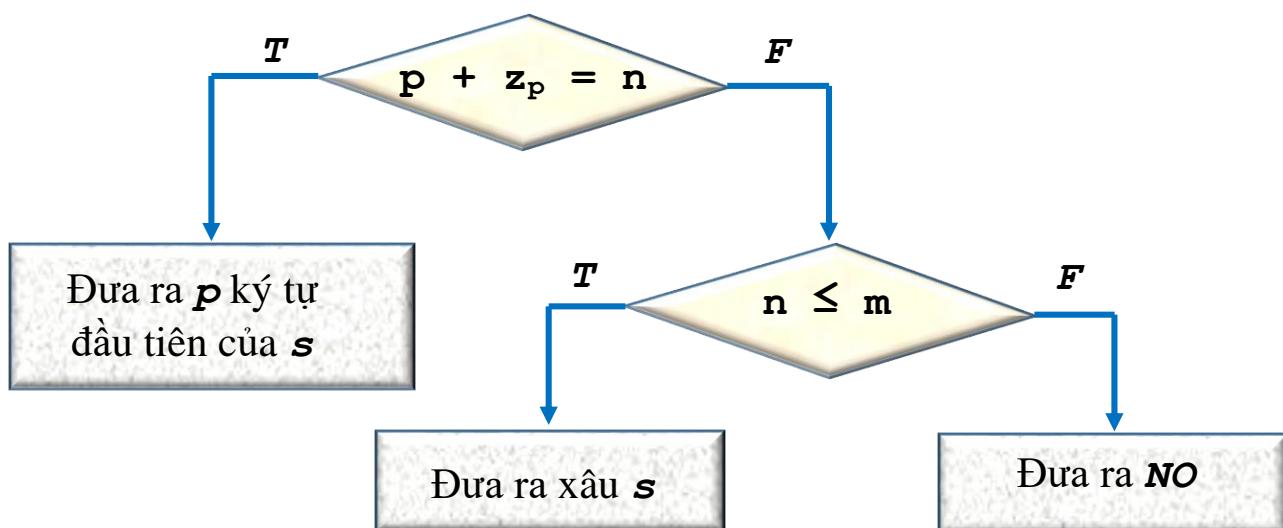
Giải thuật: *Ứng dụng hàm z.*

Tổ chức dữ liệu:

Mảng **int** $\mathbf{z}[1000010]$ lưu trữ độ dài tiền tố, \mathbf{z}_i – độ dài lớn nhất của xâu ký tự bắt đầu từ i là tiền tố của xâu s .

Các bước xử lý:

- ✚ Nhập dữ liệu,
- ✚ Tính \mathbf{z} ,
- ✚ Tính $\mathbf{k} = \min\{\mathbf{n}, \mathbf{m}\}$,
- ✚ Xác định $\mathbf{zx} = \max\{\mathbf{z}_i, i = 1 \div \mathbf{k}\} = \mathbf{z_p}$,
- ✚ Kiểm tra:



Độ phức tạp của giải thuật: $O(n)$.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "pure."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
string s,sp;
int n,m,z[1000010];

void calc_z()
{int j,l=0,r=0;
z[0]=n;
for(int i=1;i<n;++i)
{
    if(i>r)
    {   for(j=0; ((j+i<n) && (s[i+j]==s[j])) ;++j);
        z[i]=j; l=i; r=i+j-1;
    }
    else
        if(z[i-1]<r-i+1) z[i]=z[i-1];
        else
        {
            for(j=1; ((j+r<n) && (s[r+j]==s[r-i+j])) ;++j);
            z[i]=r-i+j; l=i; r+=(j-1);
        }
}
}

int main()
{
    fi>>m;
    fi>>s; n=s.size();
    calc_z();
    int k,p,zx;
    k=min(n,m);
    zx=0;
    for(int i=1;i<=k;++i) if(zx<z[i]) zx=z[i],p=i;
    if(p+z[p]==n) for(int i=0;i<p;++i) fo<<s[i];
    else if(n<=m) fo<<s; else fo<<"NO";

    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



GIẢI THUẬT MANAKER TÌM PALINDROME

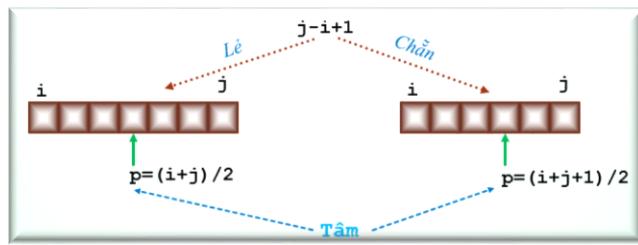
Bài toán

Cho xâu **s** độ dài n . Xâu **u** tạo thành từ dãy các ký tự liên tiếp nhau của **s** được gọi là xâu con của **s**. Hai xâu con **u** và **v** được gọi là khác nhau nếu tồn tại ít nhất một i để ký tự s_i tham gia vào **u** và không tham gia vào **v** hoặc tham gia vào **v** và không tham gia vào **u**.

Hãy xác định số lượng xâu con độ dài lớn hơn 1 là palindrome của **s**.

Giải thuật

Xét xâu con $u = s[i..j]$. Nếu $j-i+1$ là lẻ thì $p = (i+j)/2$ được gọi là vị trí trung tâm (gọi ngắn gọn là **tâm**) của **u**, nếu $j-i+1$ chẵn thì tâm là $p = (i+j+1)/2$.

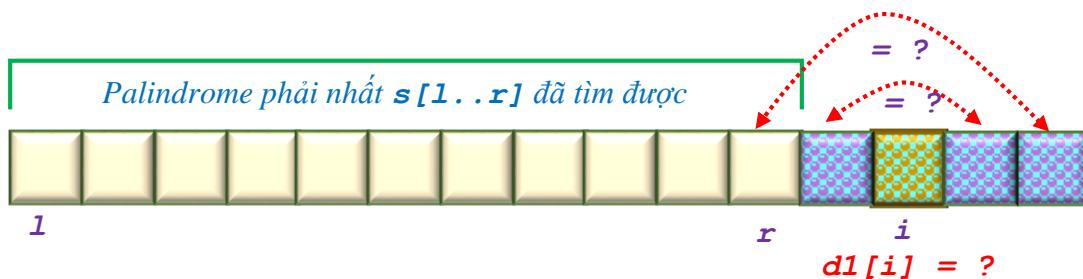


Gọi $d1[i]$ là số lượng các xâu con palindrome độ dài lẻ lớn hơn 1, có tâm là i , $d2[i]$ là số lượng các xâu con palindrome độ dài chẵn lớn hơn 1, có tâm là i . Giả thiết $s[1..r]$ là palindrome phải nhất trong số các palindrome tìm được.

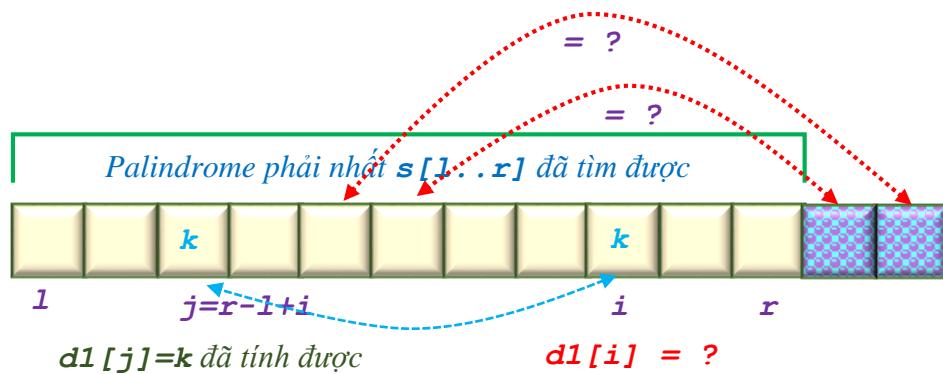
Giả thiết $d1[j]$ và $d2[j]$ đã tính được với mọi $j < i$.

Xét cách tính $d1[i]$.

Có 2 trường hợp xảy ra:



- ✚ $i > r \rightarrow$ so sánh trực tiếp s_{i-1} với s_{i+1} , s_{i-2} với s_{i+2}, \dots để tìm ra palindrome lớn nhất có tâm là i .
- ✚ $i \leq r \rightarrow s_i$ thuộc palindrome đã xác định. Do tính đối xứng của palindrome, nếu không tính đến giá trị r , ta có $d1[r-1+i] = k = d1[i]$, nhưng phần còn lại từ i đến r có thể nhỏ hơn k . Khả năng có thể mở rộng biên r được kiểm tra bằng cách so sánh trực tiếp các ký tự tiếp theo.



Trong mọi trường hợp, sau khi tính $d1[i]$ cần cập nhật **l** và **r**.

Hàm tính **d1**:

```
vector<int> calc_1()
{
    vector<int> d(n, 0);
    int l=0, r=-1;
    for(int i=0; i<n; ++i)
    {
        int k=0;
        if(i<=r) k=min(r-i, d[r-i+1]);
        while(i+k+1<n && i-k-1>=0 && s[i+k+1] == s[i-k-1]) ++k;
        d[i]=k;
        if(i+k>r) l=i-k, r=i+k;
    }
    return d;
}
```

Với sự điều chỉnh chỉ số thích hợp, ta có hàm tính **d2**:

```
vector<int> calc_2()
{
    vector<int> d(n, 0);
    int l=0, r=-1;
    for(int i=0; i<n; ++i)
    {
        int k=0;
        if(i<=r) k=min(r-i+1, d[r-i+1+1]);
        while(i+k+1<n && i-k-1>=0 && s[i+k] == s[i-k-1]) ++k;
        d[i]=k;
        if(i+k-1>r) l=i-k, r=i+k-1;
    }
    return d;
}
```

Các palindrome đếm được sẽ không bị lặp hoặc có tâm khác nhau hoặc khác nhau về độ dài với các palindrome cùng tâm.

Đánh giá độ phức tạp

Để tính $d1[i]$ cần duyệt với $i = 0 \div n-1$,

- ⊕ Với $i > r$, chu trình lặp ở trong thực hiện bao nhiêu lần thì r sẽ tăng lên bấy nhiêu,
- ⊕ Với $i \leq r$ có thể xảy ra 2 trường hợp:
 - ⓧ $i+d1[j]-1 \leq r \rightarrow$ chu trình trong sẽ có số lần lặp bằng không,
 - ⓧ $i+d1[j]-1 > r \rightarrow$ chu trình lặp ở trong thực hiện bao nhiêu lần thì r sẽ tăng lên bấy nhiêu.

Như vậy, r tăng tuyến tính theo số lần lặp của chu trình trong. r không thể vượt quá $n-1$ vì vậy độ phức tạp của giải thuật sẽ là $O(n)$.

Ví dụ ứng dụng

VV44. SỐ LƯỢNG PALINDROME

Cho xâu s độ dài n . Người ta có thể xóa một số ký tự liên tiếp (có thể là 0) ở đầu xâu và xóa một số ký tự liên tiếp (có thể là 0) ở cuối xâu, nhưng không xóa rỗng xâu. Hai cách xóa gọi là khác nhau nếu tồn tại ít nhất một ký tự bị xóa ở cách thứ nhất nhưng không bị xóa ở cách thứ hai. Lưu ý rằng xâu chỉ chứa một ký tự cũng là xâu palindrome.

Hãy xác định số cách xóa khác nhau để nhận được xâu palindrome.

Dữ liệu: Vào từ file văn bản ALL_PAL.INP gồm một dòng chứa xâu s độ dài không quá 10^6 . Các ký tự trong s đều có mã ASCII lớn hơn 32.

Kết quả: Đưa ra file văn bản ALL_PAL.OUT một số nguyên – số cách xóa khác nhau có thể thực hiện.

Ví dụ:

| | |
|-------------|-------------|
| ALL_PAL.INP | ALL_PAL.OUT |
| aabacaba | 14 |



VV44

Giải thuật: *Ứng dụng giải thuật Manacher.*

Mỗi cách xóa \rightarrow một xâu con khác nhau các ký tự liên tiếp của s ,

Cần xác định có bao nhiêu xâu con độ dài lớn hơn 0 là palindrome.

Kết quả: $ans = n + \sum_{i=0}^{n-1} (d1_i + d2_i)$

Giải thuật Manacher: tính $d1_i$ và $d2_i$, $i = 0 \div n-1$ với độ phức tạp $O(n)$.

Độ phức tạp của giải thuật: $O(n)$.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "all_pal."
#define Times fo<<"\nTime: "<<clock()/(double)1000<<" sec"
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n;
string s;int64_t ans;

void calc_1 ()
{
    vector<int> d(n, 0);
    int l=0, r=-1;
    for(int i=0;i<n;++i)
    {
        int k=0;
        if(i<=r) k=min(r-i, d[r-i+1]);
        while(i+k+1<n && i-k-1>=0 && s[i+k+1] == s[i-k-1]) ++k;
        d[i]=k;
        if(i+k>r) l=i-k, r=i+k;
    }
    for(int i=0;i<n;++i) ans+=d[i];
    return ;
}

void calc_2 ()
{
    vector<int> d(n, 0);
    int l=0, r=-1;
    for(int i=0;i<n;++i)
    {
        int k=0;
        if(i<=r) k=min(r-i+1, d[r-i+1+1]);
        while(i+k+1<n && i-k-1>=0 && s[i+k] == s[i-k-1]) ++k;
        d[i]=k;
        if(i+k-1>r) l=i-k, r=i+k-1;
    }
    for(int i=0;i<n;++i) ans+=d[i];
    return ;
}

int main()
{
    fi>>s;
    n=s.size();ans=n;
    calc_1();
    calc_2();
    fo<<ans;
    Times;
}
```



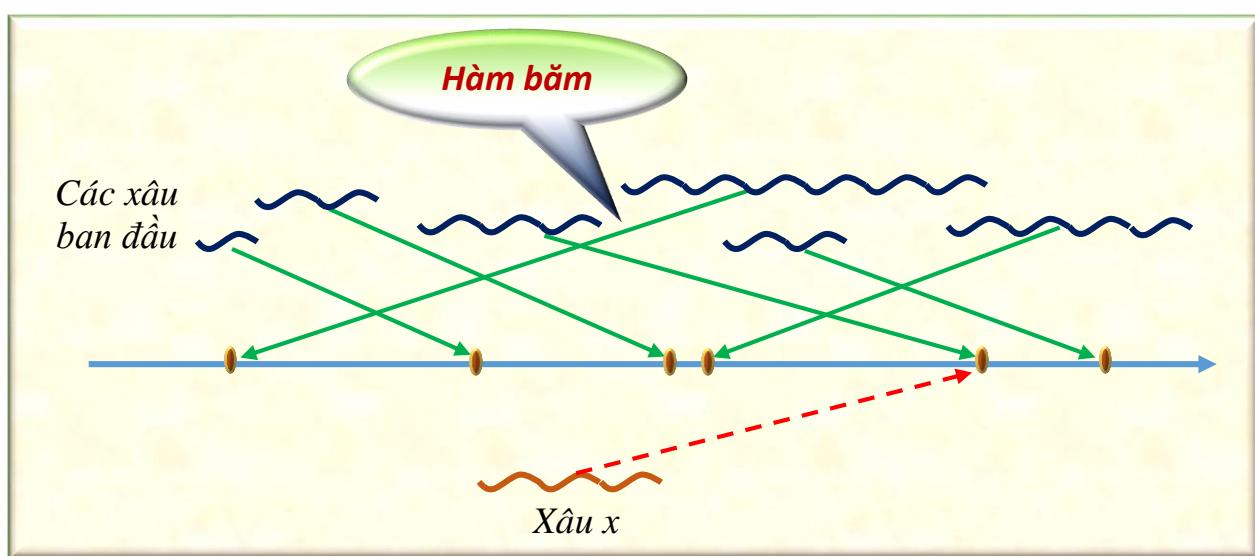
Kỹ thuật hàm băm

Băm (Hashing) là kỹ thuật dẫn xuất việc lưu trữ một tập hợp lớn về một tập nhỏ hơn. Hàm biến đổi khóa của các đối tượng ban đầu về khóa của các đối tượng trong tập nhỏ hơn được gọi là **hàm băm**.

Việc dẫn xuất các đối tượng từ tập lớn sang đối tượng tương đương ở tập nhỏ hơn sẽ làm giảm bớt các phép kiểm tra, so sánh phải thực hiện trong quá trình tìm kiếm, nhận dạng.

Ví dụ, cho tập gồm n xâu, xâu thứ i có độ dài m_i ($i = 1 \div n$). Hãy xác định xem một xâu x cho trước có mặt trong một số tập con của tập đã cho hay không. Một trong các phương pháp khá hiệu quả giải bài toán này là tổ chức một cây tiền tố, trong đó mỗi nút là một ký tự cùng với dấu hiệu cho biết đây có phải là ký tự cuối cùng của một xâu nào đó trong tập ban đầu hay không. Việc nhận dạng sự có mặt của xâu x sẽ được đưa về việc duyệt cây theo các ký tự của x . Nếu tồn tại một nhánh cây thuộc một tập con nào đó chứa các ký tự của x (theo trình tự xuất hiện trong x) và nút chứa ký tự cuối cùng của x có dấu hiệu kết thúc từ thì có nghĩa là x có mặt trong tập con đang xét, trong trường hợp ngược lại – câu trả lời là phủ định. Gọi độ dài xâu x là lx . Công việc kiểm tra sự tồn tại của x trong tập đã cho có độ phức tạp $O(lx)$. Với n , lx và các m_i đủ lớn, số tập con khá nhiều thì giải thuật nêu trên là chưa đủ hiệu quả. Giải thuật này đặc biệt không thích hợp khi phải liên tiếp kiểm tra với nhiều xâu x khác nhau. Ngoài ra việc tổ chức cây tiền tố khác phức tạp trong lập trình, tốn bộ nhớ và mất nhiều thời gian.

Với kỹ thuật hàm băm, ta có thể ánh xạ tập xâu ban đầu sang tập số nguyên, ví dụ trong phạm vi từ 1 đến 10^9 . Áp dụng ánh xạ tương tự với x , việc kiểm tra tồn tại có độ phức tạp không quá $O(\log n)$. Với một số các tổ chức ánh xạ, độ phức tạp có thể đạt đến mức $O(1)$!



Nếu hai hay nhiều đối tượng ban đầu khác nhau được hàm băm ánh xạ vào cùng một giá trị thì ta có trường hợp **va chạm**.

Hàm băm tốt là hàm ánh xạ cực tiêu hóa số lượng và chạm và kết quả ánh xạ phân bố đều trên tập kết quả. Hàm băm lý tưởng là hàm ánh xạ không có va chạm.

Có nhiều phương pháp xử lý va chạm, ví dụ tạo danh sách động lưu trữ thông tin về các đối tượng bị va chạm.

Một trong những cách hiệu quả phát hiện và khắc phục va chạm là áp dụng song song hai hàm băm. Xác xuất đồng thời xảy ra va chạm ở cùng một đối tượng với cả hai hàm băm sẽ cực nhỏ, gần như bằng 0 đối với các bài toán thực tế.

Với những bài toán olympic tin học và với cách ánh xạ sẽ xét dưới đây chỉ cần sử dụng một phép ánh xạ. Với h nguyên đủ lớn, ví dụ $h = 10^5 + 3$ hoặc $h = 2^{37}$ việc ánh xạ một xâu hoặc dãy số sang số nguyên được thực hiện như sau:

Ánh xạ xâu: xét s – xâu ký tự độ dài m cần tạo mảng giá trị b_0, b_1, \dots, b_{m-1} kiểu **int** hoặc **int64_t**:

- ❖ $b_0 = s_0,$
- ❖ $b_1 = b_0 \times h + s_1 = s_0 \times h + s_1,$
- ❖ $b_2 = b_1 \times h + s_2 = s_0 \times h^2 + s_1 \times h + s_2,$
- ❖
- ❖ $b_i = b_{i-1} \times h + s_i = s_0 \times h^i + s_1 \times h^{i-1} + \dots + s_{i-1} \times h + s_i,$
- ❖
- ❖ $b_{m-1} = b_{m-2} \times h + s_{m-1} = s_0 \times h^{m-1} + s_1 \times h^{m-2} + \dots + s_{m-2} \times h + s_{m-1}.$

Dãy số này cho phép so sánh 2 xâu con các ký tự liên tiếp độ dài bất kỳ với độ phức tạp $O(1)$.

Lưu ý rằng hệ thống lập trình C/C++ ngầm định ngăn ngừa việc bẫy và xử lý sự kiện tràn ô (*overflow*). Với các hệ thống lập trình có ngầm định việc bẫy và xử lý sự kiện tràn ô ta phải đặt chỉ thị cho chương trình dịch bỏ ngầm định này hoặc phải lấy số dư của phép chia b_i cho số nguyên đủ lớn md , như vậy kết quả ánh xạ sẽ nằm trong khoảng $[0, md-1]$.

Ánh xạ dãy số: Xét dãy số a_0, a_1, \dots, a_{m-1} . Công thức ánh xạ tương tự trường hợp xâu nếu thay s_i bằng a_i , $i = 0 \div m-1$.

Ví dụ ứng dụng

MẬT KHẨU

Ngân hàng GreenBank dùng loại mật khẩu sử dụng một lần cho mọi truy nhập tới các dịch vụ của ngân hàng.

Khi có yêu cầu truy nhập ngân hàng sẽ được cung cấp một từ khóa. Người truy nhập chỉ phải nhập vào mật khẩu là một xâu ký tự palindrome độ dài ngắn nhất có chứa từ khóa như một xâu con các ký tự liên tiếp nhau.

Với từ khóa đã cho hãy xác định mật khẩu cần nhập vào. Nếu tồn tại nhiều xâu khác nhau cùng đáp ứng yêu cầu là mật khẩu thì đưa ra xâu bất kỳ trong số đó.

Dữ liệu: Vào từ file văn bản PAROLE.INP gồm một dòng chứa từ khóa có độ dài không vượt quá 3×10^5 và chỉ bao gồm các ký tự la tinh thường.

Kết quả: Đưa ra file văn bản PAROLE.OUT mật khẩu tìm được.

Ví dụ:

| PAROLE.INP |
|------------|
| ab |
| a |

| PAROLE.OUT |
|------------|
| aba |
| a |



Giai thuật

Sử dụng hàm băm (Hashing Func.): với h nguyên dương và đủ lớn:

$$p_i = h^i \bmod d,$$

$$a_0 = s_0,$$

$$a_i = (a_{i-1} \times h^i + s_i) \bmod d, i = 1 \div n-1,$$

$$b_{n-1} = s_{n-1},$$

$$b_i = (b_{i+1} \times h^i + s_i) \bmod d, i = n-2 \div 0.$$

$$b_{n-1} = s_{n-1},$$

$$b_{n-2} = b_{n-1} \times h + s_{n-2},$$

.....

$$b_{x+1} = b_{x+2} \times h + s_{x+1} \equiv y,$$

$$b_x = b_{x+1} \times h + s_x = y \times h + s_x,$$

$$b_{x-1} = b_x \times h + s_{x-1} = y \times h^2 + s_x \times h + s_{x-1},$$

$$b_{x-2} = b_{x-1} \times h + s_{x-2} = y \times h^3 + s_x \times h^2 + s_{x-1} \times h + s_{x-2},$$

.....

$$b_0 = b_1 \times h + s_0 = y \times h^{x+1} + s_x h^x + s_{x-1} \times h^{x-1} + \dots + s_2 \times h^2 s_1 \times h^1 + s_0,$$

$$= y \times p_{x+1} + s_x h^x + s_{x-1} \times h^{x-1} + \dots + s_2 \times h^2 s_1 \times h^1 + s_0,$$

$$a_x = s_0 h^x + s_1 \times h^{x-1} + \dots + s_{x-2} \times h^2 s_{x-1} \times h^1 + s_x,$$

```
cc=s[n-1];ixb=n-1;
for(int i =0;i<n;+i)
    if(cc==s[i]) {gethr(i);
        if(t1==t2) {ixb=i;break;}}
```



```
void gethr(int x)
{ t1=b[x]; t2 = a[n-1]-a[x-1]*p[n-x]; }
```

Xâu S:



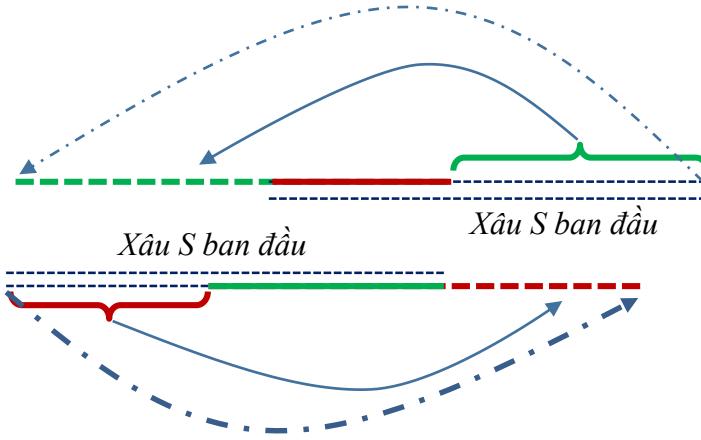
Palindrome độ dài k

```
cc=s[0];ixa=0;
for(int i=n-1;i>0;--i)
    if(cc==s[i])
        {geths(i); if(t1==t2){ixa=i;break;}}
```



```
void geths(int x)
{ t1=a[x]; t2 = b[0]-b[x+1]*p[x+1]; }
```

Xác định kết quả:



Kéo dài xâu ở đâu có palindrome độ dài lớn hơn.

```
m=ixa+1; q=0; if(m<n-ixb) {m=n-ixb; q=1;}  
if(q==0) {for(int i=n-1; i>ixa; --i) fo<<s[i]; fo<<s;}  
else {fo<<s; for(int i=ixb-1; i>=0; --i) fo<<s[i];}
```

Độ phức tạp: O(n).

Chương trình

```
#include <fstream>
#include <string>
using namespace std;
string s;
char cc;
int n,ixa,ixb,tg1,tg2,q,m;
int64_t a[3000100],b[3000100],p[3000100],d[3000100],ph=500000,t1,t2;
ifstream fi ("parole.inp");
ofstream fo ("parole.out");

void geths (int x)
{ t1=a[x]; t2 = b[0]-b[x+1]*p[x+1]; }

void gethr (int x)
{ t1=b[x]; t2 = a[n-1]-a[x-1]*p[n-x]; }

int main()
{fi>>s; n=s.size();
p[0]=1;
for(int i=1;i<n+100;++i) p[i]=p[i-1]*ph;
a[0]=s[0];
for(int i=1; i<n;++i) a[i]=a[i-1]*ph+s[i];
b[n-1]=s[n-1];
for(int i=n-2;i>=0;--i) b[i]=b[i+1]*ph+s[i];
cc=s[0];ixa=0;
for(int i=n-1;i>0;--i)
    if(cc==s[i]) {geths(i); if(t1==t2) {ixa=i; break;}}
cc=s[n-1];ixb=n-1;
for(int i = 0;i<n;++i)
    if(cc==s[i]) {gethr(i); if(t1==t2) {ixb=i; break;}}
m=ixa+1;q=0; if(m<n-ixb) {m=n-ixb; q=1;}
if(q==0) {for(int i=n-1;i>ixa;--i) fo<<s[i]; fo<<s;}
else {fo<<s; for(int i=ixb-1;i>=0;--i) fo<<s[i];}
}
```



VS05. CON ĐƯỜNG GÓM SỨ

Sau khi bê tông hóa đê chống lụt, thành phố quyết định cho khảm lên tường bê tông của đê tranh ghép tạo bởi các mảnh gốm sứ lấy từ các lò gốm nổi tiếng trong nước. Toàn bộ con đê được chia thành n phần có độ rộng giống nhau, mỗi phần gọi là một lô. Mỗi bức tranh khảm trên đó đều phải có độ rộng giống nhau, tức là bao gồm một số như nhau các lô liên tiếp và toàn bộ tường phải được phủ kín tranh từ đầu đến cuối, mỗi lô phải được tạo màu chủ đạo (gọi là màu của lô) từ một loại gốm đặc trưng lấy từ một lò gốm nào đó trong nước, ví dụ gốm màu xanh Côn Đảo từ lò gốm Ánh Hồng Quảng Ninh, gốm da lươn – từ Bát Tràng Hà Nội, gốm mộc hồng nhạt – từ Biên Hòa Đồng Nai, . . . Các loại gốm này được đánh số từ 1 đến 50 000.



Hướng dẫn viên du lịch giới thiệu với khách tham quan là có 2 nhóm nghệ nhân được giao việc tạo hình và khảm tranh. Với mỗi nhóm các bức tranh của đều được đặc trưng bởi dãy số (c_1, c_2, \dots, c_k), trong đó k là độ rộng của tranh, c_i – màu của lô, $i = 1 \div k$, các bức tranh khác nhau có thể khác nhau ở trình tự xuất hiện màu của các lô, ví dụ với dãy số đặc trưng (2, 6, 2, 9), trình tự màu trong tranh có thể là (9, 2, 2, 6) hoặc (6, 9, 2, 2) nhưng không thể là (6, 9, 2, 3). Dãy đặc trưng của 2 nhóm là khác nhau, tức là không thể bằng phép hoán vị trình tự màu của lô để đưa một dãy về dãy kia. Các bức tranh được ghép với nhau rất hài hòa và khách tham quan không nhận biết được sự chuyển tiếp từ tranh này sang tranh khác. Tuy vậy nhiều khách tham quan vẫn muốn biết có bao nhiêu bức tranh đã tạo ra và trong đó số bức tranh của mỗi nhóm là bao nhiêu.

Hãy xác định số lượng tranh có thể có và số lượng tranh mỗi nhóm đã làm. biết rằng nhóm nào cũng có tranh của mình.

Dữ liệu: Vào từ file văn bản CERAMIC.INP:

- ✚ Dòng đầu tiên chứa một số nguyên n – số lượng lô của con đê ($2 \leq n \leq 10^5$),
- ✚ Dòng thứ 2 chứa n số nguyên a_1, a_2, \dots, a_n – màu của các lô ($1 \leq a_i \leq 50\,000, i = 1 \div n$).

Kết quả: Đưa ra file văn bản CERAMIC.OUT, dòng đầu tiên chứa số nguyên m – số lượng phương án khác nhau chia con đường thành các bức tranh, nếu không có cách phân chia để đảm bảo phân biệt tranh của đúng 2 nhóm thì đưa ra số -1. Nếu có cách phân biệt thì ở mỗi dòng tiếp theo đưa ra 3 số nguyên k, p và q – độ rộng bức tranh, số tranh do nhóm 1 thực hiện và số tranh do nhóm 2 thực hiện, thông tin đưa ra theo thứ tự tăng dần của k và ở mỗi dòng có $p \geq q > 0$.

Ví dụ:

| CERAMIC.INP |
|-------------------|
| 9 |
| 1 2 3 6 4 9 3 1 2 |

| CERAMIC.OUT |
|-------------|
| 1 |
| 3 2 1 |



Giải thuật: Tổng tiền tố và hàm băm.

Nhận xét:

- ✚ Việc nhận dạng 2 dãy số cùng độ dài **x** và **y** có trùng nhau với độ chính xác hoán vị được thực hiện tương tự như phương pháp đã nêu ở bài *VS04. Độ tương đồng*,
- ✚ Cần xác định có đúng 2 loại dãy con khác nhau,
- ✚ Phải duyệt mọi dãy con độ dài m là ước của n để tìm tất cả các nghiệm.

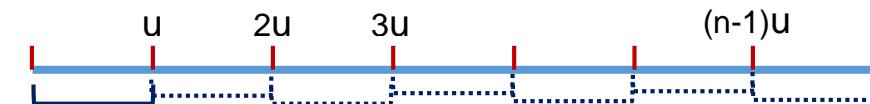
Tổ chức dữ liệu:

- ✚ Các mảng:
 - ▣ **int64_t a[100001]** – lưu tổng tiền tố nhận được từ ánh xạ mảng tần số xuất hiện của dữ liệu ban đầu sang tập số nguyên bằng kỹ thuật hàm băm,
 - ▣ **int64_t h[50001]** – mảng giá trị phục vụ tính hàm băm, $h_i = hb^i$, $i=0 \div 10^5$,
- ✚ Các vec tơ:
 - ▣ **vector<int> w** – lưu số lượng n phuong án tìm được,
 - ▣ **vector<pair<int, int> >ans** – lưu các cặp giá trị **p** và **q**,
 - ▣ Các biến **d1** và **d2** – lưu số lượng tranh mỗi nhóm thực hiện.

Các bước xử lý:

- Tính các hệ số **h[i]** của hàm băm, $i = 0 \div 500\ 000$,
- Tính các tổng tiền tố giá trị hàm băm ứng với dữ liệu vào,
- Với ước **u** của **n**:

- ❖ Kiểm tra tồn tại đúng 2 loại tranh độ rộng **u**,



- ❖ Nếu tồn tại: nạp **u** vào **w**, cặp giá trị (**d1**, **d2**) vào **ans**,

- Đưa ra kết quả tìm được.

Độ phức tạp của giải thuật: $\approx O(n \log n)$.

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "ceramic."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
uint64_t hb=5017,h[50001],x,y,a[100001];
int n,m,k,p,q,t,r;
vector<pair<int,int>>ans;
vector<int>w;

void check_m(int u)
{int t1=0,t2=0,d1=1,d2=0;
 uint64_t tg1,tg2,tg;
 tg1=a[u];tg2=0;
 for(int i=u;i<=n-u;i+=u)
 {
     tg=a[i+u]-a[i];
     if(tg==tg1)++d1;
     else
     {
         if(d2==0)tg2=tg,++d2;
         else {if(tg==tg2)++d2; else {r=-1;return;}}
     }
 }
 if(d1!=0 && d2!=0)
     {ans.push_back(make_pair(d1,d2)); w.push_back(u);}
}

int main()
{clock_t aa=clock();
 h[0]=1;
 for(int i=1;i<=50000;++i)h[i]=h[i-1]*hb;
 fi>>n;a[0]=0;
 for(int i=1;i<=n;++i){fi>>t;a[i]=a[i-1]+h[t];}
 for(int i=2;i<=n/2;++i)
     if(n%i==0)check_m(i);
 if(w.empty())fo<<-1;
 else
     {fo<<w.size()<<'\n';
      for(int i=0;i<w.size();++i)
          fo<<w[i]<<' '<<ans[i].first<<' '<<ans[i].second<<'\n';
     }

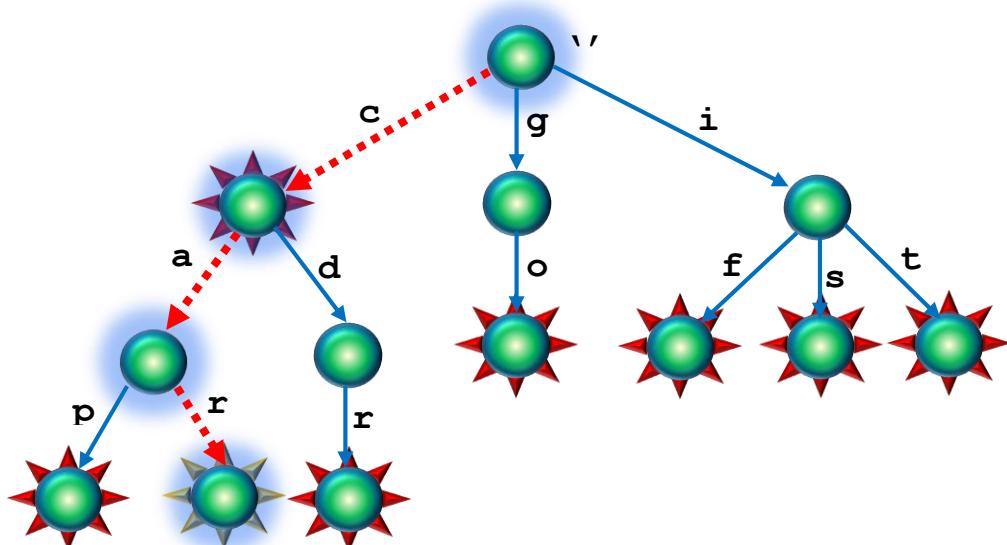
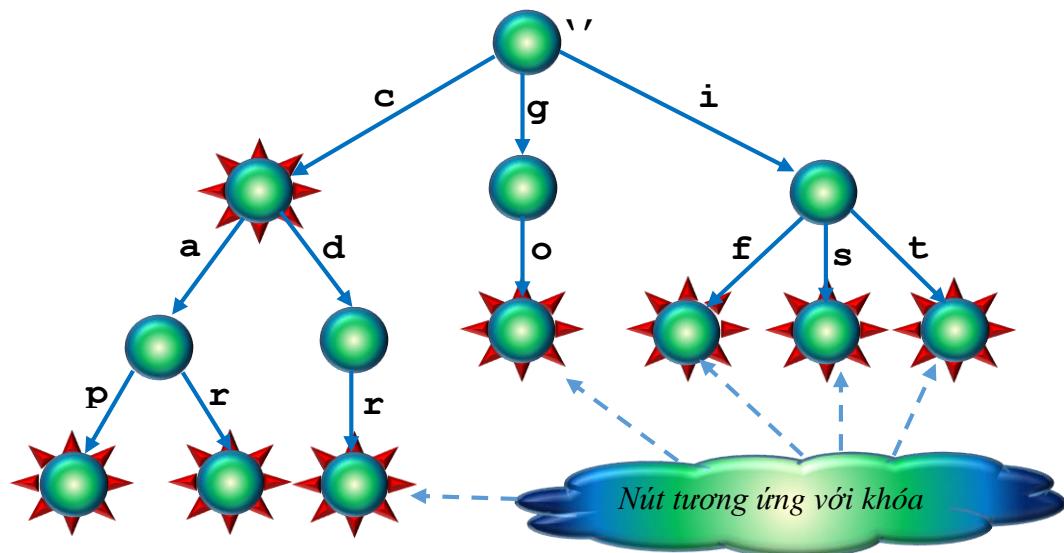
clock_t bb=clock();
fo<<"\nTime: "<<(double)(bb-aa)/1000<<" sec";
}
```



Cây tiền tố

Cây tiền tố (*Trie*) là cấu trúc dữ liệu cho phép lưu trữ các cặp (*khóa – giá trị*), trong đó khóa không lưu trữ tường minh mà được xác định dựa vào đường đi khi duyệt cây. Khóa thông thường là xâu ký tự. Tuy vậy vai trò của khóa có thể là đối tượng có cấu trúc bất kỳ, tức là các đối tượng có giá trị phụ thuộc vào vị trí xuất hiện của phần tử. Tên tiếng Anh của cấu trúc này xuất phát từ *Retrieval* (Tìm kiếm, phản hồi).

Tại các nút của Trie người ta không lưu trữ khóa mà thay vào đó – là nhãn một ký tự. Bản khóa là đường đi khi duyệt cây từ gốc tới đích cần tìm. Thông thường nút gốc của Trie là nút với nhãn rỗng. Ví dụ, khi cần làm việc với các khóa **c**, **cap**, **car**, **cdr**, **go**, **if**, **is**, **it** cây tiền tố tương ứng sẽ có dạng:



Khóa **car** tương ứng với đường đi đánh dấu đậm trên cây:

Mỗi đường đi từ gốc tới nút nào đó của cây tương ứng với một khóa. Tuy vậy, khóa đó có thể không nằm trong tập cần xét (ví dụ **ca**). Vì vậy thông thường cần có mảng đánh dấu các nút mà đường đi từ gốc tới đó tương ứng với khóa cần xét. Một kỹ thuật thường dùng khác là bổ sung vào khóa một ký tự đặc biệt làm dấu hiệu kết thúc khóa. Nút tương ứng với ký tự đặc biệt này sẽ là nút lá của cây và chứa giá trị mà khóa xác định. Để thuận tiện trong việc tổ chức dữ liệu mảng giá trị thường được lưu trữ riêng và liên kết với nút lá tương ứng của Trie thông qua các cách móc nối phù hợp.

Rõ ràng mọi cây con của Trie cũng là một Trie. Để dàng thấy rằng mọi khóa tương ứng với nút cuối đường đi trong cây con để có phần đầu giống nhau và vì vậy có thể xác định đặc trưng cần tìm của tất cả các khóa có phần đầu chung giống nhau với độ phức tạp $O(m)$ trong đó m – độ dài phần chung của các khóa.

Ứng dụng của Trie:

- + Hỗ trợ tìm kiếm, soạn thảo: Dẫn xuất tất cả các từ hoặc xâu có chung phần đầu đã được đưa vào,



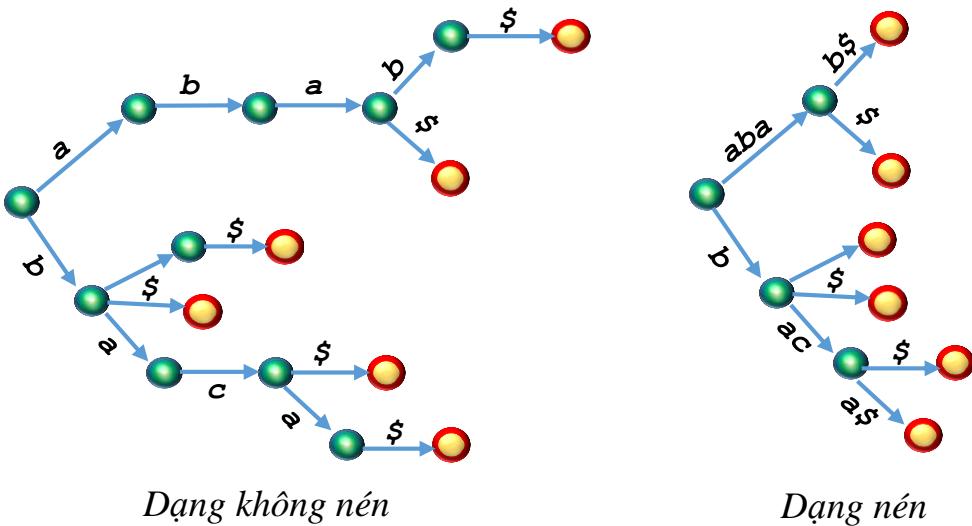
- + Kiểm tra chính tả trong các hệ soạn thảo,
- + Chọn cách ngắt từ khi xuống dòng,
- + Nhận dạng từ mới để bổ sung vào từ điển.

Các dạng tổ chức lưu trữ Trie:

- + **Dạng không nén:** Số nút ở nhánh tương ứng với số phần tử (độ dài) của khóa,
- + **Dạng nén:** Nhận được từ dạng không nén bằng cách xóa đi các nút trung gian chung trên đường đi dẫn tới lá,
- + **Dạng rút gọn (Patricia):** nhận được từ dạng không nén hoặc nén bằng cách loại bỏ các nút chỉ có một nút con.

Ví dụ, cần tổ chức Trie với các khóa **{abab\$, aba\$, bc\$, b\$, bac\$, baca\$}**, trong đó ký tự **\$** là dấu hiệu kết thúc khóa.

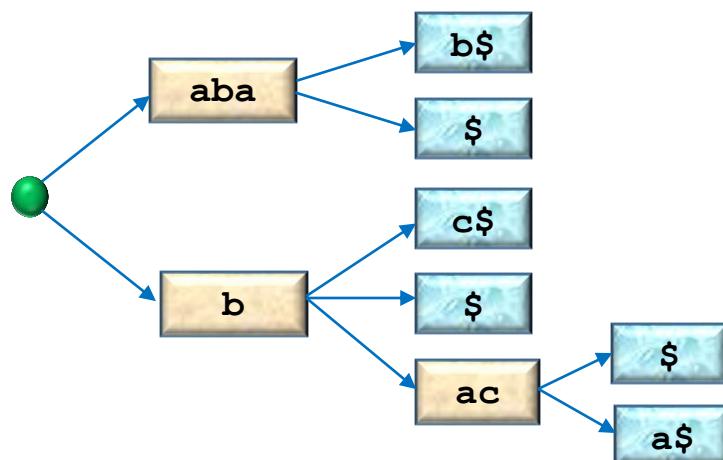
Cây tiền tố tương ứng có thể tổ chức ở dạng không nén hoặc nén như sau:



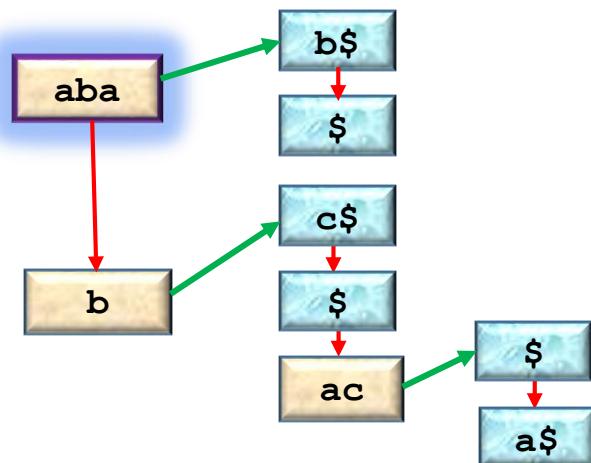
Biểu diễn Trie bằng con trỏ:

Cây có thể lưu trữ bằng cách chỉ ra quan hệ kè giữa 2 đỉnh có cạnh nối. Nhưng cách biểu diễn này không hiệu quả trong việc tìm đường đi từ gốc tới nút lá. Một trong những cách biểu diễn thích hợp để giảm độ phức tạp của các phép xử lý (tìm đường đi, bổ sung, loại bỏ) là móc nối bằng con trỏ. Thông tin trên cạnh được ghi ở đỉnh. Trừ nút gốc, các nút còn lại đều chỉ có một cung đi tới.

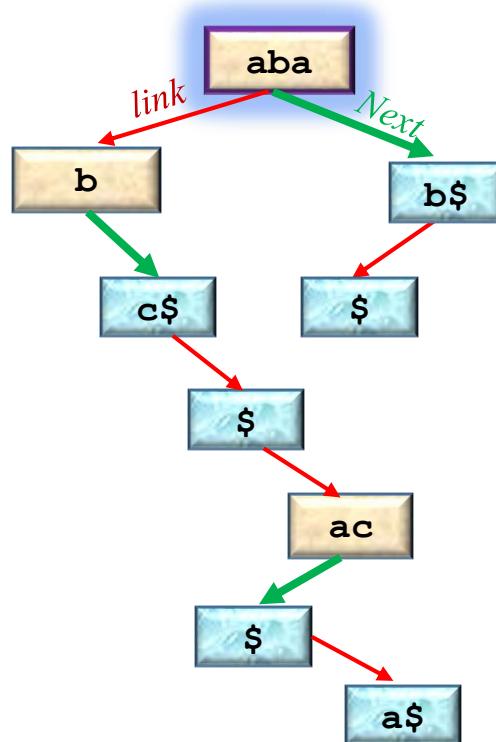
Cây ở hình trên sẽ có dạng:



Từ một đỉnh có thể có nhiều cung đi ra, vì vậy để thuận tiện xử lý cần tạo danh sách kè các các nút con của mỗi nút cha. Số lượng nút con thuộc một nút cha trực tiếp không nhiều. Nếu khóa là xâu ký tự la tinh thường thì số lượng nút con tối đa là 26. Với cách móc nối trên cây sẽ có dạng:



Như vậy Trie có thể biểu diễn dưới dạng **cây nhị phân!**



Mỗi nút của cây chứa đúng 2 con trỏ móc nối: **link** – chỉ tới nút đầu của danh sách nút con, **next** – chỉ tới nút cùng mức tiếp theo.

Tổ chức dữ liệu cho mỗi nút:

```

struct node
{
    char* key;
    int len;
    node* link;
    node* next;
    node(char* x, int n) : len(n), link(0), next(0)
    {
        key = new char[n];
        strncpy(key, x, n);
    }
    ~node() { delete[] key; }
};

```

Tìm kiếm:

Tìm tiền tố chung dài nhất giữa xâu **x** độ dài **n** và khóa **key** độ dài **k**:

```

int prefix(char* x, int n, char* key, int m)
{
    for( int k=0; k<n; k++)
        if( k==m || x[k] !=key[k] )
            return k;
    return n;
}

```

Tìm khóa **x** trong cây **t**:

```

node* find(node* t, char* x, int n=0)
{
    if( !n ) n = strlen(x)+1;
    if( !t ) return 0;
    int k = prefix(x,n,t->key,t->len);
        // Tìm ở danh sách các nút con
    if( k==0 ) return find(t->next,x,n);
    if( k==n ) return t;
        // Tìm cùng mức
    if( k==t->len ) return find(t->link,x+k,n-k);
    return 0;
}

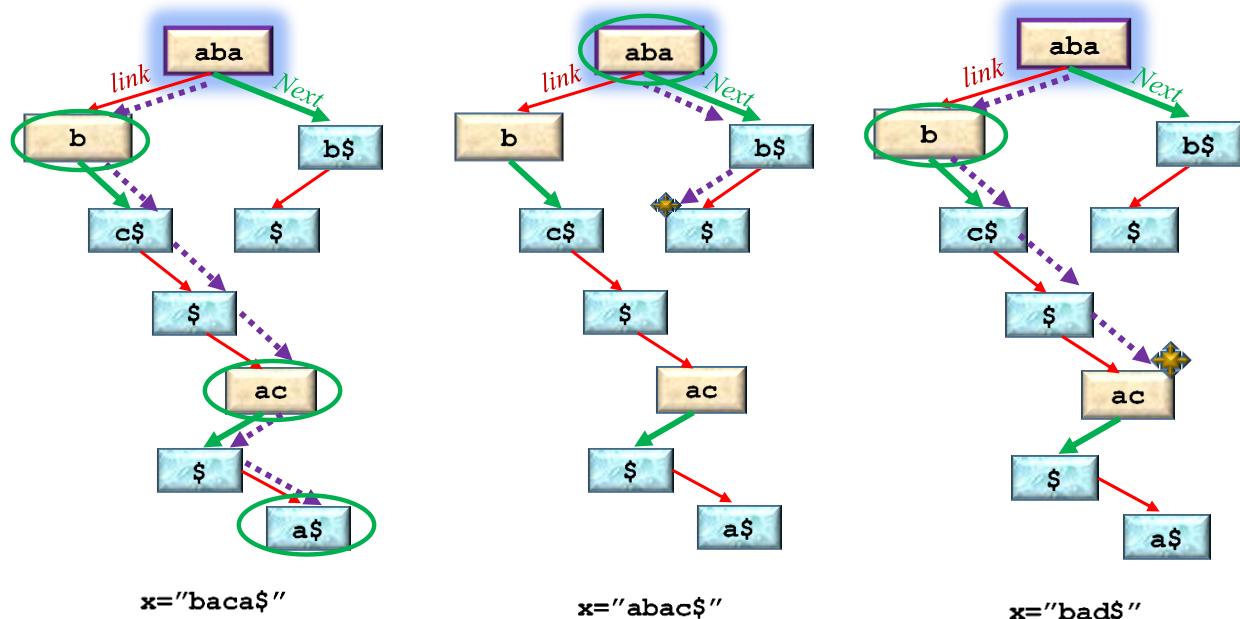
```

Có 3 trường hợp xảy ra:

- Tiền tố chung rỗng, khi đó phải chuyển theo **next** tìm ở nút cùng mức tiếp theo,
- Tiền tố chung tìm được bằng **x**, xác định được nút chứa giá trị,

- + Tiền tố chung trùng với khóa trong cây nhưng không trùng với **x**, cần theo **link** chuyển khóa khác trong cây.

Câu lệnh tìm kiếm: Độ dài **n** của khóa **x** được xác lập ngầm định là 0 và được tính trong hàm tìm kiếm. Vì vậy, trong câu lệnh tìm kiếm chỉ cần nêu cây t và khóa x cần tìm: **node* p=find(t, x)**, ví dụ **node* p=find(t, "baca\$")**; Nếu khóa **x** tồn tại trong cây thì hàm trên sẽ trả về con trỏ chỉ tới lá, nơi chứa thông tin về khóa **x**. Dưới đây là các bức tranh những nút của cây được duyệt cho 3 trường hợp cụ thể.

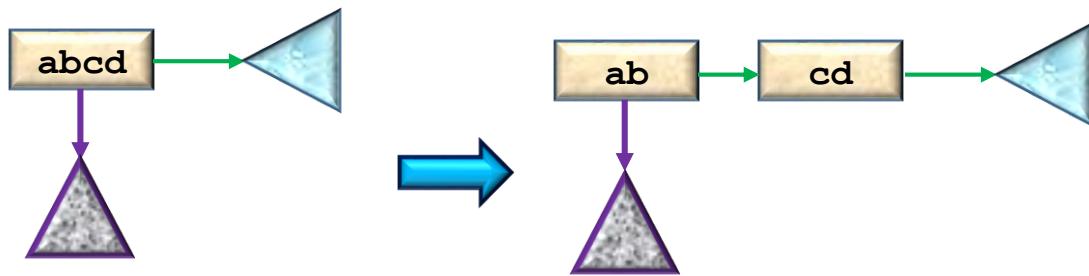


Bổ sung khóa mới:

Việc bổ sung thêm khóa vào cây Trie được thực hiện theo sơ đồ tương tự như tìm kiếm, nhưng có thêm một vài khâu xử lý bổ sung:

- + Trường hợp Trie rỗng (chưa có khóa nào được ghi nhận): Tạo nút chứa khóa đưa vào và cho con trỏ chỉ tới nút đó,
- + Trường hợp tiền tố chung của khóa lưu trong cây và **x** lớn hơn 0 nhưng nhỏ hơn độ dài của khóa (xem kết quả tìm kiếm thứ 2 nêu trên), khi đó cần tách nút tới được ra thành 2 nút, nút cha chứa phần chung, nút con **p** – chứa phần

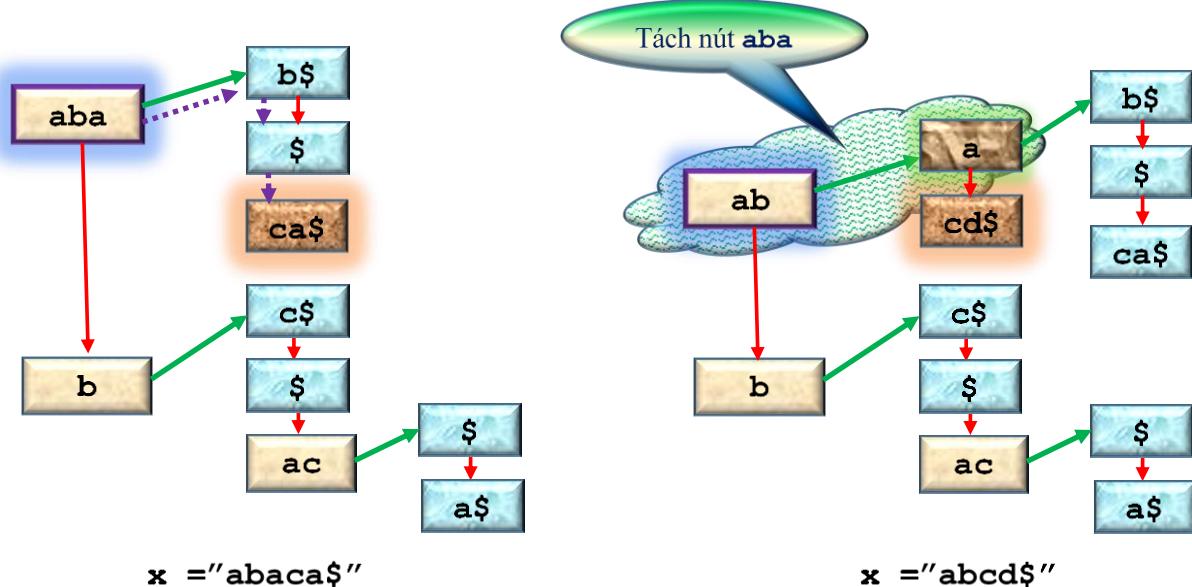
còn lại của khóa. Hàm **split** thực hiện phép xử lý này. Sau khi phân chia tiếp tục từ nút **p** việc bổ sung phần còn lại của **x** (đã loại bỏ phần tiền tố chung)



```
void split(node* t, int k)
{
    // Tách nút của t theo ký tự thứ k của khóa
    node* p = new node(t->key+k, t->len-k);
    p->link = t->link;
    t->link = p;
    char* a = new char[k];
    strncpy(a, t->key, k);
    delete [] t->key;
    t->key = a;
    t->len = k;
}
```

Hàm *split* tách nút:Hàm bổ sung khóa mới:

Ví dụ: Bổ sung 2 khóa **abaca\$** và **abcd\$** vào cây đã xét ở trên:



Lưu ý là nếu **x** đã có trong **t** thì sẽ không có thông tin nào được bổ sung, Trie *hoạt động như một tập hợp*.

```

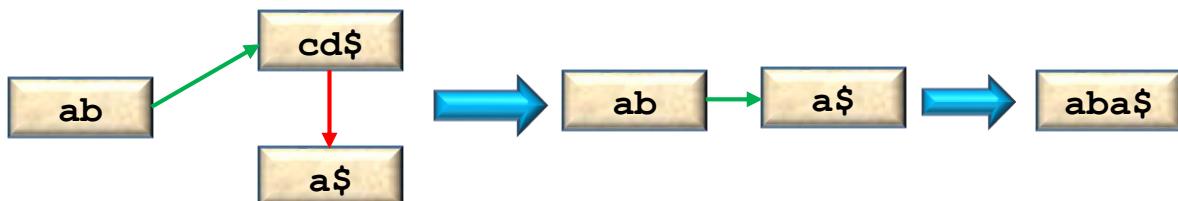
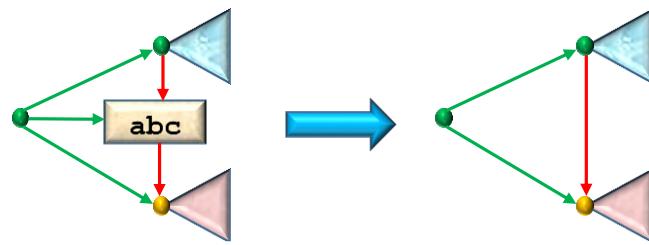
node* insert(node* t, char* x, int n=0)
// Bổ sung khóa x vào cây t
{
    if( !n ) n = strlen(x)+1;
    if( !t ) return new node(x,n);
    int k = prefix(x,n,t->key,t->len);
    if( k==0 ) t->next = insert(t->next,x,n);
    else if( k<n )
    {
        if( k<t->len ) // Tách nút hay không?
            split(t,k);
        t->link = insert(t->link,x+k,n-k);
    }
    return t;
}

```

Loại bỏ khóa:

Loại bỏ khóa thông thường là một công việc phức tạp, tuy vậy, đối với Trie thì không đáng ngại lắm vì chỉ cần loại bỏ có một nút lá tương ứng với hậu tố của khóa cần xóa. Trước hết cần tìm nút lá này, loại bỏ nó và đưa con trỏ tương ứng chỉ sang nút tiếp theo bên cạnh cùng mức.

Tuy vậy có thể xuất hiện trường hợp có 2 nút **t** và **p** trong đó t nhận p là



nút con duy nhất. Để đảm bảo Trie vẫn ở dạng nén cần hợp nhất 2 nút này bằng phép nối **join**.

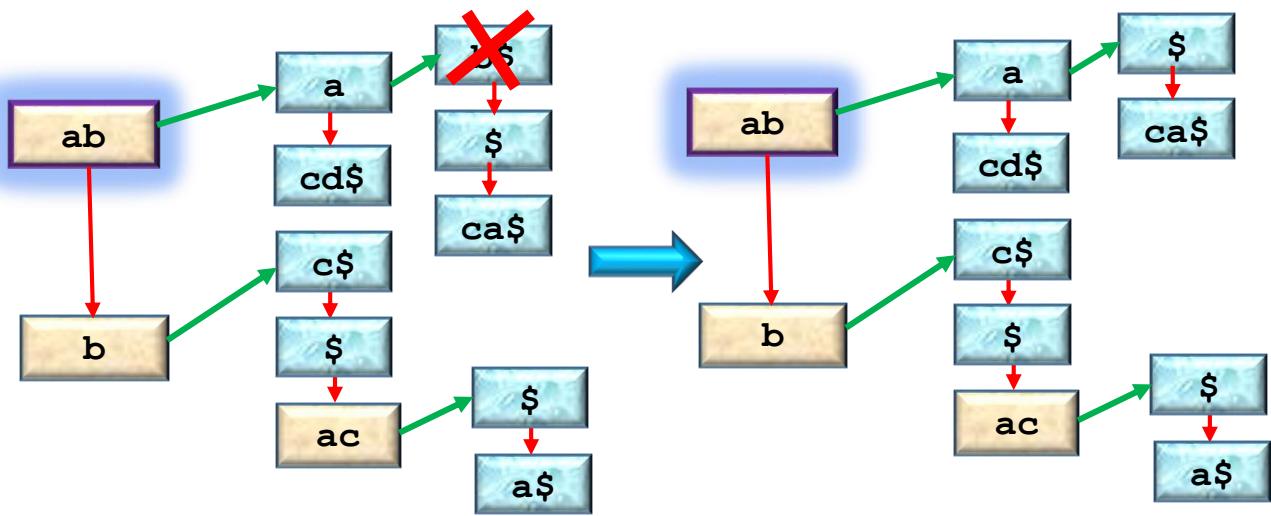
Hợp nhất nút t với t -> link: Gồm 2 công việc:Xóa khóa theo con trỏ **link** (*không*

```
void join(node* t) // Hợp nhất t và t->link
{
    node* p = t->link;
    char* a = new char[t->len+p->len];
    strncpy(a, t->key, t->len);
    strncpy(a+t->len, p->key, p->len);
    delete[] t->key;
    t->key = a;
    t->len += p->len;
    t->link = p->link;
    delete p;
}
```

phải theo next!).Ở phần tử ứng với **link** mới sẽ không chứa **next**.Hàm **remove** xóa khóa **x** khỏi Trie:

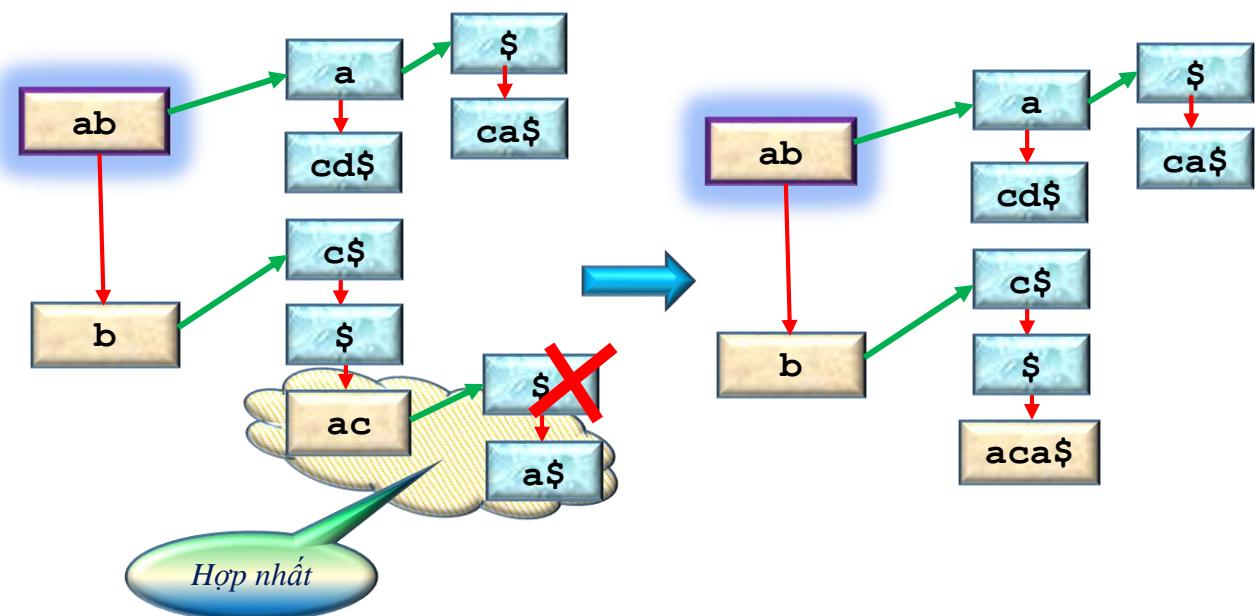
```
node* remove(node* t, char* x, int n=0)
// Xóa khóa x ở cây tiền tố t
{
    if( !n ) n = strlen(x)+1;
    if( !t ) return 0;
    int k = prefix(x,n,t->key,t->len);
    if( k==n ) // Xóa lá
    {
        znode* p = t->next;
        delete t;
        return p;
    }
    if( k==0 ) t->next = remove(t->next, x, n);
    else if( k==t->len )
    {
        t->link = remove(t->link, x+k, n-k);
        if( t->link && !t->link->next )
        // Kiểm tra nút t có một con duy nhất?
            join(t);
    }
    return t;
}
```

Ví dụ: Xóa khóa không hợp nhất nút:



Xóa khóa $x = "ababs\$"$

Xóa có hợp nhất nút:



Xóa khóa $x = "bac\$"$

Lưu ý:

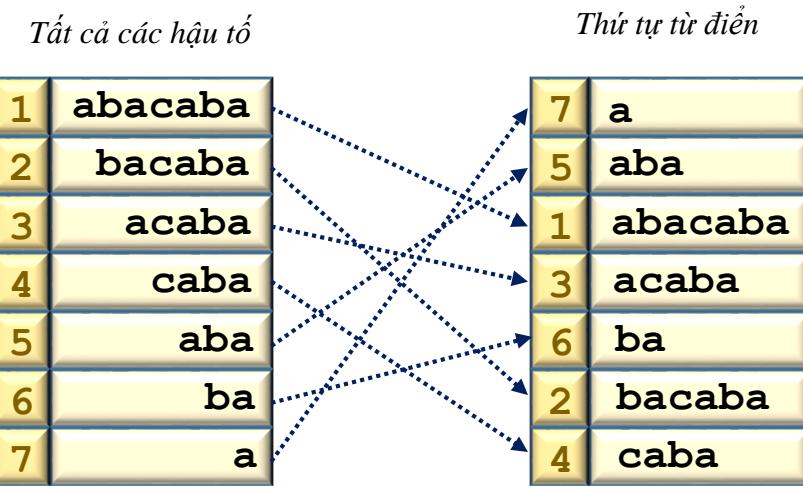
- ❖ Ngoài cây tiền tố còn có lý thuyết tổ chức và xử lý cây hậu tố,
- ❖ Chỉ cần nắm vững một trong hai loại cấu trúc nói trên vì có thể đưa bài toán ứng dụng cây tiền tố về bài toán sử dụng cây hậu tố và ngược lại,
- ❖ Nên nắm được lý thuyết chung về Cây tiền tố/Cây hậu tố để **định hướng** thiết kế dữ liệu và xử lý,
- ❖ Ngoài một số lĩnh vực ứng dụng đã liệt kê ở trên, việc triển khai ứng dụng trực tiếp các loại cây nói trên không mang lại hiệu quả ưu việt rõ ràng so với việc sử dụng các loại cấu trúc cây truyền thống khác!
- ❖ Để ứng dụng Trie một cách có hiệu quả nên chọn các biểu diễn cây trong đó **không đòi hỏi tổ chức các mốc nối link và next một cách tường minh** độ phức tạp của giải thuật và độ phức tạp lập trình sẽ giảm một cách đáng kể,
- ❖ Có thể sử dụng độ dài khóa để nhận dạng kết thúc, trong trường hợp này thông thường người ta chuẩn hóa độ dài các khóa khi xử lý.

MẢNG HẬU TỐ

Định nghĩa

Mảng hậu tố (*suffix array*) của xâu $s[1..n]$ là mảng **suf** chứa các số nguyên trong phạm vi từ 1 đến n và thỏa mãn điều kiện $s[suf[i]..n]$ là hậu tố thứ i theo thứ tự từ điển trong số các hậu tố khác rỗng của s .

Ví dụ: $s = abacaba$



Mảng hậu tố của s sẽ là [7, 5, 1, 3, 6, 2, 4].

Khôi phục xâu theo mảng hậu tố

Xét bài toán: Cho biết mảng hậu tố của xâu s . Hãy khôi phục xâu s với thời gian xử lý bậc $O(|s|)$.

Phương án bảng chữ cái vô hạn:

Vì bảng chữ cái là vô hạn nên ta có thể đặt hậu tố thứ i theo thứ tự từ điển tương ứng với chữ cái thứ i trong bảng chữ cái.

Dễ dàng thấy rằng đó là lời giải bài toán, vì nếu sắp xếp các hậu tố theo thứ tự từ điển thì những chữ cái đầu tiên sẽ đứng ở các vị trí tương ứng với vị trí trong bảng chữ cái.

Với n đã biết có thể dẫn xuất xâu s :

```
string s(n,'1'); // Khởi tạo độ dài cho s
for(int i=0;i<n;+i)s[suf[i]]=alphabet[i];
```

Phương án bảng chữ cái nhỏ nhất có thể:

Ban đầu sử dụng bảng chữ cái vô hạn ta có xâu trung gian **tmp** theo giải thuật đã nêu ở trên. Xét hậu tố thứ i theo thứ tự từ điển (điều này cũng có nghĩa là ký tự thứ i của xâu). Ký tự đầu tiên của nó sẽ bằng ký tự đầu tiên trong hậu tố trước đó theo

thứ tự từ điển nếu $\text{tmp}[\text{suf}[i-1]+1] < \text{tmp}[\text{suf}[i]+1]$, tức là 2 xâu con này vẫn giữ nguyên thứ tự từ điển nếu không có ký tự đầu.

Ví dụ: Cho mảng hậu tố **suf** = [7, 5, 1, 3, 6, 2, 4].

| Số thứ tự của hậu tố | Ký tự trong bảng chữ cái vô hạn | Ký tự trong bảng chữ cái hữu hạn |
|----------------------|---------------------------------|----------------------------------|
| 7 | 1 | 1 |
| 5 | 2, 5, 1 | 1 |
| 1 | 3, 6, 4, 7, 2, 5, 1 | 1 |
| 3 | 4, 7 2, 5, 1 | 1 |
| 6 | 5, 1 | 2 |
| 2 | 6, 4 7, 2, 5, 1 | 2 |
| 4 | 7, 2 5, 1 | 3 |

Các vòng tròn đánh dấu nơi phải bổ sung ký tự mới.

Hàm xác định xâu trong bảng chữ cái tối thiểu:

```

void fromSuffixArrayToString()
{
    string ab="#abcdefghijklmnopqrstuvwxyz";
    int tmp[maxlen], cur, j, k;
    sf="";
    for(int i=1; i<n; ++i) sf=sf+ab[i];
    for(int i=1; i<n; ++i) tmp[p[i]]=i;
    cur=1;
    sf[p[1]]=ab[1];
    for(int i=2; i<n; ++i)
    {
        j=p[i-1];
        k=p[i];
        if(tmp[j+1]>tmp[k+1]) ++cur;
        sf[p[i]]=ab[cur];
    }
}

```

Tính đúng đắn của giải thuật trên có thể dễ dàng chứng minh bằng phương pháp phản chứng.

Xây dựng mảng hậu tố

Giải thuật tầm thường

Từ xâu **s** độ dài **n** đã cho xây dựng mảng $\{(x_i, i)\}$, trong đó x_i là hậu tố thứ **i** của **s** (xâu con chứa $n-i$ ký tự cuối của **s**), $i = 0 \div n-1$, sắp xếp dãy nhận được theo thứ tự tăng dần của các hậu tố. Vị trí của cặp (x_i, i) trong dãy đã sắp xếp chính là giá trị cần tìm tương ứng với hậu tố x_i . Việc sắp xếp *có thể thực hiện không tưởng minh* bằng cấu trúc dữ liệu hàng đợi ưu tiên (**priority_queue**) hoặc tập hợp (**set**).

Lưu ý: Để thuận tiện cho các xử lý tiếp theo trong C, trong nhiều trường hợp giá trị các phần tử của mảng hậu tố được xác định trong khoảng $[0, n-1]$.

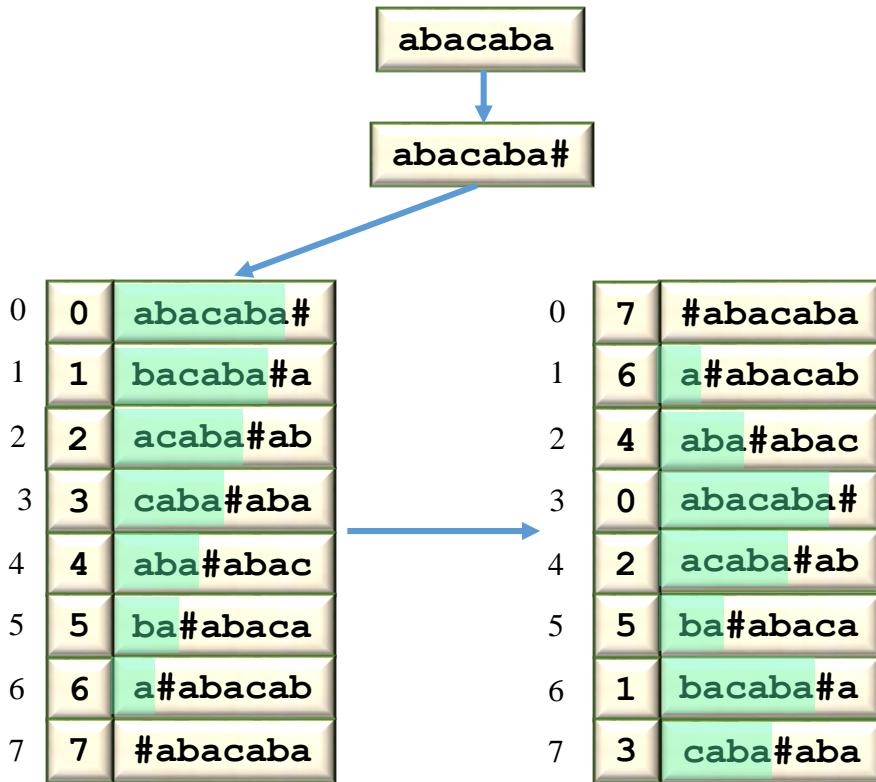
Độ phức tạp của giải thuật là $O(n\log n)$, dễ lập trình, tuy vậy đòi hỏi nhiều bộ nhớ (để lưu các xâu hậu tố) vì vậy *không thích hợp với n đủ lớn*.

```
#include <bits/stdc++.h>
#define NAME "suffix_arr."
using namespace std;
typedef pair<string, int> psi;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
string s=, sm;
set<psi> suf;
int n,d[100000],l;
psi t;

int main()
{
    fi>>s;
    n=s.size(); l=1;
    for(int i=n-1;i>=0;--i)
    {
        sm=s.substr(i,l++);
        t=make_pair(sm,i);
        suf.insert(t);
    }
    l=0;
    for( auto&& ii:suf)
    {
        t=ii;d[l++]=t.second;
    }
    fo<<"n = "<<n<<'\\n';
    for(int i=0;i<n;++i) fo<<d[i]<<' ';
    fo<<"\\nTime: "<<clock()/(double)1000<<" sec";
}
```

Giải thuật sử dụng xây đầy vòng

Xâu con $s[i..j]$ đầy vòng của s khi $i > j$ là xâu $s[i..n-1]+s[0..j]$. Để thứ tự các xâu đầy vòng trong bảng sắp xếp tương ứng với bảng sắp xếp các xâu hậu tố cần thêm vào cuối xâu s một ký tự có thứ tự thấp hơn mọi ký tự tạo ra xâu s , ví dụ ký tự '#'. Ví dụ, với $s = abacaba$ ta có:



Vấn đề ở đây là phải *xây dựng sơ đồ xử lý đủ nhanh* và *không cần lưu trữ các xâu đầy vòng* của s .

Quá trình xử lý sẽ bao gồm $\log n$ bước, ở *bước thứ k* ($k = 0, 1, 2, \dots, \lceil \log n \rceil$) sẽ *sắp xếp các xâu con độ dài 2^k* . Ở bước cuối cùng – độ dài các xâu con được sắp xếp là $2^{\lceil \log n \rceil} > n$.

Trên thực tế *thường có các xâu con đầy vòng giống nhau* và thuật toán cần có thông tin này. Tập các xâu giống nhau được gọi là *lớp tương đương*. Các lớp tương đương được sắp xếp theo thứ tự từ điển tăng dần. Thứ tự của lớp tương đương trong bảng sắp xếp được gọi là chỉ số của lớp.

Ở mỗi bước, bên cạnh hoán vị $p[0..n-1]$ ghi nhận chỉ số của các xâu con đầy vòng ta còn lưu thêm c_i xác định *chỉ số lớp tương đương* của *xâu con đầy vòng độ dài 2^k , bắt đầu từ vị trí i*. Ngoài ra, chỉ số lớp tương đương cũng hỗ trợ việc sắp xếp: nếu một hậu tố nhỏ hơn hậu tố khác thì chỉ số lớp tương đương của nó cũng nhỏ hơn chỉ số lớp tương đương của hậu tố kia. Để thuận tiện, chỉ số lớp tương

đương được đánh số bắt đầu từ 0 và số lượng lớp tương đương được lưu trữ ở biến **classes**.

Ví dụ, với xâu **s = abacaba** ta sẽ làm việc với các xâu con đầy vòng của **abacaba#**.

Các xâu đầy vòng độ dài 2^k , $k = 0, 1, 2, 3$:

| | $k = 0$ | $k = 1$ | $k = 2$ | $k = 3$ |
|---|---------|---------|-----------|--------------|
| 0 | 7 # | 0 7 #a | 0 7 #aba | 0 7 #abacaba |
| 1 | 6 a | 1 6 a# | 1 6 a#ab | 1 6 a#abacab |
| 2 | 4 a | 2 4 ab | 2 4 aba# | 2 4 aba#abac |
| 3 | 0 a | 3 0 ab | 3 0 abac | 3 0 abacaba# |
| 4 | 2 a | 4 2 ac | 4 2 acab | 4 2 acaba#ab |
| 5 | 5 b | 5 5 ba | 5 5 ba#a | 5 5 ba#abaca |
| 6 | 1 b | 6 1 ba | 6 1 bac a | 6 1 bacaba#a |
| 7 | 3 c | 7 3 ca | 7 3 caba | 7 3 caba#aba |

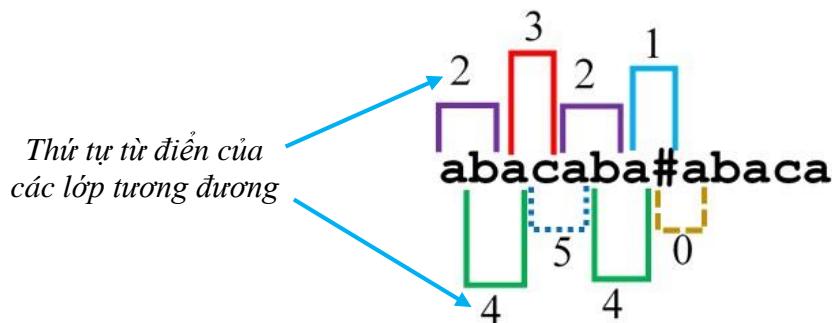
*Có thể có trình tự sắp xếp
(7, 6, 0, 4, 2, 5, 1, 3)*

Ta sẽ có:

| | |
|-------------------------------|----------------------------|
| 0: p=(7, 6, 4, 0, 2, 5, 1, 3) | c=(1, 2, 1, 3, 1, 2, 1, 0) |
| 1: p=(7, 6, 4, 0, 2, 5, 1, 3) | c=(2, 4, 3, 5, 2, 4, 1, 0) |
| 2: p=(7, 6, 4, 0, 2, 5, 1, 3) | c=(3, 6, 4, 7, 2, 5, 1, 0) |
| 3: p=(7, 6, 4, 0, 2, 5, 1, 3) | c=(3, 6, 4, 7, 2, 5, 1, 0) |

Mảng **p** có thể không đơn trị, ví dụ ở bước $k = 1$ **p** có thể nhận giá trị (7, 6, 0, 4, 2, 5, 1, 3), nhưng **c** *luôn luôn đơn trị*.

Trong ví dụ trên, các lớp tương đương ở bước $k = 1$ là:



Ở bước 0 ta phải sắp xếp các xâu con độ dài 1. Điều này có thể làm một cách đơn giản bằng cách tính tần số xuất hiện của ký tự khác nhau, từ đó xác định \mathbf{p} . Các ký tự giống nhau – thuộc cùng một lớp tương đương, các ký tự khác nhau – thuộc những lớp khác nhau.

Xét xâu chỉ chứa các ký tự trong bảng mã ASCII.

Tổ chức dữ liệu:

- Xâu **string** s – lưu xâu cần xử lý,
- Mảng **vector<int>** cnt – lưu tổng tiền tố tần số xuất hiện các ký tự,
- Mảng **vector<int>** p – lưu mảng hậu tố của xâu cần xử lý,
- Mảng **vector<int>** c – lưu chỉ số lớp tương đương.

Đoạn chương trình xử lý bước 0 có dạng:

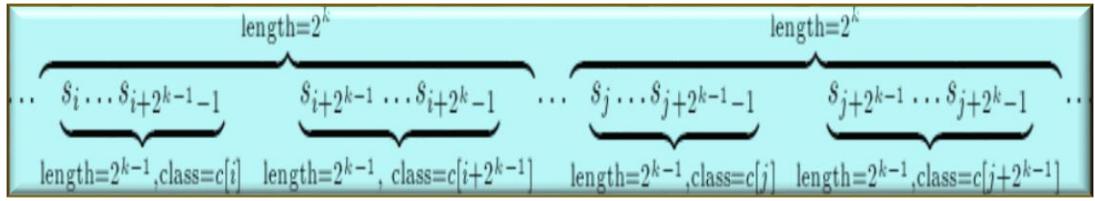
```
fi>>s; n=s.size(); s+='#'; ++n;
p.resize(n); cnt.assign(alphabet,0); c.resize(n);

for (int i=0; i<n; ++i) ++cnt[s[i]];
for (int i=1; i<alphabet; ++i) cnt[i] += cnt[i-1];
for (int i=0; i<n; ++i) p[--cnt[s[i]]] = i;

c[p[0]] = 0;
int classes = 1;
for (int i=1; i<n; ++i)
{
    if (s[p[i]] != s[p[i-1]]) ++classes;
    c[p[i]] = classes-1;
}
```

Giả thiết đã thực hiện $k-1$ bước (tức là đã tính \mathbf{p} và \mathbf{c} tương ứng). Xét việc xử lý bước k với độ phức tạp $O(n)$.

Độ dài xâu con đầy vòng ở bước này là 2^k , bao gồm hai phần, mỗi phần có độ dài 2^{k-1} và được đặc trưng bởi cặp giá trị ($\mathbf{c}[i], \mathbf{c}[i+2^{k-1}]$) xác định ở bước trước.



Toàn bộ thông tin về các xâu độ dài 2^k chứa trong cặp giá trị ($c[i], c[i+2^{k-1}]$), như vậy việc sắp xếp chúng được dẫn về sắp xếp các cặp giá trị nói trên.

Việc sắp xếp có thể thực hiện theo nhiều cách khác nhau. Dưới đây là một trong số các phương pháp đó. Giá trị các thành phần trong mỗi phần tử không vượt quá n nên ta có thể tính được vị trí của phần tử trong dãy kết quả. Đầu tiên ta sẽ tiến hành sắp xếp bộ phận theo giá trị của trường thứ 2 trong cặp, sau đó sắp xếp toàn dãy theo giải thuật sắp xếp ổn định (giữ nguyên trình tự bộ phận khi giá trị khóa sắp xếp bằng nhau).

Để có trình tự bộ phận ta chỉ cần tính $p_i - 2^{k-1}$, giá trị này xác định trình tự bộ phận – kết quả cần tìm khi sắp xếp theo trường thứ 2. Tiếp theo, cần thực hiện sắp xếp ổn định theo giá trị trường thứ nhất của cặp số.

Vấn đề còn lại là cập nhật mảng lớp tương đương c . Để cập nhật c cần duyệt mảng p mới nhận được, so sánh các giá trị cạnh nhau trong mảng.

```

vector<int> pn(n), cn(alphabet);
for (int h=0; (1<<h)<n; ++h)
{
    for (int i=0; i<n; ++i)
    {
        pn[i] = p[i] - (1<<h);
        if (pn[i] < 0) pn[i] += n;
    }
    cnt.assign(alphabet,0);
    for (int i=0; i<n; ++i) ++cnt[c[pn[i]]];
    for (int i=1; i<classes; ++i) cnt[i] += cnt[i-1];
    for (int i=n-1; i>=0; --i)p[--cnt[c[pn[i]]]] = pn[i];
    cn[p[0]] = 0;
    classes = 1;
    for (int i=1; i<n; ++i)
    {
        int mid1=(p[i]+(1<<h))%n, mid2=(p[i-1]+(1<<h))%n;
        if(c[p[i]]!=c[p[i-1]]||c[mid1]!=c[mid2]) ++classes;
        cn[p[i]] = classes-1;
    }
    c=cn;
}

```

Để xử lý ta cần thêm 2 mảng trung gian lưu các tham số phục vụ lặp.

Độ phức tạp của giải thuật: $O(n \log n)$. Khối lượng bộ nhớ trung gian cần thiết là $O(n)$.

Chương trình minh họa tính mảng hậu tố và khôi phục xâu từ mảng hậu tố:

```
#include <bits/stdc++.h>
#define NAME "suffix_arr."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
string s,sf; // input/output strings
int n; // length s
const int maxlen = 100100; // max size s
const int alphabet = 256; // alphabet length, <= maxlen
vector<int>p,cnt,c;

void fromSuffixArrayToString()
{
    string ab="#abcdefghijklmnopqrstuvwxyz";
    int tmp[maxlen],cur,j,k;
    sf="";
    for(int i=1;i<n;++i) sf=sf+'a';
    for(int i=1;i<n;++i) tmp[p[i]]=i;
    cur=1;
    sf[p[1]]=ab[1];
    for(int i=2;i<n;++i)
    {
        j=p[i-1];
        k=p[i];
        if(tmp[j+1]>tmp[k+1]) ++cur;
        sf[p[i]]=ab[cur];
    }
}

int main()
{
    fi>>s; n=s.size();s+='#';++n;
    p.resize(n);cnt.assign(alphabet,0);c.resize(n);
    for (int i=0; i<n; ++i)++cnt[s[i]];
    for (int i=1; i<alphabet; ++i)cnt[i] += cnt[i-1];
    for (int i=0; i<n; ++i)p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;

    for (int i=1; i<n; ++i)
    {
        if (s[p[i]] != s[p[i-1]]) ++classes;
        c[p[i]] = classes-1;
    }

    vector<int> pn(n), cn(alphabet);
    for (int h=0; (1<<n; ++h)
    {

```

```

for (int i=0; i<n; ++i)
{
    pn[i] = p[i] - (1<<h);
    if (pn[i] < 0) pn[i] += n;
}
cnt.assign(alphabet, 0);
for (int i=0; i<n; ++i) ++cnt[c[pn[i]]];
for (int i=1; i<classes; ++i) cnt[i] += cnt[i-1];
for (int i=n-1; i>=0; --i)p[--cnt[c[pn[i]]]] = pn[i];
cn[p[0]] = 0;
classes = 1;
for (int i=1; i<n; ++i)
{
    int mid1=(p[i+(1<<h))%n, mid2=(p[i-1]+(1<<h))%n;
    if(c[p[i]]!=c[p[i-1]] || c[mid1]!=c[mid2]) ++classes;
    cn[p[i]] = classes-1;
}
c=cn;
}

fo<<"n = "<<n<<endl;
for(int i:p) fo<<i<<' ' ; fo<<endl;
fromSuffixArrayToString();
fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}

```

Input:

abacaba

Output:

```

n = 8
7 6 4 0 2 5 1 3

... abacaba

Time: 0.002 sec

```



Ứng dụng:

- ✚ Tìm xâu **p** dài nhất chứa 2 lần trong xâu **t** và không giao nhau,
- ✚ Tìm xâu con trong xâu cho trước,
- ✚ Tìm xâu đầy vòng nhỏ nhất,
- ✚ Tìm xâu con,
- ✚ Tìm tiền tố chung dài nhất (*LCP – Longest Common Prefix*) của 2 hậu tố đứng cạnh nhau theo thứ tự từ điển,
- ✚ Tìm tiền tố chung dài nhất của hai hậu tố liên tiếp theo thứ tự từ điển,
- ✚ Tính số xâu con khác nhau của một xâu với độ phức tạp của giải thuật $O(|s|\log|s|)$ (hoặc $O(|s|)$ nếu sử dụng LCP).

Tìm xâu con dài nhất xuất hiện 2 lần và không giao nhau trong xâu cho trước

Giả thiết xâu cho trước là **t**. Xây dựng mảng hậu tố và dựa vào đó – tính LCP. Gọi **s'** là chỉ số của hậu tố **s** trong mảng hậu tố.

Xét 2 hậu tố **i** và **j** của **t** có $i' \leq j'$. Nói xâu **s** tương ứng với các hậu tố **i** và **j** nếu nó bằng tiền tố chung dài nhất của các hậu tố này. Các hậu tố **i** và **j** được gọi là tương ứng với **s** nếu **s** tham gia 2 lần không giao nhau vào **t**, các hậu tố **i** và **j** tương ứng với vị trí điểm đầu của các lần xuất hiện **s**.

Với xâu **s** bất kỳ và 2 hậu tố tương ứng với nó xét 2 điều kiện:

$$\max(|i|, |j|) \geq \min(|i|, |j|) + |s| \quad (1)$$

$$|s| = \min\{lcp[k], k= i' \dots j'\} \quad (2)$$

Định lý:

Xâu **s** tham gia vào **t** 2 lần không giao nhau khi và chỉ khi thỏa mãn điều kiện (1).

Điều kiện cần: Nếu **s** tham gia vào **t** 2 lần không giao nhau thì trong 2 hậu tố **i** và **j** có ít nhất một hậu tố dài hơn hậu tố kia một đoạn là $|s|$, đó là điều cần chứng minh.

Điều kiện đủ: Điều kiện (1) được thực hiện nói lên rằng một hậu tố dài hơn hậu tố kia ít nhất là $|s|$. Cả 2 hậu tố đều bắt đầu bằng **s**, vì vậy **s** xuất hiện 2 lần trong **t** và không giao nhau.

Định lý:

Nếu **s** là xâu dài nhất tham gia vào **t** 2 lần thì nó thỏa mãn điều kiện (2).

Giả thiết điều đó không đúng, tức là $|s| < \min\{lcp[k], k= i' \dots j'\}$ (nó không thể lớn hơn!). Từ đó suy ra $|s|$ nhỏ hơn độ dài tiền tố chung lớn nhất của các hậu tố **i** và **j**, điều không thể xảy ra theo cách xây dựng **i** và **j**!

Giải thuật:

Giải thuật tầm thường:

- Xây dựng mảng hậu tố, từ đó tính LCP,
- Duyệt và tìm tất cả các cặp **i** và **j** thỏa mãn các điều kiện (1) và (2), chọn trong số đó xâu có độ dài lớn nhất.

Độ phức tạp của giải thuật là $O(n^3 + ts)$ hoặc $O(n^2 + ts)$ tùy cách triển khai giải thuật. Ở đây **ts** là thời gian xây dựng mảng hậu tố.

Giải thuật tối ưu:

❖ Tư tưởng giải thuật:

Duyệt mọi xâu con **s** xuất hiện 2 lần trong **t** và thỏa mãn điều kiện 2 với mọi hậu tố **i** và **j** tương ứng với 2 lần xuất hiện bất kỳ của **s** trong **t** (tức là có thể giao nhau). Với mỗi **s** tìm được cỗ găng xác định **i** và **j** thỏa mãn điều kiện (1). Như vậy tất cả các xâu thỏa mãn đồng thời các điều kiện (1) và (2) đều được xét, từ đó dễ dàng nhận được kết quả cần tìm.

Lưu ý là các xâu **s** cần tìm là tiền tố của các hậu tố **k** độ dài **lcp[k]**. Cấu trúc dữ liệu stack sẽ hỗ trợ có hiệu quả cho việc tìm các hậu tố **i** và **j** đối với mỗi xâu **s** được xử lý.

❖ Giải thuật:

- ✚ Duyệt mảng hậu tố đã sắp xếp theo thứ tự từ điển, lưu vào stack các hậu tố **k** đã xét độ dài **lcp[k']** (tức là các xâu **s**) theo chiều tăng dần của độ dài. Đối với mỗi xâu trong stack – lưu hậu tố **i** độ dài ngắn nhất và hậu tố **j** độ dài lớn nhất. Ký hiệu **st** là đỉnh của stack và **s** – hậu tố đang xét.
- ✚ Phân biệt 3 trường hợp:
 - $|st| = lcp[s'] \rightarrow$ cập nhật **i** và **j** gắn với đỉnh stack,
 - $|st| \geq lcp[s'] \rightarrow$ nạp vào stack đỉnh mới tính **i**, **j** đối với nó,
 - $|st| \leq lcp[s'] \rightarrow$ loại bỏ giá trị đỉnh stack, gắn giá trị **i** và **j** của đỉnh bị loại bỏ cho đỉnh mới (để phục vụ tìm xâu độ dài lớn nhất).
- ✚ Nếu gặp **i** và **j** thỏa mãn điều kiện (1) \rightarrow cập nhật kết quả.

Đánh giá độ phức tạp của giải thuật: độ phức tạp giải thuật tính **lcp** là $O(n)$, với mỗi hậu tố cần $O(1)$ phép tính. Như vậy độ phức tạp chung sẽ là $O(n+ts)$, trong đó **ts** – thời gian xây dựng mảng hậu tố.

Tìm xâu đầy vòng nhỏ nhất

Giải thuật nêu trên thực hiện việc sắp xếp các xâu đầy vòng vì vậy **p₀** sẽ cho vị trí của xâu đầy vòng có thứ tự từ điển nhỏ nhất. Trong trường hợp này *không bô sung ký tự #* vào cuối xâu đã cho!

Độ phức tạp của giải thuật là $O(n \log n)$.

Chương trình minh họa:

```
#include <bits/stdc++.h>
#define NAME "suffix_arr."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
string s,ans;
int n; // length s
const int alphabet = 256; // alphabet length, <= maxlen
vector<int>p,cnt,c;

int main()
{
    fi>>s; n=s.size();
    p.resize(n);cnt.assign(alphabet,0);c.resize(n);
    for (int i=0; i<n; ++i)++cnt[s[i]];
    for (int i=1; i<alphabet; ++i)cnt[i] += cnt[i-1];
    for (int i=0; i<n; ++i)p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;

    for (int i=1; i<n; ++i)
    {
        if (s[p[i]] != s[p[i-1]]) ++classes;
        c[p[i]] = classes-1;
    }

    vector<int> pn(n), cn(alphabet);
    for (int h=0; (1<<h)<n; ++h)
    {
        for (int i=0; i<n; ++i)
        {
            pn[i] = p[i] - (1<<h);
            if (pn[i] < 0) pn[i] += n;
        }
        cnt.assign(alphabet,0);
        for (int i=0; i<n; ++i) ++cnt[c[pn[i]]];
        for (int i=1; i<classes; ++i)cnt[i] += cnt[i-1];
        for (int i=n-1; i>=0; --i)p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i=1; i<n; ++i)
        {
            int mid1=(p[i]+(1<<h))%n, mid2=(p[i-1]+(1<<h))%n;
            if(c[p[i]]!=c[p[i-1]] || c[mid1]!=c[mid2])++classes;
            cn[p[i]] = classes-1;
        }
    }
}
```

```

c=cn;
}

s+=s;
ans=s.substr(p[0],n);
fo<<ans;

fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}

```

Input: **abacaba**

Output: **aabacab**



Tìm xâu con

Cho văn bản **t**. Yêu cầu tìm vị trí xuất hiện của xâu **s** trong *chế độ online*, tức là **s** chỉ được biết trong quá trình thực hiện chương trình.

Xây dựng mảng hậu tố với văn bản **t**. Thời gian thực hiện sẽ là $O(|t|\log|t|)$. Việc này thực hiện độc lập với mục tiêu tìm kiếm là **s**. Xâu **s** (sau khi đã biết) sẽ phải là *tiền tố của một hậu tố* nào đó của **t**. Các hậu tố đã được sắp xếp (từ thông tin của mảng hậu tố) vì vậy việc xác định xâu chứa **s** có thể được thực hiện bằng phương pháp tìm kiếm nhị phân. Có nhiều kỹ thuật hỗ trợ việc so khớp **s** với tiền tố của hậu tố đang xét với độ phức tạp $O(1)$. Như vậy độ phức tạp của phần tìm kiếm sẽ là $O(\log|t|)$.

Giải thuật này áp dụng cho các *bài toán tương tác* người – máy (*interactive tasks*).

Ví dụ: Xét bài toán

Cho xâu **t** và **m** xâu **x₁**, **x₂**, ..., **x_m**. Các xâu đã cho chỉ chứa các ký tự la tinh thường và hoa. Với mỗi xâu **x_i** hãy xác định vị trí **k_i** trong **t** mà bắt đầu từ đó chứa **x_i** như một xâu con các ký tự liên tiếp, **i** = 1 ÷ **m**. Nếu có nhiều vị trí đồng thời thỏa mãn thì đưa ra vị trí tùy chọn. Nếu **x_i** không tham gia vào **t** như một xâu con thì đưa ra thông báo **No**. Các ký tự trong **t** được đánh số bắt đầu từ 0.

Dữ liệu: Vào từ file văn bản SUFF_ARR.INP:

- ✚ Dòng đầu tiên chứa xâu **t** độ dài không vượt quá 2×10^5 ,
- ✚ Dòng thứ 2 chứa số nguyên **m**,
- ✚ Dòng thứ **i** trong **m** dòng sau chứa xâu **x_i** độ dài không vượt quá 2×10^5 .

Kết quả: Đưa ra file văn bản SUFF_ARR.OUT giá trị hoặc thông báo tìm được ứng với các xâu \mathbf{x}_i , $i = 1 \div m$.

Ví dụ:

| SUFF_ARR.INP | SUFF_ARR.OUT |
|-----------------------|--------------|
| abacabaabadaba | |
| 2 | 1 |
| bac | No |
| abc | |



Chương trình:

```
#include <bits/stdc++.h>
#define NAME "suffix_arr."
#define Times fo<<"\nTime: "<<clock() / (double)1000<<" sec"
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
string s,x;
int n,m; // length s
const int alphabet = 256; // alphabet length, <= maxlen
vector<int>p,cnt,c;
bool flg;
int main()
{
    fi>>s; s+= '#'; n=s.size();
    p.resize(n); cnt.assign(alphabet,0); c.resize(n);
    for (int i=0; i<n; ++i)++cnt[s[i]];
    for (int i=1; i<alphabet; ++i)cnt[i] += cnt[i-1];
    for (int i=0; i<n; ++i)p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;

    for (int i=1; i<n; ++i)
    {
        if (s[p[i]] != s[p[i-1]]) ++classes;
        c[p[i]] = classes-1;
    }

    vector<int> pn(n), cn(n);
    for (int h=0; (1<<h)<n; ++h)
    {
        for (int i=0; i<n; ++i)
        {
            pn[i] = p[i] - (1<<h);
        }
    }
}
```

```

        if (pn[i] < 0) pn[i] += n;
    }
    cnt.assign(n, 0);
    for (int i=0; i<n; ++i) ++cnt[c[pn[i]]];
    for (int i=1; i<classes; ++i) cnt[i] += cnt[i-1];
    for (int i=n-1; i>=0; --i)p[~cnt[c[pn[i]]]] = pn[i];
    cn[p[0]] = 0;
    classes = 1;
    for (int i=1; i<n; ++i)
    {
        int mid1=(p[i]+(1<<h))%n, mid2=(p[i-1]+(1<<h))%n;
        if(c[p[i]]!=c[p[i-1]] || c[mid1]!=c[mid2])++classes;
        cn[p[i]] = classes-1;
    }
    c=cn;
}
fi>>m;
for(int i = 0; i<m; ++i)
{
    fi>>x; flg=false;
    int mx=x.size();
    int lf=1, rt=n, mid, imd;
    string u;
    while(rt-lf>1)
    {
        mid=(lf+rt)/2;
        imd=p[mid];
        u=s.substr(imd,mx);
        if(u==x) { fo<<imd<<'\n'; flg=true; break; }
        if(x<u) rt=mid; else lf= mid;
    }
    if(flg) continue;
    imd=p[lf];
    u=s.substr(imd,mx);
    if(u==x) fo<<imd<<'\n'; else fo<<"No\n";
}
Times;
}

```



Thực hiện các truy vấn so sánh hai xâu con

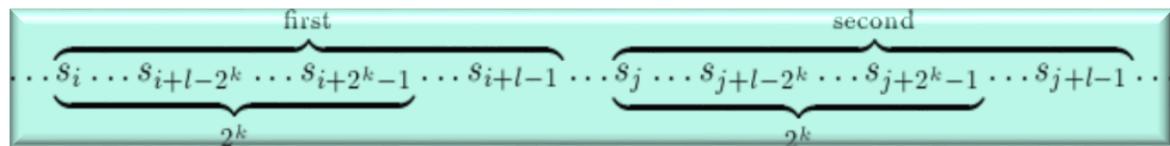
Cho xâu s và một số các truy vấn, mỗi truy vấn đòi hỏi cho kết quả so sánh 2 xâu con của s (đưa ra kết quả xâu con thứ nhất nhỏ hơn/bằng/lớn hơn xâu con thứ hai).

Sau khi xây dựng mảng hậu tố đối với s với chi phí thời gian $O(|s|\log|s|)$, mỗi truy vấn có thể thực hiện với chi phí thời gian $O(1)$.

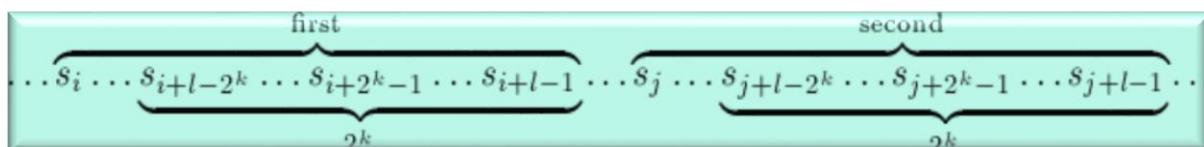
Trong quá trình xây dựng mảng hậu tố cần lưu lại các mảng lớp tương đương c ở mỗi bước. Điều này đòi hỏi chi phí bộ nhớ là $O(|s|\log|s|)$.

Với chi phí $O(1)$ ta có thể so sánh 2 xâu con độ dài 2^k bất kỳ từ thông tin về lớp tương đương.

Xét trường hợp tổng quát: hai xâu con độ dài m , bắt đầu từ vị trí i và j trong s . Ta có thể tìm k lớn nhất thỏa mãn điều kiện $2^k \leq m$. Để dàng so sánh 2 đoạn đầu độ dài 2^k và bắt đầu tương ứng từ i và j :



Nếu kết quả so sánh là bằng nhau thì so sánh 2 đoạn độ dài 2^k kết thúc tương ứng ở $i+m-1$ và $j+m-1$:



Việc tính k không phải là một chuyện khó khăn.

Nếu tính cả việc dẫn xuất k thì việc xử lý một truy vấn có độ phức tạp $O(\log m)$. Hàm so sánh trả về giá trị -1, 0 hoặc 1 tương ứng với kết quả so sánh là bé hơn, bằng hoặc lớn hơn:

```
int compare(int i, int j, int m)
{
    int k=0, t;
    while((1<<k)<=m)++k; t=1<<(--k);
    pair<int,int> a = make_pair(c[k][i],c[k][i+m-t]);
    pair<int,int> b = make_pair(c[k][j],c[k][j+m-t]);
    return a == b ? 0 : a < b ? -1 : 1;
}
```

Ví dụ: Xét bài toán

Cho xâu **s** chỉ chứa các ký tự la tinh thường và hoa. Các ký tự trong **s** được đánh số bắt đầu từ 1.

Xét q truy vấn, mỗi truy vấn có dạng **x y m** xác định 2 xâu con các ký tự liên tiếp nhau cùng độ dài **m** của **s**, bắt đầu từ ký tự thứ **x** và từ ký tự thứ **y**.

Với mỗi truy vấn hãy xác định kết quả so sánh xâu con thứ nhất với xâu con thứ 2 và đưa ra **-1**, **0** hoặc **1** tương ứng với kết quả nhận được là bé hơn, bằng hay lớn hơn.

Dữ liệu: Vào từ file văn bản SUFF_ARR.INP:

- ⊕ Dòng đầu tiên chứa xâu **s** độ dài không vượt quá 2×10^5 ,
- ⊕ Dòng thứ 2 chứa số nguyên **q**,
- ⊕ Dòng thứ **i** trong **q** dòng sau chứa 3 số nguyên **x, y** và **m** ($1 \leq x, y \leq n$, $0 < m$, $x+m, y+m \leq n$, trong đó **n** – độ dài xâu **s**).

Kết quả: Đưa ra file văn bản SUFF_OUT các kết quả so sánh, mỗi kết quả trên một dòng.

Ví dụ:

| SUFF_ARR.INP | SUFF_ARR.OUT |
|----------------|--------------|
| abacaba | -1 |
| 3 | 1 |
| 1 4 2 | 0 |
| 2 3 3 | |
| 1 5 3 | |



Chương trình:

```
#include <bits/stdc++.h>
#define NAME "suffix_arr."
#define Times fo<<"\nTime: "<<clock() / (double)1000<<" sec"
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
string s;
int n,m,n2,p2,q,x,y;
const int alphabet = 256;
vector<int> p,cnt;
vector<vector<int>> c;

int compare(int i,int j,int m)
```

```

int k=0,t;
while((1<<k)<=m)++k; t=1<<(--k);
pair<int,int> a = make_pair(c[k][i],c[k][i+m-t]);
pair<int,int> b = make_pair(c[k][j],c[k][j+m-t]);
return a == b ? 0 : a < b ? -1 : 1;
}

int main()
{
    fi>>s; s+='#'; n=s.size();
    n2=0;
    while((1<<n2)<=n)++n2;

    p.resize(n);cnt.assign(alphabet,0);

    c.resize(n2,vector<int>(n));

    for (int i=0; i<n; ++i)++cnt[s[i]];
    for (int i=1; i<alphabet; ++i)cnt[i] += cnt[i-1];
    for (int i=0; i<n; ++i)p[--cnt[s[i]]] = i;
    c[0][p[0]] = 0;
    int classes = 1;

    for (int i=1; i<n; ++i)
    {
        if (s[p[i]] != s[p[i-1]]) ++classes;
        c[0][p[i]] = classes-1;
    }

    vector<int> pn(n), cn(n);
    for (int h=0; h<n2-1; ++h)
    {
        for (int i=0; i<n; ++i)
        {
            pn[i] = p[i] - (1<<h);
            if (pn[i] < 0) pn[i] += n;
        }
        cnt.assign(n,0);
        for (int i=0; i<n; ++i) ++cnt[c[h][pn[i]]];
        for (int i=1; i<classes; ++i)cnt[i] += cnt[i-1];
        for (int i=n-1; i>=0; --i)p[--cnt[c[h][pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i=1; i<n; ++i)
        {
            int mid1=(p[i]+(1<<h))%n, mid2=(p[i-1]+(1<<h))%n;
            if(c[h][p[i]]!=c[h][p[i1]]||c[h][mid1]!=c[h][mid2])
                ++classes;
        }
    }
}

```

```

        cn[p[i]] = classes-1;
    }
    c[h+1]=cn;
}

fi>>q;
for(int i=0; i<q; ++i)
{
    fi>>x>>y>>m;
    fo<<compare(x-1,y-1,m)<<'\n';
}

Times;
}

```



Tìm tiền tố chung dài nhất của 2 xâu con (Phương pháp sử dụng bộ nhớ trung gian)

Cho xâu s và một số các truy vấn, mỗi truy vấn đòi hỏi tìm tiền tố chung dài nhất (Longest Common Prefix – LCP) 2 xâu con của s bắt đầu từ các vị trí i và j .

Giải thuật này đòi hỏi lưu các mảng c nhận được ở mỗi bước tính mảng hậu tố. Chi phí bộ nhớ cho việc lưu trữ này là $O(|s|\log|s|)$.

Với mỗi truy vấn i , j ta sẽ so sánh 2 tiền tố độ dài 2^k với k từ lớn đến bé, khi gặp trường hợp bằng nhau thì ghi nhận k và tìm tiếp tương tự các phần bằng nhau còn lại, bắt đầu từ $i+2^k$ và $j+2^k$.

```

int lcp (int i, int j)
{
    int ans = 0;
    //log_n - phần nguyên của log2n
    for (int k=log_n; k>=0; --k)
        if (c[k][i] == c[k][j])
    {
        ans += 1<<k;
        i += 1<<k;
        j += 1<<k;
    }
    return ans;
}

```

Chương trình minh họa xử lý q truy vấn, mỗi truy vấn yêu cầu xác định tiền tố chung dài nhất của 2 xâu con trong xâu s , bắt đầu tương ứng từ i và từ j , các ký tự của s được đánh số bắt đầu từ 1.

```

#include <bits/stdc++.h>
#define NAME "suffix_arr."
#define Times fo<<"\nTime: "<<clock() / (double)1000<<" sec"
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
string s;
int n,m,log_n,n2,p2,q,x,y;
const int alphabet = 256;
vector<int> p,cnt;
vector<vector<int>> c;

int lcp (int i, int j)
{
    int ans = 0;
    for (int k=log_n; k>=0; --k)
        if (c[k][i] == c[k][j])
    {
        ans += 1<<k;
        i += 1<<k;
        j += 1<<k;
    }
    return ans;
}

```

```

int main()
{
    fi>>s; s+='#'; n=s.size();
    n2=0;
    while( (1<<n2)<=n) ++n2; log_n=n2-1;

    p.resize(n); cnt.assign(alphabet,0);

    c.resize(n2,vector<int>(n));

    for (int i=0; i<n; ++i)++cnt[s[i]];
    for (int i=1; i<alphabet; ++i)cnt[i] += cnt[i-1];
    for (int i=0; i<n; ++i)p[--cnt[s[i]]] = i;
    c[0][p[0]] = 0;
    int classes = 1;

    for (int i=1; i<n; ++i)
    {
        if (s[p[i]] != s[p[i-1]]) ++classes;
        c[0][p[i]] = classes-1;
    }

    vector<int> pn(n), cn(n);
    for (int h=0; h<n2-1; ++h)
    {
        for (int i=0; i<n; ++i)
        {
            pn[i] = p[i] - (1<<h);
            if (pn[i] < 0) pn[i] += n;
        }
        cnt.assign(n,0);
        for (int i=0; i<n; ++i) ++cnt[c[h][pn[i]]];
        for (int i=1; i<classes; ++i)cnt[i] += cnt[i-1];
        for (int i=n-1; i>=0; --i)p[--cnt[c[h][pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i=1; i<n; ++i)
        {
            int mid1=(p[i]+(1<<h))%n, mid2=(p[i-1]+(1<<h))%n;
            if(c[h][p[i]]!=c[h][p[i-1]]||c[h][mid1]!=c[h][mid2])
                ++classes;
            cn[p[i]] = classes-1;
        }
        c[h+1]=cn;
    }

    fi>>q;
}

```

```

for (int i=0; i<q; ++i)
{
    fi>>x>>y;
    fo<<lcp(x-1, y-1)<< '\n';
}
Times;
}

```

| <i>Input</i> | <i>Output</i> |
|-----------------------|---------------|
| abacabaabadaba | 0 |
| 3 | 2 |
| 1 4 | 3 |
| 2 6 | |
| 1 5 | |

Tìm tiền tố chung dài nhất của 2 xâu con (Phương pháp không sử dụng bộ nhớ trung gian). Tìm tiền tố chung dài nhất của 2 hậu tố liên tiếp.

Giải thuật được xây dựng dựa trên cơ sở tìm LCP với mỗi cặp hậu tố liên tiếp nhau trong dãy sắp xếp theo thứ tự từ điển. Nói một cách khác, cần khởi tạo mảng **lcp[0..n-2]**, trong đó **lcp[i]** là tiền tố chung dài nhất của 2 hậu tố **p[i]** và **p[i+1]**. Tiền tố chung của 2 hậu tố bất kỳ không nhất thiết liên tiếp nhau có thể được dẫn xuất từ mảng này.

Với mỗi truy vấn **i, j** tìm vị trí **k₁** và **k₂** tương ứng trong mảng hậu tố. Không mất tính chất tổng quát, có thể coi **k₁ < k₂**. Khi đó kết quả truy vấn sẽ là min các phần tử của **lcp** trong khoảng **[k₁, k₂-1]**: việc chuyển từ hậu tố **i** đến hậu tố **j** có thể thực hiện bằng chuỗi chuyển tiếp từ một hậu tố tới hậu tố tới hậu tố tiếp theo trong dãy đã sắp xếp, bắt đầu từ **i** và kết thúc ở **j**. Đây là bài toán kinh điển tìm *min* trên một đoạn (*Range Minimum Query – RMQ*), có nhiều giải thuật khác nhau và có nhiều ứng dụng trong thực tế.

Việc xây dựng mảng **lcp** được thực hiện đồng thời trong quá trình tính mảng hậu tố cho các xâu con đầy vòng.

Ban đầu, **lcp** được khởi tạo với các giá trị bằng 0.

Giả thiết đã thực hiện **k-1** bước tính mảng hậu tố và nhận được **lcp** tương ứng. Xét việc cập nhật **lcp** ở bước thứ **k**. Khi tính mảng hậu tố ở bước thứ **k** xâu con đầy vòng độ dài **2^k** được chia thành 2 xâu có độ dài **2^{k-1}**. Kỹ thuật này cũng được dùng để cập nhật **lcp**. Sau khi cập nhật mảng hậu tố **p**, ta duyệt mảng, xét các cặp xâu con liên tiếp nhau **p[i]** và **p[i+1]**, **i = 0, 1, ..., n-2**. Chia mỗi xâu con thành 2

phân bằng nhau, bằng cách so sánh lớp tương đương \mathbf{c} tương ứng ở bước trước, phân biệt 2 trường hợp:

- ✚ Hai phần đầu của $\mathbf{p}[i]$ và $\mathbf{p}[i+1]$ khác nhau,
- ✚ Hai phần đầu của $\mathbf{p}[i]$ và $\mathbf{p}[i+1]$ bằng nhau.

Xét cách xử lý từng trường hợp.

Hai phần đầu khác nhau: Ở bước trước 2 phần đầu này phải đứng cạnh nhau trong dãy sắp xếp bởi vì lớp tương đương không thể biến mất (chỉ có thể xuất hiện mới). Do đó ở bước thứ \mathbf{k} , tất cả các xâu con khác nhau độ dài 2^{k-1} sẽ tạo ra các xâu con khác nhau độ dài 2^k với cùng một thứ tự như trước. Như vậy giá trị $\mathbf{lcp}[i]$ mới vẫn như cũ.

Hai phần đầu bằng nhau: Hai phần sau có thể giống nhau hoặc khác nhau và nếu khác nhau thì không nhất thiết phải liên tiếp nhau ở bước trước. Việc xác định $\mathbf{lcp}[i]$ dựa về việc xác định tiền tố chung dài nhất của 2 xâu con bất kỳ bằng cách giải bài toán RMQ trên đoạn tương ứng đối với \mathbf{lcp} cũ.

Như vậy, mỗi lớp tương đương xuất hiện mới sẽ kéo theo một phép xử lý RMQ. Số lớp tương đương có thể đạt tới n , vì vậy độ phức tạp của việc tìm kiếm \min có thể là $O(\log n)$ với sự hỗ trợ của cấu trúc dữ liệu [cây phân đoạn](#).

Độ phức tạp chung của giải thuật sẽ là $O(n \log n)$.

Số lượng xâu con khác nhau

Cho xâu \mathbf{s} . Hãy tính số lượng các xâu con khác nhau của \mathbf{s} .

Trước hết tính độ dài của tiền tố dài nhất đối với mỗi cặp hậu tố liên tiếp nhau theo thứ tự từ điển, trên cơ sở đó – tính số xâu con mới bắt đầu từ các vị trí $\mathbf{p}_0, \mathbf{p}_1, \dots$. Vì các hậu tố được sắp xếp theo thứ tự từ điển nên hậu tố \mathbf{p}_i đang xét sẽ cho tất cả các tiền tố mới của mình khác với các tiền tố do \mathbf{p}_{i-1} cung cấp ngoại trừ những xâu con thuộc tiền tố chung dài nhất, tức là \mathbf{lcp}_{i-1} xâu con đầu tiên.

Độ dài của hậu tố \mathbf{p}_i đang xét là $n - \mathbf{p}_i$ nên số lượng xâu con khác nhau mới sẽ là $n - \mathbf{p}_i - \mathbf{lcp}_{i-1}$. Riêng hậu tố đầu tiên (\mathbf{p}_0) không sản sinh các xâu con trùng lặp và cho số lượng xâu con là $n - \mathbf{p}_0$. Như vậy kết quả cần tính sẽ là

$$\sum_{i=0}^n (n - p[i]) - \sum_{i=0}^{n-1} \mathbf{lcp}[i]$$

Chương trình: Tính số lượng các xâu con khác nhau của s.

```
#include <bits/stdc++.h>
#define NAME "difsub_MH."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int maxlen = 500010;
const int alphabet = 256;
int n,l,k,m,len[100010],pl[maxlen],pr[maxlen];
string s,st;
vector<int> suff,ord,Lcp;
int64_t ans;

vector<int> buildLCP(string s,vector<int>& p)
{ int n, pos[maxlen];
  vector<int> lcp;
  n=s.size(); lcp.reserve(n);
  for(int i=0;i<n;++i) pos[p[i]]=i;
  int k=0;
  for(int i=0;i<n;++i)
  {
    if(k>0) --k;
    if(pos[i]==n-1) lcp[n-1]=-1, k=0;
    else
    {
      int j=p[pos[i]+1];
      while(max(i+k,j+k)<n && (s[i+k]==s[j+k])) ++k;
      lcp[pos[i]]=k;
    }
  }
  return lcp;
}

vector<int> calc_suff(string s)
{vector<int>p,c,cnt,Lcp;
 int n;
 p.reserve(maxlen);cnt.reserve(maxlen);
 c.reserve(maxlen);Lcp.reserve(maxlen);
 n=s.size();
 fill_n(cnt.begin(),maxlen,0);
 for (int i=0; i<n; ++i)++cnt[s[i]];
 for (int i=1; i<alphabet; ++i)cnt[i] += cnt[i-1];
 for (int i=0; i<n; ++i)p[--cnt[s[i]]] = i;
 c[p[0]] = 0;
 int classes = 1;
 for (int i=1; i<n; ++i)
 {
   if (s[p[i]] != s[p[i-1]]) ++classes;
   c[p[i]] = classes-1;
 }
 vector<int> pn(maxlen), cn(maxlen);
 for (int h=0; (1<<h)<n; ++h)
 {
   for (int i=0; i<n; ++i)
   {
     pn[i] = p[i] - (1<<h);
     if (pn[i] < 0) pn[i] += n;
   }
 }
```

```

    }
    fill_n(cnt.begin(), maxlen, 0);
    for (int i=0; i<n; ++i) ++cnt[c[pn[i]]];
    for (int i=1; i<classes; ++i) cnt[i] += cnt[i-1];
    for (int i=n-1; i>=0; --i)p[--cnt[c[pn[i]]]] = pn[i];
    cn[p[0]] = 0;
    classes = 1;
    for (int i=1; i<n; ++i)
    {
        int mid1 = (p[i]+(1<<h))%n, mid2=(p[i-1] + (1<<h))% n;
        if (c[p[i]]!=c[p[i-1]]||c[mid1]!= c[mid2]) ++classes;
        cn[p[i]] = classes-1;
    }
    c=cn;
}
return p;
}

int main()
{
    fi>>s;s+='#';
    ord.reserve(maxlen); Lcp.reserve(maxlen);
    n=s.size();
    suff=calc_suff(s);
    Lcp=buildLCP(s,suff);
    ans=n-suff[0];
    for(int i=1;i<n;++i)ans+=(n-suff[i]);
    for(int i=0;i<n;++i)ans-=Lcp[i];
    fo<<ans-n-1;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}

```



Tìm LCP của các hậu tố một xâu. Giải thuật Kasai.

Giải thuật *Kasai*, Arimura, Arikawa, Lee, Park cho phép với, độ phức tạp tuyến tính, tính độ dài tiền tố chung dài nhất (*Longest Common Prefix – LCP*) của tất cả các hậu tố một xâu, liên tiếp nhau theo thứ tự từ điển.

Ký hiệu

Các ký hiệu sau đây sẽ được sử dụng trong giải thuật:

- ✚ s – xâu cho trước,
- ✚ s_i – hậu tố của s bắt đầu từ ký tự thứ i ,
- ✚ suf – mảng hậu tố,
- ✚ suf^{-1} – mảng truy ngược của suf , nếu $suf[k] = i$ thì $suf^{-1}[i] = k$,
- ✚ $LCP(s_{suf[x]}, s_{suf[z]})$ – độ dài tiền tố chung dài nhất của 2 xâu $s_{suf[x]}$ và $s_{suf[z]}$,
- ✚ $lcp[i]$ – độ dài tiền tố chung dài nhất của 2 xâu liên tiếp nhau $i-1$ và i ,
 $lcp[i] = LCP(s_{suf[i-1]}, s_{suf[i]})$.

Tính chất của LCP

Tính chất 1:

$$LCP(S_{suf[y-1]}, S_{suf[y]}) \geq LCP(S_{suf[x]}, S_{suf[z]}), x < y \leq z$$

LCP của 2 hậu tố là min LCP tất cả các cặp hậu tố liên tiếp nhau theo thứ tự từ điển bắt đầu từ hậu tố thứ nhất và kết thúc bởi hậu tố thứ 2 đã nói:

$$LCP(S_{suf[x]}, S_{suf[z]}) = \min_{x < y \leq z} LCP(S_{suf[y-1]}, S_{suf[y]})$$

Từ đây suy ra LCP của *cặp hậu tố liên tiếp* nhau lớn hơn hoặc bằng LCP của *cặp hậu tố bao cặp hậu tố liên tiếp*. Ngoài ra, ta còn có:

$$LCP(S_{suf[x]}, S_{suf[z]}) = \min_{i=x+1 \dots z} lcp[i]$$

Tính chất 2:

Nếu

$$Suf^{-1}[Suf[x-1]+1] < Suf^{-1}[Suf[x]+1]$$

thì

$$LCP(S_{suf[x-1]}, S_{suf[x]}) > 1$$

Xét cặp hậu tố cạnh nhau trong **suf**. Nếu LCP của chúng lớn hơn 1 thì có thể xóa ký tự đầu trong mỗi hậu tố, thứ tự từ điển giữa 2 hậu tố này vẫn giữ nguyên, tức là xâu $\mathbf{s}_{\text{suf}[x]+1}$ vẫn đúng sau xâu $\mathbf{s}_{\text{suf}[x-1]+1}$.

Tính chất 3:

Nếu $LCP(S_{\text{suf}[x-1]}, S_{\text{suf}[x]}) > 1$ thì

$$LCP(S_{\text{suf}[x-1]+1}, S_{\text{suf}[x]+1}) = LCP(S_{\text{suf}[x-1]}, S_{\text{suf}[x]}) - 1$$

Trong trường hợp này LCP giữa $\mathbf{s}_{\text{suf}[x-1]+1}$ và $\mathbf{s}_{\text{suf}[x]+1}$ nhỏ hơn 1 so với LCP $\mathbf{s}_{\text{suf}[x-1]}$ và $\mathbf{s}_{\text{suf}[x]}$.

Ví dụ:

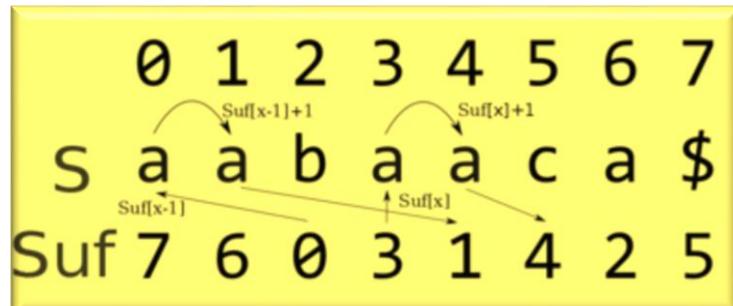
Xét xâu $\mathbf{s} = \mathbf{aabaaca\$}$

Mảng hậu tố của \mathbf{s} :

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|---|---|---|---|---|---|---|---|
| $Suf[i]$ | 7 | 6 | 0 | 3 | 1 | 4 | 2 | 5 |

Các hậu tố của \mathbf{s} :

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------|----|----|---|---|---|----|----|----|
| $Suf[i]$ | 7 | 6 | 0 | 3 | 1 | 4 | 2 | 5 |
| 0 | \$ | a | a | a | a | b | c | |
| 1 | | \$ | a | a | b | c | a | a |
| 2 | | | b | c | a | a | a | \$ |
| 3 | | | | a | a | a | \$ | c |
| 4 | | | | | a | \$ | c | a |
| 5 | | | | | | c | a | \$ |
| 6 | | | | | | | a | \$ |
| 7 | | | | | | | | \$ |



Bảng minh họa các tính chất 2 và 3

Mảng LCP:

| | | | | | | | | |
|----------|---------|---|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $lcp[i]$ | \perp | 0 | 1 | 2 | 1 | 1 | 0 | 0 |

$lcp[3] = 2$ là độ dài tiền tố chung **aa** dài nhất của 2 hậu tố
 $s_{suf[2]} = aabaaca\$$ và $s_{suf[3]} = aaca\$$.

Các nhận xét bổ trợ:

Xét bài toán sau: Hãy tính LCP giữa hậu tố s_i và hậu tố cạnh nó trong mảng **suf** khi biết giá trị LCP giữa s_{i-1} và hậu tố cạnh nó.

Để thuận tiện ta đặt $p = suf^{-1}[i-1]$ và $q = suf^{-1}[i]$, $j-1 = suf[p-1]$ và $k = suf[q-1]$.

Nói một cách đơn giản, ta muốn tính $lcp[q]$ khi biết $lcp[p]$.

Bổ đề:

Nếu $LCP(s_{j-1}, s_{i-1}) > 1$ thì $LCP(s_k, s_i) \geq LCP(s_j, s_i)$.

Chứng minh:

Vì $LCP(s_{j-1}, s_{i-1}) > 1$ nên từ tính chất 2, có $suf^{-1}[j] < suf^{-1}[i]$.

Vì $suf^{-1}[j] \leq suf^{-1}[k] = suf^{-1}[i]-1$, nên từ tính chất 1 có

$$LCP(s_k, s_i) \geq LCP(s_j, s_i)$$

Định lý:

Nếu $lcp[p] = LCP(s_{j-1}, s_{i-1}) > 1$ thì $lcp[q] = LCP(s_k, s_i) \geq lcp[p]-1$.

Chứng minh:

$LCP(s_k, s_i) \geq LCP(s_j, s_i)$ (theo bổ đề),

$LCP(s_j, s_i) = LCP(s_{j-1}, s_{i-1}) - 1$ (theo tính chất 3).

Điều này có nghĩa là $LCP(s_k, s_i) \geq LCP(s_{j-1}, s_{i-1}) - 1$.

Giải thuật

Hàm **buildLCP** tính mảng LCP dựa trên mảng hậu tố đã biết. Dựa vào định lý đã xét ở trên ta không cần so sánh tất cả các ký tự khi tính LCP giữa hậu tố s_i và hậu tố cạnh nó trong mảng **suf**. Để tính LCP tất cả các cặp hậu tố liên tiếp nhau trong mảng **suf** một cách có hiệu quả ta sẽ xét lần lượt tất cả các hậu tố, bắt đầu từ s_1 cho đến s_n .

Tham số đầu vào của hàm là xâu **s** đã được bổ sung vào cuối ký tự **\$** hoặc **#** và mảng hậu tố **suf**. Hàm trả về mảng **lcp**.

Độ phức tạp của giải thuật:

Đối với hậu tố đang xét, các ký tự được duyệt không phải từ đầu mà từ vị trí xác định trước, điều này cho phép xây dựng LCP với chi phí thời gian tuyến tính. Thật vậy, ở mỗi bước lặp giá trị LCP đang xử lý không thể nhỏ hơn giá trị trước đó quá một đơn vị. Như vậy tổng cộng LCP sẽ tăng không quá $2n$. Như vậy giải thuật tính LCP sẽ có độ phức tạp $O(n)$.

```
vector<int> buildLCP(string s, vector<int>& suf)
{ int n, pos[maxlen]; // pos[] - mảng truy ngược của suf
  vector<int>lcp;
  n=s.size(); lcp.reserve(n);
  for(int i=0;i<n;++i) pos[suf[i]]=i;
  int k=0;
  for(int i=0;i<n;++i)
  {
    if(k>0) --k;
    if(pos[i]==n-1) lcp[n-1]=-1, k=0;
    else
    {
      int j=p[pos[i]+1];
      while(max(i+k, j+k)<n && (s[i+k]==s[j+k])) ++k;
      lcp[pos[i]]=k;
    }
  }
  return lcp;
}
```



Ô TÔ MÁT HẬU TỐ

Ô tô mát hậu tố (Suffix Automata) còn được gọi là *Đồ thị từ có hướng không chứa chu trình* (Directed Acyclic Word Graph - DAWG). Ô tô mát hậu tố là một cấu trúc dữ liệu cho phép giải quyết một cách hiệu quả nhiều bài toán xử lý xâu. Ví dụ, ô tô mát hậu tố cho phép tìm các vị trí xuất hiện của một xâu trong xâu khác, tìm tất cả các xâu con khác nhau của một xâu có thể thực hiện với độ phức tạp $O(n)$.

Có thể coi ô tô mát hậu tố là bảng thông tin nén về tất cả các xâu con của một xâu. Độ nén thông tin là cực cao, với xâu độ dài n ô tô mát hậu tố chỉ cần dùng $O(n)$ bộ nhớ! Hơn thế nữa, bảng thông tin này được xây dựng với thời gian bậc $O(n)$. Chính xác hơn, nếu kích thước bảng chữ cái là k thì thời gian xây dựng là $O(n \log k)$.

Cách *lưu trữ* ô tô mát hậu tố với *chi phí bộ nhớ tuyến tính* được Blumer đề xuất năm 1983. Vào các năm 1985, 1986 Crochemure và Blumer công bố các *giải thuật đầu tiên xây dựng* ô tô mát hậu tố với chi phí *thời gian tuyến tính*.

Định nghĩa

Ô tô mát hậu tố của xâu s là một ô tô mát hữu hạn tiền định tối thiểu nhận tất cả các hậu tố của s .

Nếu đi sâu vào chi tiết, ta có thể hiểu ô tô mát hậu tố như sau:

- ⊕ Đó là một đồ thị có hướng không chứa chu trình, mỗi đỉnh là một trạng thái, mỗi cung là một phép chuyển trạng thái,
- ⊕ Tồn tại một trạng thái t_0 được gọi là trạng thái đầu, tương ứng với gốc của đồ thị. Từ t_0 có thể chuyển tới tất cả các trạng thái khác của ô tô mát,
- ⊕ Mỗi *phép chuyển* trong ô tô mát là một cung của đồ thị và được đánh dấu bằng một *ký tự*, các phép chuyển từ một trạng thái được đánh dấu bằng các ký tự khác nhau (tồn tại các trạng thái không có phép di chuyển từ đó tới trạng thái khác),
- ⊕ Một hoặc một số trạng thái được đánh dấu là *trạng thái kết thúc*, nếu di chuyển từ trạng thái đầu tới trạng thái kết thúc và ghi nhận các ký tự đánh dấu những cung đã đi qua ta được một xâu nhất thiết phải là một trong số các hậu tố của s ,
- ⊕ Ô tô mát hậu tố chứa *số lượng đỉnh ít nhất* trong số các ô tô mát thỏa mãn các yêu cầu nêu trên (không cần nhấn mạnh yêu cầu số phép chuyển là ít nhất vì đó là hệ quả của số lượng đỉnh ít nhất).

Các tính chất đơn giản nhất của ô tô mát hậu tố

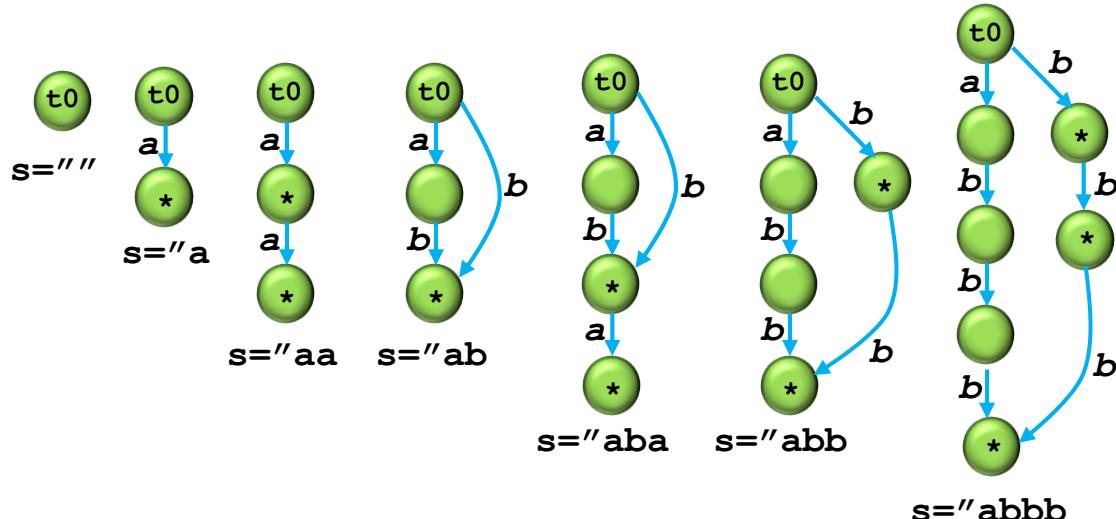
Tính chất đơn giản nhất và cũng là quan trọng nhất: chứa toàn bộ thông tin về tất cả các xâu con của xâu s .

Từ định nghĩa dễ dàng thấy rằng với đường đi bất kỳ xuất phát từ trạng thái đầu t_0 , nếu ghi lại các ký tự đánh dấu đường đi, ta sẽ được một xâu con của s và ngược lại, mọi xâu con của s đều tương ứng với một đường đi từ trạng thái đầu t_0 .

Để đơn giản ta sẽ dùng các thuật ngữ “*xâu tương ứng với đường đi*” hoặc “*đường đi tương ứng với xâu*”. Tới mỗi trạng thái của ô tô mát có một hoặc một số đường đi từ trạng thái đầu. Ta nói *trạng thái tương ứng với xâu* hoặc nhóm xâu tương ứng với các đường đi này.

Ví dụ về ô tô mát hậu tố

Trạng thái đầu được đánh dấu là ***t0***, trạng thái cuối (*trạng thái kết thúc*) được đánh dấu bằng ký tự *******.



Giải thuật xây dựng ô tô mát hâu tố với chi phí thời gian tuyển tính

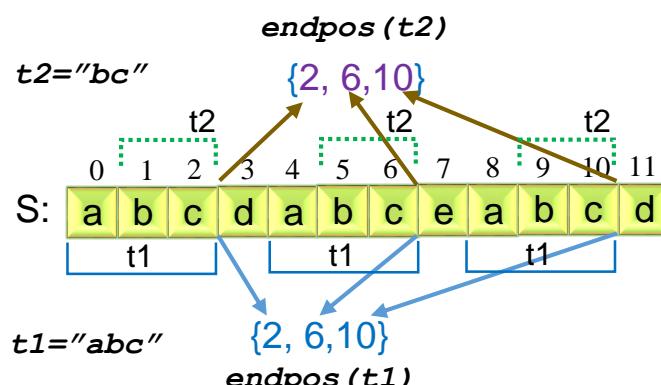
Trước hết ta sẽ xét một số khái niệm mới và chứng minh một số bô đề đơn giản nhưng rất quan trọng của ô tô mát hậu tố.

Các vị trí kết thúc endpos, tính chất và quan hệ của chúng với ô tô mát hậu tố

Xét xâu con khác rỗng t bất kỳ của xâu s . Tập kết thúc $\text{endpos}(t)$ là tập tất cả các vị trí trong xâu s tương ứng với vị trí kết thúc đánh dấu t tham gia vào s như một xâu con các ký tự liên tiếp nhau.

Hai xâu con t_1 và t_2 được gọi là ***endpos - tương đương*** nếu chúng có tập kết thúc trùng nhau:
 $\text{endpos}(t_1) = \text{endpos}(t_2)$.

Như vậy tất cả các xâu con khác rỗng của s có thể được phân



thành các *lớp tương đương* tương ứng với tập *endpos* của chúng.

Dễ dàng thấy rằng các lớp *endpos* – tương đương *tương ứng với cùng một trạng thái* của ô tô mát. Nói một cách khác – số trạng thái trong một ô tô mát hậu tố bằng số lượng lớp *endpos* – tương đương của các xâu con cộng thêm một trạng thái đầu. Mỗi trạng thái của ô tô mát hậu tố tương ứng với một hoặc một số xâu con có chung *endpos*.

Ví dụ: với $s = "abcdbc"$ ta có:

| Số TT | Xâu con | Endpos | Lớp tương đương |
|-------|---------|--------|-----------------|
| 1 | a | 0 | 0 |
| 2 | b | 1, 3 | 1 |
| 3 | c | 2, 4 | 2 |
| 4 | ab | 1 | 3 |
| 5 | bc | 2, 4 | 2 |
| 6 | cb | 3 | 4 |
| 7 | abc | 2 | 5 |
| 8 | bcb | 3 | 4 |
| 9 | cbc | 4 | 6 |
| 10 | abcb | 3 | 4 |
| 11 | bcdbc | 4 | 6 |
| 12 | abcdbc | 4 | 6 |

Ô tô mát hậu tố

Nhận xét trên sẽ được coi như *tiên đề*.

Xét một số tính chất đơn giản nhưng quan trọng liên quan tới giá trị *endpos*.

Bổ đề 1.

Hai xâu con khác rỗng u và w ($length(u) \leq length(w)$) là *endpos* – tương đương khi và chỉ khi xâu u xuất hiện trong s dưới dạng hậu tố của xâu w .

Chứng minh: Điều khẳng định trên gần như hiển nhiên. Một mặt, nếu u và w có cùng một trạng thái cuối thì có nghĩa u là hậu tố của w , mặt khác nếu u là hậu tố của w và chỉ của w mà thôi thì giá trị *endpos* của chúng phải bằng nhau theo định nghĩa.

Bổ đề 2.

Xét hai xâu con khác rỗng u và w ($\text{length}(u) \leq \text{length}(w)$). Tập endpos của chúng hoặc không giao nhau hoặc $\text{endpos}(w)$ nằm hoàn toàn trong $\text{endpos}(u)$ và điều này phụ thuộc vào việc u có phải là hậu tố của w hay không:

$$\begin{cases} \text{endpos}(w) \subset \text{endpos}(u) \text{ nếu } u \text{ là hậu tố của } w, \\ \text{endpos}(w) \cap \text{endpos}(u) = \emptyset, \text{ trong trường hợp ngược lại.} \end{cases}$$

Chứng minh: Giả thiết các tập $\text{endpos}(u)$ và $\text{endpos}(w)$ có ít nhất một phần tử chung, khi đó các xâu u và w kết thúc ở cùng một chỗ, điều này có nghĩa u là hậu tố của w . Nhưng như vậy mỗi lần w xuất hiện đều kéo theo sự xuất hiện u ở cuối, điều này có nghĩa tập $\text{endpos}(w)$ nằm gọn trong tập $\text{endpos}(u)$.

Bố đề 3.

Xét một lớp endpos – tương đương. Sắp xếp các xâu con thuộc lớp này theo chiều không tăng của độ dài. Khi đó mỗi xâu con trong dãy sẽ có độ dài ngắn hơn 1 so với xâu con đứng trước và là hậu tố của xâu trước. Nói một cách khác, *các xâu con thuộc một lớp tương đương là hậu tố của nhau và có độ dài là tất cả các giá trị khác nhau trong đoạn $[x, y]$ nào đó.*

Chứng minh: Xét một lớp endpos – tương đương. Nếu lớp tương đương này chỉ chứa một xâu thì kết luận ở bố đề là điều hiển nhiên.

Xét trường hợp có nhiều hơn một xâu. Theo bố đề 1, với 2 xâu khác nhau thuộc cùng một lớp endpos – tương đương, một xâu sẽ là hậu tố của xâu kia. Như vậy chúng không thể có cùng độ dài. Gọi w là xâu có độ dài lớn nhất và u – xâu có độ dài ngắn nhất. Theo bố đề 1, u là hậu tố của w . Xét một hậu tố bất kỳ của w với độ dài trong đoạn $[\text{length}(u), \text{length}(v)]$. Ta sẽ chứng minh là nó nằm trong lớp tương đương đang xét. Thật vậy, xâu này tham gia vào s dưới dạng là hậu tố của chỉ duy nhất w (vì hậu tố u ngắn hơn cũng chỉ tham gia duy nhất vào w). Như vậy, theo bố đề 1, hậu tố này có endpos – tương đương với xâu w và đó là điều phải chứng minh.

Móc nối hậu tố

Xét trạng thái $v \neq t0$ của ô tô mát. Trạng thái v sẽ tương ứng với lớp tương đương các xâu có cùng giá trị endpos . Gọi w là xâu có độ dài lớn nhất trong các xâu thuộc lớp tương đương này, các xâu còn lại của lớp tương đương sẽ là hậu tố của w . Xét các hậu tố theo chiều giảm dần của độ dài. Một số hậu tố đầu tiên của w sẽ thuộc cùng một lớp tương đương với w , các hậu tố còn lại (bao gồm hậu tố nhỏ nhất, hậu tố rỗng) – thuộc các lớp tương đương nào đó khác. Gọi t hậu tố đầu tiên trong số đó, ta sẽ tiến hành móc nối hậu tố tới nó.

Nói một cách khác, *móc nối hậu tố link(v)* chỉ tới trạng thái ứng với *hậu tố dài nhất của w nằm trong lớp endpos – tương đương khác*.

Ở đây ta coi trạng thái đầu t_0 thuộc một lớp tương đương riêng (chỉ chứa xâu rỗng) và coi $\text{endpos}(\mathbf{t}_0) = [-1 \dots \text{length}(\mathbf{s})-1]$.

Bổ đề 4.

Các mốc nối hậu tố tạo thành một cây với gốc là trạng thái đầu \mathbf{t}_0 .

Chứng minh: Xét trạng thái bất kỳ $\mathbf{v} \neq \mathbf{t}_0$. Mốc nối hậu tố $\text{link}(\mathbf{v})$ dẫn từ nó tới trạng thái của xâu có **độ dài nhỏ hơn** (suy từ định nghĩa mốc nối hậu tố và từ bô đê 3). Như vậy, nếu di chuyển theo các mốc nối hậu tố, sớm hay muộn ta cũng sẽ tới trạng thái \mathbf{t}_0 tương ứng với xâu rỗng.

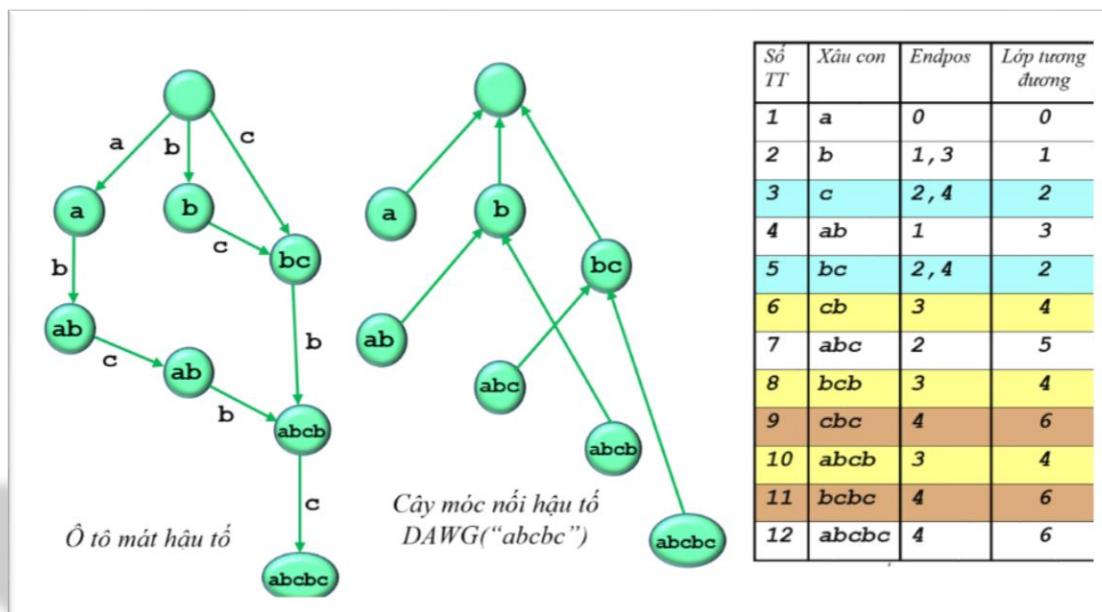
Bổ đề 5.

Nếu ta xây dựng cây từ tất cả các tập endpos có được theo nguyên lý “tập cha chứa tất cả các con của mình dưới dạng các tập con” thì cây đó có cấu trúc trùng với cấu trúc cây các mốc nối hậu tố.

Chứng minh: Việc có thể xây dựng cây từ các tập endpos được suy ra từ bô đê 2 (2 tập endpos bất kỳ hoặc không giao nhau hoặc một tập chứa trong trong tập kia).

Xét trạng thái bất kỳ $\mathbf{v} \neq \mathbf{t}_0$ và mốc nối hậu tố $\text{link}(\mathbf{v})$ của nó. Từ định nghĩa mốc nối hậu tố và từ bô đê 2 có $\text{endpos}(\mathbf{v}) \subset \text{endpos}(\text{link}(\mathbf{v}))$. Với kết luận của bô đê 4 ta có điều phải chứng minh.

Ví dụ: Xét cây mốc nối hậu tố trong ô tô mát hậu tố cho xâu “**abcdbc**”:



Tóm tắt các kết quả lý thuyết

- ✚ Tập các xâu con của s có thể phân thành các lớp tương đương dựa vào tập kết thúc $endpos$,
- ✚ Ô tô mát hậu tố bao gồm trạng thái đầu $t0$ và các trạng thái kết thúc, mỗi trạng thái tương ứng với một lớp $endpos$ – tương đương,
- ✚ Mỗi trạng thái v tương ứng với một hoặc một số xâu. Gọi $longest(v)$ là xâu dài nhất trong số đó và $len(v)$ – độ dài của nó, $shortest(v)$ là xâu ngắn nhất và $minlen(v)$ là độ dài của xâu này, khi đó độ dài tất cả các xâu tương ứng với trạng thái này là các hậu tố khác nhau của xâu $longest(v)$ với mọi độ dài thuộc đoạn $[minlen(v), len(v)]$,
- ✚ Với mỗi trạng thái $v \neq t0$ có mốc nối hậu tố chỉ tới trạng thái tương ứng với hậu tố độ dài $minlen(v)-1$ của $longest(v)$. Các mốc nối hậu tố tạo thành cây với gốc là $t0$ và theo bản chất – là cây quan hệ bao trùm giữa các tập $endpos$,
- ✚ Như vậy $minlen(v)$ với $v \neq t0$ được xác định theo công thức

$$minlen(v) = len(link(v)) + 1,$$

- ✚ Nếu xuất phát từ $v0$ bất kỳ và chuyển theo mốc nối hậu tố thì sớm hoặc muộn cũng sẽ tới trạng thái đầu $t0$. Quá trình di chuyển cho ta dãy các đoạn $[minlen(v_i), len(v_i)]$ không giao nhau, mà nếu hợp nhất tất cả lại ta được đoạn *các giá trị liên tiếp*.

Giải thuật xây dựng ô tô mát hậu tố

Ô tô mát sẽ được mở rộng dần theo từng ký tự của xâu s .

Để đảm bảo nhu cầu sử dụng bộ nhớ kích thước tuyến tính, với mỗi trạng thái chỉ lưu các giá trị **len**, **link** và danh sách các phép chuyển từ trạng thái này. Ta cũng sẽ không đánh nhãn cho các trạng thái kết thúc (nếu có nhu cầu, có thể tìm và gán nhãn cho các trạng thái kết thúc sau khi đã xây dựng xong ô tô mát hậu tố).

- ✚ Ban đầu, ô tô mát chỉ bao gồm một trạng thái $t0$ duy nhất, đánh số là trạng thái 0 (các trạng thái còn lại sẽ có số 1, 2, 3, ...). Trạng thái này có $len = 0$ và $link = -1$ (chỉ tới trạng thái ảo),
- ✚ Toàn bộ giải thuật dựa trên cơ sở tổ chức mở rộng ô tô mát khi bổ sung thêm một ký tự c tiếp theo từ xâu s vào cuối xâu đang xét.
- ✚ Gọi **last** – trạng thái tương ứng với toàn bộ xâu đang xét, ban đầu **last** = 0,
- ✚ Tạo trạng thái mới **cur** với giá trị $len(cur) = len(last) + 1$, giá trị $link(cur)$ tạm thời chưa xác định,

- + Xuất phát từ trạng thái **last**, nếu từ đó chưa có bước chuyển theo ký tự **c** thì bổ sung bước chuyển mới theo **c** vào trạng thái **cur**, chuyển theo mốc nối hậu tố, kiểm tra nếu chưa có bước chuyển – bổ sung thêm. Nếu ở một thời điểm nào đó gặp trạng thái đã có bước chuyển: ghi nhận **p** là số của trạng thái tìm được và xử lý kết thúc,
- + Nếu sau toàn bộ quá trình tìm kiếm không gặp bước chuyển nào theo ký tự **c** và đã tới trạng thái ảo -1 thì đơn giản gán **link(cur)** = 0 và xử lý kết thúc,
- + Trường hợp tìm thấy trạng thái **p**: gọi **q** là trạng thái dẫn tới từ **p**,
- + Phân biệt 2 trường hợp: thỏa mãn điều kiện **len(p)+1 = len(q)** hoặc không,
- + Nếu **len(p)+1 = len(q)** thì gán **link(cur) = q** và xử lý kết thúc,
- + Trong trường hợp ngược lại: tạo trạng thái mới **clone** sao chép các dữ liệu của đỉnh **q** (mốc nối hậu tố, các bước chuyển), ngoại trừ giá trị **len**: **len(clone) = len(p) + 1**. Sau khi tạo dữ liệu cho **clone** ghi nhận mốc nối hậu tố của **cur** tới trạng thái **clone** và đồng thời chuyển mốc nối hậu tố từ **q** tới **clone**. Công việc tiếp theo là chuyển theo mốc nối hậu tố bắt đầu từ **p**, nếu gặp phép chuyển theo ký tự **c** tới **q** thì đổi mốc nối đó cho chuyển sang **clone**.
- + **Xử lý kết thúc:** Trong mọi trường hợp, trước khi kết thúc cần cập nhật **last** bằng cách gán **cur** cho nó.

Nếu cần biết các trạng thái kết thúc ta duyệt cây bắt đầu từ **last** (nút tương ứng với toàn bộ xâu), chuyển theo mốc nối hậu tố cho tới khi gặp trạng thái đầu, đánh dấu tất cả các trạng thái đã gặp khi duyệt, đó là các trạng thái tương ứng với các hậu tố của **s**.

Mỗi lần mở rộng thêm một ký tự sẽ kéo theo việc bổ sung một hoặc 2 trạng thái, vì vậy nhu cầu sử dụng bộ nhớ kích thước tuyến tính là hiển nhiên.

Độ phức tạp tuyến tính sẽ được khẳng định khi chứng minh tính đúng đắn của giải thuật.

Chứng minh tính đúng đắn của giải thuật

Gọi bước chuyển (**p, q**) là **liên tiếp** nếu $len(p) + 1 = len(q)$, trong trường hợp ngược lại, tức là khi $len(p) + 1 < len(q)$, bước chuyển được gọi là **nhảy cách**.

Các bước chuyển liên tiếp cho trạng thái mà việc chuyển tới nó sẽ không thay đổi ở các bước xử lý tiếp theo.

Để tiện trình bày ta ký hiệu **s** là xâu tương ứng với ô tô mát hậu tố **đã xây dựng** ở bước đang xét, trước khi bổ sung thêm ký tự **c**.

Giải thuật bắt đầu từ việc tạo trạng thái **cur** mới để ghi nhận xâu mới là **s+c**. Việc bổ sung thêm ký tự **c** sẽ làm xuất hiện llop tương đương mới tương ứng với các xâu kết thúc bởi **c**.

Ta phải gắn việc chuyển sang **c** (tới trạng thái **cur**) sau mỗi hậu tố đã có. Nếu trước đó chưa có ký tự nào là **c**, khi duyệt cây từ cuối về đầu ta sẽ tới trạng thái ảo (-1), việc bổ sung phép chuyển đơn thuần là gắn vào sau last và đảm bảo mọi hậu tố đều kết thúc bởi **c**. Nhưng nếu trước đó đã có ký tự **c**, trong quá trình duyệt cây từ cuối lên sẽ phát hiện bước chuyển tới **c** ở bước chuyển (**p, q**). Điều này nói lên rằng chúng ta muốn bổ sung vào ô tô mát xâu **x + c**, trong đó **x** là một hậu tố độ dài $\text{len}(p)$ nào đó của **s**, nhưng xâu **x + c** này trước đó đã được ghi nhận! Cần phải tạo ra mốc nối hậu tố tới trạng thái tương ứng với hậu tố độ dài $\text{len}(p) + 1$.

Nếu bước chuyển (**p, q**) là liên tiếp, tức là $\text{len}(q) = \text{len}(p) + 1$ thì chỉ cần cho **cur** chỉ tới **q**.

Nếu bước chuyển (**p, q**) là nhảy cách thì **q** chỉ tới hậu tố có độ dài lớn hơn $\text{len}(p) + 1$, ta phải tách hậu tố dài hơn này thành 2 phần, trong đó phần đầu có độ dài $\text{len}(p) + 1$. Để mốc nối 2 phần, thông tin tương ứng với **q** được lưu lại trong **clone**, **clone** sẽ chỉ tới nơi trước đây xác định bởi **q**, còn **q** – mốc nối tới **clone**. Mốc nối mới **cur** sẽ chỉ tới **clone**. Ngoài ra, ta còn phải chỉnh lý tất cả các mốc nối trước đó chỉ tới **q**.

Chứng minh độ phức tạp tuyến tính

Xét trường hợp bảng chữ cái được xét có kích thước **k** cố định. Nếu **k** đủ nhỏ thì có thể tổ chức lưu trữ mỗi mốc nối dưới dạng mảng kích thước **k** và danh sách động quản lý các mốc nối. Chi phí bộ nhớ sẽ là $O(n \times k)$ với thời gian xử lý là $O(n)$.

Khi chưa tồn tại **c** trong phần đã xét việc bổ sung nút mới có độ phức tạp $O(1)$.

Nếu **c** đã có, việc tạo clone có độ phức tạp $O(1)$. Xét chi phí thời gian chỉnh mốc nối. Ký hiệu **v** = *longest(p)*. Đó là hậu tố của **s**. Theo quá trình duyệt, độ dài của nó sẽ giảm, tức là vị trí của **v** trong **s** đơn điệu tăng vì số lần không vượt quá **n**.

Tổ chức dữ liệu

Thông tin về mỗi bước chuyển gồm 3 thành phần:

Khai báo dữ liệu:

```
const int MAXLEN = 100000;
state st[MAXLEN*2];
int sz, last;
```

```
struct state
{
    int len, link;
    map<char,int> next;
};
```

Danh sách các
bước chuyển

Khởi tạo:

```
void sa_init() {
    sz = last = 0;
    st[0].len = 0;
    st[0].link = -1;
    ++sz;
    /*
    // Chỉ cần khi thực hiện CT nhiều lần với các xâu khác nhau
    for (int i=0; i<MAXLEN*2; ++i)
        st[i].next.clear();
    */
}
```

Hàm xác định ô tô mát hậu tố:

```
void sa_extend (char c) {
    int cur = sz++;
    st[cur].len = st[last].len + 1;
    int p;
    for (p=last; p!=-1 && !st[p].next.count(c); p=st[p].link)
        st[p].next[c] = cur;
    if (p == -1)
        st[cur].link = 0;
    else {
        int q = st[p].next[c];
        if (st[p].len + 1 == st[q].len)
            st[cur].link = q;
        else {
            int clone = sz++;
            st[clone].len = st[p].len + 1;
            st[clone].next = st[q].next;
            st[clone].link = st[q].link;
            for (; p!=-1 && st[p].next[c]==q; p=st[p].link)
                st[p].next[c] = clone;
            st[q].link = st[cur].link = clone;
        }
    }
    last = cur;
}
```

Một số tính chất của ô tô mát hậu tố

Số trạng thái

Số trạng thái của ô tô mát hậu tố xây dựng cho xâu s độ dài n là không quá $2n-1$ (với $n \geq 3$).

Điều này có thể dễ dàng suy ra từ cách xây dựng ô tô mát.

Tồn tại các xâu có số trạng thái của ô tô mát đúng bằng $2n-1$, ví dụ $s = "abbbb"$.

Số phép chuyển

Số phép chuyển của ô tô mát hậu tố xây dựng cho xâu s độ dài n là không quá $3n-4$ (với $n \geq 3$).

Trước hết xác định số phép chuyển liên tiếp. Xét cây khung từ các đường dài nhất của ô tô mát, bắt đầu từ $t0$. Cây khung này chỉ chứa các cạnh chuyển liên tiếp, như vậy số cạnh sẽ nhỏ hơn số đỉnh một đơn vị, tức là có $2n-2$ phép chuyển.

Xét số phép chuyển nhảy cách. Giả thiết (p, q) – phép chuyển nhảy cách theo ký tự c . Phép chuyển này tương ứng với xâu $u+c+w$, trong đó u là xâu dài nhất nhận được khi đi từ t_0 đến p , w là xâu dài nhất nhận được khi đi từ q đến trạng thái cuối. Một mặt, mọi xâu $u+c+w$ với mọi phép nhảy cách đều khác nhau, mặt khác mỗi xâu trong số đó đều là hậu tố của s và không thể là cả xâu s , vậy số lượng phép nhảy cách không thể quá $n - 1$. Tổng cộng ta có số phép chuyển là $3n - 3$. Nhưng số trạng thái nhiều nhất đạt được ở xâu dạng $s = "abbb...bbbc"$ vì vậy số bước chuyển nhiều nhất chỉ có thể là $3n - 4$.

Số bước chuyển nhiều nhất này đạt được ở xâu dạng “**abbb...bbbc**”.

Mối quan hệ với cây hậu tố

Ta chỉ xét các xâu trong đó mỗi hậu tố có một đỉnh riêng trong cây hậu tố (không phải xâu nào cũng có tính chất như vậy, ví dụ xâu “aaaaaa...”) Để có được điều đó, thông thường người ta thêm vào cuối xâu một ký tự đặc biệt, ví dụ ‘\$’.

Ký hiệu s' là xâu s viết theo trình tự ngược lại, $DAWG(s)$ – ô tô mát hậu tố, $ST(s)$ – cây hậu tố của s .

Xét khái niệm *móc nối mở rộng*: với đỉnh v cố định của cây hậu tố và ký tự c , móc nối mở rộng $ext[v, c]$ chỉ tới đỉnh của cây tương ứng với xâu $c + v$ (nếu đường đi $c + v$ kết thúc ở giữa cạnh thì móc nối này chỉ tới điểm cuối của cạnh). Nếu đường đi $c + v$ không có trong cây thì móc nối chỉ tới rỗng. Trong một chừng mực nào đó, móc nối mở rộng có ý nghĩa ngược lại so với khái niệm móc nối hậu tố.

Định lý 1

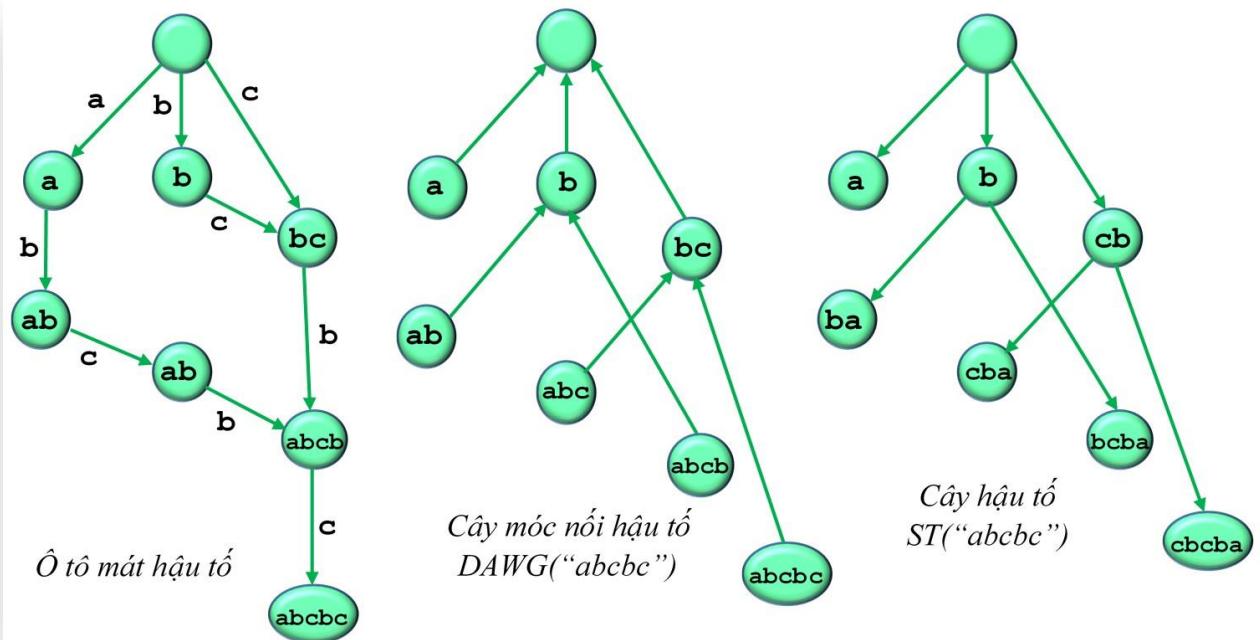
Cây tạo bởi các móc nối hậu tố trong $DAWG(s)$ là cây hậu tố $ST(s')$.

Định lý 2

$DAWG(s)$ là đồ thị các móc nối mở rộng của cây hậu tố $ST(s')$. Các cạnh liên tiếp của $DAWG(s)$ là móc nối hậu tố ngược trong $ST(s')$.

Hai định lý này cho phép khi biết một loại cây – có thể xây dựng cây loại kia với độ phức tạp $O(n)$.

Ví dụ: ô tô mát hậu tố và các loại cây.



Bổ đề

Ba điều khẳng định sau là tương đương đối với 2 xâu con u và w bất kỳ:

- + $\text{endpos}(u) = \text{endpos}(w)$ trong xâu s ,
- + $\text{firstpos}(u') = \text{firstpos}(w')$ trong xâu s' ,
- + u' và w' nằm trên cùng một đường đi từ gốc trong cây hậu tố $ST(s')$.

Nếu 2 xâu có cùng vị trí bắt đầu thì một xâu sẽ là tiền tố của xâu kia và do đó trong cây hậu tố một xâu sẽ nằm trên đường đi của xâu kia.

Chứng minh định lý 1

Trạng thái của ô tô mát hậu tố tương ứng với đỉnh của cây hậu tố.

Xét móc nối hậu tố bất kỳ $y = link(x)$. Theo định nghĩa của móc nối hậu tố $longest(y)$ là hậu tố của $longest(x)$ và với mọi y thỏa mãn xâu có $len(y)$ lớn nhất sẽ được chọn.

Đối với xâu viết ngược s' điều này có nghĩa là móc nối hậu tố $link[x]$ chỉ tới tiền tố dài nhất của xâu tương ứng với trạng thái x , sao cho tiền tố này tương ứng với trạng thái y . Nói một cách khác, móc nối hậu tố $link[x]$ chỉ tới cha của đỉnh x trong cây hậu tố. Đó là điều phải chứng minh.

Chứng minh định lý 2

Trạng thái của ô tô mát hậu tố tương ứng với đỉnh của cây hậu tố.

Xét phép chuyển bất kỳ (\mathbf{x} , \mathbf{y} , \mathbf{c}) trong ô tô mát hậu tố DAWG(\mathbf{s}). Sự tồn tại của phép chuyển này nói lên rằng \mathbf{y} là trạng thái mà lớp tương đương của nó chứa xâu con $\text{longest}(\mathbf{x}) + \mathbf{c}$. Trong xâu đảo ngược \mathbf{s}' điều này có nghĩa \mathbf{y} là trạng thái tương ứng với xâu con có firstpos (trong \mathbf{s}') trùng với firstpos của xâu con $\mathbf{c} + \text{longest}(\mathbf{x})'$.

Điều này nói lên rằng $\text{longest}(\mathbf{y})' = \text{ext}[\mathbf{c}, \text{longest}(\mathbf{x})']$.

Phần thứ nhất của định lý đã được chứng minh.

Ta sẽ chứng minh tiếp phần thứ 2. Đặc trưng của phép chuyển liên tục là $\text{length}(\mathbf{y}) = \text{length}(\mathbf{x}) + 1$, tức là sau khi gắn thêm ký tự \mathbf{c} ta rời vào trạng thái với xâu của lớp tương đương xác định bởi trạng thái này. Điều này có nghĩa khi tính mốc nối mở rộng tương ứng $\text{ext}[\mathbf{c}, \text{longest}(\mathbf{x})']$ ta tới ngay đỉnh của cây mà không phải đi dần xuống tới đỉnh gần nhất của cây. Như vậy, sau khi gắn thêm một ký tự vào đầu ta tới được đỉnh khác của cây, tức là có mốc nối hậu tố ngược trong cây. Đó là điều phải chứng minh.

Định lý 3

Khi có ô tô mát hậu tố DAWG(\mathbf{s}), có thể xây dựng cây hậu tố ST(\mathbf{s}') với chi phí thời gian $O(n)$.

Định lý 4

Khi có cây hậu tố ST(\mathbf{s}'), có thể xây dựng ô tô mát hậu tố DAWG(\mathbf{s}) với chi phí thời gian $O(n)$.

Chứng minh định lý 3:

Cây hậu tố ST(\mathbf{s}') có số đỉnh đúng bằng số trạng thái trong DAWG(\mathbf{s}), hơn thế nữa, đỉnh của cây nhận được từ trạng thái \mathbf{v} của ô tô mát tương ứng với xâu độ dài $\text{len}(\mathbf{v})$.

Theo định lý 1, các cạnh của cây do mốc nối hậu tố ngược tạo ra, nhãn của cung được tính như hiệu len các trạng thái và với mỗi trạng thái của ô tô mát ta còn biết thêm một phần tử thuộc tập endpos của nó (phần tử này có thể được lưu trữ trong quá trình xây dựng ô tô mát).

Mốc nối hậu tố trong cây có thể được xác định theo định lý 2: chỉ cần xét các phép chuyển liên tục trong ô tô mát và với mỗi phép chuyển (\mathbf{x} , \mathbf{y}) loại này – bỏ sung thêm mốc nối $\text{link}(\mathbf{y}) = \mathbf{x}$.

Như vậy, với chi phí thời gian $O(n)$ ta có thể xây dựng cây hậu tố cùng với các mốc nối hậu tố trong cây.

Lưu ý: Nếu kích thước k của bảng chữ cái không phải là một hằng thì chi phí thời gian sẽ là $O(n \log k)$.

Chứng minh định lý 4:

Ô tô mát hậu tố DAWG(s) chứa số trạng thái đúng bằng số đỉnh trong cây hậu tố ST(s'). Xâu dài nhất $\text{longest}(v)$ của trạng thái v tương ứng với đảo ngược kết quả đường đi từ gốc tới đỉnh v của cây.

Theo định lý 2, để xác định các phép chuyển trong ô tô mát hậu tố cần tìm mốc nối mở rộng $\text{ext}[c, v]$.

Ta nhận thấy rằng một bộ phận của mốc nối mở rộng nhận được trực tiếp từ mốc nối hậu tố trong cây. Thực vậy, nếu với đỉnh x bất kỳ ta xét mốc nối hậu tố $y = \text{link}(x)$, thì điều đó có nghĩa là cần thực hiện mốc nối mở rộng từ y sang x theo ký tự đầu tiên của xâu tương ứng với đỉnh x .

Tuy nhiên, nếu chỉ vậy ta vẫn chưa tìm hết được các mốc nối mở rộng. Cần phải duyệt từ lá tới gốc của cây hậu tố và ở mỗi đỉnh v – duyệt tất cả các con của nó, với mỗi con – xét tất cả các mốc nối mở rộng $\text{ext}[c, w]$, ghi nhận chúng vào đỉnh v nếu chưa tìm thấy mốc nối theo ký tự c tới đỉnh w .

Quá trình xử lý này đòi hỏi chi phí thời gian $O(n)$ nếu kích thước bảng chữ cái là cố định.

Mốc nối hậu tố trong ô tô mát đơn giản là các cạnh của cây hậu tố ST(s').

Ứng dụng ô tô mát hậu tố

Kiểm tra xâu con

Bài toán: Cho văn bản T và các truy vấn, mỗi truy vấn có dạng “kiểm tra xem xâu P có phải là xâu con các ký tự liên tiếp nhau của T hay không”, đưa ra câu trả lời **YES** hoặc **NO**.

Giải thuật:

Chuẩn bị: Xây dựng ô tô mát hậu tố quản lý văn bản T với chi phí thời gian $O(\text{length}(T))$.

Xử lý truy vấn: xuất phát từ trạng thái $v = t0$, duyệt các ký tự của xâu P và chuyển v sang trạng thái tương ứng. Nếu ở một thời điểm nào đó không tìm thấy trạng thái theo ký tự đang xét – đưa ra kết quả **NO**. Nếu duyệt được hết các ký tự của P – kết quả là **YES**.

Độ phức tạp xử lý một truy vấn: $O(\text{length}(P))$.

Trên thực tế, giải thuật thực hiện việc tìm độ dài tiền tố dài nhất của P gấp trong văn bản, vì vậy có thể áp dụng để xử lý các yêu cầu liên quan đến tiền tố.

Tính số lượng các xâu con khác nhau

Bài toán: Cho xâu s . Hãy tính số lượng các xâu con khác nhau của s . Mỗi xâu con là một hoặc một dãy các ký tự liên tiếp của s .

Giải thuật:

Xây dựng ô tô mát hậu tố tương ứng với s . Mỗi xâu con của s tương ứng với một đường đi nào đó trong ô tô mát và trong ô tô mát không có 2 đường đi giống nhau, vì vậy số xâu con khác nhau là **số đường đi khác nhau** trong ô tô mát, bắt đầu từ $t0$. Ô tô mát hậu tố là một đồ thị không chứa chu trình, vì vậy số đường đi khác nhau có thể tính theo phương pháp quy hoạch động.

Gọi $d[v]$ là số đường đi khác nhau xuất phát từ trạng thái v (kể cả đường đi độ dài 0), ta có:

$$d[v] = 1 + \sum_{\substack{w : \\ (v, w, c) \in DAWG}} d[w],$$

Số lượng xâu con khác nhau sẽ là $d[t0] - 1$ (*loại bỏ xâu rỗng*).

Độ phức tạp của giải thuật: $O(\text{length}(s))$.

Tổng độ dài các xâu con khác nhau

Bài toán: Cho xâu s . Hãy tính tổng độ dài các xâu con khác nhau của s . Mỗi xâu con là một hoặc một dãy các ký tự liên tiếp của s .

Giải thuật:

Bài toán có hướng giải quyết tương tự như bài trên, nhưng trong sơ đồ quy hoạch động cần tích lũy 2 đại lượng: $d[v]$ – số xâu con khác nhau và $ans[v]$ – tổng độ dài của chúng.

$d[v]$ được tính theo công thức đã nêu.

Tính $ans[v]$:

Ở mỗi nút w cứ mỗi đường đi ra thì phải thêm 1 vào tổng độ dài và số lượng đường đi ra là $d[w]$.

$$ans[v] = \sum_{\substack{w : \\ (v, w, c) \in DAWG}} d[w] + ans[w],$$

Độ phức tạp của giải thuật: $O(\text{length}(\mathbf{s}))$.

Tìm xâu con thứ k theo thứ tự từ điển

Bài toán: Cho xâu \mathbf{s} và các truy vấn, mỗi truy vấn là một số nguyên k . Với mỗi truy vấn hãy đưa ra xâu con thứ k theo thứ tự từ điển.

Giải thuật:

Xâu con thứ k (theo thứ tự từ điển) cũng là đường đi thứ k trong ô tô mát. Ở mỗi trạng thái, tính số đường đi ra từ đó và dễ dàng tìm được đường đi thứ k nếu bắt đầu duyệt từ gốc.

Độ phức tạp với mỗi truy vấn: $O(\text{length}(\mathbf{ans}))$, trong đó \mathbf{ans} – độ dài xâu kết quả.

Xâu đầy vòng nhỏ nhất

Bài toán: Cho xâu \mathbf{s} . Hãy tìm xâu đầy vòng có thứ tự từ điển nhỏ nhất.

Giải thuật:

Xây dựng ô tô mát hậu tố cho xâu $\mathbf{s+s}$. Ô tô mát sẽ chứa tất cả các xâu đầy vòng của \mathbf{s} dưới dạng đường đi. Bài toán ban đầu được đưa về việc tìm đường đi nhỏ nhất độ dài $\text{length}(\mathbf{s})$: Xuất phát từ trạng thái đầu $\mathbf{t0}$, ở mỗi trạng thái – rẽ theo ký tự nhỏ nhất.

Độ phức tạp: $O(\text{length}(\mathbf{s}))$.

Số lần gấp

Bài toán: Cho xâu \mathbf{T} và các truy vấn, mỗi truy vấn có dạng: “Hãy xác định số lần xâu \mathbf{P} tham gia vào \mathbf{T} như một xâu con. Các lần xuất hiện có thể giao nhau”.

Giải thuật:

Tạo ô tô mát hậu tố cho \mathbf{T} ,

Với mỗi trạng thái \mathbf{v} của ô tô mát tính số $\mathbf{cnt[v]}$ – kích thước của tập $\text{endpos}(\mathbf{v})$: tất cả các xâu tương ứng với cùng một trạng thái có số lần tham gia vào \mathbf{T} bằng số vị trí trong tập, việc lưu tường minh các tập là không thể, vì vậy ta chỉ lưu số lượng tập,

Tính \mathbf{cnt} :

- Không tính đối với $\mathbf{t0}$ và các nút bị tách ra khi gấp ký tự trùng, số lượng trạng thái nhận được không phải bằng cách tách từ trạng thái cũ là $\text{length}(\mathbf{T})$,
- Giá trị đầu: $\mathbf{cnt[v]} = 1$,
- Với các nút còn lại: $\mathbf{cnt[link(v)] += cnt[v]}$ (nếu phần đầu của một xâu đã gấp bao nhiêu lần thì hậu tố của nó cũng sẽ gấp bấy nhiêu lần).

Với mỗi truy vấn: đưa ra $\mathbf{cnt[u]}$, trong đó \mathbf{u} – trạng thái tương ứng với mẫu \mathbf{P} .

Độ phức tạp xử lý mỗi truy vấn: $O(\text{length}(\mathbf{P}))$.

Tìm vị trí xuất hiện đầu tiên

Bài toán: Cho văn bản \mathbf{T} và các truy vấn dạng “ Hãy tìm vị trí đầu tiên trong \mathbf{T} nơi xâu \mathbf{P} xuất hiện như một xâu con. Nếu \mathbf{P} không là xâu con của \mathbf{P} thì đưa ra giá trị -1”.

Giải thuật:

Xây dựng ô tô mát hậu tố cho văn bản \mathbf{T} ,

Với mỗi trạng thái v của ô tô mát: tìm $\mathbf{firstpos}[v]$ – điểm cuối của lần gặp đầu tiên, tức là tìm phần tử nhỏ nhất từ mỗi tập $\text{endpos}(\mathbf{v})$,

Việc tính $\mathbf{firstpos}$ được thực hiện song song với quá trình xây dựng ô tô mát: khi tạo trạng thái mới cur ở hàm $\text{sa_extend}()$, tính $\mathbf{firstpos}[\text{cur}] = \mathbf{len}[\text{cur}] - 1$. Khi sao chép \mathbf{q} vào \mathbf{clone} cần lưu thêm $\mathbf{firstpos}[\mathbf{clone}] = \mathbf{firstpos}[\mathbf{q}]$.

Kết quả cần tìm với truy vấn sẽ là $\mathbf{firstpos}[\mathbf{t}] - \mathbf{length}[\mathbf{P}] + 1$, trong đó \mathbf{t} – trạng thái ứng với mẫu \mathbf{P} .

Tìm vị trí tất cả các vị trí xuất hiện

Bài toán: Cho văn bản \mathbf{T} và các truy vấn dạng “ Hãy tìm tất cả các vị trí trong \mathbf{T} mà từ đó xâu \mathbf{P} xuất hiện như một xâu con (các xâu con có thể đè nhau).

Giải thuật:

Xây dựng ô tô mát hậu tố cho \mathbf{T} ,

Tương tự như ở bài trên – tính $\mathbf{firstpos}$,

Gọi \mathbf{t} – trạng thái tương ứng với \mathbf{P} ,

Vị trí đầu tiên $\mathbf{firstpos}[\mathbf{t}]$ là một trong số các kết quả cần tìm,

Ta cần tìm tất cả các trạng thái mà từ đó có thể *tới được \mathbf{t} theo mốc nối hậu tố*,

Để làm được việc đó, với mỗi trạng thái của ô tô mát cần lưu danh sách mốc nối hậu tố dẫn tới nó. Để nhận được kết quả truy vấn cần loang theo chiều rộng hoặc chiều sâu (BFS/DFS) bắt đầu từ \mathbf{t} , theo các mốc nối hậu tố đảo ngược. Độ phức tạp của quá trình loang là $O(\text{answer}(\mathbf{P}))$ vì mỗi trạng thái sẽ chỉ thăm một lần. Lưu ý là việc loang có thể cho kết quả trùng lặp bởi phép sao trạng thái (\mathbf{clone}), vì vậy cần có mảng đánh dấu $\mathbf{is_clon}$ phân biệt trạng thái thường với trạng thái được sao chép và bỏ qua kết quả ở các trạng thái được sao chép.

Sơ đồ xử lý trong chương trình:

```

struct state
{
    ...
    bool is_clon;
    int first_pos;
    vector<int> inv_link;
};

... sau khi xây dựng ô tô mát...
for (int v=1; v<sz; ++v)
    st[st[v].link].inv_link.push_back (v);
...
// Đưa ra kết quả

void output_all_occurrences (int v, int P_length)
{
    if (! st[v].is_clon)
        cout << st[v].first_pos - P_length + 1 << endl;
    for (size_t i=0; i<st[v].inv_link.size(); ++i)
        output_all_occurrences (st[v].inv_link[i], P_length);
}

```

Tìm xâu ngắn nhất không phải là xâu con

Bài toán: Cho xâu **s** và bảng chữ cái. Hãy tìm xâu ngắn nhất không phải là xâu con của **s**.

Giải thuật:

Quy hoạch động theo ô tô mát xây dựng cho **s**.

Gọi **d[v]** là độ dài ngắn nhất của xâu thỏa mãn điều kiện bài toán, xác định được ở trạng thái **v**.

d[v] = 1 nếu từ **v** không có phép chuyển theo một ký tự nào đó của bảng chữ cái, trong trường hợp ngược lại có

$$d[v] = 1 + \min_{\substack{w : \\ (v, w, c) \in DAWG}} d[w].$$

Độ dài xâu cần tìm sẽ là **d[t0]**. Truy vết ngược của sơ đồ quy hoạch động sẽ cho xâu cần tìm.

Độ phức tạp của giải thuật: $O(\text{length}(\mathbf{s}))$.

Tìm xâu con chung dài nhất của hai xâu

Bài toán: Cho hai xâu s và t . Hãy tìm xâu x dài nhất đồng thời là xâu con của s và của t .

Giải thuật:

Xây dựng ô tô mát hậu tố với xâu s ,

Duyệt các tiền tố của xâu t , với mỗi tiền tố: tìm hậu tố dài nhất của tiền tố đó xuất hiện trong s , nói một cách khác, với mỗi vị trí trong t tìm xâu con chung của s và t kết thúc ở vị trí này.

Cần có 2 biến điều khiển: trạng thái đang xét v và độ dài đang xét l . Hai biến này sẽ quản lý xâu con chung cần tìm.

Ban đầu $v = t0$, $l = 0$ (xâu con chung rỗng).

Xét ký tự $t[i]$:

- ❖ Nếu từ v có phép chuyển theo $t[i]$ thì thực hiện phép chuyển và tăng độ dài lên 1,
- ❖ Nếu không tồn tại phép chuyển: giảm phần trùng nhau (gán $v = \text{link}(v)$, cập nhật lại độ dài hiện có $l = \text{len}(v)$).
- ❖ Cập nhật lại độ dài max và vị trí nơi đạt max.

Độ phức tạp của giải thuật: ta phải duyệt tất cả các ký tự của t , ở mỗi lần duyệt độ dài đang xét tăng lên 1 hoặc đơn điệu giảm, nhưng mức giảm không thể vượt quá độ dài của t , vì vậy độ phức tạp của phần duyệt là $O(\text{length}(t))$. Kết hợp với việc xây dựng ô tô mát, ta có độ phức tạp của giải thuật là $O(\text{length}(s) + \text{length}(t))$.

Hàm xử lý:

```
string lcs (string s, string t)
{
    sa_init();
    for (int i=0; i<(int)s.length(); ++i)
        sa_extend (s[i]);

    int v = 0, l = 0,
        best = 0, bestpos = 0;
    for (int i=0; i<(int)t.length(); ++i)
    {
        while (v && ! st[v].next.count(t[i]))
        {
            v = st[v].link;
            l = st[v].length;
        }
        if (st[v].next.count(t[i]))
        {
            v = st[v].next[t[i]];
            ++l;
        }
        if (l > best)
            best = l, bestpos = i;
    }
    return t.substr (bestpos-best+1, best);
}
```

Tìm xâu con chung lớn nhất của một số xâu

Bài toán: Cho k xâu s_1, s_2, \dots, s_k . Hãy tìm xâu x dài nhất đồng thời là xâu con của tất cả các xâu đã cho.

Giải thuật:

Gắn cuối mỗi xâu s_i một ký hiệu kết thúc D_i riêng và kết nối tất cả thành một xâu $T = s_1 D_1 s_2 D_2 \dots s_k D_k$.

Xây dựng ô tô mát hậu tố cho T . Ta thấy, nếu một xâu con nào đó có trong s_j thì trong ô tô mát hậu tố từ xâu con này có đường tới D_j và không chứa D_i , $i = 1 \div k$, $i \neq j$.

Như vậy với mỗi trạng thái của ô tô mát cần kiểm tra xem có đường chứa D_i ($i = 1 \div k$) và không chứa các D_j với $j \neq i$. Nếu v là trạng thái như vậy thì kết quả cần tìm sẽ là longest(v). Việc kiểm tra có thể được thực hiện bằng

bằng phương pháp loang (BFS/DFS).

Độ phức tạp của giải thuật: $O(\sum \text{length}(s_i) \cdot K)$



Cấu trúc ROPE

Rope là cấu trúc dữ liệu lưu trữ xâu dưới dạng cây nhị phân cân bằng cho phép thực hiện các phép bổ sung, loại bỏ, kết nối xâu với độ phức tạp $O(\log n)$.

Trong một số các trường hợp, khi làm việc với xâu ta cần:

- ✚ Thực hiện nhanh các phép ghép nối, trích xâu con, . . .
- ✚ Với các xâu dài: thời gian thực hiện các phép xử lý trên không tỷ lệ với độ dài của xâu,
- ✚ Lưu trữ trạng thái và quay về trạng thái cũ khi cần thiết.

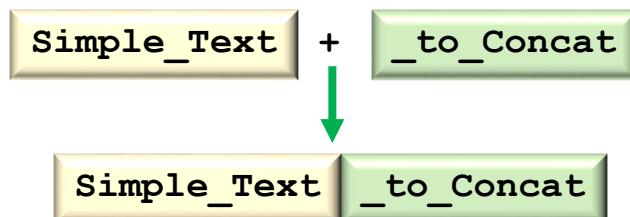
Việc lưu trữ tuyến tính các ký tự của xâu không đáp ứng được các yêu cầu nêu trên.

Mô tả cấu trúc

Xâu được lưu trữ theo mô hình Treap với khóa ẩn. Mỗi lá của cây sẽ lưu đoạn liên tục các ký tự của xâu và độ dài đoạn đó. Ở các nút trong – lưu tổng độ dài các lá của cây con. Ban đầu cây chỉ bao gồm một đỉnh lưu xâu đang xét. Dựa vào thông tin ở các đỉnh trong có thể tìm các ký tự của xâu theo chỉ số. Không cần lưu thông tin đánh dấu lá vì mỗi nút trong có đúng 2 nút con, còn lá – không có nút con. Vì vậy để nhận dạng lá chỉ cần kiểm tra một nút có nút con hay không.

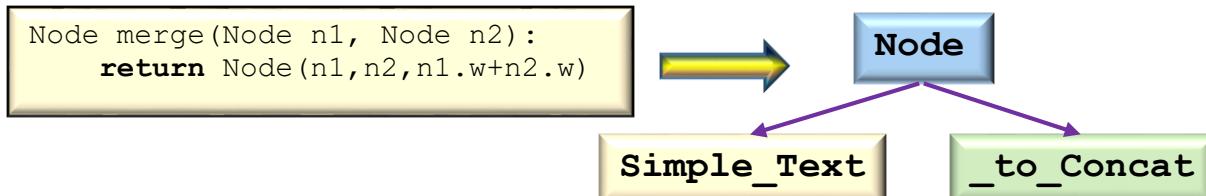
Cộng xâu – merge

Giả thiết ta cần hợp nhất 2 xâu, tạo xâu mới kết quả ghép liên tiếp 2 xâu ban đầu:

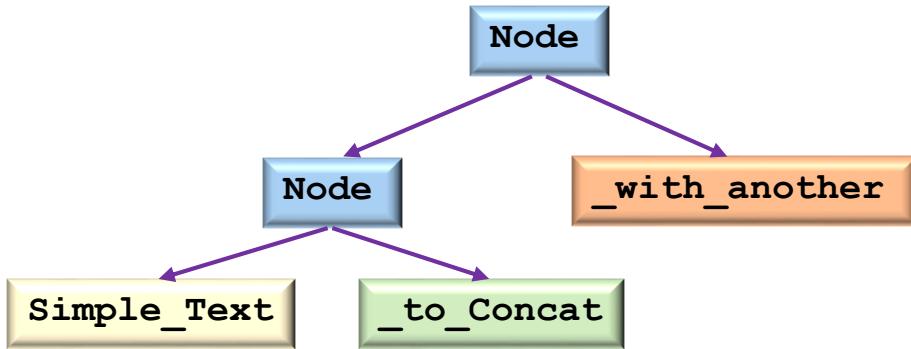


Việc hợp nhất xâu bình thường đòi hỏi dự trữ bộ nhớ cho xâu kết quả, lần lượt copy các ký tự của xâu thứ nhất và sau đó – các ký tự của xâu thứ 2 vào xâu kết quả. Độ phức tạp của giải thuật là $O(n)$.

Với cấu trúc cây, kết nối được thực hiện với độ phức tạp $O(1)$ và chỉ cần thêm bộ nhớ hỗ trợ tổ chức cây.



Độ phức tạp xử lý vẫn là $O(1)$ nếu ta cần kết nối thêm:



Xác định ký tự theo chỉ số - hàm get

Để tìm ký tự theo chỉ số i cần duyệt cây bắt đầu từ nút gốc, dựa vào trọng số ở mỗi nút để quyết định đi sang cây con trái hay phải:

- ♣ Nếu nút đang xét không phải là lá:
 - ♠ Nếu nút trái có trọng số w và $w \geq i \rightarrow$ chuyển sang cây con trái,
 - ♠ Nếu $w < i \rightarrow i = w$, chuyển sang cây con phải,
- ♣ Nếu nút đang xét là nút lá: truy nhập tới kết quả là ký tự thứ i .

```

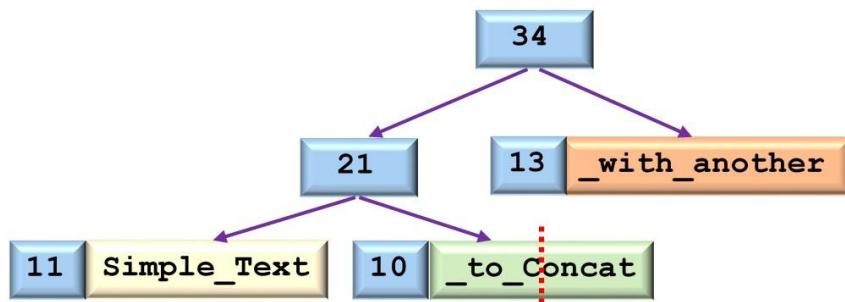
char get(int i, Node node) {
    if node.left != Ø
        if node.left.w >= i
            return get(i, node.left)
        else
            return get(i - node.left.w, node.right)
    else
        return node.s[i]
  
```

Độ phức tạp của giải thuật: $O(h)$, trong đó h – độ cao của cây.

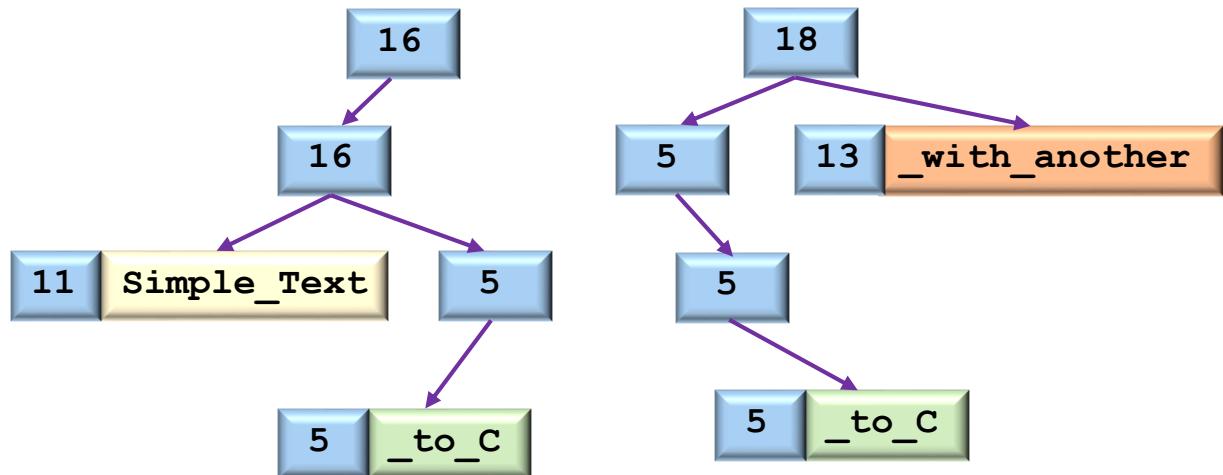
Tách xâu – hàm split

Để tách xâu thành 2 phần với ký tự cuối của phần đầu ở vị trí i : tìm phần xâu chứa ký tự thứ i (xử lý tương tự hàm **get**), mỗi nút gặp trên đường đi được chia thành hai để gắn với 2 cây, sau khi chia đổi cần tính lại trọng số.

Ví dụ dưới đây xét việc tách xâu ở vị trí 16.



Kết quả duyệt và tách đỉnh cho 2 cây:



Với những nút trong chỉ có một đỉnh con: cần thay đỉnh đó bằng đỉnh con của nó.



Giải thuật trên ngôn ngữ giả định:

```

Pair<Node, Node> split(Node node, int i):
    Node tree1, tree2
    if node.left != ∅
        if node.left.w >= i
            res = split(node.left, i)
            tree1 = res.first
            tree2.left = res.second
            tree2.right = node.right
            tree2.w = tree2.left.w + tree2.right.w
        else
            res = split(node.right, i - node.left.w)
            tree1.left = node.left
            tree1.right = res.first
            tree1.w = tree1.left.w + tree1.right.w
            tree2 = res.second
    else
        tree1.s = node.s.substr(0, i)
        tree2.s = node.s.substr(i, node.s.len)
        tree1.w = i
        tree2.w = node.s.len - i
    return (tree1, tree2)
  
```

Độ phức tạp của giải thuật: $O(h)$, trong đó h – độ cao của cây.

Các phép xóa, bổ sung – delete và insert

Các phép **delete** và **insert** dễ dàng thực hiện thông qua các hàm **merge** và **split** đã xét ở trên.

Giải thuật xóa các ký tự từ vị trí **beginIndex** cho tới trước vị trí **endIndex**:

```
Node delete(Node node, int beginIndex, int endIndex):  
    (tree1, tree2) = split(node, beginIndex)  
    tree3 = split(tree2, endIndex - beginIndex).second  
    return merge(tree1, tree3)
```

Giải thuật bổ sung xâu **s** vào xâu ban đầu từ vị trí **insertIndex**:

```
Node insert(Node node, int insertIndex, string s):  
    (tree1, tree3) = split(node, insertIndex)  
    tree2 = Node(s)  
    return merge(merge(tree1, tree2), tree3)
```

Độ phức tạp của giải thuật: $O(h)$.

Cân bằng cây

Mỗi lần ghép xâu đều cần kiểm tra xem cây có bị rơi vào dạng “bậc thang” hay không, trong trường hợp cần thiết phải tiến hành cân bằng cây.

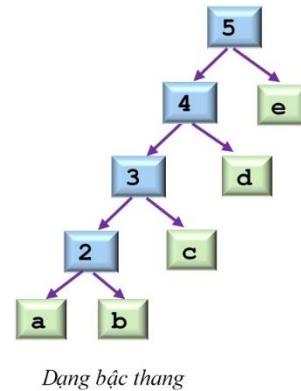
Một cây được cân bằng tốt nếu độ dài của xâu không nhỏ hơn số Fibonacci thứ $(h+2)$.

Việc cân bằng cây được thực hiện theo các giải thuật áp dụng với cây nhị phân cân bằng.

Các giải pháp nâng cao hiệu quả

Việc tồn tại các xâu quá ngắn dễ làm cây mất cân bằng và làm giảm hiệu quả của cấu trúc, vì vậy người ta thường trực tiếp *ghép các xâu con ngắn* thành một xâu có độ dài không nhỏ hơn một giá trị m nào đó (ví dụ, $m = 32$). Giải pháp này giúp tiết kiệm đáng kể bộ nhớ và vẫn đảm bảo các ưu việt của cấu trúc rope.

Trong phần lớn các trường hợp xử lý cần duyệt các ký tự liên tiếp hoặc có vị trí gần nhau. Xác xuất của 2 ký tự ở các vị trí liên tiếp hoặc gần nhau thuộc cùng một xâu trong cấu trúc là rất cao. Vì vậy ở các bài toán thực tế, khi áp dụng cấu trúc này người ta thường tạo *phòng đệm vòng tròn* lưu vị trí kết quả tìm kiếm của các truy vấn. Quá trình tìm kiếm bắt đầu từ việc kiểm tra các kết quả đã nhận được ở các truy vấn trước.



Quy hoạch động

Nhiều bài toán tối ưu có thể giải theo sơ đồ sau:

- ⊕ Phân rã bài toán ban đầu thành k bài toán con tương tự như bài toán ban đầu nhưng có kích thước nhỏ hơn,
- ⊕ Kết quả cần tìm của bài toán ban đầu được tổng hợp từ kết quả của mỗi bài toán con,
- ⊕ Với mỗi bài toán nhận được: lặp lại hai bước trên,
- ⊕ Quá trình phân rã được thực hiện cho đến khi nhận được bài toán con đủ đơn giản, có thể dễ dàng tìm ra lời giải.

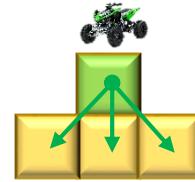
Nguyên lý Bellman

Trước hết ta xét một số bài toán ví dụ để hiểu rõ các khái niệm sẽ sử dụng trong nguyên lý quy hoạch động.

Bài 1a. VỀ ĐÍCH

Chặng cuối cùng trước khi cán đích của một cuộc đua xe địa hình là một vùng chướng ngại hình chữ nhật kích thước $n \times m$ ô (n hàng và m cột). Các tay đua có thể và chỉ có thể vào ô bất kỳ ở hàng đầu tiên và ra khỏi vùng chướng ngại từ ô bất kỳ ở hàng cuối cùng. Xe không được ra khỏi vùng chướng ngại trước khi tới được hàng cuối cùng. Từ một ô (i, j) xe có thể di chuyển sang một trong số các ô $(i+1, j-1), (i+1, j), (i+1, j+1)$ nếu ô đó tồn tại. Thời gian vượt qua ô (i, j) là $a_{i,j}$, $i = 1 \div n$, $j = 1 \div m$.

Hãy xác định thời gian nhỏ nhất một tay đua có thể vượt qua vùng chướng ngại.



Dữ liệu: Vào từ file văn bản FINISH.INP:

- ⊕ Dòng đầu tiên chứa 2 số nguyên n và m ($1 \leq n, m \leq 2000$, $n \times m \leq 3 \times 10^6$),
- ⊕ Dòng thứ i trong n dòng sau chứa m số nguyên $a_{i,1}, a_{i,2}, \dots, a_{i,m}$ ($0 \leq a_{i,j} \leq 10^9$, $j = 1 \div m$).

Kết quả: Đưa ra file văn bản FINISH.OUT một số nguyên – thời gian nhỏ nhất tìm được.

Ví dụ:

| FINISH.INP |
|------------|
| 3 4 |
| 6 5 1 3 |
| 2 8 7 8 |
| 9 4 6 4 |

| FINISH.OUT |
|------------|
| 11 |



Phân tích:

Các khái niệm cơ bản:

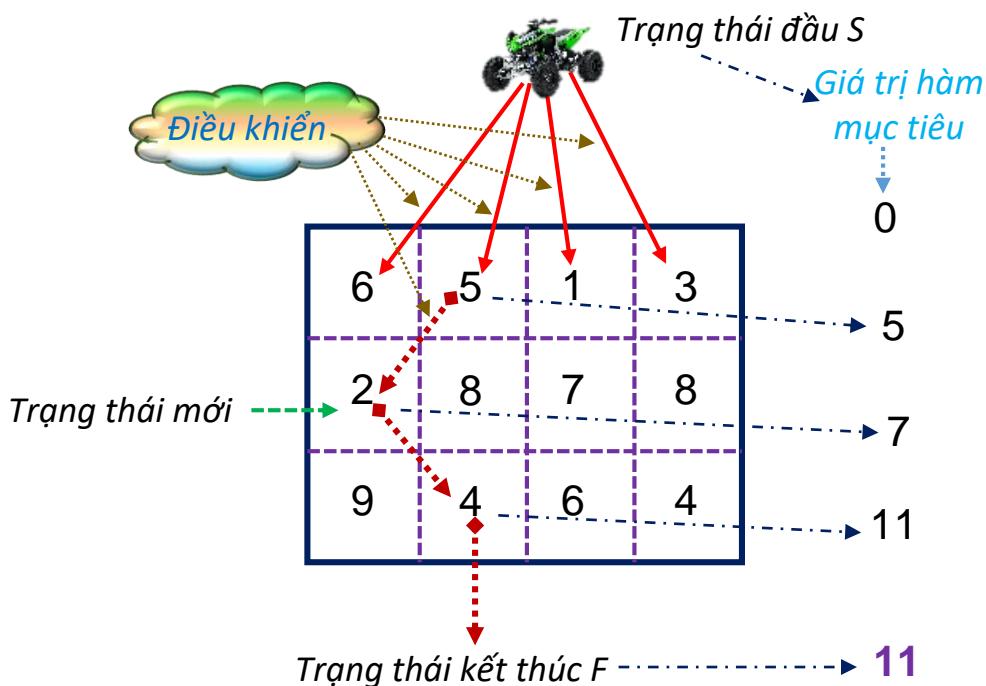
Trạng thái: Vị trí của xe được gọi là **trạng thái**,

Hàm mục tiêu: Chi phí thời gian để xe *tới và vượt qua được một trạng thái* nào đó (tức là tới một nào đó và đi qua khỏi ô đó) được gọi là **giá trị hàm mục tiêu**,

Điều khiển: Từ một trạng thái xe phải di chuyển tới trạng thái mới, việc lựa chọn một cách di chuyển được gọi là **điều khiển**.

Trạng thái đầu S – xe ở ngoài vùng chướng ngại và chuẩn bị vượt qua vùng này,

Trạng thái kết thúc F – xe đã vượt qua vùng chướng ngại và đang ở ngoài vùng này.



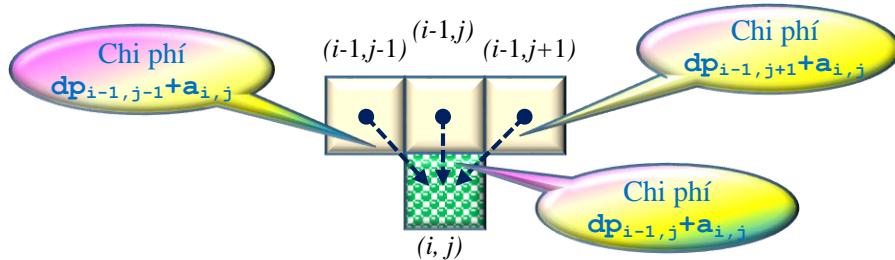
Thay vì giải quyết bài toán ban đầu ta xét **bài toán tổng quát hơn**: tính thời gian nhỏ nhất để tới và vượt qua ô (i, j) , $1 \leq i \leq n$, $1 \leq j \leq m$. Gọi chi phí này là $\text{dp}_{i,j}$.

Dễ dàng thấy rằng $\text{dp}_{1,j} = a_{1,j}$.

Giả thiết đã tính được $\text{dp}_{i-1,j}$ với mọi $j = 1 \div m$.

Xét cách tính $\text{dp}_{i,j}$.

Có 3 cách đi qua ô (i, j) :



Rõ ràng, cần chọn cách đi có chi phí nhỏ nhất:

$$dp_{i,j} = a_{i,j} + \min\{dp_{i-1,j-1}, dp_{i-1,j}, dp_{i-1,j+1}\}$$

Công thức này hợp lý với $1 < j < m$.

Để thuận tiện tính toán, cần thêm 2 giá trị đóng vai trò “hàng rào”:

$$dp_{i0-1,0} = dp_{i-1,m+1} = +\infty.$$

Công thức tính $dp_{i,j}$ có dạng:

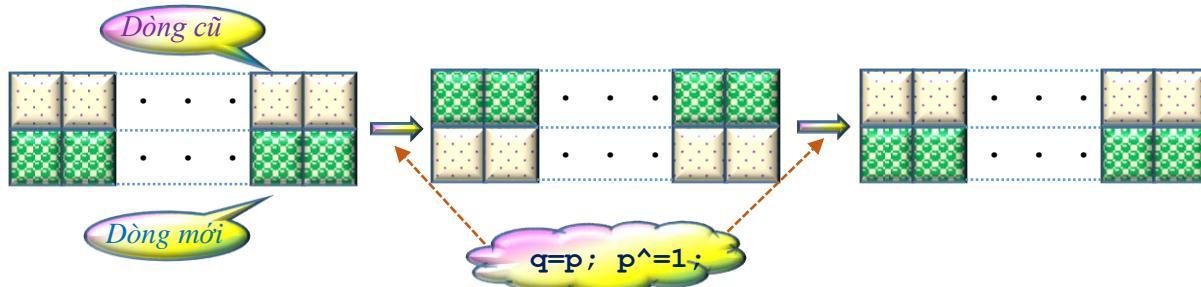
$$dp_{i,j} = \begin{cases} +\infty \text{ với } j = 0 \text{ hoặc } j = m+1, \\ a_{i,j} + \min\{dp_{i-1,j-1}, dp_{i-1,j}, dp_{i-1,j+1}\} \text{ với } 1 \leq j \leq m. \end{cases}$$

Công thức trên cho thấy, ở bài toán cụ thể này, $dp_{i,*}$ chỉ phụ thuộc vào $dp_{i-1,*}$, tức là để tính dòng i của mảng dp ta chỉ cần biết dòng $i-1$ của dp và dòng i của mảng a (ở đây dấu * tương với mọi giá trị có thể của chỉ số j).

Như vậy, thay vì giữ mảng 2 chiều dp kích thước $(n+2) \times (m+2)$ ta chỉ cần lưu 2 mảng một chiều old_dp và new_dp , mỗi mảng kích thước $m+2$.

Biết được new_dp , ta chỉ cần biến nó thành old_dp và từ đó tính được new_dp mới, tức là chuyển được tới dòng tiếp theo trong bảng 2 chiều. Quá trình này được lặp lại cho đến khi ta tới được dòng cuối cùng (dòng thứ n) của bảng hai chiều.

Để tránh việc phải gán từng giá trị của new_dp sang old_dp ta có thể dùng kỹ thuật con lắc với 2 con trỏ p và q . Các giá trị new_dp và old_dp lưu dưới dạng mảng 2 chiều $f[2][m+2]$. Con trỏ p chỉ tới dòng mới, con trỏ q – chỉ tới dòng cũ. Sau khi tính được dòng mới ta chỉ cần hoán đổi giá trị của p và q , mảng $f[p][*]$ luôn chỉ tới dòng mới, còn $f[q][*]$ – chỉ tới dòng cũ.

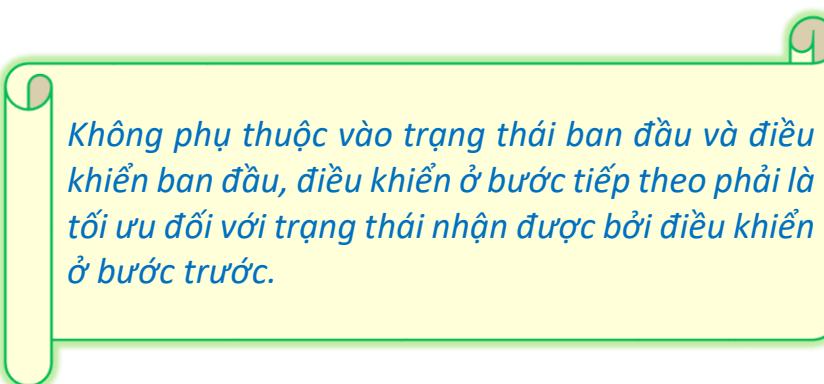


Về nguyên tắc, ta phải *tính lùi từ cuối về đầu* để hạn chế số nhánh (số giá trị tính được ở mỗi bước) cần lưu trữ. Nhưng ở bài toán chuyển động này, đường đi từ cuối về đầu tương đương với đường đi từ đầu về cuối, vì vậy ta có thể xét việc di chuyển bắt đầu từ hàng 1 xuống các hàng ở dưới. Trình tự xét này cho phép tránh được việc lưu trữ toàn bộ mảng A, thay vào đó, ta chỉ phải lưu trữ một dòng của A để tính dòng kết quả mới.

Giá trị hàm mục tiêu cần tìm sẽ là

$$\text{ans} = \min_{j=1 \dots m} \{f_{p,j}\}$$

Sơ đồ xử lý trên được gọi là *Quy hoạch động* và dựa trên nguyên lý được R. Bellman phát biểu năm 1953(*Nguyên lý Bellman*):



Điểm quan trọng cần lưu ý là việc chọn điều khiển ở bước tiếp theo *không phụ thuộc vào lịch sử điều khiển trước đó*. Điều này có nghĩa là ở bước nào đó điều khiển bị chọn sai, thì sai lầm này không thể khắc phục được trong tương lai!

Trong trường hợp có yêu cầu dẫn xuất phương án tối ưu thì cần phải tổ chức một ma trận lưu giữ các điều khiển nhận được trong quá trình tính toán.

Các giải thuật tính toán dựa trên sơ đồ lặp đơn thuần, không có việc phải lựa chọn tối ưu cũng được xếp vào lớp quy hoạch động và được gọi là *Quy hoạch động đơn giản*.

Trong nhiều trường hợp quy hoạch động được sử dụng để tạo *Bảng phương án* (*Decide Table*) cho phép phân tích và tìm lời giải theo chiều *từ trên xuống* (từ trạng thái xuất phát Start tới trạng thái kết thúc Finish) cực kỳ hiệu quả.

Các bước xử lý

Dấu hiệu nhận dạng có thể áp dụng giải thuật Quy hoạch động:

- ➡ Bài toán ban đầu có thể chia thành các bài toán con tương tự,
- ➡ Lời giải mỗi bài toán con chỉ phụ thuộc vào giá trị hàm mục tiêu của những bài toán con khác đã giải và không phụ thuộc vào phương án tối ưu ứng với giá trị của hàm mục tiêu.

Các vấn đề cần giải quyết khi giải bài toán theo giải thuật Quy hoạch động:

1. Tổ chức dữ liệu: Khai báo các biến phục vụ sơ đồ lặp, đặc biệt lưu ý cách chọn kiểu dữ liệu phù hợp,
2. Khởi tạo giá trị đầu,
3. Xác định ý nghĩa của biến điều khiển và công thức lặp,
4. Xác định giá trị tham số quản lý miền tính lặp,
5. Xác định vị trí hoặc cách tính kết quả cần tìm của hàm mục tiêu.

Các công việc nêu trên có thể thực hiện theo *trình tự tùy ý*.

Phương pháp quy hoạch động cũng thường được sử dụng như *công cụ phân tích*, *đoán nhận giải thuật* trong các bài toán điều khiển, trò chơi, bài toán tương tác người – máy.

Quy hoạch động còn được sử dụng như *công cụ xây dựng bảng phương án (Decision Table)* trong các bài toán lô gic phức tạp.

Áp dụng với bài toán đang xét:

Bài toán đang xét có tính chất đối xứng: việc tìm đường đi từ trên xuống dưới tương đương với việc tìm đường đi từ dưới lên trên,

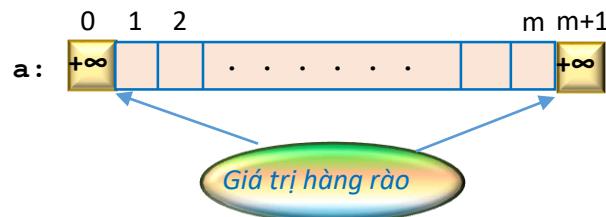
Quy trình xử lý từ trên xuống dưới cho phép vòng tránh việc lưu trữ toàn bộ dữ liệu input, thay vào đó có thể đọc từng dòng và xử lý.

Tổ chức dữ liệu:

- ➡ Mảng động `vector<int64_t> a` – lưu một dòng của ma trận dữ liệu vào,
- ➡ Mảng động `vector<int64_t>f[2]` – lưu 2 mảng phục vụ tính lặp.

Khởi tạo giá trị đầu:

Với mảng **a**:



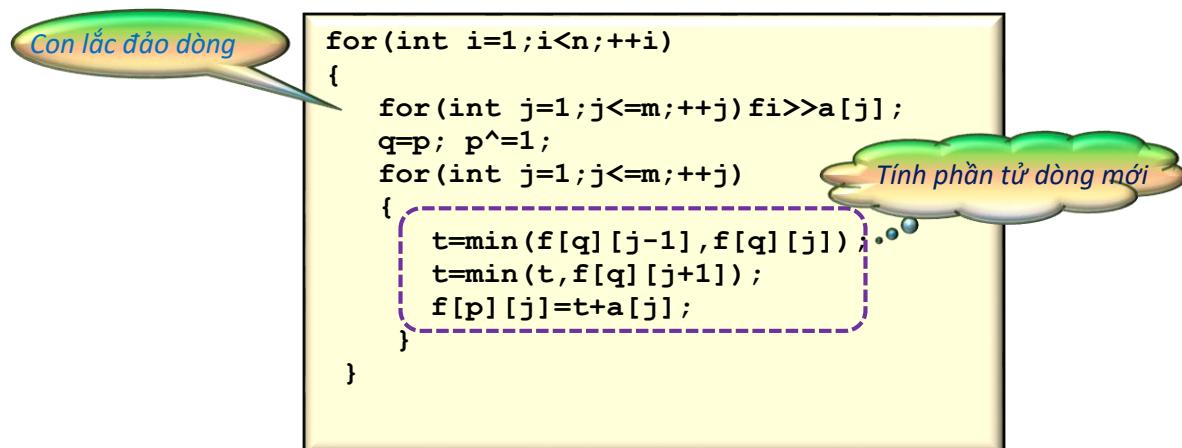
Tương tự như vậy với các mảng $f[0]$ và $f[1]$,

Khởi tạo giá trị để lặp:

```
p=0; for(int j=1;j<=m;++j)
{
    fi>>t; f[p][j]=t;
}
```

Miền xử lý lặp: các dòng từ 1 đến **n-1**, các cột từ 1 đến **m**,

Công thức lặp:



Kết quả: Giá trị *min* ở dòng mới.

Độ phức tạp của giải thuật: $O(n \times m)$.

Chương trình: Chỉ yêu cầu đưa ra thời gian nhỏ nhất.

```
#include <bits/stdc++.h>
#define NAME "finish."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int64_t INF=(int64_t)1e17;
int n,m,p,q;
int64_t t;
vector<int64_t>f[2];
int main()
{
    fi>>n>>m; p=0;
    vector<int64_t>a(m+2);
    f[0].resize(m+1);
    f[1].resize(m+1);
    a[0]=INF; a[m]=INF;
    f[0][0]=INF; f[0][m]=INF;
    f[1][0]=INF; f[1][m]=INF;

    for(int j=1;j<=m;++j)
    {
        fi>>t;
        f[p][j]=t;
    }
    for(int i=1;i<n;++i)
    {
        for(int j=1;j<=m;++j) fi>>a[j];
        q=p; p^=1;
        for(int j=1;j<=m;++j)
        {
            t=min(f[q][j-1],f[q][j]);
            t=min(t,f[q][j+1]);
            f[p][j]=t+a[j];
        }
    }
    int64_t ans=INF;
    for(int j=1;j<=m;++j) ans=min(ans,f[p][j]);
    fo<<ans;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Bài 1b.

Xét bài toán trên với yêu cầu bổ sung:

Kết quả: Đưa ra file văn bản FINISH.OUT:

- Dòng đầu tiên chứa một số nguyên – thời gian nhỏ nhất tìm được,
- Dòng thứ 2 chứa n số nguyên xác định các cột lần lượt phải tới theo đường đi tối ưu. Trường hợp có nhiều đường đi tối ưu – đưa ra đường đi tùy chọn.

Ví dụ:

| FINISH.INP |
|------------|
| 3 4 |
| 6 5 1 3 |
| 2 8 7 8 |
| 9 4 6 4 |

| FINISH.OUT |
|------------|
| 11 |
| 2 1 2 |



Phân tích:

Cần tổ chức thêm các mảng:

- Mảng dữ liệu `vector<int>res (n)` – lưu vết đường đi của phương án tối ưu,
- Mảng `vector<vector<int>>route` – lưu điều khiển tối ưu ở từng ô.

Chương trình: Yêu cầu đưa ra thời gian nhỏ nhất và cách đi.

```
#include <bits/stdc++.h>
#define NAME "finish."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int64_t INF=(int64_t)1e17;
int n,m,p,q;
int64_t t,v;
vector<int64_t>f[2];
vector<vector<int>>route;
int main()
{
    fi>>n>>m; p=0;
    vector<int64_t>a (m+2);
    vector<int>res (n);
    f[0].resize(m+1);
    f[1].resize(m+1);
    route.resize(n, vector<int>(m+2));
    a[0]=INF; a[m]=INF;
    f[0][0]=INF; f[0][m]=INF;
    f[1][0]=INF; f[1][m]=INF;
```

```

for(int j=1;j<=m;++j)
{
    fi>>t;
    f[p][j]=t; route[0][j]=0;
}
for(int i=1;i<n;++i)
{
    for(int j=1;j<=m;++j) fi>>a[j];
    q=p; p^=1;
    for(int j=1;j<=m;++j)
    {
        t=f[q][j-1];v=j-1;
        if(t>f[q][j])t=f[q][j],v=j;
        if(t>f[q][j+1])t=f[q][j+1],v=j+1;
        f[p][j]=t+a[j]; route[i][j]=v;
    }
}

int64_t ans=INF;
for(int j=1;j<=m;++j) if(ans>f[p][j]) ans=f[p][j],v=j;
fo<<ans<<'`n';
res[n-1]=v;
for(int i=n-1;i>0;--i)v=route[i][v],res[i-1]=v;
for(int i=0;i<n;++i) fo<<res[i]<<'`';
fo<<"`nTime: "<<clock() / (double)1000<<" sec";
}

```



Các bài tập ứng dụng giải thuật quy hoạch động

Bài 2. NÓI ĐIỂM

Tên chương trình: JOIN.CPP

Người ta kẻ một đường thẳng và đánh dấu n điểm trên đường đã kẻ, điểm thứ i ở vị trí tọa độ x_i , $i = 1 \div n$. Không có 2 điểm nào có cùng một tọa độ. Sau đó người ta dùng sơn phản quang tô một số đoạn thẳng, mỗi đoạn đều có độ dài khác 0. Các điểm đã đánh dấu trở thành điểm phản quang nếu nó nằm trong miền được sơn, kể cả là điểm đầu hoặc cuối đoạn được sơn.

Hãy xác định tổng độ dài ngắn nhất các đoạn được sơn để mọi điểm đã chọn đều trở thành điểm phản quang.

Dữ liệu: Vào từ file văn bản JOIN.INP:

- ✚ Dòng đầu tiên chứa một số nguyên n ($2 \leq n \leq 10^5$),
- ✚ Dòng thứ 2 chứa n số nguyên x_1, x_2, \dots, x_n ($|x_i| < 10^9$, $i = 1 \div n$).

Kết quả: Đưa ra file văn bản JOIN.OUT một số nguyên – tổng độ dài nhỏ nhất tìm được.

Ví dụ:

| JOIN.INP |
|----------------|
| 6 |
| 4 30 0 10 22 2 |

| JOIN.OUT |
|----------|
| 16 |



Giải thuật: Quy hoạch động.

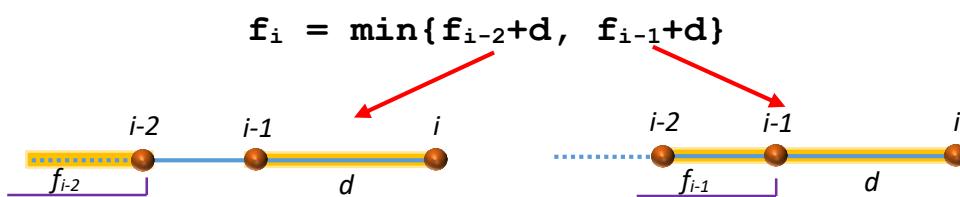
Nhận xét:

Để tổng độ dài đoạn cần sơn là nhỏ nhất thì mỗi đoạn phải bắt đầu và kết thúc bằng điểm đã đánh dấu,

Khi $n = 2$ hoặc bằng 3 – chỉ có một cách sơn: từ điểm trái nhất đến điểm phải nhất và có $\text{ans} = |\mathbf{x}_1 - \mathbf{x}_2|$ hoặc $\text{ans} = \max\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\} - \min\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$,

Sắp xếp các điểm theo thứ tự tăng dần của tọa độ,

Gọi f_i là tổng độ dài nhỏ nhất cần sơn để các điểm từ 1 đến i ($i > 3$) đều là điểm phản quang và $d = \mathbf{x}_i - \mathbf{x}_{i-1}$, ta có



Công thức lặp trên được tính với $i = 4 \div n$,

Kết quả cần tìm sẽ là f_n .

Các giá trị đầu cần chuẩn bị: $f_1 = +\infty$, $f_2 = \mathbf{x}_2 - \mathbf{x}_1$, $f_3 = \mathbf{x}_3 - \mathbf{x}_1$.

Tổ chức dữ liệu:

Hai mảng `int x[100000], f[100000]` với các chức năng như đã nói ở trên.

Lưu ý:

- ✿ Bài toán có tính chất đối xứng: duyệt từ cuối về đầu hoặc ngược lại là như nhau,
- ✿ Cách chuẩn bị trên tương ứng với việc các điểm được đánh số từ 0 , nếu đánh số từ 1 – có thể chuẩn bị đơn giản hơn.

Độ phức tạp của giải thuật: $O(nlnn)$.

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "join."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int64_t INF=(int)1e9;
int n,t;

int x[100000],f[100000];
int main()
{
    fi>>n;
    for(int i=0;i<n;++i) fi>>x[i];
    if(n==2) {fo<<abs(x[0]-x[1]); return 0;}
    sort(x,x+n);
    f[0]=INF; f[1]=x[1]-x[0]; f[2]=f[1]+x[2]-x[1];
    for(int i=3;i<n;++i)
        {t=x[i]-x[i-1]; f[i]=min(f[i-1]+t,f[i-2]+t);}

    fo<<f[n-1];
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Bài 3. DÃY KÝ TỰ CON CHUNG DÀI NHẤT

Tên chương trình: LCS.CPP

Cho hai xâu **a** và **b**. Xâu **z** được gọi là dãy con của **a** nếu có thể gạch bỏ một số(có thể là 0) ký tự trong **a** để nhận được **z**. Ví dụ, **a** = “**abcdef**” và **z** = “**bde**” , thì **z** là dãy con của **a**, nhưng “**bed**” – không phải là dãy con của **a**.

Nếu **z** đồng thời là dãy con của **a** và của **b** thì **z** được gọi là dãy con chung.

Hãy xác định độ dài dãy con chung lớn nhất của hai xâu **a** và **b**.

Dữ liệu: Vào từ file văn bản LCS.INP:

- + Dòng đầu tiên chứa xâu **a** các ký tự la tinh thường độ dài không quá 10^4 ,
- + Dòng thứ hai chứa xâu **b** các ký tự la tinh thường độ dài không quá 10^4 .

Kết quả: Đưa ra file văn bản LCS.OUT một số nguyên – độ dài lớn nhất tìm được.

Ví dụ:

| LCS.INP | LCS.OUT |
|------------------------------------|----------|
| abcdabcdef cabgad | 4 |



Giải thuật: Quy hoạch động.

Nhận xét:

Gọi n và m là độ dài các xâu a và b ,

Vấn đề cần phải giải quyết là tìm độ dài của dãy con chung dài nhất (**LCS** – *Longest Common Subsequence*),

Bài toán có tính chất đối xứng (xét từ cuối xâu về đầu hay ngược lại là như nhau), nhưng vì C++ lưu các ký tự bắt đầu từ vị trí 0 và không cho phép sử dụng chỉ số âm, để tiện lưu trữ giá trị hàng rào ta sẽ xử lý theo đúng sơ đồ lý thuyết tổng quát, xét các xâu từ cuối về đầu.

Gọi $\text{lcs}_{i,j}$ là LCS của hậu tố xâu a bắt đầu từ ký tự i và hậu tố xâu b bắt đầu từ ký tự j ,

Ta có:

$$\text{lcs}_{i,j} = \begin{cases} 0 & i=n \text{ hoặc } j=m \\ \text{lcs}_{i+1,j+1} + 1 & x_i = y_j \\ \max\{\text{lcs}_{i+1,j+1}, \text{lcs}_{i+1,j}\} & x_i \neq y_j \end{cases}$$

Việc tính dòng i và cột j của bảng **lsc** chỉ cần dựa trên dòng $i+1$ và cột $j+1$ vì vậy không cần thiết phải lưu trữ mảng hai chiều mà chỉ cần 2 mảng một chiều, lưu trữ dưới dạng `int f[2][10000]` với kỹ thuật con lắc để hoán đổi vị trí các mảng giá trị mới và giá trị cũ,

Các giá trị đầu có thể khởi tạo trước mỗi bước lặp,

Phạm vi duyệt: $i = n \div 0$, $j = m \div 0$ (bắt đầu từ n , m để khởi tạo giá trị đầu),

Kết quả: ở đầu dòng mới (biến chỉ số 0).

Độ phức tạp của giải thuật: $O(n \times m)$.

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "lcs."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int64_t INF=(int)1e9;
int n,m;
string a,b;
int f[2][10000];

int main()
{
    fi>>a>>b;p=1,q=0;
    n=a.size(); m=b.size();
    for(int i=n;i>=0;--i)
    {
        p^=1;q^=1;
        for(int j=m;j>=0;--j)
            if(i==n||j==m)f[p][j]=0;
            else if(a[i+1]==b[j+1])f[p][j]=f[q][j]+1;
            else f[p][j]=max(f[q][j],f[p][j+1]);
    }
    fo<<f[p][0];
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Quy hoạch động đơn giản

Quy hoạch động đơn giản là quy trình tính toán theo sơ đồ lặp *không đòi hỏi lựa chọn tối ưu*.

Phần lớn giải thuật giải các bài toán lặp đều có mô hình toán học đệ quy. Có thể lập trình trực tiếp theo mô hình đệ quy, chương trình rất ngắn gọn, nhưng độ phức tạp của giải thuật sẽ rất cao và tốn rất nhiều bộ nhớ, nhiều khi vượt quá khả năng cung cấp của hệ thống.

Thông thường tính toán theo sơ đồ lặp là việc quản lý hai trạng thái: *Trạng thái cũ* và *Trạng thái mới*.

Sơ đồ giải thuật lặp:

```
Chuẩn bị Trạng thái mới,  
while (!Điều kiện kết thúc)  
{  
    Biến trạng thái mới thành trạng thái cũ,  
    Tính lại trạng thái mới,  
}  
Đưa ra kết quả: Trạng thái mới.
```

Việc chuẩn bị trạng thái mới cho phép nhận được kết quả đúng, ngay khi số lần lặp bằng 0.

Khi trạng thái có cấu trúc dữ liệu phức tạp, việc biến *trạng thái mới* thành *trạng thái cũ* có thể thực hiện bằng *kỹ thuật con lắc*.

Trong một số trường hợp sơ đồ tính toán sẽ ngắn gọn hơn nhiều nếu có thể *cập nhật tại chỗ trạng thái* mà không cần phải ghi sang nơi mới. Việc cập nhật tại chỗ đặc biệt có hiệu quả khi trạng thái cần được lưu trữ không phải bằng các đại lượng vô hướng, ví dụ như mảng, véc tơ, tập hợp, . . .

Sơ đồ tính toán sẽ có dạng:

```
Chuẩn bị Trạng thái ,  
while (!Điều kiện kết thúc) Cập nhật Trạng thái,  
Đưa ra kết quả: Trạng thái nhận được.
```

Xét một số bài toán quy hoạch động đơn giản minh họa cho sơ đồ lặp.

Xét biểu thức chỉ chứa các ngoặc ‘(‘ và ‘)’.

Biểu thức ngoặc đúng được định nghĩa như sau:

- ✿ () là biểu thức ngoặc đúng,
- ✿ Nếu \mathbf{A} là một biểu thức ngoặc đúng thì (\mathbf{A}) , $\mathbf{A}()$ và (\mathbf{A}) cũng là những biểu thức ngoặc đúng.

Cho n ngoặc mở (và n ngoặc đóng) (gọi tắt là n cặp ngoặc). Số lượng biểu thức ngoặc đúng chứa n cặp ngoặc là số Catalan thứ n , ký hiệu C_n .

Ví dụ, với $n = 3$ ta có $C_3 = 5$:

$$()()() \quad ()(()) \quad ((()) \quad ((()) \quad ((())$$

Cho số nguyên n . Hãy xác định 9 chữ số cuối cùng của C_n , không dẫn xuất các số 0 không có nghĩa ở đầu.

Dữ liệu: Vào từ file văn bản CATALAN.INP gồm một dòng chứa số nguyên n ($1 \leq n \leq 10^4$).

Kết quả: Đưa ra file văn bản CATALAN.OUT một số nguyên – số tìm được.

Ví dụ:

| |
|-------------|
| CATALAN.INP |
| 3 |

| |
|-------------|
| CATALAN.OUT |
| 5 |



Giải thuật: sơ đồ lặp (Quy hoạch động đơn giản).

Số Catalan có nhiều ứng dụng trong lý thuyết cũng như trong thực tế và có nhiều cách tính khác nhau:

$$C_0 = 1 \quad \text{và} \quad C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$$

$$C_0 = 1 \quad \text{và} \quad C_n = \frac{2(2n-1)}{n+1} C_{n-1}$$

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{1}{2n+1} \binom{2n+1}{n} = \binom{2n}{n} - \binom{2n}{n-1}$$

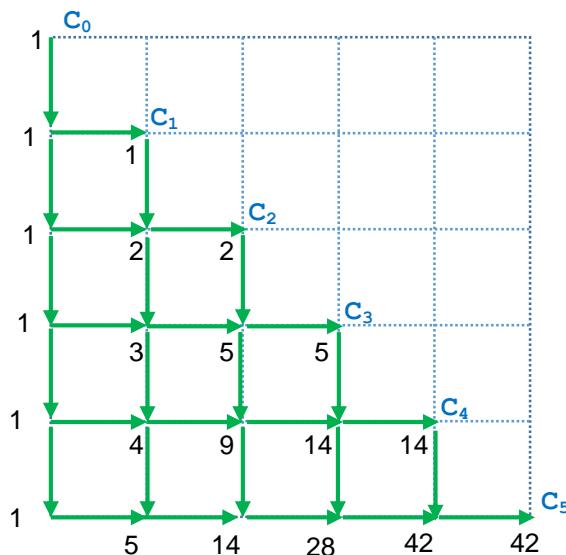


Với yêu cầu của đầu bài, có thể sử dụng công thức lặp thứ nhất hoặc thứ 3 nhưng việc dẫn xuất kết quả khá rắc rối với cùng độ phức tạp $O(n^2)$.

Ta có thể xây dựng sơ đồ tính toán đơn giản hơn ở mỗi bước lặp.

Trên lưới ô vuông kích thước $n \times n$ xét các đường đi từ góc trên trái xuống góc dưới, mỗi bước đi xuống tương ứng với (, bước đi ngang – tương ứng với), các đường đi phải thỏa mãn 2 tính chất sau:

- Từ mỗi nút chỉ có thể đi xuống dưới hoặc sang phải,
- Số bước đi sang phải không được vượt quá số bước đi xuống dưới tính từ đầu đường đi.



Để dàng thấy rằng từ dòng ($i-1$) ta có thể tính ra các số trên dòng i . Gọi các số trên dòng $i-1$ là $D = (d_0, d_1, d_2, \dots, d_{i-2}, d_{i-1}, d_i, \dots)$, $d_j = 0$ với $j > i-1$.

Ta có công thức lặp cập nhật tại chỗ thông tin:

$$d_j = d_j + d_{j-1}, \quad j = 1 \div i.$$

Công thức trên chỉ sử dụng phép cộng, vì vậy việc dẫn xuất kết quả sẽ đơn giản hơn so với các công thức lặp khác.

Kết quả: biến d_n .

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "catalan."
#define times clock() / (double)1000
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int p=1e9;
int n,d[10001]={0};

int main()
{
    fi>>n;
    d[0]=1;
    for(int i=1;i<=n;++i)
        for(int j=1;j<=i;++j) d[j]=(d[j]+d[j-1])%p;
    fo<<d[n];
    fo<<"\nTime: "<<times<<" sec";
}
```



LÁT ĐƯỜNG VIỀN

Tên chương trình: PAVE.CPP

Người ta dùng 2 loại gạch có kích thước 1×2 và 2×2 để lát một đường viền kích thước $n \times 2$. Số lượng mỗi loại gạch là vô hạn. Khi dùng, các viên gạch kích thước 1×2 có thể đặt ngang hay dọc tùy ý. Hai cách lát gọi là khác nhau nếu tìm thấy một vị trí i trên đường viền được lát bằng các cách khác nhau.

Ví dụ với $n = 3$ ta có 5 cách lát khác nhau:



Hãy đưa ra số cách lát theo mô đun $10^9 + 7$.

Dữ liệu: Vào từ file văn bản PAVE.INP gồm một dòng chứa số nguyên n ($1 \leq n \leq 10^5$).

Kết quả: Đưa ra file văn bản PAVE.OUT một số nguyên – số cách lát theo mô đun $10^9 + 7$.

Ví dụ:

| PAVE.INP | PAVE.OUT |
|----------|----------|
| 3 | 5 |



Giải thuật: sơ đồ lắp (Quy hoạch động đơn giản).

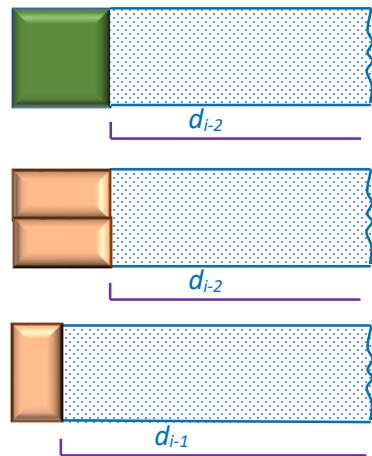
Với $n = 1$ có một cách lát,

Với $n = 2$ có 3 cách lát.



Xét $n \geq 3$: Gọi d_i là số cách lát đường viền độ dài i , ta có:

$$d_i = d_{i-1} + 2 \times d_{-2}$$



Độ phức tạp của giải thuật: $O(n)$.

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "pave."
#define times clock() / (double)1000
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int p=1e9+7;
int64_t d2=1,d3=3,d1;
int n;
int main()
{
    fi>>n;
    if(n==1) d3=1;
    else
        if(n>2) for(int i=2;i<n;++i)
            {d1=d2; d2=d3; d3=(d2+2*d1)%p; }
    fo<<d3;
    fo<<"\nTime: "<<times<<" sec";
}
```



HAI SỐ 1

Tên chương trình: TWOONE.CPP

Cho số nguyên dương n . Hãy xác định số lượng số nguyên dương không lớn hơn n và trong dạng biểu diễn nhị phân của các số đó không có hai số 1 đứng tiếp nhau.

Ví dụ, với $n = 6$ ta có 4 số cần tìm: $1_{10} = 1_2$, $2_{10} = 10_2$, $4_{10} = 100_2$, $5_{10} = 101_2$.

Dữ liệu: Vào từ file văn bản TWOONE.INP:

- ⊕ Dòng đầu tiên chứa một số nguyên t ($1 \leq t \leq 10^5$), trong đó t – số lượng tests,
- ⊕ Dòng thứ i trong t dòng sau chứa số nguyên n_i ($1 \leq n_i \leq 10^{18}$).

Kết quả: Đưa ra file văn bản TWOONE.OUT t số nguyên – các kết quả tìm được, mỗi số trên một dòng.

Ví dụ:

| TWOONE.INP | TWOONE.OUT |
|------------|------------|
| 4 | 4 |
| 6 | 7 |
| 10 | 13 |
| 32 | 88 |
| 446 | |



Giải thuật: Sơ đồ lắp (Quy hoạch động đơn giản), Số Fibonacci.

Xét các số có i bít có nghĩa, gọi a_i – số xâu bít độ dài m bắt đầu bằng 0 và không có hai ký tự 1 đứng liên tiếp, b_i – tổng số xâu bít không có hai ký tự 1 đứng liên tiếp, bắt đầu bằng 1 và có độ dài không vượt quá i ,

Đặt $a_0 = 1$, $b_0 = 0$, ta có: $a_1 = 1$, $b_1 = 1$, với $i > 1$ có:

$$\textcolor{blue}{\oplus} \quad a_i = a_{i-1} + a_{i-2},$$

$$\textcolor{blue}{\oplus} \quad b_i = b_{i-1} + a_{i-1}.$$

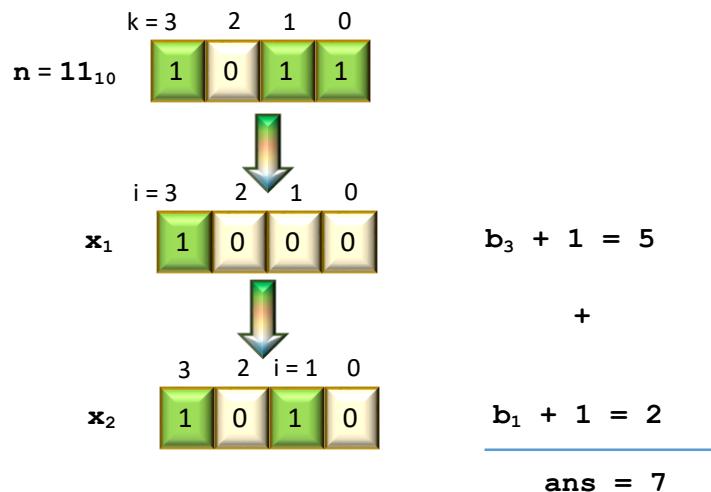
Các số nguyên n cần xét không vượt quá 10^{18} vì vậy các xâu bít tương ứng có độ dài không quá 64. Gọi k là vị trí bít 1 trái nhất của n .

Tạo các số mới bằng cách xóa hết các bít 1 trong n , chỉ giữ lại 1 bít 1 trái nhất, giữ lại 2 bít 1 trái nhất, giữ lại 3 bít 1 trái nhất, ... Quá trình tạo số mới chấm dứt khi không còn bít 1 để xóa hoặc lần đầu tiên gặp bít 1 thứ 2 trong cặp 2 bít 1 liên tiếp.

Giả thiết x_j là một trong các số mới nhận được, trong đó j là vị trí bít 1 phải nhất, khi đó số lượng số thỏa mãn yêu cầu đề bài với x_j sẽ là b_{j+1} .

Kết quả là tổng các số lượng tìm được.

Ví dụ với $n = 11_{10}$:



Độ phức tạp của giải thuật: $O(t)$.

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "twoone."
#define times clock()/(double)1000
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int64_t n,ans;
int64_t a[64],b[64];
int k,t,d;

int main()
{
    a[0]=1;b[0]=0; a[1]=1; b[1]=1;
    for(int i=2;i<=63;++i)
    {
        a[i]=a[i-1]+a[i-2];
        b[i]=b[i-1]+a[i-1];
    }
    fi>>t;
    for(int j=1;j<=t;++j)
    {
        fi>>n;
        for(int i=63;i>=0;--i) if( (n>>i) &1) {k=i; break;}
        ans=b[k];
        d=1;
        for(int i=k-1;i>=0;--i)
        {
            if( (n>>i) &1)
            {
                if( (n>>(i+1)) &1) {ans+=b[i]; break;}
                ans+=b[i]; ++d;
            }
        }
        fo<<ans+d<<'\n';
    }
    fo<<"\nTime: "<<times<<" sec";
}
```



Trong máy ATM có k ngăn lưu trữ các tờ tiền mệnh giá a_1, a_2, \dots, a_k , mỗi mệnh giá có số lượng tờ đủ nhiều.

Khi khách hàng có yêu cầu rút n đồng, chương trình điều khiển sẽ xác định xem có cách trả đúng n đồng hay không. Nếu có cách trả thì xác định số tờ tiền ít nhất cần sử dụng và số lượng tờ mỗi mệnh giá cần đưa ra.

Ví dụ với 3 mệnh giá 10, 60, 100 và số tiền cần rút là 130 thì số lượng tờ tiền ít nhất cần sử dụng là 3, phương án đưa ra là 2 tờ mệnh giá 60 và 1 tờ mệnh giá 10.

Dữ liệu: Vào từ file văn bản ATM.INP:

- ⊕ Dòng đầu tiên chứa một số nguyên k ($1 \leq k \leq 20$),
- ⊕ Dòng thứ 2 chứa k số nguyên a_1, a_2, \dots, a_k theo thứ tự tăng dần ($1 \leq a_i < a_j \leq 10^5$, $1 \leq i < j \leq k$),
- ⊕ Dòng thứ 3 chứa số nguyên n ($1 \leq n \leq 10^6$).

Kết quả: Đưa ra file văn bản ATM.OUT nếu không có cách trả thì đưa ra số -1, trong trường hợp có cách trả: dòng thứ nhất chứa một số nguyên – số lượng tờ tiền cần sử dụng, dòng thứ 2 chứa k số nguyên, số thứ i cho biết phải sử dụng bao nhiêu tờ tiền mệnh giá a_i . Nếu tồn tại nhiều cách chi trả – đưa ra cách tùy chọn.

Ví dụ:

| ATM.INP | ATM.OUT |
|----------------------------|--------------------|
| <pre>3 10 60 100 130</pre> | <pre>3 1 2 0</pre> |



Giải thuật: Quy hoạch động.

Gọi $f[j]$ – cách chi trả tối ưu số tiền j ($0 \leq j \leq n$), nếu không có cách chi trả thì $f[j] = +\infty$, $f[0]=0$, $f[j] = +\infty$ với $0 < j < a_1$.

Nếu $j \geq a_i$: ta có thể sử dụng một lần tờ mệnh giá a_i và số tờ tiền phải sử dụng sẽ là $f[j-a[i]] + 1$,

Ghi nhận các cách chi trả tối ưu đối với j :

$$f[j] = \min\{f[j-a[1]], f[j-a[2]], \dots, f[j-a[k]]\} + 1$$

Lần lượt duyệt với j từ a_1 đến n và có số lượng tờ cần dùng ít nhất ở $f[n]$.

Xác định cách chi trả (khi $f[n] < +\infty$):

Điều kiện có sử dụng đồng tiền mệnh giá a_i : $f[n] = f[n-a[i]] + 1$,

Nếu có sử dụng đồng tiền mệnh giá a_i : giảm n ($n=a[i]$), tích lũy tàn số sử dụng và tiếp tục kiểm tra.

Độ phức tạp của giải thuật: O(k×n).

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "atm."
#define times clock() / (double)1000
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int INF=1e9;
int n,k;

int main()
{
    fi>>k;
    vector<int>a(k);
    for(int i=0;i<k;++i) fi>>a[i];
    fi>>n;
    vector<int>f(n+1); f[0]=0;
    for(int j=1;j<a[1];++j) f[j]=INF;
    for(int j=a[1];j<=n;++j)
    {
        f[j]=INF;
        for(int i=0;i<k;++i)
            if(j>=a[i] && f[j-a[i]]+1<f[j]) f[j]=f[j-a[i]]+1;
    }
    if(f[n]==INF) {fo<<"-1";return 0;}
    vector<int>b(k,0);
    fo<<f[n]<<'\
';
    while(n>0)
        for(int i=0;i<k;++i)
            if(f[n-a[i]]+1==f[n]) {++b[i]; n-=a[i]; break;}
    for(int i=0;i<k;++i) fo<<b[i]<<' ';
    fo<<"\nTime: "<<times<<" sec";
}
```

