เฉลย

Thailand Online Competitive Programming Contest 2021

โจทย์การแข่งขันวันที่ 1 และ 2

วันที่ 20-21 พฤศจิกายน 2021

ID โจทย์	ชื่อโจทย์	Time	Memory	คะแนนชุดทดสอบย่อย	รวม (คะแนน)
woody	Woody	3 s	256 MB	10 40 50	100
longjump	กระโดดไกล	1 s	256 MB	10 25 45 20	100
parentheses	Royal Parentheses	1 s	256 MB	20 20 60	100
moles	Moles	1 s	256 MB	10 25 25 40	100
arranging	จัดหนังสือ	1 s	256 MB	9 16 35 40	100
guitar	กีตาร์	1 s	256 MB	5 5 20 70	100

Woody

3 second, 256 megabytes

Subtask 1 ($N \leq 20$)

เราสามารถลองทุกวิธีในการเลือกแบตเตอรี่ N ก้อนและดูว่าในแต่ละกรณีที่ทำให้ม้าไปถึงจุดสิ้นสุด ใช้เวลาชาร์จเท่าไร และเก็บกรณีที่ใช้เวลาชาร์จน้อยที่สุด

Time Complexity: $O(2^N)$

Subtask 2 ($N \leq 1000, X \leq 500, Y = 0$)

ใน subtask นี้ เราไม่จำเป็นต้องคำนึงถึงค่า Y เนื่องจากจุดเริ่มต้นเรามีค่า $Y = 0$

สังเกตว่าปัญหานี้คล้ายกับปัญหา Knapsack แต่แทนที่เราจะทำให้ผลรวม x_i ไม่เกิน X และผลรวม c_i มากที่สุด เราต้องทำให้ผลรวม x_i เกิน X และผลรวม c_i น้อยที่สุด

คำนึงด้วยว่าหากม้าไปเกิน X สามารถเก็บค่าไว้ที่ช่อง X ได้เลย เพราะถือว่าถึงจุดหมายเหมือนกัน

`https://en.wikipedia.org/wiki/Knapsack_problem`

Time Complexity: $O(NX)$

Subtask 3 (ไม่มีเงื่อนไขเพิ่มเติม)

สังเกตว่าเราต้องพิจารณาทั้งสองแกนในการคำนึงว่าเราถึงจุดหมายแล้วหรือไม่ เราจึงต้องเพิ่มมิติในการเก็บค่าของตาราง knapsack จากตารางขนาด $N \times X$ เป็นขนาด $N \times X \times Y$

$$dp(i,x,y) = \begin{cases} dp(i-1,x,y) & x < x_i \text{ และ } y < y_i \\ \min(dp(i-1,x,y), \min_{1 \leq j \leq x_i}\{dp(i-1,x-j,y-y_i)+c_i\}) & x = X \\ \min(dp(i-1,x,y), \min_{1 \leq j \leq y_i}\{dp(i-1,x-x_i,y-j)+c_i\}) & y = Y \\ \min(dp(i-1,x,y), \min_{1 \leq j \leq x_i}\{\min_{1 \leq k \leq y_i}\{dp(i-1,x-j,y-k)+c_i\}\}) & x = X \text{ and } y = Y \\ \min(dp(i-1,x,y), dp(i-1,x-x_i,y-y_i)+c_i) & \text{กรณีอื่น} \end{cases}$$

เนื่องจากตารางขนาด $N \times X \times Y$ มีขนาดเกิน Memory Limit เราสามารถลดการใช้ Memory ด้วยการเก็บแค่ค่า N ปัจจุบันและค่า N ก่อนหน้าทำให้ขนาดตารางเหลือ $2 \times X \times Y$

Time Complexity: $O(NXY)$

Solution Code:

```cpp
#include <bits/stdc++.h>
using namespace std;
#define MAXL 10000000000000000011
int n, x, y;
array<int, 3> arr[1010];
long long dp[2][550][550];
int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n >> x >> y;
  for (int i = 0; i < n; i++) {
    cin >> arr[i][0] >> arr[i][1] >> arr[i][2];
  }
  for (int i = 0; i <= x; i++) {
    for (int j = 0; j <= y; j++) {
      dp[0][i][j] = MAXL;
    }
  }
  dp[0][0][0] = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j <= x; j++) {
      for (int k = 0; k <= y; k++) {
        dp[(i + 1) % 2][j][k] = dp[i % 2][j][k];
      }
    }
    for (int j = 0; j <= x; j++) {
      for (int k = 0; k <= y; k++) {
        dp[(i + 1) % 2][min(j + arr[i][0], x)][min(k + arr[i][1], y)] =
            min(dp[(i + 1) % 2][min(j + arr[i][0], x)][min(k + arr[i][1], y)],
                dp[i % 2][j][k] + arr[i][2]);
      }
    }
  }
  long long ans = dp[n % 2][x][y];
  cout << ((ans == MAXL) ? -1 : ans) << '\n';
  return 0;
}
```

กระโดดไกล

1 second, 256 megabytes

Subtask 1 ($W = 0$)

สังเกตุว่าการกระโดดของจิงโจ้ในกรณีนี้จะเกิดขึ้นได้ก็ต่อเมื่อจิงโจ้เริ่มยืนติดกำแพง ซึ่งคือขอบทั้งสี่ขอบของตารางเท่านั้น จึงมีความเป็นไปได้แค่ 3 กรณีเท่านั้นคือ

1. จิงโจ้อยู่ที่มุมใดมุมหนึ่งใน 4 มุมของห้อง จิงโจ้จะสามารถกระโดดไปตำแหน่งใดๆในห้องโดยการกระโดด 1 ครั้งหากจุดหมายอยู่ในแถวหรือคอลลัมน์เดียวกับจุดเริ่มต้น และ 2 ครั้งไปตำแหน่งอื่นๆ ใดๆ

2. จิงโจ้อยู่ที่ขอบของห้องแต่ไม่อยู่ที่มุมของห้อง จิงโจ้จะสามารถกระโดดไปจุดใดๆในแถวเดียวกันหากเป็นขอบซ้ายขวา และจุดใดๆในคอลลัมน์เดียวกันหากเป็นขอบบนล่าง โดยจะกระโดดเพียงแค่ครั้งเดียว นอกจากตำแหน่งดังกล่าวแล้ว จิงโจ้ไม่สามารถกระโดดไปตำแหน่งอื่นได้

3. ในกรณีอื่นๆ จิงโจ้ไม่สามารถกระโดดไปไหนได้เลย

Time Complexity: $O(1)$

Subtask 2 ($N, M \leq 10^3$ และ $W \leq 10^4$)

เนื่องจากขนาดของห้องในปัญหาย่อยนี้มีขนาดเล็ก จึงสามารถมองห้องเป็นตาราง 2 มิติ ตาราง 2 มิติดังกล่าวจะมีขนาด $N \times M$ ช่อง ซึ่งรวมไม่เกิน 10^6 ช่อง จิงโจ้ที่อยู่ที่จุดเริ่มต้นจะเคลื่อนที่ได้ก็ต่อเมื่อมันอยู่ติดกับกำแพง โดยอาจเคลื่อนที่ไปได้หลายจุด ปัญหานี้จึงสามารถแก้ได้ด้วยการทำ breadth-first search หรือ flood-filling บนตารางโดยเริ่มจากตำแหน่งที่จิงโจ้อยู่

สังเกตุว่าช่องที่จิงโจ้สามารถกระโดดไปทั้งหมดนั้น จะอยู่ในแถวหรือคอลลัมน์เดียวกับตำแหน่งของจิงโจ้เท่านั้น เพราะฉะนั้นจะมีตำแหน่งที่จิงโจ้กระโดดไปได้ไม่เกิน $N + M$ ตำแหน่งเท่านั้น ไม่ว่าจิงโจ้จะอยู่ที่ไหนก็ตาม

ในการทำ breadth-first search หรือ flood-filling โดยวนหาช่องที่ไปต่อในเวลา $O(N + M)$ สำหรับทุกช่องในตาราง

Time Complexity: $O(NM \cdot (N + M))$

Subtask 3 ($N, M \leq 10^5$ และ $W \leq 10^4$)

ในปัญหาย่อยนี้จะไม่สามารถทำ Breadth-first Search แบบเดิมได้ แต่จะสังเกตุว่าจิงโจ้ไม่มีเหตุผลในการไปช่องที่ไม่ติดกำแพงที่ไม่ใช่จุดจบ เนื่องจากการไปช่องที่ไม่ใช่กำแพงที่ไม่ใช่จุดจบจะทำให้จิงโจ้ไม่สามารถกระโดดต่อไปได้ เนื่องจากขอบของห้องก็เป็นกำแพง และมีกำแพงอีก W อัน จะมีช่องที่ต้องพิจารณาเพียงแค่ $2N + 2M + 4W - 2$ ช่องเท่านั้น

ปัญหาย่อยนี้จึงสามารถแก้ได้โดยใช้ Breadth-first Search บนช่องที่ต้องพิจารณาจำนวน $2N + 2M + 4W - 2$ ช่องเท่านั้น โดยการหาว่าช่องถัดไปที่ไปได้สามารถใช้การ Binary Search ช่วยทำให้หาได้ทัน

Time complexity: $O((N + M + W) \log^2(N + M + W))$

Subtask 4 (ไม่มีเงื่อนไขเพิ่มเติม)

ปัญหานี้สามารถแก้ได้หลายวิธี เช่นการ Optimize วิธีการในปัญหาย่อยก่อนหน้าให้ใช้เวลาน้อยลงด้วยวิธีต่างๆ

อย่างไรก็ตาม ปัญหานี้สามารถแก้ได้ง่ายขึ้นเมื่อพิจารณากราฟย้อนทิศทาง และการเดินทางบนกราฟย้อนทิศทางตั้งแต่จุดปลายสุดกลับมาถึงจุดเริ่มต้น

เมื่อพิจารณากราฟย้อนทิศทาง จะพบว่าบนกราฟย้อนทิศทาง แต่ละโหนดจะมีเส้นเชื่อมออกไปไม่เกิน 4 เส้นเชื่อม นั่นคือไม่เกิน 2 เส้นเชื่อมในแถวเดียวกัน และไม่เกิน 2 เส้นเชื่อมในคอลลัมน์เดียวกัน

การหาเส้นที่เชื่อมจากโหนด ๆ หนึ่งจึงสามารถทำโดยการ Binary Search บนจำนวนกำแพงทั้งหมด W ตำแหน่ง ในแถว และคอลลัมน์เดียวกัน และจะมั่นใจได้ว่าการ Breadth-first Search จะค้นหาเฉพาะช่องที่ติดกับกำแพง หรือช่องเริ่มต้นและช่องสุดท้ายเท่านั้น จึงมีจำนวนช่องที่ต้องค้นหาแค่ไม่เกิน $O(N+M+W)$ ช่อง

Time complexity: $O((N+M+W)\log(N+M+W))$

Solution Code:

```cpp
#include<bits/stdc++.h>
using namespace std;
#define x first
#define y second
#define all(x) x.begin(), x.end()

map<int, vector<int>> wallx, wally;
map<pair<int,int>, int> vis;
queue<pair<int,pair<int,int>>> q;

int main(){
  ios_base::sync_with_stdio(false); cin.tie(NULL);
  int n, m, w; cin >> n >> m >> w;
  int sx, sy, ex, ey; cin >> sx >> sy >> ex >> ey;
  for (int i = 0; i < w; i++) {
    int x, y; cin >> x >> y;
    wallx[x].push_back(y);
    wally[y].push_back(x);
  }

  q.push({0, {ex, ey}});
  int ans = -1;
  while (!q.empty()) {
    auto t = q.front(); q.pop();
    if (vis[t.y]) continue;
    vis[t.y] = 1;

    if (t.y == make_pair(sx, sy)) { ans = t.x; break; }

    int idx = upper_bound(all(wally[t.y.y]), t.y.x) - wally[t.y.y].begin();
    if (idx == wally[t.y.y].size()) q.push({t.x+1, {n, t.y.y}});
    else q.push({t.x+1, {wally[t.y.y][idx]-1, t.y.y}});
    if (idx == 0) q.push({t.x+1, {1, t.y.y}});
    else q.push({t.x+1, {wally[t.y.y][idx-1]+1, t.y.y}});

    int idy = upper_bound(all(wallx[t.y.x]), t.y.y) - wallx[t.y.x].begin();
    if (idy == wallx[t.y.x].size()) q.push({t.x+1, {t.y.x, m}});
    else q.push({t.x+1, {t.y.x, wallx[t.y.x][idy]-1}});
    if (idy == 0) q.push({t.x+1, {t.y.x, 1}});
    else q.push({t.x+1, {t.y.x, wallx[t.y.x][idy-1]+1}});
  }
  cout << ans << endl;
  return 0;
}
```

Royal Parentheses

1 second, 256 megabytes

พิจารณาผังเมืองเป็น Graph ประเภท Tree ที่ประกอบไปด้วยโหนด N โหนดและเส้นเชื่อม $N-1$ เส้น

Subtask 1 ($N \leq 200$)

สังเกตว่าในแต่ละคู่โหนด (x, y) ที่ถูกเลือก จะสามารถแบ่งการเดินทางออกได้เป็นสามช่วงคือ

1. เดินจาก U ไปยัง x ผ่านถนนธรรมดา
2. เดินจาก x ไปยัง y ผ่านถนนพระราชทาน
3. เดินจาก y ไปยัง V ผ่านถนนธรรมดา

ซึ่งในช่วงที่หนึ่งและสามเป็น Path จากโหนดหนึ่งไปยังอีกโหนดหนึ่งบนเส้นของต้นไม้ (Tree) และช่วงที่สองเป็นการเดินผ่านเส้นเชื่อม (Edge) หนึ่งเส้น วิธีการต่อไปนี้ จะใช้ข้อสังเกตที่ว่า เส้นทางจาก U ไป x และ y ไป V มีเพียงหนึ่งเส้นทางเท่านั้นจากสมบัติของ Tree โดยเราจะตรวจสอบว่าวงเล็บของช่วงที่หนึ่งรวมกับช่วงที่สามสมดุลหรือไม่

ในการจัดเก็บค่าของวงเล็บ สามารถทำโดยการแปลงลำดับวงเล็บเป็นลำดับตัวเลข โดยให้ $+1$ แทน วงเล็บเปิด และ -1 แทน วงเล็บปิด

เช่น $(())()$ ได้เป็น $+1+1-1-1+1-1$

โดยสังเกตว่าลำดับวงเล็บที่สมดุลนั้นจะมีคุณสมบัติคือ

1. จะต้องมีวงเล็บเปิดกับวงเล็บปิดที่สามารถเลือกมาคู่กันได้ กล่าวคือ ผลรวมของลำดับตัวเลขจะต้องท่ากับศูนย์
2. เมื่อพิจารณาลำดับวงเล็บจากซ้ายไปขวา ณ ตำแหน่งใด ๆ จะต้องมีจำนวนวงเล็บเปิดมากกว่าหรือเท่ากับวงเล็บปิดเสมอ กล่าวคือ ทุก Prefix Sum ของลำดับตัวเลขจะต้องมากกว่าเท่ากับศูนย์

จะได้ว่า ใน Subtask นี้จึงสามารถไล่ทุกคู่โหนด (x, y) และทำ Graph Traversal (DFS/BFS) จากโหนด U ไป x แล้วเก็บผลรวมของลำดับวงเล็บไว้ เพื่อใช้คำนวณต่อเมื่อทำ Graph Traversal จากโหนด y ไป V โดยระหว่างการทำ Graph Tranversal ทั้งสองรอบ ให้ตรวจสอบว่า Prefix Sum ปัจจุบันที่ได้นั้น น้อยกว่าศูนย์หรือไม่ และตรวจสอบตอนท้ายว่าผลรวมทั้งหมดเป็นศูนย์หรือไม่

จะได้คำตอบเป็น จำนวณคู่โหนด (x, y) ที่ได้ผลรวมทั้งหมดเป็นศูนย์ และไม่มี Prefix Sum ใดที่น้อยกว่าศูนย์เลย ตรงตามเงื่อนไขข้างต้น

Time Complexity: $O(N^3)$

Subtask 2 ($N \leq 1000$)

สังเกตว่าสามารถลดเวลาการทำงานของวิธีการใน Subtask ที่แล้วได้ โดยการเช็คว่ามีเส้นทาง $U \to x$ ที่สามาร$y \to V$ กี่เส้น ซึ่งทำได้โดยการตรวจสอบว่าผลรวมของทั้งสองลำดับจะต้องเท่ากับศูนย์ และผลรวมของลำดับแรก ($U \to x$) จะต้องมากกว่าหรือเท่ากับ Prefix Sum ที่ต่ำที่สุดของลำดับสอง ($y \to V$) กล่าวคือ

ผลรวมของลำดับวงเล็บ $U \to x$ ต้องมากกว่าเท่ากับ $\min\{$Prefix Sum ของ $y \to V\}$

จึงสามารถทำได้โดยการทำ Graph Traversal N ครั้งจากทุกโหนดไปยังทุกโหนดอื่น ๆ และเก็บผลรวมกับ Prefix Sum ที่มีค่าน้อยที่สุดของแต่ละคู่โหนด (i, j) ไว้

หลังจากนั้น นับจำนวณคู่โหนด (x, y) ที่ (U, x) และ (y, V) ผลรวมกับ Prefix Sum ที่มีค่าน้อยที่สุดตรงตามเงื่อนไขข้างต้น

Time Complexity: $O(N^2)$

Subtask 3 (ไม่มีเงื่อนไขเพิ่มเติม)

เราสามารถดัดแปลงลำดับ ให้ไม่ต้องเช็ค Prefix Sum ที่มีค่าน้อยที่สุดระหว่างทั้งสองลำดับวงเล็บได้ โดยการกลับค่าและกลับฝั่งของลำดับหลัง แล้วตรวจสอบเหมือนวิธีที่ใช้ในลำดับแรก ดังนี้

ทำการสลับวงเล็บจากวงเล็บเปิดเป็นวงเล็บปิด และวงเล็บปิดเป็นวงเล็บเปิด หลังจากนั้นก็กลับข้างลำดับ เช่น

...()((())) กลายเป็น ((()))()...

เนื่องจากลำดับวงเล็บที่สมดุลนั้น จะสมดุลทั้งเมื่ออ่านจากซ้ายไปขวา หรืออ่านจากขวาไปซ้ายแล้วสลับวงเล็บเปิด/วงเล็บปิด

จึงทำให้สามารถตรวจสอบแค่ ผลรวมของทั้งสองลำดับ และตรวจสอบ Prefix Sum ของลำดับแรก และลำดับสองที่ถูกแปลงแล้ว

ซึ่งสามารถทำได้โดยการทำ Graph Traversal

1. จากโหนด U ไปยังทุกโหนด และเก็บผลรวมกับเช็ค Prefix Sum
2. จากโหนด V ไปยังทุกโหนด โดยคำนวณในแบบที่ลำดับถูกดัดแปลงแล้ว และเก็บผลรวมกับเช็ค Prefix Sum บนลำดับที่ถูกแปลง

หลังจากนั้น นับจำนวณคู่โหนด (x, y) ที่ผ่านเงื่อนไขและมีผลรวมที่รวมกันได้พอดี (ถ้าเป็นลำดับแรกคำนวณแบบปกติ กับลำดับที่สองคำนวณแบบดัดแปลงแล้ว กล่าวคือ มีค่าผลรวมที่เท่ากันนั่นเอง)

ซึ่งสามารถทำได้โดยอาจจะใช้ Data Structure เช่น std::map ในการเก็บและนับ

Time Complexity: $O(N \log N)$

หมายเหตุ: สามารถเก็บโดยใช้ อาเรย์ และให้ index ของช่องอาเรย์เป็นค่าผลรวม และให้ค่าในอาเรย์เก็บจำนวนโหนด ก็จะทำให้สามารถเพิ่มประสิทธิภาพของโปรแกรมให้ใช้ Time Complexity เหลือ $O(N)$ ได้

Solution Code:

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

const int N = 1e5 + 1;

int n,u,v;
vector<int> adj[N];
string s;
int val[N],sum[N][2];
bool inv[N][2];
map<int,int> res;
long long ans;

void dfs(int u,int p,int t)
{
    if(!t) sum[u][t] = sum[p][t]+val[u];
    else sum[u][t] = sum[p][t]-val[u];
    if(sum[u][t]<0 or inv[p][t]) inv[u][t] = true;
    for(int v : adj[u]) if(v!=p) dfs(v,u,t);
}


int main()
{
    ios_base::sync_with_stdio(0); cin.tie(0);

    cin >> n >> u >> v;
    for(int i = 0;i < n-1;i++)
    {
        int a,b;
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }
    cin >> s;
    for(int i = 1;i <= n;i++) if(s[i-1]=='(') val[i] = 1; else val[i] = -1;
    dfs(u,0,0);
    dfs(v,0,1);
    for(int i = 1;i <= n;i++) if(!inv[i][0]) res[sum[i][0]]++;
    for(int i = 1;i <= n;i++) if(!inv[i][1]) ans+=(long long)(res[sum[i][1]]);
    cout << ans;
}
```

Moles

1 second, 256 megabytes

สังเกตว่าเราจะสามารถทุบตัวตุ่นตัวที่ j ต่อจาก i ก็ต่อเมื่อ $|s_j - s_i| \leq t_j - t_i$ เมื่อ $t_i \leq t_j$

Subtask 1 ($|s_{i+1} - s_i| \leq t_{i+1} - t_i$ และ $t_i \leq t_{i+1}$ สำหรับ $1 \leq i < N$)

จากเงื่อนไขของ Subtask นี้ ทำให้สามารถทุบตัวตุ่นทุกตัวตามลำดับ $1, 2, 3, ..., N$ ได้เสมอ ทำให้คำตอบคือค่า N

Time Complexity: $O(1)$

Subtask 2 ($s_i < s_{i+1}, t_i < t_{i+1}$ สำหรับ $1 \leq i < N$ และ $N \leq 5000$)

จากเงื่อนไขของ Subtask สังเกตได้ว่าลำดับของตัวตุ่นที่โดนทุบจะเป็นลำดับย่อยของตัวตุ่นตัวที่ $1, 2, 3, ..., N$ เนื่องจากตัวตุ่นตัวที่ $i + 1$ จะโผล่มาหลังตัวที่ i ดังนั้นเราสามารถใช้ Dynamic Programming ในการแก้ Subtask นี้ได้

กำหนดให้ $dp(i)$ เป็นจำนวนตัวตุ่นโดนทุบมากที่สุดที่เป็นไปได้เมื่อพิจารณาจากตัวที่ 1 ถึงตัวที่ i และทุบตัวที่ i ด้วย

Base Case:

$$dp(i) = \begin{cases} 1 & |s_i - s_0| \leq t_i \text{ (สามารถทุบตัวตุ่นตัวนี้เป็นตัวแรกได้)} \\ 0 & \text{หากไม่เป็นไปตามเงื่อนไขด้านบน} \end{cases}$$

Recurrence Formula:

$$dp(i) = \max_{1 \leq j < i} \{dp(j)\} + 1$$

เมื่อ $dp(j) \neq 0$ (รองรับกรณีที่ตัวตุ่นไม่สามารถโดนทุบได้เป็นตัวแรก) และ $s_j - s_i \leq t_j - t_i$ ไม่จำเป็นต้องใช้ค่าสัมบูรณ์เนื่องจากเงื่อนไขของ Subtask

คำตอบข้อนี้เป็น $\max_{1 \leq i \leq N} \{dp(i)\}$

Time Complexity: $O(N^2)$

Subtask 3 ($N \leq 5000$)

สำหรับ Subtask นี้ ใช้วิธีแก้คล้ายกันกับ Subtask 2 โดยเพื่อทำให้วิธีการแก้ Subtask นี้ง่ายขึ้น เราจะสมมติตัวตุ่นตัวที่ 0 ตำแหน่ง s_0 ณ เวลา $t = 0$ แทนตำแหน่งเริ่มต้นของเรา จากนั้น เรียงตัวตุ่นตามเวลา t_i จากน้อยไปมาก แล้วทำ Dynamic Programming ดังนี้

Base Case:

$$dp(0) = 1 \text{ (เริ่มที่ตัวตุ่นสมมติจากที่นิยามไว้ข้างต้น)}$$

Recurrence Formula:

$$dp(i) = \max_{1 \leq j < i} \{dp(j)\} + 1$$

เมื่อ $dp(j) \neq 0$ (รองรับกรณีที่ตัวตุ่นไม่สามารถโดนทุบได้เป็นตัวแรก) และ $|s_j - s_i| \leq t_j - t_i$ ไม่จำเป็นต้องใช้ค่าสัมบูรณ์ของ $t_j - t_i$ เนื่องจากเราได้เรียง t_i จากน้อยไปมากแล้ว แต่ยังคงต้องใช้ค่าสัมบูรณ์ของ $s_j - s_i$ เพราะ Subtask นี้ไม่ได้รับประกันว่า $s_j > s_i$

คำตอบข้อนี้เป็น $\max_{1 \leq i \leq N} \{dp(i)\}$

Time Complexity: $O(N^2)$

Subtask 4 (ไม่มีเงื่อนไขเพิ่มเติม)

พิจารณาเงื่อนไข $|s_j - s_i| \leq t_j - t_i$ จะได้ว่า $t_i - t_j \leq s_j - s_i \leq t_j - t_i$ หรือ $t_i - t_j \leq s_j - s_i$ และ $s_j - s_i \leq t_j - t_i$

เราสามารถเขียนอสมการ 2 อสมการดังกล่าวในรูปต่อไปนี้: $t_i + s_i \leq t_j + s_j$ และ $t_i - s_i \leq t_j - s_j$ ตามลำดับ สังเกตว่าหากทั้งสองเงื่อนไขนี้เป็นจริง เราจะสามารถทุบตัวตุ่นตัวที่ j หลังจาก i ทันทีได้ นอกจากนี้ หาก $t_j < t_i$ อสมการดังกล่าวจะเป็นเท็จเสมอ (เนื่องจาก $|s_j - s_i| \geq 0$)

ดังนั้น เราจะเรียงตัวตุ่นตามค่า $t_i + s_i$ จากน้อยไปมาก ทำให้สำหรับตัวตุ่นตัวที่ i, j ใด ๆ ที่ $i < j, t_i + s_i \leq t_j + s_j$ เสมอ จากนั้น เราจะหาลำดับย่อยของตัวตุ่นที่ยาวที่สุด ที่มีเงื่อนไขเป็น $t_i - s_i \leq t_j - s_j$ สำหรับตัวตุ่นตัวที่ $i < j$ ในลำดับย่อยนี้ นอกจากนี้ เราจะไม่สนใจตัวตุ่นที่ไม่สามารถโดนทุบเป็นตัวแรกได้เช่นเดิม

การหาลำดับย่อยดังกล่าว เป็นการหา Longest Non-decreasing Subsequence สามารถหาได้ภายในเวลา $O(N \log N)$ ด้วยวิธี Greedy Algorithm

Time Complexity: $O(N \log N)$

Solution Code:

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

#define pii pair<int, int>
#define x first
#define y second

using namespace std;

const int N = 1e6 + 5;

int n, st;
int dp[N];
pii A[N];

int main() {
  scanf("%d %d", &n, &st);
  for (int i = 1; i <= n; i++) {
    scanf("%d %d", &A[i].x, &A[i].y);
    int a = A[i].y - A[i].x;
    int b = A[i].y + A[i].x;
    A[i] = pii(a, b);
  }
  sort(A + 1, A + n + 1);
  vector<int> lis;
  for (int i = 1; i <= n; i++) {
    if (A[i].x < -st || A[i].y < st)
      continue;
    if (lis.empty() || lis.back() <= A[i].y) {
      lis.emplace_back(A[i].y);
      continue;
    }
    auto it = upper_bound(lis.begin(), lis.end(), A[i].y);
    if (it == lis.end())
      continue;
    *it = A[i].y;
  }
  printf("%d\n", (int)lis.size());

  return 0;
}
```

จัดหนังสือ

1 second, 256 megabytes

Subtask 1 ($K = 1, N \leq 20$)

เราสามารถลองเลือกเอาหนังสือออกในทุกรูปแบบแล้ว เช็คว่าแบบใดที่ตรงกับเงื่อนไขเพื่อหาจำนวนหนังสือที่น้อยที่สุดที่ต้องเอาออก

Time Complexity: $O(2^N)$

Subtask 2-3

สังเกตว่าการที่เราหาว่าการที่เราหาวิธีการเอาหนังสือออกน้อยที่สุดนั้นทำได้ยาก เราสามารถเปลี่ยนมุมมองของโจทย์เป็นว่าเราจะเก็บหนังสือไว้บนชั้นได้มากที่สุดกี่เล่มแทน โดยโจทย์นี้เราสามารถมองเป็น weight scheduling problem โดยเราสามารถเก็บค่าตำแหน่งของหนังสือเล่มที่อยู่ซ้ายที่สุดกับอยู่ขวาที่สุด และเก็บจำนวนของหนังสือประเภทนั้น เนื่องจากหากเราเลือกจะเก็บหนังสือประเภทใดก็ตาม เราจะไม่สามารถเก็บหนังสือประเภทอื่น ระหว่างเหลื่อมหรือทับกับหนังสือช่วงที่เราจะเลือกเก็บได้

ให้ $dp[i]$ แทนจำนวนหนังสือที่เก็บได้มากที่สุด เมื่อมองแค่ i ช่องแรก

ให้ l หนังสือเล่มซ้ายสุดที่มีประเภทเหมือนหนังสือเล่มที่ i

ให้ si จำนวนหนังสือที่มีประเภทเหมือนหนังสือเล่มที่ i

ถ้าหนังสือเล่มที่ i เป็นหนังสือเล่มขวาสุดของหนังสือประเภทนั้น

$dp[i] = max(dp[i-1], dp[k] + si)$

ถ้าหนังสือเล่มที่ i ไม่ใช่หนังสือเล่มขวาสุดของหนังสือประเภทนั้น

$dp[i] = dp[i-1]$

คำตอบของคำถามคือ $N - dp[N]$

Time Complexity: $O(N)$

Subtask 4 ($K = 1$)

เราสามารถสังเกตเพิ่มเติมจาก Subtask 2-3 ได้ว่าหนังสือที่เราควรจะเลือกออกนั้นจะเป็นหนังสือที่เป็นเล่มที่อยู่ซ้ายสุด หรือขวาสุดของประเภทนั้นๆเท่านั้น โดยเราสามารถ 2D dynamic programming โดยเพิ่มมาหนึ่งมิติเพื่อเก็บว่าเราเคยเอาทำหนังสือออกแค่เล่มเดียวหรือยังในการแก้ไขปัญหา

Time Complexity: $O(N)$

Solution Code:

```
#include<bits/stdc++.h>
using namespace std;
#define pb push_back
#define F first
#define S second
int n,m,k,x[1000005],ty[1000005];
int dp[1000005][2],mx[1000005][2],mn[1000005][2];
vector<pair<int,pair<int,int> > >v[1000005];
int main() {
    scanf("%d %d",&n,&k);
    for(int i=1;i<=n;i++) {
        scanf("%d",&x[i]);
        if(ty[x[i]]==0)mx[x[i]][0]=i,mx[x[i]][1]=i,mn[x[i]][0]=i,mn[x[i]][1]=i;
        else {
            mx[x[i]][1]=mx[x[i]][0];
            mx[x[i]][0]=i;
            if(ty[x[i]]==1)mn[x[i]][1]=i;
        }
        ty[x[i]]++;
    }
    for(int i=1;i<=n;i++) {
        if(ty[i]==0)continue;
        else if(ty[i]==1)v[mx[i][0]].pb({mn[i][0]-1,{1,0}});
        else {
            v[mx[i][0]].pb({mn[i][0]-1,{ty[i],0}});
            v[mx[i][1]].pb({mn[i][0]-1,{ty[i]-1,1}});
            v[mx[i][0]].pb({mn[i][1]-1,{ty[i]-1,1}});
        }
    }
    for(int i=1;i<=n;i++) {
        for(int j=0;j<v[i].size();j++) {
            if(v[i][j].S.S==1) {
                if(dp[i][1]<dp[v[i][j].F][0]+v[i][j].S.F)dp[i][1]=dp[v[i][j].F][0]+v[i
                    ][j].S.F;
            }
            else {
                for(int cc=0;cc<2;cc++) {
                    if(dp[i][cc]<dp[v[i][j].F][cc]+v[i][j].S.F)dp[i][cc]=dp[v[i][j].F
                        ][cc]+v[i][j].S.F;
                }
            }
        }
        if(dp[i][0]<dp[i-1][0])dp[i][0]=dp[i-1][0];
        if(dp[i][0]<dp[i-1][1])dp[i][0]=dp[i-1][1];
    }
    int ans=0;
    for(int i=0;i<=k;i++)ans=max(ans,dp[n][i]);
    printf("%d",n-ans);
}
```

Guitar

1 second, 256 megabytes

หมายเหตุ ทุก Subtask จำเป็นต้องรับ input ด้วย $O(N^2)$

Subtask 1 ($K = 0$)

เนื่องจากเราไม่สามารถลบส่วนใด ๆ ของเพลงได้เลย ฉะนั้นคำตอบจึงเป็น

$$\sum_{i=1}^{M-1} p[s_i][s_{i+1}]$$

Time Complexity: $O(M)$

Subtask 2 ($K = 1$)

เราสามารถ loop เพื่อหาตำแหน่งของเพลงที่ลบแล้วทำให้เกิดค่าความเหนื่อยน้อยที่สุดได้ ฉะนั้นคำตอบจึงเป็น

$$\sum_{i=1}^{M-1} p[s_i][s_{i+1}] - \max_{1 \le i \le M} \{p[s_{i-1}][s_i] + p[s_i][s_{i+1}] - p[s_{i-1}][s_{i+1}]\}$$

โดยกำหนดให้ $p[s_0][s_i] = p[s_i][s_{M+1}] = 0$ สำหรับ $1 \le i \le M$ อย่างไรก็ตาม คำตอบที่ถูกต้องอาจไม่จำเป็นต้องลบส่วนใดของเพลงเลยก็ได้

Time Complexity: $O(M)$

Subtask 3 ($N, M, K \le 20$)

เราสามารถลองทุกวิธีในการลบลำดับของเพลง โดยมีเงื่อนไขว่าต้องลบไม่เกิน K ตำแหน่งเท่านั้น

Time Complexity: $O(2^M)$

Subtask 4 (ไม่มีเงื่อนไขเพิ่มเติม)

ปัญหานี้สามารถแก้ได้ด้วย Dynamic Programming โดยเราจะสมมุติ $s_0 = 0$ และ $s_{M+1} = M + 1$ เป็นลำดับของเพลงที่ต้องเล่นก่อนและหลังดีด s_i ใด ๆ ตามลำดับ โดยที่ $p[s_0][s_i] = p[s_i][s_{M+1}] = 0$ สำหรับ $1 \le i \le M$

กำหนดให้ $dp(i, j)$ คือค่าความเหนื่อยที่น้อยที่สุดหลังจากเล่นลำดับของเพลงตั้งแต่ $s_0, s_1, s_2, ..., s_i$ โดยที่ลบลำดับของเพลงไป j ตำแหน่ง

Base Case:

$$dp(0, 0) = 0$$

Recurrence Formula:

$$dp(i, j) = \min_{1 \le k \le min(i, j+1)} \{dp(i - k, j - k + 1) + p[s_{i-k}][s_i]\}$$

คำตอบคือ $\min_{0 \le i \le K} dp(M + 1, i)$

Time Complexity: $O(NMK)$

Solution Code:

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

const int N = 305;

int s[N];
long long p[N][N], dp[N][N];

long long solve(int m, int k)
{
    if(dp[m][k] != -1) return dp[m][k];
    if(m == 0) return 0;

    long long ret = 1e18;
    for(int i=0 ; i<=k && m-i-1>=0 ; i++) ret = min(ret, solve(m-i-1, k-i) + p[s[m-i
        -1]][s[m]]);
    return dp[m][k] = ret;
}

int main()
{
    int n, m, k;
    long long ans = 1e18;

    scanf(" %d %d %d",&n,&m,&k);
    for(int i=1 ; i<=n ; i++) for(int j=1 ; j<=n ; j++) scanf(" %lld",&p[i][j]);
    for(int i=1 ; i<=m ; i++) scanf(" %d",&s[i]);

    memset(dp, -1, sizeof dp);
    for(int i=0 ; i<=k ; i++) ans = min(ans, solve(m+1, k));

    printf("%lld\n",ans);
    return 0;
}
```