

MỤC LỤC ATHENA XV

CÂY PHÂN KHÚC (Segment Tree)	5
1. Cây phân khúc cơ sở	5
Cấu trúc cây	5
Khởi tạo cây	6
Thay đổi giá trị của một nút	8
Tính tổng các phần tử	10
Chương trình minh họa	13
2. Cây phân khúc không đồng bộ – mô hình tổng quát	17
Khái niệm cây không đồng bộ	17
Cập nhật trên đoạn	17
4. Cây quản lý Max có cập nhật đoạn	25
5. Xác định Max trên đoạn con	28
6. Xác định Min/Max và số lần xuất hiện trong đoạn	35
7. Xác định số nhỏ nhất lớn hơn hoặc bằng số cho trước trên đoạn con	39
8. Gán giá trị cho đoạn	46
9. Cây phân khúc hai chiều	54
Quản lý tổng với mảng hai chiều có cập nhật giá trị phần tử mảng	54
Bài toán	54
Xây dựng cây	54
Xử lý truy vấn tính tổng	55
Xử lý truy vấn cập nhật	56
Chương trình minh họa	58
Tìm phần tử 0 thứ k trong đoạn	60
Tổng Minkowski	64
1. Định nghĩa	64
2. Xây dựng tổng hai đa giác lồi	64
3. Ứng dụng	65
3.1 Kiểm tra giao hai đa giác lồi	66
3.2 Tính khoảng cách giữa 2 đa giác lồi	66

Tìm kiếm tam phân	67
1. Hàm unimodal	67
2. Giải thuật tìm kiếm	69
GIẢI THUẬT MANAKER TÌM PALINDROME	70
<i>Bài toán</i>	70
<i>Giải thuật</i>	70
<i>Đánh giá độ phức tạp</i>	72
NGUYÊN LÝ BÙ – TRỪ	73
1. Phát biểu nguyên lý	73
1.1 Mô tả phương pháp	73
1.2 Trên ngôn ngữ tập hợp	73
1.3 Biểu đồ Venn	74
1.3 Chứng minh	75
2. Ứng dụng	76
2.1 Bài toán hoán vị	76
2.2 Dãy số (0, 1, 2)	76
2.3 Số lượng nghiệm nguyên của phương trình	77
2.4 Số lượng số nguyên tố cùng nhau trên đoạn thẳng	78
2.5 Số lượng bội trong đoạn	80
2.6 Số lượng đường đi	82
2.7 Ghép cặp	94
Hệ số nhị thức Newton	97
Công thức tính	97
Tính chất	98
Tính hệ số nhị thức theo mô đun p	99
PHÂN RÃ BẬC CĂN N	101
Cấu trúc dữ liệu trên cơ sở phân rã căn n	101
Bài toán	101
Xây dựng cấu trúc	101
Chương trình minh họa tính tổng	103
Cập nhật	104
Phân rã các truy vấn	104
Tăng giá trị trên đoạn	105

Bổ sung và loại bỏ cạnh của đồ thị.....	105
Sơ đồ tổng quát xử lý truy vấn ở chế độ offline trên cơ sở phân rã	106
Giải thuật Mo (Mo's Algorithm).....	106
GIẢI THUẬT KRUSKAL.....	111
Một số tính chất của cây khung cực tiểu.....	111
<small>MỘT SỐ TÍNH CHẤT CỦA CÂY KHUNG CỰC TIỂU</small>	
Nguyên lý giải thuật Kruskal	111
<small>NGUYÊN LÝ GIẢI THUẬT KRUSKAL</small>	
Giải thuật đơn giản	113
<small>GIẢI THUẬT ĐƠN GIẢN</small>	
Giải thuật cải tiến	115
<small>GIẢI THUẬT CẢI TIẾN</small>	
GIẢI THUẬT RABIN – KARP	117
SỐ LƯỢNG CÁCH LIÊN THÔNG HÓA ĐỒ THỊ	119
Mã Prüfer	119
<small>MÃ PRÜFER</small>	
Xây dựng mã Prüfer.....	119
Xây dựng mã Prüfer với độ phức tạp tuyến tính.....	121
Một số tính chất của mã Prüfer.....	123
Khôi phục cây theo mã Prüfer.....	123
Khôi phục cây với độ phức tạp tuyến tính.....	124
Tương ứng đơn trị giữa cây và mã Prüfer	125
Công thức Cayley (Cayley's Formula)	126
<small>CÔNG THỨC CAYLEY</small>	
Số cách bổ sung ít cạnh nhất để có đồ thị liên thông	127
Một số bài toán ứng dụng	128
<small>MỘT SỐ BÀI TOÁN ỨNG DỤNG</small>	
Bài 1 MÃ PRUFER.....	128
Bài 2. MANH MỐI.....	132
BÀI TẬP THEO CÁC CHUYÊN ĐỀ KHÁC NHAU	136
VZ14. ĐẠO CHƠI Ở BROOKLYN Tên chương trình: WALK.CPP	136
VZ15. THỜI GIAN Tên chương trình: INTERVAL.CPP	144
VZ16. CẤP MÃ TRUY NHẬP Tên chương trình: PAIR_K.CPP	146
VZ17. KHỞI NGHIỆP Tên chương trình: STARTUP.CPP	149
VZ18. BÃI SINH Tên chương trình: MUD.CPP	155
VZ19. MÃ SỬA SAI Tên chương trình: SH_CODE.CPP	165
VZ20. TIẾT KIÊM Tên chương trình: PROVIDENT.CPP	169
VZ21. TRUNG VI Tên chương trình: MEDIAN.CPP	171

VZ22. BÁO CÁO	Tên chương trình: <i>RAPORTS.CPP</i>	173
VZ23. GIẢI MÃ	Tên chương trình: <i>DECODING.CPP</i>	176
VZ24. BẰNG NHAU	Tên chương trình: <i>EQ.CPP</i>	181
VZ25. ĐƠN VỊ	Tên chương trình: <i>UNIT.CPP</i>	187
VZ26. CỘNG TRỪ	Tên chương trình: <i>PM.CPP</i>	190

CÂY PHÂN KHÚC (Segment Tree)

Cây phân khúc là cấu trúc dữ liệu lưu trữ thông tin mảng $\mathbf{A} = (\mathbf{a}_0, \mathbf{a}_1, \dots, \mathbf{a}_{n-1})$ cho phép thực hiện có hiệu quả các truy vấn trên đoạn $[lf..rt]$ theo nhiều phép tìm kiếm khác nhau. Nếu \mathbf{A} là dãy số nguyên thì các phép tìm kiếm có thể là tổng, min, max, gcd, ...

Cấu trúc dữ liệu này cũng cho phép thay đổi giá trị một phần tử \mathbf{a}_i , gán giá trị val cho các phần tử của \mathbf{A} trong đoạn $[lf..rt]$ hoặc tăng mỗi phần tử trong đoạn trên lên val .

Cấu trúc dữ liệu này đơn giản, đủ linh hoạt và hiệu quả để giải quyết nhiều lớp bài toán tìm kiếm. Mỗi phép xử lý cập nhật và tìm kiếm được thực hiện với độ phức tạp $O(\log n)$.

Đặc điểm quan trọng của cấu trúc là bộ nhớ sử dụng để làm việc tỷ lệ tuyến tính với kích thước mảng. Bộ nhớ đòi hỏi để quản lý \mathbf{A} là $4 \times n$.

Cây phân khúc là phương án đơn giản hóa của [cấu trúc TREAP](#).

1. Cây phân khúc cơ sở

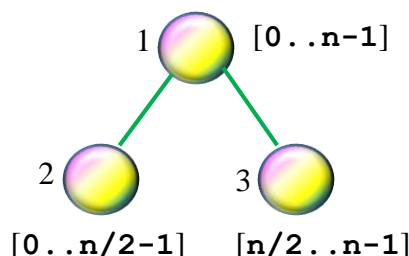
Cấu trúc cây

Để đơn giản trong mô tả ta sẽ xét trường hợp mảng \mathbf{A} là dãy số nguyên và hàm tìm kiếm là tổng (nếu chỉ tìm kiếm theo tổng thì [hàm tổng tiền tố](#) hay [cây Fenwick](#) sẽ làm việc hiệu quả hơn).

Đầu tiên xét trường hợp $n = 2^k$.

Nút gốc của cây được đánh số là 1 và quản lý tổng các phần tử đoạn $[0..n-1]$.

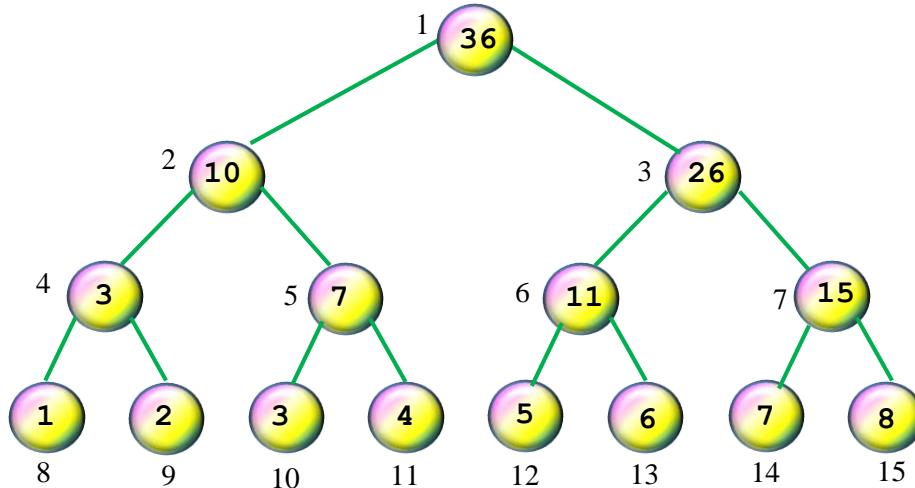
Nút gốc có 2 nút con. Nút con trái (đánh số 2) quản lý tổng các phần tử đoạn $[0..n/2-1]$. Nút con phải (đánh số 3) quản lý tổng các phần tử đoạn $[n/2..n-1]$.



Mỗi nút con lại có 2 nút trái và phải quản lý hai nửa đoạn được nút con quản lý.

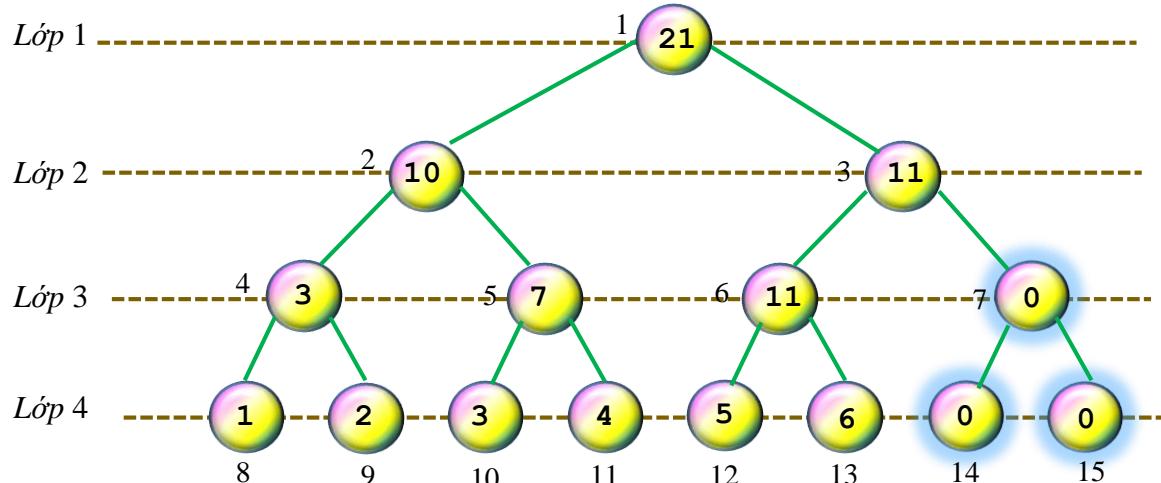
Quá trình phân chia tiếp tục cho đến khi nút quản lý đoạn chỉ chứa một phần tử. Những nút này sẽ là nút lá của cây.

Ví dụ, với $n = 8$ và $\mathbf{A} = (1, 2, 3, 4, 5, 6, 7, 8)$ ta có cây:



Trong trường hợp tổng quát ta có thể tìm k nhỏ nhất thỏa mãn điều kiện $2^k \geq n$ và xây dựng cây với $m = 2^k$, cho $a_j = 0$ với $j \geq n$.

Ví dụ, với $n = 6$, $\mathbf{A} = (1, 2, 3, 4, 5, 6)$ ta có cây:



Việc mở rộng kích thước đến 2^k không tốn nhiều bộ nhớ và làm đơn giản hóa sơ đồ xử lý.

Các nút của cây được chia thành k lớp:

- ⊕ Lớp 1 chứa nút gốc,
- ⊕ Lớp j chứa các nút con của các nút lớp $j-1$, $j = 2 \div k$.

Đây là trường hợp cây được tổ chức với móc nối ẩn: mỗi nút u ở các lớp từ 1 đến $k-1$ có 2 nút con là $2 \times u$ (*nút con trái*) và $2 \times u + 1$ (*nút con phải*).

Ngược lại, mỗi nút v ở các lớp từ k đến 2 có nút cha là $v/2$.

Khởi tạo cây

Có 2 cách:

- Đi từ lớp **k** lên lớp 1 (*Sơ đồ lặp*),
- Đi từ lớp 1 xuống lớp **k** (*Sơ đồ đệ quy*).

Sơ đồ lặp thực hiện nhanh hơn nhưng khó triển khai thực hiện truy vấn, đặc biệt là trong trường hợp có các phép cập nhật **A**.

Sơ đồ đệ quy dễ mô tả các phép xử lý khi có các phép cập nhật và mang tính chất tổng quát, dễ dàng áp dụng cho trường hợp khi cây được tổ chức với các mốc nối tường minh.

Khai báo dữ liệu:

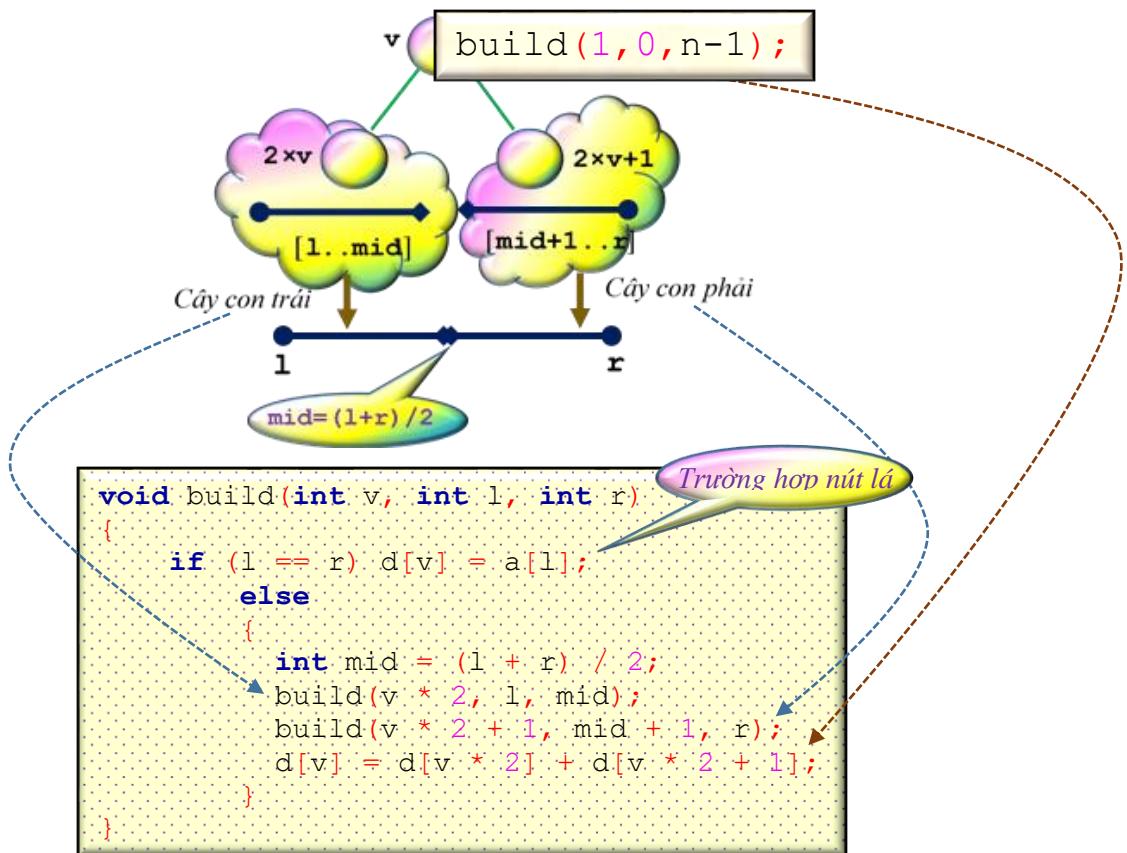
- `vector<int> t` - Lưu trữ cây,
- `vector<int> a` - Lưu trữ mảng dữ liệu **A** (Có thể vòng tránh, không cần lưu trữ **A**).

Việc xin cấp phát bộ nhớ sẽ thực hiện khi biết **n**.

Gải thuật đệ quy:

Hàm `void build(int v, int l, int r)` xác định giá trị ở nút gốc **v** của cây con quản lý các nút từ **l** đến **r**.

Đệ quy được tổ chức duyệt theo trình tự postfix (thứ tự sau): Tính giá trị cây con trái, tính giá trị cây con phải, sau đó tính giá trị gốc.

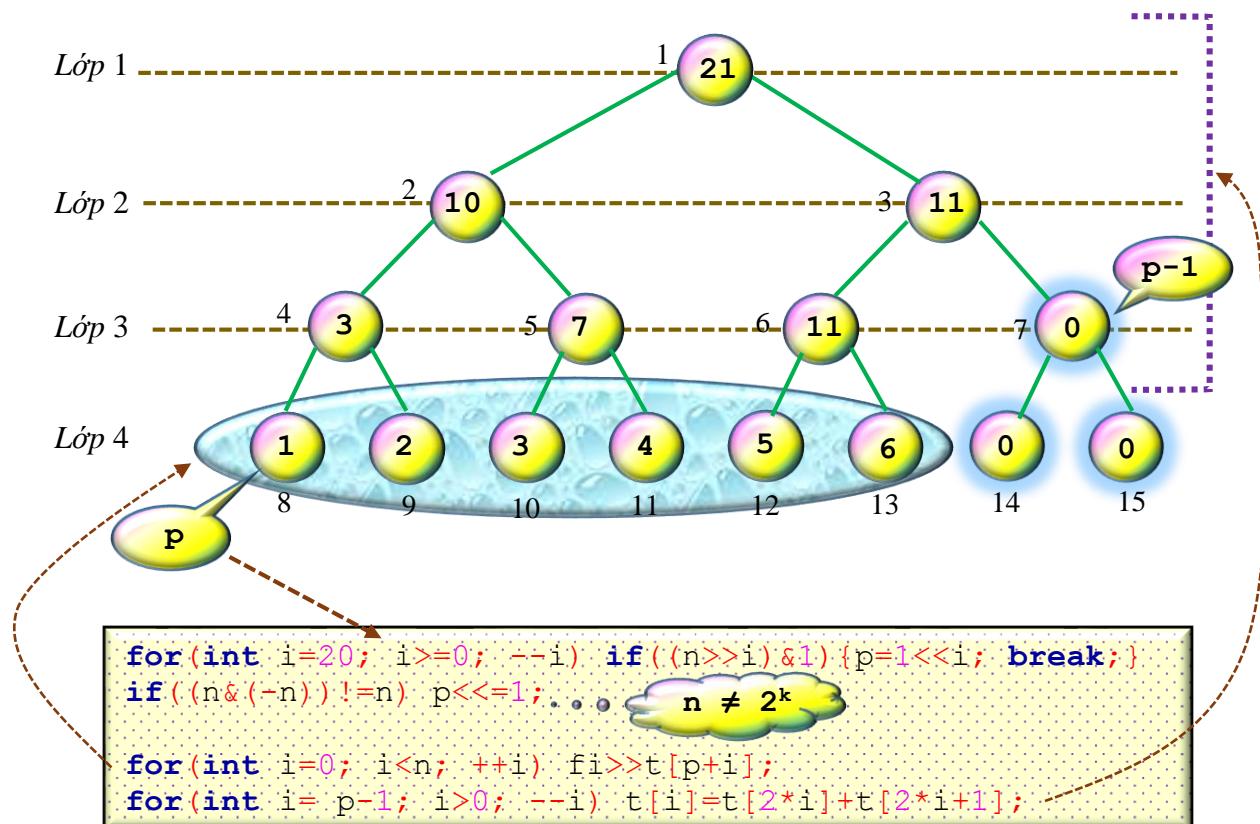


Lời gọi ở chương trình chính:

Sơ đồ lặp:

Bố trí dữ liệu vào các nút lá (các nút mức **k**),

Tính giá trị các nút mức **k-1**, sau đó – mức **k-2**, ..., mức **1**.

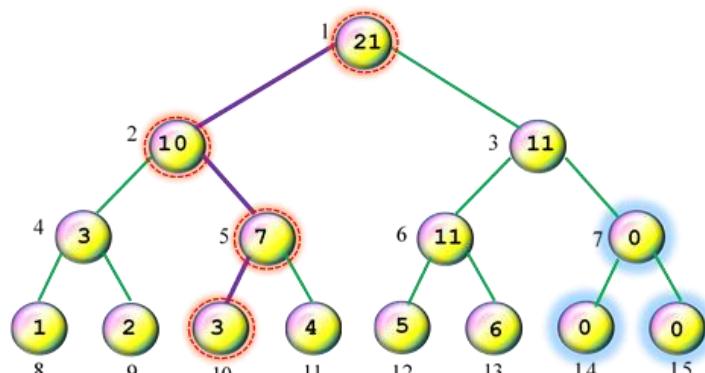


Độ phức tạp của giải thuật: $O(n \log n)$

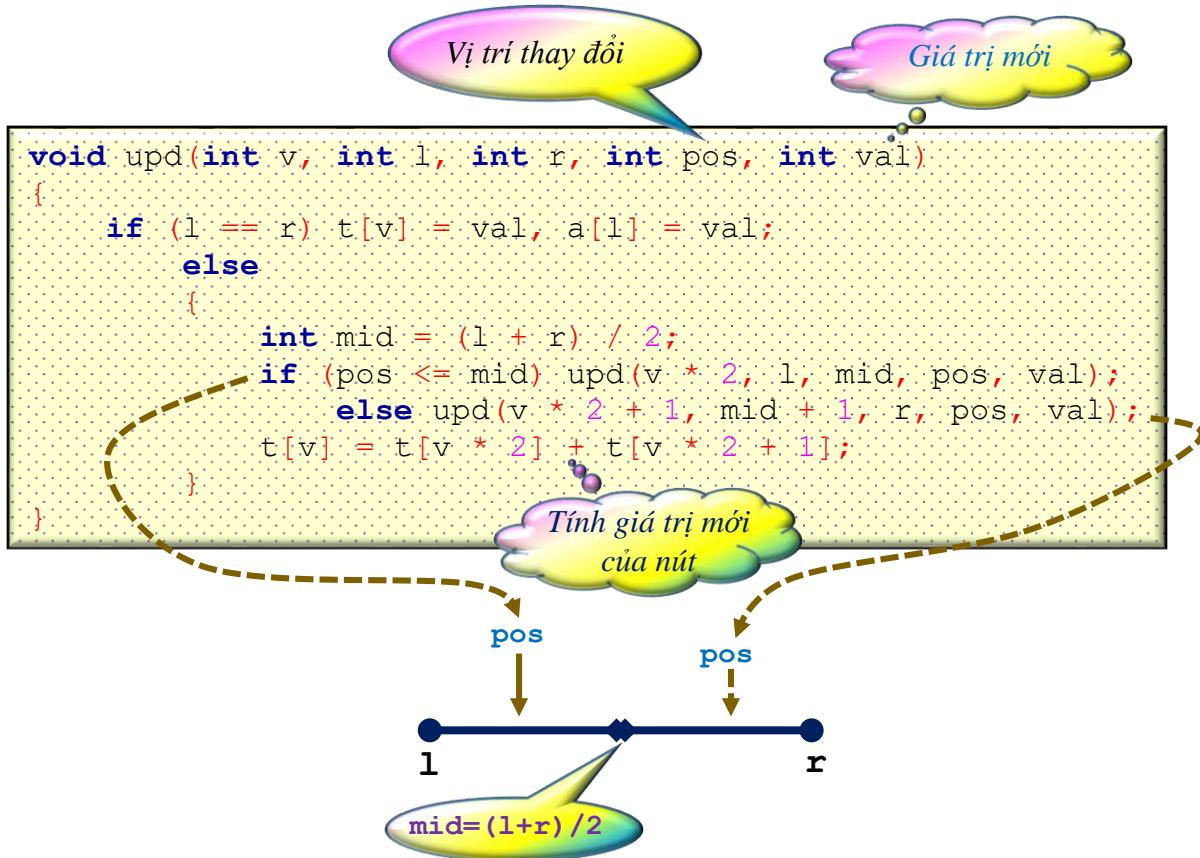
Thay đổi giá trị của một nút

Trong quá trình xử lý có thể xuất hiện một vài yêu cầu thay đổi giá trị nút **a_i** thành **x**, tức là cần thực hiện phép gán **a[i]=x** với một **i** và **x** nào đó.

Việc thay đổi giá trị một **a_i** chỉ dẫn tới việc thay đổi giá trị các nút trên đường đi từ nút lá **p+i** tới nút gốc **1** của cây.



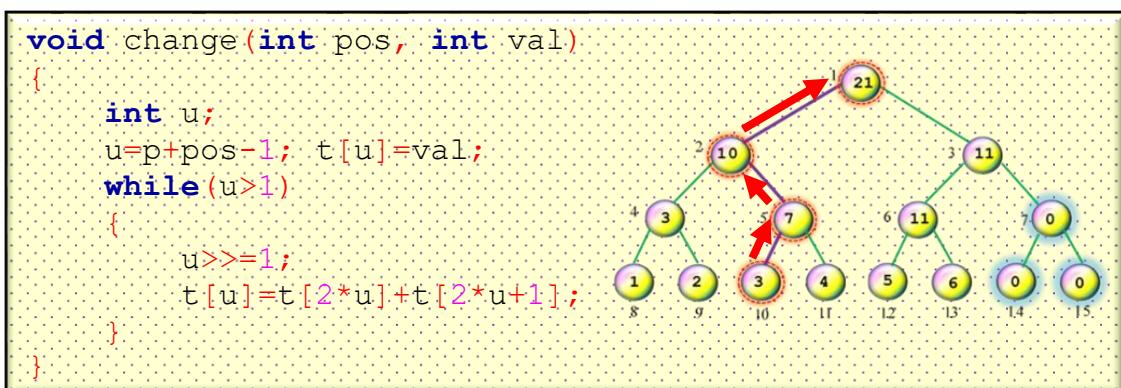
Giải thuật đệ quy:



Lời gọi ở chương trình chính:

upd(1, 0, n-1, pos, x);

Sơ đồ lặp:



Từ nút lá, với giá trị mới, theo nút cha (từ v tới $v/2$) đi ngược về nút 1 và cập nhật giá trị các nút trên đường đi.

Việc tăng giá trị a_i thành a_i+d (d có thể dương hoặc âm) được thực hiện hoàn toàn tương tự (thay $a[1]=val$ bằng $a[1]+=d$ hay $t[u]=val$ bằng $t[u]+=val$). Có thể đưa 2 phép thay đổi về cùng một loại, nhưng trước đó phải

thực hiện truy vấn (với chi phí $O(\log n)$) để xác định giá trị hiện tại của biến cần thay đổi. Vì vậy vẫn hợp lý hơn nếu xây dựng hàm riêng cho mỗi loại thay đổi.

Các hàm tăng giá trị một biến:

Hàm đệ quy:

```

void incr_1(int v, int l, int r, int pos, int d)
{
    if (l == r) t[v] = val, a[l] += val;
    else
    {
        int mid = (l + r) / 2;
        if (pos <= mid) upd(v * 2, l, mid, pos, val);
        else upd(v * 2 + 1, mid + 1, r, pos, val);
        t[v] = t[v * 2] + t[v * 2 + 1];
    }
}

```

Sơ đồ lặp:

```

void incr_1(int pos, int val)
{
    int u;
    u=p+pos-1; t[u]+=val;
    while(u>1)
    {
        u>>=1;
        t[u]=t[2*u]+t[2*u+1];
    }
}

```

Tính tổng các phần tử

Xét việc tính tổng các phần tử thuộc đoạn $[ql, qr]$, trong đó $ql < qr$. Nếu chỉ đơn thuần tính tổng khi dãy \mathbf{A} đã biết trước và không thay đổi trong toàn bộ quá trình xử lý thì công cụ hàm *Tổng tiền tố* (*Prefix Sum*) có thể thực hiện với độ phức tạp $O(1)$. Nhưng hàm Tổng tiền tố không thích hợp khi dãy \mathbf{A} thay đổi hoặc hàm xử lý không phải là tổng (ví dụ, \min , \max , \gcd , ...).

Sơ đồ tính tổng các phần tử trong một đoạn dễ thực hiện nhất và từ sơ đồ này ta có thể dễ dàng mở rộng việc tính toán với các loại hàm khác áp dụng trên dãy số.

Tồn tại giải thuật đệ quy và giải thuật theo công thức lặp tính tổng. Giải thuật lặp thực hiện nhanh hơn giải thuật đệ quy vài ba lần, nhưng kém chặt chẽ về lý thuyết so với cách tiếp cận đệ quy.

Giải thuật đệ quy:

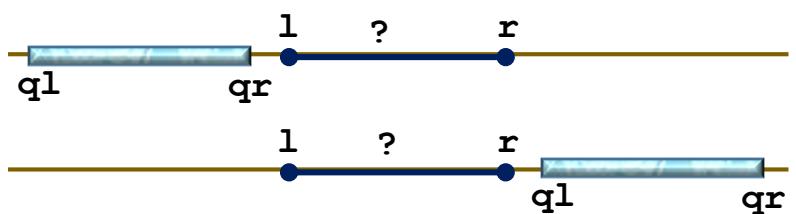
Hàm đệ quy

```
int get(int v, int l, int r, int ql, int qr)
```

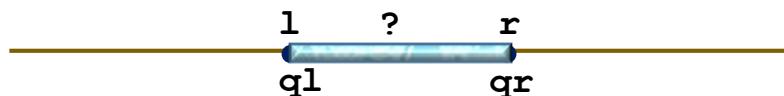
tính tổng các phần tử thuộc khoảng $[ql, qr]$ *nằm trong cây con* đỉnh **v**, quản lý các nút trong thuộc khoảng $[l, r]$.

Các trường hợp có thể xảy ra:

- Đoạn cần tính nằm ngoài cây con: kết quả ứng với cây con này sẽ là 0.



- Đoạn cần tính trùng khớp với phạm vi quản lý của cây con: kết quả ở gốc cây con.

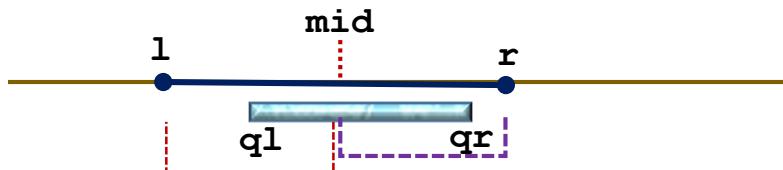


cây con.

- Các trường hợp còn lại: Đoạn cần tính không trùng khớp với phạm vi quản lý của cây con.

Gọi **mid** – điểm giữa của $[l, r]$, $mid = (l+r)/2$.

Kết quả cần tìm gồm 2 phần: phần giao của nửa cây trái với $[ql, qr]$ và phần giao của nửa cây phải với $[ql, qr]$.



Với mỗi nửa cây con – có thể xảy ra một trong 3 trường hợp đã nêu → ta có đệ quy!

Cây con trái có đỉnh là $2 \times v$, còn cây con phải – đỉnh $2 \times v + 1$.

Sớm hay muộn việc phân chia sẽ dẫn đến một trong hau trường hợp đầu.

```

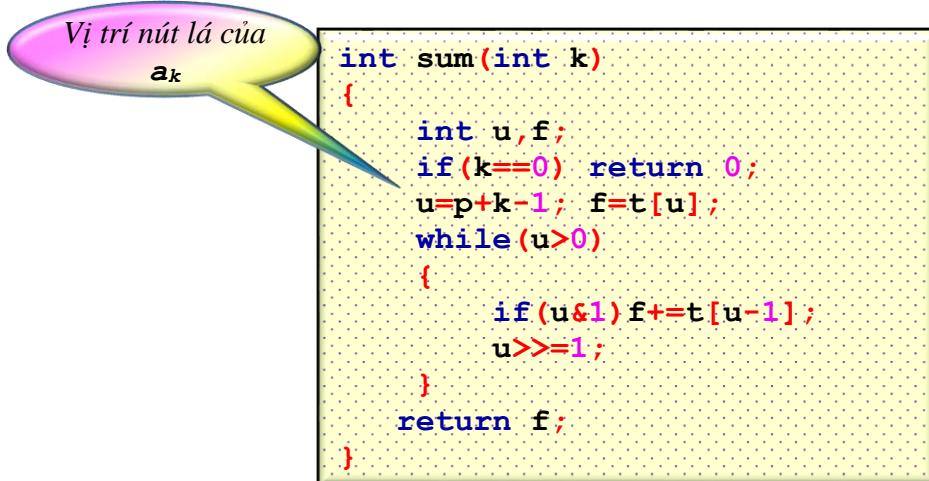
int get(int v, int l, int r, int ql, int qr)
{
    if (l > qr || r < ql) return 0;
    if (ql <= l && r <= qr) return t[v];
    int mid = (l + r) / 2;
    return get(v * 2, l, mid, ql, qr) +
        get(v * 2 + 1, mid + 1, r, ql, qr);
}

```

Sơ đồ lắp:

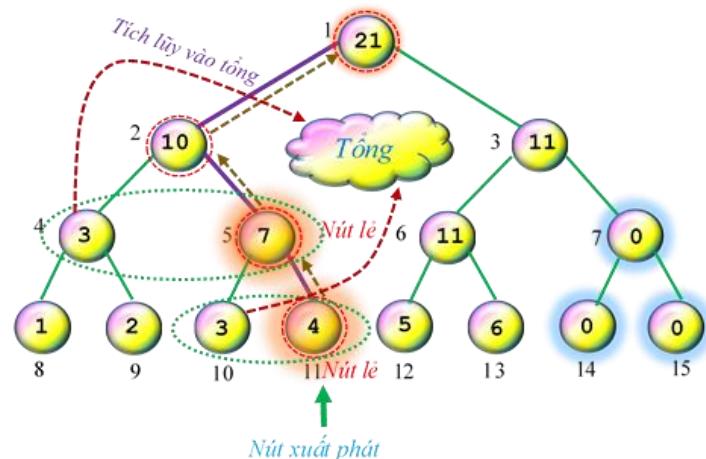
Xây dựng hàm tính tổng $k+1$ phần tử đầu tiên của dãy:

$$\text{sum}(k) = a_0 + a_1 + a_2 + \dots + a_k$$



Duyệt đường đi từ đỉnh lá tương ứng về đỉnh 1 theo công thức từ đỉnh u lên đỉnh $u/2$. Ban đầu tổng cần tìm nhận giá trị $t[u]$. Ở mỗi bước, khi chưa tới gốc, nếu u là lẻ thì tích lũy giá trị ở nút chẵn cùng mức bên trái (tức là $t[u-1]$) vào tổng.

Có thể dừng duyệt sớm khi đi tới nút ở nhánh trái nhất của cây (với dấu hiệu $u \& (-u) == u$).



Tổng các phần tử $a_{lf} + a_{lf+1} + \dots + a_{rt} = \text{sum(rt)} - \text{sum(lf-1)}$

Việc thay đổi giá trị một nút không làm thay đổi các hàm tính tổng vì giá trị lưu ở các nút tương ứng với giá trị thực sự của dãy số **A**.

Độ phức tạp tính toán của các hàm: $O(\log n)$.

Chương trình minh họa

Sơ đồ đệ quy

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("Segm_New.inp");
ofstream fo ("intv_tree.out");
int n;
vector<int> t,a,vd;

void build(int v, int l, int r)
{
    if (l == r) t[v] = a[l];
    else
    {
        int mid = (l + r) / 2;
        build(v * 2, l, mid);
        build(v * 2 + 1, mid + 1, r);
        t[v] = t[v * 2] + t[v * 2 + 1];
    }
}

void upd(int v, int l, int r, int pos, int val)
{
    if (l == r) t[v] = val, a[l] = val;
    else
    {
        int mid = (l + r) / 2;
        if (pos <= mid) upd(v * 2, l, mid, pos, val);
        else upd(v * 2 + 1, mid + 1, r, pos, val);
        t[v] = t[v * 2] + t[v * 2 + 1];
    }
}

int get(int v, int l, int r, int ql, int qr)
{
    if (l > qr || r < ql) return 0;
    if (ql <= l && r <= qr) return t[v];
    int mid = (l + r) / 2;
    return get(v * 2, l, mid, ql, qr) +
           get(v * 2 + 1, mid + 1, r, ql, qr);
}

int main()
{
    fi>>n;
    a.resize(n);
    t.assign(4*n+4, 0);
    for(int i=0; i<n; ++i) fi>>a[i];
```

```

build(1,0,n-1);
upd(1,0,n,0,8); // Replace a0 for 8
int x=get(1,0,n,1,4); //Count a1+a2+a3
fo<<"Sum 1..4: "<<x<<'\n';
fo<<"Sum 0..2: "<<get(1,0,n,0,2)<<'\n'; //Count a0+a1
    fo<<"-----\n";
}

```

Input

6
1 2 3 4 5 6

Output

Sum 1..4: 9
Sum 0..2: 10

Sơ đồ lắp

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("intv_tree.inp");
ofstream fo ("intv_tree.out");
int n,p;
vector<int> t;

int sum(int k)
{
    int u,v,w,f;
    if(k==0) return 0;
    u=p+k-1; f=t[u];
    while(u>0)
    {
        if(u&1) f+=t[u-1];
        u>>=1;
    }
    return f;
}

void inc_1(int pos, int val) // apos → apos+val
{
    int u;
    u=p+pos-1; t[u]+=val;
    while(u>1)
    {
        u>>=1;
        t[u]=t[2*u]+t[2*u+1];
    }
}

void change(int pos, int val) // apos → apos=val
{
    int u;
    u=p+pos-1; t[u]=val;
    while(u>1)
    {
        u>>=1;
        t[u]=t[2*u]+t[2*u+1];
    }
}

int main()
{
    fi>>n;
    t.assign(4*n+4, 0);

    for(int i=20; i>=0; --i) if((n>>i)&1){p=1<<i; break;}
    if((n&(-n))!=n)p<<=1;
```

```

for(int i=0; i<n; ++i) fi>>t[p+i];
for(int i = p-1; i>0; --i)
    t[i]=t[2*i]+t[2*i+1];

fo<<"Sum 1..3: "<<sum(3)<<endl;
fo<<"Sum 1..4: "<<sum(4)<<endl;
fo<<"Sum 1..1: "<<sum(1)<<endl;
fo<<"Sum 2..5: "<<sum(5)-sum(1)<<endl;
fo<<"Sum 4..6: "<<sum(6)-sum(3)<<endl;

inc_1(3,10);
fo<<"\n-----\n";
fo<<"Sum 1..3: "<<sum(3)<<endl;
fo<<"Sum 1..4: "<<sum(4)<<endl;
fo<<"Sum 1..1: "<<sum(1)<<endl;
fo<<"Sum 2..5: "<<sum(5)-sum(1)<<endl;
fo<<"Sum 4..6: "<<sum(6)-sum(3)<<endl;

change(3,3);
fo<<"\n-----\n";
fo<<"Sum 1..3: "<<sum(3)<<endl;
fo<<"Sum 1..4: "<<sum(4)<<endl;
fo<<"Sum 1..1: "<<sum(1)<<endl;
fo<<"Sum 2..5: "<<sum(5)-sum(1)<<endl;
fo<<"Sum 4..6: "<<sum(6)-sum(3)<<endl;
}

```

Input

6
1 2 3 4 5 6

Output

Sum 1..3: 6
Sum 1..4: 10
Sum 1..1: 1
Sum 2..5: 14
Sum 4..6: 15

Sum 1..3: 16
Sum 1..4: 20
Sum 1..1: 1
Sum 2..5: 24
Sum 4..6: 15

Sum 1..3: 6
Sum 1..4: 10
Sum 1..1: 1
Sum 2..5: 14
Sum 4..6: 15

2. Cây phân khúc không đồng bộ – mô hình tổng quát

Xét cây phân khúc theo phép xử lý \oplus nào đó (tính *tổng*, *min*, *max*, *gcd*, ...) và phép cập nhật \odot với các phần tử trong đoạn $[q_l, q_r]$, ví dụ cộng vào tất cả các phần tử trong đoạn nói trên một giá trị **add** (**add** có thể nhận giá trị âm hoặc dương).

Khái niệm cây không đồng bộ

Cây không đồng bộ là cây mà giá trị ở mỗi nút không phải là giá trị thực của phép \oplus , tuy nhiên các phép xử lý vẫn cho phép dẫn xuất giá trị đúng cho mỗi truy vấn. Tại mỗi nút lưu trữ 2 giá trị: Giá trị **val** – kết quả thực hiện phép \oplus và giá trị không đồng bộ **d** hỗ trợ việc dẫn xuất giá trị thực cần tìm. **d** có thể nhận giá trị \perp trung tính với phép cập nhật \odot (ví dụ 0 với phép cập nhật cộng một số vào các phần tử trong đoạn, tức là $\odot \equiv +$).

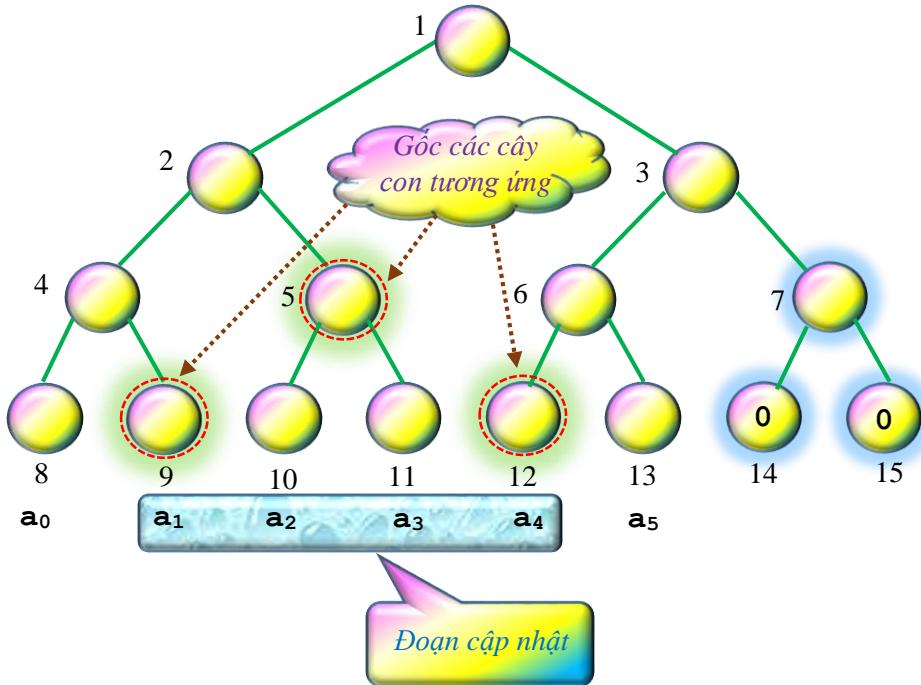
Xét trường hợp phép \odot có tính kết hợp và các phép tính áp dụng trong quản lý đoạn thỏa mãn các tính chất:

- $a \odot (b \odot c) = (a \odot b) \odot c$
- $(a \oplus b) \odot c = (a \odot c) \oplus (b \odot c)$

Cập nhật trên đoạn

Xét việc cập nhật với phép \odot áp dụng lên tất cả các phần tử trong đoạn $[q_l, q_r]$.

Gọi v đỉnh cây con quản lý các phần tử của A trong đoạn $[t1, tr)$. Giá trị d ở đỉnh v sẽ tác động tất cả các phần tử $a_{t1}, a_{t1+1}, \dots, a_{tr-1}$ theo phép cập nhật \odot



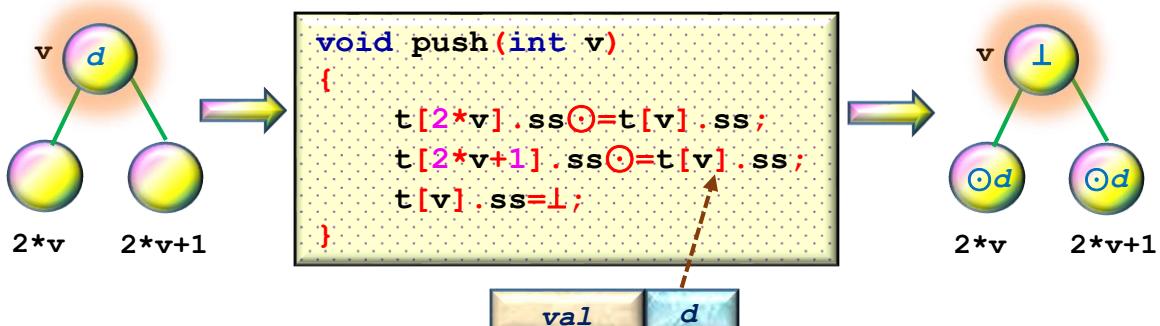
Nội dung của giải thuật cập nhật là xác định một số cây con không giao nhau có hợp các miền quản lý vừa đúng các phần tử trong đoạn $[ql, qr)$ và ghi nhận d tương ứng ở gốc của các cây con này.

Việc xác định đỉnh các cây con có thể thực hiện với độ phức tạp $O(\ln n)$ không phụ thuộc vào độ dài dãy các phần tử cần cập nhật.

Truyền hiệu ứng cập nhật

Trong các giải thuật đệ quy việc duyệt được thực hiện từ trên xuống dưới, tức là từ gốc xuống lá. Từ một nút, trước khi xuống các cây con cần truyền hiệu ứng cập nhật xuống cây con để đảm bảo kết quả nhận được sẽ tương ứng với giá trị thực.

Việc truyền hiệu ứng cập nhật từ nút v được thực hiện bởi hàm **push (v)**.



Cập nhật

```

Gốc cây con
[lf, rt) – đoạn xét trong cây
[qf, qr) – đoạn cần cập nhật
Giá trị cập nhật

void update(int v, int lf, int rt, int qf, int qr, int add)
{
    if [lf, rt) ∩ [ql, qr) ==  $\emptyset$  return;
    if [lf, rt) ⊂ [ql, qr)
    {
        t[v].d = t[v].d  $\odot$  add;
        return;
    }
    push(v);
    // Cập nhật các cây con
    tm = (lf+rt)/2;
    update(2*v, lf, tm, ql, qr, add);
    update(2*v+1, tm, rt, ql, qr, add);
    // Tính lại giá trị ở nút gốc
    t[v].val = (t[2*v].val  $\odot$  t[2*v].d)  $\oplus$ 
                (t[2*v+1].val  $\odot$  t[2*v+1].d);
}

```

Dẫn xuất kết quả

Kiểu dữ liệu
của cây

```
T query(int v,int lf,int rt, int ql,int qr)
{
    if [lf, rt) ∩ [ql, qr) == Ø return 1;
    if [lf, rt) ⊂ [ql, qr) return t[v].val ⊕ t[v].d;
    push(v);
    tm=(lf+rt)/2;
    T ans = query(v*2,1,tm,ql,qr) ⊕
            query(v*2+1,tm,rt,ql,qr));
    t[v].val = (t[2*v].val ⊕ t[2*v].d) ⊕
                (t[2*v+1].val ⊕ t[2*v+1].d);
    return ans;
}
```

3. Cây không đồng bộ quản lý tổng và cập nhật tăng giá trị trên đoạn

Xét cây quản lý tổng các đoạn của dãy số nguyên $\mathbf{A} = (\mathbf{a}_0, \mathbf{a}_1, \dots, \mathbf{a}_{n-1})$ và các phép cập nhật tăng các phần tử có chỉ số $\in [\mathbf{ql}, \mathbf{qr})$ một giá trị \mathbf{x} .

Mỗi phần tử của cây \mathbf{t} là cặp giá trị



Hàm khởi tạo cây

```
void build(int v, int l, int r)
{
    if (l == r) t[v].ff = a[l];
    else
    {
        int mid = (l + r) / 2;
        build(v * 2, l, mid);
        build(v * 2 + 1, mid + 1, r);
        t[v].ff = t[v * 2].ff + t[v * 2 + 1].ff;
    }
}
```

Lời gọi

build(1,0,n-1);

Tăng giá trị trong đoạn

– Hàm update

Các tham số:

- + **v** – đỉnh gốc cây/cây con,
- + **lf, rt** – xác định $[lf, rt)$ đoạn các nút cần xét trong cây con,
- + **ql, qr** – xác định đoạn phần tử $[ql, qr)$ của dãy **A** cần tăng giá trị,
- + **x** – giá trị tăng cho mỗi phần tử trong đoạn $[ql, qr)$.

Chức năng: Tăng **x** với các phần tử thuộc $[lf, rt) \cap [ql, qr)$.

Giải thuật:

Tăng trường giá trị ở nút gốc: **t[v].first+=x*(qr-ql)**,

Nếu **lf==ql** và **rt==qr** \rightarrow **t[v].second+=x; return;**

Nếu **qr < lf** hoặc **ql > rt** \rightarrow **return;**

Tính **mid=(lf+rt)/2;**

Nếu **qr ≤ mid** \rightarrow **update(2*v, l, mid, ql, qr, x);**

Nếu **ql ≥ mid** \rightarrow **update(2*v+1, mid, rt, ql, qr, x);**

Trường hợp còn lại: Chia $[lf, rt)$ thành 2 đoạn $[lf, mid)$ và $[mid, rt)$, lần lượt thực hiện cập nhật trên các đoạn đó:

```
update(2*v, l, mid, ql, qr, x);
update(2*v+1, mid, rt, ql, qr, x);
```

```
void update(int v, int l, int r, int ql, int qr, int x)
{
    t[v].ff+=x*(qr-ql);

    if(l==ql && r==qr) {t[v].ss+=x; return;}
    if(qr<l || r<ql) return;
    int mid=(l+r)/2;

    if(qr<=mid) update(2*v, l, mid, ql, qr, x);
    else if(ql>=mid) update(2*v+1, mid, r, ql, qr, x);
    else
    {
        update(2*v, l, mid, ql, mid, x);
        update(2*v+1, mid, r, mid, qr, x);
    }
}
```

*Nhận xét: Với việc tích lũy tổng thay đổi trong đoạn ở ngay đầu hàm cho phép vòng tránh phép truyền hiệu ứng cập nhật **push(v)**.*

Lời gọi:

```
update(1, 0, n, ql, qr, x);
```

Tính tổng trên đoạn $[ql, qr)$ – Hàm **count**

Các tham số:

- + **v** – đỉnh gốc cây/cây con,
- + **lf, rt** – xác định $[lf, rt)$ đoạn các nút cần xét trong cây con,
- + **ql, qr** – xác định đoạn phần tử $[ql, qr)$ của dãy **A** cần tính tổng.

Giải thuật:

Nếu **lf==ql** và **rt==qr** → Trả về $t[v].first$,

Xác định **mid=(lf+rt)/2**,

Nếu **qr ≤ mid** → Tính **count(2*v, l, mid, ql, qr) + t[v].ss * (qr-ql);**

Nếu **ql ≥ mid** → Tính **count(2*v+1, mid, r, ql, qr) + t[v].ss * (qr-ql);**

Các trường hợp còn lại → tính

```
count(2*v, l, mid, ql, mid) +
    count(2*v+1, mid, r, ql, qr) + t[v].ss * (qr-ql);
```

```
int count(int v, int l, int r, int ql, int qr)
{
    if(l==ql && r==qr) return t[v].ff;
    int mid=(l+r)/2;
    if(qr<=mid) return count(2*v, l, mid, ql, qr)+t[v].ss*(qr-ql);
    else if(ql>=mid) return count(2*v+1, mid, r, ql, qr)+t[v].ss*(qr-ql);
    else
        return(
            count(2*v, l, mid, ql, mid) +
            count(2*v+1, mid, r, mid, qr) + t[v].ss*(qr-ql));
}
```

Nhận xét: *Việc tích lũy trực tiếp tổng số giá cho phép vòng tránh **push(v)**.*

Lời gọi:

```
res=count(1,0,n,ql,qr)
```

Tìm phần tử của **A** ở vị trí **pos**

Nhận xét: *Duyệt từ lá về gốc → không cần sử dụng hàm push(v).*

```
int getpos(int pos)
{
    pos+=p; // Xác định vị trí nút lá
    int r=t[pos].ff; // Giá trị ban đầu của nút
    pos>>=1; // Tới nút cha
    while(pos>0)
    {
        r+=t[pos].ss; // Tích lũy số giá
        pos>>=1; // Tới nút cha
    }
    return r;
}
```

Chương trình minh họa

```
//Create entier segment with delta = val

#include <bits/stdc++.h>
#define ff first
#define ss second
using namespace std;
typedef pair<int,int> pii;
ifstream fi ("Segm_New.inp");
ofstream fo ("Segm_New.out");
int n,p,x;
vector<pii> t;
vector<int> a;

void wtr()
{
    for(int i=1; i<p+n; ++i)
    {
        if((i&(-i))==i) fo<<endl;
        fo<<(' '<<t[i].ff<<', '<<t[i].ss<<") ";
    }
    fo<<endl;
}

void build(int v, int l, int r)
{
    if (l == r) t[v].ff = a[l];
    else
    {
        int mid = (l + r) / 2;
        build(v * 2, l, mid);
        build(v * 2 + 1, mid + 1, r);
        t[v].ff = t[v * 2].ff + t[v * 2 + 1].ff;
    }
}

void push(int v)
{
    t[2*v].ss+=t[v].ss;
    t[2*v+1].ss+=t[v].ss;
    t[v].ss=0;
}
int count(int v, int l, int r, int ql, int qr)
{
    if(l==ql && r==qr) return t[v].ff;
    int mid=(l+r)/2;
    if(qr<=mid) return count(2*v,l,mid,ql,qr)+t[v].ss*(qr-ql);
    else if(ql>=mid) return count(2*v+1,mid,r,ql,qr)+t[v].ss*(qr-ql);
    else
        return (
            count(2*v,l,mid,ql,mid)+
            count(2*v+1,mid,r,mid,qr)+ t[v].ss*(qr-ql));
}

void update(int v, int l, int r, int ql, int qr,int x)
{
    t[v].ff+=x*(qr-ql);
    if(l==ql && r==qr) {t[v].ss+=x; return;}
}
```

```

if(qr<=l || r<ql) return;
int mid=(l+r)/2;

if(qr<=mid) update(2*v,l,mid,ql,qr,x);
else if(ql>=mid) update(2*v+1,mid,r,ql,qr,x);
else
{
    update(2*v,l,mid,ql,mid,x);
    update(2*v+1,mid,r,mid,qr,x);
}
}

int getpos(int pos)
{
    pos+=p;           // Xác định vị trí nút lá
    int r=t[pos].ff; // Giá trị ban đầu của nút
    pos>>=1;
    while(pos>0)
    {
        r+=t[pos].ss; // Tích lũy số gia
        pos>>=1;       // Tới nút cha
    }
    return r;
}

int main()
{
    fi>>n;
//n=8;
for(int i=20; i>=0; --i) if((n>i)&1){p=1<<i; break;}
if(n&(-n)!=n)p<<=1;

t.assign(4*n,{0,0});
a.resize(n);
//for(int i=0; i<n; ++i) t[i+p].ff=i+1;
//for(int i=p-1; i>0; --i) t[i].ff=t[2*i].ff+t[2*i+1].ff;
for(int i=0; i<n; ++i) fi>>a[i];
build(1,0,n-1);
wtr();
fo<<".....\n";

update(1,0,n,1,4,10);
x=count(1,0,n,1,5);
fo<<"C 1..5: "<<x<<endl;
x=count(1,0,n,3,6);
fo<<"C 3..6: "<<x<<endl;

update(1,0,n,3,6,10);
update(1,0,n,0,8,1);
for(int i=0; i<n; ++i)
    fo<<"e "<<i<<": "<<getpos(i)<<endl;
x=count(1,0,n,2,6);
fo<<"C 2..6: "<<x<<endl;
fo<<"-----\n";
}

/*
Input
8
1 2 3 4 5 6 7 8

Output

```

```

(36,0)
(10,0) (26,0)
(3,0) (7,0) (11,0) (15,0)
(1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
-----
C 1..5: 44
C 3..6: 25
e 0: 2
e 1: 13
e 2: 14
e 3: 25
e 4: 16
e 5: 17
e 6: 8
e 7: 9
C 2..6: 72
-----
*/

```

4. Cây quản lý Max có cập nhật đoạn

Cập nhật tăng giá trị trên đoạn

Các tham số của hàm **update**: như ở mục trên.

Giải thuật:

Nếu **lf==ql** và **rt==qr** \rightarrow **t[v].ff+=x, t[v].ss+=x.**

mid = (lf+rt)/2

Nếu **qr ≤ mid** \rightarrow **update(2*v, lf, mid, ql, qr, x);**

Nếu **ql ≥ mid** \rightarrow **update(2*v+1, mid, rt, ql, qr, x);**

Trong trường hợp còn lại: Thực hiện việc cập nhật trên 2 đoạn **[lf, mid)** và **[mid, rt)**:

```

update(2*v, lf, mid, ql, mid, x);
update(2*v+1, mid, rt, mid, qr, x);

```

Trong **mọi trường hợp**: Trước khi trở về cần cập nhật thông tin ở đỉnh gốc cây/cây con: **t[v].ff = max(t[2*v].ff, t[2*v+1].ff)+t[v].ss;**

Lời gợi:

```

update(1,0,n,ql,qr,x);

```

```

void update(int v, int l, int r, int ql, int qr, int x)
{
    if(l==ql && r==qr) {t[v].ff+=x; t[v].ss+=x; return;}
    int mid=(l+r)/2;
    if(qr<=mid)
    {
        update(2*v,l,mid,ql,qr,x);
        t[v].ff=max(t[2*v].ff,t[2*v+1].ff)+t[v].ss;
    }
    else if(ql>=mid)
    {
        update(2*v+1,mid,r,ql,qr,x);
        t[v].ff=max(t[2*v].ff,t[2*v+1].ff)+t[v].ss;
    }
    else
    {
        update(2*v,l,mid,ql,mid,x);
        update(2*v+1,mid,r,mid,qr,x);
        t[v].ff=max(t[2*v].ff,t[2*v+1].ff)+t[v].ss;
    }
}

```

Tìm max trên đoạn

Hàm **count** với các tham số tương tự như ở phần trước.

Giải thuật:

Nếu **lf==ql** và **rt==qr** → Trả về **t[v].ff**.

mid = (lf+rt)/2

Nếu **qr ≤ mid** → **count(2*v,lf,mid,ql,qr)+t[v].ss;**

Nếu **ql ≥ mid** → **count(2*v+1,mid,rt,ql,qr) +t[v].ss;**

Trong trường hợp còn lại: trả về giá trị

max(count(2*v,l,mid,ql,mid),
count(2*v+1,mid,r,mid,qr))+ t[v].ss;

Lời gọi:

count(1,0,n,ql,qr)

```

int count(int v, int l, int r, int ql, int qr)
{
    if(l==ql && r==qr) return t[v].ff;
    int mid=(l+r)/2;
    if(qr<=mid) return count(2*v,l,mid,ql,qr)+t[v].ss;
    else if(ql>=mid)
        return count(2*v+1,mid,r,ql,qr)+t[v].ss;
    else
        return max(count(2*v,l,mid,ql,mid),
                  count(2*v+1,mid,r,mid,qr))+ t[v].ss;
}

```

Chương trình minh họa

```

// Get Max and Increase entier segment with delta = val

#include <bits/stdc++.h>
#define ff first
#define ss second
using namespace std;
typedef pair<int,int> pii;
ifstream fi ("intv_tree.inp");
ofstream fo ("Segm_New.out");
const int INF=-2e9;
int n,p,x;
vector<pii> t;

void wtr()
{
    for(int i=1; i<p+n; ++i)
    {
        if((i&(-i))==i) fo<<endl;
        fo<<'('<<t[i].ff<<', '<<t[i].ss<<") ";
    }
    fo<<endl;
}

void update(int v, int l, int r, int ql, int qr, int x )
{
    if(l==ql && r==qr) {t[v].ff+=x; t[v].ss+=x; return;}
    int mid=(l+r)/2;
    if(qr<=mid) {update(2*v,l,mid,ql,qr,x);
    t[v].ff=max(t[2*v].ff,t[2*v+1].ff)+t[v].ss;}
    else if(ql>=mid)
    {update(2*v+1,mid,r,ql,qr,x);t[v].ff=max(t[2*v].ff,t[2*v+1].ff)+t[v].ss;}
    else
    {
        update(2*v,l,mid,ql,mid,x);
        update(2*v+1,mid,r,mid,qr,x);
        t[v].ff=max(t[2*v].ff,t[2*v+1].ff)+t[v].ss;
    }
}

int count(int v, int l, int r, int ql, int qr)
{

```

```

if(l==ql && r==qr) return t[v].ff;
int mid=(l+r)/2;
if(qr<=mid) return count(2*v,l,mid,ql,qr)+t[v].ss;
else if(ql>=mid) return count(2*v+1,mid,r,ql,qr)+t[v].ss;
else
    return
        max(count(2*v,l,mid,ql,mid),
            count(2*v+1,mid,r,mid,qr))+ t[v].ss;
}

int main()
{
    fi>>n;
    for(int i=20; i>=0; --i) if((n>>i)&1){p=1<<i; break;}
    if(n&(-n)!=n)p<<=1;

    t.assign(4*n,{INF,0});
    for(int i=0; i<n; ++i) fi>>t[i+p].ff;
    for(int i=p-1; i>0; --i)t[i].ff=max(t[2*i].ff,t[2*i+1].ff);
    wtr();
    fo<<"M 4 . . 8: "<<count(1,0,n,4,8)<<endl;
    update(1,0,n,3,5,12);
    fo<<"U 3 . . 5 12\n";
    fo<<"M 4 . . 8: "<<count(1,0,n,4,8)<<endl;
    update(1,0,n,1,4,-10);
    fo<<"U 1 . . 4 -10\n";
    fo<<"M 0 . . 5: "<<count(1,0,n,0,5)<<endl;
}

/*
Input
8
4 -3 6 5 -2 8 1 2

Output
(8,0)
(6,0) (8,0)
(4,0) (6,0) (8,0) (2,0)
(4,0) (-3,0) (6,0) (5,0) (-2,0) (8,0) (1,0) (2,0)
M 4 . . 8: 8
U 3 . . 5 12
M 4 . . 8: 10
U 1 . . 4 -10
M 0 . . 5: 10
*/

```

5. Xác định Max trên đoạn con

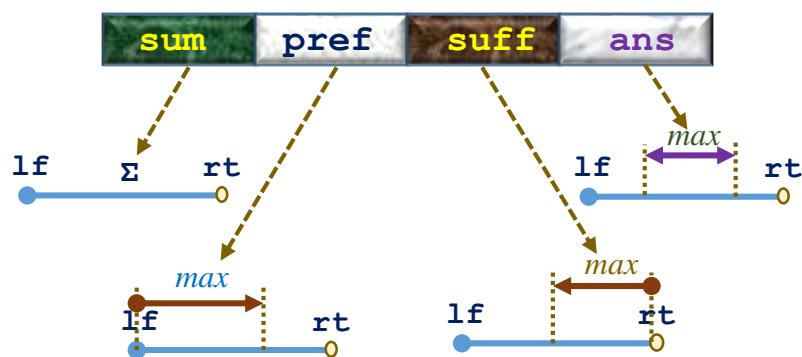
Xét ứng dụng cây phân khúc cho phép thực hiện các truy vấn:

- Thay đổi giá trị một phần tử của dãy **A**: thay a_i bằng giá trị **x**,
- Tìm đoạn con trong đoạn $[lf, rt]$ có tổng các phần tử là lớn nhất. Nếu các phần tử trong đoạn nói trên đều âm thì kết quả là đoạn con rỗng với tổng là 0.

Mỗi phép truy vấn được xử lý với độ phức tạp $O(\log n)$.

Mỗi nút của cây cần lưu trữ 4 thông tin:

- Tổng các phần tử trên đoạn,
- Tổng lớn nhất của các tiền tố của đoạn,
- Tổng lớn nhất của các hậu tố của đoạn,
- Tổng lớn nhất trong số các đoạn con thuộc đoạn.



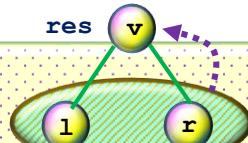
Khai báo dữ liệu:

```
struct data
{
    int sum, pref, suff, ans;
};

vector<data> t;
vector<int> a;
```

Tạo thông tin cho nút lá:

```
data combine (data l, data r)
{
    data res;
    res.sum = l.sum + r.sum;
    res.pref = max (l.pref, l.sum + r.pref);
    res.suff = max (r.suff, r.sum + l.suff);
    res.ans = max (max (l.ans, r.ans), l.suff + r.pref);
    return res;
}
```

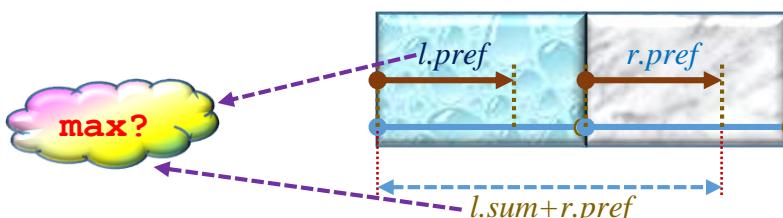


Tổng hợp thông tin lên nút cha:

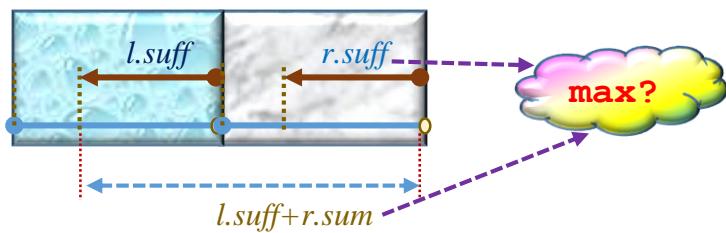
```
data make_data (int val)
{
    data res;
    res.sum = val;
    res.pref = res.suff = res.ans = max (0, val);
    return res;
}
```

Trường **sum**: Hiển nhiên,

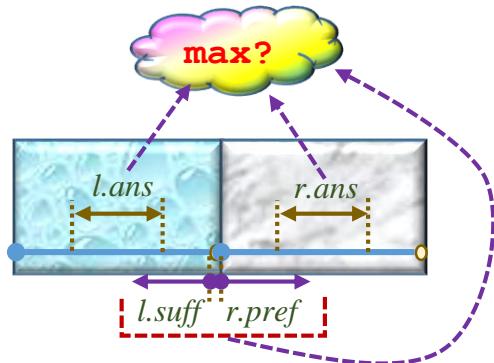
Trường **pref**:



Trường suff:



Trường ans:



Hàm tạo cây

```
void build (vector<int> a, int v, int tl, int tr)
{
    if (tl == tr) t[v] = make_data (a[tl]);
    else
    {
        int tm = (tl + tr) / 2;
        build (a, v*2, tl, tm);
        build (a, v*2+1, tm+1, tr);
        t[v] = combine (t[v*2], t[v*2+1]);
    }
}
```

Tạo nút lá

Tạo nút cha

Lời gọi:

```
build(a, 1, 0, n-1);
```

Thay đổi giá trị một phần tử của **A**: phần tử a_{pos} nhận giá trị **new_val**:

Việc cập nhật được thực hiện như sơ đồ tạo cây, nhưng việc tính lại giá trị các nút chỉ thực hiện theo một nhánh: từ gốc tới nút lá tương ứng.

```
void update (int v, int tl, int tr, int pos, int new_val)
{
    if (tl == tr) t[v] = make_data (new_val);
    else
    {
        int tm = (tl + tr) / 2;
        if (pos < tm) update (v*2, tl, tm, pos, new_val);
        else update (v*2+1, tm+1, tr, pos, new_val);

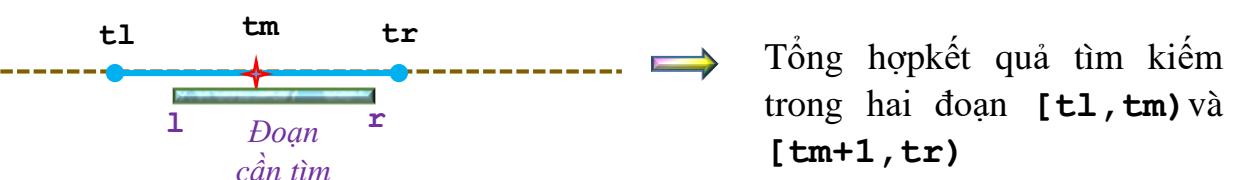
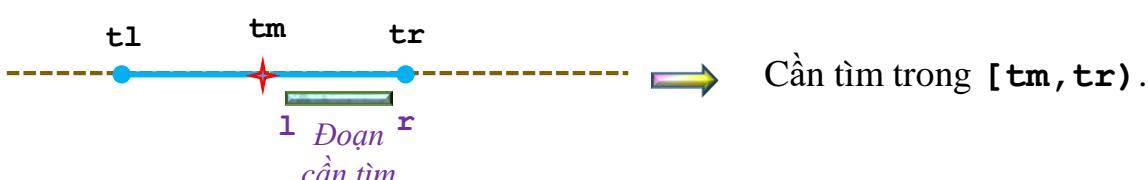
        t[v] = combine (t[v*2], t[v*2+1]);
    }
}
```

Lời gọi:

update(1, 0, n, pos, x);

Truy vấn: Tìm đoạn con có tổng lớn nhất trong đoạn $[l, r]$

Có 4 trường hợp: Đánh **v** quản lý các nút trong đoạn $[tl, tr]$



```

data query (int v, int tl, int tr, int l, int r)
{
    if (l == tl && tr == r) return t[v];
    int tm = (tl + tr) / 2;
    if (r <= tm) return query (v*2, tl, tm, l, r);
    if (l > tm) return query (v*2+1, tm+1, tr, l, r);
    return combine (
        query (v*2, tl, tm, l, tm),
        query (v*2+1, tm+1, tr, tm+1, r)
    );
}

```

Lời gọi:

query (1,0,n,l,r);

Chương trình minh họa:

```

// Determine Max in subsegment with change one element
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("intv_tree.inp");
ofstream fo ("intv_tree.out");
int n,x;

struct data
{
    int sum, pref, suff, ans;
};
vector<data> t;
vector<int> a;

data combine (data l, data r)
{
    data res;
    res.sum = l.sum + r.sum;
    res.pref = max (l.pref, l.sum + r.pref);
    res.suff = max (r.suff, r.sum + l.suff);
    res.ans = max (max (l.ans, r.ans), l.suff + r.pref);
    return res;
}

data make_data (int val)
{
    data res;
    res.sum = val;
    res.pref = res.suff = res.ans = max (0, val);
    return res;
}

void build (vector<int> a, int v, int tl, int tr)
{
    if (tl == tr) t[v] = make_data (a[tl]);
    else

```

```

    {
        int tm = (tl + tr) / 2;
        build (a, v*2, tl, tm);
        build (a, v*2+1, tm+1, tr);
        t[v] = combine (t[v*2], t[v*2+1]);
    }
}

void update (int v, int tl, int tr, int pos, int new_val)
{
    if (tl == tr) t[v] = make_data (new_val);
    else
    {
        int tm = (tl + tr) / 2;
        if (pos < tm) update (v*2, tl, tm, pos, new_val);
        else update (v*2+1, tm+1, tr, pos, new_val);

        t[v] = combine (t[v*2], t[v*2+1]);
    }
}

data query (int v, int tl, int tr, int l, int r)
{
    if (l == tl && tr == r) return t[v];
    int tm = (tl + tr) / 2;
    if (r <= tm) return query (v*2, tl, tm, l, r);
    if (l > tm) return query (v*2+1, tm+1, tr, l, r);
    return combine (
        query (v*2, tl, tm, l, tm),
        query (v*2+1, tm+1, tr, tm+1, r)
    );
}

int main()
{
    fi>>n;
    a.resize(n); t.resize(4*n);
    for(int i=0; i<n; ++i) fi>>a[i];

    build(a,1,0,n-1);
    data z;
    z=query(1,0,n,0,n-1);
    fo<<"0..n-1: "<<z.ans<<endl;

    z=query(1,0,n,0,7);
    fo<<"0..7: "<<z.ans<<endl;

    update(1,0,n,6,2); fo<<"U 6 . . 2\n";
    z=query(1,0,n,0,5);
    fo<<"0..5: "<<z.ans<<endl;
    z=query(1,0,n,3,7);
    fo<<"3..7: "<<z.ans<<endl;
    z=query(1,0,n,3,n);
    fo<<"3..n: "<<z.ans<<endl;
}
/*
Input
10
3 -2 -2 5 -2 6 -1 3 -5 3

```

```

Output
0..n-1: 11
0..7: 9
U 6 .. 2
0..5: 5
3..7: 11
3..n: 14
*/

```

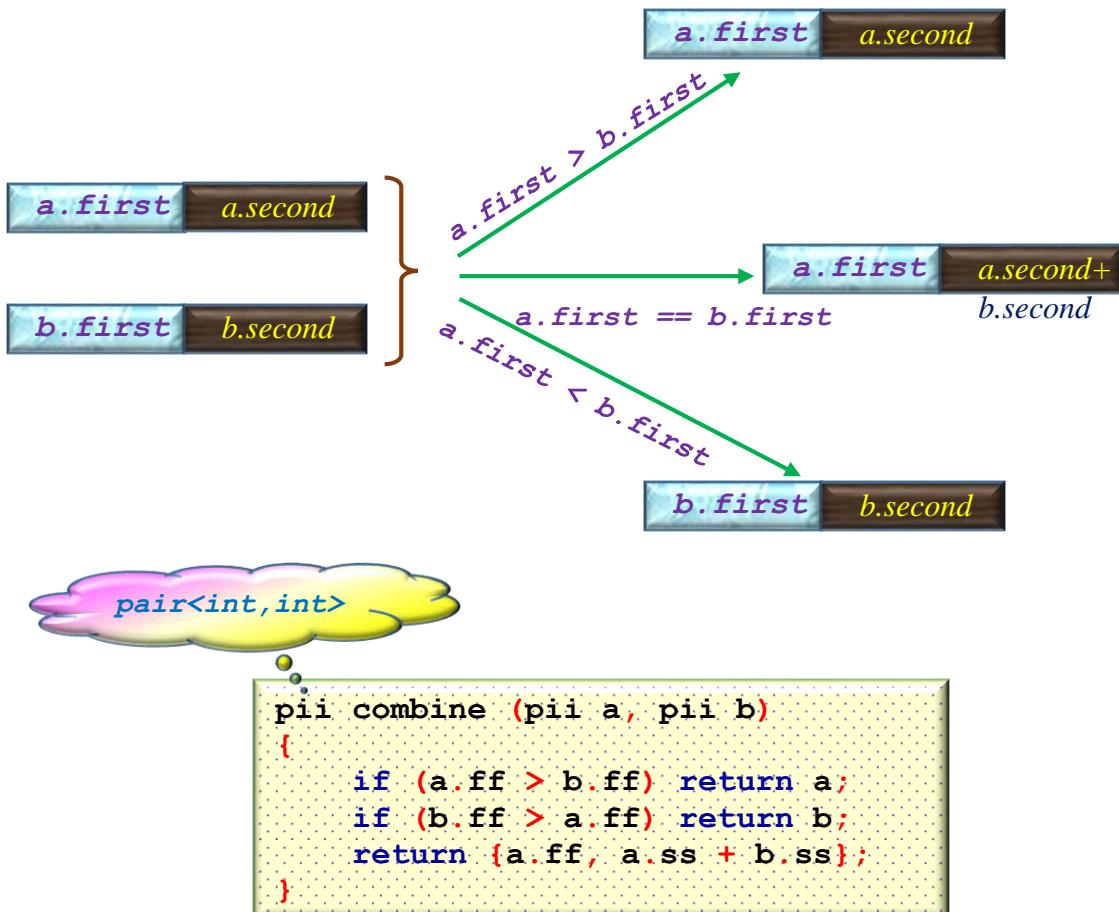
6. Xác định Min/Max và số lần xuất hiện trong đoạn

Yêu cầu tìm *min* (hoặc *max*) trong đoạn $[l,r]$ và số lần xuất hiện của *min* (hoặc *max*) trong đoạn đó. Trong quá trình xử lý có thể xuất hiện truy vấn thay đổi giá trị của một biến a_i nào đó của dãy **A**.

Không mất tính chất tổng quát, ta sẽ xét yêu cầu tìm *max*.

Mỗi nút của cây cần lưu trữ 2 giá trị (*Max*, Số lần gặp).

Thông tin ở nút cha được tổng hợp từ thông tin ở hai nút con theo quy tắc:



Xây dựng cây:

Khởi tạo nút lá với giá trị a_i :
 $(a_i, 1)$,

Các nút trong và nút gốc của cây: dựa vào hàm
combine.

```
void build (vector<int> a, int v, int tl, int tr)
{
    if (tl == tr) t[v] = {a[tl], 1};
    else
    {
        int tm = (tl + tr) / 2;
        build (a, v*2, tl, tm);
        build (a, v*2+1, tm+1, tr);
        t[v] = combine (t[v*2], t[v*2+1]);
    }
}
```

Thay đổi giá trị một phần tử của dãy số:

Việc cập nhật được thực hiện như sơ đồ tạo cây, nhưng việc tính lại giá trị các nút chỉ thực hiện theo một nhánh: từ gốc tới nút lá tương ứng.

```
void update (int v, int tl, int tr, int pos, int new_val)
{
    if (tl == tr) t[v] = {new_val, 1};
    else
    {
        int tm = (tl + tr) / 2;
        if (pos <= tm) update (v*2, tl, tm, pos, new_val);
        else update (v*2+1, tm+1, tr, pos, new_val);
        t[v] = combine (t[v*2], t[v*2+1]);
    }
}
```

update (1, 0, n-1, pos, x);

Lời gọi:

Xử lý truy vấn: Tìm *max* và số lần xuất hiện trong đoạn $[l, r]$

Xét các trường hợp tương tự như phần tương ứng ở mục trước.

```
pii get_max (int v, int tl, int tr, int l, int r)
{
    if (l > r) return {-INF, 0};
    if (l == tl && r == tr) return t[v];
    int tm = (tl + tr) / 2;
    return combine (
        get_max (v*2, tl, tm, l, min(r, tm)),
        get_max (v*2+1, tm+1, tr, max(l, tm+1), r)
    );
}
```

Lời gọi:

```
get_max(1, 0, n-1, l, r);
```

Chương trình minh họa

```
// Max in segment and how many times

#include <bits/stdc++.h>
using namespace std;
#define ff first
#define ss second
ifstream fi ("intv_tree.inp");
ofstream fo ("intv_tree.out");
typedef pair<int,int> pii;
int n;
const int INF=1e9;
vector<pii> t;
vector<int> a;
pii r;

pii combine (pii a, pii b)
{
    if (a.ff > b.ff) return a;
    if (b.ff > a.ff) return b;
    return {a.ff, a.ss + b.ss};
}

void build (vector<int> a, int v, int tl, int tr)
{
    if (tl == tr) t[v] = {a[tl], 1};
    else
    {
        int tm = (tl + tr) / 2;
        build (a, v*2, tl, tm);
        build (a, v*2+1, tm+1, tr);
        t[v] = combine (t[v*2], t[v*2+1]);
    }
}

pii get_max (int v, int tl, int tr, int l, int r)
{
    if (l > r) return {-INF, 0};
    if (l == tl && r == tr) return t[v];
    int tm = (tl + tr) / 2;
    return combine (
        get_max (v*2, tl, tm, l, min(r,tm)),
        get_max (v*2+1, tm+1, tr, max(l,tm+1), r)
    );
}

void update (int v, int tl, int tr, int pos, int new_val)
{
    if (tl == tr) t[v] = {new_val, 1};
    else
    {
        int tm = (tl + tr) / 2;
        if (pos <= tm) update (v*2, tl, tm, pos, new_val);
        else update (v*2+1, tm+1, tr, pos, new_val);
        t[v] = combine (t[v*2], t[v*2+1]);
    }
}
```

```

}

int main()
{
    fi>>n;
    t.assign(4*n+4,{0,0});
    a.resize(n+1);
    for(int i=0; i<n; ++i) fi>>a[i];
    build(a,1,0,n-1);

    r=get_max(1,0,n-1,0,n-1);
    fo<<"0..n-1: "; fo<<r.ff<<' '<<r.ss<<endl;
    r=get_max(1,0,n-1,1,n-1);
    fo<<"1..n-1: "; fo<<r.ff<<' '<<r.ss<<endl;
    r=get_max(1,0,n-1,1,n-2);
    fo<<"1..n-2: "; fo<<r.ff<<' '<<r.ss<<endl;
    r=get_max(1,0,n-1,1,2);
    fo<<"1..2: "; fo<<r.ff<<' '<<r.ss<<endl;
    r=get_max(1,0,n-1,0,2);
    fo<<"0..2: "; fo<<r.ff<<' '<<r.ss<<endl;

    update(1,0,n-1,2,8);
    fo<<"-----\n";
    r=get_max(1,0,n-1,0,n-1);
    fo<<"0..n-1: "; fo<<r.ff<<' '<<r.ss<<endl;
    r=get_max(1,0,n-1,1,n-1);
    fo<<"1..n-1: "; fo<<r.ff<<' '<<r.ss<<endl;
    r=get_max(1,0,n-1,1,n-2);
    fo<<"1..n-2: "; fo<<r.ff<<' '<<r.ss<<endl;
    r=get_max(1,0,n-1,1,2);
    fo<<"1..2: "; fo<<r.ff<<' '<<r.ss<<endl;
    r=get_max(1,0,n-1,0,2);
    fo<<"0..2: "; fo<<r.ff<<' '<<r.ss<<endl;
}

/*
input
10
8 8 4 3 8 2 6 8 6 8

Output
0..n-1: 8 5
1..n-1: 8 4
1..n-2: 8 3
1..2: 8 1
0..2: 8 2
-----
0..n-1: 8 6
1..n-1: 8 5
1..n-2: 8 4
1..2: 8 2
0..2: 8 3
*/

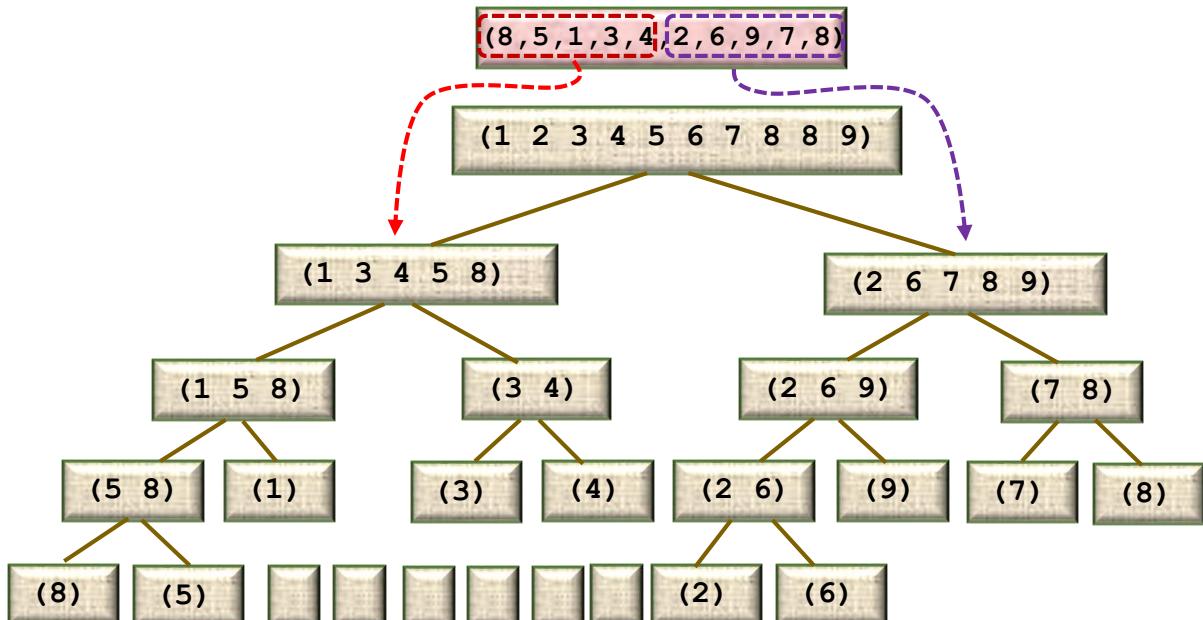
```

7. Xác định số nhỏ nhất lớn hơn hoặc bằng số cho trước trên đoạn con

Xét truy vấn với các tham số (l, r, x) yêu cầu tìm số nhỏ nhất trên đoạn $[l, r]$ lớn hơn hoặc bằng x .

Để có thể xử lý truy vấn với độ phức tạp $O(\log^2 n)$ cần tổ chức cây dưới dạng hai chiều: tại *mỗi nút* lưu trữ *danh sách giá trị* các nút lá được quản lý.

Ví dụ, với $n = 10$ và $\mathbf{A} = (8, 5, 1, 3, 4, 2, 6, 9, 7, 8)$ ta có cây



Khối lượng bộ nhớ cần thiết có bậc $O(n \log n)$.

Tại mỗi nút các giá trị được lưu trữ dưới dạng đã sắp xếp, vì vậy cấu trúc dữ liệu thích hợp cho mỗi nút sẽ là **multiset** hoặc **map**. Tuy vậy, nếu không có yêu cầu cập nhật dữ liệu ta có thể lưu dữ liệu dưới dạng kiểu **vector** vì hệ thống lập trình có nhiều công cụ hiệu quả hỗ trợ tìm kiếm.

a – Trường hợp không có yêu cầu cập nhật

Khai báo và khởi tạo cây:

```
vector<vector<int>> t;
```

```
void build (vector<int> a, int v, int tl, int tr)
{
    if (tl == tr)
        t[v] = vector<int> (1, a[tl]);
    else {
        int tm = (tl + tr) / 2;
        build (a, v*2, tl, tm);
        build (a, v*2+1, tm+1, tr);
        merge (t[v*2].begin(), t[v*2].end(), t[v*2+1].begin(),
               t[v*2+1].end(), back_inserter (t[v]));
    }
}
```

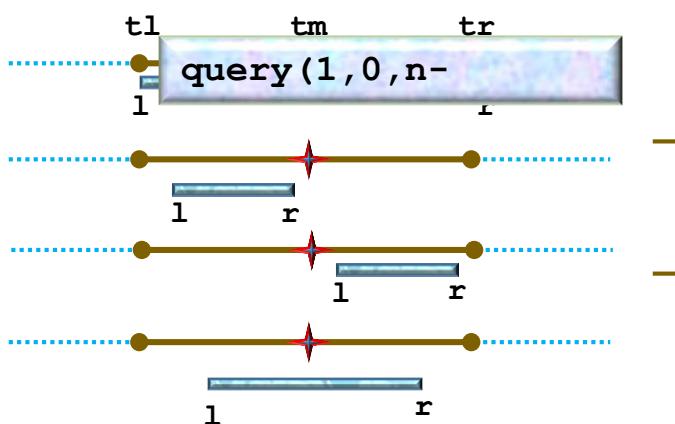
Sắp xếp và hợp nhất
2 vectors

Lời gọi: **build(a, 1, 0, n-1);**

Xử lý truy vấn:

```
int query (int v, int tl, int tr, int l, int r, int x)
{
    if (l > r)
        return INF;
    if (l == tl && tr == r) {
        vector<int>::iterator pos=lower_bound(t[v].begin(), t[v].end(), x);
        if (pos != t[v].end())
            return *pos;
        return INF;
    }
    int tm = (tl + tr) / 2;
    return min (
        query (v*2, tl, tm, l, min(r, tm), x),
        query (v*2+1, tm+1, tr, max(l, tm+1), r, x));
}
```

Phân biệt các trường hợp $[l, r]$ khác rỗng:



Một trong 2 đoạn
sẽ rỗng

Lời gọi:

Chương trình minh họa

```
// Tim so nho nhat lon hon k cho trwoc trong [lf..rt]
// Khong cap nhat day so

#include <bits/stdc++.h>
using namespace std;
ifstream fi ("intv_tree.inp");
ofstream fo ("intv_tree.out");
const int INF=2000000001;
int n,x;
vector<int> a;
vector<vector<int>> t;

void build (vector<int> a, int v, int tl, int tr)
{
    if (tl == tr)
        t[v] = vector<int> (1, a[tl]);
    else {
        int tm = (tl + tr) / 2;
        build (a, v*2, tl, tm);
        build (a, v*2+1, tm+1, tr);
        merge (t[v*2].begin(), t[v*2].end(), t[v*2+1].begin(), t[v*2+1].end(),
               back_inserter (t[v]));
    }
}

int query (int v, int tl, int tr, int l, int r, int x)
{
    if (l > r)
        return INF;
    if (l == tl && tr == r) {
        vector<int>::iterator pos=lower_bound(t[v].begin(), t[v].end(), x);
        if (pos != t[v].end())
            return *pos;
        return INF;
    }
    int tm = (tl + tr) / 2;
    return min (
        query (v*2, tl, tm, l, min(r,tm), x),
        query (v*2+1, tm+1, tr, max(l,tm+1), r, x)
    );
}

int main()
{
    fi>>n;
    a.resize(n);
    t.resize(4*n);
    for(int i=0; i<n; ++i) fi>>a[i];
    build(a,1,0,n-1);
    int
    x = query(1,0,n-1,2,8,5);
    fo<<"2..8/5: "<<x<<endl;
    x = query(1,0,n-1,2,8,4);
    fo<<"2..8/4: "<<x<<endl;
    x = query(1,0,n-1,2,6,5);
    fo<<"2..6/5: "<<x<<endl;
    x = query(1,0,n-1,2,6,7);
    fo<<"2..6/7: "<<x<<endl;
}
```

Input
10
8 5 1 3 4 2 6 9 7 8
Output
2..8/5: 6
2..8/4: 4
2..6/5: 6
2..6/7: 2000000001

B – Trường hợp có cập nhật:

Song song với truy vấn (l, r, x) yêu cầu tìm số nhỏ nhất trên đoạn $[l, r]$ lớn hơn hoặc bằng x còn có thể tồn tại những truy vấn dạng (pos, val) yêu cầu gán cho phần tử a_{pos} giá trị val .

Như đã nói ở trên, giá trị mà mỗi nút quản lý được lưu trữ dưới dạng tập hợp multiset. Việc khai báo cây và khởi tạo có vài thay đổi để tương thích với kiểu dữ liệu.

Khai báo cây:

```
vector<multiset<int>>t;
```

Khởi tạo:

```
void build (vector<int> a, int v, int tl, int tr)
{
    if (tl == tr) t[v].insert(a[tl]);
    else
    {
        int tm = (tl + tr) / 2;
        build (a, v*2, tl, tm);
        build (a, v*2+1, tm+1, tr);
        t[v] = t[2*v];
        for (auto ii : t[2*v+1]) t[v].insert(ii);
    }
}
```

Hợp nhát hai
tập hợp

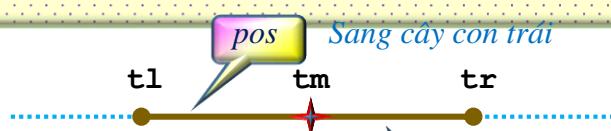
Lời gọi:

```
build(a,1,0,n-1);
```

Cập nhật:

```
void update (int v, int tl, int tr, int pos, int new_val)
{
    t[v].erase (t[v].find (a[pos]));
    t[v].insert (new_val);
    if (tl != tr)
    {
        int tm = (tl + tr) / 2;
        if (pos <= tm) update (v*2, tl, tm, pos, new_val);
        else update (v*2+1, tm+1, tr, pos, new_val);
    }
    else a[pos] = new_val;
}
```

Cập nhật ở gốc
cây/cây con



Lời gọi:

```
update(1,0,n-1,pos,new_val);
```

Truy vấn: Tương tự như trường hợp không có truy vấn cập nhật.

```
int query (int v, int tl, int tr, int l, int r, int x)
{
    if (l > r) return INF;
    if (l == tl && tr == r)
    {
        multiset<int>::iterator pos=lower_bound(t[v].begin(),t[v].end(),x);
        if (pos != t[v].end()) return *pos;
        return INF;
    }
    int tm = (tl + tr) / 2;

    return min (
        query (v*2, tl, tm, l, min(r,tm), x),
        query (v*2+1, tm+1, tr, max(l,tm+1), r, x)
    );
}
```

Lời gọi:

```
query(1,0,n-1,l,r,x);
```

Chương trình minh họa

```
// Tim so nho nhat lon hon k cho trwoc trong [lf..rt]
// Co update day so

#include <bits/stdc++.h>
using namespace std;
ifstream fi ("intv_tree.inp");
ofstream fo ("intv_tree.out");
const int INF=2000000001;
const int MN=100000;
int n,x;
vector<int> a;
vector<multiset<int>>t;

void build (vector<int> a, int v, int tl, int tr)
{
    if (tl == tr)t[v].insert(a[tl]);
    else
    {
        int tm = (tl + tr) / 2;
        build (a, v*2, tl, tm);
        build (a, v*2+1, tm+1, tr);
        t[v]=t[2*v];
        for(auto ii:t[2*v+1])t[v].insert(ii);
    }
}

void update (int v, int tl, int tr, int pos, int new_val)
{
    t[v].erase (t[v].find (a[pos]));
    t[v].insert (new_val);
    if (tl != tr)
    {
        int tm = (tl + tr) / 2;
        if (pos <= tm) update (v*2, tl, tm, pos, new_val);
        else update (v*2+1, tm+1, tr, pos, new_val);
    }
    else a[pos] = new_val;
}

int query (int v, int tl, int tr, int l, int r, int x)
{
    if (l > r) return INF;
    if (l == tl && tr == r)
    {
        multiset<int>::iterator pos=lower_bound(t[v].begin(),t[v].end(),
x);
        if (pos != t[v].end()) return *pos;
        return INF;
    }
    int tm = (tl + tr) / 2;

    return min (
        query (v*2, tl, tm, l, min(r,tm), x),
        query (v*2+1, tm+1, tr, max(l,tm+1), r, x)
    );
}

int main()
```

```

{
    fi>>n;
    a.resize(n);
    t.resize(4*n);
    for(int i=0; i<n; ++i) fi>>a[i];
    build(a,1,0,n-1);
    int
    x = query(1,0,n-1,2,8,5);
    fo<<"2..8/5: "<<x<<endl;
    x = query(1,0,n-1,2,8,4);
    fo<<"2..8/4: "<<x<<endl;
    fo<<"-----\n";
    update(1,0,n-1,6,10); fo<<"U 6 10\n";
        for(int i:a) fo<<i<<' '; fo<<endl;
    x = query(1,0,n-1,2,8,5);
    fo<<"2..8/5: "<<x<<endl;
}
/*
Input
10
8 5 1 3 4 2 6 9 7 8

Output
2..8/5: 6
2..8/4: 4
-----
U 6 10
8 5 1 3 4 2 10 9 7 8
2..8/5: 7
*/

```

8. Gán giá trị cho đoạn

Xét việc biến đổi dãy **A** với phép cập nhật là gán giá trị **x** cho các phần tử trong đoạn $[q_l, q_r]$.

Phép biến đổi này thường áp dụng cho *bài toán tô màu*:

Đường phố trung tâm có **n** nhà mặt tiền. Ban đầu nhà thứ **i** được sơn màu **a_i**, $i = 1 \div n$. Định kỳ người ta phải sơn lại nhà để đảm bảo mặt phố luôn sạch đẹp, mỗi lần các nhà từ **q_l** đến **q_r** (kể cả nhà **q_r**) được sơn lại bằng một màu **x** nào đó.

Kể từ mói xây dựng cho đến nay đã có một số lần thành phố cho sơn lại một số nhà mặt tiền, lần thứ **j** các nhà trong đoạn $[q_{l_j}, q_{r_j}]$ được sơn bằng màu **x_j**.

Khách du lịch được bố trí ở khách sạn là nhà thứ **i** và trên đường đi tới nơi nghỉ thường quan tâm khách sạn có màu gì.

Hãy xác định màu của khách sạn.

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

- Dòng đầu tiên chứa 2 số nguyên **n** và **m** ($1 \leq n, m \leq 10^5$),
- Dòng thứ 2 chứa **n** số nguyên **a₁, a₂, ..., a_n** ($-10^9 \leq a_i \leq 10^9$, $i = 1 \div n$),
- Mỗi dòng trong **m** dòng sau chứa thông tin về một truy vấn, mỗi truy vấn có một trong 2 dạng:
 - 1 **q_l q_r x** – sơn các nhà trong đoạn $[q_l, q_r]$ bằng màu **x** ($1 \leq q_l \leq q_r \leq n, |x| \leq 10^9$),
 - 2 **p** – yêu cầu xác định màu của nhà thứ **p** ($1 \leq p \leq n$).

Kết quả: Đưa ra file văn bản HOUSES.OUT kết quả ứng với truy vấn xác định màu, mỗi kết quả đưa ra trên một dòng.

Ví dụ:

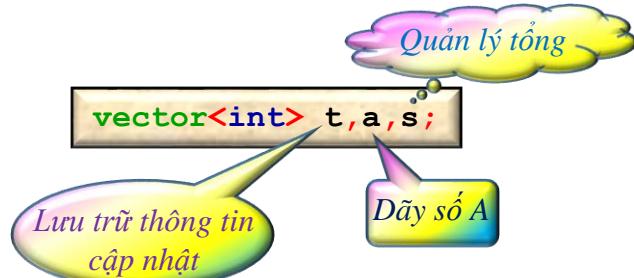
HOUSES . INP	HOUSES . OUT
10 6	2
8 5 1 3 4 2 6 9 7 8	7
2 6	10
2 9	15
1 4 7 10	
2 6	
1 3 5 15	
2 4	

Để giải quyết bài toán trên ta cần tổ chức cây phân khúc **t**. Ngoài việc quản lý giá trị của **A** cây phân khúc còn cho phép thực hiện các truy vấn **tính tổng** (*Range Sum Query* – **RSQ**), truy vấn **tìm min/max** (*Range Maximum Query* – **RMQ**), . . .

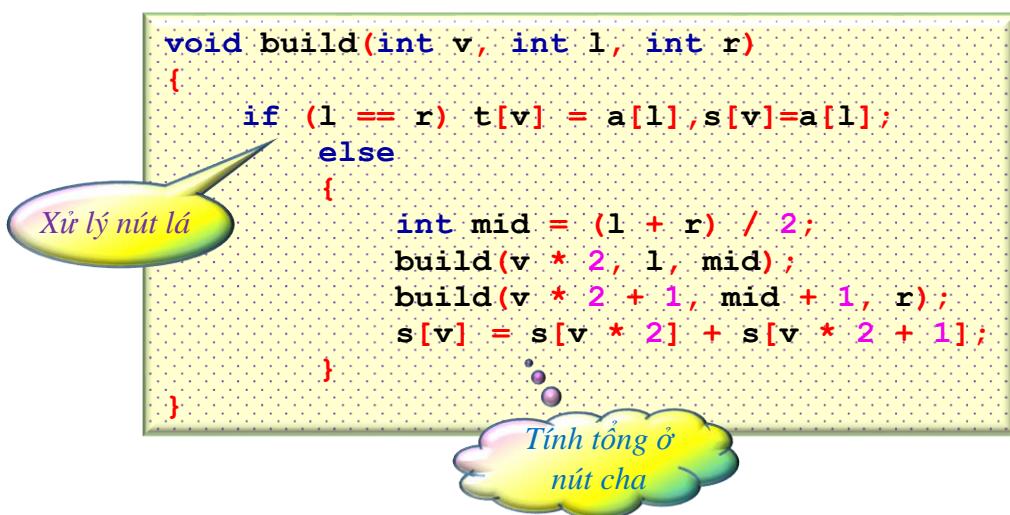
Có thể tổ chức cây mỗi nút có số lượng trường tương ứng với số loại truy vấn (như các cách xử lý đã nêu ở trên) hoặc có thể tổ chức các cây cùng kích thước riêng biệt.

Dưới đây ta sẽ xem xét cách tổ chức các cây riêng biệt cho từng loại truy vấn.

Khai báo dữ liệu:



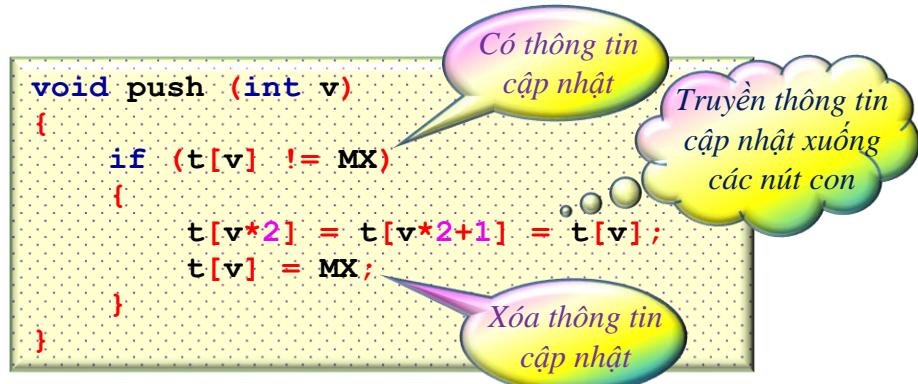
Khởi tạo cây:



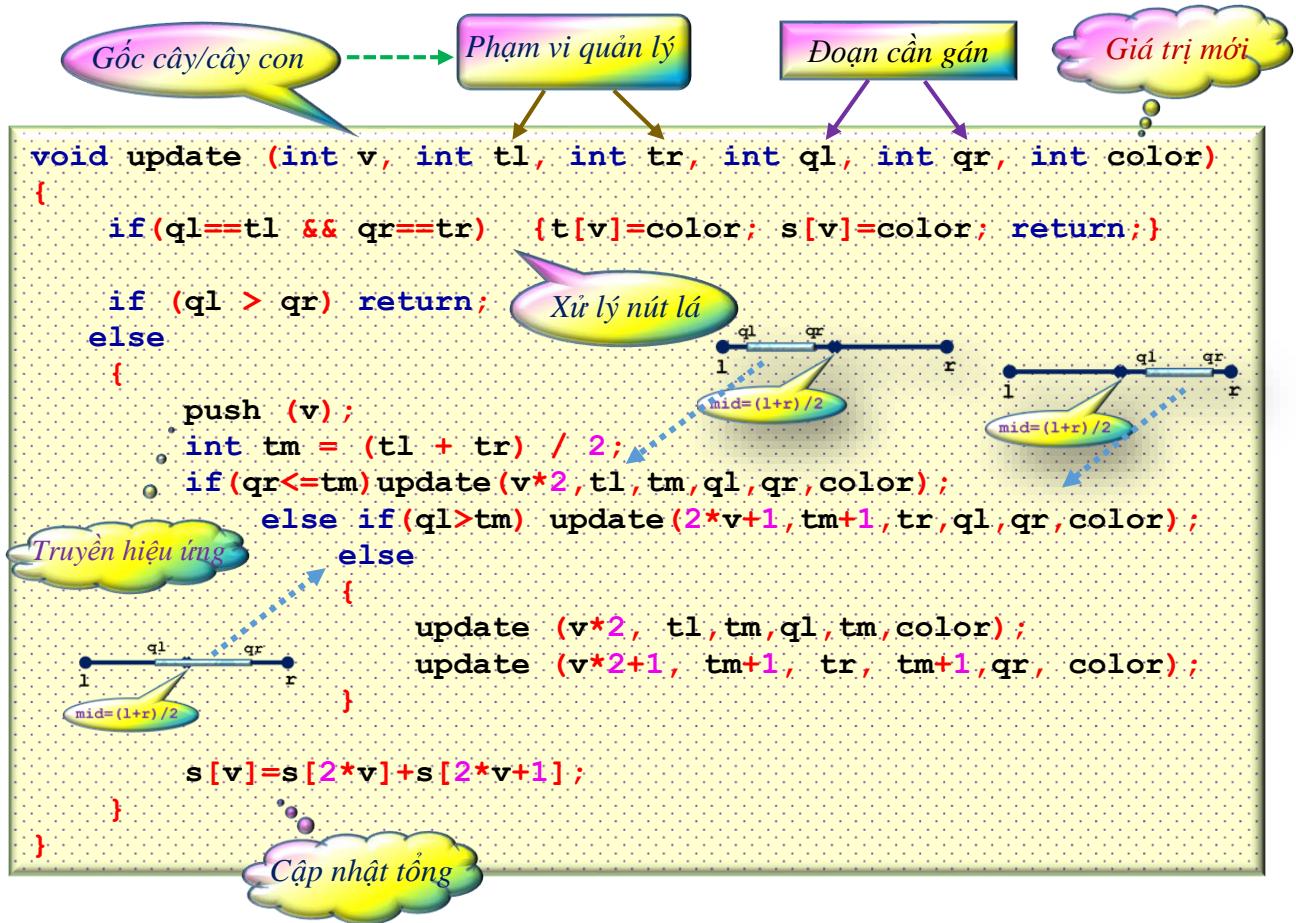
Lời gọi:

```
build(1, 0, n-1);
```

Truyền hiệu ứng cập nhật xuống các nhánh:



Cập nhật:



Tính tổng các phần tử trong đoạn $[ql, qr]$:

```
int get_s(int v, int tl, int tr, int ql, int qr)
{
    if(t[v] != MX) return t[v] * (qr - ql + 1); // Nâng gọn trong đoạn
    if(tl == tr) return s[v]; // gán giá trị
    int mid = (tl + tr) / 2;
    if(qr <= mid) return get_s(v * 2, tl, mid, ql, qr);
    if(ql > mid) return get_s(v * 2 + 1, mid + 1, tr, ql, qr);

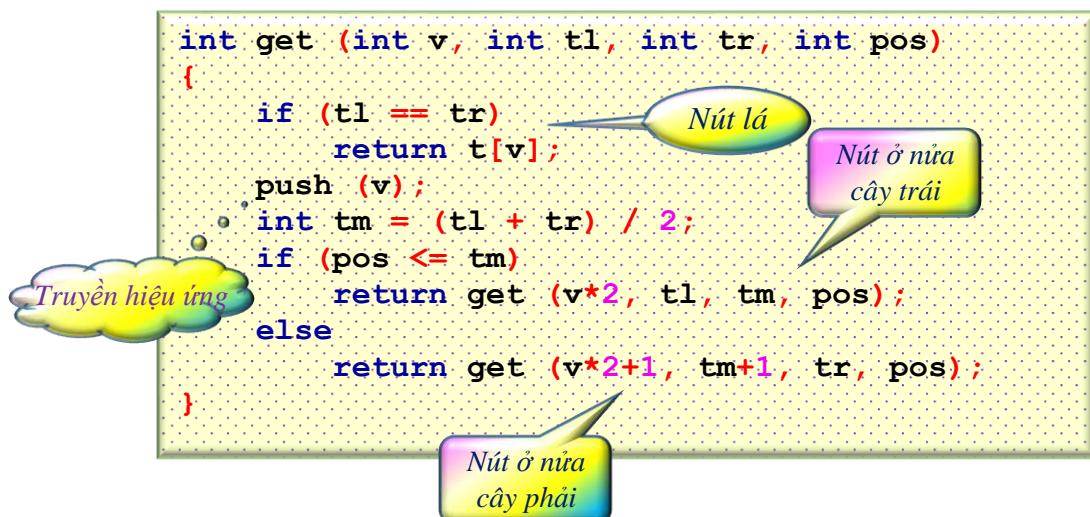
    return get_s(v * 2, tl, mid, ql, mid) +
           get_s(v * 2 + 1, mid + 1, tr, mid + 1, qr);
}
```

Lời gọi:

```
x = get_s(1, 0, n-1, ql, qr);
```

Sơ đồ xử lý: Tương tự trường hợp cập nhật.

Tìm phần tử ở vị trí pos:



Lưu ý:

Trong chương trình các phần tử được *dánh số bắt đầu từ 0*,

Có thể *sử dụng hàm tính tổng* để dẫn xuất giá trị,

Xét hàm dẫn xuất giá trị *chỉ dựa vào bảng thông tin cập nhật*.

Lời gợi:

```
x=get(1,0,n-1,pos);
```

Chương trình minh họa giải thuật (*Không phải lời giải bài toán đã nêu!*)

```
//Gan gia tri cho doan - To mau
// t - quan ly phan tu, s - quan ly sum
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("intv_tree.inp");
ofstream fo ("intv_tree.out");
const int MX=2000000001;
int n,p;
vector<int> t,a,s;

void wtr()
{
    for(int i=1; i<p+n; ++i)
    {
        if((i&(-i))==i) fo<<endl;
        fo<<t[i]<<' ';
    }
    fo<<endl;
}

void build(int v, int l, int r)
{
    if (l == r) t[v] = a[l], s[v]=a[l];
    else
    {
        int mid = (l + r) / 2;
        build(v * 2, l, mid);
        build(v * 2 + 1, mid + 1, r);
        s[v] = s[v * 2] + s[v * 2 + 1];
    }
}

void push (int v)
{
    if (t[v] != MX)
    {
        t[v*2] = t[v*2+1] = t[v];
        t[v] = MX;
    }
}

void update (int v, int tl, int tr, int ql, int qr, int color)
{
    if(ql==tl && qr==tr) {t[v]=color; s[v]=color; return;}

    if (ql > qr) return;
    else
    {
        push (v);
        int tm = (tl + tr) / 2;
        if(qr<=tm) update(v*2,tl,tm,ql,qr,color);
        else if(ql>tm) update(2*v+1,tm+1,tr,ql,qr,color);
        else
        {
            update (v*2, tl,tm,ql,tm,color);
            update (v*2+1, tm+1, tr, tm+1,qr, color);
        }
    }

    s[v]=s[2*v]+s[2*v+1];
}
```

```

        }

}

int get_s(int v, int tl,int tr, int ql, int qr)
{
    if(t[v]!=MX) return t[v]*(qr-ql+1);
    if(tl==tr) return s[v];
    int mid = (tl + tr) / 2;
    if(qr<=mid) return get_s(v * 2, tl, mid, ql, qr);
    if(ql>mid) return get_s(v * 2 + 1, mid+1 , tr, ql, qr);

    return get_s(v*2, tl, mid, ql, mid) +
           get_s(v*2+1, mid+1 , tr, mid+1, qr);
}

int get (int v, int tl, int tr, int pos)
{
    if (tl == tr)
        return t[v];
    push (v);
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return get (v*2, tl, tm, pos);
    else
        return get (v*2+1, tm+1, tr, pos);
}

int main()
{
    fi>>n;
    a.resize(n);
    t.resize(4*n,MX); s.resize(4*n);
    for(int i=20; i>=0; --i) if((n>>i)&1){p=1<<i; break;}
    if((n&(-n))!=n)p<<=1;

    for(int i=0; i<n; ++i) fi>>a[i];
    build(1,0,n-1);
    for(int i=0; i<n; ++i) fo<<"e"<<i<<": "<< get(1,0,n-1,i)<<endl;
    fo<<" ..... \n";

    int x;
    x=get(1,0,n-1,5); fo<<"v 5: "<<x<<endl;
    x=get(1,0,n-1,7); fo<<"v 7: "<<x<<endl;

    update(1,0,n-1,3,6,10); fo<<"\nU 3 . . 6 10\n";
    for(int i=0; i<n; ++i) fo<<"e"<<i<<": "<< get(1,0,n-1,i)<<endl;
    x=get_s(1,0,n-1,0,4); fo<<"S 0 . . 4: "<<x<<endl;

    fo<<" ----- \n";
    update(1,0,n-1,2,4,15); fo<<"\nU 2 . . 4 15\n";
    for(int i=0; i<n; ++i) fo<<"e"<<i<<": "<< get(1,0,n-1,i)<<endl;
    x=get_s(1,0,n-1,1,4); fo<<"S 1 . . 4: "<<x<<endl;

    fo<<" ~~~~~~ \n";
    update(1,0,n-1,0,4,0); fo<<"\nU 0 . . 4 0\n";
    for(int i=0; i<n; ++i) fo<<"e"<<i<<": "<< get(1,0,n-1,i)<<endl;
    x=get_s(1,0,n-1,0,n-1); fo<<"S 0 . . 9: "<<x<<endl;
}

/*
Input
10

```

```
8 5 1 3 4 2 6 9 7 8
```

```
Output
```

```
e0: 8  
e1: 5  
e2: 1  
e3: 3  
e4: 4  
e5: 2  
e6: 6  
e7: 9  
e8: 7  
e9: 8  
.....  
v 5: 2  
v 7: 9
```

```
U 3 . . 6 10
```

```
e0: 8  
e1: 5  
e2: 1  
e3: 10  
e4: 10  
e5: 10  
e6: 10  
e7: 9  
e8: 7  
e9: 8  
S 0 . . 4: 34
```

```
-----
```

9. Cây phân khúc hai chiều

Cây phân khúc có thể mở rộng áp dụng cho mảng nhiều chiều. Nguyên lý triển khai cho mảng nhiều chiều khác biệt không nhiều với việc áp dụng cho mảng hai chiều. Vì vậy dưới đây ta sẽ xét việc tổ chức cây cho trường hợp hai chiều. Cũng như trong trường hợp một chiều, cây phân khúc phát huy được hiệu quả tối đa với các bài toán tìm Min/Max (RMQ). Với bài toán tìm tổng (RSQ) cây chỉ mang lại hiệu quả cao khi có các truy vấn cập nhật giá trị phần tử của mảng. Trong trường hợp không có truy vấn cập nhật, hàm tổng tiền tố hai chiều sẽ là công cụ hiệu quả nhất (cả về chi phí bộ nhớ lẫn thời gian thực hiện).

Quản lý tổng với mảng hai chiều có cập nhật giá trị phần tử mảng

Bài toán

Cho mảng $\mathbf{A} = ((\mathbf{a}_{x,y}))$, trong đó $\mathbf{a}_{x,y}$ – nguyên, $x = 1 \div n$, $y = 1 \div m$ và yêu cầu xử lý các truy vấn, mỗi truy vấn thuộc một trong 2 loại:

1 **lx rx ly ry** – Tìm $\sum_{i=lx}^{rx} \sum_{j=ly}^{ry} a_{i,j}$,

2 **qx qy val** – cho $\mathbf{a}_{qx,qy} = \mathbf{val}$.

Xây dựng cây

Xây dựng cây phân khúc bình thường, trong đó mỗi phần tử quản lý một dòng và vì vậy mỗi phần tử sẽ là một cây phân khúc một chiều!

Với cây này khi xét các dòng trong đoạn **[lx, rx]** ta có toàn bộ thông tin về băng **[lx, rx] [0 .. m-1]** của ma trận **A**.

Dễ dàng thấy rằng để xây dựng cây cần có 2 hàm:

- Hàm **build_x** xây dựng cây cho các dòng,
- Hàm **build_y** xây dựng cây cho mỗi nút của cây quản lý dòng.

Hàm xây dựng cây cho mỗi nút của cây quản lý dòng không khác hàm xây dựng cây thông thường ngoại trừ phải phân biệt trường hợp nút lá của cây trong một dòng và nút lá của cây quản lý dòng.

Tham số của hàm **build_y** :

- **vx** – gốc cây/cây con quản lý dòng,
- **lx, rx** – phạm vi quản lý các dòng của gốc **vx**,
- **ly, ry** – phạm vi cột do nút quản lý.

Lời gợi:

```
build_y (vx, lx, rx, 1, 0, m-1);
```

```

void build_y (int vx, int lx, int rx, int vy, int ly, int ry)
{
    if (ly == ry)
        if (lx == rx)
            t[vx][vy] = a[lx][ly];
        else
            t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
    else
    {
        int my = (ly + ry) / 2;
        build_y (vx, lx, rx, vy*2, ly, my);
        build_y (vx, lx, rx, vy*2+1, my+1, ry);
        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
    }
}

```

Tham số hàm **build_x**:

- + **vx** – gốc cây/cây con quản lý dòng,
- + **lx, rx** – phạm vi quản lý các dòng của gốc **vx**,

Lời gọi:

```
build_x (1, 0, n-1);
```

```

void build_x (int vx, int lx, int rx)
{
    if (lx != rx)
    {
        int mx = (lx + rx) / 2;
        build_x (vx*2, lx, mx);
        build_x (vx*2+1, mx+1, rx);
    }
    build_y (vx, lx, rx, 1, 0, m-1);
}

```

Chi phí bộ nhớ: **16nm**,

Độ phức tạp: Tuyến tính.

Xử lý truy vấn tính tổng

Tách thành hai hàm tương tự như khi xây dựng cây.

Độ phức tạp: $O(\log n \log m)$.

Lời gọi:

```
sum_x (1, 0, n-1, lx, rx, ly, ry);
```

Dòng cần tính

Cây con y

*Phạm vi tính
theo y*

```
int sum_y(int vx, int vy, int tly, int try, int ly, int ry)
{
    if (ly > ry)
        return 0;
    if (ly == tly && try == ry)
        return t[vx][vy];
    int tmy = (tly + try) / 2;
    return sum_y (vx, vy*2, tly, tmy, ly, min(ry, tmy))
        + sum_y (vx, vy*2+1, tmy+1, try, max(ly, tmy+1), ry);
}

int sum_x(int vx, int tlx, int trx, int lx, int rx, int ly, int ry)
{
    if (lx > rx)
        return 0;
    if (lx == tlx && trx == rx)
        return sum_y (vx, 1, 0, m-1, ly, ry);
    int tmx = (tlx + trx) / 2;
    return sum_x (vx*2, tlx, tmx, lx, min(rx, tmx), ly, ry)
        + sum_x (vx*2+1, tmx+1, trx, max(lx, tmx+1), rx, ly, ry);
}
```

Xử lý truy vấn cập nhật

Đầu tiên duyệt theo x, sau đó – theo y,

Các trường hợp cần xét khi duyệt theo y: Tương tự như ở các hàm đã nêu.

Độ phức tạp: O(lognlogm).

Lời gọi:

```
update_x (1, 0, n-1, x, y, new_val);
```

```

void update_y (int vx, int lx, int rx, int vy, int ly,
              int ry, int x, int y, int new_val)
{
    if (ly == ry)
    {
        if (lx == rx)
            t[vx][vy] = new_val;
        else
            t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
    }
    else
    {
        int my = (ly + ry) / 2;
        if (y <= my)
            update_y(vx, lx, rx, vy*2, ly, my, x, y, new_val);
        else
            update_y(vx, lx, rx, vy*2+1, my+1, ry, x, y, new_val);
        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
    }
}

void update_x(int vx,int lx,int rx,int x,int y,int new_val)
{
    if (lx != rx)
    {
        int mx = (lx + rx) / 2;
        if (x <= mx)
            update_x (vx*2, lx, mx, x, y, new_val);
        else
            update_x (vx*2+1, mx+1, rx, x, y, new_val);
    }
    update_y (vx, lx, rx, 1, 0, m-1, x, y, new_val);
}

```

Chương trình minh họa

```
// Two_Dimensions Segment Tree with value update

#include <bits/stdc++.h>
#define ff first
#define ss second
using namespace std;
typedef pair<int,int> pii;
ifstream fi ("a.inp");
ofstream fo ("a.out");
int n,m,p,x;
int t[400][400],a[10][10];

void build_y (int vx, int lx, int rx, int vy, int ly, int ry)
{
    if (ly == ry)
        if (lx == rx)
            t[vx][vy] = a[lx][ly];
        else
            t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
    else
    {
        int my = (ly + ry) / 2;
        build_y (vx, lx, rx, vy*2, ly, my);
        build_y (vx, lx, rx, vy*2+1, my+1, ry);
        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
    }
}

void build_x (int vx, int lx, int rx)
{
    if (lx != rx)
    {
        int mx = (lx + rx) / 2;
        build_x (vx*2, lx, mx);
        build_x (vx*2+1, mx+1, rx);
    }
    build_y (vx, lx, rx, 1, 0, m-1);
}

int sum_y (int vx, int vy, int tly, int try, int ly, int ry)
{
    if (ly > ry)
        return 0;
    if (ly == tly && try == ry)
        return t[vx][vy];
    int tmy = (tly + try) / 2;
    return sum_y (vx, vy*2, tly, tmy, ly, min(ry,tmy))
        + sum_y (vx, vy*2+1, tmy+1, try, max(ly,tmy+1), ry);
}

int sum_x (int vx, int tlx, int trx, int lx, int rx, int ly, int ry)
{
    if (lx > rx)
        return 0;
    if (lx == tlx && trx == rx)
        return sum_y (vx, 1, 0, m-1, ly, ry);
    int tmx = (tlx + trx) / 2;
    return sum_x (vx*2, tlx, tmx, lx, min(rx,tmx), ly, ry)
        + sum_x (vx*2+1, tmx+1, trx, max(lx,tmx+1), rx, ly, ry);
}
```

```

void update_y (int vx, int lx, int rx, int vy, int ly, int ry,
               int x, int y,      int new_val)
{
    if (ly == ry)
    {
        if (lx == rx)
            t[vx][vy] = new_val;
        else
            t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
    }
    else
    {
        int my = (ly + ry) / 2;
        if (y <= my)
            update_y (vx, lx, rx, vy*2, ly, my, x, y, new_val);
        else
            update_y (vx, lx, rx, vy*2+1, my+1, ry, x, y, new_val);
        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
    }
}

void update_x (int vx, int lx, int rx, int x, int y, int new_val)
{
    if (lx != rx)
    {
        int mx = (lx + rx) / 2;
        if (x <= mx)
            update_x (vx*2, lx, mx, x, y, new_val);
        else
            update_x (vx*2+1, mx+1, rx, x, y, new_val);
    }
    update_y (vx, lx, rx, 1, 0, m-1, x, y, new_val);
}

int main()
{
    fi>>n>>m;
    for(int i=0; i<n; ++i)
        for(int j=0; j<m; ++j) fi>>a[i][j];
    build_x(1,0,n-1);
    fo<<"-----\n";
    x=sum_x(1,0,n-1,1,2,2,3);
    fo<<"1-2,2-3 : "<<x<<endl;
    update_x(1,0,n-1,1,2,9);  fo<<"U 1,2: 9\n";
    x=sum_x(1,0,n-1,1,2,2,3);
    fo<<"1-2,2-3 : "<<x<<endl;
}
/*
Input
4 4
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

Output
-----
1-2,2-3 : 38
U 1,2: 9
1-2,2-3 : 40
*/

```

Tìm phần tử 0 thứ k trong đoạn

Việc tìm phần tử giá trị x thứ k trong đoạn $[ql, qr]$ có thể đưa về việc tìm phần tử 0 thứ k nếu thay a_i bằng $a_i - x$ với $i = 1 \dots n$.

Nếu không có các truy vấn cập nhật dữ liệu yêu cầu nêu trên có thể đáp ứng bằng cách dùng hàm tiền tố $f(i)$ tính số lượng số 0 từ đầu dãy tới vị trí i và tổ chức tìm kiếm nhị phân trong đoạn $[ql, qr]$. Độ phức tạp của giải thuật sẽ là $O(\log n)$.

Xây dựng cây

Nút lá t_i ứng với phần tử a_i được gán giá trị 0 nếu $a_i \neq 0$ và bằng 1 trong trường hợp ngược lại.

```
void build(int v, int l, int r)
{
    if (l == r) t[v] = (a[l]==0);
    else
    {
        int mid = (l + r) / 2;
        build(v * 2, l, mid);
        build(v * 2 + 1, mid + 1, r);
        t[v] = t[v * 2] + t[v * 2 + 1];
    }
}
```

Cập nhật giá trị phần tử ở vị trí pos cũng được chỉnh lý tương tự:

```
void upd(int v, int l, int r, int pos, int val)
{
    if (l == r) t[v] = (val==0), a[l] = val;
    else
    {
        int mid = (l + r) / 2;
        if (pos <= mid) upd(v * 2, l, mid, pos, val);
        else upd(v * 2 + 1, mid + 1, r, pos, val);
        t[v] = t[v * 2] + t[v * 2 + 1];
    }
}
```

Để giảm chi phí lập trình ta xây dựng hàm hỗ trợ `int get_kth (int v, int tl, int tr, int p)` tìm phần tử 0 thứ p kể từ đầu dãy.

Hàm này đồng thời phục vụ kiểm có tồn tại số 0 thứ k trong đoạn đang xét hay không.

Ngoài ra, việc sử dụng hàm `int get (int v, int l, int r, int ql, int qr)` tính tổng trong đoạn $[ql, qr]$ đã xây dựng ở các phần trên sẽ đơn giản hóa quá trình tìm kiếm.

Việc tìm phần tử 0 thứ **k** trong đoạn [**t_l**,**t_r**] bao gồm các bước:

Kiểm tra số lượng phần tử 0 trong đoạn cần tìm có nhiều hơn hoặc bằng **k** hay không, nếu ít hơn – đưa ra kết quả -1,

Xác định phần tử 0 cần tìm trong đoạn tương ứng với phần tử 0 nào tính từ đầu dãy,

Tiến hành tìm kiếm vị trí của phần tử 0 tương ứng đã xác định ở bước trên.

Đây là sơ đồ lập trình theo *nguyên lý triển khai các Macro* sẵn có.

Độ phức tạp của giải thuật: O(logn).

Mô đun dẫn xuất vị trí số 0 thứ **k** trong đoạn:

```
int find_kth(int l, int r, int k)
{
    int t1, t2;
    t1 = get(l, 0, n-1, 1, r);
    if(t1 < k) return -1;
    if(l == 0) t1 = 0; else t1 = get(l, 0, n-1, 0, l-1);
    return get_kth(l, 0, n-1, t1+k);
}
```

Chương trình minh họa

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("Segment_Kzr.inp");
ofstream fo ("Segment_Kzr.out");
int n;
vector<int> t, a, vd;

void build(int v, int l, int r)
{
    if (l == r) t[v] = (a[l]==0);
    else
    {
        int mid = (l + r) / 2;
        build(v * 2, l, mid);
        build(v * 2 + 1, mid + 1, r);
        t[v] = t[v * 2] + t[v * 2 + 1];
    }
}

void upd(int v, int l, int r, int pos, int val)
{
    if (l == r) t[v] = (val==0), a[l] = val;
    else
    {
        int mid = (l + r) / 2;
        if (pos <= mid) upd(v * 2, l, mid, pos, val);
        else upd(v * 2 + 1, mid + 1, r, pos, val);
        t[v] = t[v * 2] + t[v * 2 + 1];
    }
}

int get_kth (int v, int tl, int tr, int k)
{
    if (k > t[v]) return -1;
    if (tl == tr) return tl;
    int tm = (tl + tr) / 2;
    if (t[v*2] >= k) return get_kth (v*2, tl, tm, k);
    else return get_kth (v*2+1, tm+1, tr, k - t[v*2]);
}

int get(int v, int l, int r, int ql, int qr)
{
    if (l > qr || r < ql) return 0;
    if (ql <= l && r <= qr) return t[v];
    int mid = (l + r) / 2;
    return get(v * 2, l, mid, ql, qr) +
           get(v * 2 + 1, mid + 1, r, ql, qr);
}
```

```

int find_kth(int l,int r, int k)
{
    int t1,t2;
    t1=get(1,0,n-1,l,r);
    if(t1<k) return -1;
    if(l==0)t1=0; else t1=get(1,0,n-1,0,l-1);
    return get_kth(1,0,n-1,t1+k);
}

int main()
{
    fi>>n;
    a.assign(2*n,0); t.assign(4*n+4,0); vd.assign(4*n+4,0);
    for(int i=0; i<n; ++i) fi>>a[i];
    build(1,0,n-1);
    int res,t1,t2;
    res=find_kth(3,9,3);
    fo<<res<<endl;
    res=find_kth(1,8,2);
    fo<<res<<endl;
    res=find_kth(1,5,4);
    fo<<res<<endl;
    upd(1,0,n-1,3,0);
    res=find_kth(1,5,4);
    fo<<res<<endl;
}

```

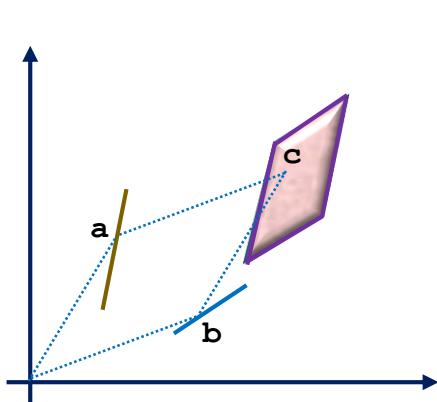


Tổng Minkowski

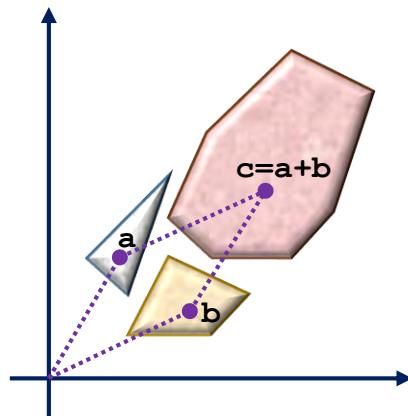
1. Định nghĩa

Cho 2 điểm trên mặt phẳng $\mathbf{a}(\mathbf{x}_a, \mathbf{y}_a)$ và $\mathbf{b}(\mathbf{x}_b, \mathbf{y}_b)$, tổng $\mathbf{a}+\mathbf{b}$ theo Minkowski là điểm $\mathbf{c}(\mathbf{x}_c, \mathbf{y}_c)$, trong đó $\mathbf{x}_c = \mathbf{x}_a + \mathbf{x}_b$, $\mathbf{y}_c = \mathbf{y}_a + \mathbf{y}_b$.

Tổng Minkowski của 2 tập \mathbf{A} và \mathbf{B} là tập $\mathbf{C} = \{\mathbf{a}+\mathbf{b}: \mathbf{a} \in \mathbf{A}, \mathbf{b} \in \mathbf{B}\}$.



Tổng 2 đoạn thẳng



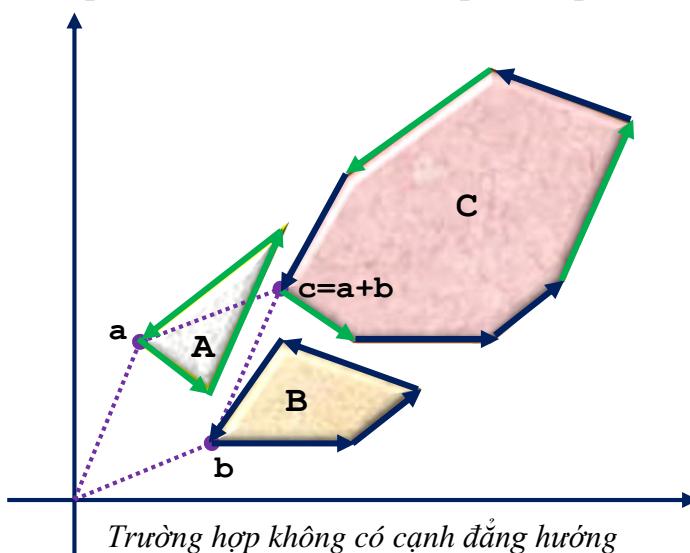
Tổng 2 đa giác lồi

2. Xây dựng tổng hai đa giác lồi

Có thể chứng minh được rằng tổng Minkowski của 2 đa giác lồi là một đa giác lồi.

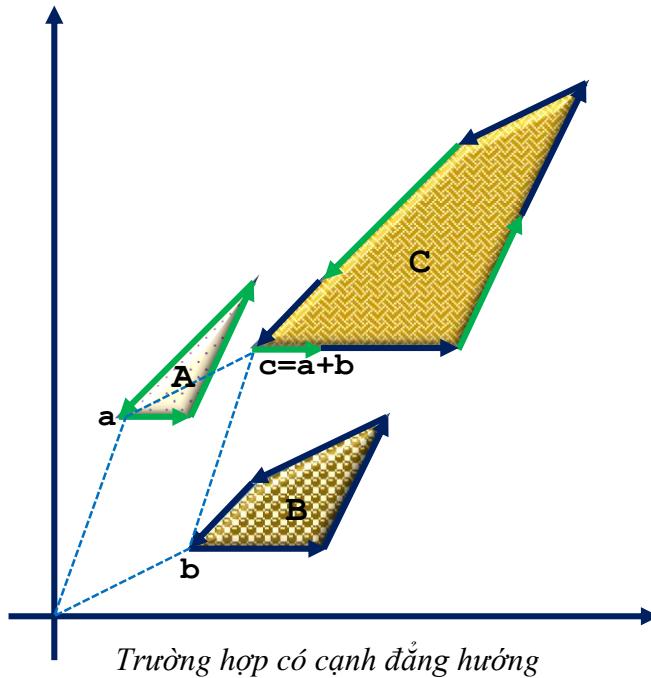
Trong trường hợp không có cặp cạnh nào của 2 đa giác (một cạnh thuộc đa giác thứ nhất, cạnh kia – thuộc đa giác thứ 2) đồng hướng thì đa giác tổng có số đỉnh bằng tổng số đỉnh của 2 đa giác ban đầu.

Nếu duyệt cạnh các đa giác theo cùng một chiều (cùng chiều hoặc ngược chiều kim đồng hồ), tập các cạnh của \mathbf{C} sẽ là hợp các tập cạnh của \mathbf{A} và \mathbf{B} .

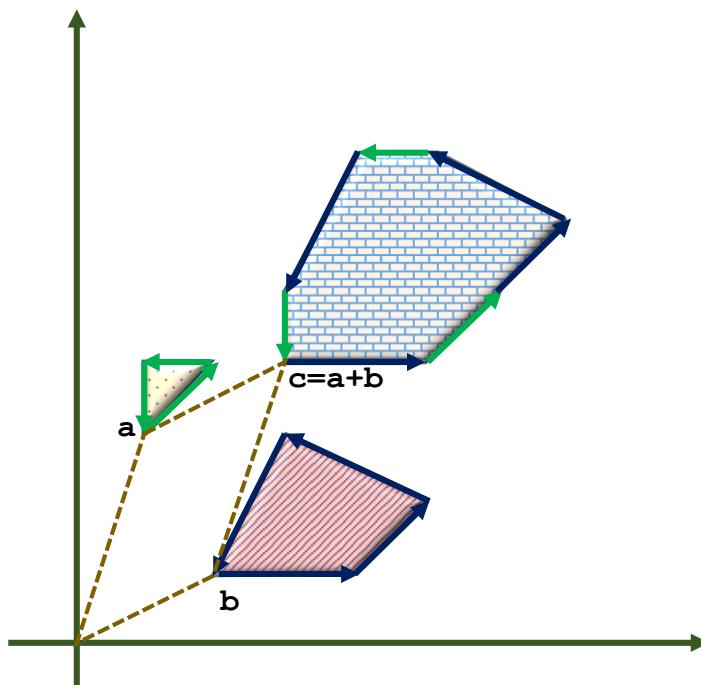


Trường hợp không có cạnh đồng hướng

Trường hợp **A** có cạnh \vec{a} đồng hướng với \vec{b} , trong đa giác **C** sẽ có một số đỉnh liên tiếp thẳng hàng.



Để xác định **C** cần liệt kê các đỉnh của **A** và **B** theo cùng một chiều, bắt đầu từ đỉnh có thứ tự từ điểm nhỏ nhất. Gọi 2 đỉnh tương ứng là **a** và **b**. Khi đó $\mathbf{c} = \mathbf{a} + \mathbf{b}$ là đỉnh có thứ tự từ điểm nhỏ nhất của **C**. Bắt đầu từ điểm **c** lần lượt đặt các cạnh của **A** và **B** trong dãy đã sắp xếp.



3. Ứng dụng

Hai trong số các ứng dụng của tổng Minkowski là kiểm tra sự giao nhau của 2 đa giác lồi và tìm khoảng cách giữa 2 đa giác lồi.

3.1 Kiểm tra giao hai đa giác lồi

Cho 2 đa giác lồi A và B . Hãy xác định giao của 2 đa giác này có khác rỗng hay không

$$A \cap B \neq \emptyset \iff \exists p, p \in A \text{ và } p \in B$$

Xét B' là hình đối xứng qua tâm của B với tâm đối xứng là gốc tọa độ. B' nhận được từ B bằng cách đổi dấu tọa độ các đỉnh.

$$A \cap B \neq \emptyset \iff \exists p, p \in A \text{ và } -p \in B' \iff 0 \in A + B'$$

Như vậy việc kiểm tra gốc tọa độ có thuộc $A + B'$ hay không sẽ trả lời được câu hỏi đã nêu.

3.2 Tính khoảng cách giữa 2 đa giác lồi

Gọi $dist(A, B)$ là khoảng cách giữa 2 tập A và B .

$$dist(A, B) = \min_{a \in A, b \in B} |a - b|$$

Xét trường hợp A, B – hai đa giác lồi. Dễ dàng xác định được B' – hình *đối xứng qua tâm* của B với *tâm đối xứng* là *gốc tọa độ*.

Gọi C là tổng Minkowski của A và B' : $C = A + B'$.

Ta có

$$dist(A, B) = \min_{a \in A, b \in B} |a - b| = \min_{a \in A, b \in B'} |a + b| = \min_{a \in A + B'} |c|$$

Từ đây suy ra khoảng cách giữa 2 đa giác lồi A, B bằng *khoảng cách từ gốc tọa độ tới các cạnh của $A + B'$* hoặc *bằng 0 nếu A và B giao nhau* (trong trường hợp này $A + B'$ chứa gốc tọa độ).

Tìm kiếm tam phân (*Ternary Search*)

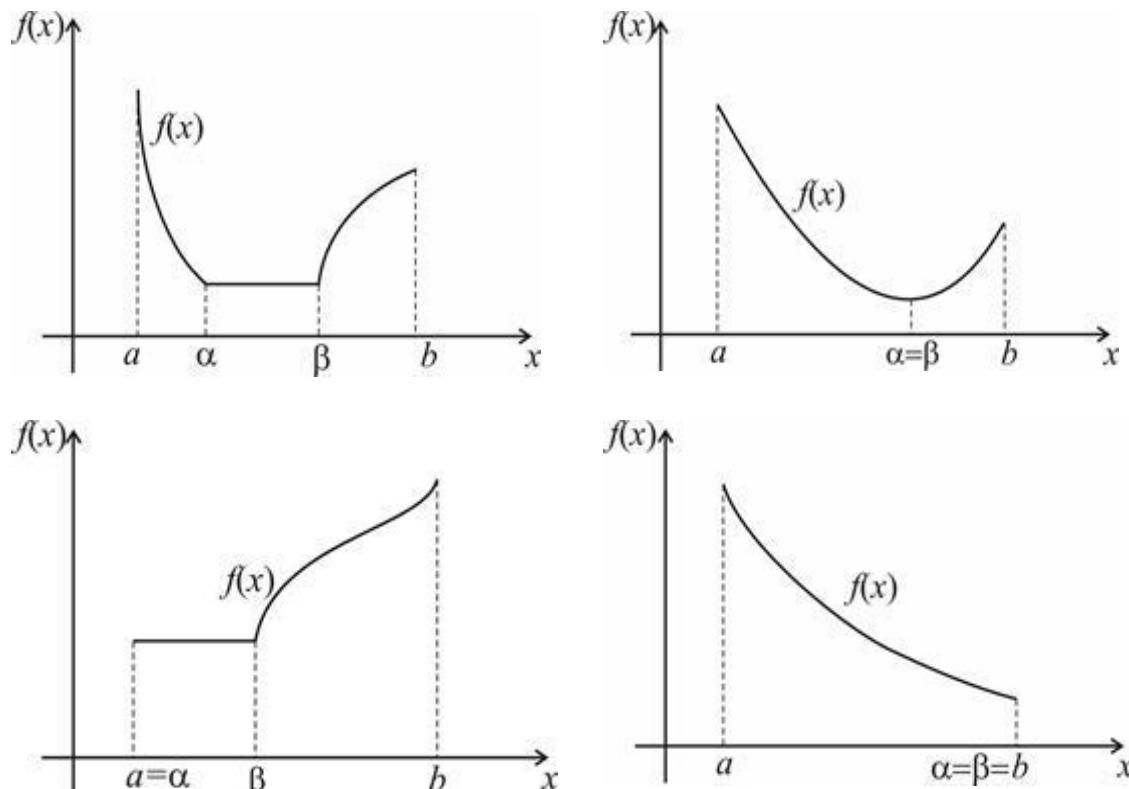
1. Hàm unimodal

Hàm $f(x)$ xác định trên đoạn $[a,b]$ được gọi là unimodal nếu thuộc một trong 2 dạng sau:

Dạng thứ I:

- ✚ Liên tục trên đoạn $[a,b]$,
- ✚ Tồn tại α và β thỏa mãn các điều kiện:
 - $a \leq \alpha \leq \beta \leq b$,
 - $f(x)$ đơn điệu giảm với $a \leq x \leq \alpha$ (nếu $a < \alpha$),
 - $f(x)$ đơn điệu tăng với $\beta \leq x \leq b$ (nếu $\beta < b$),
 - $f(x) = f(y)$ với mọi $x, y \in [\alpha, \beta]$.

Ví dụ:



Đồ thị hàm unimodal dạng I

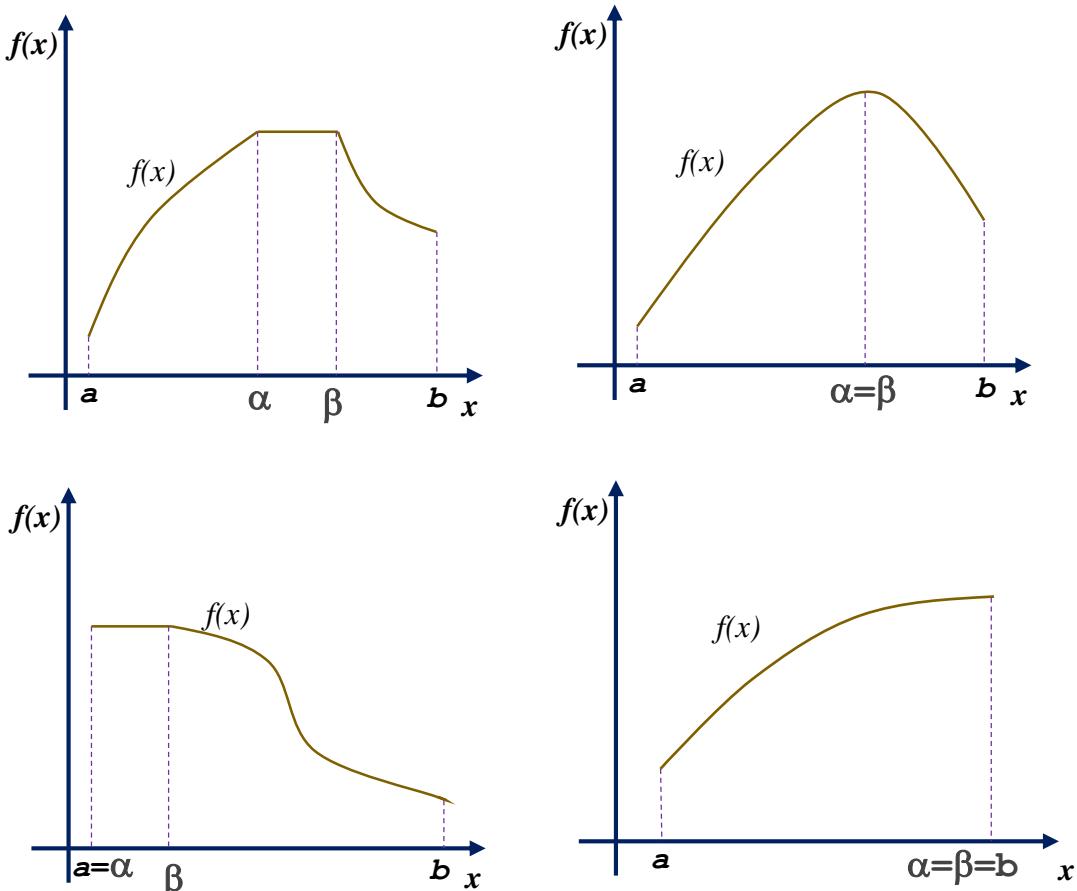
Dạng thứ II:

- ✚ Liên tục trên đoạn $[a,b]$,

➡ Tồn tại α và β thỏa mãn các điều kiện:

- $a \leq \alpha \leq \beta \leq b$,
- $f(x)$ đơn điệu tăng với $a \leq x \leq \alpha$ (nếu $a < \alpha$),
- $f(x)$ đơn điệu giảm với $\beta \leq x \leq b$ (nếu $\beta < b$),
- $f(x) = f(y)$ với mọi $x, y \in [\alpha, \beta]$.

Ví dụ:



Đồ thị hàm unimodal dạng II

Dưới đây chúng ta sẽ xét các *hàm unimodal dạng II*. Lập luận đối xứng ta sẽ có kết quả tương ứng với hàm dạng I.

Tính chất:

Cực đại địa phương bất kỳ trên đoạn $[a,b]$ cũng là cực đại toàn cục,

Hàm $f(x)$ unimodal trên đoạn $[a,b]$ cũng là unimodal trên đoạn $[c,d] \subset [a,b]$,

Gọi $Q(a,b)$ tập giá trị của $f(x)$ trên $[a,b]$, x^* là một trong số các điểm $f(x)$ đạt cực trị và $a \leq x_1 < x_2 \leq b$, khi đó

- ✳ $f(x_1) \geq f(x_2) \rightarrow x^* \in [x_1, b]$
- ✳ $f(x_1) < f(x_2) \rightarrow x^* \in [a, x_2]$.

2. Giải thuật tìm kiếm

Cho hàm $f(x)$ unimodal loại II trên đoạn $[lf, rt]$. Yêu cầu tìm giá trị lớn nhất của $f(x)$ trên đoạn $[lf, rt]$.

Giải thuật:

- + Chọn 2 điểm $m1$ và $m2$ bất kỳ thỏa mãn điều kiện $lf < m1 < m2 < rt$,
- + Tính $f(m1), f(m2)$,
- + Điều chỉnh miền tìm kiếm:
 - ✳ Nếu $f(m1) > f(m2) \rightarrow rt = m2$,
 - ✳ Nếu $f(m1) < f(m2) \rightarrow lf = m1$,
 - ✳ Nếu $f(m1) = f(m2) \rightarrow$ Tìm được *max*, điểm đạt *max* – chọn điểm tùy ý thuộc $[m1, m2]$, dừng tìm kiếm.
- + Lặp lại các bước trên cho đến khi tìm được *max* hoặc đoạn $[lf, rt]$ nhỏ hơn độ chính xác cần thiết.

Các phương pháp chọn $m1$ và $m2$ tùy thuộc vào từng bài toán cụ thể và quyết định tốc độ hội tụ.

Thông thường $m1$ và $m2$ được chọn để chia đoạn $[lf, rt]$ thành 3 phần bằng nhau:

$$m1 = lf + \frac{rt-lf}{3}$$

$$m2 = rt - \frac{rt-lf}{3}$$

Lưu ý trường hợp đổi số x chỉ nhận giá trị nguyên:

Đoạn $[lf, rt]$ vẫn được chia thành 3 phần nhưng có thể không bằng nhau, Khi $rt-lf < 3$ – không thể chia đoạn thành 3 phần → cần duyệt trực tiếp giá trị $f(x)$ trong đoạn để tìm *max*.

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

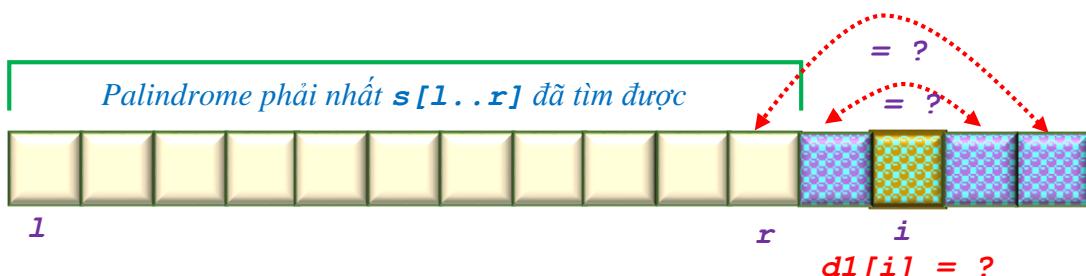
Xét xâu con $u = s[i..j]$. Nếu $j-i+1$ là lẻ thì $p = (i+j)/2$ được gọi là vị trí trung tâm (gọi ngắn gọn là **tâm**) của u , nếu $j-i+1$ chẵn thì tâm là $p = (i+j+1)/2$.

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 . Giá thiết $s[1..r]$ là palindrome phải nhất trong số các palindrome tìm được.

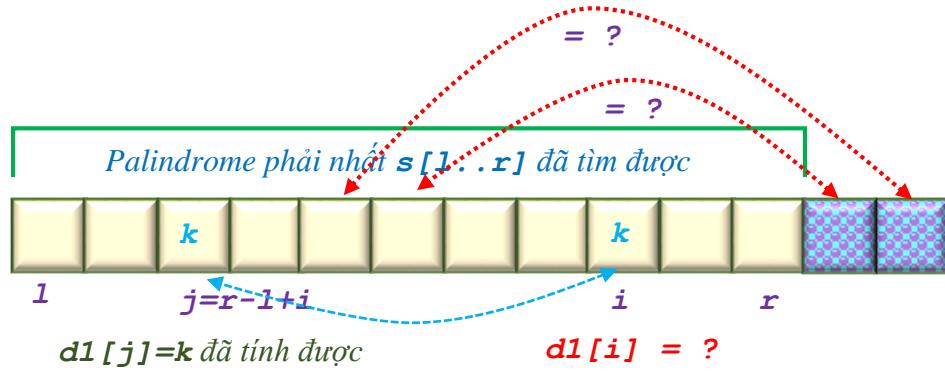
Giả thiết $d1[j]$ và $d2[j]$ đã tính được với mọi $j < i$.

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} , ... 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 $O(n)$.



NGUYÊN LÝ BÙ – TRỪ

Nguyên lý bù trừ (*The principle of inclusions-exceptions*) là một cách tiếp cận có hiệu quả để giải quyết các bài toán tổ hợp đòi hỏi đếm cấu hình, tính số lượng phương án, tính xác suất sự kiện, tính kích thước tập hợp, . . .

1. Phát biểu nguyên lý

1.1 Mô tả phương pháp

Để tính kích thước hợp của các tập hợp đã cho cần tính tổng **kích thước của từng tập hợp**, sau đó trừ đi kích thước **giao của từng cặp tập hợp**, cộng tiếp vào kết quả nhận được kích thước **giao của từng nhóm 3 tập hợp**, trừ tiếp kích thước **giao của các nhóm 4 tập hợp**, . . . Tiếp tục quá trình điều chỉnh kết quả nêu trên cho đến khi xử lý xong kích thước **giao của tất cả các tập hợp**.

1.2 Trên ngôn ngữ tập hợp

Abraham de Moivre đã đưa ra phát biểu nguyên lý bù trừ trên ngôn ngữ đại số tập hợp:

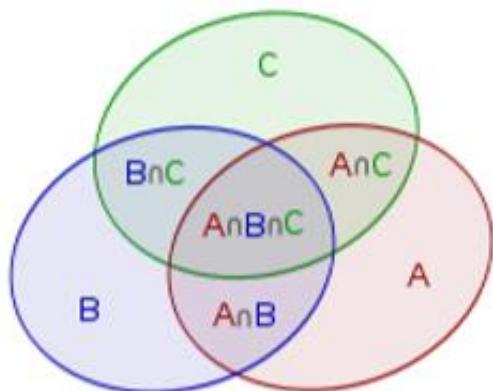
$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{i=1}^n |A_i| - \sum_{\substack{i,j: \\ 1 \leq i < j \leq n}} |A_i \cap A_j| + \sum_{\substack{i,j,k: \\ 1 \leq i < j < k \leq n}} |A_i \cap A_j \cap A_k| - \dots + (-1)^{n-1} |A_1 \cap \dots \cap A_n|.$$

Một cách ngắn gọn hơn, nguyên lý có thể được thể hiện qua công thức:

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{C \subseteq B} (-1)^{\text{size}(C)-1} \left| \bigcap_{e \in C} e \right|$$

1.3 Biểu đồ Venn

Xét 3 hình tròn A, B và C. Yêu cầu xác định diện tích vùng được phủ bởi 3 hình tròn này, tức là cần tính $A \cap B \cap C$.



$$S(A \cup B \cup C) = S(A) + S(B) + S(C) - S(A \cap B) - S(A \cap C) - S(B \cap C) + S(A \cap B \cap C)$$

Cần lưu ý Euler cũng đề xuất biểu đồ dạng tương tự nhưng xuất phát từ cơ sở lý thuyết khác.

Biểu đồ Venn được xây dựng dựa trên lý thuyết và ngôn ngữ của đại số tập hợp.

Biểu đồ Euler được xây dựng theo lý thuyết đại số Boolean, chính xác hơn – theo nguyên lý Tam đoạn luận của Aristotle.

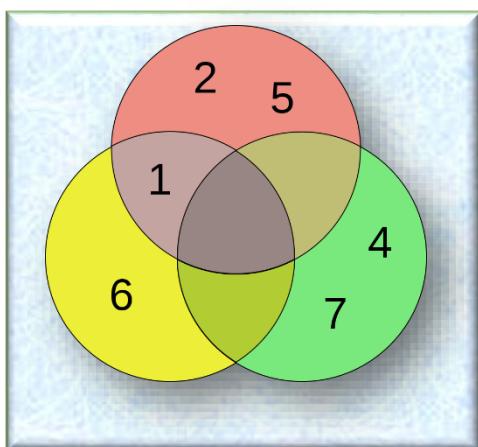
Ví dụ, có 3 tập hợp chứa các số nguyên:

$$A = \{1, 2, 5\}$$

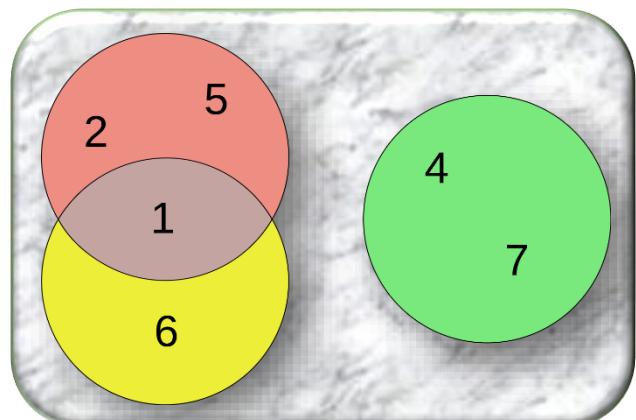
$$B = \{1, 6\}$$

$$C = \{4, 7\}$$

Các biểu đồ Vienn và Euler có dạng



Biểu đồ Venn



Biểu đồ Euler
(Vòng tròn Euler)

Nhóm biểu đồ này được gọi chung là biểu đồ Vienn – Euler hay đôi khi – gọi ngắn gọn là Vòng tròn Euler.

1.3 Chứng minh

Để đảm bảo ngắn gọn và chặt chẽ ta sẽ dùng ngôn ngữ đại số tập hợp trong chứng minh

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{C \subseteq B} (-1)^{\text{size}(C)-1} \left| \bigcap_{e \in C} e \right|$$

Trong đó \mathbf{B} – tập hợp hình thành từ các \mathbf{A}_i .

Cần phải chứng minh mỗi phần tử thuộc một trong các tập \mathbf{A}_i được đếm đúng một lần.

Xét \mathbf{x} – phần tử bất kỳ thuộc đúng \mathbf{k} tập hợp trong số các \mathbf{A}_i ($\mathbf{k} \geq 1$).

Đễ dàng thấy rằng:

- ✿ Trong số các tập hợp toán hạng có $\text{size}(\mathbf{C}) = 1$, \mathbf{x} được tính đúng \mathbf{k} lần với dấu +,
- ✿ Trong số các tập hợp toán hạng có $\text{size}(\mathbf{C}) = 2$, \mathbf{x} được tính đúng C_k^2 lần với dấu trừ (-) vì \mathbf{x} được tính trong các cặp tập hợp có chứa nó,
- ✿ Trong số các tập hợp toán hạng có $\text{size}(\mathbf{C}) = 3$, \mathbf{x} được tính đúng C_k^3 lần với dấu +,
- ✿
- ✿ Trong số các tập hợp toán hạng có $\text{size}(\mathbf{C}) = \mathbf{k}$, \mathbf{x} được tính đúng C_k^k lần với dấu $(-1)^{k-1}$,
- ✿ Trong số các tập hợp toán hạng có $\text{size}(\mathbf{C}) > \mathbf{k}$, \mathbf{x} không được tính lần nào.

Gọi \mathbf{T} – kết quả tính được đối với \mathbf{x} , ta có

$$T = C_k^1 - C_k^2 + C_k^3 - \dots + (-1)^{i-1} \cdot C_k^i + \dots + (-1)^{k-1} \cdot C_k^k$$

Xét việc khai triển nhị thức Newton $(1-\mathbf{x})^{\mathbf{k}}$:

$$(1-x)^k = C_k^0 - C_k^1 \cdot x + C_k^2 \cdot x^2 - C_k^3 \cdot x^3 + \dots + (-1)^k \cdot C_k^k \cdot x^k$$

Với $x = 1$ ta có $T = 1 - (1-\mathbf{x})^{\mathbf{k}} = 1$ - Điều phải chứng minh.

2. Ứng dụng

Để hiểu rõ nguyên lý bù trừ ta sẽ xét một số ví dụ từ đơn giản đến các bài toán có nội dung thực tế khó giải quyết nếu vòng tránh nguyên lý bù trừ.

Nguyên lý bù trừ cũng cho phép, trong một số trường hợp, thay thế giải thuật độ phức tạp lũy thừa bằng giải thuật độ phức tạp đa thức.

2.1 Bài toán hoán vị

Xét dãy các chữ số từ 0 đến 9. Có bao nhiêu cách đổi chỗ các chữ số để chữ số đầu tiên lớn hơn 1 và chữ số cuối cùng nhỏ hơn 8?

Trước hết tính số hoán vị “xấu”, trong đó chữ số đầu tiên nhỏ hơn 2 và hay hoặc chữ số cuối cùng lớn hơn hoặc bằng 8.

Gọi X là tập các hoán vị cho chữ số đầu tiên nhỏ hơn 2, Y – tập các hoán vị cho chữ số cuối cùng lớn hơn 7.

Ta có số lượng các hoán vị cần loại bỏ là $|X|+|Y|-|X \cap Y|$.

Số lượng các hoán vị bắt đầu bằng 0 là $9!$,

Số lượng các hoán vị bắt đầu bằng 1 là $9!$,

Như vậy $X = 2 \times 9!$

Lý luận tương tự khi xét các hoán vị kết thúc bằng 8 hoặc 9 ta có $Y = 2 \times 9!$

Số lượng các hoán vị bắt đầu bằng 0 và kết thúc bằng 8 là $8!$,

Số lượng các hoán vị bắt đầu bằng 1 và kết thúc bằng 8 là $8!$,

Số lượng các hoán vị bắt đầu bằng 0 và kết thúc bằng 9 là $8!$,

Số lượng các hoán vị bắt đầu bằng 1 và kết thúc bằng 9 là $8!$.

Như vậy $|X|+|Y|-|X \cap Y| = 2 \times 9! + 2 \times 9! - 4 \times 8!$

Kết quả cần tìm sẽ là.

$$10! - (2 \times 9! + 2 \times 9! - 4 \times 8!)$$

2.2 Dãy số (0, 1, 2)

Có bao nhiêu xâu độ dài n từ các ký tự trong tập $\{0, 1, 2\}$ trong đó mỗi ký tự thuộc tập đã nêu xuất hiện ít nhất một lần?

Xét bài toán ngược lại: Có bao nhiêu xâu không chứa ít nhất một ký tự trong tập $\{0, 1, 2\}$?

Ký hiệu A_i – tập các xâu độ dài n không chứa i , $i = 0, 1, 2$.

Công thức dẫn xuất kết quả bài toán ngược sẽ là

$$|A_0| + |A_1| + |A_2| - |A_0 \cap A_1| - |A_0 \cap A_2| - |A_1 \cap A_2| + |A_0 \cap A_1 \cap A_2|$$

Lực lượng của mỗi tập \mathbf{A}_i sẽ là 2^n (vì được xây dựng từ 2 ký tự khác nhau).

Lực lượng của giao $\mathbf{A}_i \cap \mathbf{A}_j = 1$ (*chỉ còn một ký tự!*).

Lực lượng của giao $\mathbf{A}_i \cap \mathbf{A}_j \cap \mathbf{A}_k = 0$ (*không còn ký tự nào!*).

Tổng số xâu độ dài n không có ràng buộc nào là 3^n .

Kết quả cần tìm của bài toán ban đầu sẽ là

$$3^n - 3 \times 2^n + 3 - 0$$

2.3 Số lượng nghiệm nguyên của phương trình

Cho phương trình

$$\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + \mathbf{x}_4 + \mathbf{x}_5 + \mathbf{x}_6 = 20$$

trong đó $0 \leq \mathbf{x}_i \leq 8$, $i = 1 \div 6$.

Hãy xác định số nghiệm của phương trình.

Đầu tiên, ta xác định số nghiệm của phương trình đã có, bỏ qua điều kiện $0 \leq \mathbf{x}_i \leq 8$ với mọi i .

Việc xác định nghiệm tương tự như ta có dãy 20 chiếc hộp độ rộng mỗi hộp là đơn vị, ta chia dãy này thành 6 khúc bằng cách chèn thêm các hộp làm vách ngăn. Độ rộng từ đầu dãy tới vách ngăn đầu tiên sẽ là \mathbf{x}_1 , độ rộng giữa vách ngăn cuối cùng tới cuối dãy là \mathbf{x}_6 , độ rộng giữa vách ngăn thứ $i-1$ và thứ i sẽ là \mathbf{x}_i . Tất cả có 5 vách ngăn. Như vậy độ dài toàn dãy, kể cả vách ngăn là 25. Có bao nhiêu cách đặt vách ngăn – có bấy nhiêu nghiệm.



Số cách đặt vách ngăn là C_{25}^5 .

Cần tính và loại bỏ các nghiệm trong đó có một hoặc vài $\mathbf{x}_i \geq 9$.

Gọi \mathbf{A}_k ($k = 1 \div 6$) – số lượng nghiệm trong đó $\mathbf{x}_k \geq 9$, các nghiệm còn lại – không âm. Bỏ 9 hộp chắc chắn thuộc nghiệm thứ k , ta có dãy độ dài 16.

Lập luận đúng như cách đã nêu, ta có $|\mathbf{A}_k| = C_{16}^5$.

Dễ dàng thấy rằng $|\mathbf{A}_k \cap \mathbf{A}_m| = C_7^5$.

Giao của 3 hay nhiều tập hơn bằng 0 vì $3 \times 9 > 20$.

Nghiệm cần tìm của bài toán ban đầu sẽ là

$$C_{25}^5 - C_6^1 \times C_{16}^5 + C_6^2 \times C_7^5$$

2.4 Số lượng số nguyên tố cùng nhau trên đoạn thẳng

Bài toán:

Cho hai số nguyên dương n và r . Yêu cầu xác định số lượng số nguyên trên đoạn $[1, r]$ nguyên tố cùng nhau với n ($1 < n, r \leq 10^9$).

Giải thuật:

Xét phần bù của miền cần tìm: Xác định số lượng các số có ước chung khác 1 với n . Bài toán mới này dễ giải quyết hơn, dựa trực tiếp vào nguyên lý bù trừ.

Phân tích n ra thừa số nguyên tố. Gọi p_1, p_2, \dots, p_k là các số nguyên tố khác nhau nhận được.

Khi đó $q_i = \lfloor \frac{r}{p_i} \rfloor$ là số lượng các số có ước chung p_i với n .

Tổng $\sum_{i=1}^k q_i$ là số lượng các số không nguyên tố cùng nhau với n , nhưng trong đó các số đồng thời chia hết cho p_i và p_j với $i \neq j$ bị tính hai lần, do đó cần loại bỏ số lượng các số đồng thời là ước của cặp số nguyên tố của n .

Mỗi số là bội của 3 thừa số nguyên tố khác nhau sẽ bị đếm 2 lần ở bước đếm các số là bội của 2 thừa số nguyên tố khác nhau vì vậy cần cộng số lượng số là bội của 3 thừa số nguyên tố khác nhau vào tổng để chống việc bị trừ 2 lần,

Tương tự như vậy với số lượng các số là bội của 4, 5, 6, ... thừa số nguyên tố khác nhau.

Tổ chức tính toán:

Thay vì lần lượt tính tổ hợp chap 2, chap 3, ... tích các thừa số nguyên tố và điều chỉnh kết quả ta có thể dùng mặt nạ k bít.

Xét số nguyên $m \in [0, 2^k]$. Số bít 1 và vị trí của chúng xác định một tổ hợp các thừa số nguyên tố. Sẽ không có tổ hợp nào bị bỏ sót hay tính lặp. Số lượng phần tử trong tổ hợp, tức là số lượng bít 1 sẽ quyết định cần cộng hay trừ vào kết quả giá trị chính lý tìm được.

Với $n \leq 10^9$, số lượng thừa số khác nhau của n là không quá 9, việc tổ chức mặt nạ là khả thi và hiệu quả cao.

Độ phức tạp của giải thuật: $O(\sqrt{n})$.

Chương trình

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

int n, r, ans;

int main()
{
    fi>>n>>r;
    vector<int> p;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0)
    {
        p.push_back (i);
        while (n % i == 0) n /= i;
    }
    if (n > 1) p.push_back (n);
    int sum = 0;
    for (int msk=1; msk<(1<<p.size()); ++msk)
    {
        int mult = 1;
        int flg=0;
        for (int i=0; i<(int)p.size(); ++i)
            if (msk & (1<<i))
        {
            flg ^=1;
            mult *= p[i];
        }
        int cur = r / mult;
        if (flg) sum += cur; else sum -= cur;
    }
    ans= r - sum;
    fo<<ans;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



2.5 Số lượng bội trong đoạn

Bài toán:

Cho dãy số nguyên dương $\mathbf{A} = (a_1, a_2, \dots, a_n)$ và số nguyên r . Hãy xác định số lượng các số trong đoạn $[1, r]$ chia hết cho ít nhất một số của dãy \mathbf{A} .

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

- Dòng đầu tiên chứa 2 số nguyên n và r ($1 \leq n \leq 20, 1 \leq r \leq 10^9$),
- Dòng thứ 2 chứa n số nguyên a_1, a_2, \dots, a_n ($1 \leq a_i \leq 10^9, i = 1 \div n$).

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

Ví dụ:

MULTIPLE INP
4 30
5 6 8 21

MULTIPLE.OUT
13

Giải thuật

Áp dụng nguyên lý bù trừ với $a_i, i = 1 \div n$ tương tự như ở mục trên. Thay vì tích các số ở đây ta phải tìm bội số chung nhỏ nhất.

Chương trình

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

int n, r, ans=0;

int main ()
{
    fi>>n>>r;
    vector<int64_t> a(n);
    for (int64_t &i:a) fi>>i;

    for (int msk=1; msk<(1<<n); ++msk)
    {
        int64_t mult = 1;
        int flg=0;
        for (int i=0; i<n; ++i)
            if (msk & (1<<i))
            {
                flg^=1;
                mult = (mult*a[i]) / __gcd(mult,a[i]);
            }
        int cur = r / mult;
        if (flg) ans += cur; else ans -= cur;
    }
    fo<<ans;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



2.6 Số lượng đường đi

Bài toán:

Cho lưới ô vuông kích thước $n \times m$, ô $(1, 1)$ ở góc dưới trái, ô (n, m) – trên phải. Có k ô chứa chướng ngại vật, ô thứ i ở tọa độ (x_i, y_i) , $1 \leq x_i \leq n$, $1 \leq y_i \leq m$, $i = 1 \div k$. Không có chướng ngại vật ở ô $(1, 1)$ và (n, m) .

Rô bốt xuất phát từ ô $(1, 1)$, ở mỗi bước được chuyển sang ô kế cạnh bên phải hoặc bên trên nếu ô tới không chứa chướng ngại vật.

Hãy xác định số lượng đường rô bốt có thể đi từ ô $(1, 1)$ đến ô (n, m) và đưa ra số lượng theo mô đun p , trong đó p – số nguyên tố.

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

- ⊕ Dòng đầu tiên chứa 4 số nguyên n, m, k và p ($1 \leq n, m \leq 10^5$, $0 \leq k \leq 100$, $2 \times \max\{m, n\} < p < 2 \times 10^9$),
- ⊕ Nếu $k > 0$, dòng thứ i trong k dòng tiếp theo chứa 2 số nguyên x_i, y_i ($1 \leq x_i \leq n$, $1 \leq y_i \leq m$).

Kết quả: Đưa ra file văn bản ROUTE.OUT một số nguyên không âm – số lượng đường tìm được theo mô đun p .

Ví dụ:

ROUTE INP	ROUTE.OUT
5 6 3 101 2 2 3 5 4 2	25

Giải thuật

Giải thuật I

Với n và m đủ nhỏ ($n, m \leq 1000$) bài toán tìm số lượng đường đi dễ dàng giải quyết bằng *phương pháp quy hoạch động* với *độ phức tạp* $O(n \times m)$.

Gọi $dp_{i,j}$ – số lượng đường đi hợp lệ từ ô $(1,1)$ tới ô (i,j) , \mathcal{K} – tập các ô có vật cản.

Ban đầu $dp_{i,j} = -1$ nếu $(i, j) \in \mathcal{K}$.

Công thức lặp:

$$dp_{i,j} = \begin{cases} 1 & \text{với } i = 1 \text{ hoặc } j = 1, (i, j) \notin \mathcal{K}, \\ 0 & \text{nếu } (i, j) \in \mathcal{K}, \\ (dp_{i-1,j} + dp_{i,j-1}) \bmod p & \text{với } (i, j) \notin \mathcal{K}. \end{cases}$$

Tổng số đường đi cần tìm là giá trị $dp_{n,m}$.

Chương trình I:

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

int n,m,k,p,x,y;
vector<vector<int>> dp;

int main ()
{
    fi>>n>>m>>k>>p;
    dp.resize(n+1,vector<int> (m+1,0));

    for(int i=0; i<k; ++i)
    {
        fi>>x>>y;
        dp[x][y]=-1;
    }
    dp[0][1]=1;
    for(int i=1; i<=n; ++i)
        for(int j=1; j<=m; ++j)
            if(dp[i][j]==-1) dp[i][j]=0;
            else dp[i][j]=(dp[i-1][j]+dp[i][j-1])%p;

    fo<<dp[n][m];
    fo<<"\nTime: "<<clock() / (double)1000<<" sec ";
}
```

Giai thuật II

Với n, m lớn hơn 10^3 , việc sử dụng mảng dp 2 chiều đòi hỏi bộ nhớ sử dụng lớn, vượt quá khả năng phục vụ của hệ thống lập trình.

Dòng thứ i của bảng dp được tính dựa trên dòng $i-1$. Vì vậy chỉ cần giữ 2 dòng của bảng và dùng kỹ thuật con lắc để từ dòng cũ tính dòng mới.

Trong trường hợp này cần lưu các ô chứa vật cản và sắp xếp thứ tự từ điển tăng dần của các tọa độ.

Độ phức tạp của giải thuật vẫn là $O(n \times m)$, nhưng bộ nhớ sử dụng chỉ thuộc bậc $O(m)$.

Chương trình II:

```
#include <bits/stdc++.h>
#define ff first
#define ss second
using namespace std;
ifstream fi ("route.inp");
ofstream fo ("route.out");
typedef pair<int,int> pii;
int n,m,k,p,x,y,u=1,v=0,ib=0;
int main()
{
    fi>>n>>m>>k>>p;
    vector<vector<int>> dp(2,vector<int> (m+1,0));
    vector<pii> ob(k);                                // Lưu các ô cấm
    for(int i=0; i<k; ++i)
    {
        fi>>x>>y;
        ob[i]={x,y};
    }
    sort(ob.begin(),ob.end());
    ob.push_back({n+1,m+1});                         // Phần tử hàng rào
    dp[0][1]=1;
    for(int i=1; i<=n; ++i)
    {
        u^=1; v^=1;                                // Tạo con lắc
        for(int j=1; j<=m; ++j)
            if(ob[ib]==make_pair(i,j)) dp[v][j]=0,++ib;
            else dp[v][j]=(dp[v][j-1]+dp[u][j])%p;
    }
    fo<<dp[v][m];
    fo<<"\nTime: "<<clock()/(double)1000<<" sec ";
}
```

Thời gian thực hiện chương trình lớn vì độ phức tạp của giải thuật là $O(n \times m)$.

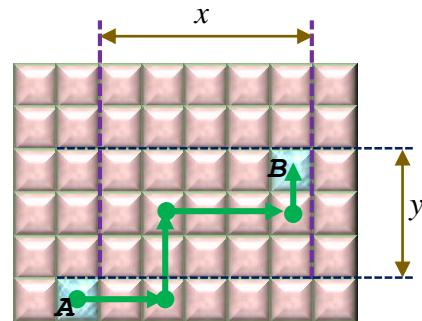
Giải thuật III

Áp dụng nguyên lý bù trừ có thể xây dựng giải thuật có *độ phức tạp chỉ phụ thuộc vào k* . Như vậy sẽ xây dựng cho phép làm việc với *bảng kích thước rất lớn* ($n, m \leq 10^9$). Tuy vậy dưới đây ta chỉ xét *chi tiết cách triển khai* giải thuật cho trường hợp $n, m \leq 10^6$.

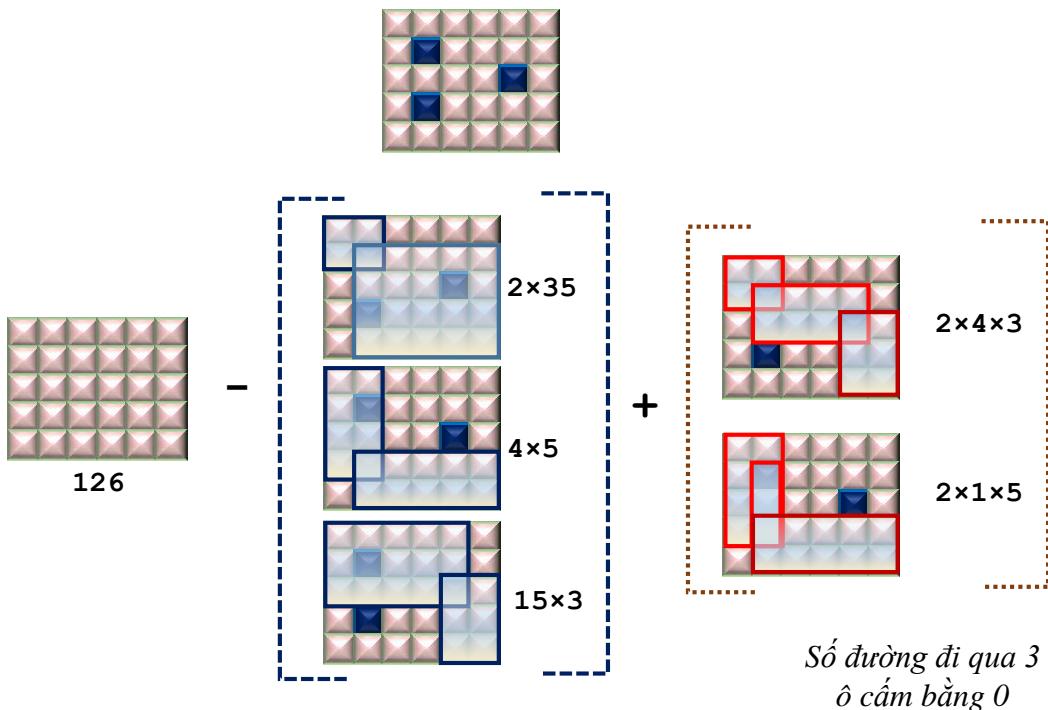
Mảng tọa độ ô cấm cần lưu dưới dạng sắp xếp theo thứ tự từ điển.

Xét số lượng đường đi từ **A** tới **B**, kể cả đi qua ô cấm (nếu có). Gọi số ô cần chuyển sang phải là **x** và sau đó phải chuyển lên trên **y** ô. Khi đó số lượng đường đi từ **A** tới **B** sẽ là C_{x+y}^x .

Ở ví dụ trong hình bên phải số lượng đường đi sẽ là $C_8^5 = C_8^3 = \frac{8 \times 7 \times 6}{1 \times 2 \times 3} = 56$.



Áp dụng đúng so đồ lý thuyết ta có nghiệm là tổng số đường đi qua mọi ô (kể cả ô cấm) trừ đi số đường đi qua từng ô cấm, cộng với số đường đi qua 2 ô cấm, trừ số đường đi qua 3 ô cấm, ...



Giải thuật có độ phức tạp $O(2^k \times k)$.

Giải thuật IV

Giải thuật III có *độ phức tạp hàm mũ* vì vậy chỉ có hiệu quả khi k đủ nhỏ. Để giảm độ phức tạp của giải thuật cần kết hợp giữa nguyên lý bù trừ và quy hoạch

động. Việc kết hợp 2 phương pháp nói trên sẽ tạo ra *giải thuật độ phức tạp đa thức đối với k* và vì vậy có thể áp dụng một cách có hiệu quả với k bậc 10^2 , không phụ thuộc vào n và m .

Để tiện xử lý, các dòng và cột được đánh số bắt đầu từ 0.

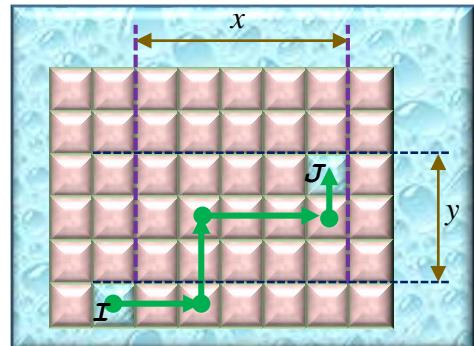
Xét \mathcal{K} – tập các ô có vật cản, ô xuất phát $(0, 0)$ và ô đích $(n-1, m-1)$. Tập sẽ có $k+2$ phần tử, đánh số từ 0 đến $k+1$.

Sắp xếp các phần tử của \mathcal{K} theo thứ tự tăng dần.

Gọi $dp_{i,j}$ – số lượng đường đi từ ô $i \in \mathcal{K}$ tới ô $j \in \mathcal{K}$, không chứa ô nào khác thuộc \mathcal{K} .

Dựa vào tọa độ của các điểm i và j , như đã nêu ở trên, ta có thể dễ dàng tính được $rt_{i,j}$ – tất cả các đường đi từ i tới j , kể cả các đường đi qua ô cấm.

$$rt_{i,j} = \begin{cases} 0 & \text{nếu không có cách đi,} \\ C_{x+y}^x & \text{trong trường hợp ngược lại.} \end{cases}$$



Gọi t – ô cấm nằm giữa i và j trong \mathcal{K} đã sắp xếp, $i < t < j$, có

$$dp_{i,j} = rt_{i,j} - \sum_{\forall t} dp_{i,t} \times rt_{t,j}$$

Nghiệm của bài toán là $dp_{0,k+1}$.

Độ phức tạp của giải thuật: $O(k^3)$.

Để giảm thời gian thực hiện cần tính sẵn bảng giá trị $i!$ với $i = 0, 1, 2, \dots$

Xét *giải thuật IV_a* tính trực tiếp giá trị thực của $dp_{i,j}$ với n, m đủ nhỏ, trên cơ sở đó – cải tiến để nhận được lời giải bài toán đã nêu.

Giải thuật IV_a

Tổ chức dữ liệu

- ─ Mảng `int64_t ft[20]` – lưu các giai thừa, $ft_i = i!$,
- ─ Mảng `int64_t dp[100][100]` – phục vụ sơ đồ quy hoạch động,
- ─ Mảng `vector<pii> ob` – lưu các phần tử thuộc tập \mathcal{K} .

Xử lý

Tính hệ số nhị thức:

```

int64_t comb(int u, int v)
{
    int q=u+v;
    u=min(u,v);
    return ft[q]/ft[u]/ft[q-u];
}

```

$$C_q^u = \frac{q!}{u! \times (q-u)!}$$

Tính $\text{rt}_{i,j}$ – tất cả các đường đi từ i tới j , kề các đường đi qua ô cấm:

```

int all_r(int i, int j)
{
    x=ob[j].ff-ob[i].ff;
    y=ob[j].ss-ob[i].ss;
    if(y<0) return 0;
    if(x==0 || y==0) if(i+1==j) return 1;
    return comb(x,y);
}

```

Tính $\text{rt}_{t,j}$ – tất cả các đường đi *từ t tới j* (kề các đường đi qua ô cấm):

```

int get_cb(int i, int j, int u)
{
    if((ob[u].ss>ob[j].ss) || (ob[u].ss<ob[i].ss))
        return 0;
    x=ob[j].ff-ob[u].ff;
    y=ob[j].ss-ob[u].ss;
    if(x==0 || y==0) return 1;
    return comb(x,y);
}

```

Tính $dp_{i,j}$:

Trình tự tính:

$dp_{0,1}, dp_{1,2}, dp_{2,3} \dots$

$dp_{0,2}, dp_{1,3}, dp_{2,4} \dots$

$dp_{0,3}, dp_{1,4}, dp_{2,5} \dots$

$\dots \dots \dots \dots \dots$

```
for(int t=1; t<=k; ++t)
    for(int i=0; i<=k-t; ++i)
    {
        int j=i+t;
        r=0;
        for(int it=i+1; it<j; ++it)
            r+=dp[i][it]*get_cb(i, j, it);
        dp[i][j]=all_r(i, j)-r;
    }
```

Các đường đi bị loại

Tổng hợp kết quả:

```
r=0;
for(int it=1; it<=k; ++it)
    r+=dp[0][it]*get_cb(0, k+1, it);

ans = comb(n-1, m-1)-r;
```

*Tính số đường đi
cần loại bỏ*

Chương trình IV_a:

```
#include <bits/stdc++.h>
#define ff first
#define ss second
using namespace std;
ifstream fi ("route.inp");
ofstream fo ("route.out");
typedef pair<int,int> pii;
int n,m,k,p,x,y,u=1,v=0,ib=0;
int64_t ft[20],dp[100][100];
vector<pii>ob;
int64_t ans,r;

int64_t comb(int u, int v)
{
    int q=u+v;
    u=min(u,v);
    return ft[q]/ft[u]/ft[q-u];
}

int get_cb(int i, int j, int u)
{
    if ((ob[u].ss > ob[j].ss) || (ob[u].ss < ob[i].ss)) return 0;
    x=ob[j].ff-ob[u].ff;
    y=ob[j].ss-ob[u].ss;
    if (x==0 || y==0) return 1;
    return comb(x,y);
}

int all_r(int i, int j)
{
    x=ob[j].ff-ob[i].ff;
    y=ob[j].ss-ob[i].ss;
    if (y<0) return 0;
    if (x==0 || y==0) if (i+1==j) return 1;
    return comb(x,y);
}

int main()
{
    fi>>n>>m>>k>>p;
    ob.resize(k);
    ft[0]=1;
    for(int i=1; i<19; ++i) ft[i]=ft[i-1]*i;
    for(int i=0; i<k; ++i)
    {
        fi>>x>>y; --x; --y;
        ob[i]={x,y};
    }
}
```

```

ob.push_back({0,0}); ob.push_back({n-1,m-1});
sort(ob.begin(), ob.end());

for(int t=1; t<=k; ++t)
    for(int i=0; i<=k-t; ++i)
    {
        int j=i+t;
        r=0;
        for(int it=i+1; it<j; ++it)
            r+=dp[i][it]*get_cb(i,j,it);
        dp[i][j]=all_r(i,j)-r;
    }
r=0;
for(int it=1; it<=k; ++it) r+=dp[0][it]*get_cb(0,k+1,it);
ans = comb(n-1,m-1)-r;
fo<<ans;
fo<<"\nTime: "<<clock() / (double)1000<<" sec ";
}

```

Giải thuật IV_b – Lời giải bài toán

Hệ số C_{x+y}^x của nhị thức Newton tăng rất nhanh: $C_{2n}^n \approx \frac{2^{2n}}{\sqrt{\pi n}}$

Giải thuật đòi hỏi phải tính nhiều C_{x+y}^x với các \mathbf{x}, \mathbf{y} khác nhau,

Để giảm độ phức tạp của giải thuật:

- + Lưu trữ bảng giá trị $\mathbf{q}! \pmod{\mathbf{p}}$ với $\mathbf{q} = 0, 1, 2, \dots, 2 \times 10^5$,
- + Tính C_{x+y}^x theo công thức $C_{x+y}^x = \frac{(x+y)!}{x! \times y!}$ và thay việc thực hiện phép chia bằng cách tính nghịch đảo theo mô đun.

Dựa vào bảng giá trị giao thừa đã lưu, có

$$(\mathbf{x}+\mathbf{y})! = \mathbf{a} \pmod{\mathbf{p}},$$

$$\mathbf{x}! = \mathbf{b} \pmod{\mathbf{p}},$$

$$\mathbf{y}! = \mathbf{c} \pmod{\mathbf{p}}.$$

$$\frac{(x+y)!}{x! \times y!} = \mathbf{z} \pmod{\mathbf{p}} \rightarrow \mathbf{a} = \mathbf{z} \times \mathbf{b} \times \mathbf{c} \rightarrow \mathbf{a} = \mathbf{z} \times \mathbf{r}, \text{ trong đó } \mathbf{r} = \mathbf{b} \times \mathbf{c}.$$

Lưu ý, ở đây *tất cả các số chia hết cho mấu số*.

$$\mathbf{a} = \mathbf{z} \times \mathbf{r} \rightarrow \mathbf{z} = \mathbf{a} \times \mathbf{r}^{p-2}$$

Bằng sơ đồ *tính nhanh lũy thừa* ta dễ dàng tìm được \mathbf{z} .

```

int64_t pw(int64_t u)
{
    int64_t res=1, tm=u, tp=p-2;
    while(tp)
    {
        if(tp&1) res=res*tm%p;
        tm=tm*tm%p;
        tp>>=1;
    }
    return res;
}

int64_t comb(int u, int v)
{
    int64_t a,b,c;
    int q=u+v;
    u=min(u,v);
    a=ft[q]; b= ft[u]*ft[q-u]%p;
    b=pw(b);
    return a*b%p;
}

```

Trong phần tính $\mathbf{dp}_{i,j}$ và dẫn xuất kết quả: lấy mô đun theo \mathbf{p} ở các đại lượng tính được.

Chương trình IV_b:

```
#include <bits/stdc++.h>
#define ff first
#define ss second
using namespace std;
ifstream fi ("route.inp");
ofstream fo ("route.out");
typedef pair<int,int> pii;
int n,m,k,p,x,y,u=1,v=0,ib=0;
int64_t ft[2000001],dp[102][102];
vector<pii>ob;
int64_t ans,r;

int64_t pw(int64_t u )
{
    int64_t res=1,tm=u, tp=p-2;
    while(tp)
    {
        if(tp&1) res=res*tm%p;
        tm=tm*tm%p;
        tp>>=1;
    }
    return res;
}

int64_t comb(int u, int v)
{
    int64_t a,b,c;
    int q=u+v;
    u=min(u,v);
    a=ft[q]; b= ft[u]*ft[q-u]%p;
    b=pw(b);
    return a*b%p;
}

int get_cb(int i, int j, int u)
{
    if((ob[u].ss > ob[j].ss) || (ob[u].ss < ob[i].ss)) return 0;
    x=ob[j].ff-ob[u].ff;
    y=ob[j].ss-ob[u].ss;
    if(x==0 || y==0) return 1;
    return comb(x,y);
}

int all_r(int i, int j)
{
    x=ob[j].ff-ob[i].ff;
    y=ob[j].ss-ob[i].ss;
    if(y<0) return 0;
    if(x==0 || y==0) if(i+1==j) return 1;
```

```

    return comb(x, y);
}

int main()
{
    fi>>n>>m>>k>>p;
    ob.resize(k);
    ft[0]=1;
    for(int i=1; i<=2000000; ++i) ft[i]=ft[i-1]*i%p;
    for(int i=0; i<k; ++i)
    {
        fi>>x>>y; --x; --y;
        ob[i]={x, y};
    }
    ob.push_back({0, 0}); ob.push_back({n-1, m-1});
    sort(ob.begin(), ob.end());

    for(int t=1; t<=k; ++t)
        for(int i=0; i<=k-t; ++i)
        {
            int j=i+t;
            r=0;
            for(int it=i+1; it<j; ++it)
                r=(r+dp[i][it]*get_cb(i, j, it))%p;
            dp[i][j]=(all_r(i, j)-r)%p;
        }
    r=0;
    for(int it=1; it<=k; ++it)
        r=(r+dp[0][it]*get_cb(0, k+1, it))%p;
    ans = (comb(n-1, m-1)-r+p)%p;
    fo<<ans;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec ";
}

```



2.7 Ghép cặp

Nhà trường quyết định xây dựng riêng cho mình một mạng xã hội để các bạn trẻ có điều kiện giao lưu một cách tốt nhất. Hệ thống sẽ tự động chọn và giới thiệu cho mỗi người những người bạn tiềm năng trong trường. Khi đăng ký, người tham gia sẽ phải trải qua thủ tục trắc nghiệm tâm lý. Kết quả trắc nghiệm cho biết giá trị tâm lý theo ba chỉ số, mỗi giá trị là một số nguyên dương.

Thực tế cuộc sống cho thấy, nếu 2 người có giá trị khác nhau ở cả 3 chỉ số thì họ sẽ thường xuyên rơi vào tranh luận, cãi nhau bất tận, còn nếu có giá trị ở 2 hay 3 chỉ số trùng nhau thì mối quan hệ, nếu có – sẽ rất buồn chán. Như vậy, để có một mối quan hệ có lợi và duy trì được lâu dài thì hai người phải có cùng giá trị ở một chỉ số nào đó, còn giá trị ở các chỉ số còn lại phải khác nhau.

Với n nhóm ba (a_i, b_i, c_i) hãy cho biết có bao nhiêu cặp $i < j$ mà số lượng đăng thức $a_i = a_j, b_i = b_j, c_i = c_j$ chỉ đúng có một.

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

- ➡ Dòng đầu tiên chứa số nguyên n ($1 \leq n \leq 10^5$),
- ➡ Dòng thứ i trong n dòng sau chứa 3 số nguyên a_i, b_i và c_i ($1 \leq a_i, b_i, c_i \leq 100$).

Kết quả: Đưa ra file văn bản ONLYONE.OUT một số nguyên – số lượng cặp tìm được.

Ví dụ:

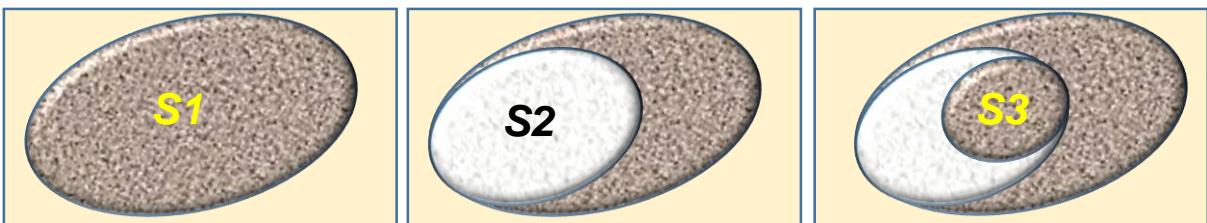
ONLYONE.INP	ONLYONE.OUT
4 100 100 100 100 100 100 100 99 99 99 99 100	5



Giải thuật: Tính lực lượng tập hợp, Nguyên lý bù – trừ.

Nguyên lý giải:

- ⊕ Tạm thời không xét một số điều kiện, tìm lực lượng của tập S1 rộng hơn, bao tập cần tìm S0 (*xử lý điều kiện cần*), thông thường chỉ giữ lại một điều kiện,
- ⊕ Xét tiếp một kiện mới loại bỏ khỏi S1 các phần tử không thuộc S0, tìm lực lượng tập $S_2 \subset S_1$ (thông thường – S2 chứa các phần tử thỏa mãn điều kiện đủ),
- ⊕ Nếu S2 không trùng với S0: Bổ sung thêm điều kiện tìm lực lượng tập S3 – mở rộng S2,
- ⊕ Hai bước cuối cùng được lặp lại nhiều lần cho đến khi nhận được S0.



Tổ chức dữ liệu:

- ☞ `int va[101]={0}, vb[101]={0}, vc[101]={0}: vai` lưu trữ số người có giá trị chỉ số thứ nhất bằng i , tương tự như vậy với v_{bi} và v_{ci} ,
- ☞ `int vab[101][101]={0}, vac[101][101]={0}, vbc[101][101]={0}: vabi,j` lưu trữ số người có giá trị chỉ số thứ nhất bằng i và giá trị chỉ số thứ 2 bằng j ,
- ☞ `int vabc[101][101][101]={0}: vabci,j,k` lưu trữ số người có giá trị chỉ số thứ nhất bằng i , giá trị chỉ số thứ 2 bằng j và giá trị chỉ số thứ 3 bằng k .

Xử lý:

- Tính tổng các cặp có giá trị một chỉ số giống nhau, không phụ thuộc vào các chỉ số còn lại, với $t = va_i$, tổng số các cặp có giá trị chỉ số thứ nhất bằng i sẽ là $t \times (t-1)/2$, tương tự như vậy với các chỉ số thứ 2 và thứ 3,
- Mỗi cặp giống nhau ở 2 chỉ số bị tính lặp 2 lần, vì vậy cần trừ 2 lần số lượng các cặp giống nhau ở 2 chỉ số,
- Mỗi cặp giống nhau ở cả 3 chỉ số bị tính lặp 3 lần, nhưng ở bước trên chúng bị trừ 6 lần (2 lần với `vab`, 2 lần với `vac` và 2 lần với `vbc`), vì vậy phải cộng thêm 3 lần số lượng các cặp giống nhau ở cả 3 chỉ số.

Độ phức tạp của giải thuật: O(n).

Chương trình:

```
#include <bits/stdc++.h>
#define NAME "onlyone."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
int n,a,b,c,va[101]={0},vb[101]={0},vc[101]={0},
    vab[101][101]={0},vac[101][101]={0},vbc[101][101]={0},
    vabc[101][101][101]={0};
int64_t ans=0,t;

int main()
{
    fi>>n;
    for(int i=1;i<=n;++i)
    {
        fi>>a>>b>>c;
        ++va[a]; ++vb[b]; ++vc[c];
        ++vab[a][b]; ++vac[a][c]; ++vbc[b][c];
        ++vabc[a][b][c];
    }

    for(int i=1;i<=100;++i)
    {
        t=va[i]; if(t>1)ans+=t*(t-1)/2;
        t=vb[i]; if(t>1)ans+=t*(t-1)/2;
        t=vc[i]; if(t>1)ans+=t*(t-1)/2;
    }

    for(int i=1;i<=100;++i)
        for(int j=1;j<=100;++j)
    {
        t=vab[i][j]; if(t>1)ans-=t*(t-1);
        t=vac[i][j]; if(t>1)ans-=t*(t-1);
        t=vbc[i][j]; if(t>1)ans-=t*(t-1);
    }

    for(int i=1;i<=100;++i)
        for(int j=1;j<=100;++j)
            for(int k=1;k<=100;++k)
    {
        t=vabc[i][j][k];
        if(t>1)ans+=3*t*(t-1)/2;
    }
    fo<<ans;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Hệ số nhị thức Newton

Số lượng cách chọn k phần tử trong số n phần tử, không phân biệt trình tự chọn là *tổ hợp chập k của n* và ký hiệu C_n^k .

Các số C_n^k là hệ số của khai triển nhị thức $(a+b)^n$:

$$(a+b)^n = C_n^0 a^n + C_n^1 a^{n-1} b + C_n^2 a^{n-2} b^2 + \dots + C_n^k a^{n-k} b^k + \dots + C_n^n b^n$$

Các tính chất và công thức tính C_n^k đã được nhà toán học Ấn độ *Pingala* nghiên cứu từ thế kỷ thứ 3 trước công nguyên. Vào thế kỷ 17 nhà toán học Pháp *Blaise Pascal* sắp xếp các số này dưới dạng hình tam giác và chỉ ra công thức lặp đơn giản dẫn xuất theo dòng, vì vậy C_n^k còn được gọi các số của tam giác Pascal.

Cuối thế kỷ 17 – đầu thế kỷ 18 *Isaac Newton* đã tổng quát hóa công thức triển khai nêu trên cho trường hợp n bất kỳ, không nhất thiết là nguyên và vì vậy nhị thức này thường được gọi là nhị thức Newton.

Công thức tính

Công thức giải tích:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Chứng minh:

Đầu tiên xét tập đã sắp xếp (theo một trình tự nào đó).

Ta có:

- n cách chọn phần tử thứ nhất,
- $n-1$ cách chọn phần tử thứ hai,
- $n-2$ cách chọn phần tử thứ ba,
-
- $n-k+1$ cách chọn phần tử thứ k .

Như vậy tổng số lượng cách chọn k phần tử trong số n phần tử là

$$n \times (n-1) \times (n-2) \times \dots \times (n-k+1)$$

Mỗi cách sắp xếp k phần tử tương ứng với một hoán vị các số từ 1 đến k , như vậy có $k!$ cách sắp xếp. Điều này có nghĩa, nếu không phân biệt trình tự chọn, ở công thức trên mỗi nhóm k phần tử bị tính lặp $k!$ lần.

Từ đây suy ra, nếu không tính trình tự thì số cách khác nhau chọn k trong số n phần tử sẽ là

$$C_n^k = \frac{n(n-1)(n-2)\dots(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$$

Công thức lặp:

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

$$C_n^k = \begin{cases} 1 \text{ với } k=0, \\ 0 \text{ với } k > n \end{cases}$$

Tính chất

Một số tính chất cần lưu ý của hệ số nhị thức Newton:

Đối xứng

$$C_n^k = C_n^{n-k}$$

Đưa ra ngoài ngoặc

$$C_n^k = \frac{n}{k} C_{n-1}^{k-1}$$

Tổng theo k

$$\sum_{k=0}^n C_n^k = 2^n$$

Tổng theo n

$$\sum_{m=0}^n C_m^k = C_{n+1}^{k+1}$$

Tổng theo n và k

$$\sum_{k=0}^m C_{n+k}^k = C_{n+m+1}^m$$

Tổng bình phương

$$(C_n^0)^2 + (C_n^1)^2 + \dots + (C_n^n)^2 = C_{2n}^n$$

Tổng có trọng số

$$1C_n^1 + 2C_n^2 + \dots + nC_n^n = n2^{n-1}$$

Mối quan hệ với số Fibonacci

$$C_n^0 + C_{n-1}^1 + \dots + C_{n-k}^k + \dots + C_0^n = F_{n+1}$$

Tính hệ số nhị thức theo mô đun p

Bài toán:

Cho các số nguyên n, k và p, $0 \leq k \leq n \leq 10^3$, $0 < p \leq 2 \times 10^9$.

Yêu cầu tính và đưa ra C_n^k theo mô đun p.

Giải thuật độ phức tạp $O(n^2)$:

Áp dụng sơ đồ lặp và cập nhật tại chỗ, chỉ cần dùng một mảng một chiều để lưu trữ các giá trị phục vụ dẫn xuất kết quả.

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("pascal.inp");
ofstream fo ("pascal.out");
int n, k, p;

int main()
{
    fi>>n>>k>>p;
    vector<int64_t> d(n+2, 0);
    d[0]=1;
    for(int i=1; i<=n; ++i)
        for(int j=i; j>0; --j) d[j] = (d[j-1]+d[j])%p;
    fo<<d[k];
}
```

Lưu ý: Có thể cài tiến sơ đồ tính để giáp kích thước mảng d xuống còn một nửa.

Giải thuật tính hệ số nhị thức với n bậc 10^6 và p – nguyên tố:

Hệ số C_n^k của nhị thức Newton tăng rất nhanh, $C_{2n}^n \approx \frac{2^{2n}}{\sqrt{\pi n}}$, với n bậc 10^2 trở lên cần nói chung cần mô phỏng xử lý số lớn. Một số hệ thống lập trình (cho Java, Python) có cài đặt công cụ xử lý số lớn, với một số bước chuẩn bị cần thiết có thể dễ dàng tính C_n^k theo mô đun p bất kỳ.

Với các hệ thống lập trình không trang bị công cụ xử lý số lớn, với p không phải là nguyên tố, sơ đồ tính toán khá phức tạp.

Dưới đây ta sẽ xét trường hợp p là nguyên tố và n có bậc 10^6 .

Chuẩn bị: Tính $\text{pr}_i = i! \pmod{p}$. với $i = 0, 1, 2, \dots, n$.

Với cặp giá trị (n, k) cụ thể ta có $\frac{pr_n}{pr_k \times pr_{n-k}} = x \pmod{p}$.

Ta có phương trình $ax = b \pmod{p}$, trong đó $a = pr_k \times pr_{n-k}$, $b = pr_n$.

Nhân cả 2 vế phương trình với a^{p-2} ta sẽ nhận được nghiệm cần tìm (tính nghịch đảo theo mô đun nguyên tố).

Đánh giá độ phức tạp:

Độ phức tạp của bước chuẩn bị: $O(n)$,

Độ phức tạp tính nghịch đảo theo mô đun nguyên tố: $O(\log p)$ theo sơ đồ tính nhanh lũy thừa.

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("pascal.inp");
ofstream fo ("pascal.out");
int n,k,p;
int64_t pr[1000001];
int64_t ans;

int64_t pw(int64_t u )
{
    int64_t res=1,tm=u, tp=p-2;
    while(tp)
    {
        if (tp&1) res=res*tm%p;
        tm=tm*tm%p;
        tp>>=1;
    }
    return res;
}

int64_t comb(int u, int v)
{
    int64_t a,b;
    b=pr[u]; a= pr[v]*pr[u-v]%p;
    a=pw(a);
    return a*b%p;
}
int main()
{
    fi>>n>>k>>p;
    pr[0]=1;
    for(int i=1; i<= 1000000; ++i) pr[i]=(pr[i-1]*i)%p;
    ans=comb(n, k);
    fo<<ans;
}
```

PHÂN RÃ BẬC CĂN N

Phân rã bậc căn n là phương pháp cho phép thực hiện các phép xử lý trên đoạn với độ phức tạp $O(\sqrt{n})$. Phân rã bậc căn n cũng có thể coi như một cấu trúc dữ liệu trừu tượng. Với các phép xử lý chuẩn như tìm tổng, tìm Min/Max cấu trúc Phân rã làm việc không hiệu quả bằng Cây phân khúc hoặc Treap nhưng nó là công cụ hỗ trợ cung cấp thông tin để giải các truy vấn có tính đặc thù. Phạm vi ứng dụng có hiệu quả của cấu trúc phân rã sẽ được xét trong phần ứng dụng.

Để hiểu cách tổ chức và hoạt động của *Cấu trúc phân rã căn n* ta sẽ xét bắt đầu từ các phép xử lý chuẩn.

Cấu trúc dữ liệu trên cơ sở phân rã căn n

Bài toán

Cho dãy số nguyên $\mathbf{A} = (\mathbf{a}_0, \mathbf{a}_1, \dots, \mathbf{a}_{n-1})$.

Hãy xây dựng cấu trúc dữ liệu cho phép xử lý \mathbf{q} truy vấn tính tổng các phần tử từ vị trí \mathbf{lf} đến \mathbf{rt} với độ phức tạp $O(\sqrt{n})$.

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

- + Dòng đầu tiên chứa số nguyên n và q ($1 \leq n \leq 10^5$, $1 \leq q \leq 10^4$),
- + Dòng thứ 2 chứa n số nguyên a_1, a_2, \dots, a_n ($|a_i| \leq 10^9$, $i = 1 \div n$),
- + Mỗi dòng trong q dòng sau chứa 2 số nguyên lf và rt ($1 \leq lf \leq rt \leq n$).

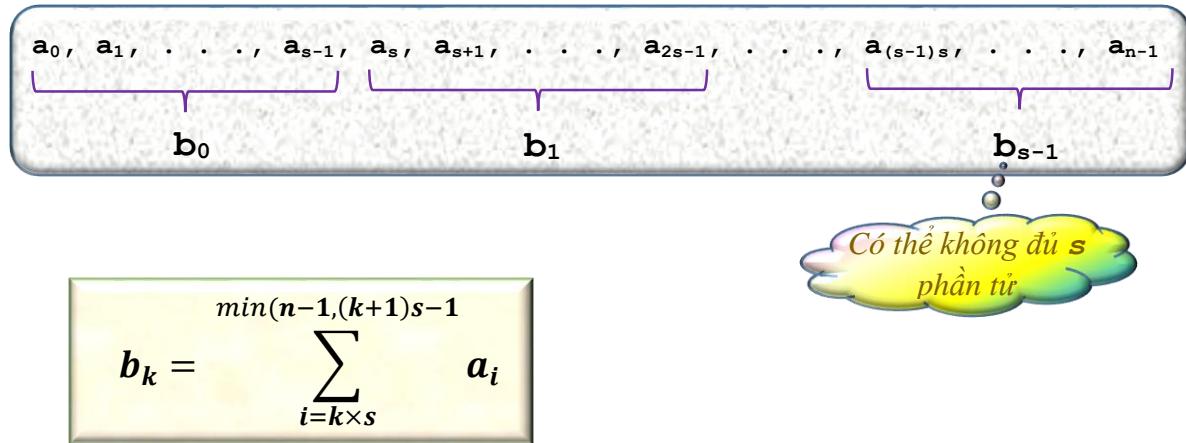
Kết quả: Đưa ra file văn bản DECOMP.OUT các tổng tìm được, mỗi số trên một dòng.

Ví dụ:

DECOMP.INP	DECOMP.OUT
15 4	42
2 5 4 7 9 6 3 2 5 10 8 4 1 3 6	59
7 15	16
2 11	53
6 9	
1 10	

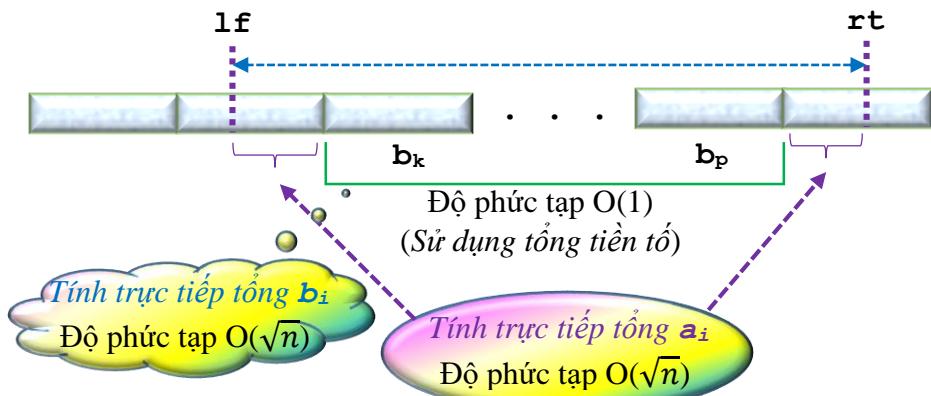
Xây dựng cấu trúc

Chia dãy số \mathbf{A} thành các khối, mỗi khối gồm s phần tử. Các cách chọn s khác nhau sẽ dẫn tới các cấu trúc và khả năng khác nhau. Ví dụ, để so sánh bằng hoặc khác s được chọn là các giá trị 2^k . Trong cấu trúc phân rã căn n , $s = \lceil \sqrt{n} \rceil$.



Ở đây \mathbf{b}_k được tính để phục vụ bài toán RSQ. Mảng \mathbf{b} được chuẩn bị với độ phức tạp $O(n)$.

Mỗi truy vấn tính tổng trong đoạn $[lf, rt]$ bao gồm 3 phần thực hiện:



Thay đổi cách tính \mathbf{b}_k (và cách xử lý 2 phần đầu, cuối đoạn) ta có thể đáp ứng các truy vấn tìm min, max, số phần tử 0, ...

Chương trình minh họa tính tổng

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

int n,s,q,lf,rt;
int64_t ans;

int main ()
{
    fi>>n>>q;
    vector<int> a(n);
    for(int &i:a) fi>>i;
    s = (int)sqrt(n+0.0)+1;
    vector<int64_t> b(s,0);
    for(int i=0; i<n; ++i) b[i/s]+=a[i];
    for(int i=0; i<q; ++i)
    {
        fi>>lf>>rt;
        --lf; --rt;
        ans=0;
        for(int i=lf; i<=rt;)
            if(i%s==0 && i+s-1<=rt) ans+=b[i/s], i+=s;
            else ans+=a[i++];
        fo<<ans<<'\
';
    }
    fo<<"\nTime: "<<clock () / (double)1000<<" sec";
    return 0;
}
```

Cập nhật

*Thay đổi giá trị một phần tử của **A**:* Việc thay đổi một giá trị a_i thành giá trị mới x được ghi nhận trực tiếp vào b_k tương ứng và vào **A** với chi phí O(1):

- + $b[i/s] += x - a[i];$
- + $a[i] = x;$

Lưu ý: Trường hợp cần quản lý không phải là tổng thì $b_{i/s}$ được cập nhật bằng cách loại bỏ phần có a_i tham gia, sau đó tính lại với sự tham gia của a_i mới.

Cách tiếp cận này cho phép ta tìm số lượng các phần tử 0, tìm phần tử đầu tiên khác 0, đếm số phần tử thỏa mãn một tính chất nào đó, . . .

Thay đổi các giá trị trong một đoạn:

Tăng mỗi phần tử dãy **A** từ vị trí **lf** đến vị trí **rt** một giá trị **x**:

Tạo mảng **c** kích thước **s** (như mảng **b**) ghi nhận cập nhật. Ban đầu, $c_j = 0$ với mọi j . Nếu k là khối nằm gọn trong đoạn **[lf, rt]** thì $c_k += x$.

Với các phần tử a_i thuộc các khối có chứa các phần tử không cập nhật (phần đầu và cuối của **[lf, rt]**) – với chi phí $O(\sqrt{n})$ cập nhật trực tiếp vào dãy **A**.

Thông qua **c**, việc dẩn xuất giá trị thực của a_i được thực hiện với chi phí O(1).

Kết hợp với việc cập nhật **b** theo cách tương tự ta có thể tìm giá trị cần quản lý theo các truy vấn trên đoạn.

Việc gán giá trị **x** cho các phần tử dãy **A** từ vị trí **lf** đến vị trí **rt** dễ dàng thực hiện theo sơ đồ xử lý tương tự.

Phân rã các truy vấn

Thông thường, các cấu trúc dữ liệu chỉ áp dụng với dữ liệu vào là đối tượng quản lý phục vụ dẩn xuất kết quả. Phương pháp và cấu trúc phân rã có thể *áp dụng với cả chính các truy vấn*.

Xét bài toán trong đó cho các dữ liệu cần xử lý, sau đó cho k truy vấn, một số truy vấn là lệnh thay đổi dữ liệu (*Truy vấn cập nhật*), một số khác là yêu cầu dẩn xuất kết quả theo một hàm nào đó với dữ liệu (*Truy vấn tìm kiếm*).

Việc xử lý truy vấn theo trình tự xuất hiện (Xử lý online) có thể có độ phức tạp rất cao do việc cập nhật đòi hỏi nhiều thời gian xử lý.

Trong trường hợp này các truy vấn có thể xử lý theo chế độ offline: đọc hết các truy vấn vào dòng xếp hàng chờ đợi và chỉ xử lý truy vấn cập nhật khi thực sự bắt đầu cần đến nó.

Ta sẽ xét một số bài toán cụ thể để hiểu rõ nguyên lý tiếp cận trên.

Tăng giá trị trên đoạn

Bài toán

Cho mảng số $\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n)$ và m truy vấn, mỗi truy vấn thuộc một trong 2 dạng:

- + Xác định giá trị \mathbf{a}_i ,
- + Tăng các phần tử của mảng ở các vị trí trong đoạn $[l_f, r_t]$ lên \mathbf{x} .

Việc xử lý phân rã không phải là sơ đồ tối ưu, ở đây ta xét chỉ để hiểu rõ bản chất nguyên lý tư tưởng phân rã đã nêu.

Phân các yêu cầu xử lý thành từng nhóm \sqrt{m} truy vấn.

Ban đầu không cần tổ chức dữ liệu hỗ trợ, làm việc trực tiếp với mảng \mathbf{A} . Xét các truy vấn ở khối đầu tiên. Nếu đó là truy vấn cập nhật – tạm thời bỏ qua. Gặp truy vấn tìm kiếm phần tử \mathbf{a}_i – gán \mathbf{a}_i cho biến lưu giá trị kết quả, duyệt các truy vấn cập nhật bỏ qua trước đó và thay đổi giá trị tìm được với các truy vấn cập nhật trùm lên vị trí i . Như vậy mỗi truy vấn tìm kiếm được thực hiện với độ phức tạp không quá $O(\sqrt{m})$.

Sau khi xử lý hết các truy vấn tìm kiếm ở mỗi khối cần tiến hành cập nhật mảng A với chi phí $O(n)$ dựa trên các yêu cầu cập nhật bị tạm gác ở khối truy vấn vừa xét. Để làm được việc này cần tạo mảng hỗ trợ \mathbf{c} kích thước n . Với mỗi truy vấn cập nhật (l_f, r_t, \mathbf{x}) cho $c_{l_f} += \mathbf{x}$ và $c_{r_t+1} -= \mathbf{x}$, tiến hành tính tổng tích lũy tiền tố với \mathbf{c} ($c_i += c_{i-1}$, $i = 2 \div n$). Các phần tử của \mathbf{A} được cập nhật theo công thức $a_i += c_i$ với mọi i .

Như vậy, độ phức tạp của giải thuật sẽ là $O(\sqrt{m}(n + m))$.

Bổ sung và loại bỏ cạnh của đồ thị

Bài toán

Cho đồ thị vô hướng n đỉnh, m cạnh và các truy vấn thuộc một trong 3 loại:

- + Bổ sung thêm cạnh $(\mathbf{x}_i, \mathbf{y}_i)$,
- + Loại bỏ cạnh $(\mathbf{x}_i, \mathbf{y}_i)$,
- + Kiểm tra tồn tại đường đi từ \mathbf{x}_i tới \mathbf{y}_i .

Nếu không có yêu cầu xóa cạnh thì bài toán kiểm tra liên thông dễ dàng giải quyết dựa trên cấu trúc dữ liệu DSU (*Disjoin Set Union*).

Các truy vấn cũng sẽ được chia thành nhóm. Với mỗi nhóm, đầu tiên *thực hiện tất cả các truy vấn loại bỏ cạnh*, sau đó xây dựng hệ thống tập không giao nhau cho các đỉnh (DSU) với đồ thị nhận được.

Với mỗi nhóm truy vấn bài toán dẫn về việc chỉ thêm cạnh, điều mà cấu trúc DSU dễ dàng giải quyết. Số lượng cạnh cần bổ sung (từ các truy vấn thêm/bớt cạnh) là không quá kích thước mỗi khối, vì vậy độ phức tạp sẽ là $O(\sqrt{m})$ vì chỉ cần xét \sqrt{m} cạnh bằng cách loang theo chiều rộng.

Sơ đồ tổng quát xử lý truy vấn ở chế độ offline trên cơ sở phân rã

Xét lớp bài toán, trong đó không có truy vấn cập nhật dữ liệu, các truy vấn được biết trước, mỗi truy vấn trên đoạn $[lf, rt]$ đòi hỏi dẫn xuất một đại lượng nào đó trong đoạn này từ mảng số $\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n)$ và nếu biết lời giải trên đoạn $[lf, rt]$ thì có thể dễ dàng xác định lời giải cho các đoạn $[lf+1, rt]$, $[lf-1, rt]$, $[lf, rt-1]$ và $[lf, rt+1]$.

Các truy vấn đã biết trước, vì vậy có thể xử lý kiểu offline. Các cặp thông tin về truy vấn (lf, rt) được chia thành các khối, mỗi khối chứa s phần tử. Các truy vấn được sắp xếp theo trình tự tăng dần của $(lf/s, rt)$. Như vậy, trong mỗi khối các truy vấn được sắp xếp tăng dần theo rt .

Xuất phát từ đoạn rỗng $[lf, lf-1]$, mở rộng dần các biên ta sẽ tìm được lời giải cho các truy vấn trong khối.

Với một số bài toán, khó có cách giải hiệu quả hơn sơ đồ xử lý đã nêu, ví dụ như việc xác định số lượng số khác nhau trên đoạn $[lf, rt]$ của dãy số \mathbf{A} .

Giải thuật Mo (Mo's Algorithm)

Giải thuật Mo dùng để xử lý các truy vấn trên đoạn $[lf, rt]$ của dãy số \mathbf{A} ở chế độ offline, trong đó không có các truy vấn thay đổi giá trị của \mathbf{A} với độ phức tạp $O(q \log q + (n+q) \sqrt{n})$, trong đó q – số lượng truy vấn, n – kích thước mảng \mathbf{A} .

Gọi $[\mathbf{a} \dots \mathbf{b}]$ là đoạn trung gian, lưu các phần tử liên tiếp của \mathbf{A} từ vị trí \mathbf{a} đến vị trí \mathbf{b} .

Đoạn trung gian lưu trữ dưới dạng cho phép thực hiện các xử lý:

- ❖ **addLeft(a-1), addRight(r+1)** – bổ sung một phần tử vào mảng trung gian từ bên trái hoặc bên phải,
- ❖ **delLeft(a), delRight(r)** – loại bỏ một phần tử phía trái hoặc phía phải của mảng trung gian,
- ❖ **answer()** – dẫn xuất kết quả theo tiêu chuẩn cần tìm từ mảng trung gian.

Ban đầu mảng trung gian rỗng: $\mathbf{a}=1, \mathbf{b}=0$.

Các truy vấn được lưu dưới dạng cặp giá trị (\mathbf{lf}, \mathbf{rt}) và được sắp xếp theo cách sê trình bày trong các phần tiếp theo.

Lần lượt xử lý các truy vấn theo trình tự lưu trữ.

Giả thiết mảng trung gian hiện tại là $[\mathbf{a} \dots \mathbf{b}]$, truy vấn tiếp theo chưa xử lý là $[\mathbf{lf}_i, \mathbf{rt}_i]$.

Các thao tác tiếp theo sẽ là:

- + Nếu $\mathbf{a} > \mathbf{lf}_i$ – bổ sung một phần tử vào trái mảng trung gian cho đến khi biên trái trùng nhau,
- + Nếu $\mathbf{a} < \mathbf{lf}_i$ – loại bỏ một phần tử ở phía trái mảng trung gian cho đến khi biên trái trùng nhau,
- + Nếu $\mathbf{a} = \mathbf{lf}_i$ – không cần xử lý biên trái.
- + Xử lý tương tự với biên phải của mảng trung gian để có $\mathbf{b} = \mathbf{rt}_i$.

Nói một cách ngắn gọn, ta mở rộng $[\mathbf{a} \dots \mathbf{b}]$ đến $[\mathbf{u} \dots \mathbf{v}]$, trong đó

$$\mathbf{u} = \min\{\mathbf{a}, \mathbf{lf}_i\}, \quad \mathbf{v} = \max\{\mathbf{b}, \mathbf{rt}_i\}$$

sau đó xóa các phần tử thừa để nhận được mảng $[\mathbf{lf}_i, \mathbf{rt}_i]$ và từ đó – dẫn xuất kết quả cần tìm.

Cách phân nhóm và sắp xếp truy vấn:

Gọi \mathbf{k} là kích thước nhóm, những truy vấn có \mathbf{lf}_i thỏa mãn điều kiện $1 \leq \mathbf{lf}_i \leq \mathbf{k}$ đưa vào nhóm thứ nhất, các truy vấn có \mathbf{lf}_i thỏa mãn điều kiện $\mathbf{k+1} \leq \mathbf{lf}_i \leq 2\mathbf{k}$ – vào nhóm thứ 2, ... Các khối truy vấn xử lý độc lập với nhau. Có thể sắp xếp truy vấn trong mỗi nhóm theo thứ tự tung dần của rti. Khi đó độ phức tạp xử lý nhóm thứ i sẽ là $O(n + q_i * k)$, trong đó q_i – số truy vấn trong nhóm i .

Có thể chứng minh được rằng, để xử lý các truy vấn thuộc một khối không cần quá $3 * n + q_i * k$ phép **add** và **del**:

Ban đầu, mảng trung gian là $[\mathbf{a} \dots \mathbf{b}]$,

Để xử lý truy vấn đầu tiên có thể cần đến $2 * n$ phép **add** và **del**,

Không phải sử dụng **delRight** lần nào khi xử lý các truy vấn còn lại trong khối vì biên phải không giảm,

Phép **addRight** được sử dụng không quá n lần vì giá trị nhỏ nhất của biên phải là 1 và lớn nhất là n ,

Với 2 phép thao tác còn lại: xét 2 truy vấn liên tiếp $[\mathbf{lf}_i \dots \mathbf{rt}_i]$ và $[\mathbf{lf}_j \dots \mathbf{rt}_j]$. Hai truy vấn thuộc cùng một khối nên $|\mathbf{lf}_i - \mathbf{lf}_j| < \mathbf{k}$, như vậy số lượng

thao tác **addLeft** và **delLeft** không vượt quá **K**, và tính toàn bộ khối – không vượt quá $q_i * k$.

Tổng thời gian xử lý tất cả các truy vấn sẽ là $O(\frac{n^2}{K} + K \times q)$.

Nếu chọn $K = \sqrt{n}$ và có sắp xếp truy vấn trong khối theo giá trị tăng dần của **rt** ta có ta có độ phức tạp giải thuật là $O(q * \log q + (n+q) * \sqrt{n})$.

Ví dụ

Bài toán Trọng số

Cho mảng số nguyên a_1, a_2, \dots, a_n . Xét dãy con $a_{lf}, a_{lf+1}, \dots, a_{rt}$, trong đó $1 \leq lf \leq rt \leq n$. Gọi k_s là số lần xuất hiện của s trong dãy con. Trọng số của dãy con là tổng các $k_s \times k_s \times s$ với các s khác nhau thuộc dãy con.

Cho m dãy con, mỗi dãy con được xác định bởi cặp giá trị **lf** và **rt**.

Với mỗi dãy con hãy xác định và đưa ra trọng số của nó.

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

- ⊕ Dòng đầu tiên chứa số nguyên n và m ($1 \leq n, m \leq 2 \times 10^5$),
- ⊕ Dòng thứ 2 chứa n số nguyên a_1, a_2, \dots, a_n ($1 \leq a_i \leq 10^6$, $i = 1 \div n$),
- ⊕ Mỗi dòng trong m dòng sau chứa 2 số nguyên **lf** và **rt** ($1 \leq lf \leq rt \leq n$).

Kết quả: Đưa ra file văn bản MO_ALG.OUT m số nguyên, mỗi số trên một dòng xác định các trọng số tìm được.

Ví dụ:

MO_ALG.INP	MO_ALG.OUT
10 3	60
1 2 3 2 1 1 3 3 1 9	12
1 10	33
2 5	
3 8	



Tổ chức dữ liệu:

- Mảng **int** $a[N]$ – lưu dãy số a_1, a_2, \dots, a_n ,
- Mảng $t3i$ $q[M]$ – lưu các truy vấn, mỗi phần tử là một nhóm 3 **tuple<int, int, int>** ghi nhận (**lf, rt, i**)
- Mảng **int** $answer[M]$ – lưu kết quả (các trọng số),
- Cấu trúc **s** – lưu thông tin về đoạn trung gian, kết gắn với các phép xử lý điều chỉnh biên và dẫn xuất kết quả.

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("Mo_Alg.inp");
ofstream fo ("Mo_Alg.out");
typedef tuple<int,int,int> t3i;
const int N=100000, M = 100000, MAX=50000;
int a[N];
t3i q[M];

int main()
{
    int n,m,lf,rt,k;
    fi>>n>>m;
    for(int i=0;i<n;++i) fi>>a[i];
    for(int i=0;i<m;++i)
    {
        fi>>lf>>rt;
        --lf; --rt;
        q[i]=make_tuple(lf,rt,i);
    }
    sort(q,q+m);
    int answer[M];
    struct state
    {
        int l,r;
        int64_t sum;
        int c[MAX];
        state():l(0),r(-1), sum(0) { fill(c,c+MAX,0); }
        void change(int v, int d)
        {
            sum-= (int64_t)c[v]*c[v]*v;
            c[v]+=d;
            sum+= (int64_t)c[v]*c[v]*v;
        }
        void addLeft() { change(a[--l],1); }
        void addRight() { change(a[++r],1); }
        void delLeft() { change(a[l++],-1); }
        void delRight() { change(a[r--],-1); }
        int answer() const { return sum; }
    };
    state s;
    for(int i=0;i<m;i++)
    {
        if(q[i].first==0)
        {
            s.addLeft();
            answer[i]=s.sum;
        }
        else if(q[i].first==q[i].second)
        {
            s.delRight();
            answer[i]=s.sum;
        }
        else
        {
            s.change(q[i].second,-1);
            s.change(q[i].first,1);
            answer[i]=s.sum;
        }
    }
    fo<<answer[0];
}
```

```

} ;

state s;
for(int i=0; i<m; ++i)
{
    tie(lf, rt, k) = q[i];
    while(s.l > lf) s.addLeft();
    while(s.r < rt) s.addRight();
    while(s.l < lf) s.delLeft();
    while(s.r > rt) s.delRight();
    answer[k] = s.answer();
}
for(int i=0; i<m; ++i) fo << answer[i] << '\n';
fo << "\nTime: " << clock() / (double) 1000 << " sec";
}

```



GIẢI THUẬT KRUSKAL

Cho đồ thị G vô hướng liên thông có trọng số. Yêu cầu tìm cây con của đồ thị thỏa mãn các tính chất:

- ✚ Nối tất cả các đỉnh của đồ thị đã cho,
- ✚ Có tổng trọng số (trọng số của cây con) là nhỏ nhất.

Cây con có tính chất trên được gọi là cây khung cực tiểu của G .

Một trong số các giải thuật hiệu quả tìm cây khung được Kruskal đề xuất năm 1956.

Một số tính chất của cây khung cực tiểu

Cây khung cực tiểu là duy nhất nếu trọng số các cạnh của đồ thị khác nhau từng đôi một, trong trường hợp ngược lại, có thể tồn tại một số cây khung cực tiểu khác nhau,

Cây khung cực tiểu đồng thời cũng là cây khung có tích các trọng số nhỏ nhất,

Cây khung cực tiểu là cây khung với cạnh trọng số lớn nhất là nhỏ nhất,

Việc tìm cây khung cực đại có thể thực hiện tương tự như tìm cây khung cực tiểu, hơn thế nữa, có thể áp dụng trực tiếp giải thuật tìm cây khung cực tiểu với trọng số các cạnh được đổi dấu.

Nguyên lý giải thuật Kruskal

- ✿ Ban đầu xây dựng đồ thị gồm các đỉnh của G và không chứa cạnh nào, gọi n là số đỉnh của G , như vậy ban đầu có n thành phần liên thông,
- ✿ Sắp xếp các cạnh của G theo thứ tự tăng dần của trọng số,
- ✿ Mỗi bước tiếp theo – liên kết 2 thành phần liên thông của đồ thị đang xây dựng:
 - Duyệt các cạnh theo trình tự đã sắp xếp,
 - Tìm cạnh trọng số nhỏ nhất chưa được chọn nối 2 thành phần liên thông của đồ thị đang xây dựng,
 - Kết nạp cạnh tìm được vào đồ thị cần tìm,
- ✿ Quá trình tìm và kết nạp cạnh mới được thực hiện cho đến khi trong đồ thị đang xây dựng chỉ có một thành phần liên thông.

Ví dụ

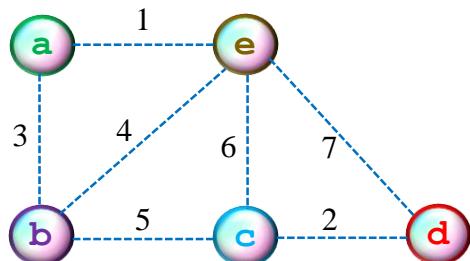
Xét đồ thị G với $n = 5$ và $m = 7$.

Để thuận tiện trình bày các đỉnh được đánh dấu bằng các ký tự a, b, c, d, e .

Trọng số cạnh (Đã sắp xếp tăng dần)

Cạnh	ae	cd	ab	be	bc	ce	de
Trọng số	1	2	3	4	5	6	7

Định đại diện của các miền liên thông



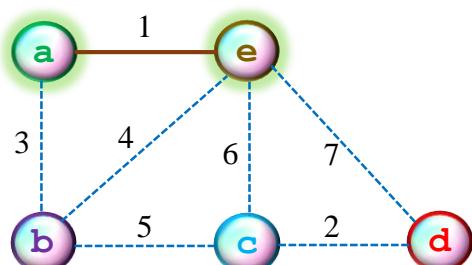
Vector $P \ a \ b \ c \ d \ e$

Cạnh có trọng số nhỏ nhất: ae ,

Các đỉnh a và e thuộc 2 miền liên thông,



Ghi nhận vào cây khung.



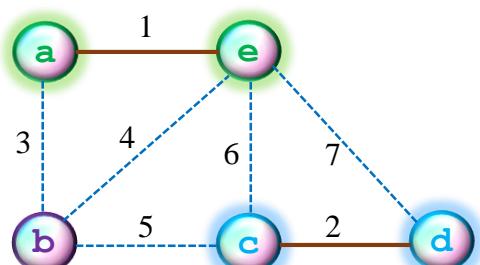
Vector $P \ a \ b \ c \ d \ a$

Cạnh có trọng số nhỏ nhất: cd ,

Các đỉnh c và d thuộc 2 miền liên thông,



Ghi nhận vào cây khung.



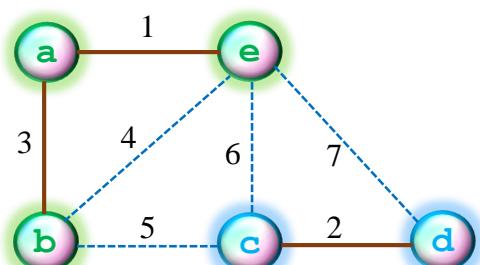
Vector $P \ a \ b \ c \ d \ a$

Cạnh có trọng số nhỏ nhất: ab ,

Các đỉnh a và b thuộc 2 miền liên thông,



Ghi nhận vào cây khung.



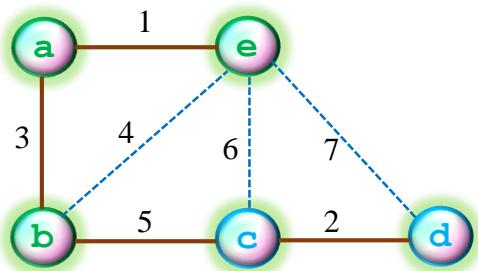
Vector $P \ a \ b \ c \ d \ a$

Cạnh có trọng số nhỏ nhất: be ,

Các đỉnh b và e cùng miền liên thông,



Bỏ qua.



Vector $P \ a \ b \ c \ d \ a$

Cạnh có trọng số nhỏ nhất: \mathbf{bc} ,

Các đỉnh \mathbf{b} và \mathbf{c} khác miền liên thông,

Ghi nhận vào cây khung

Kết thúc xử lý vì đã có đủ $n-1$ cạnh.

Giải thuật đơn giản

Tổ chức dữ liệu:

- Mảng **vector** `<t3i > g(m)` – lưu thông tin về các cạnh của đồ thị G,
- Mảng **vector** `< pair<int, int> > res` – lưu cạnh của cây khung,
- Mảng **vector<int>** `tree_id(n)` – ghi nhận đỉnh đại diện của miền lên thông.

Độ phức tạp của giải thuật: $O(m \log n + n^2)$.

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("kraskal_dsu.inp");
ofstream fo ("kraskal_V1.out");
typedef tuple<int,int,int> t3i;
int n,m,l,a,b,cost;

int main ()
{
    fi>>n>>m;
    vector <t3i > g(m);
    vector < pair<int,int> > res;
    for(int i=0; i<m; ++i)
    {
        fi>>l>>a>>b; --a; --b;
        g[i]=make_tuple(l,a,b);
    }
    sort (g.begin(), g.end());
    vector<int> tree_id (n);
```

```

for (int i=0; i<n; ++i) tree_id[i] = i;
for (auto i:g)
{
    tie(l,a,b) = i;
    if (tree_id[a] != tree_id[b])
    {
        cost += 1;
        res.push_back ({a, b});
        int old_id = tree_id[b], new_id = tree_id[a];
        for (int j=0; j<n; ++j)
            if (tree_id[j] == old_id)
                tree_id[j] = new_id;
    }
}
fo<<cost<<'\n';
for (auto i:res) fo<<i.first+1<< ' '<<i.second+1<<'\n';
}

```

Giải thuật cải tiến

Khâu kiểm tra 2 đỉnh của một cạnh thuộc một hay 2 miền liên thông khác nhau trong đồ thị đang xây dựng là điểm yếu của sơ đồ xử lý đơn giản.

Để nâng cao hiệu quả của giải thuật: dùng phương pháp DSU kết nối các miền liên thông và xác định đỉnh đại diện cho miền liên thông.

Tổ chức dữ liệu: Tương tự như ở giải thuật trên.

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

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("kruskal_dsu.inp");
ofstream fo ("kruskal_dsu.out");
typedef tuple<int,int,int> t3i;
vector<int> p;

int dsu_get (int v)
{
    return (v == p[v]) ? v : (p[v] = dsu_get (p[v]));
}

void dsu_unite (int a, int b)
{
    a = dsu_get (a);
    b = dsu_get (b);
    if (rand() & 1) swap (a, b);
    if (a != b) p[a] = b;
}

int main ()
{
    int n,m,w,x,y;
    fi>>n>>m;
    vector<t3i>g (m);
    for (int i=0; i<m; ++i)
    {
        fi>>w>>x>>y; --x; --y;
        g[i]=make_tuple (w,x,y);
    }
    int cost = 0;
    vector < pair<int,int> > res;

    sort (g.begin(), g.end());
    p.resize (n);
    for (int i=0; i<n; ++i) p[i] = i;
    for (int i=0; i<m; ++i)
```

```

{
    int a,b,l;
    tie(l,a,b)=g[i];
    if (dsu_get(a) != dsu_get(b))
    {
        cost += l;
        res.emplace_back(make_pair(a,b));
        dsu_unite (a, b);
    }
}
fo<<cost<<'\\n';
for(auto i:res) fo<<i.first<<' ' <<i.second+1<<'\\n';
}

```



GIẢI THUẬT RABIN – KARP

Giải thuật Rabin – Karp (1987) phục vụ tìm số lần xuất hiện xâu **s** trong xâu **t** với độ phức tạp $O(|s|+|t|)$, trong đó $|s|, |t|$ - độ dài các xâu **s** và **t**.

Hai lần xuất hiện của **s** gọi là khác nhau nếu nó bắt đầu từ các vị trí khác nhau trong xâu **t**.

Để đơn giản trong trình bày, ta xét trường hợp các xâu chỉ chứa ký tự la tinh thường, tức là có không quá 26 ký tự khác nhau.

Ký hiệu:

- ✚ **ls** và **lt** – độ dài tương ứng của các xâu **s** và **t**,
- ✚ **t[i...j]** – xâu con các ký tự liên tiếp nhau của **t** từ ký tự thứ **i** đến ký tự thứ **j**.

Để **s** có phải là xâu con của **t** bắt đầu từ vị trí **i** hay không cần kiểm tra điều kiện **t[i...i+ls-1]==s**.

Việc kiểm tra có thể thực hiện với độ phức tạp $O(1)$ nếu thay vì so sánh xâu ta so sánh các giá trị băm tương ứng.

Giá trị băm của **t[i...i+ls-1]** có thể tính với chi phí $O(1)$ nếu chuẩn bị trước mảng giá trị băm của các tiền tố xâu **t**. Phụ thuộc vào cách chuẩn bị giá trị băm của tiền tố, việc tính các giá trị để so sánh có thay đổi đôi chút, nhưng độ phức tạp vẫn là $O(1)$.

Vì các xâu chỉ chứa ký tự la tinh thường, chỉ cần chọn cơ sở tính băm **p=31** là đủ.

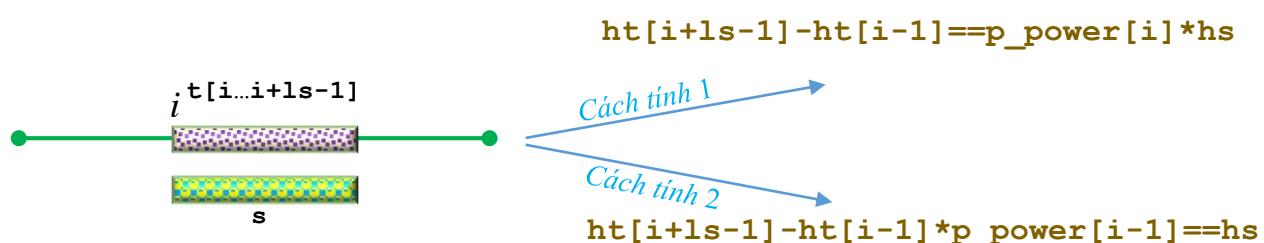
Gọi **ht_i** là giá trị băm của **t[0...i]**, **ht₀** = **t₀**-97, **p_power_i** = **pⁱ**.

Có thể tính **ht_i** theo một trong 2 cách:

$$ht[i] = ht[i-1] + (t[i]-97) * p_power[i];$$

hoặc

$$ht[i] = ht[i-1] * p + t[i] - 97;$$



Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("Rabin.inp");
ofstream fo ("Rabin.out");
string s,t;
int ls,lt,lx;

int main ()
{
    fi>>s>>t;
    ls=s.size(); lt=t.size();
    lx=max (ls,lt);
    const int p = 31;
    vector<int64_t> p_pow (lx);
    p_pow[0] = 1;
    for (int i=1; i<lx; ++i)
        p_pow[i] = p_pow[i-1] * p;
    vector<int64_t> h(lt);
    for (int i=0; i<lt; i++)
    {
        h[i] = (t[i] - 'a' + 1) * p_pow[i];
        if (i) h[i] += h[i-1];
    }
    int64_t h_s = 0;
    for (int i=0; i<ls; ++i)
        h_s += (s[i] - 'a' + 1) * p_pow[i];
    vector<int> res;
    for (int i = 0; i + ls - 1 < lt; ++i)
    {
        int64_t cur_h = h[i+ls-1];
        if (i) cur_h -= h[i-1];
        if (cur_h == h_s * p_pow[i]) res.push_back(i);
    }
    fo<<res.size()<<'\n';
    for(int i:res) fo<<i<<' ';
}
```



SỐ LƯỢNG CÁCH LIÊN THÔNG HÓA ĐỒ THỊ

Xét đồ thị G có ít nhất 2 đỉnh. Nếu đồ thị không liên thông, cần bổ sung thêm một số cạnh ít nhất để có đồ thị liên thông. Số lượng cách bổ sung khác nhau liên quan tới các vấn đề cần giải quyết trong nhiều bài toán đồ thị.

Để xác định cách bổ sung cạnh cần phải có cách biểu diễn đồ thị đã cho một cách đơn trị dưới dạng dãy số.

Heinz Prüfer (1918) đã đề xuất một cách ánh xạ đồ thị sang dãy số, ánh xạ này được gọi là mã Prüfer.

Mã Prüfer

Mã Prüfer là ánh xạ đơn trị hai chiều một *cây n đỉnh* sang *dãy n-2 số nguyên*, mỗi số trong khoảng $[1, n]$.

Mỗi đồ thị liên thông sẽ có một hoặc một số cây khung. Mã Prüfer là một song ánh cây khung với dãy số nguyên.

Do tính đặc thù của dãy số, Mã Prüfer ít khi được dùng để lưu trữ và xử lý đồ thị, nhưng nó được sử dụng trong nhiều bài toán tổ hợp.

Xây dựng mã Prüfer

Việc xác định mã Prüfer bao gồm $n-2$ bước, ở mỗi bước: chọn lá có số nhỏ nhất, xóa nó khỏi cây và bổ sung nút cha của lá đó vào mã cần tìm.

Khi cây chỉ còn 2 nút – giải thuật kết thúc. Bản thân 2 nút này không được ghi nhận (một cách tường minh) vào mã cần tìm.

Với các cấu trúc dữ liệu hỗ trợ tìm min, việc xây dựng mã Prüfer có thể được thực hiện với độ phức tạp $O(n \log n)$.

Tổ chức dữ liệu:

- ─ Mảng `vector<int>` `g` [`MAXN`] – lưu cạnh của cây,
- ─ Mảng `int` `degree` [`MAXN`] – lưu bậc các nút của cây,
- ─ Mảng `bool` `killed` [`MAXN`] – đánh dấu nút bị xóa,
- ─ Tập `set<int>` `leaves` – lưu đỉnh lá,
- ─ Mảng `vector<int>` `ans` (`n-2`) – lưu kết quả.

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("prufer.inp");
ofstream fo ("prufer.out");
const int MAXN = 100001;
int n,a,b;
vector<int> g[MAXN];
int degree[MAXN];
bool killed[MAXN];
vector<int> prufer_code()
{
    set<int> leaves;
    for (int i=0; i<n; ++i)
    {
        degree[i] = (int) g[i].size();
        if (degree[i] == 1)
            leaves.insert (i);
        killed[i] = false;
    }
    vector<int> result (n-2);
    for (int iter=0; iter<n-2; ++iter)
    {
        int leaf = *leaves.begin();
        leaves.erase (leaves.begin());
        killed[leaf] = true;
        int v;
        for (size_t i=0; i<g[leaf].size(); ++i)
            if (!killed[g[leaf][i]])
                v = g[leaf][i];
        result[iter] = v;
        if (--degree[v] == 1)
            leaves.insert (v);
    }
    return result;
}

int main()
{
    fi>>n;
    for(int i=0; i<n-1; ++i)
    {
        fi>>a>>b;
        --a; --b;
        g[a].push_back(b); g[b].push_back(a);
    }
    vector<int> ans (n-2);
    ans = prufer_code();
    for(int i:ans) fo<<i+1<<' ';
}
```

Xây dựng mã Prüfer với độ phức tạp tuyến tính

Để giảm thời gian xử lý ta sử dụng con trỏ **ptr** chỉ đến các đỉnh của cây với giá trị tăng dần.

Thoạt nhìn, ta thấy khó hiểu về vai trò của con trỏ này vì số của nút lá có thể tăng hoặc giảm trong quá trình xử lý. Nhưng dễ dàng nhận thấy rằng số của nút lá bị giảm chỉ trong một trường hợp duy nhất: khi xóa một nút lá, số của nút cha của lá đó nhỏ hơn số bị xóa, nhưng khi đó nó sẽ bị xóa ngay ở bước tiếp theo trong quá trình xử lý!

Dựa vào tính chất này ta có thể cải tiến giải thuật để có độ phức tạp $O(n)$.

Tổ chức dữ liệu:

- Mảng **vector<int>** **g** [MAXN] – lưu cạnh của cây,
- Mảng **int** **degree** [MAXN] – lưu bậc các nút của cây,
- Mảng **int** **parent** [MAXN] – lưu số của nút cha,
- Mảng **vector<int>** **ans** (n-2) – lưu kết quả.

Xử lý:

Với mỗi đỉnh **i** – tìm đỉnh cha của nó,

Đỉnh số **n-1** không bị xóa, vì vậy có thể cho đỉnh cha của nó là -1.

Với mỗi đỉnh – tìm bậc **degree[i]** của nó,

Con trỏ **ptr** với giá trị tăng dần, được khởi tạo bằng **-1**, trong quá trình xử lý luôn chỉ tới nút lá cần loại bỏ,

Biến **leaf** chỉ tới nút lá cần loại bỏ,

Trong quá trình xử lý, nếu **leaf > parent[leaf]** thì ghi nhận **leaf** và gán **leaf=parent[leaf]**, trong trường hợp ngược lại – gán giá trị **ptr** chỉ tới cho **leaf**.

Như vậy, trong quá trình xử lý không có bước quay lui nào với **ptr** và mỗi nút chỉ được xử lý một lần, do đó độ phức tạp của giải thuật sẽ là $O(n)$.

Chương trình

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

const int MAXN = 100001;
int n, a, b;
vector<int> g[MAXN];
int degree[MAXN], parent[MAXN];
```

```

void dfs (int v)
{
    for (size_t i=0; i<g[v].size(); ++i)
    {
        int to = g[v][i];
        if (to != parent[v])
        {
            parent[to] = v;
            dfs (to);
        }
    }
}

vector<int> prufer_code()
{
    parent[n-1] = -1;
    dfs (n-1);
    int ptr = -1;
    for (int i=0; i<n; ++i)
    {
        degree[i] = (int) g[i].size();
        if (degree[i] == 1 && ptr == -1) ptr = i;
    }
    vector<int> result;
    int leaf = ptr;
    for (int iter=0; iter<n-2; ++iter)
    {
        int next = parent[leaf];
        result.push_back (next);
        --degree[next];
        if (degree[next] == 1 && next < ptr)
            leaf = next;
        else
        {
            ++ptr;
            while (ptr<n && degree[ptr] != 1)
                ++ptr;
            leaf = ptr;
        }
    }
    return result;
}

int main()
{
    fi>>n;
    for (int i=0; i<n-1; ++i)
    {
        fi>>a>>b;
        --a; --b;
}

```

```

        g[a].push_back(b); g[b].push_back(a);
    }
vector<int> ans(n-2);
ans = prufer_code();
for (int i:ans) fo<<i+1<<' ';
}

```

Một số tính chất của mã Prufer

Sau khi xử lý xong, trong cây còn 2 đỉnh, một trong 2 đỉnh đó có số là **n-1**, về đỉnh kia – không thể nói gì!

Só lần mỗi đỉnh tham gia vào mã Prufer bằng số bậc của nó trừ 1. Điều này không khó hiểu vì đỉnh sẽ bị loại bỏ khi có bậc bằng 1. Điều này cũng đúng với cả 2 đỉnh còn lại của cây!

Khôi phục cây theo mã Prufer

Dựa vào tính chất của mã Prufer có thể xác định bậc của mỗi đỉnh trong cây,

Từ đây có thể xác định lá đầu tiên bị loại trong cây: đó là lá có số nhỏ nhất và đỉnh cha của nó là số đầu tiên trong mã Prufer,

Ghi nhận cạnh này và giảm bậc ở các đỉnh tương ứng,

Lặp lại các bước xử lý này cho đến khi còn 2 đỉnh bậc 1 không được ghi nhận trong mã đang xét, bổ sung cạnh nối 2 đỉnh này vào kết quả.

Sử dụng các cấu trúc dữ liệu hỗ trợ tìm min (**set**, **priority_queue**, ...) để dàng xây dựng chương trình với hiệu quả thực hiện $O(n \log n)$.

Chương trình

```

#include <bits/stdc++.h>
using namespace std;
ifstream fi ("pruf_dec.inp");
ofstream fo ("pruf_dec.out");
int n, a;

vector < pair<int,int> > prufer_decode (const vector<int> &
prufer_code)
{
    n+= 2;
    vector<int> degree (n, 1);
    for (int i=0; i<n-2; ++i)
        ++degree[prufer_code[i]];
    set<int> leaves;
    for (int i=0; i<n; ++i)
        if (degree[i] == 1)
            leaves.insert (i);
    vector < pair<int,int> > result;
    for (int i=0; i<n-2; ++i)

```

```

    {
        int leaf = *leaves.begin();
        leaves.erase (leaves.begin());
        int v = prufer_code[i];
        result.push_back (make_pair (leaf, v));
        if (--degree[v] == 1)
            leaves.insert (v);
    }
    result.push_back (make_pair (*leaves.begin(),
                                 *--leaves.end()));
    return result;
}

int main()
{
    fi>>n;
    vector<int> prufer_code(n);
    for (int i=0; i<n; ++i)
    {
        fi>>a;
        prufer_code[i]=--a;
    }
    vector<pair<int, int>> ans;
    ans=prufer_decode (prufer_code);
    fo<<n<<'\\n';
    for (int i=0; i<n-1; ++i)
        fo<<ans[i].first+1<<' ' <<ans[i].second+1<<'\\n';
}

```

Khôi phục cây với độ phức tạp tuyến tính

Để khôi phục cây với độ phức tạp tuyến tính ta sử dụng kỹ thuật đã áp dụng để tính mã Prufer với độ phức tạp O(n).

Để tìm được lá có số nhỏ nhất không nhất thiết phải sử dụng cấu trúc dữ liệu hỗ trợ tìm min.

Sau khi tìm và xử lý xong một nút lá con trỏ tăng dần sẽ tới nút mới cần xử lý.

Chương trình

```

#include <bits/stdc++.h>
using namespace std;
ifstream fi ("prufer_dec.inp");
ofstream fo ("prufer_dec.out");
int n, a;

vector<pair<int,int>> prufer_decode_linear
    (const vector<int> & prufer_code)
{
    n +=2;
    vector<int> degree (n, 1);
    for (int i=0; i<n-2; ++i)

```

```

        ++degree[prufer_code[i]];
    int ptr = 0;
    while (ptr < n && degree[ptr] != 1) ++ptr;
    int leaf = ptr;
    vector<pair<int,int>> result;
    for (int i=0; i<n-2; ++i)
    {
        int v = prufer_code[i];
        result.push_back (make_pair (leaf, v));
        --degree[leaf];
        if (--degree[v] == 1 && v < ptr)
            leaf = v;
        else
        {
            ++ptr;
            while (ptr < n && degree[ptr] != 1)
                ++ptr;
            leaf = ptr;
        }
    }
    for (int v=0; v<n-1; ++v)
        if (degree[v] == 1)
    result.push_back ({v, n-1});
    return result;
}

int main()
{
    fi>>n;
    vector<int> pruf_code(n);
    for(int i=0; i<n; ++i)
    {
        fi>>a;
        pruf_code[i]=--a;
    }
    vector<pair<int, int>> ans;
    ans=prufer_decode_linear(pruf_code);
    fo<<n<<'\\n';
    for(int i=0; i<n-1; ++i) fo<<ans[i].first+1<<' '
                                <<ans[i].second+1<<'\\n';
}

```

Tương ứng đơn trị giữa cây và mã Prufer

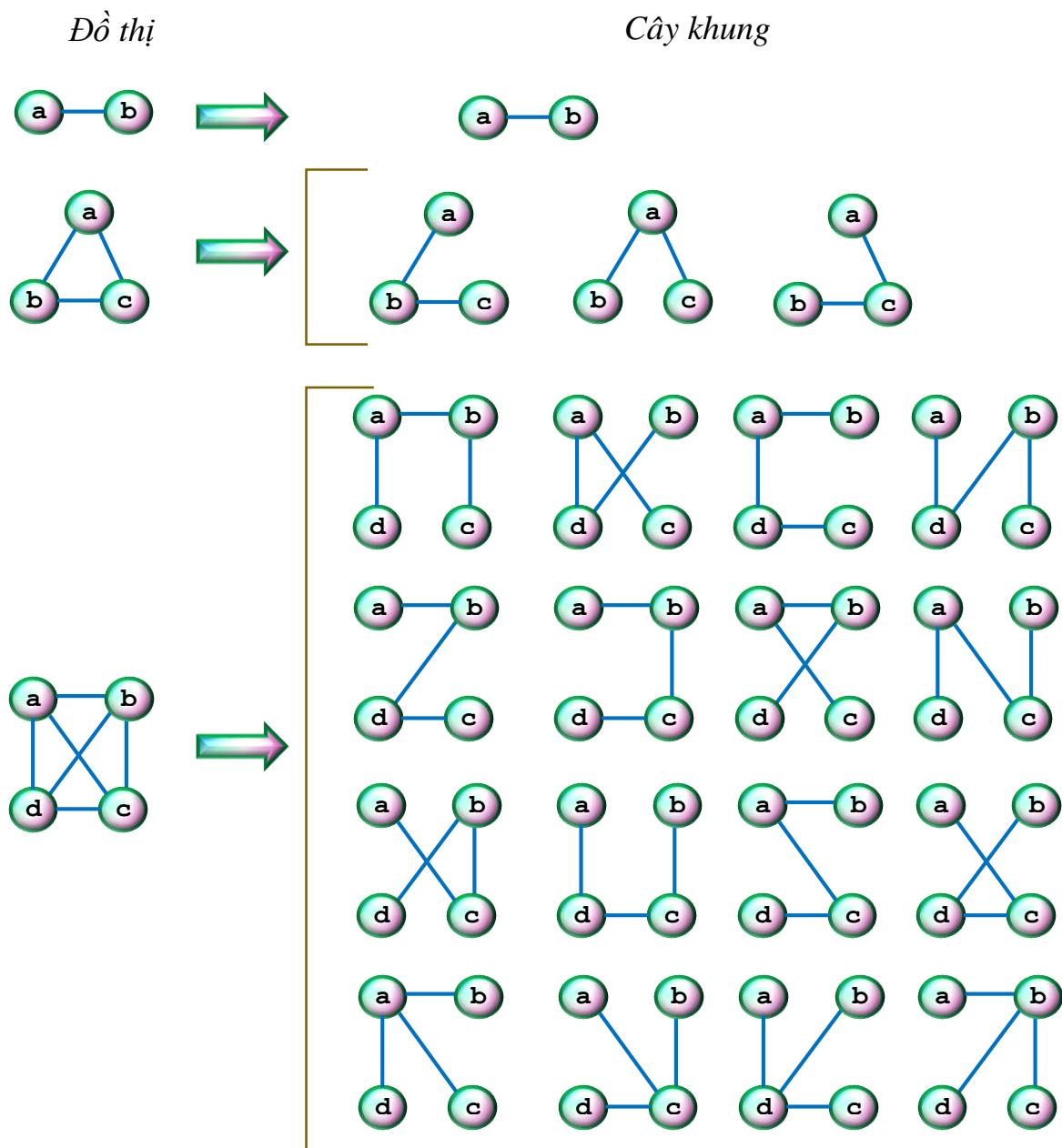
Từ cách xây dựng mã Prufer ta thấy cây đã cho sẽ sinh ra một mã Prufer duy nhất.

Từ giải thuật khôi phục cây, ta thấy mỗi mã Prufer chỉ có thể tương ứng với một cây.

Như vậy tồn tại quan hệ tương ứng đơn trị giữa cây và mã Prufer.

Công thức Cayley (Cayley's Formula)

Số lượng cây khung khác nhau của đồ thị đầy đủ n đỉnh là n^{n-2} .



Có nhiều cách chứng minh công thức này, nhưng đơn giản và dễ hiểu hơn cả là chứng minh thông qua mã Prufer.

Thật vậy, mỗi bộ dữ liệu $n-2$ số, mỗi số thuộc đoạn $[1, n]$ tương ứng đơn trị với cây n đỉnh. Số lượng mã Prufer khác nhau là n^{n-2} . Ở đồ thị đầy đủ n đỉnh, mọi cây đều là cây khung. Đó là điều cần chứng minh.

Số cách bổ sung ít cạnh nhất để có đồ thị liên thông

Mã Prufer cho phép ta suy diễn và kết xuất nhiều kết quả hơn Công thức Cayley. Một trong những vấn đề cần đến sự hỗ trợ của mã Prufer là tính số cách bổ sung cạnh ít nhất để liên thông hóa đồ thị.

Xét đồ thị vô hướng G có n đỉnh và m cạnh. Gọi k là số thành phần liên thông của G . Hãy xác định số cách khác nhau bổ sung $k-1$ cạnh để G trở thành đồ thị liên thông (rõ ràng không thể bổ sung ít cạnh hơn!).

Vận dụng mã Prufer ta có thể dẫn xuất công thức tính số lượng cách bổ sung khác nhau.

Gọi s_1, s_2, \dots, s_k – kích thước các thành phần liên thông của G .

Việc bổ sung cạnh trong một thành phần liên thông sẽ không làm giảm số thành phần liên thông, với số cạnh cho phép còn lại ta không thể biến G thành đồ thị liên thông. Như vậy mỗi cạnh sẽ nối 2 thành phần liên thông.

Mỗi thành phần liên thông có thể coi như một siêu đỉnh. Kết quả bổ sung cạnh cho ta được một cây khung k đỉnh. Vì có thể nối 2 siêu đỉnh bất kỳ nên số cách nối k siêu đỉnh này tương đương với bài toán tính số cây khung trong đồ thị đầy đủ k đỉnh. Vấn đề phức tạp ở chỗ mỗi siêu đỉnh có trọng số riêng: siêu đỉnh thứ i có trọng số s_i vì ta có s_i cách chọn đỉnh trong thành phần liên thông đó.

Để xác định số lượng cần tìm, ta phải biết mỗi đỉnh trong cây khung k đỉnh có những bậc nào. Với mỗi bậc có thể có – tính số lượng cách bổ sung tương ứng. Lấy tổng các số lượng tính được ta sẽ có số lượng cần tìm.

Gọi d_1, d_2, \dots, d_k – bậc của các đỉnh trong cây khung. Tổng tất cả các bậc sẽ gấp đôi số cạnh, do đó có

$$\sum_{i=1}^k d_i = 2k - 2$$

Nếu đỉnh i có bậc d_i , nó sẽ có mặt trong mã Prufer d_{i-1} lần.

Mã Prufer của cây k đỉnh có $k-2$ số. Số cách chọn $k-2$ số, trong đó số i gấp đúng d_i-1 lần là hệ số đa thức (tương tự như hệ số nhị thức):

$$\binom{k-2}{d_1-1, d_2-1, \dots, d_k-1} = \frac{(k-2)!}{(d_1-1)! (d_2-1)! \dots (d_k-1)!}$$

Mỗi số i có s_i cách chọn, vì vậy phải nhân tất cả các cách chọn và kết quả với các bậc d_1, d_2, \dots, d_k sẽ là

$$s_1^{d_1} \cdot s_2^{d_2} \cdot \dots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)! (d_2-1)! \dots (d_k-1)!}$$

Để có kết quả bài toán ta cần lấy tổng kết quả trên với mọi khả năng của các \mathbf{d}_i , tức là lấy tổng với các bộ dữ liệu $\{d_i\}_{i=1}^{i=k}$.

$$\sum_{\substack{d_i \geq 1, \\ \sum_{i=1}^k d_i = 2k-2}} s_1^{d_1} \cdot s_2^{d_2} \cdot \dots \cdot s_k^{d_k} \cdot \frac{(k-2)!}{(d_1-1)! (d_2-1)! \dots (d_k-1)!}$$

Dựa vào công thức triển khai đa thức:

$$(x_1 + \dots + x_m)^p = \sum_{\substack{c_i \geq 0, \\ \sum_{i=1}^m c_i = p}} x_1^{c_1} \cdot x_2^{c_2} \cdot \dots \cdot x_m^{c_m} \cdot \binom{m}{c_1, c_2, \dots, c_k}$$

So sánh công thức này với công thức trước và nếu đặt $\mathbf{e}_i = \mathbf{d}_i - \mathbf{1}$, ta có

$$\sum_{\substack{e_i \geq 0, \\ \sum_{i=1}^k e_i = k-2}} s_1^{e_1+1} \cdot s_2^{e_2+1} \cdot \dots \cdot s_k^{e_k+1} \cdot \frac{(k-2)!}{e_1! e_2! \dots e_k!}$$

Khai triển và rút gọn biểu thức trên ta có kết quả cần tìm:

$$s_1 \cdot s_2 \cdot \dots \cdot s_k \cdot (s_1 + s_2 + \dots + s_k)^{k-2} = s_1 \cdot s_2 \cdot \dots \cdot s_k \cdot n^{k-2}$$

Một số bài toán ứng dụng

Trong các bài toán ví dụ dưới đây, một số khái niệm được *cục bộ hóa trong phạm vi bài toán* và có thể có một khác biệt về mặt kỹ thuật, nhưng *không dẫn đến các mâu thuẫn* với lý thuyết chung.

Bài 1 MÃ PRUFER

Cho cây n đỉnh, các đỉnh được đánh số từ 1 đến n . Mã Prüfer với cây được xây dựng như sau:

- Nút lá (nút có bậc bằng 1) có số nhỏ nhất được chọn,
- Ghi nhận số của nút được chọn vào mã,
- Xóa nút được chọn và cạnh nối tới nó khỏi cây.

Quá trình trên được lặp lại cho đến khi cây chỉ còn một nút.

Dãy các số được ghi nhận (theo trình tự xuất hiện) được gọi là mã Prufer của cây.

Cho mã Prufer, hãy khôi phục lại cấu trúc của cây bằng cách đưa ra danh sách đỉnh kề của mỗi đỉnh.

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

- ─ Dòng đầu tiên chứa số nguyên n ($1 < n \leq 10^5$),
- ─ Dòng thứ 2 chứa $n-1$ số nguyên c_1, c_2, \dots, c_{n-1} xác định mã Prufer của cây ($1 \leq c_i \leq n, i = 1 \div n-1$).

Kết quả: Đưa ra file văn bản PRUFCODE.OUT n dòng, dòng thứ i chứa số nguyên i , dấu “:” và sau đó là các số nguyên xác định các đỉnh kề với đỉnh i .

Ví dụ:

PRUFCODE.INP	PRUFCODE.OUT
6	1: 4 6 2: 3 5 6 3: 2 4: 1 5: 2 6: 1 2
2 1 6 2 6	



Giải thuật; Lý thuyết mã Prufer .

Đây là bài tập, vì vậy khái niệm về mã Prufer được phát biểu hơi khác một chút so với lý thuyết chung về mã Prufer.

Trong dữ liệu vào, số cuối cùng (số thứ $n-1$) là tiền định đối với các số trước đó, vì vậy ta có thể bỏ qua nó.

Áp dụng lý thuyết khôi phục cây từ mã Prufer với chi phí thời gian $O(n)$ ta có thể tìm được danh sách các cạnh,

Từ danh sách cạnh có thể dễ dàng xác định danh sách đỉnh kề với chi phí thời gian $O(n)$.

Tổ chức dữ liệu:

- ─ Mảng `vector<int>` pruf_code (n) – lưu mã Prufer,
- ─ Mảng `vector<int>` degree ($n, 1$) – lưu bậc của mỗi đỉnh,

- Mảng `vector<pair<int, int>>` `ans` – lưu danh sách các cạnh,
- Mảng `vector<vector<int>>` `g(n+1)` – lưu danh sách đỉnh kề.

Độ phức tạp của giải thuật: $O(n)$.

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("prufcode.inp");
ofstream fo ("prufcode.out");
int n,a,b;

vector<pair<int,int>>prufer_decode_linear(const vector<int> &prufer_code)
{
    n +=2;
    vector<int> degree (n, 1);
    for (int i=0; i<n-2; ++i)
        ++degree[prufer_code[i]];
    int ptr = 0;
    while (ptr < n && degree[ptr] != 1) ++ptr;
    int leaf = ptr;
    vector < pair<int,int> > result;
    for (int i=0; i<n-2; ++i)
    {
        int v = prufer_code[i];
        result.push_back ({leaf, v});
        --degree[leaf];
        if (--degree[v] == 1 && v < ptr)
            leaf = v;
        else
        {
            ++ptr;
            while (ptr < n && degree[ptr] != 1)
                ++ptr;
            leaf = ptr;
        }
    }
    for (int v=0; v<n-1; ++v)
        if (degree[v] == 1)
            result.push_back ({v, n-1});
    return result;
}

int main()
{
    fi>>n; n-=2;
    vector<int> pruf_code(n);
    for(int i=0; i<n; ++i)
    {
        fi>>a;
        pruf_code[i] = -a;
    }
    vector<pair<int, int>> ans;
    ans = prufer_decode_linear(pruf_code);

    vector<vector<int>> g(n+1);
}
```

```
for(int i=0; i<n-1; ++i)
{
    a=ans[i].first+1;
    b=ans[i].second+1;
    g[a].push_back(b);
    g[b].push_back(a);
}
for(int i=1; i<= n; ++i)
{
    fo<<i<<" ";
    for(int j:g[i]) fo<<j<<' ';
    fo<<'\n';
}
fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



Bài 2. MANH MỐI

Sherlock Holmes đang điều tra phá một vụ án. Ông tìm được n manh mối và đang có gắng tìm cách kết nối chúng với nhau. Các manh mối được đánh số từ 1 đến n . Ông đã tìm được m mối quan hệ trực tiếp giữa một số cặp manh mối. Hai manh mối có thể có mối liên hệ trực tiếp với nhau hoặc liên hệ qua các manh mối trung gian khác. Mỗi liên hệ giữa 2 manh mối là 2 chiều, tức là A có mối quan hệ với B thì suy ra B cũng có quan hệ tới A. Dĩ nhiên, không có đường liên kết một manh mối tới chính nó.

Holmes có thể tìm và xác định mối quan hệ giữa 2 manh mối bất kỳ để phá án, nhưng điều đó sẽ tốn rất nhiều thời gian và thủ phạm sẽ trốn mất. Sau một thời gian suy nghĩ, ông thấy rằng để phá án chỉ cần tìm thêm một số ít nhất các quan hệ giữa các cặp manh mối để đảm bảo giữa 2 manh mối bất kỳ có một đường xác định quan hệ.

Holmes đang phân vân xem cần chọn bộ các cặp manh mối nào cần mối quan hệ giữa chúng. Hai cách chọn gọi là khác nhau nếu tồn tại ít nhất một cặp manh mối có trong cách chọn thứ nhất và không có trong cách thứ hai.

Hãy xác định số bộ các cặp manh mối khác nhau có thể được chọn và đưa ra theo mô đun p .

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

- ✚ Dòng đầu tiên chứa số nguyên n , m và p ($1 \leq n \leq 10^5$, $0 \leq m \leq 10^5$, $1 \leq p \leq 10^9$),
- ✚ Mỗi dòng trong m dòng tiếp theo chứa 2 số nguyên a và b xác định cặp manh mối đã xác định được quan hệ trực tiếp ($1 \leq a, b \leq n$, $a \neq b$). Không có cặp quan hệ nào bị lặp.

Kết quả: Đưa ra file văn bản CLUES.OUT một số nguyên – số bộ theo mô đun p các cặp manh mối khác nhau có thể được chọn.

Ví dụ:

CLUES.INP
4 1 1000000
1 4

CLUES.OUT
8



Giải thuật: Tìm số miền liên thông, Tính số cách liên thông hóa đồ thị .

Các manh mồi tạo thành đồ thị vô hướng **n** đỉnh và **m** cạnh.

Trước tiên cần xác định số miền liên thông và kích thước (số đỉnh) của mỗi miền liên thông.

Gọi **k** là số miền liên thông xác định được và **sz_i** – kích thước miền liên thông thứ **i** (**i** = 1 ÷ **k**).

Kết quả cần tìm:

$$\text{ans} = \mathbf{sz}_1 \times \mathbf{sz}_2 \times \dots \times \mathbf{sz}_k \times n^{k-2} \quad (\text{mod } p)$$

Để tính lũy thừa cần dùng sơ đồ nhân Ai Cập tính nhanh lũy thừa.

Tổ chức dữ liệu:

- Mảng **vector <int>** **g[N]** – lưu danh sách đỉnh kề của đồ thị ban đầu,
- Mảng **vector<bool>** **used** – đánh dấu đỉnh đã thăm trong quá trình tìm miền liên thông,
- Mảng **vector<int>** **sz(n+1)** – lưu kích thước các miền liên thông.

Độ phức tạp của giải thuật: O(*n*).

Chi phí thời gian chủ yếu ở khâu xác định số miền liên thông và kích thước của mỗi miền.

Chương trình

```
#include <bits/stdc++.h>
#define NAME "clues."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100000;

int n,m,p,q,k=0,x,y;
vector<bool> used;
vector <int> g[N];

int64_t pw(int64_t u )
{
    int64_t res=1,tm=n, tp=u;
    while(tp)
    {
        if (tp&1) res=res*tm%p;
        tm=tm*tm%p;
        tp>>=1;
    }
    return res;
}

void dfs(int v)
{
    used[v]=true; ++q;
    for(int i:g[v])
        if (!used[i]) dfs(i);
}

int main()
{
    fi>>n>>m>>p;
    for(int i=0;i<m;++i)
    {
        fi>>x>>y;
        --x; --y; g[x].push_back(y); g[y].push_back(x);
    }
    used.assign(n,false);
    vector<int>sz(n+1);
    for(int i=0;i<n;++i)
    {
        if (!used[i])
        {
            ++k; q=0;
            dfs(i);
            sz[k]=q;
        }
    }
}
```

```
int64_t ans=1;
for (int i=1; i<=k; ++i) ans=ans*sz[i]%p;
ans=ans*pw(k-2)%p;
fo<<ans;
fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



BÀI TẬP THEO CÁC CHUYÊN ĐỀ KHÁC NHAU

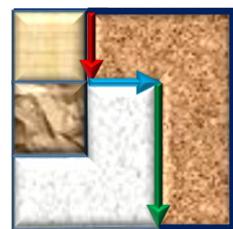
VZ14. RẠO CHƠI Ở BROOKLYN

Tên chương trình: WALK.CPP

Từ trên cao nhìn xuống Brooklyn có hình chữ nhật kích thước $n \times m$ ô vuông, có các cạnh chạy từ bắc xuống nam và từ tây sang đông. Mỗi nhà ở Brooklyn chiếm một ô hoặc một số ô kề cạnh. Trên bản đồ các ô thuộc cùng một nhà được ký hiệu bằng cùng một chữ cái la tinh hoa. Các nhà kề nhau được ký hiệu bằng những ký tự khác nhau. Biên của khu phố và đường phân cách giữa các nhà là đường đi.

Maicon đang đứng ở đường biên phía bắc của khu phố và có việc phải đi tới một tòa nhà ở đường biên phía nam. Liếc nhìn đồng hồ, thấy còn dư một chút thời gian Maicon hào phóng tự thưởng cho mình một chuyến du ngoạn xuyên qua Brooklyn chứ không cần phải đi theo con đường ngắn nhất tới đích. Tuy vậy thời gian thừa cũng không phải quá nhiều nên Maicon chỉ đi theo các hướng: xuống phía nam, sang đông hoặc sang tây, ngoài ra còn không đi một đoạn đường nào quá một lần kề cả khác chiều. Để khỏi phân vân suy nghĩ nhiều khi gấp ngã rẽ, Maicon quyết định sẽ đi sao cho chênh lệch giữa diện tích của phần khu phố ở phía tây đường đi của mình với diện tích ở phía đông của đường đi là nhỏ nhất.

ABB
BAB
AAB



Hãy xác định chênh lệch nhỏ nhất mà Maicon có thể đạt được.

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

- ✚ Dòng đầu tiên chứa 2 số nguyên n và m ($1 \leq n, m \leq 300$),
- ✚ Mỗi dòng trong n dòng sau chứa xâu độ dài m chỉ chứa các ký tự la tinh in hoa xác định một dòng của bản đồ theo thứ tự từ bắc xuống nam.

Kết quả: Đưa ra file văn bản WALK.OUT một số nguyên – chênh lệch nhỏ nhất có thể đạt được,

Ví dụ:

WALK.INP
3 3
ABB
BAB
AAB

WALK.OUT
1

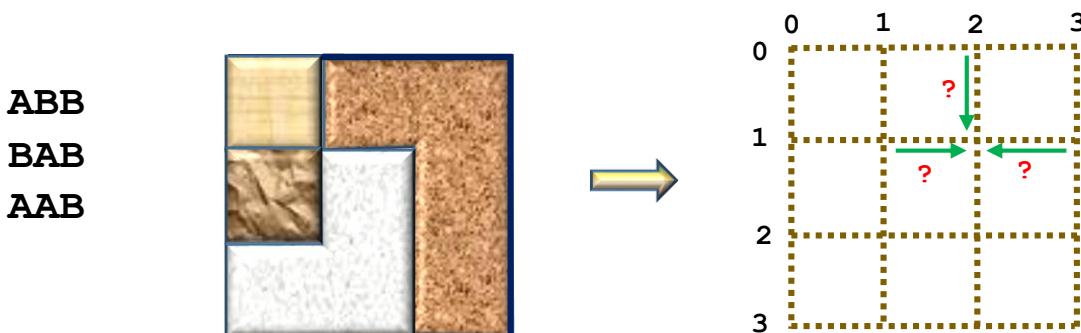


Giải thuật: Quy hoạch động, Kỹ thuật bảng phương án.

Nhận xét:

- ✿ Việc di chuyển được thực hiện theo từng lópvà có hướng, từ trên cùng (lớp 0) xuống lớp thấp nhất (lớp n),
- ✿ Trong một lớp: Di chuyển có hướng, từ tây sang đông hoặc ngược lại.
- ✿ Như vậy, không có điểm nào trên đường đi bị lặp lại!

Bài toán với các đặc trưng trên luôn luôn có thể giải bằng quy hoạch động (cũng có thể tồn tại những giải thuật lớp khác với hiệu quả hơn hoặc kém).



Điểm giao của 2 đường đi được gọi là giao lộ.

Tối đa có $n \times m$ giao lộ (trường hợp mỗi nhà chiếm đúng một ô).

Mỗi giao lộ có thể tới từ bên trái hoặc từ bên phải → phải quản lý $2 \times n \times m$ trạng thái mô tả đường đi.

Mục tiêu cần tối ưu hóa là chênh lệch diện tích 2 phần bị phân chia bởi đường đi.

Chênh lệch diện tích d : nhận giá trị trong phạm vi từ 0 đến $n \times m$.

Ta cần biết, có *tới được giao lộ (x, y) với chênh lệch diện tích là d* hay không?

Nếu có thông tin trả lời câu hỏi trên, ta chỉ cần duyệt các giao lộ ở dòng cuối cùng, tìm *min* của chênh lệch d .

Để trả lời câu hỏi trên cần xây dựng bảng $dp_{x,y,d}$ lưu trữ câu trả lời (*có* hoặc *không*).

$dp_{x,y,d}$ được tính từ $dp_{x-1,y,d}$.

Vấn đề phức tạp ở đây là *tổ chức dữ liệu*.

Tổng số đại lượng của bảng dp là $(n \times m)^2$.

Việc lưu trữ toàn bộ bảng sẽ cho giải thuật với độ phức tạp thấp nhất, nhưng chỉ thích hợp khi $n, m \leq 100$. Với kích thước lớn hơn – không đủ bộ nhớ.

Từ kết quả dòng trên ta tính được các giá trị ứng với dòng dưới tiếp theo. Do đó chỉ cần lưu trữ hai dòng của bảng.

Để biến dòng mới thành dòng cũ: áp dụng kỹ thuật con lắc, tránh việc gán giá trị của một dòng sang dòng khác.

Giá trị cần lưu trữ thuộc loại **bool** nên thông tin về một dòng có thể tổ chức dưới dạng nén bằng kiểu dữ liệu **bitset**. Các công cụ lưu trữ và xử lý thông tin bit dưới dạng nén của C++ cho phép giảm thời gian thực hiện 32 lần với dữ liệu trong phạm vi **int**.

Trọng tâm của giải thuật:

Sơ đồ lặp, tính giá trị dòng mới từ giá trị dòng cũ. Sơ đồ lặp còn thường được gọi là *Quy hoạch động đơn giản*.

Việc lựa chọn phương án tối ưu được dựa trên bảng giá trị của mọi khả năng (*Kỹ thuật bảng phương án*).

Bài toán không yêu cầu dẫn xuất đường đi vì vậy không cần lưu vết để truy ngược.

Các phương án hiện thực hóa giải thuật khác nhau có *độ phức tạp của giải thuật* khác nhau phụ thuộc vào *cách khai báo dữ liệu* và *thời điểm xin cấp phát bộ nhớ*.

Dưới đây là 3 cách tiếp cận có *độ phức tạp của giải thuật* và *độ phức tạp lập trình* khác nhau. Mỗi cách tiếp cận có sơ đồ tổ chức dữ liệu khác nhau.

Độ phức tạp của giải thuật: thuộc lớp $O(n^4)$, nhưng chênh lệch thời gian thực hiện là đáng kể!

Giải thuật I:

Áp dụng với $n, m \leq 100$,

Thời gian thực hiện: Tối ưu,

Nhược điểm: Tốn bộ nhớ.

Tổ chức dữ liệu:

- Mảng **vector<string>** **a (n)** – lưu bản đồ khu phố,
- Mảng **bool dp [MAXN+1] [MAXN+1] [2*SH+1]** – phục vụ sơ đồ quy hoạch động với **MAXN = 110, SH = MAXN*MAXN**.

Xử lý:

Chuẩn bị ban đầu:

```
dp[0][0][SH] = true;
```

Với mỗi dòng: Xét *lần lượt*:

- Khả năng di chuyển từ trái qua phải,
- Khả năng di chuyển từ phải qua trái,
- Ở mỗi nút: duyệt mọi khả năng của **d** có thể tới được, tính hiệu chênh lệch và ghi nhận **dp_{x,y,d}**.

Giải thuật II:

Tương tự như giải thuật I, nhưng áp dụng sơ đồ phân phối bộ nhớ động,

Dùng 2 mảng 2 chiều thay cho việc sử dụng mảng 3 chiều:

```
vector<vector<bool>> d(m+1, vector<bool>(n*m*2+1));
for (int i = 0; i <= m; ++i) d[i][n * m] = 1;
vector<vector<bool>> next(m+1, vector<bool>(n*m*2+1));
```

Chuẩn bị giá trị đầu: Cho cả một dòng!

Thời gian thực hiện:

- ✚ Rất lớn,
- ✚ Lý do:
 - ✿ Phải phân phối động vùng bộ nhớ lớn,
 - ✿ Mỗi biến loại **bool** chỉ chiếm một byte, trong C/C++ việc truy nhập vào byte của biến loại **bool** được thực hiện như truy nhập vào số nguyên loại **int_8** và tốn nhiều thời gian hơn truy nhập vào biến nguyên loại **int_32**.

Giải thuật III:

Tương tự như giải thuật II, nhưng áp dụng kiểu dữ liệu **bitset** cho phép tiết kiệm bộ nhớ và giảm thời gian truy cập 32 – 63 lần.

Chương trình I

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("walk.inp");
//ifstream fi ("100");
ofstream fo ("walk.out");

const int MAXN = 110;
const int SH = MAXN * MAXN;
bool dp[MAXN + 1][MAXN + 1][2 * SH + 1];

int main()
{
    int n, m;
    fi >> n >> m;
/*
    int t = max(n,m);
    int SH=t*t;
    vector<vector<vector<bool>>>
        dp(t+1,vector<vector<bool>>(t+1,vector<bool>(SH * 2 + 1)));
*/
    vector<string> a(n);
    for (int i = 0; i < n; i++) fi >> a[i];

    dp[0][0][SH] = true;
    for (int x = 0; x < n; x++)
    {
        for (int y = 0; y <= m; y++)
            for (int d = SH - x * m; d <= SH + x * m; d++)
            {
                if (!dp[x][y][d]) continue;

                // left
                if (y > 0)
                    if (x == 0 || a[x - 1][y - 1] != a[x][y - 1])
                        dp[x][y - 1][d] = true;

                // right
                if (y < m)
                    if (x == 0 || a[x - 1][y] != a[x][y])
                        dp[x][y + 1][d] = true;

                // down
                if (y == 0 || y == m || a[x][y - 1] != a[x][y])
                    dp[x + 1][y][d + y - (m - y)] = true;
            }

        for (int y = m - 1; y >= 0; y--)
            for (int d = SH - x * m; d <= SH + x * m; d++)
            {
                if (!dp[x][y][d]) continue;

                // left
                if (y > 0)
                    if (x == 0 || a[x - 1][y - 1] != a[x][y - 1])
                        dp[x][y - 1][d] = true;
```

```

        //right
        if (y < m)
            if (x == 0 || a[x - 1][y] != a[x][y])
                dp[x][y + 1][d] = true;

        // down
        if (y == 0 || y == m || a[x][y - 1] != a[x][y])
            dp[x + 1][y][d + y - (m - y)] = true;
    }

}

int ans = SH;
for (int y = 0; y <= m; y++)
    for (int d = 0; d <= 2 * SH; d++)
        if (dp[n][y][d]) ans = min(ans, abs(d - SH));
fo << ans << endl;

fo<<"\nTime: "<<clock() / (double)1000<<" sec";
return 0;
}

```

Chương trình II

```

#include <bits/stdc++.h>
using namespace std;
ifstream fi ("walk.inp");
ofstream fo ("walk.out");
typedef long long ll;
int const INF = (int)1e9 + 1e3;

int main() {
    int n, m;
    fi >> n >> m;
    vector<string> field(n);
    for (int i = 0; i < n; ++i) fi>> field[i];
    vector<vector<bool>> d(m + 1, vector<bool>(n * m * 2 + 1));
    for (int i = 0; i <= m; ++i) d[i][n * m] = 1;
    vector<vector<bool>> next(m + 1, vector<bool>(n * m * 2 + 1));
    for (int i = 0; i < n; ++i)
    {
        if (i)
        {
            for (int j = 0; j < m; ++j)
                if (field[i - 1][j] != field[i][j])
                    for (int k = 0; k <= n * m * 2; ++k)
                        d[j + 1][k] = d[j + 1][k] | d[j][k];

            for (int j = m; j > 0; --j)

                if (field[i - 1][j - 1] != field[i][j - 1])
                    for (int k = 0; k <= n * m * 2; ++k)
                        d[j - 1][k] = d[j - 1][k] | d[j][k];
        }
        for (int j = 0; j < m + 1; ++j)
    }
}

```

```

        fill(next[j].begin(), next[j].end(), 0);

    for (int j = 0; j <= m; ++j)
        if (j == 0 || j == m || field[i][j - 1] != field[i][j])
            for (int k = 0; k <= n * m * 2; ++k)
            {
                int to = k + j - (m - j);
                if (0 <= to && to <= n * m * 2)
                    next[j][to] = next[j][to] | d[j][k];
            }
        swap(next, d);
    }
int ans = INF;
for (int i = 0; i < m + 1; ++i)
    for (int j = 0; j <= n * m * 2; ++j)
        if (d[i][j]) ans = min(ans, abs(j - n * m));

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

```

Chương trình III

```

#include <bits/stdc++.h>
using namespace std;
int const INF = (int)1e9 + 1e3;
ifstream fi ("walk.inp");
ofstream fo ("walk.out");
const int MAXN = 305;
const int MAXS = MAXN * MAXN;
bitset<MAXS * 2> d[2][MAXN];
bitset<MAXS * 2> ZERO;

int main() {
    int n, m;
    fi >> n >> m;
    vector<string> field(n);
    for (int i = 0; i < n; ++i) fi>>field[i];

    for (int i = 0; i <= m; ++i) d[0][i][n * m] = 1;

    for (int i = 0; i < n; ++i)
    {
        int cur = i & 1;
        int next = 1 - cur;
        if (i)
        {
            for (int j = 0; j < m; ++j)
                if (field[i - 1][j] != field[i][j])
                    d[cur][j + 1] |= d[cur][j];
        }

        for (int j = m; j > 0; --j)

```

```

    if (field[i - 1][j - 1] != field[i][j - 1])
        d[cur][j - 1] |= d[cur][j];
}

for (int j = 0; j <= m; ++j)
    if (j == 0 || j == m || field[i][j - 1] != field[i][j])
    {
        int val = j - (m - j);
        if (val < 0) d[next][j] = d[cur][j] >> (-val);
        else d[next][j] = d[cur][j] << val;
    } else d[next][j] = ZERO;
}

int ans = INF;
for (int i = 0; i <= m; ++i)
    for (int j = 0; j <= n * m * 2; ++j)
        if (d[n & 1][i][j]) ans = min(ans, abs(n * m - j));
fo << ans << "\n";
fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}

```



VZ15. THỜI GIAN

Tên chương trình: INTERVAL.CPP

Thời gian và địa điểm một cuộc họp quan trọng được giữ bí mật đến phút chót. Vào thời điểm **hh1 : mm1** (lúc **hh1** giờ **mm1** phút) người ta mới có quyết định nó sẽ được tổ chức vào thời điểm **hh2 : mm2** ngày hôm sau.

Đĩ nhiên các bộ phận hữu quan phải vắt chân lên cổ chạy đi chuẩn bị. Tùy theo khoảng thời gian còn lại người ta sẽ quyết định làm món ăn gì khai vị trước khi vào cuộc họp vì các món ăn đòi hỏi có thời gian tầm ướp và ủ ngầm gia vị khác nhau. Cần phải báo cho bếp trưởng khoảng thời gian này để ông có kế hoạch phù hợp. Lưu ý rằng, sau 23h 59m là 0h 0m của ngày hôm sau.

Hãy đưa ra khoảng thời gian cần báo tới bếp trưởng.

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

- ✚ Dòng đầu tiên chứa xâu xác định thời điểm **hh1 : mm1** (xem quy cách nêu ở ví dụ),
- ✚ Dòng thứ 2 chứa xâu xác định thời điểm **hh2 : mm2**.

Kết quả: Đưa ra file văn bản INTERVAL.OUT: xâu xác định khoảng thời gian (theo quy cách như khi xác định thời điểm).

Ví dụ:

INTERVAL.INP	INTERVAL.OUT
12 : 34	
21 : 43	33 : 09



Giải thuật; Đổi cơ số, Xử lý xâu.

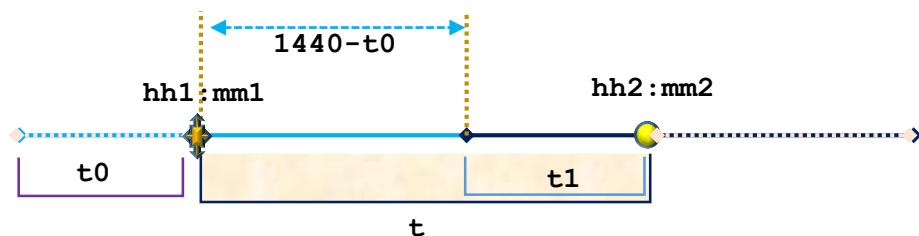
Giờ, phút, giây được biểu diễn theo cơ số 60,

Để thuận tiện xử lý ta cần đưa thời gian về cùng một đơn vị là phút,

Mỗi ngày có $24 \times 60 = 1440$ phút.

Khoảng thời gian t từ thời điểm thứ nhất thời thời điểm thứ hai thuộc ngày khác có thể chia thành 2 phần:

- Từ thời điểm thứ nhất tới cuối ngày,
- Từ đầu ngày mới tới đến thời điểm thứ hai.



Khi tính được khoảng thời gian theo phút: dễ dàng chuyển sang cách tính theo giờ và phút.

Lưu ý quy cách đưa ra dữ liệu.

Dộ phức tạp của giải thuật: O(1).

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("interval.inp");
ofstream fo ("interval.out");
int t1,t2,t;
string s1,s2;

int main()
{
    fi>>s1>>s2;
    t0=((s1[0]-48)*10+s1[1]-48)*60+(s1[3]-48)*10+s1[4]-48;
    t2=((s2[0]-48)*10+s2[1]-48)*60+(s2[3]-48)*10+s2[4]-48;
    t=1440-t0+t2;

    fo<<setw(2)<<setfill('0')<<t/60<<':':<<setw(2)<<setfill('0')<<t%60;
}
```



VZ16. CẤP MÃ TRUY NHẬP

Tên chương trình: PAIR_K.CPP

Một trang WEB phục vụ trang bị kiến thức xử lý xâu chứa rất nhiều giải thuật và bài toán về xâu. Nội dung của trang WEB được chia thành nhiều mức tùy theo trình độ người vào. Để xác định mức kiến thức phù hợp cung cấp cho người truy nhập một xâu s chỉ chứa các ký tự trong tập $\{0, 1\}$ xuất hiện trên màn hình và yêu cầu người vào phải chỉ ra 2 đoạn $[l_1, r_1]$ và $[l_2, r_2]$ thỏa mãn các điều kiện:

- ✿ $l_1 \leq r_1, l_2 \leq r_2,$
- ✿ Hai đoạn có độ dài giống nhau, tức là $r_1 - l_1 = r_2 - l_2,$
- ✿ Hai đoạn đã nêu phải khác nhau, tức là $l_1 \neq l_2$ (hoặc $r_1 \neq r_2$),
- ✿ Tổng các chữ số của s trong đoạn $[l_1, r_1]$ bằng tổng các chữ số của s trong đoạn $[l_2, r_2],$
- ✿ $r_1 - l_1$ – lớn nhất.

Nếu người truy nhập đưa ra đáp án đúng thì sẽ được truy nhập tới mọi thông tin trong trang WEB. Nếu đáp án sai – tùy theo mức độ sai sót người vào sẽ chỉ được vào một số phần này hay phần khác của trang.

Hãy xác định l_1, r_1, l_2, r_2 để có thể truy nhập được tới mọi thông tin trong trang WEB.

Dữ liệu: Vào từ file văn bản PAIR_K.INP gồm một dòng chứa xâu s độ dài không quá 10^6 .

Kết quả: Đưa ra file văn bản PAIR_K.OUT trên một dòng 4 số nguyên l_1, r_1, l_2, r_2 . Nếu tồn tại nhiều phương án khác nhau – đưa ra phương án tùy chọn. Nếu không tồn tại cách chọn hai đoạn – đưa ra số **-1**.

Ví dụ:

PAIR_K.INP	PAIR_K.OUT
010101	2 5 3 6



VZ16_l020190302_B_AXV

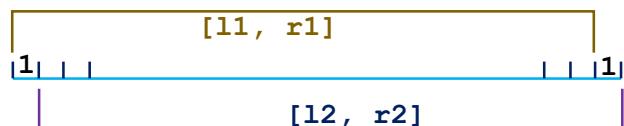
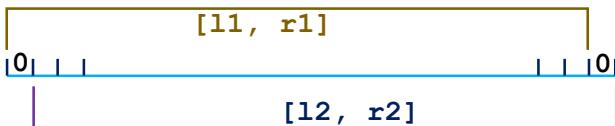
Giải thuật: Kỹ năng nhận dạng tình huống .

Các trường hợp vô nghiệm:

- $n = 1$,
- $n = 2$ và các bít giống nhau.

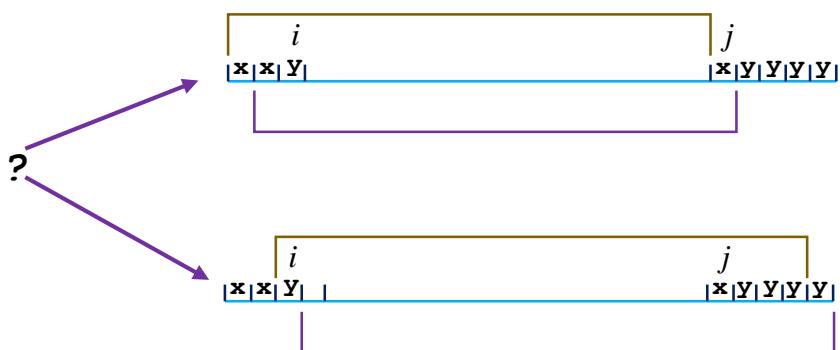
Các trường hợp còn lại – có 2 khả năng:

Hai ký tự đầu và cuối của xâu giống nhau ($s_0 = s_{n-1}$):



Trường hợp $s_0 \neq s_{n-1}$:

- Tìm i nhỏ nhất thỏa mãn $s_0 \neq s_i$,
- Tìm j lớn nhất thỏa mãn $s_j \neq s_{n-1}$,
- Lựa chọn một trong 2 khả năng:



Độ phức tạp của giải thuật: $O(n)$.

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("pair_k.inp");
ofstream fo ("pair_k.out");
int n;
string s;

int main()
{
    fi >> s;
    int n = s.size();
    if (n == 1)
    {
        fo << "-1\n";
        return 0;
    }
    if (s[0] == s[n - 1])
    {
        fo << 1 << ' ' << n - 1 << ' ' << 2 << ' ' << n << '\n';
        return 0;
    }
    if (n == 2)
    {
        fo << "-1\n";
        return 0;
    }
    int i = 0, j = n - 1;
    while (i < n && s[i] == s[0]) i++;
    while (j >= 0 && s[j] == s[n - 1]) j--;
    if (j - 1 > n - i - 2 || i + 1 > n - 1)
        fo << 1 << ' ' << j << ' ' << 2 << ' ' << j + 1 << '\n';
    else
        fo << i + 1 << ' ' << n - 1 << ' ' << i + 2 << ' ' << n << '\n';
    fo << "\nTime: "<< clock() / (double) 1000 << " sec";
    return 0;
}
```



Một nhóm n sinh viên tốt nghiệp tập hợp nhau thành lập một công ty cung cấp dịch vụ phần mềm. Công ty hoạt động theo mô hình gia đình: có một số cặp người (a_i, b_i), trong đó b_i có vai trò như nhân viên của a_i —người a_i có thể phân công công việc cho người b_i , đến lượt mình, người b_i có thể thực hiện công việc được giao hoặc phân công cho người có vai trò như nhân viên của mình. Từ bản ghi chép hoạt động hàng ngày có thể ghi nhận được m cặp quan hệ. Tuy nhiên trong số đó có thể có các cặp bị lặp!

Với năng lực và nhiệt huyết của những người sáng lập, công việc kinh doanh phát triển tốt và mô hình quan hệ tự phát trên không còn phù hợp. Cần phải có sự phân cấp chính thức với trách nhiệm và quyền lợi rõ ràng. Mọi người quyết định bầu một người đứng đầu làm Giám đốc và đó là người không có ai được quyền điều khiển, giao việc. Mỗi người trong số còn lại chịu sự lãnh đạo của đúng một người. Mọi công việc do Giám đốc đưa ra đều có thể tới được người bất kỳ trong Công ty. Ngoài ra, để tránh xáo trộn trong hoạt động hàng ngày, với mỗi người x trong Công ty, tập những người mà từ đó họ có thể nhận công việc là không thay đổi.

Hãy xác định, có thể tạo ra sơ đồ hoạt động mới hay không, nếu không được – đưa ra thông báo “**No**”. Trong trường hợp xây dựng được – đưa ra thông báo “**Yes**” và chỉ ra dãy số p_i xác định người có quyền phân công công việc cho người i , $i = 1 \div n$. Với Giám đốc p_i tương ứng có giá trị là **-1**.

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

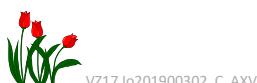
- ✚ Dòng đầu tiên chứa 2 số nguyên n và m ($1 \leq n, m \leq 5 \times 10^5$),
- ✚ Dòng thứ i trong m dòng sau chứa 2 số nguyên a_i và b_i ($1 \leq a_i, b_i \leq n, a_i \neq b_i$).

Kết quả: Đưa ra file văn bản STARTUP.OUT đưa ra thông báo “**No**” hoặc “**Yes**”. Nếu kết quả là “**Yes**” – trên dòng thứ 2 đưa ra n số nguyên p_1, p_2, \dots, p_n tương ứng.

Ví dụ:

STARTUP.INP
3 3
1 2
2 3
1 3

STARTUP.OUT
-1 1 2



VZ17_Io201900302_C_AXV

Giải thuật: Xây dựng cây, Loang theo chiều sâu (dfs) .

Cây cần xây dựng là đồ thị có hướng, vì vậy với cặp quan hệ (**x**, **y**) cho trong input chỉ cần ghi nhận quan hệ kè **x** → **y**.

Nếu các quan hệ đã cho tạo thành chu trình – bài toán vô nghiệm.

Bằng phương pháp đánh dấu khi duyệt dữ liệu, các cặp dữ liệu lặp có thể tự động được bỏ qua, không cần lọc dữ liệu khi nhập để giảm thời gian thực hiện chương trình.

Nếu tồn tại nhiều cây con độc lập → bài toán vô nghiệm.

Quan hệ **x** → **y** trong input cần được đảm bảo trong output (trực tiếp hoặc qua một số nút trung gian của cây) có nghĩa là đường đi từ gốc tới **y** phải qua **x**. Điều này có thể kiểm tra thông qua thời điểm vào và ra của mỗi nút khi loang theo chiều sâu (dfs) để duyệt cây. Gọi **tin_i** – thời điểm vào nút **i**, **tout_i** – thời điểm ra của nút **i**, quan hệ **x** → **y** ban đầu phải dẫn đến quan hệ **tin_x ≤ tin_y ≤ tout_y ≤ tout_x**.

Tổ chức dữ liệu:

- Mảng **vector <int>** **g[N]** – lưu danh sách đỉnh kè,
- Mảng **int visit[N] = {0}** – đánh dấu đỉnh đã thăm,
- Mảng **int par[N]** – lưu quan hệ mốc nối trong cây,
- Mảng **vector <int>** **t** – lưu vết khi loang,
- Mảng **int tin[N]** – lưu thời điểm vào,
- Mảng **int tout[N]** – lưu thời điểm ra.

Xử lý:

Ghi nhận dữ liệu và chuẩn bị xử lý:

```
int n, m;
fi >> n >> m;
for (int i = 0; i < m; i++)
{
    int a, b;
    fi >> a >> b;
    a--, b--;
    g[a].push_back(b);
}
for (int i = 0; i < n; i++) par[i] = -1;
```

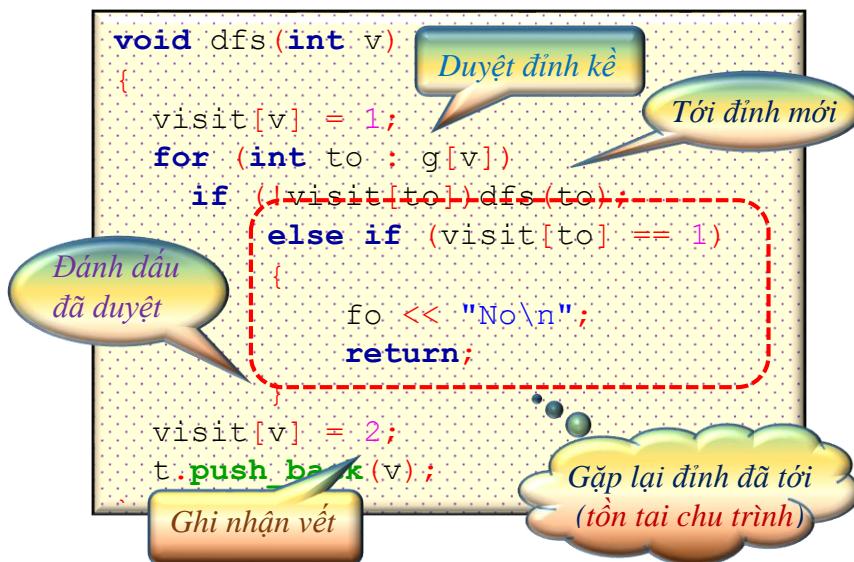
Ghi nhận đỉnh kè

Chưa có mốc nối

Lọc dữ liệu và ghi nhận vết loang:

```
for (int i = 0; i < n; i++)
    if (!visit[i]) dfs(i);
```

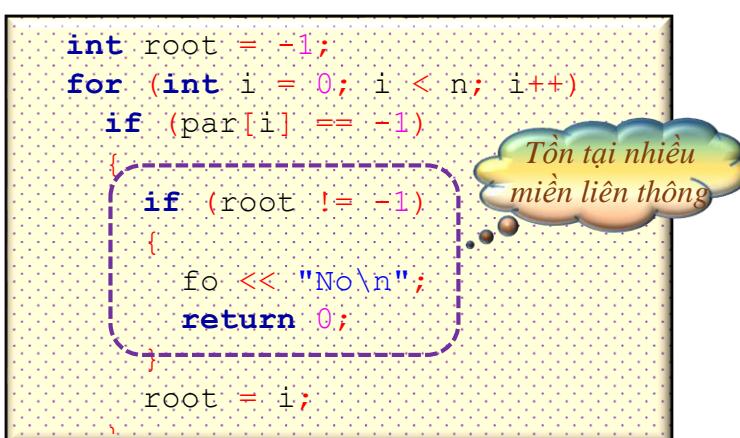
Hàm loang theo chiều sâu:



Ghi nhận cây:

```
for (int v : t)
    for (int to : g[v])
        if (par[to] == -1) par[to] = v;
```

Kiểm tra số miền liên thông và xác định gốc:



Tính thời điểm vào – ra cho mỗi nút:

```
void check_t(int v)
{
    tin[v] = tt++;
    for (int to : g[v]) check_t(to);
    tout[v] = tt++;
}
```

Kiểm tra việc bảo tồn trình tự quan hệ:

```
bool pr(int a, int b)
{
    return tin[a] <= tin[b] && tout[a] >= tout[b];
}
```

```
check_t(root);
for (int i = 0; i < n; i++)
    for (int to : g[i])
        if (!pr(i, to))
        {
            fo << "No\n";
            return 0;
        }
    
```

Không bảo tồn
trình tự

Dẫn xuất kết quả:

```
fo << "Yes\n";
for (int i = 0; i < n; i++)
{
    if (par[i] == -1) par[i]--;
    fo << par[i] + 1 << ' ';
}
```

Độ phức tạp của giải thuật: $O(n+m)$.

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
typedef int64_t ll;
ifstream fi ("startup.inp");
ofstream fo ("startup.out");
const int N = 5e5 + 7;
vector <int> g[N];
int visit[N]={0};
int par[N];
//set<pair<int,int>> s;
vector <int> t;

void dfs(int v)
{
    visit[v] = 1;
    for (int to : g[v])
        if (!visit[to])dfs(to);
        else if (visit[to] == 1)
        {
            fo << "No\n";
            return;
        }
    visit[v] = 2;
    t.push_back(v);
}
int tt = 0;
int tin[N], tout[N];

void check_t(int v)
{
    tin[v] = tt++;
    for (int to : g[v]) check_t(to);
    tout[v] = tt++;
}

bool pr(int a, int b)
{
    return tin[a] <= tin[b] && tout[a] >= tout[b];
}

int main()
{
    int n, m;
    fi >> n >> m;
    for (int i = 0; i < m; i++)
    {
        int a, b;
        fi >> a >> b;
        a--, b--;
        //if(s.insert({a,b}).second) g[a].push_back(b);
    }
}
```

```

        g[a].push_back(b);
    }
    for (int i = 0; i < n; i++) par[i] = -1;
    for (int i = 0; i < n; i++)
        if (!visit[i]) dfs(i);
    for (int v : t)
        for (int to : g[v])
            if (par[to] == -1) par[to] = v;

    int root = -1;
    for (int i = 0; i < n; i++)
        if (par[i] == -1)
    {
        if (root != -1)
        {
            fo << "No\n";
            return 0;
        }
        root = i;
    }
    check_t(root);
    for (int i = 0; i < n; i++)
        for (int to : g[i])
            if (!pr(i, to))
            {
                fo << "No\n";
                return 0;
            }

    fo << "Yes\n";
    for (int i = 0; i < n; i++)
    {
        if (par[i] == -1) par[i]--;
        fo << par[i] + 1 << ' ';
    }
    fo << endl;
    fo << "\nTime: " << clock() / (double)1000 << " sec";
}

```

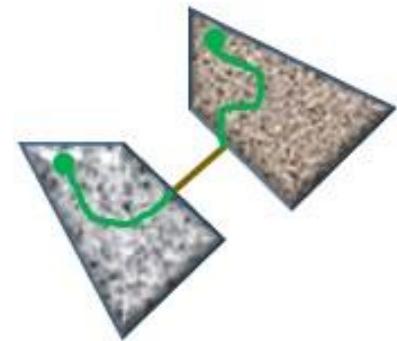


VZ18. BÃI SÌNH

Tên chương trình: MUD.CPP

Khu bảo tồn sinh thái vùng đất ngập mặn có n hòn đảo. Khi thủy triều rút toàn bộ khu bảo tồn sẽ là một bãi sinh lầy với các đảo nằm rải rác. Trên bản đồ, mỗi hòn đảo có hình đa giác lồi và không có đa giác lồi nào giao nhau, các đảo được đánh số từ 1 đến n . Đảo thứ i là đa giác lồi có k_i đỉnh, đỉnh thứ j có tọa độ $(x_{k_i,j}, y_{k_i,j})$, $j = 1, 2, \dots, k_i$. Các tọa độ đều nguyên.

Cơ sở cứu hộ và nuôi dưỡng động vật hoang dã được xây dựng ở đảo a . Người ta nhận được thông báo phải giải cứu một động vật quý hiếm bị kẹt ở đảo b . Việc di chuyển xuyên qua các hòn đảo không thành vấn đề, nhưng để vượt qua sinh lầy để tới đảo khác với các thiết bị lính kinh trên lưng là vô cùng vất vả, vì vậy bao giờ người ta cũng phải tìm cách đi sao cho tổng độ dài các đoạn lội bùn là ngắn nhất.



Hãy xác định tổng độ dài ngắn nhất của các đoạn lội bùn.

Dữ liệu: Vào từ file văn bản MUD.INP:

- ✚ Dòng đầu tiên chứa 3 số nguyên n , a và b ($1 \leq n \leq 200$, $1 \leq a, b \leq n$),
- ✚ Tiếp sau là n nhóm dữ liệu, nhóm i mô tả một hòn đảo thứ i :
 - Dòng đầu tiên trong nhóm chứa số nguyên k_i ($3 \leq k_i \leq 500$),
 - Mỗi dòng trong kí dòng tiếp theo chứa 2 số nguyên x và y xác định tọa độ một đỉnh ($|x|, |y| \leq 10^9$).
 - Các đỉnh được liệt kê theo chiều ngược kim đồng hồ và không có 3 đỉnh liên tiếp nằm trên một đường thẳng.

Kết quả: Đưa ra file văn bản MUD.OUT: một số thực với độ chính xác không ít hơn 9 chữ số sau dấu chấm thập phân.

Ví dụ:

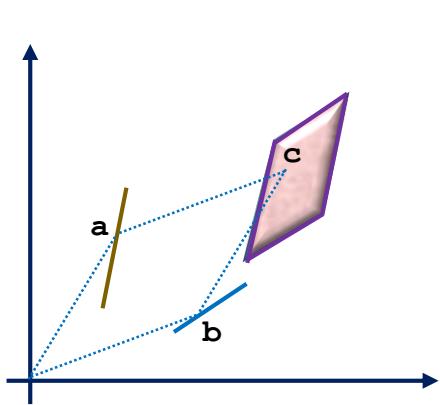
MUD.INP	MUD.OUT
<p>2 1 2 4 2 1 3 2 2 3 1 3 4 4 2 5 2 4 4 3 3</p>	<p>0.707106781186548</p>



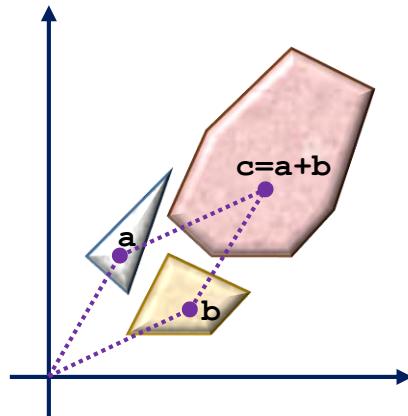
Giải thuật: Tổng Minkowski.

Cho 2 điểm trên mặt phẳng $\mathbf{a}(\mathbf{x}_a, \mathbf{y}_a)$ và $\mathbf{b}(\mathbf{x}_b, \mathbf{y}_b)$, tổng $\mathbf{a}+\mathbf{b}$ theo Minkowski là điểm $\mathbf{c}(\mathbf{x}_c, \mathbf{y}_c)$, trong đó $\mathbf{x}_c = \mathbf{x}_a + \mathbf{x}_b$, $\mathbf{y}_c = \mathbf{y}_a + \mathbf{y}_b$.

Tổng Minkowski của 2 tập \mathbf{A} và \mathbf{B} là tập $\mathbf{C} = \{\mathbf{a}+\mathbf{b}: \mathbf{a} \in \mathbf{A}, \mathbf{b} \in \mathbf{B}\}$.



Tổng 2 đoạn thẳng



Tổng 2 đa giác lồi

Có thể chứng minh được rằng tổng Minkowski của 2 đa giác lồi là một đa giác lồi.

Một trong số các ứng dụng của tổng Minkowski là tìm khoảng cách giữa 2 tập lồi.

Gọi $dist(\mathbf{A}, \mathbf{B})$ là khoảng cách giữa 2 tập \mathbf{A} và \mathbf{B} .

$$dist(\mathbf{A}, \mathbf{B}) = \min_{a \in A, b \in B} |a - b|$$

Xét trường hợp \mathbf{A}, \mathbf{B} – hai đa giác lồi. Dễ dàng xác định được \mathbf{B}' – hình *đối xứng qua tâm* của \mathbf{B} với *tâm đối xứng* là *góc tọa độ*.

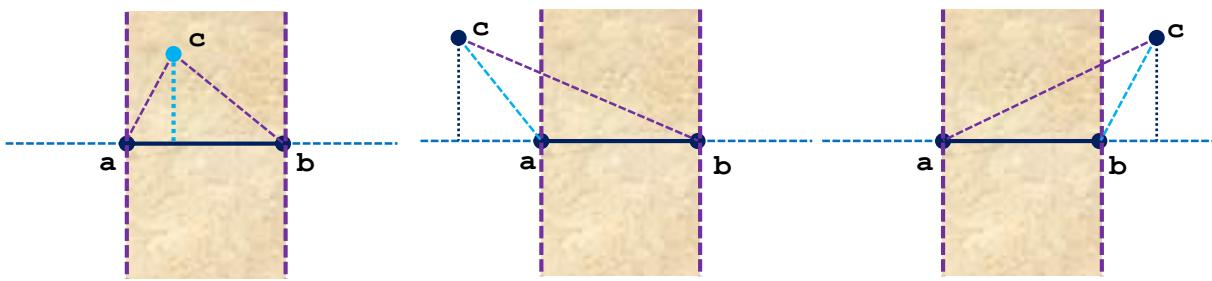
Gọi \mathbf{C} là tổng Minkowski của \mathbf{A} và \mathbf{B}' : $\mathbf{C} = \mathbf{A} + \mathbf{B}'$.

Ta có

$$dist(\mathbf{A}, \mathbf{B}) = \min_{a \in A, b \in B} |a - b| = \min_{a \in A, b \in B'} |a + b| = \min_{a \in A + B'} |c|$$

Từ đây suy ra khoảng cách giữa 2 đa giác lồi \mathbf{A}, \mathbf{B} bằng *khoảng cách từ gốc tọa độ tới các cạnh của $\mathbf{A} + \mathbf{B}'$* hoặc *bằng 0 nếu \mathbf{A} và \mathbf{B} giao nhau* (trong trường hợp này $\mathbf{A} + \mathbf{B}'$ chứa gốc tọa độ).

Như vậy ta phải tính khoảng cách từ *một điểm* **c** tới *đoạn thẳng* **[a,b]**.



Có hai trường hợp:

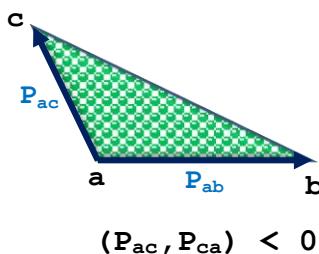
- Chân đường vuông góc hạ từ **c** xuống đường thẳng chứa **a** và **b** nằm trong đoạn **[a,b]**. Khi đó khoảng cách từ **c** tới **[a,b]** sẽ là

$$\frac{|(y_b - y_a) \times x_c - (x_b - x_a) \times y_c + x_b \times y_a - x_a \times y_b|}{\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}}$$

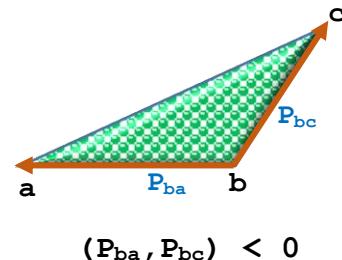
- Chân đường vuông góc hạ từ **c** xuống đường thẳng chứa **a** và **b** nằm ngoài đoạn **[a,b]**. Khi đó khoảng cách từ **c** tới **[a,b]** sẽ là *min* khoảng cách từ **c** tới **a** và khoảng cách từ **c** tới **b**.

Dấu hiệu nhận biết trường hợp 2:

Có nhiều cách để kiểm tra, một trong số đó là tính tích vô hướng



$$(P_{ac}, P_{ca}) < 0$$



$$(P_{ba}, P_{bc}) < 0$$

$$(x_b - x_a)(x_c - x_a) + (y_b - y_a)(y_c - y_a) < 0$$

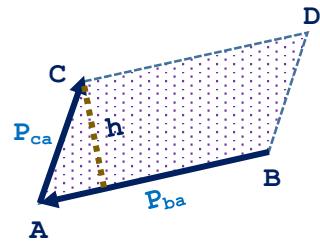
Lưu ý:

Có nhiều công thức tương đương dẫn xuất độ dài đường cao của tam giác. Tồn tại các công thức dễ nhớ và dễ lập trình. Một trong những công thức đó là biểu diễn độ dài đường cao thông qua *tích giả vô hướng* (*pseudoscalar*) 2 véc tơ cạnh.

Xét $\mathbf{P} = (\mathbf{x}_p, \mathbf{y}_p)$, $\mathbf{Q} = (\mathbf{x}_q, \mathbf{y}_q)$ – 2 véc tơ trên mặt phẳng. Ký hiệu $\mathbf{P} \wedge \mathbf{Q}$ – tích giả vô hướng của \mathbf{P} và \mathbf{Q} .

$$\mathbf{P} \wedge \mathbf{Q} = \begin{vmatrix} x_p & y_p \\ x_q & y_q \end{vmatrix} = \mathbf{x}_p \times \mathbf{y}_q - \mathbf{y}_p \times \mathbf{x}_q$$

Tích giả vô hướng $\mathbf{P}_{ba} \wedge \mathbf{P}_{ca}$ cho diện tích có hướng của hình bình hành ABCD.



Độ cao h từ đỉnh **C** của hình bình hành **ABCD** sẽ là:

$$h = \frac{\mathbf{P}_{ba} \wedge \mathbf{P}_{ca}}{\text{length}(\mathbf{P}_{ba})}$$

Khi tính toán theo công thức trên cần chú ý tính phản đối xứng của tích giả vô hướng, tức là $\mathbf{P}_{ba} \wedge \mathbf{P}_{ca} = -(\mathbf{P}_{ca} \wedge \mathbf{P}_{ba})$.

Với số lượng đa giác không nhiều ($n \leq 200$) ta có thể tính khoảng cách giữa 2 đa giác **i** và **j** ($1 \leq i, j \leq n$) với mọi **i**, **j** và bằng quy hoạch động – xác định khoảng cách ngắn nhất giữa **a** và **b**.

Tổ chức dữ liệu:

- Bán ghi **struct point** với phép xử lý gắn kèm xử lý cặp điểm: +, -, < và tính độ dài đoạn thẳng,
- Mảng 2 chiều **vector<vector<point>> polys** – lưu đỉnh các đa giác,
- Mảng 2 chiều **vector<vector<point>> ipolys** – lưu đỉnh các đa giác đối xứng qua gốc tọa độ,
- Mảng 2 chiều **vector<vector<double>> graph(n, vector<double>(n))** – ghi nhận khoảng cách ngắn nhất trực tiếp giữa 2 đảo,
- Mảng **vector<bool> used(n)** – đánh dấu đảo đã xét,
- Mảng **vector<double> dist(n, 1e100)** – ghi nhận khoảng cách ngắn nhất từ **a**.

Xử lý:

Nhập tọa độ đỉnh các đa giác và chuẩn bị dữ liệu để tính khoảng cách:

```
for (int i = 0; i < n; ++i)
{
    polys.push_back({});
    ipolys.push_back({});
    int k;
    fi >> k;
    int mem = 0;
    for (int j = 0; j < k; ++j)
    {
        int x, y;
        fi >> x >> y;
        polys.back().push_back(point(x, y));
        ipolys.back().push_back(point(-x, -y));
        if (polys[i][j] < polys[i][mem]) mem = j;
    }
    rotate(polys[i].begin(), polys[i].begin() + mem, polys[i].end());
    mem = 0;
    for (int j = 0; j < k; ++j)
        if (ipolys[i][j] < ipolys[i][mem]) mem = j;
    rotate(ipolys[i].begin(), ipolys[i].begin() + mem, ipolys[i].end());
}
```

Khởi tạo
dòng mới

Tìm đỉnh trái
nhất và thấp nhất

Danh số lại các
đỉnh

Các phép tính gắn với điểm

```
struct point
{
    int x, y;

    point(int _x, int _y) : x(_x), y(_y) {}

    point operator+(point const& other) const
    {
        return point(x + other.x, y + other.y);
    }

    point operator-(point const& other) const
    {
        return point(x - other.x, y - other.y);
    }

    double len() const
    {
        return sqrt((ll)x * x + (ll)y * y);
    }

    bool operator<(point const& other) const
    {
        return x < other.x || (x == other.x && y < other.y);
    }
};
```

Tính diện tích hình bình hành theo tích giả vô hướng:

```
ll p_scalar(point const& p1, point const& p2)
{
    return (ll) p1.x * p2.y - (ll) p1.y * p2.x;
}
```

Tính khoảng cách từ điểm **C** tới đoạn thẳng **AB**:

```
vector<double> dist(n, 1e100);
dist[a] = 0;
vector<bool> used(n, 0);

for (int i = 0; i < n; ++i)
{
    int mem = -1;
    for (int j = 0; j < n; ++j)
        if (!used[j])
            if (mem == -1 || dist[mem] > dist[j]) mem = j;
    used[mem] = true;
    if (mem == b)
    {
        fo<<fixed<<setprecision(15)<<dist[b]<<'\n';
        fo<<"\nTime: "<<clock()/(double)1000<<" sec";
        return 0;
    }
    for (int j = 0; j < n; ++j)
        dist[j] = min(dist[j], dist[mem] + graph[mem][j]);
}
```

Tính khoảng cách ngắn nhất từ gốc tọa độ tới **A+B'**:

```

double calc_dist(vector<point> const& poly1, vector<point> const& poly2)
{
    int c1 = 0, c2 = 0;
    point cur = poly1[0] + poly2[0];
    double ans = 1e100;
    while (c1 < poly1.size() || c2 < poly2.size())
    {
        point shift(0, 0);
        int next1 = c1 + 1;
        if (next1 == poly1.size()) next1 = 0;
        int next2 = c2 + 1;
        if (next2 == poly2.size()) next2 = 0;

        if (c1 == poly1.size() || (c2 == poly2.size() &&
            p_scalar(poly1[next1] - poly1[c1], poly2[next2] - poly2[c2]) < 0))
        {
            shift = poly2[next2] - poly2[c2];
            c2++;
        }
        else
        {
            shift = poly1[next1] - poly1[c1];
            c1++;
        }
        ans = min(ans, get_dist(cur, cur + shift, point(0, 0)));
        cur = cur + shift;
    }
    return ans;
}

```

Tính độ dài đường đi ngắn nhất từ **a** tới **b**:

```

double get_dist(point const& a, point const& b, point const& c)
{
    if (scalar(b - a, c - a) < 0 || scalar(a - b, c - b) < 0)
        return min((c - a).len(), (c - b).len());
    return abs(p_scalar(b - a, c - a) / (b - a).len());
}

```

*Chân đường cao nằm
ngoài đoạn thẳng*

Độ phức tạp của giải thuật: $O(k \times n^2)$.

Chương trình

```
#include <bits/stdc++.h>
#define ff first
#define ss second

using namespace std;
typedef long long ll;
typedef long double ld;
typedef pair<int, int> pii;
ifstream fi ("mud.inp");
//ifstream fi ("85");
ofstream fo ("mud.out");

struct point
{
    int x, y;

    point (int _x, int _y) : x(_x), y(_y) {}

    point operator+(point const& other) const
    {
        return point(x + other.x, y + other.y);
    }

    point operator-(point const& other) const
    {
        return point(x - other.x, y - other.y);
    }

    double len() const
    {
        return sqrt((ll) x * x + (ll) y * y);
    }

    bool operator<(point const& other) const
    {
        return x < other.x || (x == other.x && y < other.y);
    }
};

ll p_scalar(point const& p1, point const& p2)
{
    return (ll) p1.x * p2.y - (ll) p1.y * p2.x;
}

ll scalar(point const& p1, point const& p2)
{
    return (ll) p1.x * p2.x + (ll) p1.y * p2.y;
}

double get_dist(point const& a, point const& b, point const& c)
{
    if (scalar(b - a, c - a) < 0 || scalar(a - b, c - b) < 0)
        return min((c - a).len(), (c - b).len());
    return abs(p_scalar (b - a, c - a) / (b - a).len());
}
```

```

double calc_dist(vector<point> const& poly1, vector<point> const& poly2)
{
    int c1 = 0, c2 = 0;
    point cur = poly1[0] + poly2[0];
    double ans = 1e100;
    while (c1 < poly1.size() || c2 < poly2.size())
    {
        point shift(0, 0);
        int next1 = c1 + 1;
        if (next1 == poly1.size()) next1 = 0;
        int next2 = c2 + 1;
        if (next2 == poly2.size()) next2 = 0;

        if (c1==poly1.size() || (c2!=poly2.size() &&
            p_scalar(poly1[next1]-poly1[c1], poly2[next2]-poly2[c2])<0))
        {
            shift = poly2[next2] - poly2[c2];
            c2++;
        } else
        {
            shift = poly1[next1] - poly1[c1];
            c1++;
        }
        ans = min(ans, get_dist(cur, cur + shift, point(0, 0)));
        cur = cur + shift;
    }
    return ans;
}

int main()
{
    int n, a, b;
    fi >> n >> a >> b;
    --a; --b;
    vector<vector<double>> graph(n, vector<double>(n));
    vector<vector<point>> polys;
    vector<vector<point>> ipolys;
    for (int i = 0; i < n; ++i)
    {
        polys.push_back({});
        ipolys.push_back({});
        int k;
        fi >> k;
        int mem = 0;
        for (int j = 0; j < k; ++j)
        {
            int x, y;
            fi >> x >> y;
            polys.back().push_back(point(x, y));
            ipolys.back().push_back(point(-x, -y));
            if (polys[i][j] < polys[i][mem]) mem = j;
        }
        rotate(polys[i].begin(), polys[i].begin() + mem, polys[i].end());
        mem = 0;
        for (int j = 0; j < k; ++j)
            if (ipolys[i][j] < ipolys[i][mem]) mem = j;
    }
}

```

```

    rotate(ipolys[i].begin(), ipolys[i].begin() + mem,
ipolys[i].end());
}

for (int i = 0; i < n; ++i)
    for (int j = i + 1; j < n; ++j)
        graph[i][j] = graph[j][i] = calc_dist(polys[i], ipolys[j]);

vector<double> dist(n, 1e100);
dist[a] = 0;
vector<bool> used(n, 0);

for (int i = 0; i < n; ++i)
{
    int mem = -1;
    for (int j = 0; j < n; ++j)
        if (!used[j])
            if (mem == -1 || dist[mem] > dist[j]) mem = j;

    used[mem] = true;
    if (mem == b) {
        fo<<fixed<<setprecision(15) << dist[b] << '\n';
        fo<<"\nTime: "<<clock() / (double)1000<<" sec";
        return 0;
    }
    for (int j = 0; j < n; ++j)
        dist[j] = min(dist[j], dist[mem] + graph[mem][j]);
}
}

```



Các tàu đánh cá được trang bị hệ thống ghi nhận tọa độ điểm đánh bắt và một thông tin khác như loại hải sản, số lượng, v.v... Thông tin được lưu lại dưới dạng xâu các ký tự số thập phân.

Tuy vậy, do điều kiện khắc nghiệt ngoài biển khơi, độ ẩm và độ mặn trong không khí cao, giông lốc, bão tố . . . nhiều khi thông tin ghi lại bị nhiễu và không còn chính xác. Loại nhiễu thường gặp nhất là một chữ số bị sai lệch hoặc bị bỏ trống.

Để khắc phục trường hợp này người ta bổ sung thêm vào cuối dòng thông tin một chữ số thập phân với quy tắc tính như sau:

- Đánh số các ký tự của xâu s ban đầu từ cuối về đầu bắt đầu từ 0,
- Tạo xâu z theo quy tắc sau:
 - ✳ $z_i = s_i$ nếu i lẻ,
 - ✳ $z_i = \text{tổng các chữ số của } 2 \times s_i$ nếu i chẵn, ví dụ, với $s_4 = 7, 2 \times s_4 = 14 \rightarrow z_4 = 1 + 4 = 5,$
- Tính tổng các chữ số của z , nhân tổng nhận được với 9, tách ra chữ số hàng đơn vị.
- Chữ số này được viết thêm vào cuối xâu s ban đầu để nhận được xâu y phục vụ lưu trữ hoặc truyền đi.

Ví dụ, xâu ban đầu s là “**5246**”, ta có:

Xâu ban đầu: **5 2 4 6**

Xâu z : **5 4 4 3**

Tổng các chữ số của z : **$5+4+4+3 = 16$**

Chữ số cần bổ sung: **$(16 \times 9) \% 10 = 4$**

Xâu kết quả r : **5 2 4 6 4**

Cho xâu kết quả r độ dài n chứa một ký tự x , các ký tự còn lại là chữ số thập phân. Hãy xác định chữ số được đánh dấu ở vị trí x .

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

- ✳ Dòng đầu tiên chứa số nguyên n ($1 < n \leq 100$),
- ✳ Dòng thứ 2 chứa xâu r độ dài n .

Kết quả: Đưa ra file văn bản SH_CODE.OUT một số nguyên – chữ số ở vị trí đánh dấu x .

Ví dụ:

SH_CODE.INP
5
x2464

SH_CODE.OUT
5



Giải thuật; Kỹ thuật tổ chức dữ liệu.

Gọi **s** – xâu dữ liệu nhập vào.

Xác định:

- Giá trị chữ số cuối cùng **1d = s_{n-1}-48** (**1d** có thể nhận giá trị lớn hơn 10 nếu vị trí cuối cùng được đánh dấu **x**),
- **ix** – Chỉ số của vị trí **x**.

Xóa ký tự cuối cùng của s và đảo ngược xâu **s**,

Tính lại vị trí của **x** trong **s** nếu nó không ở vị trí cuối cùng trong xâu ban đầu.

Tính **sum** – tổng các ký tự của **z**, các ký tự thay đổi giá trị sẽ nằm ở **các vị trí cần tìm**,

Trong quá trình tính tổng:

- Bỏ qua vị trí của ký tự ‘**x**’,
- Chỉ cần giữ lại **chữ số cuối cùng**.

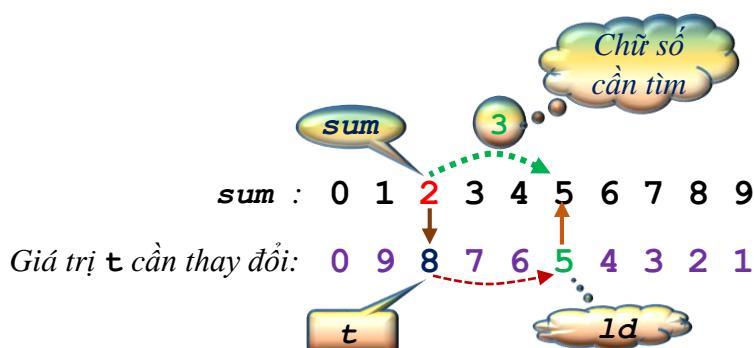
Xác định **t** – chữ số cuối cùng cần bổ sung khi bỏ qua số ở vị trí đánh dấu **x**.

Dẫn xuất kết quả:

Phân biệt 3 trường hợp:

x ở vị trí cuối cùng trong xâu dữ liệu nhập vào: Chữ số cuối cùng tính được là kết quả cần tìm,

x không phải ở vị trí cuối cùng và có vị trí lẻ: ta có



Dễ dàng thấy rằng **t-1d** xác định **khoảng cách** giữa **sum** tìm được với chữ số cần tìm **theo chiều ngược lại** (cần giảm tổng – tăng giá trị chữ số và ngược lại) nếu xác định các chữ số theo **vòng tròn**, sau **9** là **0** và trước **0** là **9**.

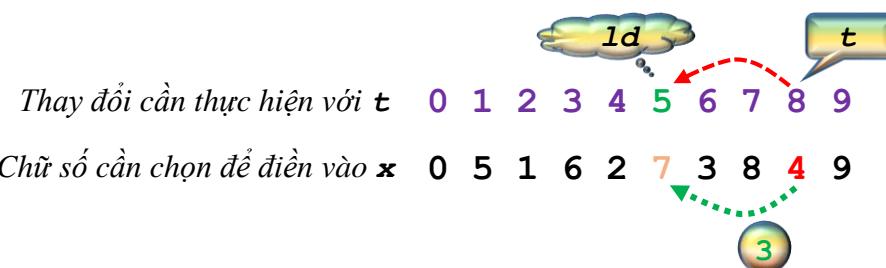
Trường hợp \mathbf{x} ở vị trí chẵn:

Chữ số được chọn: 0 1 2 3 4 5 6 7 8 9

\mathbf{z}_i : 0 2 4 6 8 1 3 5 7 9

Sự thay đổi của t : 0 8 6 4 2 9 7 5 3 1

Từ đây ta có thể lập bảng tra cứu để tìm \mathbf{x} :



Lưu ý:

- Tương tự như ở trường hợp trước, dãy chữ số cần chọn được xử lý như dãy dữ liệu vòng tròn,
- \mathbf{x} không phải ở vị trí cuối cùng và có vị trí lẻ: cũng có thể tổ chức bảng tra cứu tương tự như trên, nhưng do dãy chữ số cần chọn trong trường hợp này là **dãy đơn điệu** nên có thể dùng thuật toán thay cho bảng tra cứu.

Độ phức tạp của giải thuật: $O(n)$.

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("sh_code.inp");
ofstream fo ("sh_code.out");
string s;
int n, ix, ld, t, sum=0, ans;
int d[10]={0, 5, 1, 6, 2, 7, 3, 8, 4, 9};

int main()
{
    fi>>n>>s;
    ix=s.find('x');
    ld=s[n-1]-48;
    s.erase(n-1,1);
    reverse(s.begin(),s.end());
    --n; if(ix<n) ix=n-2-ix;
    for(int i=0; i<n; ++i)
    {
        if(s[i]=='x') continue;
        t=s[i]-48;
        if(i&1) sum+=t; else sum+=(2*t)%10+(2*t)/10;
        sum%=10;
    }
    t=sum*9%10;
    if(ix>=n) ans=t;
    else
        if(ix&1) ans=d[(t-ld+10)%10]; else ans=(t-
ld+10)%10;
    fo<<ans;
}
```



Ngạn ngữ có câu “*Buôn tàu buôn bè không bằng ăn dè hà tiện*”. Quán triệt tư tưởng này nên khi được giao nhiệm vụ soạn bài đề xuất cho kỳ thi Olympic Tin học của trường Alice quyết định sẽ hoàn thành tốt công việc được giao với chi phí chất xám bỏ ra nhỏ nhất có thể.

Alice có nhiệm vụ phải soạn n bài đưa cho Ban duyệt để chọn. Mỗi bài có độ khó xác định bởi số nguyên dương. Cô biết rằng Ban duyệt để có thể loại một số bài, nhưng số bài bị loại không bao giờ quá k . Những bài được chọn phải có tổng độ khó không ít hơn x thì cuộc thi mới có giá trị.

Bài càng khó thì công sức bỏ ra càng nhiều, vì vậy Alice cố gắng nghĩ ra n bài sao cho tổng độ khó là nhỏ nhất và dù Ban duyệt để loại bài đề xuất như thế nào thì tổng độ khó của các bài được chọn vẫn không ít hơn x .

Hãy xác định tổng độ khó các bài đề xuất của Alice.

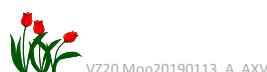
Dữ liệu: Vào từ file văn bản PROVIDENT.INP gồm một dòng chứa 3 số nguyên n , k và x ($2 \leq n \leq 10^9$, $1 \leq k < n$, $1 \leq x \leq 10^9$).

Kết quả: Đưa ra file văn bản PROVIDENT.OUT một số nguyên – tổng độ khó nhỏ nhất của các bài đề xuất.

Ví dụ:

PROVIDENT.INP
3 1 5

PROVIDENT.OUT
8



Giải thuật; Cơ sở lập trình .

Trường hợp xấu nhất có thể xảy ra: **n-k** bài được chọn có độ khó nhỏ hơn hoặc bằng độ khó các bài bị loại.

Ở phương án ra đề tối ưu, tổng độ khó các bài được chọn đúng bằng **x** trong trường số xấu nhất nói trên.

Bài dễ nhất sẽ có độ khó **y = ⌊x / (n-k)⌋**.

Nếu $\lfloor x / (n-k) \rfloor * (n-k) < x$ thì một số bài phải có độ khó **y+1**, trong trường hợp ngược lại – mọi bài đều có độ khó **y**.

Gọi **z** – độ khó của bài khó nhất trong số được chọn ở trường hợp xấu nhất.

Các bài còn lại: có độ khó bằng **z**.

Độ phức tạp của giải thuật: O(1).

Chương trình

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

int n,k,x,ans;

int main()
{
    fi>>n>>k>>x;
    int r=n-k;
    int cpl=x/r;
    ans=x+k*(cpl+(x-cpl*r > 0));
    fo<<ans;
}
```



Nhân dịp kỷ niệm một năm ngày khai trương hệ thống siêu thị bán lẻ Trăng rằm Ban giám đốc quyết định tặng quà kỷ niệm cho các khách hàng đã tới mua sắm trong năm. Dựa theo tổng số tiền khách hàng đã mua sắm tại siêu thị, món quà có giá trị nhất sẽ giành cho khách có mức chi trung bình nhất, tức là nếu sắp xếp khách hàng theo chiều tăng dần tổng số tiền họ đã chi, người đứng ở giữa trong dãy sẽ nhận được món quà có giá trị nhất. Vị trí ở giữa được gọi là trung vị.

Trên bàn họp có báo cáo của bộ phận kế toán, trong đó có thông tin về mức chi của n khách hàng, người thứ i đã chi số tiền a_i , $i = 1 \dots n$. Nếu n lẻ, sẽ có đúng 1 người ở trung vị. Nếu n chẵn, có 2 người có thể coi là ở trung vị, người ta sẽ chọn người có mức chi thấp hơn trong số 2 người đó để trao phần quà giá trị nhất.

Sau khi loại bỏ khách đã xác định được qua cân trao, người ta lại tìm người trung vị trong số những người còn lại để trao món quà tiếp theo kém giá trị hơn một chút và cứ như thế cho đến khi mọi khách hàng trong danh sách đều xác định được phần quà của mình.

Hãy đưa ra số tiền mỗi khách hàng đã chi theo thứ tự giảm dần của món quà họ được tặng.

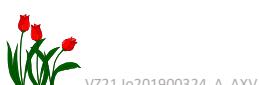
Dữ liệu: Vào từ file văn bản MEDIAN.INP:

- ✚ Dòng đầu tiên chứa số nguyên n ($1 \leq n \leq 10^5$),
- ✚ Dòng thứ 2 chứa n số nguyên a_1, a_2, \dots, a_n ($1 \leq a_i \leq 10^9$, $i = 1 \dots n$).

Kết quả: Đưa ra file văn bản MEDIAN.OUT trên một dòng số tiền mỗi khách hàng đã chi theo thứ tự giảm dần của món quà họ được tặng.

Ví dụ:

MEDIAN.INP	MEDIAN.OUT
4	2 2 1 4
4 2 2 1	



VZ21_10201900324_A_AXV

Giải thuật; Phương pháp 2 con trỏ.

Đánh số các phần tử của dữ liệu vào bắt đầu từ 0,

Gọi $m = n/2$,

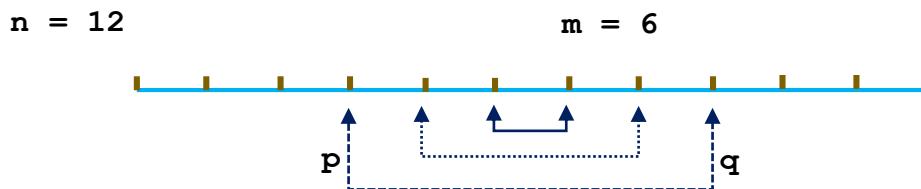
Sắp xếp a_i theo thứ tự tăng dần.

Trường hợp n chẵn:

Đưa ra phần tử cuối của nửa trái và phần tử đầu của nửa phải,

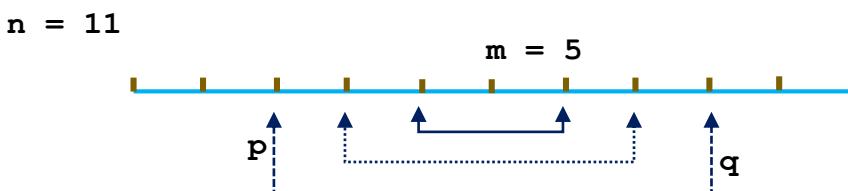
Cập nhật lại các con trỏ chỉ tới cuối của nửa trái và đầu của nửa phải.

Hai bước trên được thực hiện m lần.



Trường hợp n lẻ:

Dẫn xuất phần tử $a[m]$ và đưa về trường hợp n chẵn.



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

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("median.inp");
ofstream fo ("median.out");
int n,m,p,q;

int main()
{
    fi>>n;
    vector<int> a(n);
    for(int &i:a) fi>>i;
    sort(a.begin(),a.end());
    m = n>>1; p=m-1;q=m;
    if(n&1) fo<<a[q++]<<' ';
    for(int i=0; i<m; ++i) fo<<a[p--]<<' '<<a[q++]<<' ';
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



VZ22. BÁO CÁO

Tên chương trình: RAPORTS.CPP

Theo hiệp định thương mại ký giữa 2 nước, mỗi mặt hàng xuất sang nhau phải có báo cáo về nguồn gốc xuất xứ và các tham số về chất lượng sản phẩm. Báo cáo phải viết trên hai thứ tiếng và ghi song song lên băng giấy cuộn tròn để in lại và dán trên bao bì.

Băng giấy có độ rộng w ô. Mỗi nhóm ký tự liên tiếp nhau không chứa dấu cách được gọi là một từ. Ở ngôn ngữ thứ nhất báo cáo có n từ, từ thứ i có độ dài a_i , $i = 1 \dots n$, ở ngôn ngữ thứ 2 báo cáo có m từ, từ thứ j có độ dài b_j , $j = 1 \dots m$.

Người ta kẻ một đường thẳng đọc theo chiều dài chia băng giấy thành 2 phần, mỗi dòng sẽ bị chia thành 2 phần và đều có độ rộng là số nguyên các ô. Nội dung báo cáo ở ngôn ngữ thứ nhất được viết ở phần bên trái, mỗi từ phải nằm gọn trong một dòng, mỗi dòng có thể chứa nhiều từ, hai từ liên tiếp trên cùng dòng phải cách nhau một ô trống. Nội dung báo cáo ở ngôn ngữ thứ hai được viết ở phần bên phải theo quy tắc tương tự.

Với cách kẻ đường thẳng phân chia hợp lý độ dài cần sử dụng của băng giấy, tức là số dòng cần sử dụng, sẽ là ngắn nhất.

Hãy xác định độ dài ngắn nhất phải có.

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

- ✚ Dòng đầu tiên chứa 3 số nguyên w , n và m ($1 \leq w \leq 10^9$, $1 \leq n, m \leq 10^5$),
- ✚ Dòng thứ 2 chứa n số nguyên a_1, a_2, \dots, a_n ($1 \leq a_i \leq 10^9$ với mọi i),
- ✚ Dòng thứ 3 chứa m số nguyên b_1, b_2, \dots, b_m ($1 \leq b_j \leq 10^9$ với mọi j).

Dữ liệu đảm bảo có cách chia băng giấy.

Kết quả: Đưa ra file văn bản RAPORTS.OUT một số nguyên – độ dài ngắn nhất tìm được.

Ví dụ:

RAPORTS.INP
15 6 6
2 2 2 3 2 2
3 3 5 2 4 3

RAPORTS.OUT
3



VZ22.lm201900324_B_AXV

Giải thuật: Tìm kiếm nhị phân.

Nhận xét:

- Độ rộng của mỗi phần phải không nhỏ hơn độ dài của từ dài nhất trong phần tương ứng,
- Khi dịch đường phân chia sang phải số dòng của phần I *không tăng*, số dòng ở phần II *không giảm*.

Như vậy có thể sử dụng tìm kiếm nhị phân để xác định vị trí của đường phân chia.

Việc tính số dòng ở mỗi phần hoàn toàn tương tự như nhau về mặt giải thuật và hàm xử lý cũng đủ ngắn vì vậy không nhất thiết phải xây dựng một hàm tổng quát, thay vào đó – tạo 2 hàm riêng tính số dòng cho từng phần để giảm thiểu số lượng tham số cần truyền.

Trường hợp lý tưởng: Số dòng của 2 phần bằng nhau,

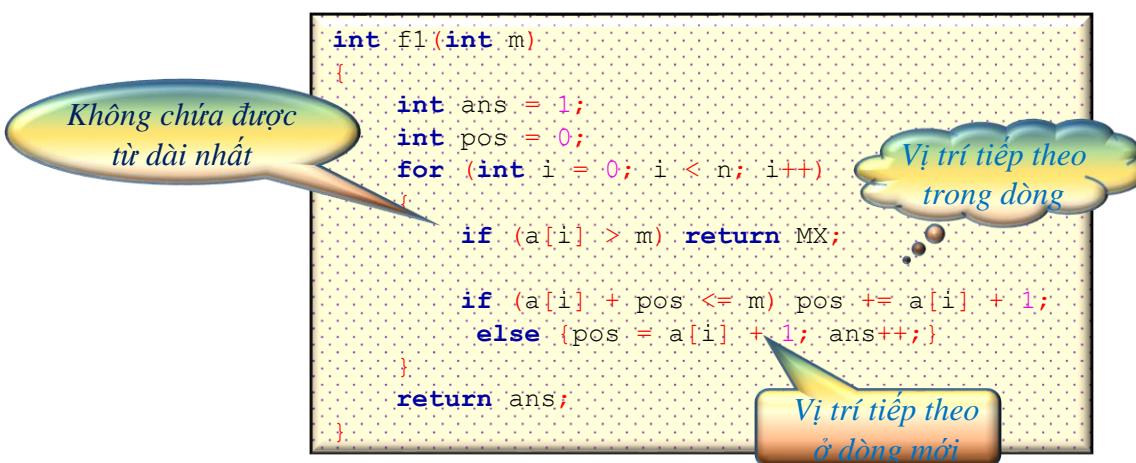
Nếu không thể đảm bảo số dòng 2 phần bằng nhau: lựa chọn kết quả tốt nhất giữa 2 phương án lân cận từ kết quả tìm kiếm nhị phân.

Việc tính lại số dòng ở mỗi phần để tìm độ dài ngắn nhất của tờ giấy sẽ đơn giản hơn việc lưu các kết quả trung gian nhận được bởi tìm kiếm nhị phân.

Tổ chức dữ liệu: Hai mảng nguyên lưu độ dài các từ ở mỗi phần.

Xử lý:

Hàm tính số dòng cho phần bên trái:



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

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
const int MX=1e9;
int n, w, k;
vector<int> a, b;

int f1(int m)
{
    int ans = 1;
    int pos = 0;
    for (int i = 0; i < n; i++)
    {
        if (a[i] > m) return MX;
        if (a[i] + pos <= m) pos += a[i] + 1;
        else { pos = a[i] + 1; ans++; }
    }
    return ans;
}

int f2(int m)
{
    int ans = 1;
    int pos = 0;
    for (int i = 0; i < k; i++)
    {
        if (b[i] > m) return MX;
        if (b[i] + pos <= m) pos += b[i] + 1;
        else { pos = b[i] + 1; ans++; }
    }
    return ans;
}

int main()
{
    freopen("raports.inp", "r", stdin);
    freopen("raports.out", "w", stdout);
    ios_base::sync_with_stdio(0);
    cin >> w >> n >> k;
    a.resize(n); b.resize(k);
    for (int i = 0; i < n; i++) cin >> a[i];
    for (int i = 0; i < k; i++) cin >> b[i];
    int l = 0;
    int r = w;
    while (l < r - 1)
    {
        int m = (l + r) / 2;
        if (f1(m) < f2(w - m)) r = m;
        else l = m;
    }
    cout << min(max(f1(l), f2(w - l)), max(f1(l + 1), f2(w - l - 1)));
    cout << "\nTime: " << clock() / (double)1000 << " sec";
}
```



VZ23. GIẢI MÃ

Tên chương trình: DECODING.CPP

Trong phần vui chơi giao lưu của một trại hè Tin học các đội phải đi tìm phần thưởng do Ban tổ chức cất dấu ở một số nơi. Chìa khóa chỉ đường đến nơi cần tìm là một dòng thông báo chỉ chứa các ký tự la tinh thường và hoa được mã hóa theo quy tắc sau:

- ✚ Chọn chữ số d trước khi mã hóa ($0 \leq d \leq 9$),
- ✚ Duyệt các ký tự của thông báo từ trái sang phải,
- ✚ Nếu ký tự đang xét có mã ASCII là số có 2 chữ số thì viết thêm chữ số d vào vị trí bắt kỵ vào số 2 chữ số đang xét để có 3 chữ số và bổ sung 3 chữ số đó vào cuối dòng mã hóa, ví dụ ký tự đang xét là ‘**a**’ (mã ASCII là 97), $d=3$, các chữ số bổ sung vào cuối dòng mã hóa có thể là 397 hoặc 937 hay 973,
- ✚ Nếu ký tự đang xét có mã ASCII là số có 3 chữ số thì bổ sung 3 chữ số đó vào cuối dòng mã hóa.

Các đội nhận dòng thông tin đã mã hóa nhưng không được biết về chữ số d . Thông tin để xác định đường đi là số lượng các xâu nguồn chưa mã hóa khác nhau có thể mã hóa thành xâu được trao. Số lượng xâu nguồn có thể rất lớn, vì vậy chỉ cần tính theo mô đun 10^9+7 .

Dữ liệu: Vào từ file văn bản DECODING.INP gồm một dòng ghi xâu **s** khác rỗng chỉ chứa các ký chữ số thập phân, độ dài xâu chia hết cho 3 và không vượt quá 10^5 .

Kết quả: Đưa ra file văn bản DECODING.OUT một số nguyên – số lượng xâu nguồn theo mô đun 10^9+7 .

Ví dụ:

DECODING.INP	DECODING.OUT
988	2



Giải thuật; Kỹ thuật tổ chức dữ liệu.

Các ký tự từ ‘d’ đến ‘z’ (có mã ASCII từ 100 đến 122) không bị mã hóa,

Các ký tự bị mã hóa có mã ASCII từ 65 đến 90 hoặc từ 97 đến 99.

Chia xâu **s** thành các nhóm 3 chữ số liên tiếp, nhóm thứ **i** sẽ tương ứng với ký tự thứ **i** trong xâu nguồn.

Nếu nhóm **i** tạo thành số ngoài đoạn [100, 122]:

- Tìm chữ số **d** cần loại bỏ để nhận được mã ASCII của ký tự la tinh,
- Ghi nhận các **d** tìm được cùng với số ký tự khác nhau có thể là nguồn.

Nếu một giá trị **d** có mặt trong ghi nhận ở mọi ký tự cần mã hóa, giá trị đó có thể sử dụng để mã hóa thông tin.

Với mỗi giá trị **d** có thể sử dụng: tính số lượng xâu khác nhau có thể là nguồn.

Tổng hợp các số lượng tính được ta có kết quả cần tìm.

Tổ chức dữ liệu:

- Mảng **vector<int>** used – lưu số ký tự được mã hóa với mỗi **d**,
- Mảng **vector<int>** digits – lưu các **d** có thể chọn được để mã hóa,
- Mảng **vector<char>** result – Đánh dấu các vị trí ký tự không bị mã hóa,
- Mảng **vector< map<int, set<char>>>** options – lưu khả năng các ký tự nguồn ứng với mỗi **d** có thể chọn cho ký tự đang xét.

Xử lý:

Ghi nhận thông tin từ ký tự được mã hóa:

```
map<int, set<char>> findOptions( string s)
{
    int c1 = stoi(s.substr(1));
    int c2 = (s[0] - '0') * 10 + s[2] - '0';
    int c3 = stoi(s.substr(0, 2));
    map<int, set<char>> ans;

    if (isValid(c1)) ans[(int)(s[0] - '0')].insert((char)c1);
    if (isValid(c2)) ans[(int)(s[1] - '0')].insert((char)c2);
    if (isValid(c3)) ans[(int)(s[2] - '0')].insert((char)c3);
    return ans;
}
```

Ghi nhận khả năng có thể được mã hóa của từng ký tự:

```

bool allThree = true;
int three = 0;
for (int i = 0; i < (int)s.size(); i += 3)
{
    int c = stoi(s.substr(i, 3));
    if (100 <= c && c <= 122)
    {
        result[i / 3] = (char)c;
        three++;
    }
    else
    {
        allThree = false;
        options[i / 3] = findOptions(s.substr(i, 3));
        for (auto p : options[i / 3]) used[p.first]++;
    }
}

```

Tính giá trị đã mã hóa của ký tự

Ký tự không bị mã hóa

Tích lũy số lần d được sử dụng

Xác định và ghi nhận các tham số mã hóa d

Ghi nhận các giá trị d có thể được sử dụng cho mọi ký tự được mã hóa:

```

vector<int> digits;
for (int i = 0; i < 10; i++)
{
    if (used[i] == (int)s.size() / 3 - three)
        digits.push_back(i);
}

```

Số lượng ký tự bị mã hóa

Tính số lượng xâu nguồn:

```

int answer = 0;
for (int d : digits)
{
    long long p = 1;
    for (int i = 0; i < (int)s.size() / 3; i++)
    {
        if (result[i] != -1) continue;
        p = (p * (long long)options[i][d].size()) % MOD;
    }
    answer = (answer + p) % MOD;
}

```

Dấu hiệu ký tự không bị mã hóa

Số khả năng của mỗi d

Độ phức tạp của giải thuật: O(nlogn).

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
const int MOD = 1e9 + 7;
string s;

bool isValid(int c)
{
    return (65 <= c && c <= 90) || (97 <= c && c <= 99);
}

map<int, set<char>> findOptions( string s)
{
    int c1 = stoi(s.substr(1));
    int c2 = (s[0] - '0') * 10 + s[2] - '0';
    int c3 = stoi(s.substr(0, 2));
    map<int, set<char>> ans;
    if (isValid(c1)) ans[(int)(s[0] - '0')].insert((char)c1);
    if (isValid(c2)) ans[(int)(s[1] - '0')].insert((char)c2);
    if (isValid(c3)) ans[(int)(s[2] - '0')].insert((char)c3);
    return ans;
}

int main()
{
    freopen("decoding.inp", "r", stdin);
    freopen("decoding.out", "w", stdout);
    ios_base::sync_with_stdio(0);
    cin >> s;
    vector<char> result(s.size() / 3, (char)-1);
    vector< map<int, set<char>>> options(s.size() / 3);
    vector<int> used(10, 0);
    bool allThree = true;
    int three = 0;
    for (int i = 0; i < (int)s.size(); i += 3)
    {
        int c = stoi(s.substr(i, 3));
        if (100 <= c && c <= 122)
        {
            result[i / 3] = (char)c;
            three++;
        }
        else
        {
            allThree = false;
            options[i / 3] = findOptions(s.substr(i, 3));
            for (auto p : options[i / 3]) used[p.first]++;
        }
    }
    if (allThree)
    {
        cout << 1 << '\n';
        return 0;
    }
    vector<int> digits;
    for (int i = 0; i < 10; i++) {
```

```

        if (used[i] == (int)s.size() / 3 - three) digits.push_back(i);
    }
int answer = 0;
for (int d : digits)
{
    long long p = 1;
    for (int i = 0; i < (int)s.size() / 3; i++)
    {
        if (result[i] != -1) continue;
        p = (p * (long long)options[i][d].size()) % MOD;
    }
    answer = (answer + p) % MOD;
}
cout << answer << '\n';
cout << "Time: "<<clock()/(double)1000<<" sec";
return 0;
}

```



VZ24. BẰNG NHAU

Tên chương trình: EQ.CPP

Có một vấn đề liên quan tới n điểm trên mặt phẳng, các điểm đều có tọa độ nguyên, tọa độ điểm thứ i là $(\mathbf{x}_i, \mathbf{y}_i)$, $i = 1 \dots n$.

Để giải quyết vấn đề giáo sư Braun sử dụng khoảng cách Manhattan, theo đó khoảng cách giữa 2 điểm i và j là $|\mathbf{x}_i - \mathbf{x}_j| + |\mathbf{y}_i - \mathbf{y}_j|$. Sau đó ông trình bày lại các lập luận của mình dựa trên cơ sở khoảng cách Euclid $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ để cho mọi người thấy là các số liệu tính toán có thể khác nhau khi sử dụng công thức tính khoảng cách khác nhau, nhưng những tính chất tìm thấy được ở một khái niệm khoảng cách vẫn đúng với khái niệm khoảng cách khác! Đó là một trong số các đặc điểm quan trọng của không gian Metric.

Tuy vậy với một số cặp điểm (i, j) giáo sư Braun không phải trình bày lại các dẫn xuất vì đơn giản là khoảng cách giữa chúng thính theo 2 cách đều bằng nhau.

Hãy xác định có bao nhiêu cặp số có khoảng cách giống nhau không phụ thuộc vào công thức áp dụng.

Dữ liệu: Vào từ file văn bản EQ.INP:

- ✚ Dòng đầu tiên chứa số nguyên n ($1 \leq n \leq 2 \times 10^5$),
- ✚ Dòng thứ i trong n dòng sau chứa 2 số nguyên \mathbf{x}_i và \mathbf{y}_i ($|\mathbf{x}_i|, |\mathbf{y}_i| \leq 10^9$).

Kết quả: Đưa ra file văn bản EQ.OUT một số nguyên – số lượng cặp tìm được.

Ví dụ:

EQ.INP
6
0 0
0 1
0 2
-1 1
0 1
1 1

EQ.OUT
11



VZ24.la201900324_Z_A XI

Giải thuật: Nguyên lý bù trừ, Kỹ thuật 2 con trỏ .

Lưu tọa độ các điểm đã cho vào mảng **a**, mỗi phần tử của mảng có kiểu **pair<int,int>**,

Khoảng cách giữa 2 điểm **i** và **j** bằng nhau theo 2 cách tính khi và chỉ khi **a[i].first==a[j].first** hoặc **a[i].second==a[j].second**.

Sắp xếp mảng **a** theo thứ tự tăng dần,

Các điểm có tọa độ đầu tiên bằng nhau sẽ tạo thành nhóm các phần tử liên tiếp,

Gọi **p** là chỉ số của phần tử đầu tiên trong nhóm,

Gọi **q** là chỉ số của phần tử cuối cùng trong nhóm,

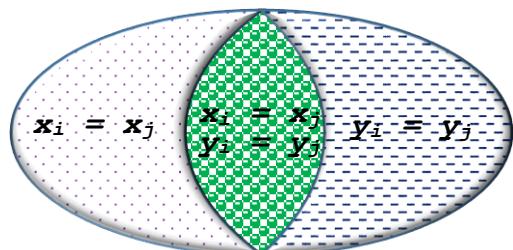
Số cặp điểm trong nhóm cho khoảng cách như nhau theo 2 cách tính là

$$(q-p) \times (q-p+1) / 2$$

Việc tìm nhóm tiếp theo được thực hiện bắt đầu từ phần tử **q+1** trong dãy **a**.

Đổi chỗ tọa độ **x** và **y** trong mỗi điểm, theo sơ đồ trên dễ dàng tính được số cặp điểm cần tìm theo tọa độ **y**.

Số cặp điểm có đồng thời tọa độ **x** bằng nhau và tọa độ **y** bằng nhau được tính 2 lần, vì vậy cần tìm các nhóm điểm này, tính số cặp tạo thành và giảm tương ứng tổng số lượng cặp đã tích lũy khi tính riêng theo các tọa độ **x** và **y**.



Tổ chức dữ liệu

Mảng **vector<pii> a** – lưu tọa độ các điểm đã cho.

Xử lý

Nhập dữ liệu và chuẩn bị:

```
fi>>n;
a.resize(n);
for(int i=0; i<n; ++i)
{
    fi>>x>>y;
    a[i] = {x, y};
}
x=(int)1e9+1;
a.push_back({x, x+1});
sort(a.begin(), a.end());
```

Phần tử hàng rào

Tìm điểm đầu đoạn các tọa độ thứ I bằng nhau:

```
int get_p()
{
    for(int i=q; i<n; ++i)
        if(a[i].ff==a[i+1].ff){p=i; return p;}
    p=n+1; return p;
}
```

Điểm xuất phát

Dấu hiệu điểm đầu

Tìm điểm cuối đoạn các tọa độ thứ I bằng nhau:

```
int get_q()
{
    for(int i=p; i<n; ++i)
        if(a[i].ff!=a[i+1].ff){q=i; return q;}
}
```

Điểm xuất phát

Dấu hiệu điểm cuối

Tìm điểm đầu đoạn tọa độ trùng nhau:

```
int get_ep()
{
    for(int i=q; i<n; ++i)
        if(a[i]==a[i+1]){p=i; return p;}
    p=n+1; return p;
}
```

Dấu hiệu điểm đầu

Tính số lượng các cặp điểm có tọa độ thứ I bằng nhau:

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

```
int64_t calc()
{
    int64_t t, r=0;
    x=(int)1e9+1; p=0; q=0;
    while (p<n)
    {
        get_p();
        if (p<n)
        {
            get_q();
            t = (int64_t)q-p;
            r+= t*(t+1)/2;
        }
    }
    return r;
}
```

Tổ hợp chap 2

Chương trình

```
#include <bits/stdc++.h>
#define ff first
#define ss second
using namespace std;
typedef pair<int,int> pii;
ifstream fi ("eq.inp");
ofstream fo ("eq.out");
int n,x,y,p,q;
vector<pii> a;

int64_t ans;

int get_q()
{
    for(int i=p; i<n; ++i)
        if(a[i].ff!=a[i+1].ff) {q=i; return q;}
}

int get_p()
{
    for(int i=q; i<n; ++i)
        if(a[i].ff==a[i+1].ff) {p=i; return p;}
    p=n+1; return p;
}

int64_t calc()
{
    int64_t t,r=0;
    x=(int)1e9+1; p=0; q=0;
    while(p<n)
    {
        get_p();
        if(p<n)
        {
            get_q();
            t = (int64_t)q-p;
            r+= t*(t+1)/2;
        }
    }
    return r;
}

int get_ep()
{
    for(int i=q; i<n; ++i)
        if(a[i]==a[i+1]) {p=i; return p;}
    p=n+1; return p;
}

int get_eq()
{
    for(int i=p; i<n; ++i)
        if(a[i]!=a[i+1]) {q=i; return q;}
}
```

```

int64_t calc_e()
{
    int64_t t, r=0;
    x=(int)1e9+1; p=0; q=0;
    while(p<n)
    {
        get_ep();
        if(p<n)
        {
            get_eq();
            t = (int64_t)q-p;
            r+= t*(t+1)/2;
        }
    }
    return r;
}

int main()
{
    fi>>n;
    a.resize(n);
    for(int i=0; i<n; ++i)
    {
        fi>>x>>y;
        a[i] = {x,y};
    }
    x=(int)1e9+1;
    a.push_back({x,x+1});
    sort(a.begin(),a.end());
    ans=calc();
    for(auto &i:a) swap(i.ff,i.ss);
    sort(a.begin(),a.end());
    p=0; q=0;
    ans+=calc();
    p=0; q=0;
    ans -= calc_e();
    fo<<"\nTime: "<<clock()/(double)1000<<" sec";
}

```



VZ25. ĐƠN VỊ

Tên chương trình: UNIT.CPP

Toàn bộ khu đất quy hoạch là khu công nghiệp có diện tích n . Người ta chia nó thành các phần bằng nhau, mỗi phần là một hình vuông để là đơn vị đấu thầu cho các công ty muốn thuê đất. Ban quản lý Khu công nghiệp muốn đơn vị đấu thầu phải càng lớn càng tốt.

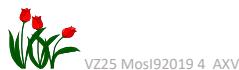
Hãy xác định diện tích lớn nhất có thể sử dụng làm đơn vị đấu thầu.

Dữ liệu: Vào từ file văn bản UNIT.INP gồm một dòng chứa số nguyên n ($1 \leq n \leq 2 \times 10^9$).

Kết quả: Đưa ra file văn bản UNIT.OUT một số nguyên – giá trị lớn nhất của đơn vị đấu thầu.

Ví dụ:

UNIT.INP	UNIT.OUT
180	36



Giải thuật: Phân tích ra thừa số nguyên tố.

Gọi **s** – diện tích lớn nhất có thể sử dụng làm đơn vị đầu thầu.

Các tính chất của **s**:

- **n** chia hết cho **s**,
- Nếu phân tích **s** ra thừa số nguyên tố, các thừa số chỉ chứa số mũ chẵn.

Từ đây suy ra $t = n/s$ chỉ chứa các thừa số nguyên tố với số mũ là 1.

Sơ đồ tính **t**:

- Phân tích **n** ra thừa số nguyên tố và giữ lại các thừa số với số mũ lẻ,
- **t** là tích các thừa số được giữ lại, số thừa số được giữ lại sẽ không quá 10.

Kết quả cần tìm sẽ là n/t .

Độ phức tạp của giải thuật: $O(n^{0.5})$.

Tổ chức dữ liệu: Mảng **vector<int>** **pr** – lưu các thừa số nguyên tố của **t**.

Chương trình

```
#include <bits/stdc++.h>
using namespace std;
ifstream fi ("unit.inp");
ofstream fo ("unit.out");
int n,k,p,ip,ans=1;

int main()
{
    fi>>n;    k=n;
    vector<int> pr;
    p=2;
    while (p*p<=n)
    {
        if (n%p==0)
        {
            ip=0;
            while (n%p==0) ++ip,n/=p;
            ip&=1;
            if (ip) pr.push_back(p);
        }
        ++p;
    }
    if (n>1) pr.push_back(n);

    for (int i:pr) ans*=i;
    ans=k/ans;
    fo<<ans;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



VZ26. CỘNG TRỪ Tên chương trình: PM.CPP

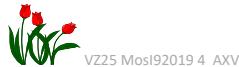
Người ta viết các số nguyên từ 1 đến n thành một dãy. Hãy tìm cách đặt trước mỗi số dấu cộng (+) hoặc dấu trừ (-) để nhận được tổng bằng 0.

Dữ liệu: Vào từ file văn bản PM.INP gồm một dòng chứa số nguyên n ($1 \leq n \leq 10^5$).

Kết quả: Đưa ra file văn bản PM.OUT xâu ký tự từ tập {+, -} xác định *một cách đặt dấu* để nhận được kết quả 0. Nếu không có cách đặt dấu để nhận kết quả 0 thì đưa ra thông báo IMPOSSIBLE.

Ví dụ:

PM.INP	PM.OUT
3	--+



Giải thuật; Cơ sở lập trình, Kỹ thuật bảng phương án.

Nhận xét:

- Việc đặt các dấu +, - không làm thay đổi tính chẵn lẻ của tổng các số từ 1 đến n ,
- Với $n = 1$ hoặc $n = 2$ – bài toán vô nghiệm,
- Với $n = 3$: Một trong số các cách đặt dấu: $++- : +1+2-3 = 0$
- Với $n = 4$: Một trong số các cách đặt dấu: $---+ : +1-2-3+4 = 0$

Với $n > 4$:

- Nếu n chia 4 dư 1 hoặc 2: Bài toán vô nghiệm (tổng các số từ 1 đến n lẻ),
- Nếu n chia hết cho 4: chia dãy bắt đầu từ đầu thành các nhóm 4 số liên tiếp nhau, đặt dấu để tổng mỗi nhóm bằng 0.
- Nếu n chia 4 dư 3: đặt dấu để đưa tổng đại số 3 số đầu tiên về 0, phần còn lại – xử lý tương tự trường hợp n chia hết 4.

Độ phức tạp của giải thuật: $O(n)$.

Chương trình

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

int n, k;

int main ()
{
    fi>>n;    k=n%4;
    if(k==1 || k==2) {fo<<"IMPOSSIBLE"; return 0;}
    if(k==3) {fo<<">+-"; n-=3;}
    for(int i=1; i<=n/4; ++i) fo<<">---";
    fo<<">  
Time: "<<clock() / (double) 1000<<" sec";
}
```

