PreTOI14 #3-4 By CrushAlgo

Editorial

ในข้อนี้เราจะมี Naïve Solution O(NQ) คือสำหรับแต่ละคำถามให้ลูปทดลองตำแหน่ง L_i ถึง R_i ของ array แล้วนับว่ามีค่า C_i กี่ตัว

สังเกตว่าเราไม่จำเป็นจะต้องไล่ดูทุกตัว เพราะเราสนใจเฉพาะตัวที่มีค่าเป็น C_i เท่านั้น ดังนั้นแทนที่จะเก็บรวมเป็น array เดียวกัน เราจะนำมาเก็บใส่ Map of Vectors (**`map<int, vector<int>>`**) แทน โดยสำหรับค่าแต่ละค่าที่พบเจอได้ใน array เราจะเก็บว่าค่าดังกล่าวจะพบได้ที่ตำแหน่งใดบ้าง (เนื่องจากมีได้หลายตำแหน่งจึงเก็บไว้ใน **`vector<int>`**) โดยเราจะเก็บแบบเรียงลำดับจากน้อยไปมาก

ในการตอบคำถามแต่ละคำถาม เพียงแค่พิจารณา Vector ของค่า C_i ที่เราต้องการ แล้ว binary search เพื่อหาตำแหน่งแรกและตำแหน่งสุดท้ายที่พบ โดยตำแหน่งแรก เราจะสนใจเฉพาะตำแหน่งที่มีค่ามากกว่าหรือเท่ากับ L_i ส่วนตำแหน่งสุดท้ายจะสนใจเฉพาะตำแหน่งที่มีค่าน้อยกว่าหรือเท่ากับ R_i เมื่อนำ index ใน vector ดังกล่าวมาลบกัน บวกด้วย 1 จะได้จำนวนค่า C_i ที่พบใน array ตัวที่ L_i ถึง R_i

ดังนั้น รวมแล้ว Time Complexity ของวิธีนี้ จะเป็น O(N + Q log N) ซึ่งจะทำให้ได้คะแนนเต็ม

Solution Code

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n, q;
    scanf("%d%d", &n, &q);
    map<int, vector<int>> pos;
    for (int i = 1; i <= n; ++i) {
        int x;
        scanf("%d", &x);
        pos[x].push_back(i);
    }
    while (q--) {
        int l, r, c;
        scanf("%d%d%d", &l, &r, &c);
        auto lb = lower_bound(pos[c].begin(), pos[c].end(), l);
        auto rb = upper_bound(pos[c].begin(), pos[c].end(), r)-1;
        printf("%d\n", (int)(rb-lb+1));
    }
    return 0;
}
```

ในข้อนี้ สังเกตว่าการทำลายทีละถนนจน **S** ไม่สามารถไปหา **T** ได้นั้นมองยาก เราจึงมองในมุมกลับแทน คือสนใจถนนที่ไม่ถูกทำลาย สังเกตว่าถนนที่ไม่ถูกทำลายจะเป็นเส้นทางใดเส้นทางหนึ่งจาก **S** ไป **T** โดยมี edge หนึ่งถูกทำลายไป (เป็น edge สุดท้ายที่ถูกทำลายก่อนที่ **S** กับ **T** จะขาดจากกัน) โดยเราต้องการเส้นทาง (เมื่อพิจารณาเฉพาะ edge ที่ไม่โดนทำลาย) ให้มีผลรวมของรางวัลที่น้อยที่สุดเท่าที่เป็นไปได้ เพื่อที่จะทำให้รางวัลที่ได้จากการทำลายถนนอื่น ๆ นอกเหนือจากถนนพวกนี้มีค่ามากที่สุด

ในข้อนี้ มี Solution ที่ไม่ถูกต้องหลักๆ อยู่ 2 วิธี ดังต่อไปนี้

1) หา Shortest Path จาก **S** ไป **T** แล้วค่อยๆไล่ดูแต่ละถนนใน Shortest Path แล้วตัดถนนนั้นออกไป (Solution นี้จะได้ประมาณ 50 คะแนน)

2) หา Shortest Path จาก **S** ไป **T** โดยมี Compare Function เป็น SUM – MAX แล้วตอบ Shortest Path นั้นเลย (Solution นี้จะได้ 95 คะแนน)

ทางผู้จัดการแข่งขันขออภัยกับเหตุการณ์ที่เกิดขึ้นระหว่างการแข่งขัน (เช่น การทำการตรวจใหม่อย่างกระทันหัน) เนื่องจากไม่ได้จัดเตรียม Countertest สำหรับ 2 วิธีที่กล่าวมา โดยในอนาคตจะระมัดระวังมากกว่านี้

ส่วน Solution ที่ถูกต้อง มีหลักๆ อยู่ 2 วิธี ดังต่อไปนี้

1) หา Shortest Path 2 ทาง คือ จาก **S** ไปยังทุก ๆ จุด และจาก **T** ไปยังทุกๆจุด หลังจากเราจะทดลองทุก edge (u, v) ที่เป็นไปได้ โดย edge ดังกล่าวคือ edge สุดท้ายที่ถูกตัดและทำให้ **S** กับ **T** ขาดจากกัน จะทำให้เราได้ edge ที่ไม่ถูกทำลายคือ เส้นทางสั้นสุดจาก **S** ไปถึง u และเส้นทางสั้นสุดจาก v ไป **T** แต่เนื่องจากคำตอบสุดท้ายคือผลรวมของ edge ที่เหลือ เราจึงจดคำตอบเป็นผลรวมของทุก edge ในกราฟลบด้วย edge ในเส้นทางสั้นสุดดังกล่าว โดยเราจะจดเฉพาะคำตอบที่มากที่สุดเท่านั้น

2) หา Shortest Path บน state ที่นิยามโดยหมายเลข node และสถานะ 2 อย่างคือ

(0) อยู่ระหว่างการหา Shortest Path แบบปกติ (คือนับผลรวมของทุก edge ใน path)

(1) เคยข้ามการนับผลรวมของ edge ไปแล้ว 1 edge

โดยระหว่างทำ Dijkstra เมื่อพิจารณา edge (u, v) เราอาจจะคงสถานะเดิม (นั่นคือรวมรางวัลของ edge เข้าไปตามปกติ) หรือเปลี่ยนจากสถานะ 0 ณ node u ไปยังสถานะ 1 ของ node v ได้โดยไม่รวมรางวัลของ edge นั้น (แต่จากสถานะ 1 จะไม่มีทางกลับไปยังสถานะ 0 ได้อีก เพราะเราอนุญาตให้ข้ามได้เส้น

เดียวเท่านั้น) เมื่อหา Shortest Path จาก node **S** ในสถานะ 0 ไปยัง node **T** ในสถานะ 1 ได้ ก็จะทำให้ได้คำตอบเป็นผลรวมของทุก edge ลบด้วยผลรวมของ edge ที่พิจารณาใน path ดังกล่าว

Solution Code 1

```cpp
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;
const int N = 100010;
const int INF = 1e9;
int n, m;
vector<pii> G[N];
vector<int> dijkstra(int s)
{
    vector<bool> vis(n, false);
    vector<int> dist(n, INF);
    dist[s] = 0;
    priority_queue<pii, vector<pii>, greater<pii>> Q;
    Q.push({dist[s], s});
    while (!Q.empty()) {
        int u = Q.top().second;
        Q.pop();
        if (vis[u])
            continue;
        vis[u] = true;
        for (auto v : G[u]) {
            if (!vis[v.first] && dist[u]+v.second < dist[v.first]) {
                dist[v.first] = dist[u]+v.second;
                Q.push({dist[v.first], v.first});
            }
        }
    }
    return dist;
}
int main()
{
    int s, t;
    scanf("%d%d%d%d", &n, &m, &s, &t);
    int sum = 0;
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        G[u].push_back({v, w});
        G[v].push_back({u, w});
        sum += w;
    }
    vector<int> S = dijkstra(s), T = dijkstra(t);
```

```
    int ans = 0;
    for (int u = 0; u < n; ++u) {
        for (auto v : G[u])
            ans = max(ans, sum - S[u] - T[v.first]);
    }
    printf("%d\n", ans);
    return 0;
}
```

Solution Code 2

```
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;
using pipii = pair<int, pii>;
const int N = 100010;
const int INF = 1e9;
int n, m;
vector<pii> G[N];
int dist[N][2];
bool vis[N][2];
int main()
{
    int s, t;
    scanf("%d%d%d%d", &n, &m, &s, &t);
    int sum = 0;
    for (int i = 0; i < n; ++i)
        dist[i][0] = dist[i][1] = INF;
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        G[u].push_back({v, w});
        G[v].push_back({u, w});
        sum += w;
    }
    dist[s][0] = 0;
    priority_queue<pipii, vector<pipii>, greater<pipii>> Q;
    Q.push({dist[s][0], {s, 0}});
    while (!Q.empty()) {
        int u = Q.top().second.first, t = Q.top().second.second;
        Q.pop();
        if (vis[u][t])
            continue;
        vis[u][t] = true;
        for (auto v : G[u]) {
            if (!vis[v.first][t] && dist[u][t]+v.second < dist[v.first][t]){
                dist[v.first][t] = dist[u][t]+v.second;
                Q.push({dist[v.first][t], {v.first, t}});
```

```
            }
            if (t == 0 && !vis[v.first][1] && dist[u][0] <
dist[v.first][1]){
                dist[v.first][1] = dist[u][0];
                Q.push({dist[v.first][1], {v.first, 1}});
            }
        }
    }
    printf("%d\n", sum-dist[t][1]);
    return 0;
}
```

สิ่งหนึ่งที่ต้องระวังในการทำโจทย์ข้อนี้คือการอ่านโจทย์ เพราะหากอ่านไม่ละเอียด อาจจะพลาดเงื่อนไขสำคัญข้อหนึ่งได้ นั่นคือ “จำนวนเส้นเชื่อมจะเท่ากับจำนวนดาวฤกษ์” อีกนัยหนึ่งคือ “จำนวน edge จะเท่ากับจำนวน node” (N = M) ในโจทย์กล่าวถึงเงื่อนไขนี้เพียงครั้งเดียวเท่านั้น เพราะฉะนั้นหากใครอ่านผ่านไปแล้วไม่ทราบเงื่อนไขนี้ ก็อาจจะทำให้ไม่สามารถทำโจทย์ข้อนี้ได้

จากเงื่อนไขดังกล่าว จะทำให้สรุปได้ว่า Supernova เป็นกราฟต้นไม้ (N-1 edge) แต่มี edge เพิ่มมาอีก 1 edge (รวมเป็น N edge พอดี) ดังนั้นหากลองวาด ๆ ดูก็จะได้กราฟที่มีลักษณะเป็น cycle ตรงกลางเพียง cycle เดียวและมีต้นไม้แตกออกไปจากแต่ละ node

การหาค่าความสำคัญของแต่ละ edge สามารถสังเกตได้จากการทดลองตัด edge นั้น แล้วสังเกตว่ามีกี่คู่ที่ไม่สามารถเดินทางไปถึงกันได้ จะเห็นได้ว่าแต่ละ edge ใน cycle จะมีค่าความสำคัญเท่ากับ 0 เนื่องจาก path ทุก path ที่เดินผ่านเส้นนี้ สามารถเปลี่ยนให้เดินอ้อมผ่านอีกด้านของ cycle ก็ได้

ส่วน edge ใน tree เมื่อตัดแล้วจะทำให้กราฟแบ่งออกเป็นสองฝั่ง node จากฝั่ง คือฝั่งที่เป็นต้นไม้ย่อย (subtree) ของต้นไม้หนึ่งใน cycle กับฝั่ง node ที่เหลือของกราฟ โดย node จากฝั่งหนึ่งจะข้ามไปอีกฝั่งหนึ่งไม่ได้ ดังนั้น ค่าความสำคัญของ edge นี้จะเท่ากับ จำนวน node ใน subtree คูณกับจำนวน node ที่เหลือ (นั่นคือ N ลบด้วยจำนวน node ใน subtree)

สำหรับการ implement ก่อนอื่นเราจะต้องหา cycle ของกราฟก่อน ซึ่งอาจจะทำได้หลายวิธี เช่น Depth-first Search หา cycle หรือตัดต้นไม้ทิ้งโดยค่อย ๆ ตัดจาก leaf ของต้นไม้เข้ามาจนเข้ามาถึง cycle (สังเกต leaf ได้จากการที่ degree ของ node เท่ากับ 1)

เมื่อหา cycle ได้แล้ว ก็เพียงแค่เริ่มทำ DFS จากแต่ละจุดใน cycle เพื่อหาขนาดของ subtree โดยแต่ละ edge ให้คำนวณคำตอบไว้ คำตอบจะเท่ากับขนาดของ subtree นั้นคูณกับจำนวน node ที่เหลือ (หรือแทนที่จะใช้ DFS เราสามารถคำนวณขนาดของ subtree ระหว่างที่หา cycle โดยใช้ degree ได้เลย)

Solution Code

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
using pii = pair<int, int>;
const int N = 100010;
```

```cpp
int n, U[N], V[N], deg[N], sz[N];
vector<pii> G[N];
ll ans[N];
void dfs(int u, int p, int e) {
    sz[u] = 1;
    for (auto v : G[u]) if (v.first != p && deg[v.first] == 0) {
        dfs(v.first, u, v.second);
        sz[u] += sz[v.first];
    }
    ans[e] = (ll)(n-sz[u]) * sz[u];
}
int main() {
    scanf("%d%*d", &n); // n = m
    for (int i = 1; i <= n; ++i) {
        int u, v;
        scanf("%d%d", &u, &v);
        U[i] = u, V[i] = v;
        G[u].push_back({v, i});
        G[v].push_back({u, i});
        ++deg[u], ++deg[v];
    }
    queue<int> Q;
    for (int u = 1; u <= n; ++u)
        if (deg[u] == 1)
            Q.push(u);
    while (!Q.empty()) {
        int u = Q.front();
        Q.pop();
        for (auto v : G[u]) if (deg[v.first] > 0) {
            --deg[u];
            if (--deg[v.first] == 1)
                Q.push(v.first);
        }
    }
    for (int u = 1; u <= n; ++u)
        if (deg[u] == 2) // start from node in cycle only
            dfs(u, 0, 0);
    for (int i = 1; i <= n; ++i) {
        if (i != 1) printf(" ");
        if (deg[U[i]] == 2 && deg[V[i]] == 2) // ignore edge in cycle
            ans[i] = 0;
        printf("%lld", ans[i]);
    }
    printf("\n");
    return 0;
}
```

ก่อนอื่น เราจะทำการ hash ทุก substring ของ **A** โดยเราจะใช้วิธีการ hash ที่มีโอกาสเกิด hash collision (การชนกันของ hash) น้อย วิธีหนึ่งที่เป็นไปได้คือทำดังนี้

1) กำหนดค่าให้ตัวอักษร A = P^1, B = P^2, ..., Z = P^{26} (P เป็นจำนวนเฉพาะ) โดยเราจะเก็บในตัวแปร `long long` - หากค่าตัวเลขใหญ่เกิน เราจะปล่อยให้ตัวเลข overflow ไป

2) ลูปทดลองทุกจุดเริ่มต้นของ substring (สมมุติว่าเริ่มต้นที่ตำแหน่ง i) เริ่มต้นกำหนดให้ hash = 0

3) ลูปทดลองจุดสิ้นสุดของ substring (สมมุติว่าจบที่ตำแหน่ง j) โดยค่อย ๆ ขยายออกจากจุดเริ่มต้น ระหว่างที่ขยาย เราจะนำค่าของตัวอักษรใหม่ที่เจอบวกเข้าไปใน hash

4) จดค่า hash ดังกล่าวลงใน container บางอย่าง (เช่น `vector`, `set`, `unordered_set`)

หลังจากนั้น เราจะทำการ hash ทุก substring ของ B ในทำนองเดียวกัน แต่ในขั้นตอนที่ 4 เราจะเปลี่ยนเป็นการเทียบค่า hash กับค่าที่เก็บไว้ใน container แทน หากพบว่ามีค่า hash ใน container นั่นแปลว่า substring ที่เราพิจารณาอยู่มีจำนวนตัวอักษร A-Z ตรงกับ substring หนึ่งของ **A** ดังนั้นเราจึงจดความยาวมากที่สุดที่เป็นไปได้ไว้เป็นคำตอบ

รวมแล้ววิธีนี้จะมี Time Complexity เป็น $O(N^2)$ หรือ $O(N^2 \log N)$ เมื่อ $N \approx |A| \approx |B|$ ขึ้นอยู่กับ container/data structure ที่เลือกใช้ในการเก็บข้อมูล hash

อนึ่ง บางคนอาจจะพยายามลด Time Complexity ด้วยการ binary search โดยอาศัยคุณสมบัตินี้

1) หากพบเจอ substring ความยาว L เป็น anagram กัน ทุกความยาว $l < L$ ก็ต้องมี substring ที่เป็น anagram กันด้วย ข้ามไปได้เลย ดังนั้นให้พิจารณาเฉพาะความยาว $l \geq L$ (เพราะเราต้องการ l มากสุด)

2) หากพบว่าไม่มี substring ความยาว L เป็น anagram กัน ทุกความยาว $l \geq L$ ก็จะไม่มีด้วย ดังนั้นเราต้องทดลองเฉพาะค่า $l < L$ เท่านั้น

แต่หากคิดดูดี ๆ จะพบว่าคุณสมบัตินี้**ไม่เป็นจริงเสมอไป** ยกตัวอย่างกรณี **A** = “ABCDE” และ **B** = “DBAEC” จะพบว่าหากเราทดลองความยาว L = 3 หรือ 4 เราจะไม่สามารถหา substring ที่เป็น anagram กันได้ แต่เมื่อทดลองความยาว L = 5 จะพบว่าเป็น anagram กันซึ่งขัดกับสมบัติที่ 2 ที่กล่าวมา

เหตุผลที่ผิดพลาด อาจเป็นเพราะความคุ้นชินกับโจทย์ประเภทนี้มากเกินไป ทำให้เมื่อเห็นโจทย์แล้วนำ binary search มาใช้โดยไม่ได้พิสูจน์สมบัติก่อน ควรสังเกต constraint ของโจทย์ให้ดี จึงจะได้ทราบว่า solution ของโจทย์ข้อนี้ต้องการเพียงแค่ $O(N^2)$ เท่านั้น ไม่ใช่ $O(N \log N)$

Solution Code

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const int N = 2010;
const ll PB = 98765431; // prime base factor
int n, m;
char A[N], B[N];
ll P[255]; // let's do 255, just because i'm too lazy to do -'A'
int main()
{
    // precompute value-to-add for each character
    P[0] = 1;
    for (int i = 1; i < 255; ++i)
        P[i] = P[i-1]*PB;
    // input
    scanf("%s %s", A, B);
    n = strlen(A);
    m = strlen(B);
    // hash all substrings of A
    vector<ll> hsh;
    for (int i = 0; i < n; ++i) {
        ll h = 0;
        for (int j = i; j < n; ++j) {
            h += P[A[j]];
            hsh.push_back(h);
        }
    }
    sort(hsh.begin(), hsh.end());
    // find longest substring of B that matches one of A's substrings
    int ans = 0;
    for (int i = 0; i < m; ++i) {
        ll h = 0;
        for (int j = i; j < m; ++j) {
            h += P[B[j]];
            if (binary_search(hsh.begin(), hsh.end(), h))
                ans = max(ans, j-i+1);
        }
    }
    printf("%d\n", ans);
    return 0;
}
```

โจทย์ข้อนี้เป็นโจทย์ Dynamic Programming ที่หลาย ๆ คนอาจจะคุ้นเคย นั่นคือเรื่อง Edit Distance แต่ความยากของโจทย์ข้อนี้คือการที่อนุญาตให้ข้ามช่วงยาวเท่าใดก็ได้ (Edit Distance ปกติอนุญาตให้ข้ามได้แค่ตัวเดียวเท่านั้น)

ก่อนอื่น เราจะนิยามให้ DP(i, j) มีค่าเท่ากับ “จำนวนครั้งผิดพลาดน้อยสุดที่เป็นไปได้ เมื่อข้อมูลดั้งเดิมมีเพียงแค่ $A_1, A_2, ..., A_i$ และนายจำท่องออกมาได้เป็น $B_1, B_2, ..., B_j$” โดยการนับจำนวนครั้งผิดพลาดในที่นี้ เราจะรวมถึงการข้ามช่วงสุดท้ายของ **A** ด้วย (หากเราต้องการข้ามช่วงสุดท้ายของ **A** ก็เพียงแค่เลือกค่า i ที่ i < **N** มา นั่นคือเลือกสนใจเฉพาะ prefix ของ **A**)

สังเกตว่าสำหรับทุก j ที่ 0 ≤ j ≤ **M** จะมี DP(0, j) เท่ากับ j เนื่องจากว่าข้อมูลดั้งเดิมว่างเปล่า แต่นายจำท่องข้อมูลเกินมา j ตัว (แน่นอนว่าเมื่อ j = 0 จะมี DP(0, j) = 0 เพราะว่านายจำไม่ได้ท่องเกินเลย)

สำหรับทุก i ที่ 1 ≤ i ≤ **N** จะมี DP(i, 0) = 1 เพราะข้อมูลดั้งเดิมมีความยาวมากถึง i ตัว นายจำไม่ท่องออกมาสักตัวถือว่าเป็นการข้ามช่วงไป 1 ช่วง (ข้ามทั้ง string)

ส่วนกรณีทั่วไป เมื่อ 1 ≤ i ≤ **N** และ 1 ≤ j ≤ **M** เราจะสนใจสิ่งที่เกิดขึ้นกับข้อมูลตัวท้ายเป็นหลัก (นั่นคือข้อมูลตัวที่ A_i และ B_j) โดยสิ่งที่เป็นไปได้มี 4 แบบคือ

1) ถ้าข้อมูลตัวสุดท้ายตรงกัน เราไม่จำเป็นต้องสนใจข้อมูลตัวสุดท้ายอีกแล้ว ดังนั้น หากนับด้วยวิธีนี้จะได้จำนวนครั้งผิดพลาดเป็น DP(i-1, j-1)

2) เมื่อข้อมูลตัวสุดท้ายไม่ตรงกัน อาจจะคิดได้ว่าเป็นการเปลี่ยนข้อมูล เพราะฉะนั้นให้นับว่าผิดพลาด 1 ครั้ง แล้วสนใจเฉพาะ DP(i-1, j-1) แทนเท่านั้น รวมแล้วจะได้จำนวนครั้งเป็น 1 + DP(i-1, j-1)

3) ข้อมูล Bj อาจจะเป็นข้อมูลที่ถูกแทรกเข้ามา ดังนั้นให้นับว่าผิดพลาด 1 ครั้ง แล้วสนใจเฉพาะ DP(i, j-1) แทน รวมแล้วจะได้จำนวนครั้งเป็น 1 + DP(i, j-1)

4) สำหรับทุก k ที่ 0 ≤ k < i อาจจะเห็นได้ว่าข้อมูล $A_{k+1}, A_{k+2}, ..., A_i$ เป็นข้อมูลที่ถูกข้ามไป ดังนั้นให้นับว่าผิดพลาด 1 ครั้งแล้วสนใจเฉพาะ DP(k, j) แทน รวมแล้วจะได้จำนวนครั้งเป็น 1 + DP(k, j)

เนื่องจากว่าเราต้องการจำนวนครั้งผิดพลาดน้อยที่สุดเท่าที่เป็นไปได้ การคำนวณค่าของ DP(i, j) เราจะคิดจากจำนวนครั้งที่น้อยที่สุดที่ได้จาก 4 กรณีดังกล่าว เมื่อคำนวณจำนวนครั้งผิดพลาดสำหรับทุก prefix ของ A และ B แล้วพบว่ามีการผิดพลาดไม่เกิน K ครั้ง ก็ให้จดคำตอบเป็นความยาวของ prefix ของ B เอาไว้

วิธีดังกล่าวมี state มากสุดถึง $O(N^2)$ state (เมื่อ $N \approx M$) และในแต่ละ state หากพิจารณาการผิดพลาดตามกรณีที่ 4 เราจำเป็นต้องลูปทดลองค่า k ที่ $0 \le k < i$ ถึง $O(N)$ ครั้ง รวมแล้ว Time Complexity จะเป็น $O(N^3)$ ซึ่งจะได้คะแนนมากสุดเพียง 50 คะแนนเท่านั้น

สังเกตว่า เมื่อเราพิจารณา state DP(i, j) เราต้องลูปค่า k ที่ $0 \le k < i$ และเมื่อเลื่อนมาพิจารณา state DP(i+1, j) เราต้องลูปค่า k ที่ $0 \le k < i+1$ - นั่นคือมีเพิ่มมาเพียงตัวเดียวเท่านั้น ดังนั้นเราสามารถนำค่า min ที่ได้จาก state DP(i, j) มาปรับใช้ต่อได้เลย ไม่จำเป็นต้องลูปใหม่ทั้งหมด วิธีนี้จะได้คะแนนมากสุดถึง 85 คะแนน เนื่องจากยังคงติดปัญหาเรื่อง Memory Limit ใน Subtask สุดท้าย

เราไม่จำเป็นต้องเก็บ DP ทุกแถวก็ได้ เก็บเพียงแค่สองแถวสุดท้ายแล้วนำมาใช้สลับกันไปเรื่อย ๆ จะทำให้ Memory ที่จำเป็นต้องใช้น้อยลง จึงได้คะแนนเต็ม 100 คะแนนในโจทย์ข้อนี้

Solution Code

```
#include <bits/stdc++.h>
using namespace std;
const int N = 5010, INF = 1e9;
int A[N], B[N], dp[2][N], mn[N];
int main()
{
    int n, m, k;
    scanf("%d%d%d", &n, &m, &k);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &A[i]);
    dp[0][0] = 0;
    for (int j = 1; j <= m; ++j) {
        scanf("%d", &B[j]);
        dp[0][j] = mn[j] = j;
    }
    int ans = 0;
    for (int i = 1; i <= n; ++i) {
        int x = i&1;
        dp[x][0] = 1;
        for (int j = 1; j <= m; ++j) {
            dp[x][j] = A[i] == B[j] ? dp[x^1][j-1] : INF;
            dp[x][j] = min(dp[x][j], dp[x][j-1]+1);
            dp[x][j] = min(dp[x][j], dp[x^1][j-1]+1);
            dp[x][j] = min(dp[x][j], mn[j]+1);
            if (dp[x][j] <= k) ans = max(ans, j);
            mn[j] = min(mn[j], dp[x][j]);
        }
    }
    printf("%d\n", ans);
    return 0;
}
```

ข้อสังเกตหนึ่งที่จะช่วยให้โจทย์ข้อนี้ง่ายขึ้นคือ วิธีที่ดีที่สุดจะไม่มีทางเลือกจับคู่โรงอาหาร a-c กับ b-d พร้อมกัน หรือ a-d กับ b-c โดยที่ $X_a < X_b < X_c < X_d$ แน่นอน เพราะเราสามารถทำให้คำตอบดีขึ้นได้โดยการเลือกคู่โรงอาหาร a-b และ c-d แทน (หากลองตำแหน่งโรงอาหาร a, b, c, d และเส้นเชื่อมระหว่างโรงอาหารที่เลือก จะเห็นได้ว่าวิธีแรกมีส่วนที่เส้นเชื่อมซ้อนกัน นั่นคือค่าความสุขถูกลดลงเนื่องจากผลรวมระยะทางมากเกินความจำเป็น วิธีที่สองจึงดีกว่า เพราะต้องเสียค่าความสุขจากระยะทางการเดินน้อยกว่า)

ดังนั้น สรุปได้ว่าวิธีที่ดีที่สุดจะเกิดจากการเลือก subset ของโรงอาหารมา (จำนวนโรงอาหารต้องเป็นจำนวนคู่) แล้วจับคู่โรงอาหารที่ 1-2, 3-4, 5-6, ... ไปเรื่อย ๆ ใน subset นั้น โดยเราจะทำการเรียงโรงอาหารจากตำแหน่งซ้ายไปขวาแล้วใช้ Dynamic Programming เพื่อหาวิธีการเลือกจับ subset ที่ดีที่สุด ดังนี้

นิยามให้ DP(i) = ค่าความสุขที่มากที่สุด เมื่อพิจารณาโรงอาหาร i ตำแหน่งแรกเท่านั้น เราจะเน้นพิจารณาสิ่งที่เกิดขึ้นกับโรงอาหารที่ i เป็นหลัก ซึ่งมีได้สองแบบ ดังนี้

1) ไม่นำโรงอาหารที่ i เข้าไปรวมใน subset ดังนั้นก็จะเหลือเพียงแค่ i-1 ตำแหน่งแรกให้พิจารณา คำตอบก็จะเป็น DP(i-1) เพียงเท่านั้น

2) จับคู่โรงอาหาร i กับ j (j < i) โดยต้องมีระยะห่างกันไม่เกิน K ($X_i - X_j \le K$) ทำให้ได้ค่าความสุขเท่ากับ $C_i + C_j - (X_i - X_j)$ แล้วนำมาบวกกับ DP(j-1) นั่นคือ พิจารณาจับคู่โรงอาหารที่ 1 ถึง j-1 ต่อ (สังเกตว่าเราจะไม่พิจารณาโรงอาหารที่ j+1 ถึง i-1 แล้ว ตามข้อสังเกตที่ระบุไว้ตอนแรก)

เราจะเลือกวิธีที่ได้ค่าความสุขสูงสุดเท่านั้น

พบว่าเราต้องพิจารณาทั้งหมด O(**N**) state คือ DP(1), DP(2), ..., DP(**N**) และสำหรับแต่ละ state เราต้องทดลองจับคู่โรงอาหาร ซึ่งอาจเลือกจับคู่ได้มากถึง O(**N**) ที่ ดังนั้น Time Complexity รวมจึงเป็น $O(N^2)$ ซึ่งวิธีนี้จะยังไม่ได้คะแนนเต็ม

สังเกตว่าส่วนที่เสียเวลา คือการลูปหาโรงอาหาร j ที่ทำให้ค่า $C_i + C_j - (X_i - X_j)$ + DP(j-1) มากที่สุด วิธีการแก้คือ แยกสมการส่วนที่เกี่ยวข้องกับ i และ j ออกจากกัน จะได้เป็น $(C_i - X_i) + (C_j + X_j$ + DP(j-1)) เมื่อเรา fix ค่า i เอาไว้จะได้ว่า $C_i - X_i$ เป็นค่าคงที่ที่เราต้องนำไปบวกอยู่แล้ว ส่วน $C_j + X_j$ + DP(j-1) คือส่วนที่เราต้องพยายามทำให้มีค่ามากที่สุด

เราจะเก็บค่า $C_j + X_j$ + DP(j-1) ทุกค่าที่เราจะพิจารณาไว้ใน Max Priority Queue โดยจะเก็บเป็นคู่อันดับพร้อมกับตำแหน่ง j ด้วย เมื่อเราต้องการคำนวณค่า DP(i) เราสามารถใช้ค่าสูงสุดจาก Priority Queue

ดังกล่าวได้เลย เมื่อคำนวณเสร็จแล้ว การที่จะคำนวณตำแหน่ง DP(i+1) ต่อไปได้นั้น เราจะมีทางเลือกเพิ่มเติม คือ สามารถเลือก $\mathbf{C_i}$ + $\mathbf{X_i}$ + DP(i-1) ได้ ก็ให้เพิ่มค่านี้เข้าไปใน Priority Queue ด้วย

เมื่อเลื่อนไปพิจารณา DP(i+1) แล้ว บางค่าใน Priority Queue อาจจะใช้ไม่ได้อีก เนื่องจากระยะห่างมากกว่าที่กำหนดไว้ ($\mathbf{X_{i+1}} - \mathbf{X_j} > \mathbf{K}$) แต่เราไม่สามารถลบค่าดังกล่าวออกจาก Priority Queue ทันทีได้ ดังนั้นเราจะปล่อยไว้ ทุกครั้งที่เราต้องการใช้ค่า Max จาก Priority Queue จึงต้องตรวจสอบเพิ่มเติมให้มั่นใจว่าค่าที่นำมาใช้ต้องเป็นค่าของโรงอาหารที่อยู่ห่างจากตำแหน่งปัจจุบันไม่เกิน **K** หากเกิน ให้นำค่าดังกล่าวออก พิจารณา Max ตัวต่อไปจนเจอตัวที่ระยะห่างไม่เกิน **K**

จากวิธีดังกล่าว จะทำให้ Time Complexity รวมลดลงเหลือ O(**N** log **N**)

นอกจากนี้ เราสามารถลด Time Complexity ส่วนนี้ลงเหลือ O(**N**) ได้โดยใช้ Double-ended Queue (Deque) เพื่อหา Sliding Window Maximum ซึ่งจะเก็บข้อมูลให้ตรงตามสมบัติดังต่อไปนี้

1) Deque จะเก็บค่าคล้ายกับ Priority Queue คือเก็บคู่อันดับ ($\mathbf{C_j}$+$\mathbf{X_j}$+DP(j-1), j)

2) ณ เวลาใด ๆ ข้อมูลใน Deque จะต้องเรียงจากมากไปน้อยเสมอ (จาก front ไปยัง back) และข้อมูลทุกตัวจะต้องเป็นข้อมูลจากโรงอาหารที่ห่างจากโรงอาหารที่ i เป็นระยะทางไม่เกิน **K** หน่วย

ดังนั้น หากเราต้องการค่ามากสุด สามารถหาได้จากตำแหน่ง front ของ Deque โดยเราสามารถรักษาคุณสมบัติของ Deque ดังกล่าวได้ดังนี้

1) เมื่อต้องการหาค่า max ต้องสังเกตตัวหน้าอยู่เสมอว่าระยะห่างเกิน K หน่วยหรือไม่ ถ้าเกินจะต้องนำออกเรื่อย ๆ จนกว่าจะระยะห่างน้อยกว่าหรือเท่ากับ K

2) เมื่อต้องการเพิ่มค่า ($\mathbf{C_j}$+$\mathbf{X_j}$+DP(j-1), j) สำหรับ j ใหม่เข้าไปที่ตำแหน่ง back จะต้องตรวจสอบว่า deque จะยังคงเรียงจากมากไปน้อยหรือไม่ หากเพิ่มเข้าไปแล้วไม่เรียง ต้องนำข้อมูลที่ตำแหน่ง back ออกก่อน จนกว่าจะพบว่าสามารถใส่ข้อมูลได้โดยข้อมูลจากมากไปน้อย

เนื่องจากข้อมูลแต่ละตัวจะถูกนำเข้าและออกจาก Deque ได้ไม่เกินอย่างละ 1 ครั้ง และข้อมูลมีเพียง **N** ตัว ดังนั้นรวมแล้วการทำงานในส่วนนี้ทั้งหมดจะมี Time Complexity เป็น O(**N**) เท่านั้น ถึงอย่างไรก็ตามโปรแกรมของเราทั้งโปรแกรมจะยังคงมี Time Complexity เป็น O(**N** log **N**) เนื่องจากเราต้องเรียงข้อมูลจากตำแหน่งซ้ายสุดไปยังตำแหน่งขวาสุด

สำหรับรายละเอียดการ Implement สามารถดูได้จาก Solution Code ซึ่งมีทั้งแบบ Priority Queue และแบบ Double-ended Queue

(Solution Code 1 อยู่ในหน้าถัดไป)

Solution Code 1

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

using pii = pair<int, int>;
using ll = long long;
using pli = pair<ll, int>;

const int N = 100010;
const ll INF = 1e18;

int n, k;
pii res[N]; // restaurant (x, c)
ll X[N], C[N], dp[N];

int main()
{
    scanf("%d%d", &n, &k);
    for (int i = 1; i <= n; ++i)
        scanf("%d%d", &res[i].first, &res[i].second);
    sort(res+1, res+n+1);
    for (int i = 1; i <= n; ++i)
        X[i] = res[i].first, C[i] = res[i].second;

    priority_queue<pli> Q; // max heap, sorted by value C[j]+X[j]+dp[j-1]
    for (int i = 1; i <= n; ++i) {
        while (!Q.empty() && X[i]-X[Q.top().second] > k)
            Q.pop();
        ll best = Q.empty() ? -INF : Q.top().first;
        dp[i] = max(dp[i-1], C[i]-X[i]+best);
        ll toadd = C[i]+X[i]+dp[i-1];
        Q.push(pli(toadd, i));
    }
    printf("%lld\n", dp[n]);

    return 0;
}
```

(Solution Code 2 อยู่ในหน้าถัดไป)

Solution Code 2

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

using pii = pair<int, int>;
using ll = long long;

const int N = 100010;
const ll INF = 1e18;

int n, k;
pii res[N]; // restaurant (x, c)
ll X[N], C[N], dp[N];

int main()
{
    scanf("%d%d", &n, &k);
    for (int i = 1; i <= n; ++i)
        scanf("%d%d", &res[i].first, &res[i].second);
    sort(res+1, res+n+1);
    for (int i = 1; i <= n; ++i)
        X[i] = res[i].first, C[i] = res[i].second;

    deque<int> Q;
    for (int i = 1; i <= n; ++i) {
        while (!Q.empty() && X[i]-X[Q.front()] > k)
            Q.pop_front();
        ll best = Q.empty() ? -INF : (C[Q.front()] + X[Q.front()] +
dp[Q.front()-1]);
        dp[i] = max(dp[i-1], C[i]-X[i]+best);
        ll toadd = C[i]+X[i]+dp[i-1];
        while (!Q.empty() && toadd >= C[Q.back()] + X[Q.back()] +
dp[Q.back()-1])
            Q.pop_back();
        Q.push_back(i);
    }
    printf("%lld\n", dp[n]);

    return 0;
}
```