# KỸ THUẬT PHÂN TÍCH THIẾT KẾ GIẢI THUẬT CHO CÁC CUỘC THI LẬP TRÌNH

Phạm Nguyên Khang
pnkhang@cit.ctu.edu.vn

## MỤC LỤC

1 Phương pháp luỹ thừa ma trận ..... 1
1.1 Tìm số Fibonacci ..... 1
1.2 Tính luỹ thừa của một số ..... 2
1.3 Tính luỹ thừa của một ma trận vuông ..... 3
1.4 Giải bài 10229 - Modular Fibonacci ..... 3
1.5 Bài học kinh nghiệm ..... 4
1.6 Các bài tương tự ..... 4
1.7 Mã nguồn cho bài 10229 ..... 4
2 Quy hoạch động ..... 5
2.1 Giới thiệu ..... 5
2.2 Bài toán sơn nhà ..... 5
2.3 Bài học kinh nghiệm ..... 6
2.4 Các bài tương tự ..... 7
3 Số học đồng dư ..... 13
3.1 Chia hết và chia có dư ..... 13
3.2 Số nguyên tố ..... 14
3.3 Bài toán t-prime ..... 15
3.4 Phân tích một số thành tích các thừa số nguyên tố ..... 16
3.5 Số dư của (a/b) % MOD ..... 21
4 Cây phân đoạn (Segment Tree/Interval Tree) ..... 24
4.1 Bài toán tính trung bình nhanh ..... 24
4.2 Cài đặt interval tree ..... 24
4.3 Giải bài tính trung bình nhanh ..... 25
4.4 Bài học kinh nghiệm ..... 30
4.5 Các bài tương tự ..... 30
4.6 Các bài nâng cao ..... 30
5 Tìm kiếm tam phân (Ternary Search) ..... Error! Bookmark not defined.
5.1 Bài toán khoảng cách ngắn nhất ..... 48
5.2 Bài tương tự ..... 50
6 Giải thuật trên đồ thị ..... 52
6.1 Giải thuật Dijkstra tìm đường đi ngắn nhất ..... 52
6.2 Giải thuật Kruskal tìm cây khung có trọng số nhỏ nhất ..... 56
7 Còn tiếp ... ..... Error! Bookmark not defined.

# 1 Phương pháp luỹ thừa ma trận

## 1.1 Tìm số Fibonacci

Xét bài toán "10229 - Modular Fibonacci" trên UVA như sau:

| Problem A: Modular Fibonacci |
|---|
| The Fibonacci numbers (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...) are defined by the recurrence:<br>$F_0 = 0$<br>$F_1 = 1$<br>$F_i = F_{i-1} + F_{i-2}$ for i>1<br>Write a program which calculates $M_n = F_n \bmod 2^m$ for given pair of n and m. $0 \le n \le 2147483647$ and $0 \le m < 20$. Note that a mod b gives the remainder when a is divided by b.<br><br>Input and Output<br>Input consists of several lines specifying a pair of n and m. Output should be corresponding $M_n$, one per line.<br>Sample Input<br>11 7<br>11 6<br>Sample Output<br>89<br>25 |

Yêu cầu của bài toán khá đơn giản là tính $F_n \bmod 2^m$. Ta có thể nghĩ ngay đến một lời giải như sau:

- Tính số fibonacci thứ n => $F_n$.
- Chia $F_n$ cho $2^m$ lấy phần dư => kết quả.

```
int F0= 0, F1 = 1, F2;
for (int i = 2; i <= n; i++) {
        F2 = F0 + F1;
        F0 = F1;
        F1 = F2;
}
cout << F2 % (1<< m); // 1<< m = 2m.
```

Tuy nhiên, nếu bạn cài đặt đơn giản như thế và submit chắc chắn sẽ bị "Time limit exceeded" ☹. Vậy vấn đề nằm ở đâu ? Chắc chắn là ở chỗ tính $F_n$ rồi vì bước 2 chỉ đơn giản là phép chia lấy phần dư. Phân tích lại đoạn code trên ta thấy thời gian để tính $F_n$ là O(n) vì vòng lặp for chạy khoảng n lần. Có cách nào tính $F_n$ mà tốn ít vòng lặp hơn không ? O(logn) chẳng hạn ? Có một cách như thế !

Ta thử viết lại công thức tính các số Fibonaccii.

$F_n = F_{n-1} + F_{n-2}$

$F_{n+1} = F_n + F_{n-1}$

Sắp xếp lại các F vào một ma trận 2 x 2 ta có:

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

Thay

$$\begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{bmatrix} \qquad (2)$$

vào (1) ta được:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{bmatrix} \qquad (3)$$

Tiếp tục triển khai:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} F_1 & F_0 \\ F_0 & F_{-1} \end{bmatrix} \quad (4)$$

Vậy $F_n$ (hàng 2 cột 1)= hàng cuối của ma trận $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ nhân với cột đầu tiên của ma trận $\begin{bmatrix} F_1 & F_0 \\ F_0 & F_{-1} \end{bmatrix}$ hay $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$= phần tử ở hàng 2 cột 1 của ma trận $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$.

Chú ý là trong công thức (4), giá trị của $F_{-1}$ bằng bao nhiêu không quan trọng vì ta không sử dụng đến để tính $F_n$.

Tóm lại để tính $F_n$ ta tính $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ và lấy phần tử ở hàng 2 cột 1.

Như thế, thay vì tính $F_n$ ta tính $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$. Tính luỹ thừa của ma trận như thế nào ? Có nhanh hơn tính $F_n$ trực tiếp không ?

## 1.2 Tính luỹ thừa của một số

Để phân tích thời gian tính luỹ thừa của ma trận, ta quay lại bài toán tính luỹ thừa của một số (số là ma trận 1 hàng 1 cột).

Giải thuật đơn giản để tính luỹ thừa của một số ($x^n$) là:

```
int lt = 1;
for (int i = 1; i <= n; i++) {
        lt = lt*x;
}
return x;
```

Vòng lặp for này vẫn chạy n lần => thời gian chạy vẫn là O(n). Vậy thì có gì hay hơn tính số Fibonacci trực tiếp ? Có cách nào tính $x^n$ nhanh hơn không ?

Tình $x^n$ bằng cách nhân liên tiếp như trên ta cần n phép nhân. Giả sử n là số chẵn, ta viết lại:

$$x^n = x^{n/2} * x^{n/2}$$

Như thế số phép nhân sẽ giảm bớt gần ½. Nếu n là số lẻ, $x^n = x^{n-1}*x$, $x^{n-1}$ sẽ được tính như trường hợp n chẵn.

Giải thuật đệ quy để tính $x^n$ như bên dưới.

```
int power(int x, int n) {
   if (n == 0) return 1;
   if (n % 1) return power(x, n-1) * x;
   int temp = power(x, n/2);
   return temp*temp;
}
```

Một tiếp cận khác để tính $x^n$ bằng phương pháp lặp như sau.

Giả sử $b_{k-1}\ b_{k-2} \ldots b_1\ b_0$ là biểu diễn nhị phân của n, ta có

$x^n = x^{(bk-1\ bk-2\ \ldots\ b1\ 0)} * x^{b0} = x^{2(bk-1\ bk-2\ \ldots\ b1)} * x^{b0} = (x\text{^}2)^{(bk-1\ bk-2\ \ldots\ b1)} * x^{b0} = ((x\text{^}2)\text{^}2)^{(bk-1\ bk-2\ \ldots\ b2)} * (x\text{^}2)^{b1} * x^{b0} = \ldots$

(với (x^2) = $x^2$).

Như thế ta thấy rằng nếu b0 = 1 ta nhân kết quả với x. Kế tiếp, loại bỏ bit cuối cùng đi. Sau cùng thay x bằng x^2.

Giải thuật lặp tính $x^n$ như sau:

```
int result = 1;
while (n > 0) {
      if (n % 2 == 1) result = result * x;
      x = x*x;
      n >>= 1;
}
```

## 1.3 Tính luỹ thừa của một ma trận vuông

Ta có thể mở rộng giải thuật tính luỹ thừa bằng phương pháp lặp cho việc tính luỹ thừa một ma trận. Việc mở rộng khá đơn giản. Ta thay 1 bằng ma trận đơn vị, phép nhân bằng phép nhân ma trận.

```
Matrix result = I; //I là ma trận đơn vị
while (n > 0) {
      if (n % 2 == 1) result = multiply(result, x);
      x = multiply(x, x);
      n >>= 1;
}
```

## 1.4 Giải bài 10229 - Modular Fibonacci

Giờ đây ta đã có đủ thông tin để giải bài toán Modular Fibonacci.
Biểu diễn ma trận: số khá lớn nên dùng kiểu long long hoặc (unsigned long long) cho chắc ăn ☺

$$\text{Matrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```
struct Matrix {
      long long a, b, c, d;
};
```

Tạo ma trận đơn vị

```
void identity(Matrix& I) {
      I.a = I.d = 1;
      I.b = I.c = 0;
}
```

Nhân ma trận

```
Matrix multiply(Matrix A, Matrix B) {
      Matrix C;
      C.a = (A.a*B.a + A.b*B.c) % mod;
      C.b = (A.a*B.b + A.b*B.d) % mod;
      C.c = (A.c*B.a + A.d*B.c) % mod;
      C.d = (A.c*B.b + A.d*B.d) % mod;
      return C;
}
```

Với mod = $2^m$ = (1<<m);

## 1.5 Bài học kinh nghiệm

- Tính luỹ thừa của 1 số hoặc 1 ma trận bằng phương pháp lặp hoặc đệ quy có độ phức tạp thời gian O(logn). Ta làm được điều này vì 2 lý do:
    - Ta chỉ cần tính $x^n$ mà không cần phải tính tất cả $x^1$, $x^2$, …, $x^{n-1}$ nên không cần phải duyệt qua tất cả các con số từ 1 đến n.
    - Tách bài toán lớn thành 2 **bài toán nhỏ giống nhau** nên chỉ cần giải 1 bài.
- Đưa bài toán tìm số Fibonacci về bài toán tìm luỹ thừa nhờ vào cách biểu diễn ma trận.

## 1.6 Các bài tương tự

- Light Oj (http://lightoj.com): chủ đề Matrix/Matrix Exponentiation
-

## 1.7 Mã nguồn cho bài 10229

Giờ đây ta đã có đủ thông tin để để bài toán Modular

Nào ta cùng submit !!!!

```
#include <iostream>
using namespace std;
long long mod;
struct Matrix {
	long long a, b, c, d;
};
Matrix F;
void identity(Matrix& I) {
	I.a = I.d = 1;
	I.b = I.c = 0;
}
Matrix multiply(Matrix A, Matrix B) {
	Matrix C;
	C.a = (A.a*B.a + A.b*B.c) % mod;
	C.b = (A.a*B.b + A.b*B.d) % mod;
	C.c = (A.c*B.a + A.d*B.c) % mod;
	C.d = (A.c*B.b + A.d*B.d) % mod;
	return C;
}


long long process(long long n, int m) {
	mod = (1 << m);
	Matrix R;
	identity(R);
	while (n > 0) {
		if (n%2) R = multiply(R, F);
		F = multiply(F, F);
		n >>= 1;
	}
	return R.c;
}

int main() {
	long long n, m;
	while (1) {
		cin >> n >> m;
```

```
            if (cin.fail())
                  break;
            if (n <= 1)
                  cout << n << endl;
            else {
                  F.a = F.b = F.c = 1; F.d = 0;
                  cout << process(n, m) << endl;
            }
      }
      return 0;
}
```

# 2 Quy hoạch động

## 2.1 Giới thiệu

Quy hoạch động là một kỹ thuật được dùng để thiết kế các giải thuật có độ phức tạp thời gian thấp (đa thức) bằng cách lưu trữ các kết quả trung gian. Ta có thể sử dụng kỹ thuật này bằng 2 cách:

- Dùng quy hoạch động để khử đệ quy và giảm thời gian thực hiện
- Phân tích trực tiếp bài toán bằng quy hoạch động

## 2.2 Bài toán sơn nhà

| 1047 - Neighbor House |
| --- |
| The people of Mohammadpur have decided to paint each of their houses red, green, or blue. They've also decided that no two neighboring houses will be painted the same color. The neighbors of house i are houses i-1 and i+1. The first and last houses are not neighbors.<br><br>You will be given the information of houses. Each house will contain three integers "R G B" (quotes for clarity only), where R, G and B are the costs of painting the corresponding house red, green, and blue, respectively. Return the minimal total cost required to perform the work.<br><br>Input<br><br>Input starts with an integer T (≤ 100), denoting the number of test cases.<br><br>Each case begins with a blank line and an integer n (1 ≤ n ≤ 20) denoting the number of houses. Each of the next n lines will contain 3 integers "R G B". These integers will lie in the range [1, 1000].<br><br>Output<br><br>For each case of input you have to print the case number and the minimal cost.<br><br>Sample Input<br><br>2<br><br><br>4<br><br>13 23 12<br><br>77 36 64<br><br>44 89 76 |

```
31 78 45


3
26 40 83
49 60 57
13 89 99
Output for Sample Input
Case 1: 137
Case 2: 96
```

Có n căn nhà cần sơn. Mỗi căn nhà được sơn bằng 1 trong 3 màu R, G, B với 3 mức giá tương ứng. Phải sơn làm sao cho hai căn nhà cạnh nhau không cùng màu và tổng giá ít nhất.

Gọi r[i], g[i] và b[i] là các giá tương ứng nếu sơn căn nhà i bằng các màu R, G, B.

Để phân tích các bài dạng như thế này ta chia bài toán thành n bước ứng với n căn nhà. Tại bước i, ta giả sử các căn nhà từ 1 đến (i-1) đã được sơn rồi và tổng giá từ căn 1 đến (i-1) cũng đã biết rồi. Nếu căn nhà thứ (i-1) sơn màu R và có tổng giá từ 1 đến (i-1) là R(i-1); nếu căn nhà thứ (i-1) sơn màu G có tổng giá là G(i-1) và nếu sơn màu B có tổng giá B(i-1) thì:

- Căn nhà thứ i sơn màu R có tổng giá là R(i) = min (G(i-1), B(i-1)) + r[i]
- Căn nhà thứ i sơn màu G có tổng giá là G(i) = min (R(i-1), B(i-1)) + g[i]
- Căn nhà thứ i sơn màu B có tổng giá là B(i) = min (G(i-1), R(i-1)) + b[i]

Vậy giá nhỏ nhất để sơn n căn nhà là = min(R(n), G(n), B(n));

Khởi tạo R(0) = G(0) = B(0) = 0;

## 2.3 Bài học kinh nghiệm

- Rất nhiều bài toán có thể giải bằng kỹ thuật quy hoạch động. Các bài toán quy hoạch động thường liên quan đến việc tìm lớn nhất, nhỏ nhất, tổng số.
- Cách tiếp cận để giải là chia bài toán thành nhiều bước, mỗi bước là một bài toán con "tương tự nhau". Tại mỗi bước ta có 1 số sự lựa chọn.
- Ở mỗi bước, sử dụng thông tin đã được tích luỹ ở các bước trước đó để tính toán kết quả cho bước hiện tại. Lưu kết quả của bước hiện tại để dành cho các bước sau.
- Cố gắng biểu diễn thông tin cần lưu trữ tại từng bước sao cho việc tính toán kết quả tại bước i chỉ cần sử dụng các thông tin tổng hợp của các bước từ 1 đến i-1. Hay nhất là để tính ở bước i ta chỉ cần thông tin của bước i-1 (không phải bài nào cũng rơi vô trường hợp này).
- Nếu các sự lựa chọn ở bước i **không phụ thuộc** vào các sự lựa chọn của các bước trước đó thì đây là bài toán dễ. Nếu có sự ràng buộc giữa các lựa chọn, cố gắng biểu diễn sao cho các sự lựa chọn ở bước i chỉ nên có ràng buộc với các lựa chọn ở bước i – 1 mà thôi. Nếu không làm được như vậy ta không thể sử dụng quy hoạch động (trường hợp này giống như vét cạn, phải xét hết tất cả các trường hợp). Vì nếu có sử dụng được cũng không tiết kiệm được thời gian. Trong bài toán coin change, sự lựa chọn của bước i không phụ thuộc vào bất cứ sự lựa chọn nào ở trước đó. Đối với bài toán sơn nhà, sự lựa chọn ở bước i chỉ phụ thuộc vào sự lựa chọn ở bước i-1 mà không quan tâm đến sự lựa chọn ở các bước i-2, i-2, …

- Đối với 1 số bài toán, biểu diễn các sự lựa chọn là một vấn đề ta cần phải quan tâm. Nếu biểu diễn không khéo, số lựa chọn sẽ lên đến $a^n$ hoặc n! (xem bài Agent 47 bên dưới).

## 2.4 Các bài tương tự

- Light Oj: Nhóm Divide and conquer/Dynamic programming

**1231 – Coin Change (I):** tìm số cách tạo số tiền có giá trị K bằng cách dùng các đồng tiền có mệnh giá: A1, A2 ... An và mỗi mệnh giá có số lượng C1, C2, ... Cn.

Ta chia bài toán thành n bước, mỗi bước ta xét từng loại mệnh giá.

Ở bước i, ta xét đồng tiền có mệnh giá A[i], ta các sự lựa chọn sau:

- Chọn 0 đồng có mệnh giá A[i]
- Chọn 1 đồng có mệnh giá A[i]
- …
- Chọn k đồng có mệnh giá A[i], k = min(C[i], K/A[i]). Ta chỉ có thể chọn nhiều nhất là k đồng.

Với mỗi sự lựa chọn như thế, ví dụ chọn j đồng có mệnh giá A[i], số tiền còn lại là K – j*A[i]. Vậy số cách tạo ra số tiền có giá trị K với lựa chọn j đồng mệnh giá A[i] sẽ là số cách tạo ra K – j*A[i] với i-1 mệnh giá còn lại. Nếu gọi F[i][K] là số cách tạo ra số tiền có giá trị K với i mệnh giá, thì:

F[i][K] = tổng F[i-1][K-j*A[i]] với j = 0, 1, 2, …, k = min(C[i], K/A[i]).

Khởi tạo:

F[i][0] = 1, F[0, j>0] = 0;

**1232 – Coin Change (II):** tìm số cách tạo số tiền có giá trị K bằng cách dùng các đồng tiền có mệnh giá: A1, A2 ... An và mỗi mệnh giá có số lượng K.

Bài này tương tự bài 1231, tuy nhiên do số lượng cho mỗi mệnh giá là K và mỗi mệnh giá đều >= 1 nên ta có thể xem như số lượng mỗi mệnh giá là không hạn chế.

Công thức cho bước i có thể viết lại như sau:

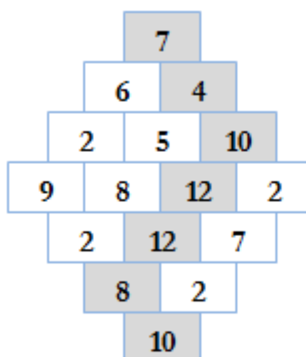
F[i][K] = tổng F[i-1][K-j*A[i]] =

= F[i-1][K - k*A[i]] + F[i-1][K - (k-1)*A[i]] + … + F[i-1][K - A[i]] + F[i-1][K]

= F[i][K-A[i]] + F[i-1][K];

Nhận xét này khá quan trọng vì ta không cần phải tính "tổng F[i-1][K-j*A[i]]" (bỏ bớt được 1 vòng lặp).

**1004 - Monkey Banana Problem**: tìm đường đi cho chú khỉ để thu được nhiều chuối nhất. Bắt đầu ô trên cùng đến ô dưới cùng. Quy tắc đi: đi xuống và chỉ đi đến 1 trong các ô có chung cạnh với ô hiện tại.



Đầu vào: N và 2*N – 1 hàng, mỗi hàng chứa các số của hàng đó.

Đầu ra: tổng số chuối nhiều nhất.

Nếu ta đánh số các hàng từ trên xuống dưới, bắt đầu từ 1và canh lề tất các các hàng về bên trái ta sẽ một hình như thế bên dưới. Gọi bảng này là A[i, j].

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 7 | | | |
| 2 | 6 | 4 | | |
| 3 | 2 | 5 | 10 | |
| 4 | 9 | 8 | 12 | 2 |
| 5 | 2 | 12 | 7 | |
| 6 | 8 | 2 | | |
| 7 | 10 | | | |

**Nhận xét:**

- Từ hàng 1 đến hàng N-1, số phần tử tăng dần. Từ hàng N đến hàng 2N-1 số phần tử giảm dần.
- Các hàng từ 2 đến N, ô (i, j) nếu j < i thì có chung cạnh với ô(i-1, j-1) và ô (i-1, j) ngược lại nó là phần tử cuối cùng, chỉ chung cạnh với ô(i-1, j-1).
- Các hàng từ N+1 đến 2N-1, ô (i, j) có chung cạnh với ô (i-1, j) và ô (i-1, j+1).

**Áp dụng kỹ thuật quy hoạch động:**

Ta chia bài toán thành N bước, mỗi bước xét 1 hàng. Ở hàng i, ta lần lượt xét các ô (i, j) và tính xem để đi đến đó thì con khỉ thu được nhiều nhất bao nhiêu chuối.

Nếu gọi F[i, j] là tổng số chuối nhiều nhất mà con khỉ thu được từ hàng 1 đến ô (i, j), ta có:

$$F[i, j] = \max \{F[i-1, k]\} + A[i, j]$$

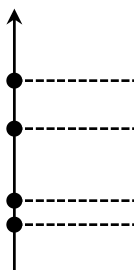
Với ô (i-1, k) là các ô có chung cạnh với ô (i, j). Các ô chung cạnh được tính theo nhận xét bên trên.

Kết quả cuối cùng thu được ở F[2N-1][1]

Các bạn thử cài đặt và submit nhé !

**1017 - Brush (III)**: Samir cần phải sơn một bức tường để xoá các điểm bụi. Bụi nằm tại các toạ độ nguyên. Cọ sơn có kích thước w. Nếu Samir đặt cọ tại toạ độ y* (không quan tâm đến x) thì tất cả điểm bụi có toạ độ y nằm trong khoảng y* đến y* + w đều được xoá. Sau khi đặt cọ và sơn xoá bụi, Samir lại tiếp tục đặt cọ tại một vị trí khác và sơn tiếp. Tuy nhiên Samir chỉ có thể sơn tối đa k lần. Hãy tìm tổng số điểm bụi lớn nhất mà Samir có thể xoá.

Sắp xếp các điểm bụi theo thứ tự tung độ giảm dần y[1], y[2], …, y[n].



Ta chia bài toán thành n bước, mỗi bước xét 1 điểm bụi. Gọi F[i, k] là số điểm bụi nhiều nhất bị xoá tính từ 1 đến i với k lần sơn. Ở mỗi bước i, ta có 2 sự lựa chọn:

- Bỏ qua điểm bụi i, số điểm bụi bị xoá nhiều nhất là F[i, k] = F[i-1, k]
- Đặt cọ tại vị trí tung độ của i, tất cả các điểm bụi từ y[i] đến y[i] – w sẽ bị xoá hết. Số lần sơn chỉ còn lại k – 1. Gọi j là điểm bụi có tung độ vừa lớn y[i] – w, số điểm bụi nhiều nhất bị xoá với lựa chọn là F[I, k] = F[j, k-1] + (i – j).

Giải thuật quy hoạch động sẽ như sau:

```
long long process(int n, int w, int K) {
   memset(F, 0, sizeof(F));

   for (int i = 1; i <= n; i++) {
      F[i][0] = 0;
      for (int k = 1; k <= K; k++) {
         int j = i;
         while (j >= 1)
            if (y[j] <= y[i] + w)
               j--;
            else
               break;
         F[i][k] = std::max(F[j][k-1] + (i - j), F[i-1][k]);
      }
   }
   return F[n][K];
}
```

**1169 - Monkeys on Twin Tower**

Có một toà tháp đôi, mỗi tháp có n tầng. Tầng trệt được đánh số 0, tầng kế tiếp được đánh số 1, …, tầng cuối cùng được đánh số n-1. Mỗi tầng có một số trái cây. Con khỉ trước biết thời gian ăn hết trái cây ở các tầng. Đầu tiên con khỉ ở tầng trệt, mỗi lần nó có thể nhảy lên tầng trên nó của cùng 1 toà tháp với thời gian xem như bằng không, hoặc đi theo cầu thang xoắn

lên tầng trên của toà tháp kia với một khoảng thời gian t nào đó. Nhiệm vụ của con khỉ là phải ăn hết trái cây của đúng **n** tầng. Tính thời gian ít nhất con khỉ hoàn thành nhiệm vụ.

Ví dụ:

| Tầng | TG ăn ở các tầng bên tháp trái<br>**L[i]** | TG ăn ở các tầng bên tháp phải<br>**R[i]** | TG nhảy từ tầng i của tháp trái sang tầng (i+1) tháp phải<br>**L2R[i]** | TG nhảy từ tầng i của tháp phải sang tầng (i+1) tháp trái<br>**R2L[i]** |
|---|---|---|---|---|
| **3** | 9 | 10 | | |
| **2** | 8 | 3 | 3 | 3 |
| **1** | 6 | 9 | 2 | 4 |
| **0** | 5 | 7 | 5 | 2 |

Kết quả: 26

Ta chia bài toán thành n bước, mỗi bước xét một tầng. Ở bước i, con khỉ có thể ở toà tháp trái hoặc toà tháp phải. Với mỗi vị trí ở bước i, trước đó ở bước i-1 nó có thể ở tầng trái hoặc tháp phải. Gọi FL[i] là thời gian nhỏ nhất để con khỉ ở tầng i của tháp bên trái và FR[i] là thời gian nhỏ nhất để con khỉ ở tầng i toà tháp bên phải, ta có:

- FL[i] = min (FL[i-1], FR[i-1] + R2L[i-1]) + L[i]
- FR[i] = min (FR[i-1], FL[i-1] + L2R[i-1]) + R[i]

Nếu nó đang ở tầng i của toà tháp trái, nếu trước đó 1 bước nó cũng ở toà tháp trái thì không cần phải cộng thêm thời gian nhảy từ phải sang trái, ngược lại ta phải cộng thêm thời gian này. Lý luận tương tự cho trường hợp nó ở tầng i của toà tháp phải.

Các bạn thử code và submit nhé.

**1037 – Agent 47:** Cho n mục tiêu (1<= n <= 15), mỗi mục tiêu có chỉ số sức khoẻ là A1, A2, …, An. Nhiệm vụ của Agent 47 là phải tiêu diệt hết tất cả các mục tiêu. Đầu tiên Agent được trang bị 1 vũ khí có khả năng làm tiêu hao 1 đơn vị sức khoẻ trên 1 lần bắn. Một mục tiêu sẽ bị tiêu diệt nếu chỉ số sức khoẻ của nó <= 0. Mỗi khi tiêu diệt được 1 mục tiêu, agent có thể thu gom vũ khí của mục tiêu đó để tiêu diệt các mục tiêu khác. Vũ khí của một mục tiêu có tác dụng khác nhau trên từng mục tiêu khác. Gọi $C_{ij}$ là khả năng làm tiêu hao sức khoẻ của vũ khí (của mục tiêu) i trên một lần bắn đối với mục tiêu j, $0 <= C_{ij} <=9$. Tìm cách cho agent 47 tiêu diệt hết tất cả các mục tiêu với số lần bắn ít nhất.

Bài này thuộc dạng khó. Ta thử phân tích các vấn đề sau:

- Chia bài toán thành n bước như thế nào ? Mỗi bước là gì ?
- Sự lựa chọn cho từng bước như thế nào ? Với mỗi lựa chọn, để tính toán thông tin ta cần thông tin gì ở bước i-1.

Nếu ta chia bài toán thành n bước, **mỗi bước chọn 1 mục tiêu để tiêu diệt**, thì việc lựa chọn mục tiêu ở bước i sẽ phụ thuộc vào **các bước trước đó mục tiêu nào đã được chọn**.

> *Kỹ thuật quy hoạch động sẽ áp dụng được trong trường hợp này nếu số cách lựa chọn của tất cả các bước trước đó (thường là tích các sự lựa chọn của các bước từ 1 đến i-1) đủ nhỏ để ta có thể lưu trữ lại được. Nếu số lựa chọn này quá lớn quy hoạch động trong trường hợp này sẽ thấy bại. Hãy tìm cách chia bước và biểu diễn lại sự lựa chọn. Một trong những cách làm giảm sự lựa chọn của tất cả các bước trước đó là: **phi thứ tự hoá** sự lựa chọn ở các bước trước đó. Ví dụ bước 1 chọn a, bước 2 chọn b, bước 3 chọn c thì cũng giống như bước 1 chọn b, bước 2 chọn c, bước 3 chọn a. Chọn cách nào thì sau 3 bước ta cũng có a, b, c. Nếu làm được như thế ta đã giảm từ **giai thừa** thành **tổ hợp** ! Số lượng lựa chọn sẽ giảm đáng kể để có thể lưu trữ.*

Quay trở lại bài toán, nếu đã biểu diễn 1 mục tiêu đã bị tiêu diệt là 1, và chưa bị tiêu diệt là 0 thì mỗi sự lựa chọn của các bước từ 1 đến i-1 có thể biểu diễn bằng 1 chuỗi nhị phân n bit. Do n <= 15 nên chuỗi nhị phân n bit tương ứng với 1 số nguyên 15 bit. Do đó, ta có thể sử dụng 1 số nguyên để biểu diễn các sự lựa chọn khác nhau của các bước từ 1 đến i-1. Nếu gọi j là số nhị phân biểu diễn các sự lựa chọn của các bước từ 1 đến i-1, thì ở bước i ta có:

- Nếu chọn mục tiêu k để tiêu diệt, thì số lần bắn chỉ phụ thuộc vào j (biểu diễn những mục tiêu đã bị tiêu diệt rồi, để ta xem lấy vũ khí nào bắn k là tốt nhất).
- Ta không cần quan tâm đến thứ tự các mục tiêu đã bị tiêu diệt trong j.

Vì thế, nếu gọi F[i-1][j] là số lần bắn ít nhất để tiêu diệt tất cả các mục tiêu diệt trong j ở bước i-1 thì số lần ít nhất để tiêu diệt mục tiêu <j+k> ở bước I là:

$$F[i][<j+k>] = \min (F[i-1][j] + h(j, k)) \text{ với mọi } j.$$

Trong đó:

- <j+k> là phép toán đánh dấu mục tiêu k đã bị tiêu diệt trong j, hay đặt bit tương ứng với k trong j bằng 1. Điều này có nghĩa là tiêu diệt thêm đối tượng k trong j. Dĩ nhiên là k chưa bị tiêu diệt trong j.
- h(j, k) = số lần bắn đối tượng k với vũ khí tốt nhất trong j.

Giải thuật quy hoạch động cho bài này như sau:

```
for(int i=1; i<=n; i++)
  for(int j=1; j<=m; j++)
    F[i][j] = inf; //inf là số lớn nhất có thể: 9999999

for(int i=1; i <= n; i++)
  F[1][1 << (i-1)] = a[i]; //a[i] là chỉ số sức khoẻ của i
```

```
for(int i=2; i<=n; i++)
  for (int j=1; j<=m; j++)
     if (F[i-1][j] < inf)
       for (int k=1; k<=n; k++)
          if (!((1<<(k-1)) & j)) {
            int h = 1;
            for (int l=1; l<=n; l++) //tìm vũ khí tốt nhất
               if ((1<<(l-1)) & j)
                 h = max(h, C[l][k];
            F[i][(1<<(k-1))|j] =
                    min(F[i][(1<<(k-1))|j],
                        F[i-1][j] + shot(h, k));
          }
```

Trong giải thuật trên,

- inf là vô cùng. Có thể chọn inf = tổng các A[i].
- C[l][k] = $C_{lk}$: sức công phá của vũ khí l đối với mục tiêu k
- shot(h, k): số lần bắn k với sức công phá h và bằng ⌈k/h⌉.
- hai vòng for (j) và for (k) có thể đổi chỗ cho nhau. Tích của 2 sự lựa chọn chính là sự lồng nhau của 2 vòng lặp.
- phép toán <j+k> được cài đặt bằng phép OR trên bit j | (1 <<(k-1)). 1 << (k-1) là 1 số nguyên có đúng 1 bít 1 nằm ở vị trí k (tính từ phải qua).
- Để kiểm tra mục tiêu k có trong j hay chưa ta sử dụng phép AND trên bit: 1<<(k-1) & j.
- Lệnh `if (F[i-1][j] < inf )` được dùng để tiết kiệm thêm 1 ít thời gian vì nếu `F[i-1][j] = inf` thì ta không cần cập nhật `F[i][(1<<(k-1))|j]` làm gì.

Trong bài này ta thấy rằng không phải quy hoạch động lúc nào cũng quy hoạch trên số nguyên mà còn có thể quy hoạch trên tập hợp. chỉ số j trong F[i][j] về mặt bản chất chính là một tập hợp. Ta xét tiếp một bài tương tự như thế.

**1021 - Painful Bases:** Cho một cơ số **base** (2 <= base <= 16), một số nguyên **K** (1<= K <=20, K được cho trong hệ thập phân) và một chuỗi ký tự **s** biểu diễn 1 số nguyên trong hệ **base-phân** (nhị phân nếu base =2, thập phân nếu base = 10, thập lục phân nếu base =16, …). Các ký tự trong chuỗi **s** đều khác nhau. Tìm số hoán vị các phần tử trong **s** sao cho tạo thành một số trong hệ base-phân chia hết cho **K**.

Ví dụ:

base = 2, K = 2, s = 10 =>kết quả 1

base = 10, K = 2, s = 5681 =>kết quả 12

Tương tự như bài Agent 47, ta chia bài toán thành n bước, mỗi bước lựa chọn 1 số trong **s** để ghép vào kết quả của các bước trước đó để tạo thành một số nguyên. Bước 1 có n cách chọn. Với mỗi cách chọn ở bước 2 có n-1 cách chọn cho bước 2, … Với mỗi cách chọn của bước 1, 2, 3, …, (i-1), bước i có (n-i+1) cách chọn. Do đó số cách chọn cho bước n sẽ là n! Ta phải tìm cách biểu diễn tốt hơn.

Giả sử ta đang xét ở bước i, gọi j là tập hợp các số đã được chọn ở các bước từ 1 đến (i-1). Cần phải chú ý là: mỗi một hoán vị trong j lại cho ta một số nguyên khác nhau, vì thế j có thứ tự. Điều này sẽ làm số lựa chọn ở bước i rất lớn. Ta làm mất tính thứ tự của j như sau: vì ta quan tâm đến số dư của các số được tạo ra từ hoán vị chia cho K, nên ta phân các hoán vị của j vào các nhóm từ 0 đến K-1 tương ứng với số dư khi chia K. Như thế số các hoán vị của j là lớn nhưng ta chỉ gom lại còn có K trường hợp khác nhau. K <= 20 nên có thể chấp nhận được. Với nhận xét này, nếu gọi F[j][r] là số các hoán vị các số trong j chia cho K dư r thì:

F[<j+k>][mod] = tổng F[j][r] với mọi j.

Trong đó, mod = (s[k]*base^(i-1) + r) % K do ở bước i, ta chọn số s[k] ghép thêm vào bên trái một trong các hoán vị j (ta cũng có thể ghép thêm vào bên phải, lúc đó công thức tính mod sẽ khác đi 1 chút). Điều này có nghĩa là nếu các hoán vị của j tạo thành số chia cho K dư r thì khi ghép thêm s[k] vào bên trái số tạo ra khi chia cho K dư mod (nếu ghép vào bên phải số dư sẽ là mod = (r*base + s[k])%K).

Giải thuật quy hoạch động cho bài này được cho như bên dưới:

```
memset(F, 0, sizeof(F));
for (int i = 1; i <= n; i++) {
        int val = digit[i];
        F[1 << (i-1)][val%K] = 1;
}

long long m = (1 << n) - 1;
long long b = 1;
for (int i = 2; i <= n; i++) {
        b = (b*base)%K; //nguy hiem cẩn thận do tràn số
        for (int r = 0; r < K; r++)
                for (long long j = 1; j <= m; j++) {
                        if (F[j][r] > 0 && num(j) == i-1) { //nguy hiem: cần kiểm
                                for (int k = 1; k <= n; k++)  // tra số phần tử của j
                                        if (!((1 << (k-1)) & j))  {
                                                int mod = (digit[k]*b + r)%K;
                                                F[j | (1 << (k-1))][mod] +=
                                                                F[j][r];
                                        }
                        }
                }
}
return F[m][0];
```

Không giống như bài Agent 47, thông tin về số bước i được lưu trong F[i][j], Bài này ta không lưu i nên phải kiểm tra num(j) == i-1 với num(j) là số bit 1 có trong j (tương ứng số phần tử đã chọn). Có thể hình dung kỹ thuật quy hoạch động được áp dụng như sau: đầu tiên j chỉ chứa một phần tử (bước 1), bước 2 chọn thêm một phần tử nữa ghép vào j, … đến bước n, j sẽ chứa n số 1 ứng với tất cả các phần tử đều đã được chọn.

# 3 Số học đồng dư

## 3.1 Chia hết và chia có dư

**Chia hết**: Cho số nguyên **a** và **số nguyên dương b**. Ta gọi **a chia hết cho b** nếu tồn tại số nguyên **q** sao cho **a = q*b**. **q** được gọi là thương của phép chia. Trong trường hợp này ta gọi **b là ước số của a** và **a là bội số của b**.

**Chia có dư**: Cho số nguyên **a** và **số nguyên dương b**. Ta nói **a chia cho b dư r** nếu tồn tại số nguyên **q** sao cho **a = q*b + r. r** được gọi là số dư ($0 \le r < b$)**, q** vẫn được gọi là thương của phép chia.
Chú ý:

- **Số dư không bao giờ âm**.
- Phép toán chia lấy phần dư trong các ngôn ngữ lập trình (% trong C/C++, mod trong PASCAL) không quan tâm đến việc số bị chia là số âm. Vì thế nếu kết quả âm ta cộng kết quả với số chia. Ví dụ -7 % 3 = -1 nên số dư là -1 + 3 = 2.

**Đồng dư:** Cho hai số nguyên **a** và **b**, ta gọi **a** và **b** đồng dư theo modulo **m** nếu **a – b** chia hết cho **n**. Kí hiệu: **a ≡ b** (mod **n**). **a** và **b** đồng dư theo modulo n cũng có thể hiểu là số dư khi chia **a** cho **n** bằng với số dư khi chia **b** cho **m**.
Một số tính chất của đồng dư có thể sử dụng để giải các bài toán. Nếu gọi % phép chia lấy phần dư:

- (a + b) % MOD = a % MOD + b % MOD
- (a - b) % MOD = a % MOD - b % MOD
- (a * b) % MOD = a % MOD * b % MOD
- $a^n$ % MOD = $(a \% MOD)^n$

Đa số các bài toán có kết quả quá lớn thường được yêu cầu chia cho 1,000,000,007 lấy phần dư.

## 3.2 Số nguyên tố

Số **p** > 1 được gọi là số nguyên tố nếu nó chỉ có đúng 2 ước số dương là 1 và chính nó, ví dụ: 2, 3, 5, 7, 11, …

Chú ý:

- **1 không phải là số nguyên tố.**
- **2 là số nguyên tố nhỏ nhất.**

**Kiểm tra một số p có phải là số nguyên tố:** giải thuật đơn giản là ta lần lượt thử chia p cho i chạy từ 2 -> sqrt(p). Nếu p chia hết cho bất kỳ 1 số nào trong khoảng từ 2 đến sqrt(p) thì p không phải là số nguyên tố.

```
for (int = 2; i <= sqrt(p); i++)
        if (p % i == 0)
                return false;
return true;
```

**Liệt kê các số nguyên tố từ 2 đến MAX**: Có thể sử dụng sàng Eratosthenes để tìm các số nguyên tố trong khoảng từ 2 đến MAX. Ý tưởng như sau:

1. Tạo một danh sách chứa các số từ 2 đến MAX.
2. Đánh dấu tất cả là số nguyên tố.
3. Lần lượt xét các số **p** từ 2 đến **sqrt(MAX)**, nếu nó được đánh dấu là số nguyên tố (đầu tiên là 2, lần thứ 2 là 3, lần thứ 3 là 5, …), đánh dấu xoá tất cả các bội số của **p** trong danh sách.

Kết thúc giải thuật, các số được đánh dấu là số nguyên tố sẽ là số nguyên tố.

```
int prime[MAX] ;
```

```
for (int i = 2; i <= MAX; i++) {
        prime[i] = true;
}


for (int p = 2; p*p <= MAX; p++)
        if (prime[i])
                for (int j = p*p; p < MAX; j += p)
                        prime[j] = false;
```

Nếu ta đã có được các số nguyên tố từ 2 đến sqrt(p), việc kiểm tra tính nguyên tố của p có thể thực hiện nhanh hơn bằng cách thử chia p cho các số nguyên tố đó.

### 3.3 Bài toán t-prime

Xem chi tiết tại đây (http://codeforces.com/problemset/problem/230/B).

Một số được gọi là t-prime nếu nó có đúng 3 ước số dương. Một số nguyên dương sẽ có 2 ước số là 1 và chính nó. Ngoài 1 và chính nó, nếu p có thêm 1 ước số nhỏ hơn sqrt(p) thì chắc chắn nó sẽ có 1 ước khác. Vì thế p có đúng 3 ước số nếu như nó có 1 ước nguyên tố đúng bằng sqrt(p).

Nói cách khác, p là t-prime nếu nó là bình phương của 1 số nguyên tố.

Trong bài toán này ta phải xét xem số x ($1 <= x <= 10^{12}$) có phải là t-prime hay không.

Như đã phân tích ở trên, x phải = $q^2$ và q là nguyên tố. Ta cần tìm tất cả các số nguyên tố từ 2 đến sqrt(x). sqrt(x) lớn nhất là sqrt($10^{12}$) = $10^6$. Dùng sàng Eratosthenes tìm các số nguyên tố từ 2 đến $10^6$. Sau đó ta kiểm tra sqrt(x) có phải là số nguyên không và sqrt(x) có phải là số nguyên tố không.

```
#include <iostream>
#include <math.h>
using namespace std;
//230 B t-prime
#define MAXPRIME 1000001
int prime[MAXPRIME];

void erathosthenes() {
   prime[0] = prime[1] = 0;
   for (long long i = 2; i < MAXPRIME; i++)
      prime[i] = 1;

   for (long long i = 2; i*i < MAXPRIME; i++) {
      if (prime[i])
         for (long long j = i*i; j < MAXPRIME; j += i)
            prime[j] = 0;
   }
}

int main() {
   long long n, x;
   erathosthenes();
```

```
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> x;
        long long q = sqrt(x);
        if (q*q == x && prime[q])
            cout << "YES" << endl;
        else
            cout << "NO" << endl;
    }
    return 0;
}
```

## 3.4 Phân tích một số thành tích các thừa số nguyên tố

Mọi số nguyên dương n đều có thể được phân tích duy nhất thành tích các thừa số nguyên tố: $n = p_1^{k1} * p_2^{k2} * \ldots * p_N^{kN}$ với $p_1, p_2, \ldots, p_N$ là các số nguyên tố, $k_1, k_2, \ldots, k_N$ là các số nguyên >= 0.

Ví dụ: $6 = 2*3$, $5 = 5$, $12 = 2^2*3$

Phân tích 1 số nguyên lớn thành tích các thừa số nguyên tố rất tốn thời gian. Tuy nhiên nếu ta đã biết các số nguyên tố từ 2 đến sqrt(n) ta có thể nhanh chóng tìm ra các thừa số nguyên tố của n và từ đó có thể tìm được tất cả các ước số của n.

**1054 - Efficient Pseudo Code.** Xét bài toán tìm tổng các ước số của $\mathbf{n^m}$ như sau: cho 2 số nguyên có dấu 32 bit n và m (n >= 1 và m >= 0), tính tổng các ước của $\mathbf{n^m}$. Chia tổng số cho 1000,000,007 để lấy phần dư.

Giả sử **n** được phân tích thành $p_1^{k1} * p_2^{k2} * \ldots * p_N^{kN}$, thì:

$$n^m = p_1^{m*k1} * p_2^{m*k2} * \ldots * p_N^{m*kN}$$

Tổng các ước của $n^m$ sẽ là:

$S = (p_1^0 + p_1^1 + \ldots + p_1^{m*k1})* (p_2^0 + p_2^1 + \ldots + p_2^{m*k1})* \ldots *(p_N^0 + p_N^1 + \ldots + p_N^{m*kN})$

Mỗi một phần tử sau khi nhân phân phối vế phải của S sẽ là 1 ước số của $n^m$.

Như vậy, để tính S trước hết ta phải tìm các $p_i$ và $k_i$, sau đó tính tích của các tổng.

Tìm $p_i$ và $k_i$ chính là phân tích n thành các thừa số nguyên tố. Giả sử các số nguyên tố từ 2 đến sqrt(n) được lưu trong danh sách pp, ta phân tích n như sau:

```
for (int i = 0; i < pp.size(); i++) {
        int k = 0;
        while (n%pp[i] == 0) {
                n /= pp[i];
                k++;
        }
        if (k > 0)
                output (pp[i], k)
}
if (n > 1)
        output (n, 1);
```

Dòng if cuối cùng kiểm tra xem nếu n còn sót lại 1 ước nguyên tố lớn hơn sqrt(n) không. Ví dụ n = 36, sqrt(36) = 6, vòng for i sẽ output ra (2, 2). Sau vòng for n = 36/2/2 = 9, ta phải output (9,1).

Vấn đề kế tiếp là tính tổng $S(k) = (p^0 + p^1 + \ldots + p^k)$ như thế nào cho nhanh. Ta có thể áp dụng phương pháp tính luỹ thừa nhanh vào việc tính tổng này. Bằng cách chia bài toán cần tính thành 2 bài toán con giống nhau, ta sẽ tiết kiệm được thời gian.

Nếu k là số lẻ thì tổng $(p^0 + p^1 + \ldots + p^k)$ có thể chia thành hai phần mỗi phần có (k+1)/2 số hạng: $(p^0 + p^1 + \ldots + p^{(k+1)/2-1}) + p^{(k+1)/2}(p^0 + p^1 + \ldots + p^{(k+1)/2-1})$

Vậy nếu k là số lẻ thì $S(k) = S((k+1)/2 - 1)*(1 + p^{(k+1)/2}) = S((k-1)/2)*(1 + p^{(k+1)/2})$.

Nếu k là số chẵn, ta bỏ bớt phần tử cuối cùng, phần còn lại sẽ là số chẵn, nói cách khác: nếu k chẵn,

$S(k) = S(k-1) + p^k \quad = p^k + S((k-2)/2)*(1 + p^{(k)/2})$

$\quad\quad = p^k + S((k-2)/2)*(1 + p^{(k+1)/2})$

Trường hợp suy biến, k = 0, S(0) = 1.

$p^k$ có thể được tính nhanh bằng giải thuật tính luỹ thừa nhanh như phần 1.

```
long long mod_power(long long p, long long k) {
      long long r = 1;
      while (k > 0) {
            if (k%2)
                  r = (r*p)%MOD;
            k >>= 1;
            p = (p*p)%MOD;
      }
      return r;
}
long long sum(long long p, long long k)  {
      if (!k) return 1;
      long long temp = mod_power(p,(k+1)>>1) + 1);
      if (k&1)   //k lẻ, nên (k-1)/2 = k/2
            return temp*sum(p, k >> 1)%MOD;
      return (mod_power(p, k) +
                  temp*sum(p, (k-2)>>1))%MOD;
}
```

Ta cũng có thể cải tiến 1 chút để việc tính toán $S = p^0 + p^1 + \ldots + p^k$ nhanh hơn. Để đơn giản việc phân tích ta bỏ bớt phần tử $p^0$ ra khỏi S để số phần tử trong tổng = k.

Nếu đặt $S1 = + p^1 + \ldots + p^k$ thì S = 1 + S1.

Ta sẽ phân tích cách tính S1 bằng phương pháp lặp. Lập luận tương tư như trên, nếu k là số chẵn:

$S1(k) \quad = (p^1 + p^2 + \ldots + p^{k/2}) + p^{k/2}(p^1 + p^2 + \ldots + p^{k/2})$

$\quad\quad = S1(k/2)*(1+p^{k/2})$

Nếu k là số lẻ

$S1(k) = S1(k-1) + p^k$

Giả sử ta đang có S1(i) thì $S1(2i) = S1(i)*(1 + p^i)$ và

$S(2i + 1) = S1(i)*(1 + p^i) + p^{2i+1} = S1(i)*(1 + p^i) + p*(p^i)$^2

Bắt đầu từ S1(0) = 0, ta tìm cách tính S1(k) bằng cách tăng k với phép *2 hoặc *2 và + 1. Ví dụ nếu k = 6, thì từ 0 -> 1 -> 3 -> 6.

Chuỗi các phép biến đổi này chính là chuỗi nhị phân biểu diễn k. Giả sử k = 6 = 110.

Đầu tiên ta có i = 0, gặp bít đầu tiên là bít 1 ta tăng i = 2*i + 1 = 1. Kế tiếp gặp bít 1 nữa tăng i = 2*i + 1 = 3, bít cuối cùng là bít 0, ta tăng i = 2*i = 6.

Đến đây ta đã nghĩ ra cách để tính S1(k) từ S1(0) rồi phải không. Nếu chưa thì tiếp tục nhé.

Giả sử chuỗi nhị phân biểu diễn k là $b_q\ b_{q-1} \ldots b_1$.

Khởi tạo: S1 = 0; pi = 1;

```
for (int j = q; j >= 1; j--) {
        S1 = S1*(1 + pi);
        pi = pi*pi;
        if (b[j] == 1) {
            pi = pi*p;
            S1 = S1 + pi;
        }
}
```

Để tìm chuỗi bq bq-1 … b1 ta áp dụng phương pháp đổi số thập phân thành nhị phân. Toàn bộ giải thuật tính $S = p^0 + p^1 + \ldots + p^k$ được trình bày như sau:

```
long long sum(long long p, long long k) {
        if (!k)
            return 1;
        int b[33], q = 0;
        long long kk = k;
        while (kk > 0) {
            b[++q] = kk&1;
            kk >>= 1;
        }
        long long S1 = 0, pi = 1;
        for (int  j = q; j >= 1; j--) {
            S1 = S1*(1 + pi) % MOD;
            pi = pi*pi % MOD;
            if (b[j]) {
                pi = pi*p % MOD;
                S1 = (S1 + pi) % MOD;
            }
        }
        return (S1 + 1) % MOD;
}
```

Ta có thể áp dụng cách này để giải bài **1142 - Summing up Powers (II)** trên Light Oj (http://lightoj.com/volume_showproblem.php?problem=1142).

Như thế ta đã đủ thông tin để giải bài tổng ước số.

```
//Tìm các số nguyên tố từ 2 đến sqrt(n) lưu vào mảng hay
vector pp
// sqrt(2^31-1) = 46340.95

#define MAXPRIME 46342
int prime[MAXPRIME];
vector<int> pp;

void erathosthenes() {
  prime[0] = prime[1] = 0;
  for (long long i = 2; i < MAXPRIME; i++)
    prime[i] = 1;

  for (long long i = 2; i*i < MAXPRIME; i++) {
    if (prime[i])
      for (long long j = i*i; j < MAXPRIME; j += i)
        prime[j] = 0;
  }
  for (long long i = 2; i < MAXPRIME; i++)
    if (prime[i])
      pp.push_back(i);
}


//Phân tích n thành các số nguyên tố và tính tổng
long long result = 1;
for (int i = 0; i < pp.size(); i++) {
     int k = 0;
     while (n%pp[i] == 0) {
          n /= pp[i];
          k++;
     }
     if (k > 0)
          result = result*sum(pp[i], k*m) %MOD;
}
if (n > 1)
     result = result*sum(n, m) % MOD;
return result;
```

Hãy dừng lại tại đây và chuẩn bị để submit code của bạn.

Nếu vẫn chưa được bạn có thể tham khảo đoạn code sau:

```
#include <iostream>
#include <vector>
#include <string.h>
#include <math.h>
using namespace std;

//1054 - Efficient Pseudo Code
#define MAXPRIME 46342
#define MOD 1000000007

char prime[MAXPRIME];
vector<long long> pp;

void erathosthens() {
   for (long long i = 2; i < MAXPRIME; i++)
     prime[i] = 1;

   for (long long i = 2; i*i < MAXPRIME; i++) {
     if (prime[i]) {
        for (long long j = i+i; j < MAXPRIME; j += i) {
          prime[j] = 0;
        }
     }
   }
   for (long long i = 2; i < MAXPRIME; i++)
     if (prime[i])
        pp.push_back(i);
}

long long sum(long long p, long long k) {
   if (!k)
     return 1;
   int b[34], q = 0;
   long long kk = k;
   while (kk > 0) {
     b[++q] = kk&1;
     kk >>= 1;
   }
   long long S1 = 0, pi = 1;
   for (int  j = q; j >= 1; j--) {
     S1 = S1*(1 + pi) % MOD;
     pi = pi*pi % MOD;
     if (b[j]) {
        pi = pi*p % MOD;
        S1 = (S1 + pi) % MOD;
     }
   }
   return (S1 + 1) % MOD;
}
long long process(long n, long long m) {
   long long result = 1;
```

```
   for (long long i = 0; i < pp.size(); i++) {
      long long k = 0;
      while (n % pp[i] == 0) {
         k++;
         n /= pp[i];
      }
      if (k > 0)
         result = result*sum(pp[i], k*m) % MOD;
   }
   if (n > 1)
      result = result*sum(n, m) % MOD;

   return result;
}

int main()
{
   long nCase, n, K, m;
   cin >> nCase;
   string s;
   erathosthens();
   for (long long no = 1; no <= nCase; no++) {
      cin >> n >> m;
      cout << "Case " << no << ": "
         << process(n, m) << endl;
   }
   return 0;
}
```

## 3.5 Số dư của (a/b) % MOD

Trong các tính chất trong phần đồng dư phía trên không có nhắc đến (a/b) % MOD. Ta sẽ xem xét vấn đề này.

Nếu b và MOD nguyên tố cùng nhau (ước số chung lớn nhất của chúng là 1), sẽ tồn tại 1 số nguyên c sao cho b*c % MOD = 1. Ta gọi c là số nghịch đảo của b theo modulo MOD và ký hiệu $b^{-1}$ = c. Cho trước b và MOD ta có thể áp dụng **giải thuật Euclide mở rộng** để tìm $b^{-1}$ (sẽ đề cập đến giải thuật này sau).

Như vậy nếu $b^{-1}$ là nghịch đảo của b thì (a/b) % MOD = $a*b^{-1}$ % MOD.

**Định lý Fermat:** Nếu a và MOD nguyên tố cùng nhau thì $a^{\varphi(n)}$ = 1 (mod n), hay $a^{\varphi(n)}$ chia n dư 1 hay $a^{\varphi(n)}$ % n = 1.

Hàm φ(n) là số các số nguyên dương <=n và nguyên tố cùng nhau với n. Ví dụ φ(9) = 6 vì các số nguyên tố cùng nhau với 9 là: 1, 2, 4, 5, 7, 8.

Nếu p là số nguyên tố thì φ(p) = p-1.

Như vậy nếu p là 1 số nguyên tố và a < p thì a nguyên tố cùng nhau với p và nghịch đảo của a theo modulo n là $a^{\varphi(n)-1} = a^{p-2}$. Ta sẽ ứng dụng tính chất để giải các bài toán.

**1067 – Combinations (Light Oj)** tính tổ hợp C(n, k) % MOD với MOD = 1000003 và 0 <= k <= n <= 1000000.

Bài này có n khá lớn nên ta không thể áp dụng quy hoạch động (theo tam giác PASCAL) để tính C(n, k). Hơn nữa ta cũng không thể áp dụng cách tính trực tiếp bằng các vòng lặp for.

Do 1000003 là số nguyên tố nên ta có thể dễ dàng áp dụng các phép chia.

Ta đã biết rằng:

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

Nếu đặt A = (n-1)! * k! thì A nguyên tố cùng nhau với MOD vì MOD là số nguyên tố nên các số nhỏ MOD đều nguyên tố cùng nhau với MOD. A là tích của các số nguyên tố cùng nhau với MOD nên A cũng nguyên tố cùng nhau với MOD.

Như thế, nghịch đảo của A là $A^{MOD - 2}$.

Vậy C(k, n) % MOD = (n! * $A^{MOD - 2}$) % MOD.

Áp dụng kết quả này ta có giải thuật như sau:

- Tính trước các F[i] = 1! (0 <= i <= MAX = 1000000)
- Tính nghịch đảo của A là $A^{-1} = A^{MOD-2}$ (áp dụng giải thuật tính luỹ thừa nhanh ở phần trên)
- Tính n! * $A^{-1}$ = F[n]*$A^{-1}$.

Bạn thử submit xem sao.

Nếu không, xem giải thuật ở đây:

```
#include <iostream>
using namespace std;

//1067 - Combinations
#define MAX 1000001
#define MOD 1000003

long long F[MAX];

void fact() {
   F[0] = 1;
   for (int i = 1; i < MAX; i++)
      F[i] = F[i-1]*i % MOD;
}

long long mod_power(long long n ,long long p) {
   long long r = 1;
   while (p > 0) {
      if (p&1)
         r = r*n % MOD;
      p >>= 1;
      n = n*n % MOD;
   }
   return r;

}

int main()
{

   long nCase, n, k;
   cin >> nCase;
   fact();
   for (long long no = 1; no <= nCase; no++) {
      cin >> n >> k;
      long long A = F[n-k]*F[k] % MOD;
      long long r = F[n]*mod_power(A, MOD - 2) %MOD;
      cout << "Case " << no << ": " << r << endl;
   }
   return 0;
}
```

# 4 Cây phân đoạn (Segment Tree/Interval Tree)

## 4.1 Bài toán tính trung bình nhanh

Xét bài toán "1183 - Computing Fast Average" trên Light Oj (http://lightoj.com/volume_showproblem.php?problem=1183). Cho 1 mảng số nguyên n phần tử $1 \leq n \leq 10^5$. Chỉ số mảng bắt đầu từ 0. Đầu tiên các phần tử đều có giá trị 0. Bạn phải thực hiện q ($1 \leq q \leq 50000$) thao tác thuộc 1 trong 2 phép toán sau đây:

1. **1 i j v** – gán giá trị các phần tử từ i đến j bằng v ($0 \leq v \leq 10000$).
2. **2 i j** – tính giá trị trung bình của các phần tử từ i đến j.

Nếu làm như thông thường, có nghĩa là sử dụng vòng lặp từ i đến j để gán hoặc cộng giá trị, thì ta sẽ bị **time limit exceeded**. Những bài toán dạng này sẽ được giải quyết hiệu quả nếu sử dụng một cấu trúc dữ liệu phù hợp: Intertal Tree.

**Ý tưởng chính** của interval tree là chia mảng dữ liệu thành những đoạn (interval), và tổ chức các đoạn thành một cây phân cấp. Mỗi nút quản lý một đoạn từ chỉ số **left** đến **right**. Nút gốc quản lý toàn bộ mảng, từ chỉ số từ 0 đến n-1.

Mỗi khi thực hiện thao tác trên một đoạn **[i, j]**, và nút **node** đang quản lý đoạn này, ta chỉ cần thực hiện lưu trữ hoặc truy vấn trên nút **node** mà không cần phải đi đến từng phần tử từ i đến j.

## 4.2 Cài đặt interval tree

Interval có thể cài đặt khá đơn giản bằng một mảng các nút **nodes**. Quan hệ **cha-con giữa các nút** và **đoạn quản lý** của từng nút được **ngầm định** như sau:

- Thông tin của nút i được lưu trong phần tử thứ i của mảng các nút: nodes[i].
- Nút gốc là phần tử 1 trong mảng nodes.
- Nút i có 2 con là nodes[2*i] và nodes[2*i + 1]
- Nút gốc quản lý đoạn từ 0 đến n-1 (hay từ 1 đến n tuỳ theo mảng dữ liệu được tổ chức như thế nào. Trong bài toán tính trung bình ta cho nút gốc quản lý từ đoạn 0 đến n-1)
- Nếu nút I quản lý đoạn từ left đến right thì nút con trái 2*i quản lý đoạn từ left đến mid và nút con phải 2*i+1 quản lý đoạn từ mid+1 đến right với mid = (left+right)/2.
- Nút lá quản lý 1 phần tử duy nhất của mảng dữ liệu (left =right).

Chú ý: do đã ngầm định như thế nên ta không cần phải lưu trữ các con trỏ chỉ đến con trái, con phải hay nút cha của một nút. Ta cũng không cần phải lưu trữ đoạn quản lý của một nút.

Thông tin của một nút thông thường gồm 2 thành phần: **thông tin riêng** và **thông tin tổng hợp của cả đoạn**. Khi thao tác trên một nút, ta lưu trữ vào phần thông tin riêng, cập nhật thông tin tổng hợp và lan truyền thông tin tổng hợp lên nút cha. Thông tin tổng hợp của một nút luôn phản ảnh thông tin của toàn bộ đoạn mà nút quản lý.

```
struct NodeType {
   thông tin riêng
   thông tin chung
};
NodeType nodes[MAX*4];
//MAX là kích thước của mảng dữ liệu
```

**Các thao tác trên interval tree:**

```
void init_tree(int node, int left, int right) {
   //khởi tạo chung cho 1 nút
   if (left == right) {
      //khởi tạo nút lá ở đây
      return;
   }
   //khởi tạo nút trong ở đây
   int mid = (left + right)/2
   init_tree(2*node, left, mid);
   init_tree(2*node+1, mid+1, right);
   //tổng hợp thông tin từ 2 nút con ở đây
}
```

*a. <u>Khởi tạo</u>*

Để gọi khởi tạo cây ta gọi: **init_tree(1, 0, n-1)**

*b. <u>Cập nhật & truy vấn</u>*

Interval tree hoạt động theo cơ chế "lazy update" có nghĩa là chỉ cập nhật khi cần. Khi cập nhật nội dung toàn bộ đoạn mà nút đang quản lý với một giá trị giống nhau, nó chỉ cập nhật nội dung này vào phần thông tin riêng mà không đi xuống cập nhật các nút con của nó. **Đây chính là mấu chốt của việc tăng tốc độ** của interval tree. Nếu cài đặt không khéo ở chỗ này, interval tree còn tệ hơn sử dụng vòng lặp !

Nếu ta dừng việc cập nhật nội dung của một đoạn tại nút cha mà không đi xuống các nút con, cháu thì khi truy vấn một đoạn do nút con quản lý thì thế nào ? Nút con hoàn toàn không hay biết gì về việc cập nhật nút cha của nó cả. Cơ chế "lazy update" hoạt đông như thế này: Khi cập nhật nội dung/truy vấn thông tin một nút con mà nút **cha của nó còn đang giữ thông tin riêng** (chưa truyền xuống) thì ta thực hiện việc lan truyền thông tin xuống, trước khi cập nhật/truy vấn. Vì thế ở mỗi nút nên có 1 dấu hiệu gì đó để báo là nút đó đang giữ thông tin chưa lan truyền.

## 4.3 Giải bài tính trung bình nhanh

Để có được trung bình của đoạn, ta cần phải có tổng giá trị các phần tử trong đoạn. Số phần tử trong một đoạn dễ dàng tính được bằng j – i +1. Vì thế **thông tin tổng hợp** của một nút là tổng các phần tử trong đoạn nút quản lý. **Thông tin riêng** của nút là giá trị được gán cho toàn đoạn mà nút quản lý. Do giá trị v gán cho các phần tử >= 0, nên ta có thể sử dụng luôn thông tin riêng này để cho biết là nút có đang giữ giá trị chưa lan truyền xuống hay không. Nếu v = -1 chẳng hạn, thì nút không giữ giá trị chưa lan truyền, ngược lại nó đang giữ giá trị chưa lan truyền.

**Khai báo:**

```
#define MAX 100005
struct NodeType {
   int value;
   int sum;
};
NodeType nodes[MAX*4];
```

**Khởi tạo:**

```
void init_tree(int node, int left, int right) {
   nodes[node].value = -1; nodes[node].sum = 0;
   if (left == right) {
      return;
   }
   int mid = (left + right)/2;
   init_tree(2*node, left, mid);
   init_tree(2*node+1, mid+1, right);
}
```

**Cập nhật 1 đoạn [u,v] với giá trị val:**

```
void update(int node, int left, int right,
            int u, int v, int val) {
   if (u > right || v < left)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= left && v >= right) {
      nodes[node].value = val;
      nodes[node].sum = (right-left+1)*val;
      return;
   }
/*Đây là trường hợp cập nhật một đoạn con của đoạn đang quản
lý. Trước hết phải lan truyền giá trị chưa cập nhật của nó
(node) xuống các con của nó*/
   int mid = (left+right)/2;
   if (nodes[node].value != -1) {
      int value = nodes[node].value;
      nodes[2*node].value = value;
      nodes[2*node].sum = (mid-left+1)*value;

      nodes[2*node+1].value = value;
      nodes[2*node+1].sum = (right-mid)*value;
      nodes[node].value = -1; //đã lan truyền xong
   }
   update(2*node, left, mid, u, v, val);
   update(2*node+1, mid+1, right, u, v, val);
   //tổng hợp tổng từ 2 con
   nodes[node].sum = nodes[2*node].sum +
                     nodes[2*node+1].sum;
}
```

**Truy vấn tổng 1 đoạn [u, v]:**

```
int query(int node, int left, int right,
          int u, int v) {
   if (u > right || v < left)
      return 0; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= left && v >= right)
      return nodes[node].sum;
/*Đây là trường hợp truy vấn một đoạn con của đoạn đang quản
lý. Trước hết phải lan truyền giá trị chưa cập nhật của nó
(node) xuống các con của nó*/
   int mid = (left+right)/2;
   if (nodes[node].value != -1) {
      int value = nodes[node].value;
      nodes[2*node].value = value;
      nodes[2*node].sum = (mid-left+1)*value;

      nodes[2*node+1].value = value;
      nodes[2*node+1].sum = (right-mid)*value;
      nodes[node].value = -1; //đã lan truyền xong
   }
   return query(2*node, left, mid, u, v) +
            query (2*node+1, mid+1, right, u, v);
}
```

**Xử lý kết quả:**

Nếu tổng chia hết cho số phần tử in tổng. Nếu không, ta phải rút gọn phân số. Tìm ước số chung lớn nhất (USCLN) bằng giải thuật Euclide và chia cả tử và mẫu cho USCLN. Giải thuật Euclide tìm USCLN được cho như bên dưới.

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

Nào các bạn thử submit đi nhé !

Nếu gặp khó khăn có thể tham khảo code mẫu:

```
//1183 - Computing Fast Average

#include <stdio.h>
#include <string.h>
#include <math.h>

#define MAX 100005
struct NodeType {
   int value;
   int sum;
};
NodeType nodes[MAX*4];

void init_tree(int node, int left, int right) {
   nodes[node].value = -1; nodes[node].sum = 0;
   if (left == right) {
      return;
   }
   int mid = (left + right)/2;
   init_tree(2*node, left, mid);
   init_tree(2*node+1, mid+1, right);
}

void update(int node, int left, int right,
            int u, int v, int val) {
   if (u > right || v < left)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= left && v >= right) {
      nodes[node].value = val;
      nodes[node].sum = (right-left+1)*val;
      return;
   }
   int mid = (left+right)/2;
   if (nodes[node].value != -1) {
      int value = nodes[node].value;
      nodes[2*node].value = value;
      nodes[2*node].sum = (mid-left+1)*value;

      nodes[2*node+1].value = value;
      nodes[2*node+1].sum = (right-mid)*value;
      nodes[node].value = -1; //đã lan truyền xong
   }
   update(2*node, left, mid, u, v, val);
   update(2*node+1, mid+1, right, u, v, val);
   //tổng hợp tổng từ 2 con
   nodes[node].sum = nodes[2*node].sum +
                     nodes[2*node+1].sum;
}

int query(int node, int left, int right,
          int u, int v) {
```

```
   if (u > right || v < left)
      return 0; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= left && v >= right)
      return nodes[node].sum;
/*Đây là trường hợp truy vấn một đoạn con của đoạn đang quản
lý. Trước hết phải lan truyền giá trị chưa cập nhật của nó
(node) xuống các con của nó*/
   int mid = (left+right)/2;
   if (nodes[node].value != -1) {
      int value = nodes[node].value;
      nodes[2*node].value = value;
      nodes[2*node].sum = (mid-left+1)*value;

      nodes[2*node+1].value = value;
      nodes[2*node+1].sum = (right-mid)*value;
      nodes[node].value = -1; //đã lan truyền xong
   }
   return query(2*node, left, mid, u, v) +
            query (2*node+1, mid+1, right, u, v);
}


int gcd(int a, int b) {
   while (b != 0) {
      int r = a%b;
      a = b;
      b = r;
   }
   return a;
}

int main() {
   int nCase, n, q, u, v, type, val;
   scanf("%d", &nCase);
   for (int no = 1; no <= nCase; no++) {
      scanf("%d %d", &n, &q);
      init_tree(1, 0, n-1);
      printf("Case %d:\n", no);
      for (int i = 1; i <= q; i++) {
         scanf("%d", &type);
         if (type == 1) {
            scanf("%d %d %d", &u, &v, &val);
            update(1, 0, n-1, u, v, val);
         } else {
            scanf("%d %d", &u, &v);
            int r = query(1, 0, n-1, u, v);
            int k = v - u + 1;
            int g = gcd(r, k);
            r /= g;
            k /= g;
            if (k > 1)
               printf("%d/%d\n", r, k);
            else
```

```
                printf("%d\n", r);
            }
        }
    }
    return 0;
}
```

Chú ý:

- Sử dụng **scanf, printf** thay cho **cin/cout** để tăng tốc độ đọc/ghi dữ liệu. Nếu không **time limit exceeded** !

## 4.4 Bài học kinh nghiệm

- Để áp dụng được kỹ thuật interval tree, ta phải mô hình bài toán về dạng cập nhật/truy vấn trên một đoạn.
- Phân biệt thông tin riêng của nút và thông tin tổng hợp.
- Xác định cách tính thông tin tổng hợp của 1 nút từ thông tin tổng hợp của các nút con
- Chú ý cơ chế "lazy update": dừng thao tác sớm nhất có thể, hạn chế đi xuống các đoạn con không cần thiết.
- Sử dụng các hàm đọc/ghi dữ liệu nhanh.

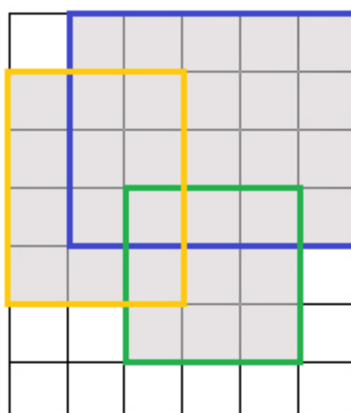
## 4.5 Các bài tương tự

- 1082 - Array Queries (Light Oj)
- 1135 - Count the Multiples of 3 (Light Oj)
- 1164 - Horrible Queries (Light Oj)
- …

## 4.6 Các bài nâng cao

Có khá nhiều bài cần phải phân tích thật kỹ mới có thể mô hình hoá về Interval Tree. Ta lần lượt xem 1 số dạng bài như thế.

**1120 - Rectangle Union (Light Oj )**
http://lightoj.com/volume_showproblem.php?problem=1120)



Cho một số hình chữ nhật có cạnh song song với các trục toạ độ, ta cần tìm tổng diện tích bị các hình chữ nhật che phủ. Trong ví dụ hình bên dưới, tổng diện tích bị che phủ là 31 ô.

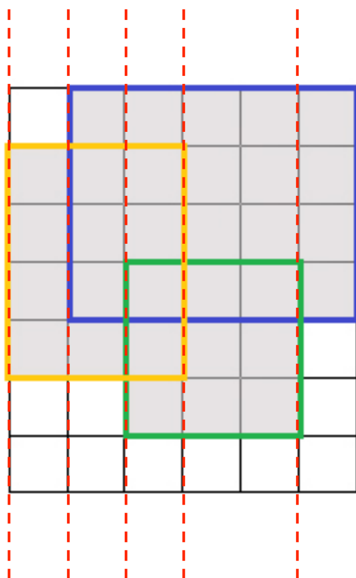
Mỗi hình chữ nhật được mô tả bằng 4 số nguyên x1, y1, x2, y2 trong đó (x1, y1) là góc dưới trái còn (x2, y2) là góc trên phải. Trong hình trên hình chữ nhật vàng có góc dưới trái là (0, 2), góc trên phải là (3, 6).

Đầu vào: n hình chữ nhật ($1 \le n \le 30000$), mỗi hình là 4 số nguyên $x_1, y_1, x_2, y_2$ ($0 \le x_1, y_1, x_2, y_2 \le 10^9$, $x_1 < x_2$, $y_1 < y_2$).

Đầu ra: tổng diện tích bị phủ.

Bài này không thể áp dụng ngay interval tree được mà phải phân tích và mô hình hoá nó về các thao tác trên đoạn.

Để tìm diện tích bị các hình chữ nhật bao phủ ta chia không gian bằng các cạnh đứng của các hình chữ nhật và tính tổng phần bị bao phủ được ngăn cách bằng các cạnh đứng. Gọi $x_1$, $x_2$, …, $x_{2n}$ là các toạ độ của các cạnh hình chữ nhật đã được sắp xếp tăng dần. Tổng diện tích bị phủ = diện tích bị phủ từ $x_1$ đến $x_2$ + diện tích bị phủ từ $x_2$ đến $x_3$ + … + diện tích bị phủ từ $x_{2n-1}$ đến $x_{2n}$. Độ rộng phần bị giới hạn bởi $x_i$ và $x_{i+1}$ là $(x_{i+1} – x_i)$. Nếu có thêm được tổng chiều cao của các vùng bị phủ (trong 1 phần có thể có 1 hoặc nhiều vùng bị phủ), ta nhân với độ rộng sẽ có được diện tích của vùng.



Nếu gọi $y_{min}$ là toạ độ của cạnh thấp nhất và $y_{max}$ là toạ độ cạnh ngang cao nhất trong tất cả các hình chữ nhật. Ta sẽ quản lý tổng chiều cao của các vùng bị bao phủ trong từng phần bị chia cách bằng các cạnh đứng. Ví dụ, trong hình trên, phần đầu tiên có tổng chiều cao bị bao phủ = 4 (chiều cao của hình chữ nhật vàng), phần thứ hai có tổng chiều cao = 5, phần thứ 3 có tổng chiều cao = 6, … Ta thấy rằng tổng chiều cao trong từng phân vùng sẽ thay đổi. Sự thay đổi này là do khi chuyển vùng, ta bắt đầu 1 hình chữ nhật mới, hoặc kết thúc một hình chữ nhật cũ. Nếu ta gặp một cạnh là cạnh bắt đầu (cạnh trái) của hình chữ nhật, toàn bộ chiều cao từ y1 đến y2 sẽ bị hình chữ nhật đang xét bao phủ. Ngược lại, nếu ta gặp một cạnh phải của hình chữ nhật, tổng chiều cao bị bao phủ có thể bị giảm cũng có thể không (do một ô có thể bị nhiều hình chữ nhật bao phủ).

Đến đây, bạn đã đoán ra cách dùng interval tree để quản lý việc bao phủ này chưa?

Ta sẽ dùng một interval tree để quản lý một đoạn từ $y_{min}$ đến $y_{max}$ (thực tế ta nên quản lý các ô từ $y_{min}$ đến $y_{max-1}$, thay quản lý vì các toạ độ). Đầu tiên tất cả các ô  đều không bị bao phủ nên có giá trị bằng 0.

Sau khi sắp xếp các cạnh đứng theo thứ tự tăng dần, ta lần lượt duyệt qua các cạnh, mỗi khi gặp một cạnh bắt đầu ta tăng giá trị của các ô từ y1 đến y2 lên 1. Mỗi khi gặp 1 cạnh kết thúc, ta ta giảm giá trị của các ô từ y1 đến y2 đi 1. Sau mỗi thao tác tăng hay giảm ta phải cập nhật số ô bị phủ (có giá trị > 0) của các nút trong interval tree. Để đếm số ô bị phủ ta truy vấn trong đoạn từ $y_{min}$ đến $y_{max}$ có bao nhiêu ô bị phủ. Đoạn này do nút gốc quản lý, nên ta chỉ cần lấy thông tin tổng hợp của nút gốc là đủ.

Tóm lại bài toán này sẽ được mô hình hoá về các thao tác trên interval tree như sau:

- Khởi tạo, các ô từ $y_{min}$ đến $y_{max}$ có giá trị 0.

- Tăng hoặc giảm giá trị của các ô trong một đoạn.

Ở mỗi nút, thông tin riêng là giá trị tăng hay giảm (+1/-1), thông tin tổng hợp là số các ô có giá trị > 0.

Khai báo

```
//10^9
#define MAX 1000000001
struct NodeType {
   int value;
   int sum;
};
NodeType nodes[MAX*4];
```

Khởi tạo:

value = 0; sum = 0

Tăng 1 đoạn:

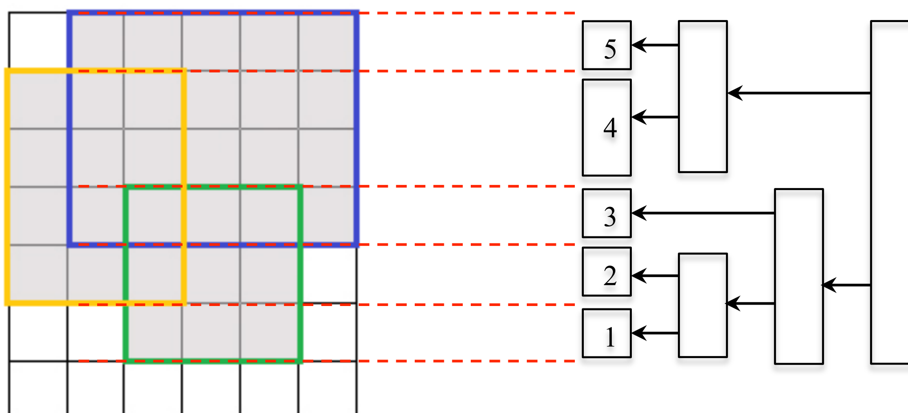
```
void increase(int node, int left, int right,
              int u, int v) {
   if (u > right || v < left)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= left && v >= right) {
      nodes[node].value++;
      nodes[node].sum = left - right + 1;
      return;
   }
   increase(2*node, left, mid, u, v);
   increase(2*node+1, mid+1, right, u, v);
   //tổng hợp tổng từ 2 con
   nodes[node].sum = nodes[2*node].sum +
                     nodes[2*node+1].sum;
}
```

Giảm 1 đoạn:

```
void decrease(int node, int left, int right,
              int u, int v) {
   if (u > right || v < left)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= left && v >= right) {
      nodes[node].value--; //1 đoạn chỉ có thể >= 0
      if (nodes[node].value == 0)
         if (left != right)
            nodes[node].sum = nodes[2*node].sum +
                              nodes[2*node+1].sum;
         else
            nodes[node].sum = 0;
      return;
   }
   //không cần lan truyền
   decrease(2*node, left, mid, u, v);
   decrease(2*node+1, mid+1, right, u, v);

   //tổng hợp tổng từ 2 con
   nodes[node].sum = nodes[2*node].sum +
                     nodes[2*node+1].sum;
}
```

Tuy nhiên do ($0 \le x_1, y_1, x_2, y_2 \le 10^9$) nên nếu quản lý trực tiếp từng ô từ $y_{min}$ đến $y_{max}$, sẽ tốn khá nhiều bộ nhớ và thời gian. Ta có thể giảm số ô trong đoạn từ $10^9$ xuống còn 30,000*2 = 60,000 khoảng thôi (số hình chữ nhật $n \le 30,000$ nên có 2*n cạnh ngang => số khoảng = 2*n - 1). Mỗi khoảng được giới hạn bằng 2 giá trị ymin và ymax. Trong ví dụ bên dưới, ta có 3 hình chữ nhật nên có 6 cạnh ngang và tạo thành 5 khoảng. Mỗi khoảng bây giờ đóng vai trò như 1 phần tử trong mảng dữ liệu trong bài toán "Fast Average".

Phương pháp này gọi là rời rạc hoá dữ liệu. Để quản lý các khoảng ta chỉ cần một mảng lưu các cao độ tăng dần, horizontals. Trong ví dụ trên, nếu tính dừ dưới lên ta lưu: 1, 2, 3, 4, 6 và 7 tương ứng với 5 khoảng. Interval tree để quản lý đoạn từ $y_{min}$ đến $y_{max}$ như sau:



Khai báo:

```
//2*30000
#define MAX 60001
struct NodeType {
   int value;
   long long sum;
   long long ymin, ymax;
};
NodeType nodes[MAX*4];
long long height[MAX];

struct Rectangle {
   long long x1, y1, x2, y2;
};

//Danh sách các hình chữ nhật
Rectangle rects[30000];

struct Vert {
   long long x;
   bool begin;
   int id;
};

//Các cạnh đứng
Vert verticals[MAX];
long long horizontals[MAX];
```

Khởi tạo:

Với mỗi nút ta lưu lại khoảng ymin và ymax mà nó quản lý. Chú ý khoảng [ymin, ymax] tương ứng với một đoạn dữ liệu nào đó trong mảng dữ liệu. Trong bài giải này, horizontal có chỉ số tính từ 0, chỉ số nút trên interval tree tính từ 1 nên khoảng i có ymin = horizontal[left-1] và ymax = horizontal[left].

```
void init_tree(int node, int left, int right) {
   nodes[node].value = 0; nodes[node].sum = 0;
   if (left == right) {
      nodes[node].ymin = horizontals[left-1];
      nodes[node].ymax = horizontals[left];
      return;
   }
   int mid = (left + right)/2;
   init_tree(2*node, left, mid);
   init_tree(2*node+1, mid+1, right);

   nodes[node].ymin = nodes[2*node].ymin;
   nodes[node].ymax = nodes[2*node+1].ymax;
}
```

Tăng 1 đoạn:

Trong các hàm tăng/giảm, ta sử dụng ymin, ymax để xác định khoảng quản lý nút thay vì đoạn trong mảng dữ liệu. Khi tăng một đoạn bao phủ hoàn toàn khoảng của một nút, chắc chắn khoảng này bị phủ ít nhất 1 hình chữ nhật nên ta không cần đi xuống các nút con của nó. Các con của nó sẽ được cập nhật dựa vào cơ chế “lazy update” nghĩa là chỉ cập nhật khi ta cập nhật/truy vấn con/cháu của nó.

```
void increase(int node, int left, int right, int u, int v) {
   if (u >= nodes[node].ymax || v <= nodes[node].ymin)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= nodes[node].ymin && v >= nodes[node].ymax) {
      nodes[node].value++;
      nodes[node].sum = nodes[node].ymax - nodes[node].ymin;
      return;
   }
   int mid = (left + right)/2;
   increase(2*node, left, mid, u, v);
   increase(2*node+1, mid+1, right, u, v);

   //tổng hợp tổng từ 2 con
   if (nodes[node].value == 0)
      nodes[node].sum = nodes[2*node].sum +
                        nodes[2*node+1].sum;
}
```

Giảm 1 đoạn:

Khi giá trị của một đoạn bị giảm về 0, số ô bị phủ của khoảng này phụ thuộc vào 2 khoảng của 2 con của nó. Nếu nó là nút lá, sô ô bị phủ chắc chắn sẽ là 0.

```
void decrease(int node, int left, int right, int u, int v) {
   if (u >= nodes[node].ymax || v <= nodes[node].ymin)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= nodes[node].ymin && v >= nodes[node].ymax) {
      nodes[node].value--; //value của 1 đoạn chỉ có thể >= 0
      if (nodes[node].value == 0) {
         if (left != right)
            nodes[node].sum = nodes[2*node].sum +
                              nodes[2*node+1].sum;
         else
            nodes[node].sum = 0;
      }
      return;
   }
   int mid = (left +right)/2;
   decrease(2*node, left, mid, u, v);
   decrease(2*node+1, mid+1, right, u, v);

   //tổng hợp tổng từ 2 con
   if (nodes[node].value == 0)
      nodes[node].sum = nodes[2*node].sum +
                        nodes[2*node+1].sum;
}
```

Bài toán này có 2 tính chất đặc biệt làm nó dễ:

- Giá trị của một nút không bao giờ âm vì ta luôn tăng 1 đoạn (khi gặp cạnh bắt đầu của 1 hình chữ nhật) trước khi giảm giá trị của nó (khi gặp cạnh kết thúc của cùng hình chữ nhật)
- 1 ô chỉ cần bị phủ ít nhất bởi 1 hình chữ nhật là được tính. Trong một số bài toán mở rộng khác, ta cần phải đếm số ô bị phủ ít nhất k hình chữ nhật như trong bài "**1204 - Weird Advertisement**". Đối với các bài như thế, mỗi khi tăng/giảm một đoạn ta không thể dừng ngay được mà phải tiếp tục đi xuống để tổng hợp số ô bị phủ trong các con của nó.

Nhiệm vụ còn lại của chúng ta sắp xếp các cạnh đứng theo thứ tự tăng dần và tạo các khoảng từ các cạnh ngang.

Sắp xếp cạnh đứng và tạo các khoảng từ các cạnh ngang. Các cạnh có tung độ bằng nhau chỉ giữ lại 1 giá trị.

```
bool my_compare(const Vert& a, const Vert& b) {
    return a.x < b.x;
}

int create_interval(int n) {
    //Cạnh ngang <y, id-rect> chỉ số của hình chữ nhật
    //dùng để cập nhật từ toạ độ sang chỉ số khoảng
    static long long ys[MAX];

    for (int i = 0; i < n; i++) {
        verticals[2*i].x = rects[i].x1;
        verticals[2*i].begin = true;
        verticals[2*i].id = i;

        verticals[2*i+1].x = rects[i].x2;
        verticals[2*i+1].begin = false;
        verticals[2*i+1].id = i;

        ys[2*i] = rects[i].y1;
        ys[2*i+1] = rects[i].y2;
    }

    sort(verticals, verticals + 2*n, my_compare);
    sort(ys, ys + 2*n);

    //unique
    int k = 0; int y = -1;
    for (int i = 0; i < 2*n; i++) {
        if (y != ys[i]) {
            horizontals[k] = ys[i];
            y = ys[i];
            k++;
        }
    }
    return k-1;
}
```

Chương trình mẫu:

```
//1120 - Rectangle Union

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <algorithm>
#include <iostream>
#include <map>
using namespace std;

//2*30000
#define MAX 60001
struct NodeType {
   int value;
   long long sum;
   long long ymin, ymax;
};
NodeType nodes[MAX*4];
long long height[MAX];

struct Rectangle {
   long long x1, y1, x2, y2;
};

//Danh sách các hình chữ nhật
Rectangle rects[30000];

struct Vert {
   long long x;
   bool begin;
   int id;
};

//Các cạnh đứng
Vert verticals[MAX];
long long horizontals[MAX];

bool my_compare(const Vert& a, const Vert& b) {
   return a.x < b.x;
}

int create_interval(int n) {
   //Cạnh ngang <y, id-rect> chỉ số của hình chữ nhật
   //dùng để cập nhật từ toạ độ sang chỉ số khoảng
   static long long ys[MAX];

   for (int i = 0; i < n; i++) {
      verticals[2*i].x = rects[i].x1;
      verticals[2*i].begin = true;
      verticals[2*i].id = i;
```

```
      verticals[2*i+1].x = rects[i].x2;
      verticals[2*i+1].begin = false;
      verticals[2*i+1].id = i;

      ys[2*i] = rects[i].y1;
      ys[2*i+1] = rects[i].y2;
   }

   sort(verticals, verticals + 2*n, my_compare);
   sort(ys, ys + 2*n);

   //unique
   int k = 0; int y = -1;
   for (int i = 0; i < 2*n; i++) {
      if (y != ys[i]) {
         horizontals[k] = ys[i];
         y = ys[i];
         k++;
      }
   }
   return k-1;
}

void init_tree(int node, int left, int right) {
   nodes[node].value = 0; nodes[node].sum = 0;
   if (left == right) {
      nodes[node].ymin = horizontals[left-1];
      nodes[node].ymax = horizontals[left];
      return;
   }
   int mid = (left + right)/2;
   init_tree(2*node, left, mid);
   init_tree(2*node+1, mid+1, right);

   nodes[node].ymin = nodes[2*node].ymin;
   nodes[node].ymax = nodes[2*node+1].ymax;
}

void increase(int node, int left, int right, int u, int v) {
   if (u >= nodes[node].ymax || v <= nodes[node].ymin)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= nodes[node].ymin && v >= nodes[node].ymax) {
      nodes[node].value++;
      nodes[node].sum = nodes[node].ymax - nodes[node].ymin;
      return;
   }
   int mid = (left + right)/2;
   increase(2*node, left, mid, u, v);
   increase(2*node+1, mid+1, right, u, v);

   //tổng hợp tổng từ 2 con
   if (nodes[node].value == 0)
      nodes[node].sum = nodes[2*node].sum +
```

```
                  nodes[2*node+1].sum;
}

void decrease(int node, int left, int right, int u, int v) {
   if (u >= nodes[node].ymax || v <= nodes[node].ymin)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= nodes[node].ymin && v >= nodes[node].ymax) {
      nodes[node].value--; //value của 1 đoạn chỉ có thể >= 0
      if (nodes[node].value == 0) {
         if (left != right)
            nodes[node].sum = nodes[2*node].sum +
                  nodes[2*node+1].sum;
         else
            nodes[node].sum = 0;
      }
      return;
   }
   int mid = (left +right)/2;
   decrease(2*node, left, mid, u, v);
   decrease(2*node+1, mid+1, right, u, v);

   //tổng hợp tổng từ 2 con
   if (nodes[node].value == 0)
      nodes[node].sum = nodes[2*node].sum +
                  nodes[2*node+1].sum;
}

int main() {
   int nCase, n;
   scanf("%d", &nCase);
   for (int no = 1; no <= nCase; no++) {
      scanf("%d", &n);
      for (int i = 0; i < n; i++)
         scanf("%lld %lld %lld %lld",
 &rects[i].x1, &rects[i].y1,
              &rects[i].x2, &rects[i].y2);

      int k = create_interval(n);

      //cout << "k = " << k << endl;

      init_tree(1, 1, k);
      long long result = 0ll;
      for (int i = 0; i < 2*n - 1; i++) {
         int id = verticals[i].id;
         if (verticals[i].begin) {
            increase(1, 1, k, rects[id].y1, rects[id].y2);
         } else {
            decrease(1, 1, k, rects[id].y1, rects[id].y2);
         }
         result += nodes[1].sum*
                  (verticals[i+1].x - verticals[i].x);
      }
```

```
        printf("Case %d: %lld\n", no, result);
    }
    return 0;
}
```

Bạn cũng có thể áp dụng cách tương tự để giải bài **1204 - Weird Advertisement** (Light Oj).

**1204 - Weird Advertisement** (Light Oj)

**2DPlaneLand** is a land just like a huge **2D** plane. The range of **X** axis is **0** to $10^9$ and the range of **Y** axis is also **0** to $10^9$. People built houses only in integer co-ordinates and there is exactly one house in each integer co-ordinate.

Now **UseAndSmile** Soap Company is launching a new soap. That's why they want to advertise this product as much as possible. So, they selected **n** persons for this task. Each person will be given a rectangular region. He will advertise the product to all the houses that lie in his region. Each rectangular region is identified by **4** integers $x_1$, $y_1$, $x_2$ and $y_2$. That means this person will advertise in all the houses whose **x** co-ordinate is between $x_1$ and $x_2$ (inclusive) and **y** co-ordinate is between $y_1$ and $y_2$ (inclusive).

Now after a while they realized that some houses are being advertised by more than one person. So, they want to find the number of houses that are advertised by at least **k** persons. Since you are one of the best programmers in the city; they asked you to solve this problem.

**Input**

Input starts with an integer **T** $(\leq 13)$, denoting the number of test cases.

Each case starts with a line containing two integers **n** $(1 \leq n \leq 30000)$, **k** $(1 \leq k \leq 10)$. Each of the next **n** lines will contain four integers $x_1, y_1, x_2, y_2$ $(0 \leq x_1, y_1, x_2, y_2 \leq 10^9, x_1 < x_2, y_1 < y_2)$ denoting a rectangular region for a person.

**Output**

For each case, print the case number and the total number of houses that are advertised by at least k people.

| Sample Input | Output for Sample Input |
|---|---|
| 2<br>2 1<br>0 0 4 4<br>1 1 2 5<br>2 2<br>0 0 4 4<br>1 1 2 5 | Case 1: 27<br>Case 2: 8 |

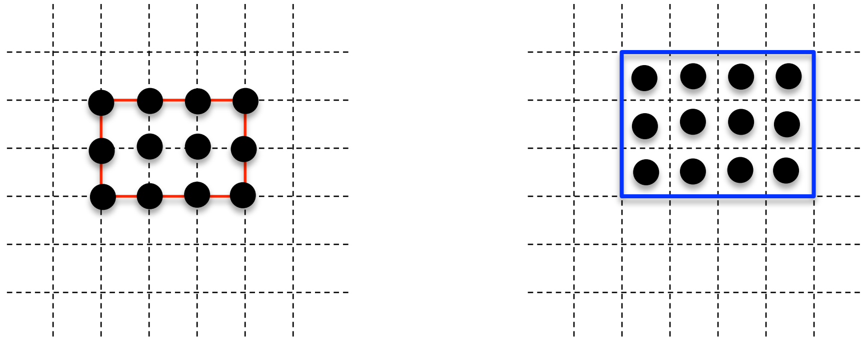
**Note**

Dataset is huge. Use faster i/o methods.

Bài này khác với bài Rectangle Union ở 2 điểm chính:

- Cách tính số nhà trong một vùng chữ nhật khác với diện tích của hình chữ nhật.
- Ta phải đếm số nhà được quảng cáo ít nhất k lần. k có thể lớn hơn 1.

Điểm khác biệt đầu tiên có thể giải quyết bằng cách di chuyển các căn nhà từ giao điểm đến tâm của các ô và nới rộng hình chữ nhật thêm 1 đơn vị cho cả chiều rộng và chiều cao.



Đối với điểm khác biệt thứ hai, khi một đoạn bị phủ bằng một hình chữ nhật nếu số lượng hình chữ nhật bị phủ còn nhỏ hơn K, ta không thể tính tổng số ô bị phủ ngay mà phải đi xuống tìm các ô bị phủ từ các nút con của nó. Quá trình đi xuống này sẽ dừng lại khi gặp 1 nút lá hoặc một đoạn bị phủ nhiều hơn k hình chữ nhật.

**Tăng một đoạn:** ta phải truyền thêm 1 tham số parent để cho biết tổng số hình chữ nhật đã phủ lên đoạn này (tính từ nút gốc)

```
void increase(int node, int left, int right,
               int u, int v, int parent) {
  if (u >= nodes[node].ymax || v <= nodes[node].ymin)
    return; //[u,v] nằm ngoài đoạn quản lý của node
  if (u <= nodes[node].ymin && v >= nodes[node].ymax) {
    nodes[node].value++;
    //Đếm số nhà và kết thúc
    count(node, left, right, parent);
    return;
  }
  int mid = (left + right)/2;
  increase(2*node, left, mid, u, v,
                      parent + nodes[node].value);
  increase(2*node+1, mid+1, right, u, v,
                      parent + nodes[node].value);

  //tổng hợp tổng từ 2 con
  if (parent + nodes[node].value < k)
    nodes[node].sum = nodes[2*node].sum +
                              nodes[2*node+1].sum;
}
```

**Giảm một đoạn:**

```
void decrease(int node, int left, int right,
                         int u, int v, int parent) {
   if (u >= nodes[node].ymax || v <= nodes[node].ymin)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= nodes[node].ymin && v >= nodes[node].ymax) {
      nodes[node].value--;
      //Đếm số nhà và kết thúc
      count(node, left, right, parent);
      return;
   }
   int mid = (left + right)/2;
   decrease(2*node, left, mid, u, v,
                         parent + nodes[node].value);
   decrease(2*node+1, mid+1, right, u, v,
                         parent + nodes[node].value);

   //tổng hợp tổng từ 2 con
   if (parent + nodes[node].value < k)
      nodes[node].sum = nodes[2*node].sum +
                              nodes[2*node+1].sum;
}
```

**Đếm số nhà của một đoạn và cập nhật vào biến tổng hợp sum:**

```
void count(int node, int left, int right, int parent) {
   if (parent + nodes[node].value >= k) {
      nodes[node].sum = nodes[node].ymax - nodes[node].ymin;
      return;
   }
   if (left == right) {
      nodes[node].sum = 0;
      return;
   }
   int mid = (left + right)/2;
   //tổng hợp tổng từ 2 con
   count(2*node, left, mid, parent + nodes[node].value);
   count(2*node+1, mid+1, right,
                    parent + nodes[node].value);
   nodes[node].sum = nodes[2*node].sum +
                    nodes[2*node+1].sum;
}
```

Bài kế tiếp theo đây sử dụng interval tree để tăng tốc cho kỹ thuật quy hoạch động.

**1415 - Save the Trees (Light Oj)**

1 hàng cây được đánh số từ 1 đến n ($n \le 2*10^5$). mỗi cây thuộc về 1 loại type[i] và có 1 chiều cao height[i]. Để tính giá trị của toàn bộ hàng cây trước hết ta gom các cây cạnh nhau để thành các nhóm. Loại của các cây trong cùng nhóm phải khác nhau. Giá trị của một nhóm bằng chiều cao của cây cao nhất. Giá trị của hàng cây bằng tổng giá trị của các nhóm. Với

mỗi cách gom nhóm khác nhau, ta có tổng giá trị của hàng cây cũng khác nhau. Nhiệm vụ của bạn là tìm giá thấp nhất.

Ví dụ: hàng 5 cây như bên dưới có giá thấp nhất là 26.

```
5
3 11
2 13
1 12
2 9
3 13
```

Ví dụ: hàng 5 cây như bên dưới có giá thấp nhất là 26.

Trước hết, ta thử sử dụng kỹ thuật quy hoạch động để phân tích bài này. Ta chia bài toán thành n bước, mỗi bước ta xét 1 cây. Nếu gọi F[j] là giá thấp nhất của các cây từ 1 đến j, ở bước i, ta xem xét cây thứ i. Ta có thể gom cây i vào một trong các nhóm (j, j+1, …, i). Gọi Left[i] là cây xa nhất về phía trái mà ta có thể gom nó vào cùng nhóm với cây i, ta có:

**F[i] = min{F[j-1] + max-height(j, i)}**

với j chạy từ Left[i] đến i và max-height(j, i) là chiều cao của cây cao nhất từ j đến i.

Giải thuật quy hoạch động cho bài này khá đơn giản:

```
for (int i = 1; i <= n; i++) {
   F[i] = inf; int h = 0;
   for (int j = left[i]; j <= i; j++) {
      h = max(h, height[j]);
      F[i] = min(F[i], F[j-1] + h);
   }
}
```

Để tính Left[i], ta có thể sử dụng kỹ thuật quy hoạch động.

Left[i] là vị trí gần i nhất trong 2 vị trí: Left[i-1] và vị trí của cây sau cùng có cùng loại với cây i hay:

**Left[i] = max(Left[i-1] + loc[type[i]] +1);**

trong đó loc[type[i]] là vị trí của cây sau cùng có cùng loại với i.

```
void compute_left(int n) {
   static int loc[MAX];
   for (int i = 0; i <= n; i++)
      loc[i] = 0;
   Left[0] = 1;
   for (int i = 1; i <= n; i++) {
      Left[i] = max(Left[i-1], loc[type[i]]);
      loc[type[i]] = i;
   }
}
```

Tuy nhiên ta không thể dùng kỹ thuật quy hoạch động để giải bài này vì n quá lớn ($n \le 2*10^5$) và có đến 2 vòng lặp lồng nhau. Sử dụng interval tree đã có thể làm giảm thời gian tính toán của vòng lặp j.

Mô hình hoá bài toán này về interval tương đối phức tạp. Ta thử phân tích nhé. Công thức tính F[i] là:

**F[i] = min{F[j-1] + max-height(j, i)}**

Đặt f[j] = F[j-1] và h[j] = max-height(j, i), ta có:

F[i] = min {f[j] + h[j]} với mọi left[i] ≤ j ≤ i.

Ta sử dụng 1 interval tree để quản lý 1 đoạn từ 1 đến n. Mỗi phần tử j lưu f[j] + h[j]. Để tìm F[i] ta truy vấn khoảng từ left[i] đến i để lấy giá thấp nhất. Mỗi khi thêm một cây mới, ta đưa i vào interval tree và cập nhật h[j] của tất cả các phần tử có h[j] height[i]. Đoạn các phần tử cần cập nhật h[j] là từ before[i] đến i với before[i] là **vị trí ngay sau cây** có chiều cao cao hơn i.

Nút **node** trong interval tree quản lý các cây từ **left** đến **right**, thông tin riêng của nút h[j]. Thông tin tổng hợp bao gồm **min_fh** lưu giá trị nhỏ nhất của f[j] + h[j] với tất cả các j trong đoạn node quản lý và **min_f** lưu giá trị nhỏ nhất của f[j].

**Khai báo:**

```
#define MAX 200001
struct NodeType {
   int h;
   long min_fh;
   long min_f;
};
NodeType nodes[MAX*4];
```

**Khởi tạo:**

```
void init_tree(int node, int left, int right) {
   nodes[node].h = 0; nodes[node].min_fh = 0;
   min_f = 0;
   if (left == right) {
      return;
   }
   int mid = (left + right)/2;
   init_tree(2*node, left, mid);
   init_tree(2*node+1, mid+1, right);
}
```

**Lan truyền**

```
void propage(int node) {
   if (nodes[node].h > 0) {
      int mid = (left+right)/2;
      int h = nodes[node].h;
      nodes[2*node].h = h;
      nodes[2*node].min_fh = nodes[2*node].min_f + h;

      nodes[2*node+!].h = h;
      nodes[2*node+1].min_fh =
                              nodes[2*node+1].min_f + h;
      nodes[node].h= 0; //đã lan truyền xong
   }
}
```

**Tổng hợp**

```
void synthesize(int node) {
   nodes[node].min_fh = min(nodes[2*node]. min_fh,
                            nodes[2*node+1]. min_fh);
   nodes[node].min_f = min(nodes[2*node].min_f,
                            nodes[2*node+1].min_f);
}
```

**Thêm cây i vào interval tree**

```
void insert(int node, int left, int right, int i) {
   if (u > right || v < left)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (left == right) {
      nodes[node].h = height[i];
      nodes[node].min_fh = height[i];
      nodes[node].min_f = F[i-1];
      return;
   }
   propagate(node);
   int mid = (left+right)/2;
   if (i <= mid)
      insert(2*node, left, mid, i);
   else
      insert(2*node+1, mid+1, right, i);
   //tổng hợp tổng từ 2 con
   synthesize(node);
}
```

**Cập nhật h[j] của đoạn [u, v]**

Do ta luôn cập nhật các đoạn có h[j] nhỏ hơn giá trị cập nhật h nên ta chỉ cần gán h[j] = h cho cả 1 đoạn.

```
void update(int node, int left, int right,
            int u, int v, int h) {
   if (u > right || v < left)
      return; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= left && v >= right) {
      nodes[node].h = h;
      nodes[node].min_fh = nodes[node].min_f + h;
      return;
   }
   propagate(node);
   int mid = (left+right)/2;
   update(2*node, left, mid, u, v, h);
   update(2*node+1, mid+1, right, u, v, h);
   //tổng hợp tổng từ 2 con
   synthesize(node);
}
```

**Truy vấn tổng 1 đoạn [u, v]:**

```
long long query(int node, int left, int right,
            int u, int v) {
   if (u > right || v < left)
      return 0; //[u,v] nằm ngoài đoạn quản lý của node
   if (u <= left && v >= right)
      return nodes[node].min_fh;
   propagate(node);
   int mid = (left+right)/2;
   return query(2*node, left, mid, u, v) +
                  query(2*node+1, mid+1, right, u, v);
}
```

**Tính toán before[i]**

Để tính before[i] ta sử dụng 1 stack lưu vị trí của các cây. Với mỗi cây i, ta lần lượt loại bỏ các phần tử trên đỉnh stack có chiều cao thấp hơn chiều cao của i. before[i] = vị trí của cây trên đỉnh stack + 1.

```
void compute_before (int n) {
   static int stack[MAX];
   int top = 0;
   stack[top++] = 0;
   for (int i = 1; i <= n; i++) {
      while (top > 0 && height[top] <= height[i])
         top--;
      before[i] = stack[top] + 1;
      stack[++top] = i;
}
```

Hàm main()

```
int main() {
   int nCase, n;
   scanf("%d", &nCase);
   for (int no = 1; no <= nCase; no++) {
      scanf("%d", &n);
      for (int i = 1; i <= n; i++)
         scanf("%d %d", &type[i], &height[i]);
      init_tree(1, 1, n);
      compute_left(n);
      compute_before(n);
      F[0] = 0;
      for (int i = 1; i <= n; i++) {
         insert(1, 1, n, i);
         update(1, 1, n, Before[i], i, height[i]);
         F[i] = query(1, 1, n, Left[i], i);
      }
      printf("Case %d: %lld\n", no, F[n]);
   }
   return 0;
}
```

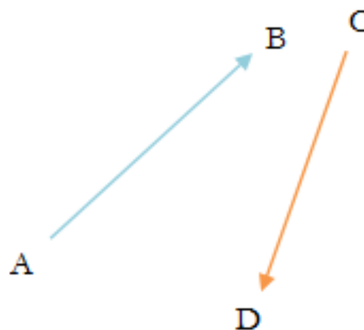
Nào ta cùng submit.

# 5 Giải tích và hình học

Có rất nhiều bài toán thay vì giải bằng cách lặp và tìm kiếm, ta có thể giải được nó bằng cách áp dụng các kết quả của toán.

## 5.1 Bài toán khoảng cách ngắn nhất

Xét bài toán 1146 - Closest Distance (Light Oj) như sau: một người đi từ A đến B và người kia đi từ C đến D. Tốc độ di chuyển của hai người tỉ lệ thuận với chiều dài của đoạn đường của họ. Tính khoảng cách ngắn nhất giữa 2 người trên đường đi.




Bài này được xếp trong nhóm tìm kiếm tam phân (Ternary Search), tuy nhiên ta có thể giải bài này bằng cách tìm cực trị của một hàm 1 biến. Để tìm cực trị ta lấy đạo hàm và giải phương trình đạo hàm = 0. So sánh giá trị của hàm tại 2 điểm cực biên và giá trị tại điểm có đạo hàm = 0.

Gọi xA, yA, xB, yB, xC, yC, xD, yD là toạ độ của các điểm A, B, C và D. Ta có phương trình chuyển động của người thứ nhất theo thời gian t như sau:

$$x = (xB - xA)t + xA$$
$$y = (yB - yA)t + yA$$

Tương tự phương trình chuyển động của người thứ 2 là:

$$x = (xD - xC)t + xC$$
$$y = (yD - yC)t + yC$$

Bình phương khoảng cách giữa hai người theo t:

$$d(t) = (xB - xA)t + xA - (xD - xC)t - xC)^2$$
$$+ (yB - yA)t + yA - (yD - yC)t - yC)^2$$

Để tìm khoảng cách nhỏ nhất ta đạo hàm d(t) theo t và giải phương trình d'(t) = 0 ta tìm được t*. Sau đó so sánh 3 khoảng cách (A, C), (B, D) và d(t*) để tìm khoảng cách lớn nhất.

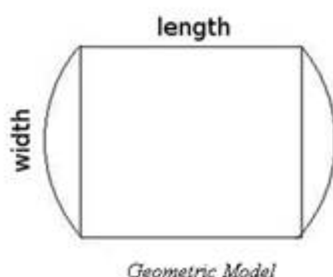
**Code chương trình mẫu:**

```
#include <iostream>
#include <string.h>
#include <math.h>
using namespace std;
double xA, xB, xC, xD, yA, yB, yC, yD;
double pow2(double x) {
   return x*x;
}
double process() {
   double deltaX = xB - xA - xD + xC;
   double deltaY = yB - yA - yD + yC;
   double dx = xA - xC;
   double dy = yA - yC;

   double dist1 = sqrt(pow2(xA - xC) + pow2(yA - yC));
   double dist2 = sqrt(pow2(xB - xD) + pow2(yB - yD));
   double dist0;

   if (deltaX*deltaX + deltaY*deltaY == 0)
      dist0 = -1;
   else {
      double t = -(deltaX * dx + deltaY * dy)/
                        (deltaX*deltaX + deltaY*deltaY);

      if (t < 0 || t > 1)
         dist0 = -1;
      else
         dist0 = sqrt(pow2(deltaX*t + dx) +
                     pow2(deltaY*t + dy));
   }
   if (dist0 < 0)
      return std::min(dist1, dist2);
   return std::min(dist1, std::min(dist2, dist0));
}
int main() {
   int nCase, n;
   cin >> nCase;
   cout.precision(12);
   for (int no = 1; no <= nCase; no++) {
      cin >> xA >> yA >> xB >> yB >> xC >> yC >> xD >> yD;
      cout << "Case " << no << ": " << process() << endl;
   }
   return 0;
}
```

## 5.2 Bài tương tự

- 1240 - Point Segment Distance (3D)

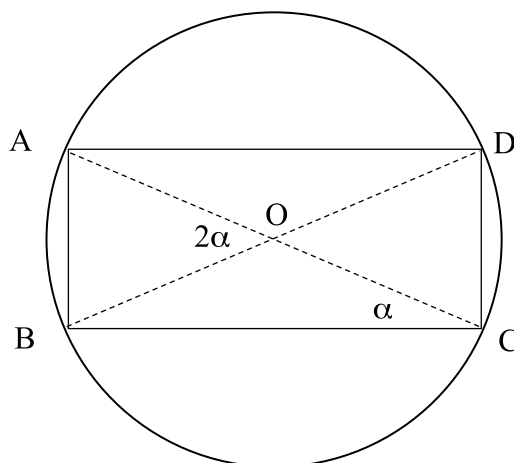
## 5.3 Olympics (1056 Light Oj)



Sân vận động của một đợt thế vận hội có hình dạng là hình chữ và hai cung tròn thuộc cùng 1 đường tròn (xem hình). Tính chiều dài (length) và chiều rộng của hình chữ nhật trên nếu biết *tỉ lệ giữa chiều dài và chiều rộng* và *chu vi của sân vận động là 400m*.
Ví dụ:

| Sample Input | Output for Sample Input |
|---|---|
| 2<br>3 : 2<br>5 : 4 | Case 1: 117.1858168 78.12387792<br>Case 2: 107.29095604 85.8327648 |

Bài này được xếp nằm trong nhóm tìm kiếm nhị phân. Tuy nhiên ta có thể bài này một cách nhanh chóng bằng cách áp dụng các kiến thức về hình học và lượng giác.




Nếu gọi góc ACB = $\alpha$, áp dụng tổng của 3 góc trong tam giác bằng $180^o$, ta dễ dàng suy ra góc AOB = $2\alpha$.
Nửa chu vi của sân vận động = AD + OA* $2\alpha$ = AD + AC*$\alpha$.
Mặt khác AC = BD/cos($\alpha$) = AD/cos($\alpha$). Thay vào trên ta đượ

$$AD + AD*\alpha/\cos(\alpha) = 400/2 = 200$$

Suy ra:

$$\text{Chiều dài} = AD = 200(1 + \alpha/\cos(\alpha))$$
$$\text{Chiều rộng} = \text{chiều dài/tỉ lệ}$$

Để tìm $\alpha$, ta xét:

$$\tan(\alpha) = \text{chiều rộng/chiều dài}$$
$$\Rightarrow \alpha = \text{atan(chiều rộng/chiều dài)}.$$

Đến đây thì xem như bài toán đã giải xong.
Nào các bạn cùng submit nhé!
Code mẫu:

```
//1056 - Olympics
#include <iostream>
#include <math.h>
using namespace std;

void solve(int l, int w) {
   double alpha = atan2(w, l);
   double length = 200.0/(1.0 + alpha/cos(alpha));
   double width = length*w/l;
   cout << length << ' ' << width << endl;
}

int main() {
   int nCase, w, l;
   char c;
   cin >> nCase;
   cout.precision(9);
   for (int no = 1; no <= nCase; no++) {
      cin >> l;
      cin >> c; //bỏ qua dấu :
      cin >> w;

      cout << "Case " << no << ": ";
      solve(l, w);
   }
   return 0;
}
```

# 6 Giải thuật trên đồ thị

## 6.1 Giải thuật Dijkstra tìm đường đi ngắn nhất

Cho đồ thị (có hướng hoặc vô hướng đều được), tìm đường đi ngắn nhất từ s đến tất cả các đỉnh còn lại.

Ý tưởng chính của giải thuật là bắt đầu từ đỉnh s (đánh dấu s được xét rồi), ta tìm đỉnh có khoảng cách ngắn nhất từ s đến nó, đặt s là đỉnh mới này. Xét các đỉnh có cung nối trực tiếp với s và cập nhật lại khoảng cách mới nếu đường đi cũ dài hơn. Đánh dấu s đã được xét. Tiếp tục quá trình này cho đến khi tất cả các đỉnh đều được xét.

**Giải thuật Dijstra**

```
#define MAX 501
#define Inf 50000
int M[MAX][MAX];
int check[MAX];
int d[MAX];

void Dijkstra (int n, int t) {
   //1. Khởi tạo chưa đỉnh nào được xét, d[i] = vô cùng
   memset(check, 0, sizeof(check));
   for (int i = 0; i < n; i++)
      d[i] = Inf;
   int s = t;
   d[s] = 0;
   for (int j = 0; j < n; j++)
      if (M[s][j] < Inf)
         d[j] = M[s][j];
   check[s] = 1;

   for (int i = 1; i < n; i++) {
      //2. tìm phần tử nhỏ nhất
      int MIN = Inf;
      s = -1;
      for (int k = 0; k < n; k++)
         if (!check[k] && d[k] < MIN) {
            MIN = d[k];
            s = k;
         }
      if (s == -1)
         break;
      //3. cập nhật d[j] của các con của s
      for (int j = 0; j < n; j++) {
         int newcost = d[s] + M[s][j];
         if (!check[j] && M[s][j] < Inf && d[j] > newcost)
            d[j] = newcost;
      }
      check[s] = 1;
   }
}
```

**1019 - Brush (V):** tìm đường đi ngắn nhất từ đỉnh 1 đến đỉnh n trong đồ thị

Áp dụng trực tiếp giải thuật Dijsktra để giải bài này.

**Biến thể của Dijstra:** Xét bài 1002 - Country Roads

I am going to my home. There are many cities and many bi-directional roads between them. The cities are numbered from **0** to **n-1** and each road has a cost. There are **m** roads. You are given the number of my city **t** where I belong. Now from each city you have to find the minimum cost to go to my city. The cost is defined by the cost of the maximum road you have used to go to my city.

For example, in the above picture, if we want to go from 0 to 4, then we can choose

1) 0 - 1 - 4 which costs 8, as 8 (1 - 4) is the maximum road we used
2) 0 - 2 - 4 which costs 9, as 9 (0 - 2) is the maximum road we used
3) 0 - 3 - 4 which costs 7, as 7 (3 - 4) is the maximum road we used

So, our result is 7, as we can use 0 - 3 - 4.

### Input

Input starts with an integer **T (≤ 20)**, denoting the number of test cases.

Each case starts with a blank line and two integers **n (1 ≤ n ≤ 500)** and **m (0 ≤ m ≤ 16000)**. The next **m** lines, each will contain three integers **u, v, w (0 ≤ u, v < n, u ≠ v, 1 ≤ w ≤ 20000)** indicating that there is a road between **u** and **v** with cost **w**. Then there will be a single integer **t (0 ≤ t < n)**. There can be multiple roads between two cities.

### Output

For each case, print the case number first. Then for all the cities (from **0** to **n-1**) you have to print the cost. If there is no such path, print **'Impossible'**.

| Sample Input | Output for Sample Input |
|---|---|
| 2<br><br>5 6<br>0 1 5<br>0 1 4<br>2 1 3<br>3 0 7<br>3 4 6<br>3 1 8<br>1<br><br>5 4<br>0 1 5<br>0 1 4<br>2 1 3<br>3 4 7<br>1 | Case 1:<br>4<br>0<br>3<br>7<br>7<br>Case 2:<br>4<br>0<br>3<br>Impossible<br>Impossible |

Tìm đường đi ngắn từ các thành phố đến thành phố t. Do các đường nối là đường hai chiều nên bài toán tương ứng với tìm đường đi ngắn nhất từ t đến các thành phố khác. Chi phí của đường đi được tính bằng độ dài cung dài nhất trên đường đi.

Tìm đường có chi phí nhỏ nhất với chi phí của đường đi được tính bằng chi phí của cung lớn nhất trên đường đi.

Với bài toán này ta chỉ cần đều chỉnh 1 chút ở phần cập nhật chi phí đường đi. Nếu d[i] là chi phí đường đi từ s đến i, và c[i][j] là chi phí của cung (i, j) thì ta sẽ cập nhật chi phí của đường đi từ s đến j, d[j] = max(d[i], c[i][j]) nếu chi phí mới này tốt hơn chi phí cũ d[j].

Cần chú ý giữa hai đỉnh có thể có nhiều cung. Ta chỉ cần giữ lại cung có chi phí nhỏ nhất.

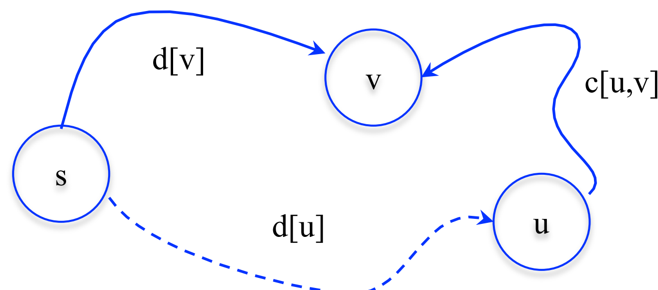
Nào, các bạn thử submit nhé !

```
//1002 - Country Roads
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <algorithm>
#include <iostream>
#include <map>
using namespace std;

#define MAX 501
#define Inf 50000
int M[MAX][MAX], check[MAX], d[MAX];

void Dijkstra(int n, int t) {
   memset(check, 0, sizeof(check));
   for (int i = 0; i < n; i++)
      d[i] = Inf;
   int s = t;
   d[s] = 0;
   for (int j = 0; j < n; j++)
      if (M[s][j] < Inf)
         d[j] = M[s][j];
   check[s] = 1;

   for (int i = 1; i < n; i++) {
      //tìm phần tử nhỏ nhất
      int MIN = Inf;
      s = -1;
      for (int k = 0; k < n; k++)
         if (!check[k] && d[k] < MIN) {
            MIN = d[k];
            s = k;
         }
      if (s == -1)
         break;
      for (int j = 0; j < n; j++) {
         int newcost = max(d[s], M[s][j]);
         if (!check[j] && M[s][j] < Inf && d[j] > newcost)
            d[j] = newcost;
      }
```

```
      check[s] = 1;
   }
   for (int i = 0; i < n; i++)
      if (d[i] < Inf)
         printf("%d\n", d[i]);
      else
         printf("Impossible\n");

}

int main() {
   int nCase, n, m, t, u, v, c;
   scanf("%d", &nCase);
   for (int no = 1; no <= nCase; no++) {
      scanf("%d %d", &n, &m);
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            M[i][j] = Inf;

      for (int i = 0; i < m; i++) {
         scanf("%d %d %d", &u, &v, &c);
         if (M[u][v] > c)
            M[u][v] = M[v][u] = c;

      }
      scanf("%d", &t);

      printf("Case %d:\n", no);
      Dijkstra(n, t);
   }
   return 0;
}
```

Chú ý: giải thuật Dijsktra chạy được là nhờ vào một giả thiết sau đây:

- Trong quá trình thực hiện giải thuật, chiều dài đường đi ngắn nhất từ đỉnh bắt đầu đến một đỉnh bất kỳ sẽ giảm dần. Tại mỗi thời điểm, nếu đường đi từ đỉnh bắt đầu đi đến 1 đỉnh v là ngắn nhất so với các đỉnh còn lại thì đường đi này là **đường đi ngắn nhất**. Giả thiết này chỉ đúng khi đồ thị không chứa cung có trọng số âm. Nếu trọng số các cung là dương thì không thể nào ta có một đỉnh nào đó có d[u] > v mà sao đó đi từ u về v lại nhỏ hơn d[v] được.



## 6.2 Giải thuật Kruskal tìm cây khung có trọng số nhỏ nhất

Cho đồ thị vô hướng G, cây T được gọi cây khung của đồ thị t nếu:

- T gồm các đỉnh của G
- T liên thông
- T không chứa chu trình

Cây khung của một đồ thị n đỉnh có đúng n-1 cạnh.

Trọng số của một cây khung bằng tổng trọng số các cung trên cây.

Cây khung nhỏ nhất là cây có trọng số nhỏ nhất.

Có 2 giải thuật tìm kiếm cây khung nhỏ nhất của một đồ thị vô hướng: giải thuật Prim (thao tác theo đỉnh) và giải thuật Kruskal thao tác theo cạnh.

Ý tưởng của giải thuật như sau:

- Sắp xếp các cạnh theo thứ tự tăng dần
- Khởi tạo cây T không có cạnh nào cả
- Lần lượt xét từng cạnh (đã sắp xếp) xem có thể đưa vào cây T không. Ta sẽ đưa một cạnh c vào cây T nếu sau khi thêm nó T không chứa chu trình.
- Giải thuật kết thúc khi: T có đủ n-1 cạnh (n là số đỉnh) hoặc tất cả các cạnh đã xem xét.
- Nếu sau khi kết thúc T không có đủ n-1 cạnh có nghĩa là đồ thị ban đầu không liên thông.

Mấu chốt của giải thuật Krusskal nằm ở chỗ xét xem T có chu trình nếu thêm cạnh c vào không. Có một cạnh khá đơn giản nhưng hiệu quả để giải quyết vấn đề này như sau:

Ta xem T như một rừng (gồm nhiều cây). Khi khởi tạo T gồm n cây, mỗi cây chỉ có 1 nút.

Khi thêm một cạnh c vào, nếu cạnh này có 2 đỉnh/nút nằm ở 2 cây khác nhau thì chắc chắn không tạo nên chu trình, ngược lại thì thêm c vào T sẽ chứa chu trình.

Để cài đặt ta sử dụng một mảng p lưu cha của một đỉnh. Bằng cách này ta sẽ lưu được các cây trong rừng T.

Khởi tạo, p[i] = i (hoặc = -1 cũng được, để cho biết i là nút gốc của cây).

Khi thêm cạnh c = (u, v) vào cây ta sẽ tìm gốc của u và gốc của v. Nếu u và v có chung gốc, có nghĩa là nó cùng chung 1 cây, ngược lại nó nằm ở 2 cây khác nhau.

Giải thuật tìm gốc của 1 nút v khá đơn giản, ta lần ngược lên cha của nó (dựa vào mảng p) được p[v], p[p[v]], … cho đến khi p[i] = i thì i là gốc. Sau khi đã tìm được gốc của v là i. Để tiết kiệm cho các lần tìm gốc sau ta gán p[v] = i luôn.

**Giải thuật Kruskal:**

```
int findRoot(int v) {
   int node = v;
   //Trong khi node chưa phải là gốc, lần lên cha của nó
   while (p[node] != node)
      node = p[node];
   //Kết thúc vòng lặp node là nút gốc
   //Gán p[v] = node luôn, để lần sau tìm gốc của v cho
   //nhanh, chỉ cần lần lên một bước.
   p[v] = node;
   return p[v];
}
```

```
int Kruskal(int n, int m) {
  //1. Khởi tạo n nút gốc của n cây trong rừng T
  for (int i = 1; i <= n; i++)
    p[i] = i;

  //2. Sắp xếp các cạnh theo trọng số tăng dần
  sort(edges, edges + m, compare);

  int sum = 0, cnt = 0;
  //3. Duyệt qua từng cạnh
  for (int i = 0; i < m && nb_edges < n-1; i++) {
    int rx = findRoot(edges[i].x);
    int ry = findRoot(edges[i].y);

    //3.1 nếu chung gốc => bỏ qua
    if (rx == ry)
      continue;
    //3.2 nếu khác gốc, gom 2 cây thành 1
    p[ry] = rx;
    sum += edges[i].c;
    //3.2 tăng số cạnh trong T
    nb_edges++;
  }
  //4. trả về trọng số của cây
  return sum;
}
```

**1040 - Donation: quyên góp cáp Ethernet**

Bạn có n căn phòng, đánh số từ 1 đến n. Giữa các căn phòng có thể được nối với nhau bằng 1 sợi cáp có chiều dài c. Việc kết nối giữa các căn phòng được mô tả bằng 1 ma trận như bên dưới:

```
0 10 10 0
0 0  1  1
0 0  0  2
0 0  0  5
```

Phần tử (i, j) cho biết chiều dài của dây cáp nối giữa phòng i và phòng j. Bạn chỉ muốn giữ lại các sợi cáp cần thiết để hai phòng bất kỳ luôn được kết nối (trực tiếp hoặc thông qua một số phòng khác). Các sợi cáp thừa bạn dùng để quyên góp. Hãy tính tổng chiều dài dài nhất của các đoạn cáp mà bạn có thể quyên góp. Nếu ban đầu có 2 phòng không được kết nối với nhau trả về -1.

Trong ví dụ trên, bạn có thể quyên góp nhiều nhất là: 10 + 2 + 5 = 17. Các phòng vẫn còn được kết nối: (1, 2) (2, 3) (3, 4).

Để giải bài này ta có thể áp dụng trực tiếp giải thuật Kruskal.

- Sắp xếp các cạnh theo thứ tự tăng dần.
- Lần lượt xét từng cạnh, với mỗi cạnh nếu thêm được vào rừng T ta tăng số cung sử dụng lên 1. Đánh dấu nó đã được sử dụng. Giải thuật dừng lại khi đã xét hết các cạnh hoặc số cạnh sử dụng = n-1.

- Nếu số cạnh sử dụng < n-1 trả về -1, ngược lại ta tính tổng chiều dài các đoạn cáp không sử dụng và trả về tổng này.

Bạn thử code và submit thử đi nhé!

**1059 - Air Ports:** xây các sân bay và các con đường nối giữa các thành phố.

Cho **n** thành phố, ta cần phải xây dựng 1 số sân bay ở tại 1 số thành phố và xây nhiều nhất là **m** con đường nối từ thành phố này đến thành phố khác. Mục tiêu là từ bất cứ thành phố nào hoặc là nó có sân bay hoặc là có đường để ta đi đến 1 thành phố khác có sân bay. Xây dựng con đường nối 2 thành phố x, y tốn 1 chi phí là c. Xây 1 sân bay tốn 1 chi phí là A. Hãy tìm cách xây các sân bay và con đường sao cho tổng chi phí thấp nhất. Nếu có 2 phương án có chi phí bằng nhau ta chọn phương án có nhiều sân bay nhất. Chú ý: ta chỉ có thể xây nhiều nhất **m** con đường.

Đây là một biến thể của bài toán cây khung nhỏ nhất. Ta nhận xét:

- Nếu có đường đi từ thành phố x đến thành phố y (trực tiếp hay gian tiếp thông qua các thành phố khác) thì ta không cần phải xây thêm đường x đến y. Và ta cũng chỉ cần 1 sân bay là đủ cho x và y.
- Nếu x đã có sân bay và y là một thành phố thì ta có 2 giải pháp: hoặc là xây thêm 1 sân bay tại y hay xây thêm 1 con đường đến 1 thành phố nào đó liên thông với x. Ta sẽ chọn giải pháp có chi phí thấp nhất.

Với nhận xét này ta có giải thuật giải bài toán nay như sau:

- Khởi tạo, rừng T có n gốc, số sân bay cần phải xây = n
- Sắp xếp các con đường theo thứ tự tăng dần theo chi phí
- Lần lượt xét các con đường, nếu chi phí xây dựng nó lớn hơn chi phí xây bay ta dừng giải thuật. Vì xây con đường đắc hơn xây sân bay. Các con đường sau nó sẽ có chi phí lớn hơn nó (các con đường đã sắp xếp).
- Nếu chi phí của con đường nhỏ hơn chi phí xây sân bay, ta sẽ thêm nó vào rừng T nếu hai đầu mút của nó nằm ở hai cây khác nhau, giảm số sân bay cần phải xây đi 1 (gom 2 cây khác nhau lại thành 1 cây thì chỉ cần 1 sân bay). Ngược lại ta bỏ qua.

Khi giải thuật kết thúc, ta có tổng chi phí của các con đường và số sân bay cần phải xây. In kết quả này ra.

Nào, các bạn thử submit nhé !

```
//1059 - Air Ports
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <algorithm>
#include <iostream>
#include <map>
using namespace std;

#define MAX 100001

struct Edge {
  int x, y, c;
};

Edge edges[MAX];
int p[10001];
```

```
bool compare(const Edge& a, const Edge& b) {
   return a.c < b.c;
}

int findRoot(int v) {
   int node = v;
   while (p[node] != node)
      node = p[node];
   p[v] = node;
   return p[v];
}

void Kruskal(int n, int m, int a) {
   for (int i = 1; i <= n; i++)
      p[i] = i;
   int sum = 0, nb_tree = n;
   sort(edges, edges + m, compare);
   for (int i = 0; i < m && edges[i].c < a; i++) {
      int px = findRoot(edges[i].x);
      int py = findRoot(edges[i].y);
      if (px == py)
         continue;

      p[py] = px;
      sum += edges[i].c;
      nb_tree--;
   }

   printf("%d %d\n", sum + a*nb_tree, nb_tree);

}

int main() {
   int nCase, n, m, a, u, v, c;
   scanf("%d", &nCase);
   for (int no = 1; no <= nCase; no++) {
      scanf("%d %d %d", &n, &m, &a);
      for (int i = 0; i < m; i++) {
         scanf("%d %d %d", &edges[i].x,
                              &edges[i].y, &edges[i].c);
      }

      printf("Case %d: ", no);
      Kruskal(n, m, a);
   }
   return 0;
}
```

**Tìm cây khung có trọng số lớn nhất:**

Để tìm cây khung có trọng số lớn nhất, ta chỉ cần đổi thứ tự sắp xếp từ tăng dần thành giảm dần. Bạn có thể áp dụng để giải bài **1029 - Civil and Evil Engineer**.

## 7 Phân đôi và tìm kiếm nhị phân (Bisection and Binary search)

Đây là một kỹ thuật được dùng để dung hoà giữa bộ nhớ và thời gian tìm kiếm của 1 phương pháp tìm kiếm: tìm kiếm tuần tự và tìm kiếm nhị phân.

Xét bài toán đơn giản như sau: cho tập N gồm n số nguyên, và 1 số nguyên k hãy tìm xem k có trong N không ?

- Nếu dùng tìm kiếm tuần tự ta sẽ lần lượt xét từng phần tử xem nó có bằng k không. Giải pháp này tốn trung bình khoảng n/2 lần lặp để có kết quả.
- Nếu sử dụng tìm kiếm nhị phân, trước hết ta sắp xếp các số này theo thứ tự tăng/giảm dần, sau đó áp dụng chiến lược tìm kiếm nhị phân. Nếu không kể thời gian sắp xếp dãy số, ta tốn log(n) lần lặp để có kết quả.

Nhận xét:

- Rõ ràng, tìm kiếm nhị phân tốt hơn. Tuy nhiên cả hai giải pháp đều phải tốn bộ nhớ lưu trữ là n ô nhớ để lưu n phần tử. Đối với một số bài toán có n lớn, cả hai phương pháp đều không thể áp dụng được.

Tuy nhiên, nếu **n** số nguyên của N có thể được tạo ra bằng cách tổ hợp (cộng 2 phần tử) từ 2 tập số nguyên L và R có kích thước $n_1$, $n_2$ ($n_1*n_2 = n$, trong nhiều trường hợp ta có thể chọn $n_1 = n_2 = \sqrt{n}$), ta sẽ có một giải pháp để tìm k trong n số một cách hiệu quả:

- Giữ nguyên tập L và sắp xếp tập R
- Để tìm xem k có trong n số nguyên này không, ta lần lượt xét từng phần tử L[i], nếu k >= L[i], thì ta tìm xem phần còn lại k – L[i] có trong tập R không. Nếu có thì k có mặt trong n số nguyên. Nếu sau khi đã duyệt qua tất cả các phần tử của L mà vẫn không tìm thấy thì k không có trong n số nguyên.

Bằng cách này ta chỉ cần tốn n1 + n2 ô nhớ (nếu n1= n2, ta chỉ tốn $2\sqrt{n}$ ô nhớ). Thời gian tìm kiếm sẽ bằng $n_1*\log(n_2)$.

Như thế ý tưởng chính của kỹ thuật này có thể tóm tắt thông qua 3 bước sau đây:

(1) Tạo 2 tập L, R sao cho việc tổ hợp L và R sẽ tạo ra n phần tử của tập N
(2) Sắp xếp tập R
(3) Lần lượt xét từng phần tử của L, với mỗi phần tử L[i], thực hiện tìm kiếm f(k, L[i]) trong tập R với f(k, L[i]) là một hàm nào đó phụ thuộc việc tổ hợp 2 tập hợp. Ví dụ: nếu phép toán tổ hợp là +, thì f(k, L[i]) = k – L[i].

**Bước (1) là quan trọng nhất !**

**1235 - Coin Change (IV)**: bài toán đổi tiền quen thuộc !

Cho n loại đồng xu mệnh giá tương ứng là A1, A2, …, An. Tìm **xem có cách nào** tạo ra số tiền có giá trị k sau cho mỗi loại mệnh giá ta chỉ được dùng tối đa 2 lần. Nếu được trả về Yes nếu không trả về No. Các ràng buộc ($1 \le n \le 18$) và ($1 \le k \le 10^9$)

Ví dụ: cho 2 loại đồng xu mệnh giá 1 và 2. Nếu k = 5 ta có tạo ra được vì 5 = 1*1 + 2*2 (1 đồng 1 xu và 2 đồng 2 xu).

Ta không thể dùng quy hoạch động để giải bài này vì k có giá trị quá lớn.

Rõ ràng nếu liệt kê tất cả các số tiền có thể được tạo ra từ các mệnh giá trên thì số phần tử sẽ là $3^{18}$ (mỗi đồng xu ta có 3 cạch chọn: 0, 1, 2 và ta có nhiều nhất là 18 đồng xu). Con số này quá lớn, không thể lưu trữ. Ta sẽ tìm 2 tập hợp khác sao cho tổ hợp hai tập này thì sẽ được $3^{18}$ phần tử theo yêu cầu.

Dưới đây là một phương pháp phân đôi dùng để tạo ra 2 tập hợp thoả mãn yêu cầu:

- Tách các đồng xu thành 2 nhóm:
  - nhóm 1 gồm $A_1, A_2, \ldots, A_m$
  - nhóm 2 gồm: $A_{m+1}, A_{m+2}, \ldots, A_n$ (với m = n/2).
- Tạo tập hợp L chứa các số tiền có thể được tạo ra từ đồng xu trong nhóm 1, mỗi đồng xu sử dụng nhiều nhất là là 2 lần
- Tạo tập hợp R chứa các số tiền có thể được tạo ra từ đồng xu trong nhóm 2, mỗi đồng xu sử dụng nhiều nhất là là 2 lần
- Rõ ràng nếu tổ hợp L và R ta sẽ có được các số tiền được tạo ra từ n loại đồng xu.

Số phần tử của L và R nhiều nhất là $3^9$ =19683 ta thể lưu trữ dễ dàng.

Vấn đề cuối cùng là tạo tập L và R như thế nào. Bài toán toán này tương ứng với việc liệt kê các (chuỗi) số tam phân có chiều dài m. Nếu ta đã quen với cách liệt kê số nhị phân ta có thể áp dụng/mở rộng cách đó cho việc liệt kê. Nếu không, ta có thể sử dụng 1 phương pháp cũng hiệu quả không kém dưới đây:

- nếu m = 0, (chuỗi) số tam phân là rỗng, tổng các phần tử có giá trị 0.
- Nếu m > 1, gọi L[1], L[2], ….L[cnt] là các phần tử có chiều dài m-1, ta tạo các phần tử có chiều dài m bằng cách lần lượt thêm 0, 1, 2 vào cuối các phần tử trên.

Giải thuật sinh ra các số tiền từ các đồng tiền từ A1 đến Am như sau:

```
L[0] = 0;
cnt = 1;
for (int i = 1; i <= m; i++) {
   N1 = cnt;
   for (int j = 0; j < N1; j++)
      L[cnt++] = L[j] + A[i];   //dùng A[i] 1 lần
      L[cnt++] = L[j] + 2*A[i]; //dùng A[i] 2 lần
      //các phần tử L[0] -> L[N1-1] là các số tiền
      //được tạo ra mà không dùng A[i]

   }
}
```

Làm tương tự để sinh ra tập R từ các đồng xu $A_{m+1}, A_{m+2}, \ldots, A_n$.

Nào, các bạn thử submit nhé !

**Bài tương tự:**

1127 – Funny Knapsack