# A. Giá trị lớn nhất [CMAX.*]

Cho $n$ đường thẳng có phương trình

$$y = a_i \cdot x + b_i \ (i = 1 \div n)$$

và $m$ giá trị $x_1, x_2, \dots, x_m$

Hãy tính giá trị của hàm:

$$f(x) = \max\{a_1 \cdot x + b_1, a_2 \cdot x + b_2, \dots, a_n \cdot x + b_n\}$$

tại các giá trị $x$ đã cho

**Input:**

- Dòng đầu tiên chứa số nguyên dương $n$ ($n \le 10^5$)
- $n$ dòng tiếp theo, dòng thứ $i$ chứa hai số nguyên $a_i, b_i$ ($|a_i|, |b_i| \le 10^9$)
- Dòng tiếp theo chứa số nguyên dương $m$ ($m \le 10^5$)
- $m$ dòng cuối cùng, dòng thứ $i$ chứa số nguyên $x_i$ ($|x_i| \le 10^9$)

**Output:** In ra $m$ dòng, dòng thứ $i$ chứa số nguyên $f(x_i)$

**Example:**

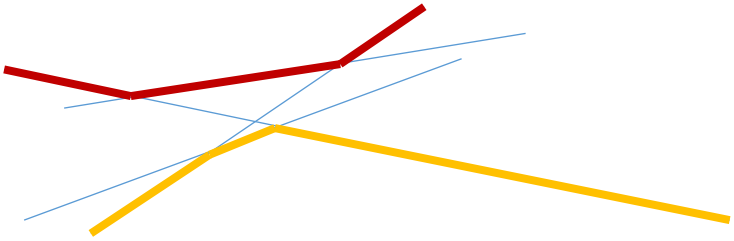
| Input | Output |
|---|---|
| 3<br>1 2<br>4 6<br>3 1<br>3<br>2<br>-10<br>100 | 14<br>-8<br>406 |

## Thuật toán (Lý thuyết)

Thuật toán bao gồm hai bước là xây dựng hàm $f(x)$ và tính giá trị $f(x)$ khi $x = x_0$

### *a) Xây dựng hàm $f(x)$*

Để minh họa ta vẽ đồ thị của hàm $f(x)$ (đường màu đỏ):



Các nhận xét quan trọng:

- Đồ thị hàm $f(x)$ là một đường gấp khúc với mỗi đoạn thẳng của đường gấp khúc này là một đoạn thẳng trên một đường thẳng đã cho.
- Hệ số góc của các đường thẳng tăng dần (theo chiều tăng của $x$)
- Các đường đoạn thẳng của đường $f(x)$ có tính chất "lồi" tức là một cạnh của một đa giác lồi nào đó.

Do vậy không mất tổng quát ta có thể coi các đường thẳng đã cho có hệ số góc tăng dần:

$$l_1 = \binom{a_1}{b_1} \le l_2 = \binom{a_2}{b_2} \le \cdots \le l_n = \binom{a_n}{b_n}$$

và việc xây dựng hàm $f$ qui về việc tìm dãy chỉ số $1 \le s_1 < s_2 < \cdots < s_m \le n$ để duy trì tính chất cực đại (được xét ở dưới đây).

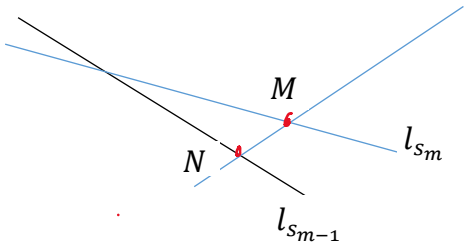
+) Đầu tiên nếu chỉ có 1 đường thẳng $l_1$ hiển nhiên $f$ cũng chỉ có 1 đoạn thẳng $s_1 = 1$

+) Giả sử đến bước $i - 1$ ta đã xây dựng được đường gấp khúc "max":

$$s_1, s_2, \dots, s_m$$

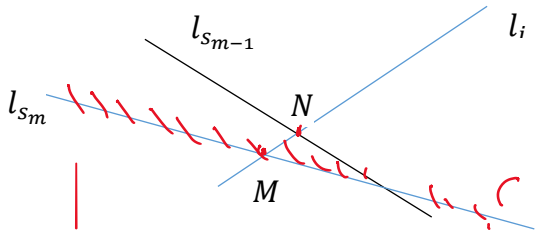
Khi thêm đường thẳng $l_i$ hai trường hợp xảy ra như hình dưới đây:

Trường hợp 1: Hoành độ giao điểm của $M$ lớn hơn hoành độ giao điểm của $N$ (ở đây $M$ là giao điểm của đường thẳng $l_i$ với $l_{s_m}$ còn $N$ là giao điểm của $l_i$ với $l_{s_{m-1}}$:




Trong trường hợp này, đường gấp khúc "max" thêm đoạn thẳng $l_i$

Trường hợp 2: Hoành độ giao điểm của $M$ nhỏ hơn hoặc bằng hoành độ giao điểm của $N$




Trong trường hợp này thì đường thẳng $l_{s_m}$ luôn nằm dưới đường thẳng $l_i$ hoặc $l_{s_{m-1}}$ do đó ta có thể bỏ $l_{s_m}$ ra khỏi đường gấp khúc max (giảm $m$). Quá trình này lặp cho đến khi gặp trường hợp 1 hoặc $m = 0$. Khi đó đường $l_i$ sẽ được thêm vào đường gấp khúc.

Như vậy $s_1, s_2, \ldots$ sẽ hoạt động như một ngăn xếp. Mỗi đường thẳng sẽ được thêm vào ngăn xếp 1 lần và lấy ra không quá 1 lần. Do vậy khi xét đến $l_n$ với thời gian $O(n)$ ta thu được đường gấp khúc max:

```
m = 0;
for (int i = 1; i <= n; ++i) {
    while (m > 0 && !ok (i))
        --m;
    s[++m] = i;
}
```

Ở đây hàm ok(i) kiểm tra xem hoành độ của $M$ có lớn hơn hoành độ của $N$:

```
bool ok (int i) {
    int u = s[m];
    if (L[i].ft==L[u].ft) return false;
    if (m==1) return true;
    double xM = 1.0 * (L[u].sc - L[i].sc) / (L[i].ft - L[u].ft);
    u = s[m - 1];
    double xN = 1.0 * (L[u].sc - L[i].sc) / (L[i].ft - L[u].ft);
    return (xM > xN);
}
```

Trong đoạn code trên đã sử dụng:

```
#define ft first
#define sc second
```

Mảng s[1], s[2], ..., s[m] là danh sách các đường thẳng tham gia vào đường gấp khúc max theo giá trị $x$ tăng dần.

Cuối cùng, để mô tả tường minh đường gấp khúc max (phục vụ cho các bài toán qui hoạch động sau này) ta xây dựng 3 mảng:

**double x[maxn];** // Mảng tọa độ các giao điểm

**int p[maxn];** // Hệ số góc của các đoạn gấp khúc

**int q[maxn];** // Hằng số của đoạn gấp khúc

Code minh họa như sau:

```
x[0] = -INF, p[0] = L[s[1]].ft, q[0] = L[s[1]].sc;
for (int i = 1; i < m; ++i) {
    x[i] = 1.0 * (L[s[i]].sc - L[s[i + 1]].sc) / (L[s[i + 1]].ft - L[s[i]].ft);
    p[i] = L[s[i + 1]].ft;
    q[i] = L[s[i + 1]].sc;
}
x[m]=INF;
```

Chú ý kết quả:

- $-\infty = x[0] < x[1] < x[2] < \ldots < x[m-1] < x[m] = \infty$
- Trên đoạn $[x_i, x_{i+1}]$ phương trình đường thẳng là $y = p_i \cdot x + q_i$

***b) Tính giá trị f(x) tại x=z:***

Trước tiên ta tìm giá trị $u$ lớn nhất để $x[u] \le z$ khi đó:

$$f(z) = p_u \cdot z + q_u$$

Code:

```
int u = upper_bound (x, x + m, z) - x - 1;
cout <<  p[u]*z + q[u]);
```

Chú ý rằng đối với bài toán tìm min xử lý tương tự nhưng các hệ số góc giảm dần

## 3. Qui hoạch động lồi A [DCVA.*]

Cho dãy số vô hạn $dp_1, dp_2, \ldots$ được xác định bởi:

- $dp_1 = c$
- $dp_i = \min\limits_{1 \le j < i} \{dp_j + b_j \cdot a_i\}$ với $i = 2,3, \ldots$

Ở đây các dãy $b_1, b_2, \ldots$ và $a_1, a_2, \ldots$ cho trước và thỏa mãn điều kiện $b_j \ge b_{j+1}$, $a_i \le a_{i+1}$

*Yêu cầu:* Tính $dp_n$

**Input:**

- Dòng đầu tiên chứa hai số nguyên $n, c$ $(1 \le n \le 10^5, |c| \le 100)$
- Dòng thứ hai chứa $n$ số nguyên $a_1, a_2, \ldots, a_n$ $(|a_i| \le 100)$
- Dòng thứ ba chứa $n$ số nguyên $b_1, b_2, \ldots, b_n$ $(|b_i| \le 100)$

**Output:** Một số nguyên là kết quả tìm được

**Example:**

| Input | Output |
|---|---|
| 4 -67<br>-2 3 6 6<br>6 5 4 -1 | -31 |

**Thuật toán:**

Xây dựng tập hợp các đường thẳng $l_j: y_j(x) = b_j \cdot x + dp_j$. Theo giả thiết của đề bài:

$$b_1 \ge b_2 \ge \cdots \ge b_n$$

Nên hệ số góc của các đường thẳng này giảm dần. Theo công thức truy hồi ta cần tìm:

$$\min\{y_1(a_i), y_2(a_i), \ldots, y_{i-1}(a_i)\}$$

Do giả thiết $a_1 \le a_2 \le \cdots \le a_n$ nên các chỉ số $u$ lớn nhất thỏa mãn $x[u] \le a_i$ tăng dần theo $i$. Vì vậy không cần phải tìm kiếm nhị phân để tính $u$. Thay vào đó sử dụng kỹ thuật "hai con trỏ" để tìm $u$ tiếp theo. Ngoài ra tập hợp các đường thẳng luôn được thêm vào khi $i$ tăng, do đó ta có thể kết hợp song song hai việc xây dựng đường gấp khúc và tính giá trị.

Tham khảo đoạn code dưới đây:

```
dp[1]=c;
k=1;
x[1]=-oo; p[1]=b[1]; q[1]=dp[1];
u=1;
for(int i=2;i≤n;++i) {
    // u là chỉ số lớn nhất mà x[u]≤a[i];
    while (u≤k && x[u]≤a[i]) ++u;
    --u;

    // Tính giá trị hàm min tại a[i]
    dp[i]=q[u]+p[u]*a[i];

    if (b[i]==p[k] && dp[i]≥q[k]) continue; // Bỏ qua dp[i]+b[i]*x

    // Tính lại đường gấp khúc min
    while (k>1 && !ok(i)) --k;
    if (u>k) u=k;
    if (k==1 && p[1]==b[i]) q[1]=dp[i]; else {
        p[++k]=b[i]; q[k]=dp[i];
        x[k]=1.0*(q[k-1]-q[k])/(p[k]-p[k-1]);
    }
}
```

Ở đây hàm ok(i) được viết như sau:

```
bool ok(int i) {
     if (b[i]==p[k]) return false;
     double xm=1.0*(dp[i]-q[k])/(p[k]-b[i]);
     double xn=1.0*(dp[i]-q[k-1])/(p[k-1]-b[i]);
     return (xn<xm);
}
```

KNUTT Optimization:

$$dp(i,j) = \min_{i \le k < j} [dp(i,k) + dp(k+1,j) + C(i,j)]$$

Điều kiện:

$$opt(i,j-1) \le opt(i,j) \le opt(i+1,j)$$

Kiểm tra với hàm C(i,j):

1. $C(b,c) \le C(a,d)$
2. $C(a,c) + C(b,d) \le C(a,d) + C(b,c)$

```
int solve() {
    int N;
    ... // read N and input
    int dp[N][N], opt[N][N];
```

```
    auto C = [&](int i, int j) {
        ... // Implement cost function C.
    };

    for (int i = 0; i < N; i++) {
        opt[i][i] = i;
        ... // Initialize dp[i][i] according to the problem
    }

    for (int i = N-2; i >= 0; i--) {
        for (int j = i+1; j < N; j++) {
            int mn = INT_MAX;
            int cost = C(i, j);
            for (int k = opt[i][j-1]; k <= min(j-1, opt[i+1][j]); k++) {
                if (mn >= dp[i][k] + dp[k+1][j] + cost) {
                    opt[i][j] = k;
                    mn = dp[i][k] + dp[k+1][j] + cost;
                }
            }
            dp[i][j] = mn;
        }
    }

    cout << dp[0][N-1] << endl;
}
```

## DIVIDE AND CONQUER DP

Chia để trị (divide and conquer) là một phương pháp tối ưu việc tính công thức qui hoạch động.

**Điều kiện để thực hiện**

Một vài bài toán qui hoạch động đưa về việc tính công thức dạng:

$$dp(i,j) = \min_{0 \le k \le j} \{dp(i-1,k-1) + C(k,j)\}$$

Ở đây $C(k,j)$ là hàm chi phí và $dp(i,j) = 0$ khi $j < 0$.

Giả sử $0 \le i < m$ và $0 \le j < n$ và hàm chi phí có thời gian tính là $O(1)$. Việc tính đơn giản công thức qui hoạch động trên có thời gian là $O(mn^2)$ vì có $m \times n$ trạng thái và $n$ trạng thái chuyển. Đặt $opt(i,j)$ là giá trị $k$ làm biểu thức trên đạt giá trị nhỏ nhất. Nếu $opt(i,j) \le opt(i,j+1)$ với mọi $i,j$ chúng ta có thể sử dụng kỹ thuật chia để trị. Đây được gọi là *điều kiện đơn điệu*: Điểm phân tách tối ưu khi $i$ cố định sẽ tăng khi $j$ tăng.

Ta sẽ tính công thức qui hoạch động hiệu quả hơn. Giả sử đã xác định được $opt(i,j)$ khi cố định $i$ và $j$. Khi đó với bất kỳ $j' < j$ ta biết rằng $opt(i,j') \le opt(i,j)$. Điều này có nghĩa là khi tính $opt(i,j')$ ta không phải kiểm tra nhiều vị trí cho vị trí đạt min (điểm chia tách).

Để giảm thiểu thời gian chạy chúng ta áp dụng kỹ thuật chia để trị. Trước tiên xác định $opt(i,n/2)$, sau đó tính $opt(i,n/4)$ biết rằng nó nhỏ hơn hoặc bằng $opt(i,n/2)$ và $opt(i,3n/4)$

biết rằng nó lớn hơn hoặc bằng $opt(i, n/2)$. Bằng cách theo dõi các giới hạn trên và giới hạn dưới của lựa chọn điểm chia chúng ta đạt được thời gian $O(mn \log n)$. Mỗi giá trị có thể có của $opt(i, j)$ chỉ xuất hiện tối đa trong $\log n$ lần trong quá trình tính.

Lưu ý rằng $opt(i, j)$ "cân bằng" như thế nào không quan trọng. Trên một mức cố định, giá trị $k$ được sử dụng nhiều nhất 2 lần và có $\log n$ mức như vậy.

**Triển khai chung**

Mặc dù có thể triển khai theo nhiều cách khác nhau tùy theo từng bài nhưng dưới đây là mẫu chung: Hàm **compute** xác định hàng $i$ lưu vào mảng **dp_cur**. Trạng thái của hàng $i - 1$ được lưu trong **dp_before**. Nó được gọi đệ qui ban đầu **compute(0,n-1,0,n-1)**. Hàm xác định $m$ hàng và trả về kết quả.

```
int m, n;
vector<long long> dp_before(n), dp_cur(n);

long long C(int i, int j);

// compute dp_cur[l], ... dp_cur[r] (inclusive)
void compute(int l, int r, int optl, int optr) {
    if (l > r)
        return;

    int mid = (l + r) >> 1;
    pair<long long, int> best = {LLONG_MAX, -1};

    for (int k = optl; k <= min(mid, optr); k++) {
        best = min(best, {(k ? dp_before[k - 1] : 0) + C(k, mid), k});
    }

    dp_cur[mid] = best.first;
    int opt = best.second;

    compute(l, mid - 1, optl, opt);
    compute(mid + 1, r, opt, optr);
}

int solve() {
    for (int i = 0; i < n; i++)
        dp_before[i] = C(0, i);

    for (int i = 1; i < m; i++) {
        compute(0, n - 1, 0, n - 1);
        dp_before = dp_cur;
    }

    return dp_before[n - 1];
}
```