Một số kĩ thuật tối ưu hoá thuật toán Quy Hoạch Động

Một số kĩ thuật tối ưu hoá thuật toán Quy Hoạch Động

Tác giả: **Lê Anh Đức** - A2K42-PBC

Quy hoạch động (QHĐ) là một lớp thuật toán rất quan trọng và có nhiều ứng dụng trong ngành khoa học máy tính. Trong các cuộc thi Olympic tin học hiện đại, QHĐ luôn là một trong những chủ đề chính. Tuy vậy, theo tôi thấy, tài liệu nâng cao về QHĐ bằng tiếng Việt hiện còn cực kỳ khan hiếm, dẫn đến học sinh/sinh viên Việt Nam bị hạn chế khả năng tiếp cận với những kỹ thuật hiện đại. Trong bài viết này, tôi sẽ trình bày một vài kỹ thuật để tối ưu hóa độ phức tạp của một số thuật toán QHĐ.

1. Đổi biến

Nhiều khi trong trạng thái QHĐ có một thành phần nào đấy với khoảng giá trị quá lớn, trong khi kết quả của hàm lại có khoảng giá trị nhỏ. Trong một vài trường hợp, ta có thể đảo nhãn để giảm số trạng thái.

Bài tập ví dụ

Longest Common Subsequence (LCS)

Đề bài

Cho xâu A độ dài m, xâu B độ dài n. Hãy tìm độ dài xâu con chung dài nhất của hai xâu, chú ý là xâu con chung có thể không liên tiếp.

Giới hạn

- $m < 10^6$
- $n \leq 5000$
- Các kí tự trong cả hai xâu là các chữ cái tiếng Anh in hoa 'A'..'Z'

Ví dụ

```
A = ADBCC
B = ABCD

LCS = ABC
Kết quả = 3
```

Lời giải

Thuật toán đơn giản

Gọi $F(i, j)$ là LCS của hai tiền tố $A_{1..i}$ và $B_{1..j}$.

Khi đó ta có thể maximize $F(i, j)$ theo $F(i-1, j)$ và $F(i, j-1)$.

Nếu $A_i = B_j$ thì ta có thể cập nhật $F(i, j)$ theo $F(i-1, j-1) + 1$.

Kết quả bài toán là $F(m, n)$.

Độ phức tạp của thuật toán này là $O(m * n)$, không khả thi với giới hạn của đề bài.

Đổi biến

Đặt $L = \min(m, n)$

Để ý rằng trong hàm QHĐ trên, các giá trị của $F(i, j)$ sẽ không vượt quá L, trong khi đó chiều thứ hai của trạng thái có thể khá lớn (lên tới $MAXM = 10^6$).

Để tối ưu hóa, ta sẽ đổi biến. Gọi $dp(i, j)$ là vị trí k nhỏ nhất sao cho $LCS(A_{1..i}, B_{1..k}) = j$.

Để tính các giá trị của dp, ta sẽ QHĐ theo kiểu cập nhật đi, thay vì đi tìm công thức trực tiếp cho các $dp(i, j)$.

Gọi $nextPos(i, c) = j > i$ nhỏ nhất mà $A_j = c$ (với c là một ký tự từ 'A' đến 'Z').

Mảng $nextPos$ có thể tính trong $T(M * 26)$.

Như vậy ta có thể tính các giá trị QHĐ như sau:

- Ban đầu khởi tạo các giá trị $dp(i, j) = \infty$, $dp(0, 0) = 0$.
- For i và j tăng dần, với mỗi giá trị $dp(i, j)$ khác vô cùng:
  - Cập nhật $dp(i+1, j)$ theo $dp(i, j)$.
  - Gọi k là vị trí xuất hiện tiếp theo của B_{i+1} trong xâu A bắt đầu từ vị trí $dp(i, j)$, tức là $k = nextPos(dp(i, j), B_{i+1})$.
  - Nếu tồn tại k, cập nhật $dp(i+1, j+1)$ theo k.

Để tính kết quả, ta sẽ chỉ cần tìm j lớn nhất mà tồn tại $dp(i, j)$ khác vô cùng.

```cpp
#include <bits/stdc++.h>

using namespace std;

const int M = 1e6 + 6;
const int N = 5005;

int dp[N][N];

char a[M], b[N];
int nextPos[M][26];
int m, n;

void minimize(int &a, int b) {
    if (a == -1 || a > b) a = b;
}

int main() {
    cin >> a + 1 >> b + 1;
    m = strlen(a + 1); n = strlen(b + 1);
    for (int c = 0; c < 26; ++c)
        for (int i = m - 1; i >= 0; --i)
            nextPos[i][c] = (a[i + 1] - 'A' == c) ? i + 1 : nextPc
    int maxLength = min(m, n);
    memset(dp, -1, sizeof dp);
    dp[0][0] = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j <= i; ++j) if (dp[i][j] >= 0) {
            minimize(dp[i + 1][j], dp[i][j]);
            int new_value = nextPos[dp[i][j]][b[i + 1] - 'A'];
            if (new_value > 0)
                minimize(dp[i + 1][j + 1], new_value);
        }
    }
    int ans = 0;
    for (int j = maxLength; j > 0; --j) {
        for (int i = j; i <= n; ++i)
            if (dp[i][j] >= 0) ans = j;
        if (ans != 0) break;
    }
    cout << ans << endl;
    return 0;
}
```

Computer (VNOI Marathon 2010)

Problem Link: COMPUTER - VNOJ .

Đề bài

Công ty phần mềm XYZ mới mua x máy tính để bàn và y máy tính xách tay. Giá một máy tính để bàn là a đồng còn giá một máy tính xách tay là b đồng. Để tránh sự thắc mắc giữa các phòng bàn, Tổng giám đốc đã đưa ra cách phân bố các máy tính này về n phòng ban như sau:

- Sắp xếp n phòng ban theo thứ tự về mức độ quan trọng của các phòng ban.
- Tiến hành phân bố các máy tính cho các phòng ban bảo đảm nếu phòng ban i có mức độ quan trọng nhỏ hơn mức độ quan trọng của phòng ban j thì tổng giá trị máy tính được phân bố cho phòng ban i không được vượt quá tổng giá trị máy tính được phân bố cho phòng ban j.
- Phòng ban nhận được tổng giá trị máy tính nhỏ nhất là lớn nhất.

Là một lập trình viên giỏi nhưng lại thuộc phòng ban có mức độ quan trọng nhỏ nhất, Thắng muốn chứng tỏ tay nghề của mình với đồng nghiệp nên đã lập trình tính ra ngay được tổng giá trị máy trình mà phòng ban mình nhận được rồi mời bạn tính lại thử xem!

Yêu cầu

Cho x, a, y, b, n. Hãy tính tổng giá trị máy tính mà phòng Thắng nhận được.

Input

x, a, y, b, n không quá 1000

Ví dụ

```
Input
3 300 2 500 2

Output
900

Input
4 300 3 500 2

Output
1300
```

Lời giải

Trước hết ta sẽ chặt nhị phân kết quả bài toán. Với mỗi giá trị chặt nhị phân, ta cần kiểm tra xem có tồn tại phương án thỏa mãn hay không.

Thuật toán sơ khai

Đặt giá trị cần kiểm tra là v.

Xét các phòng ban theo thứ tự tăng dần về mức độ quan trọng, đánh số từ 1.

Sử dụng một mảng đa chiều để đánh dấu các trạng thái có thể đạt tới. Các giá trị cần quản lí là: chỉ số của phòng ban, đã dùng số máy tính để bàn x, đã dùng số máy tính xách tay y, tổng giá trị máy tính của phòng ban trước đó.

Bắt đầu từ trạng thái (0, 0, 0, 0), ta sử dụng thuật toán loang (BFS). Cuối cùng nếu trạng thái (n, 0, 0, ...) có thể đến được, thì ta sẽ có cách phân hoạch các máy tính vào các phòng ban ứng với giá trị cận dưới v.

Không cần tính toán cụ thể cũng có thể thấy thuật toán này không thể đáp ứng về mặt thời gian (và bộ nhớ) với giới hạn của đề bài.

Nâng cấp bằng nhận xét

Nhận xét rằng ta không cần quan tâm tới thứ tự về mức độ quan trọng của các phòng ban. Với một cách phân hoạch các máy tính sao cho mỗi phòng nhận được tổng giá trị không nhỏ hơn v, ta luôn có thể sắp xếp các bộ theo giá trị không giảm ứng với các phòng ban.

Ta có trạng thái QHĐ là $F(i, x, y, value) = true$ nếu có thể phân bổ máy tính cho i phòng ban, đã dùng x máy tính để bàn và y máy tính xách tay, đã gom được tổng giá trị v cho phòng thứ $i + 1$. Cách làm này số trạng thái vẫn như trước nhưng ta đã có thể chuyển trạng thái trong $O(1)$. Cụ thể từ $F(i, x, y, value)$ ta chuyển đến $F(i, x + 1, y, value + a)$ hoặc $F(i, x, y + 1, value + b)$, chú ý là chỉ có thể dùng thêm máy xách tay nếu $x < X$ và dùng thêm máy để bàn nếu $y < Y$, đồng thời nếu giá trị value đủ lớn hơn hoặc bằng v thì ta chuyển sang trạng thái $F(i + 1, x, y, 0)$ luôn.

Đổi biến

Ở bài này, ta có thể dễ dàng đổi biến value ra làm hàm mục tiêu. Nhưng không chỉ có vậy, ta có thể đẩy cả i ra ngoài! Cụ thể, $F(x, y)$ = một cặp số $(i, value)$ lần lượt là số phòng phân bố được và số tiền gom được. Hàm mục tiêu của $F(x, y)$ là một cặp số hoàn toàn có thể so sánh được, trong đó giá trị đầu (i) được ưu tiên so sánh trước.

Cách cập nhật các $F(x, y)$ giống như phần trước, độ phức tạp vẫn là O(1) cho bước chuyển trạng thái, trong khi số trạng thái lúc này là đủ nhỏ đối với giới hạn của đề bài.

```
#include <bits/stdc++.h>

using namespace std;

const int N = 1010;

int x, y, a, b, n;

pair<int, int> F[N][N];
```

```cpp

pair<int, int> newState(pair<int, int> s, int a, int v) {
    s.second += a;
    if (s.second >= v) {
        ++s.first;
        s.second = 0;
    }
    return s;
}

bool dp(int value) {
    for (int i = 0; i <= x; ++i) for (int j = 0; j <= y; ++j)
        F[i][j] = make_pair(0, 0);
    for (int i = 0; i <= x; ++i) for (int j = 0; j <= y; ++j) {
        if (F[i][j].first == n) return 1;
        if (i < x)
            F[i + 1][j] = max(F[i + 1][j], newState(F[i][j], a, va
        if (j < y)
            F[i][j + 1] = max(F[i][j + 1], newState(F[i][j], b, va
    }
    return 0;
}

int solve() {
    int l = 0, r = (a * x + b * y) / n;
    int ans = 0;
    while (l <= r) {
        int mid = l + r >> 1;
        if (dp(mid)) {
            ans = mid;
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return ans;
}

int main() {
    cin >> x >> a >> y >> b >> n;
    cout << solve() << endl;
    cin >> x >> a >> y >> b >> n;
    cout << solve() << endl;
    return 0;
}
```

Bài luyện tập

- VNOJ - BINPACK

2. Chia để trị

Đây là kỹ thuật khá hiếm gặp, tuy nhiên lại cực kỳ mạnh.

Bài tập ví dụ

Hai nhà máy CEOI 2004

Đề bài

Có n cây cổ thụ được trồng trên một con đường từ đỉnh đổi đến chân đồi. Chính phủ địa phương quyết định cắt bỏ chúng. Để tránh hoang phí, mỗi cái cây cần được chuyển đến một nhà máy cưa.

Cây chỉ có thể được vận chuyển theo một chiều duy nhất: hướng về chân đồi. Có một nhà máy cưa ở cuối con đường. Hai nhà máy cưa có thể được xây dựng dọc theo con đường. Hãy xác định vị trí tối ưu để xây dựng chúng, để cực tiểu hóa chi phí vận chuyển. Chi phí vận chuyển 1kg gỗ đi 1 mét là 1 cent.

Yêu cầu

Viết chương trình:

- đọc dữ liệu từ đầu vào chuẩn số lượng cây, khối lượng và vị trí của chúng,
- tính toán chi phí vận chuyển tối ưu nhất,
- xuất kết quả ra đầu ra chuẩn.

Input

Dòng đầu tiên chứa số n - số lượng cây $(2 \leq n \leq 20,000)$. Các cây được đánh số $1, 2, \ldots, n$, theo chiều từ đỉnh đồi đến chân đồi.

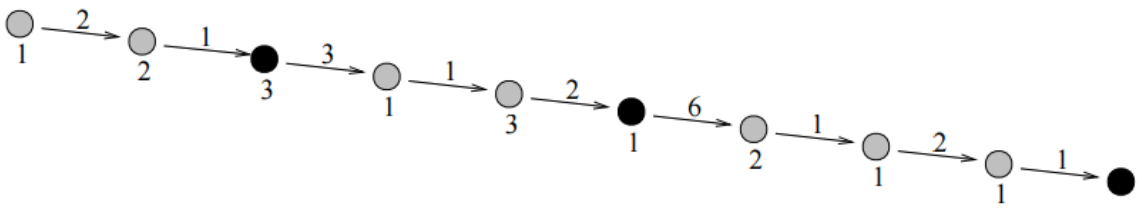
n dòng tiếp theo mỗi dòng chứa hai số nguyên dương cách nhau bởi dấu cách. Dòng thứ $i+1$ chứa w_i - khối lượng tính theo kg của cái cây thử i và d_i - khoảng cách tính theo mét giữa cây thứ i và cây $i+1$, $1 \leq w_i \leq 10000, 0 \leq d_i \leq 10000$. Số cuối cùng, d_n là khoảng cách từ cây thứ n đến chân đồi.
Dữ liệu vào đảm bảo kết quả của bài toán không vượt quá $2 * 10^9$ cent.

Output

Một dòng duy nhất chứa một số là kết quả bài toán: chi phí vận chuyển nhỏ nhất.

Ví dụ

```
Input

9
1 2
2 1
3 3
1 1
3 2
1 6
2 1
1 2
1 1

Output

26
```



Hình vẽ trên minh họa cho test ví dụ. Các hình tròn được tô đen là các vị trí có nhà máy. Kết quả sẽ là:

$1 * (2 + 1) + 2 * 1 + 1 * (1 + 2) + 3 * 2 + 2 * (1 + 2 + 1) + 1 * (2 + 1) + 1 * 1 = 26$

.

Lời giải

Trước hết ta sẽ giải quyết vấn đề tính chi phí vận chuyển nếu biết vị trí của hai nhà máy đặt thêm.

Nếu ta có thể tính được chi phí này trong $O(1)$, bài toán sẽ có thể giải được trong $O(N^2)$ - ta có thể for hết các cặp vị trí có thể đặt nhà máy.

Gọi:

- $sumW_i$ là tổng của các w_j với $i \leq j$.
- $sumD_i$ là tổng của các d_j với $i \leq j$.
- $sumWS_i$ là tổng của các $w_j * sumD_j$ với $i \leq j$.

Khi đó $cost(L, R)$ là chi phí vận chuyển các cây có chỉ số trong đoạn $[L, R]$ đến nhà máy đặt ở R là: $sumWS_L - sumWS_R - sumD_R * (sumW_L - sumW_R)$.

Như vậy ta có thể xây dựng hàm $eval(i, j)$ = chi phí nếu đặt thêm hai nhà máy ở i và j = $cost(1, i) + cost(i + 1, j) + cost(j + 1, n + 1)$.

Tuy nhiên lời giải $O(N^2)$ là chưa đủ tốt để có thể giải quyết trọn vẹn bài toán này.

Gọi $best(i)$ là vị trí $j > i$ tốt nhất nếu ta đã đặt một nhà máy ở i.

Như vậy kết quả của bài toán sẽ là $\min(eval(i, best_i))$ với $1 \leq i < n$.

Nhận xét:

- $best_i \leq best_{i+1}$. Có thể viết tường minh công thức để chứng minh.
- Ta có thể tính các $best_i$ theo thứ tự bất kỳ. Vì các giá trị $best$ không liên quan đến nhau nên VD ta có thể tính $best(3)$ rồi $best(1)$ và $best(2)$.

Như vậy ta có thuật toán sử dụng tư tưởng chia để trị như sau:

Hàm $solve(L, R, from, to)$ sẽ đi tính các $best(L..R)$, biết rằng chúng nằm trong đoạn $[from..to]$.

```
void solve(int L, int R, int from, int to) {
    if (L > R) return;
    int mid = L + R >> 1;
    best[mid] = from;
    for (int i = from + 1; i <= to; ++i)
        if (eval(mid + 1, best[mid]) > eval(mid + 1, i))
            best[mid] = i;
    solve(L, mid - 1, from, best[mid]);
    solve(mid + 1, R, best[mid], to);
}
```

Đánh giá độ phức tạp thuật toán: vì mỗi lần gọi để quy khoảng $[L, R]$ được chia đôi, nên sẽ có $O(\log N)$ tầng, mỗi tầng vòng for chỉ chạy qua $O(N)$ phần tử, vì vậy độ phức tạp của thuật toán là $O(N \log N)$.

SEQPART - Hackerrank

Đề bài

Cho dãy L số $C[1..L]$, cần chia dãy này thành G đoạn liên tiếp. Với phần tử thứ i, ta định nghĩa chi phí của nó là tích của $C[i]$ và số lượng số nằm cùng đoạn liên tiếp với nó. Chi phí của dãy số ứng với một cách phân hoạch là tổng các chi phí của các phần tử.

Hãy xác định cách phân hoạch dãy số để chi phí là nhỏ nhất.

Input

- Dòng đầu tiên chứa 2 số L và G.
- L dòng tiếp theo, chứa giá trị của dãy C.

Output

- Một dòng duy nhất chứa chi phí nhỏ nhất.

Giới hạn

- $1 < L < 8000$.
- $1 \leq G \leq 800$.
- $1 \leq C(i) \leq 10^9$.

Ví dụ

```
Input
6 3
11
11
11
24
26
100

Output
299
```

Giải thích: cách tối ưu là $C[] = (11, 11, 11), (24, 26), (100)$.

Chi phí là $11 * 3 + 11 * 3 + 11 * 3 + 24 * 2 + 26 * 2 + 100 * 1 = 299$.

Lời giải

Đây là dạng bài toán phân hoạch dãy số có thể dễ dàng giải bài QHĐ. Gọi $F(g, i)$ là chi phí nhỏ nhất nếu ta phân hoạch i phần tử đầu tiên thành g nhóm, khi đó kết quả bài toán sẽ là $F(G, L)$.

Để tìm công thức truy hồi cho hàm $F(g, i)$, ta sẽ quan tâm đến nhóm cuối cùng. Coi phần tử 0 là phần tử cầm canh ở trước phần tử thứ nhất, thì người cuối cùng không thuộc nhóm cuối có chỉ số trong đoạn $[0, i]$. Giả sử đó là người với chỉ số k, thì chi phí của cách phân hoạch sẽ là $F(g-1, k) + Cost(k+1, i)$, với $Cost(i, j)$ là chi phí nếu phân $j - i + 1$ người có chỉ số $[i, j]$ vào một nhóm. Như vậy:

$F(g, i) = \min(F(g-1, k) + Cost(k+1, l))$ với $0 <= k <= i$.

Chú ý là công thức này chỉ được áp dụng với $g > 1$, nếu $g = 1, F(1, i) = Cost(1, i)$, đây là trường hợp cơ sở.

Việc cài đặt chỉ đơn giản là dựng mảng 2 chiều $F[][]$, code như sau:

```cpp
#include <iostream>

using namespace std;

const int MAXL = 8008;
const int MAXG = 808;
const long long INF = (long long)1e18;

long long C[MAXL];
long long sum[MAXL];
long long F[MAXG][MAXL];

long long cost(int i, int j) {
    return (sum[j] - sum[i - 1]) * (j - i + 1);
}

int main() {
    int G, L;
    cin >> L >> G;
    for (int i = 1; i <= L; ++i) {
        cin >> C[i];
        sum[i] = sum[i - 1] + C[i];
    }

    for (int g = 1; g <= G; ++g) {
        for (int i = 0; i <= L; ++i) {
            if (g == 1) {
                F[g][i] = cost(1, i);
            } else {
                F[g][i] = INF;
                for (int k = 0; k <= i; ++k) {
                    long long new_cost = F[g - 1][k] + cost(k + 1,
                    if (F[g][i] > new_cost) F[g][i] = new_cost;
                }
            }
        }
    }
    cout << F[G][L] << endl;
    return 0;
}
```

Chú ý là ta sử dụng mảng $sum[]$ tiền xử lí $O(L)$ để có thể truy vấn tổng một đoạn (dùng ở hàm $cost()$) trong $O(1)$. Như vậy độ phức tạp của thuật toán này là $O(G*L*L)$.

Thuật toán tối ưu hơn

Gọi $P(g,i)$ là k nhỏ nhất để cực tiểu hóa $F(g,i)$, nói cách khác, $P(g,i)$ là k nhỏ nhất mà $F(g,i) = F(g-1,k) + Cost(k+1,i)$.

Tính chất quan trọng để có thể tối ưu thuật toán trên là dựa vào tính đơn điệu của $P(g,i)$, cụ thể:

$$P(g,0) \leq P(g,1) \leq P(g,2) \leq \cdots \leq P(g,L-1) \leq P(g,L)$$

Ta sẽ không chứng minh điều này ở đây, độc giả có thể tự thuyết phục rằng điều này là đúng.

Chia để trị

Để ý rằng để tính $F(g,i)$, ta chỉ cần quan tâm tới hàng trước $F(g-1)$ của ma trận:

$$F(g-1,0), F(g-1,1), \ldots, F(g-1,L).$$

Như vậy, ta có thể tính hàng $F(g)$ theo thứ tự bất kỳ.

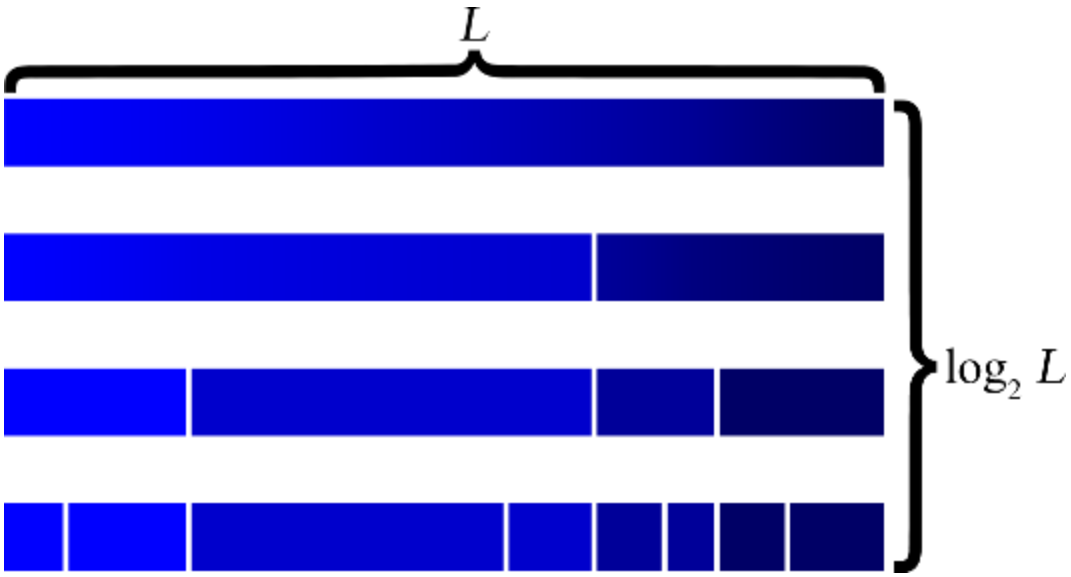
Ý tưởng là với hàng g, trước hết ta tính $F(g,mid)$ và $P(g,mid)$ với $mid = L/2$, sau đó sử dụng tính chất nêu trên $P(g,i) \leq P(g,mid)$ với $i < mid$ và $P(g,i) \geq P(g,mid)$ với $i > mid$ để đi gọi đệ quy đi tính hai nửa còn lại.

```cpp
#include <iostream>

const int MAXL = 8008;
const int MAXG = 808;
const long long INF = (long long)1e18;

using namespace std;

long long F[MAXG][MAXL], sum[MAXL], C[MAXL];
int P[MAXG][MAXL];

long long cost(int i, int j) {
    if (i > j) return 0;
    return (sum[j] - sum[i - 1]) * (j - i + 1);
}

void solve(int g, int L, int R, int optL, int optR) {
    if (L > R) return;
    int mid = (L + R) / 2;
    F[g][mid] = INF;
    for (int i = optL; i <= optR; ++i) {
```

```
        long long new_cost = F[g - 1][i] + cost(i + 1, mid);
        if (F[g][mid] > new_cost) {
            F[g][mid] = new_cost;
            P[g][mid] = i;
        }
    }
    solve(g, L, mid - 1, optL, P[g][mid]);
    solve(g, mid + 1, R, P[g][mid], optR);
}

int main() {
    int G, L;
    cin >> L >> G;
    for (int i = 1; i <= L; ++i) {
        cin >> C[i];
        sum[i] = sum[i - 1] + C[i];
    }
    for (int i = 1; i <= L; ++i) F[1][i] = cost(1, i);
    for (int g = 2; g <= G; ++g) solve(g, 1, L, 1, L);
    cout << F[G][L] << endl;
    return 0;
}
```

Chú ý rằng ta không thể đảm bảo rằng $P(g, mid)$ chia đôi đoạn $[optL, optR]$, thực tế một vài hàm $solve()$ sẽ chạy chậm hơn nhiều hàm $solve()$ khác.

Tuy nhiên ta có thể chứng minh được, xét về tổng thể thuật toán này chạy đủ nhanh. Mỗi lần ta chia đôi đoạn $[L, R]$, nên ta sẽ đảm bảo có tối đa $O(\log L)$ tầng đệ quy, như vậy với mỗi hàng g, ta chỉ mất $O(L \log L)$ để tính. Toàn bộ thuật toán có độ phức tạp là $O(G \cdot L \cdot \log L)$.



Điều kiện để Chia để trị đúng

Điều kiện theo best

Như ở bài Hai nhà máy CEOI 2004:

Nếu $best(i) \leq best(i+1)$, ta có thể sử dụng chia để trị.

Điều kiện theo cost

Nếu hàm cost thoả mãn quadrangle inequalities:

$cost(a,d) + cost(b,c) \geq cost(a,c) + cost(b,d)$ với mọi $a < b < c < d$,

ta cũng có thể sử dụng QHĐ chia để trị.

Bài luyện tập

- F - ACM ICPC Vietnam Regional 2017
- Hackerrank - Mining

3. Bao lồi đường thẳng

Các bạn có thể đọc thêm về kỹ thuật bao lồi ở link này

4. Tối ưu bằng stack

Các tính chất của stack cho phép ta xây dựng một số kỹ thuật để tối ưu thuật toán.

Bài tập ví dụ

BLOCKS - IZHO 2014

Đề bài

Cho dãy số nguyên dương $a[1], a[2], \ldots, a[N]$.
Xét các chia dãy số a thành K nhóm sao cho mỗi nhóm chứa một đoạn liên tiếp các phần tử của a. Gọi trọng số của một cách chia là tổng các phần tử lớn nhất của mỗi nhóm.

Yêu cầu

Tìm cách chia dãy số thành K nhóm sao cho trọng số của cách chia là nhỏ nhất.

Input

- Dòng 1 chứa hai số nguyên dương N và K ($K \leq N$)
- Dòng 2 gồm N số nguyên dương $a[1], a[2], \ldots, a[N]$

Output

- Gồm một số nguyên duy nhất là trọng số tìm được

Ví dụ

```
Input
5 1
1 2 3 4 5
Output
5

Input
5 2
1 2 3 4 5
Output
6
```

Giới hạn

- $1 \leq N \leq 100000$
- $1 \leq K \leq 100$
- $a[i] \leq 1000000$

Lời giải

Thuật toán QHĐ cơ sở

Gọi $F(i, j)$ là tổng trọng số nhỏ nhất để chia j số đầu tiên của dãy thành i nhóm. Công thức truy hồi là $F(i, j) = \min[F(i-1, j') + \max(a[j'+1..j])]$ với $j' < j$.

Công thức QHĐ này có thể giải trong $O(N^2 * K)$, tuy nhiên như vậy cũng chưa đạt yêu cầu.

Nâng cấp thuật toán

Ta thấy rằng chi phí chuyển trạng thái của công thức QHĐ trên đang là $O(N)$, ta có thể tập trung để tối ưu hóa điểm này.

Với mỗi vị trí i, ta gọi $L[i]$ là vị trí $j < i$ lớn nhất thỏa mãn $a[j] > a[i]$.
Như vậy trong công thức chuyển trạng thái trên, ta không cần phải for $j' < L[i]$ vì khi đó ta chuyển trực tiếp $F(i, j) = F(i, L[j])$.

Giờ ta chỉ cần quan tâm tới các j' thuộc đoạn $[L[j], j)$. Lúc này $\max(a[j'+1..j]) = a[j]$, nên ta chỉ cần tìm $\min(F(i-1, j'))$. Đây là bài toán truy vấn đoạn có thể giải trong $O(\log N)$ mỗi truy vấn. Độ phức tạp bài toán đến đây là $O(N * K * \log N)$.

Ta vẫn có thể tối ưu hơn nữa bằng cách sử dụng stack để hỗ trợ xử lí các truy vấn. Ta duy trì môt stack, mỗi phần tử chứa hai tham số là $minF$ và $index$. Stack luôn chứa các $a[index]$ giảm dần, còn $minF$ được cập nhật lại để chứa $minF$ trong đoạn $[L[index]..index-1]$.

```cpp
#include <bits/stdc++.h>

using namespace std;

const int N = 1e5 + 5;
const int K = 105;
const int INF = 0x3f3f3f3f;

int a[N];
int L[N], minF[N];
int dp[K][N];
int n, k;

int main() {
    cin >> n >> k;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    memset(dp, 0x3f, sizeof dp);
    dp[1][0] = 0;
    for (int i = 1; i <= n; ++i) dp[1][i] = max(dp[1][i - 1], a[i]
    for (int i = 2; i <= k; ++i) {
        stack<pair<int, int> > S;
        for (int j = i; j <= n; ++j) {
            int minF = dp[i - 1][j - 1];
            while (!S.empty() && a[S.top().second] <= a[j]) {
                minF = min(minF, S.top().first);
                S.pop();
            }
            dp[i][j] = min(dp[i][S.empty() ? 0 : S.top().second],
            S.push(make_pair(minF, j));
        }
    }
    cout << dp[k][n] << endl;
    return 0;
}
```

Ở bài này ta cũng có thể thay thế stack hoàn toàn bằng cấu trúc dữ liệu Left-Right:

```cpp
#include <bits/stdc++.h>

```

```cpp
using namespace std;

const int N = 1e5 + 5;
const int K = 105;
const int INF = 0x3f3f3f3f;

int a[N], L[N], minF[N];
int dp[K][N];
int n, k;

int main() {
    cin >> n >> k;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    memset(dp, 0x3f, sizeof dp);
    dp[1][0] = 0;
    for (int i = 1; i <= n; ++i) dp[1][i] = max(dp[1][i - 1], a[i]
    for (int i = 1; i <= n; ++i)
        for (L[i] = i - 1; L[i] && a[L[i]] <= a[i]; ) L[i] = L[L[i
    for (int i = 2; i <= k; ++i) {
        minF[i - 1] = INF;
        for (int j = i; j <= n; ++j) {
            minF[j] = dp[i - 1][j - 1];
            for (int t = j - 1; t > L[j]; t = L[t])
                minF[j] = min(minF[j], minF[t]);
            dp[i][j] = min(dp[i][L[j]], minF[j] + a[j]);
        }
    }
    cout << dp[k][n] << endl;
    return 0;
}
```

Đốn cây (Đề thi HSG quốc gia năm 2016)

Đề bài

Hùng đang làm việc trong Công ty cao su X. Công ty có rừng cao su rất rộng, với những hàng cây cao su trồng cách đều thẳng tắp. Theo định kỳ, người ta thường phải chặt hạ cả hàng cây cao su đã hết hạn khai thác để trồng thay thế bằng hàng cây mới. Hùng phát hiện ra một bài toán tin học liên quan đến vấn đề này: Một nhóm công nhân được giao nhiệm vụ chặt hạ hàng cây gồm N cây được trồng dọc theo một đường thẳng với khoảng cách cố định giữa hai cây liên tiếp. Nếu các công nhân cưa đổ một cây, họ có thể cho nó đổ về phía bên trái hoặc bên phải dọc theo hàng cây. Một cây khi đổ có thể lật đổ cây khác bị nó rơi vào và có thể làm đổ nhiều cây khác, theo hiệu ứng lan truyền domino. Sau khi khảo sát kỹ, Hùng đã mô tả được hiệu ứng lan truyền domino như sau: Giả sử các cây trên hàng cây được đánh số từ 1 đến N, từ trái qua phải và chiều cao của cây i là h_i $(1 \leq i \leq N)$

- Nếu cây i đổ về bên trái thì tất cả các cây j với $i - h_i < j < i$ cũng sẽ đổ;

- Nếu cây i đổ về bên phải thì tất cả các cây j với $i < j < i + h_i$ cũng sẽ đổ;
- Mỗi cây chỉ đổ một lần về bên trái hoặc bên phải.

Do đó bài toán đặt ra đối với Hùng là: Xác định số lượng nhỏ nhất các cây mà các công nhân cần cưa đổ đảm bảo hạ đổ toàn bộ hàng cây.

Yêu cầu: Giúp Hùng giải quyết bài toán đặt ra.

Dữ liệu

- Dòng đầu tiên ghi số nguyên dương N;
- Dòng thứ hai chứa N số nguyên dương $h_1, h_2, \ldots, h_n$ được ghi cách nhau bởi dấu cách, mỗi số không vượt quá 10^6.

Kết quả

- Dòng đầu tiên ghi số nguyên dương k là số lượng cây mà các công nhân cần cưa đổ;
- Dòng thứ hai ghi dãy số nguyên $c_1, c_2, \ldots, c_k$ trong đó $\|c_j\|$ $(1 \leq j \leq k)$ là dãy chỉ số của các cây theo thứ tự các công nhân phải lần lượt cưa đổ, c_j là số dương nếu cây cần cho đổ về bên phải và là số âm nếu cây cần cho đổ về bên trái.

Nếu có nhiều cách thì chỉ cần đưa ra một cách tùy ý.

Ví dụ

```
INPUT
5
1 2 3 1 1

OUTPUT
2
3 -2
```

Chú ý: Còn một cách đốn cây khác: Cưa hai cây 1 và 2 và đều cho đổ về bên phải.

Ràng buộc

- Có 40% số test ứng với 40% số điểm của bài có $1 \leq N \leq 10000$.
- Có 40% số test khác ứng với 40% số điểm của bài có $1 \leq N \leq 100000$.
- Có 20% số test còn lại ứng với 20% số điểm của bài có $1 \leq N \leq 4000000$.

Lời giải

Bước 1: Chuẩn bị

Ta sẽ xây dựng hai mảng $L[]$ và $R[]$, trong đó $L[i]$ là vị trí j nhỏ nhất mà bị cây i làm đổ nếu đẩy về bên trái, tương tự với R.

$L[i] = \min[i, \min(L[j])]$ với $i - h[i] < j < i$

Để tính $L[]$ ta duy trì một $stack$ chứa các chỉ số tăng dần. Trước khi thêm một cây i mới vào, các cây bị nó trực tiếp làm đổ sẽ bị pop ra, đồng thời ta cập nhật $L[i]$.

Bước 2: Quy hoạch động

Gọi $F(i)$ là số cây cần phải đổ nhỏ nhất để các cây có chỉ số $1..i$ đều đổ.

Để tính $F(i)$ cần xét 2 trường hợp:

- Nếu ta đẩy cây i qua trái:
  $F(i) = \min[F(j-1)+1]$ với $L[i] \le j \le i$ (1)
- Nếu cây i bị đẩy qua phải bởi cây j
  $F(i) = \min[F(j-1)+1]$ với $1 \le j \le i$ và $R[j] \ge i$ (2)

Có thể dễ dàng tính các $F[]$ trong $O(N^2)$. Có thể dùng các cấu trúc dữ liệu quản lí đoạn để giảm xuống $O(N \log N)$.

Ta có thể sử dụng $stack$ để giảm độ phức tạp xuống $O(N)$.

Để xử lí (1) ta có thể sử dụng kỹ thuật tương tự như bài BLOCK đã trình bày, tuy nhiên ta có thể đánh giá để cài đặt được ngắn gọn hơn:

$F[L[i]-1] = \min[F(j-1)+1]$ với $L[i] \le j \le i$
(phần chứng minh xin dành lại cho độc giả)

Để xử lí (2) ta sẽ sử dụng một $stack$ để lưu các vị trí có $R[]$ giảm dần, đồng thời luôn duy trì sao cho giá trị ở top của $stack$ luôn là tốt nhất. Chú ý là với $j < i$ và $R[j] \ge i$ thì $R[j] \ge R[i]$. Như vậy nếu tại mỗi bước ta pop các vị trí j có $R[j] < i$ ra khỏi $stack$, thì sẽ luôn duy trì được tính chất của $stack$ vì lúc này đảm bảo được $R[i]$ là nhỏ hơn các $R[]$ đang ở trong $stack$, đồng thời nếu $F(i-1)$ không tốt bằng giá trị ở đầu $stack$ thì ta sẽ không đẩy i vào (để đảm bảo giá trị ở top luôn là tốt nhất).

```
#include <bits/stdc++.h>
using namespace std;

const int N = 4e6 + 6;

int n;
int a[N];
int L[N], R[N];
int dp[N], trace[N];
```

```cpp
void initialize() {
    vector<int> S;
    for (int i = 1; i <= n; ++i) {
        L[i] = i;
        while (!S.empty() && S.back() > i - a[i])
        L[i] = min(L[i], L[S.back()]), S.pop_back();
        S.push_back(i);
    }
    S.clear();
    for (int i = n; i >= 1; --i) {
        R[i] = i;
        while (!S.empty() && S.back() < i + a[i])
        R[i] = max(R[i], R[S.back()]), S.pop_back();
        S.push_back(i);
    }
}

void solve() {
    for (int i = 1; i <= n; ++i) dp[i] = i, trace[i] = -i;
    vector<int> S;
    for (int i = 1; i <= n; ++i) {
        if (dp[i] > dp[L[i] - 1] + 1) dp[i] = dp[L[i] - 1] + 1, tr
        while (!S.empty() && R[S.back()] < i) S.pop_back();
        if (!S.empty() && dp[i] > dp[S.back() - 1] + 1) {
            dp[i] = dp[S.back() - 1] + 1;
            trace[i] = S.back();
        }
        if (S.empty() || (dp[S.back() - 1] > dp[i - 1])) S.push_ba
    }
    cout << dp[n] << endl;
    for (int i = n; i; i = abs(trace[i]) - 1)
    cout << (trace[i] < 0 ? -i : trace[i]) << ' ';
}

int main() {
    ios::sync_with_stdio(false);
    cin >> n;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    initialize();
    solve();
    return 0;
}
```

5. Quản lí đồ thị hàm quy hoạch động (Slope Trick)

Ở phần này ta hãy xem xét một bài toán cụ thể về ý tưởng quan sát đồ thị của hàm QHĐ để tối ưu độ phức tạp

Bài tập ví dụ

Biến đổi dãy số

Link: Codeforces

Đề bài

Cho dãy số N phần tử. Mỗi phép biến đổi có thể tăng/giảm một phần tử bất kỳ của dãy 1 đơn vị. Hãy tìm số phép biến đổi ít nhất để dãy trở thành dãy tăng.

Input

- Dòng đầu tiên là số tự nhiên N. Dòng tiếp theo là N số nguyên $A[1..N]$

Output

- Một dòng duy nhất chứa số phép biến đổi ít nhất.

Giới hạn

- $N < 100000$
- $1 \leq A[i] \leq 1000000000$

Lời giải

Trước hết ta gán $A[i] = A[i] - i$ với mọi i. Bài toán trở thành biến đổi để dãy trở thành dãy không giảm.

Thuật toán QHĐ cơ sở

Đặt $F(i, j) =$ số phép biến đổi ít nhất để biến đổi dãy $A[1..i]$ thành dãy không giảm sao cho $A[i] \leq j$. Ta có:

- Với $i = 1$: $F(i, j) = ||A[i] - j||$
- Với $i > 1$: $F(i, j) = \min(F(i-1, k) + ||A[i] - k||)\ \forall k \leq j$

Kết hợp với nhận xét: Luôn tồn tại dãy cuối cùng với số phép biến đổi tối ưu mà chỉ chứa các giá trị có trong dãy ban đầu. Ta có thể giải công thức QHĐ này với độ phức tạp $O(N^2)$

Quan sát đồ thị của hàm QHĐ

Ở đây $F_i(j)$ là một hàm theo biến j. Nhận xét rằng hàm số này không tăng, tức là nó có độ dốc âm tại mọi điểm (ở đây ta coi $F_i(j+1) - F_i(j)$ là "độ dốc" tại điểm j). Để quản lí hàm số này, ta sẽ lưu lại tập hợp S những điểm mà tại đó độ dốc của hàm thay đổi, chú ý là nếu độ dốc tại x so với độ dốc tại $x-1$ khác nhau bao nhiêu thì ta lưu lại x bấy nhiêu lần.

Gọi $Opt(i)$ là vị trí x nhỏ nhất mà tại đó $F_i(x)$ đạt giá trị nhỏ nhất, tức là từ x thì độ dốc của F_i bằng 0. Vậy $F_n(Opt(n))$ là đáp án của bài toán.

Hãy cố gắng phác họa hàm F trên giấy để có thể dễ dàng hình dung phần còn lại của lời giải.

Để cập nhật lại tập hợp S_i thành S_{i+1} ta quan tâm tới 2 trường hợp sau:

- $Opt(i-1) \leq a[i]$: Trường hợp này, dễ thấy dộ dốc của hàm số của tất cả những điểm nhỏ hơn $a[i]$ sẽ giảm đi 1 (bởi vì phần này của hàm số được cộng thêm bởi một hàm bậc nhất có độ dốc là -1). Đồng thời $Opt(i) = a[i]$.
- $Opt(i-1) > a[i]$: Trường hợp này ta cần đẩy $a[i]$ vào tập hợp S hai lần. Bởi phần hàm số bên trái $a[i]$ sẽ có độ dốc giảm đi 1, trong khi phần từ $a[i]$ đến $Opt(i-1)$ có độ dốc tăng thêm 1. Để ý là $Opt(i-1)$ không còn là điểm đầu tiên mà từ đó hàm F đạt cực tiểu nữa, ta xóa $Opt(i-1)$ khỏi tập hợp S.

Các thao tác chèn xóa và lấy max của tập hợp có thể dễ dàng cài đặt bằng std::multiset trong C++, hay sử dụng Binary Heap nếu code Pascal.

Như vậy độ phức tạp của lời giải trên là $O(N \log N)$.

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    int n; cin >> n;
    multiset<int> slope_changing_points;
    long long answer = 0;
    for (int i = 1; i <= n; ++i) {
        int Ai; cin >> Ai;
        Ai -= i;
        slope_changing_points.insert(Ai);
        if (i == 1) continue;
        int opt = *slope_changing_points.rbegin();
        if (Ai < opt) {
            slope_changing_points.erase(--slope_changing_points.er
            slope_changing_points.insert(Ai);
            answer += opt - Ai;
        }
    }
    cout << answer << endl;
```

```
23 |     return 0;
   }
```

Đào vàng

Link: Topcoder SRM 610 Div1.Level3

Đề bài

Mỏ vàng là có thể được coi là một lưới gồm $(M+1) * (N+1)$ ô vuông. Các hàng được đánh số từ 0 đến N, các cột được đánh số từ 0 đến M.

Bạn có thể làm việc ở mỏ vàng một vài ngày. Bạn có thể chọn vị trí để đào trong ngày đầu tiên (ngày 0). Trong những ngày tiếp theo, bạn có thể giữ nguyên vị trí, hoặc di chuyển đến một ô vuông khác nằm trong giới hạn được mô tả sau đây.

Khi vàng được tìm thấy ở một ô bất kỳ nào đấy, bạn sẽ nhận được lợi nhuận. Tiền lời được tính bằng $N + M$ trừ đi khoảng cách Manhattan từ ô bạn đang đứng đến ô mà vàng được tìm thấy. Cụ thể nếu vàng được tìm thấy ở ô (a, b), và bạn đang đứng ở ô (c, d), tiền lời bạn nhận được là $N + M - ||a - c|| - ||b - d||$, trong đó kí hiệu $||$ biểu thị giá trị tuyệt đối.

Bạn được cho hai số nguyên N và M, hai mảng số nguyên $event_i$ và $event_j$. Với mỗi số k hợp lệ, có một mỏ vàng xuất hiện vào ngày thứ k, ở ô $(event_i[k], event_j[k])$.

Cuối cùng bạn có 2 mảng số nguyên $event_{di}$ và $event_{dj}$. Số phần tử của hai mảng này ít hơn số phần tử của hai mảng trên 1 đơn vị. Hai mảng này thể hiện giới hạn của việc di chuyển của bạn trong ngày. Cụ thể, với mỗi k, giữa ngày k và $k+1$, bạn đang ở ô (a, b) và chỉ đi được đến ô (c, d) nếu $||a - c|| \leq event_{di}[k]$ và $||b - d|| \leq event_{dj}[k]$.

Hãy xác định tổng lợi nhuận cực đại bạn có thể kiếm được nếu lựa chọn tối ưu làm việc tại ô nào trong mỗi ngày.

Input

- Dòng đầu tiên chứa hai số nguyên N và M.
- Dòng thứ 2 chứa số ngày mà bạn có thể làm việc K.
- Dòng thứ 3 chứa K số nguyên mô tả mảng $event_i$.
- Dòng thứ 4 chứa K số nguyên mô tả mảng $event_j$.

Nếu $K > 1$

- Dòng thứ 5 chứa $K - 1$ số nguyên mô tả mảng $event_{di}$.
- Dòng cuối cùng chứa $K - 1$ số nguyên mô tả mảng $event_{dj}$.

Output

Một số nguyên duy nhất chứa số tiền lớn nhất mà bạn có thể kiếm được.

Giới hạn

- $1 \leq N, M \leq 1000000$
- $1 < K < 1000$

Ví dụ

```
Input
3 3
1
1
1
Output
6

Vàng tìm được ở ô (1, 1). Chiến lược tốt nhất là làm việc tại đó.


Input
3 3
2
0 2
0 2
1
1
Output
10

Vàng sẽ được phát hiện ở ô (0, 0) hôm nay và ô (2, 2) vào ngày mai


Input
4 2
3
1 4 4
1 2 0
1 1
1 1
Output
15

Input
6 6
6
0 2 1 5 6 4
4 3 1 6 2 0
```

```
2 3 1 5 6
2 4 0 5 1
Output
63
```

Lời giải

Trước hết ta tóm tắt lại đề bài. Có K ngày ứng với K sự kiện. Mỗi ngày vàng được tìm thấy ở các ô được mô tả qua hai mảng $event_i[]$ và $event_j[]$. Ngày đầu tiên bạn có thể đứng ở vị trí bất kỳ, nhưng từ ngày thứ hai chỉ có thể di chuyển đến một số ô trong khoảng xác định qua hai mảng $event_{di}[]$ và $event_{dj}[]$.

Hàm mục tiêu là $N + M - ||e_i - x|| - ||e_j - y||$, trong đó (e_i, e_j) là vị trí xuất hiện vàng, còn giới hạn di chuyển là d_i theo chiều dọc và d_j theo chiều ngang. Như vậy lời giải của bài toán là độc lập đối với chiều tọa độ. Chỉ cần xem $N - ||e_i - x||$ và $M - ||e_j - y||$ là các thành phần độc lập của hàm mục tiêu.

Bài toán hai chiều giờ trở thành hai bài toán một chiều: Có $N + 1$ điểm $x[] = 0..N$; ở bước đầu tiên ta có thể chọn xuất phát ở điểm bất kỳ, sau sự kiện i và bạn ở vị trí e_i, bạn kiếm được $N - ||e_i - x||$ và có quyền tăng/giảm x một lượng tối đa là d_i, nhưng không được ra ngoài đoạn $[0..N]$. Nếu ta giải được bài toán này, ta hoàn toàn có thể giải tương tự bài toán đối với trục y.

Thuật toán quy hoạch động cơ sở

Gọi $F(i, x)$ là số tiền lời nhiều nhất có thể có nếu hiện tại đang bắt đầu lượt thứ i và đang đứng ở tọa độ x.

Nếu $i = K$, $F(i, x) = 0$.

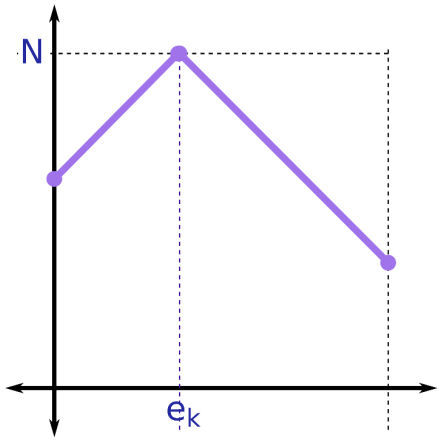
Với $i < K$, ta sẽ lời thêm $N - ||e_i - x||$, và cần phải quyết định xem tiếp theo sẽ đi tới ô nào. Cần chọn một giá trị x' thỏa mãn $0 \le x' \le N$ và $||x - x'|| \le d_i$, đồng thời giá trị $F(i + 1, x')$ là lớn nhất. Khi đó $F(i, x) = N - ||e_i - x|| + F(i + 1, x')$.

Độ phức tạp của thuật toán nếu cài đặt thông thường là $O(NNK)$, có thể tối ưu thành $O(NK)$ sử dụng deque nhưng vẫn chưa đạt yêu cầu với giới hạn đề bài.

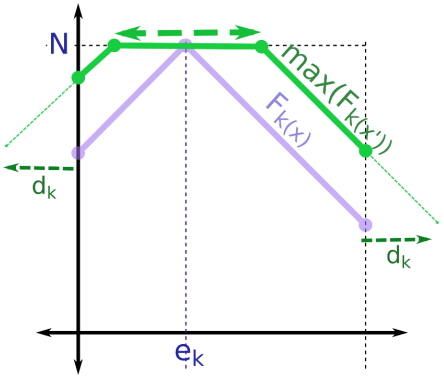
Đồ thị của hàm QHĐ

Ta có thể coi hàm QHĐ $F(i, x)$ ở trên là một hàm $f_i(x)$ nhận x là biến. Xét đồ thị của hàm số này. Dễ thấy $f_k(x) = F(K, x) = 0$, đồ thị của hàm số này là một đường thẳng.

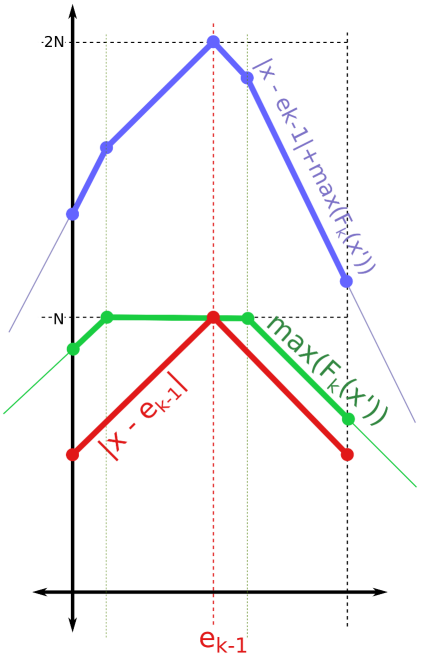
Xét hàm số $f_{k-1}(x) = N - ||e_k - 1–x||$. Đồ thị của nó sẽ có dạng:



Vấn đề trở nên phức tạp hơn với hàm f_{k-2}. Đặt $g_{k-1}(x) = \max(f_{k-1}(x'))$ với $||x'-x|| \leq d_{k-2}$. Đồ thị của hàm số này có dạng tương tự như đồ thị của hàm số $f_{k-1}(x)$:

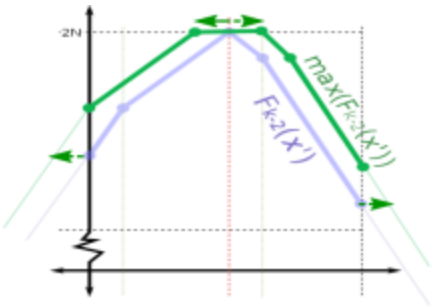


Ta cộng thêm $N - ||e_{k-2}-x||$ vào hàm $g_{k-1}(x)$, ta sẽ được đồ thị dạng:



Tương tự như vậy, ý tưởng ở đây là ta sẽ duy trì đồ thị của các hàm số $f_i(x)$ với i từ k về 0. Để làm được điều này ta cần phải thực hiện một vài thao tác:

- Tịnh tiến về hai phía: Để tìm được hàm $f(x)$ thì trước hết cần xây dựng được hàm $g(x) = \max(f_i(x') : \|x'-x\| \leq d)$. Ta chỉ cần tìm được đỉnh của hàm số, rồi tịnh tiến cả hai phía trái phải của hàm thêm một khoảng d.



- Tịnh tiến theo trục tung: Ta biểu diễn hàm số bằng danh sách các đỉnh của đường gấp khúc thì thao tác này có thể dễ dàng thực hiện.

Cuối cùng ta chỉ cần chứng minh hàm $f(x)$ luôn là hàm lõm thì cách làm trên là đúng. Ban đầu hàm số chỉ là một đường thẳng $y = 0$, sau đó ta thực hiện hai thao tác tịnh tiến về hai phía (1) và tịnh tiến theo trục tung (2). Trong đó (1) chỉ thực hiện được với hàm lõm, và sau thao tác này thì hàm số vẫn là hàm lõm. Với (2) thì ta cộng thêm một hàm lõm (hàm $N - \|e_i - x\|$), mà ta có định lý tổng hai hàm lõm cũng là hàm lõm, nên thao tác này cũng vẫn đảm bảo $f(x)$ là hàm lõm.

Độ phức tạp của thuật toán

Ta biểu diễn đồ thị của hàm số bằng danh sách các điểm, sau mỗi thao tác thì số định của đường gấp khúc tăng thêm tối đa là 1, nên số đỉnh này là một đại lượng $O(K)$. Như vậy độ phức tạp của toàn bộ thuật toán là $O(K^2)$.

```
#include <bits/stdc++.h>

using namespace std;

const int INF = 2e9;

void maximize(int &a, int b) {
    if (a < b) a = b;
}

struct Point {
    long long x, y;
    Point(long long x, long long y): x(x), y(y) {}
    bool operator < (const Point &o) const {
        return x < o.x;
    }
};
```

```
vector<Point> H;

void incConst(int delta) {
    for (int i = 0; i < H.size(); ++i) H[i].y += delta;
}

void expand(int delta) {
    int L = 0, R = 0;
    for (int i = 1; i < H.size(); ++i) {
        if (H[i].y > H[L].y) {
            L = R = i;
        } else if (H[i].y == H[L].y) {
            R = i;
        }
    }
    if (L == R) {
        H.insert(H.begin() + L + 1, H[L]);
        ++R;
    }
    for (int i = 0; i <= L; ++i) H[i].x -= delta;
    for (int i = R; i < H.size(); ++i) H[i].x += delta;
}

int calc(Point P, Point Q, int x) {
    if (P.y == Q.y) return P.y;
    int diff = P.y - Q.y;
    long long y = min(P.y, Q.y);
    long long L = x - P.x;
    long long R = Q.x - x;
    if (L == 0) return P.y;
    if (R == 0) return Q.y;
    if (diff < 0)
        y -= L * diff / (L + R);
    else
        y += R * diff / (L + R);
    return y;
}

int eval(int x) {
    for (int i = 0; i + 1 < H.size(); ++i) if (H[i].x <= x && x <=
        return calc(H[i], H[i + 1], x);
}

void mergeHull(int v) {
    //merge with y = -abs(x - v)
    int exist = -1;
    for (int i = 0; i < H.size(); ++i) if (H[i].x == v) {
        exist = i;
```

```
        break;
    }
    if (exist == -1) {
        H.push_back(Point(v, eval(v)));
        sort(H.begin(), H.end());
    }
    for (int i = 0; i < H.size(); ++i) H[i].y -= abs(H[i].x - v);
}

int solve(int len, vector<int> pos, vector<int> range) {
    H.clear();
    int n = pos.size();
    H.push_back(Point(0, len - pos[0]));
    if (pos[0] != 0) H.push_back(Point(pos[0], len));
    if (pos[0] != len) H.push_back(Point(len, len - abs(pos[0] - l
    for (int i = 1; i < n; ++i) {
        expand(range[i - 1]);
        mergeHull(pos[i]);
        incConst(len);
    }
    int ans = 0;
    int last = 0;
    for (int x = 0; x <= len; ++x) {
        while (last < H.size() && H[last].x <= x) ++last;
        if (last == H.size()) --last;
        ans = max(ans, calc(H[last - 1], H[last], x));
    }
    return ans;
}

int getMaximumGold(int N, int M, vector<int> event_i, vector<int>
    return solve(N, event_i, event_di) + solve(M, event_j, event_c
}

int main() {
    int N, M, D;
    cin >> N >> M >> D;
    vector<int> event_i(D), event_j(D), event_di(D - 1), event_dj(
    for (int i = 0; i < D; ++i) cin >> event_i[i];
    for (int i = 0; i < D; ++i) cin >> event_j[i];
    for (int i = 0; i < D - 1; ++i) cin >> event_di[i];
    for (int i = 0; i < D - 1; ++i) cin >> event_dj[i];
    cout << getMaximumGold(N, M, event_i, event_j, event_di, event
}
```