September 11, 2025

New Zealand Informatics Competition 2025

Round Three

Editorial by Zalan V

Contents

Introduction

The solutions to this round of NZIC problems are discussed in detail below. In a few questions we may refer to the Big O complexity of a solution, e.g. $O(N)$. There is an explanation of Big O complexity at the end of this document.

Resources

Ever wondered what the error messages mean?

https://www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf

Read about how the server marking works:

https://www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf

Ever wondered why your submission scored zero?

https://www.nzoi.org.nz/nzic/resources/why-did-i-score-zero.pdf

See our list of other useful resources here:

https://www.nzoi.org.nz/nzic/resources

Trek 2

Problem authored by Anatol C
https://train.nzoi.org.nz/problems/1565

Subtask 1

In this subtask there is only a single mountain. So in any case, the highest and the lowest mountain we visit are the same mountain — so the difference in their heights is zero. Hence, the answer is always zero.

Python Solution

```
print(0)
```

Ruby Solution

```
p 0
```

Subtask 2

In this subtask there are exactly two mountains. It is always optimal to climb both mountains (since otherwise the max enjoyment would be zero) so the answer is the difference in height between the two mountains.

Python Solution

```
N = int(input())
h1 = int(input())
h2 = int(input())
print(abs(h1 - h2))
```

Subtask 3

In this subtask there are at most $1,000$ mountains. We can try every possible pair of mountains to the be tallest and shortest along our hike. The overall time complexity of this solution is $O(N^2)$.

Python Solution

```
N = int(input())
heights = [int(input()) for _ in range(N)]
max_enjoyment = 0
for i in range(N):
    for j in range(N):
        max_enjoyment = max(max_enjoyment, abs(heights[i] - heights[j]))
print(max_enjoyment)
```

Full Solution

We can observe that one optimal solution is to climb all the mountains. Thus, we simply need to find the tallest and shortest mountain. The overall time complexity of this solution is $O(N)$.

Python Solution

```
N = int(input())
heights = [int(input()) for _ in range(N)]
print(max(heights) - min(heights))
```

C++ Solution

The code for this solution can be found at the end of this document.

Mural

Problem authored by Zalan V
https://train.nzoi.org.nz/problems/1538

Subtask 1

In this subtask $N = 2$ so there are exactly two segments in the mural and two cans of spray paint. We can try using the first can for the first segment and the second can for the second segment or vice versa.

Python Solution

```
N = int(input())
mural = list(map(int, input().split()))
paint = list(map(int, input().split()))
min_inaccuracy = min(
    abs(mural[0] - paint[0]) + abs(mural[1] - paint[1]),
    abs(mural[0] - paint[1]) + abs(mural[1] - paint[0])
)
print(min_inaccuracy)
```

Subtask 2

In this subtask $N \leq 1,000$. We can observe that it is optimal to assign the smallest colour of paint to the smallest coloured segment, the second smallest to the second smallest, and so on. We can sort the colours in the mural and repeatedly remove the smallest coloured can of spray paint. The time complexity of this solution is $O(N^2)$.

Python Solution

The code for this solution can be found at the end of this document .

Full Solution

We can optimise the solution to Subtask 2 by sorting both the mural colours and the paint colours and pairing them up. The time complexity of this solution is $O(N)$.

Python Solution

```
N = int(input())
mural = list(map(int, input().split()))
paint = list(map(int, input().split()))
mural.sort()
paint.sort()

total_inaccuracy = 0
for i in range(N):
    total_inaccuracy += abs(mural[i] - paint[i])
print(total_inaccuracy)
```

Duck Tape

Problem authored by Thomas M
https://train.nzoi.org.nz/problems/1522

Subtask 1

In this subtask $N \leq 500$ and all positions are in the inclusive range 0 to 500. Thus, we can represent the box as an array of length 501. We can loop through all the positions that each piece of tape covers, and mark them as covered. Then, we can check if all positions between S and E are covered. The overall time complexity of this solution is $O(N \times L)$ where L is the total length of the box.

Python Solution

```
N, S, E = map(int, input().split())

covered = [False] * 501
for _ in range(N):
    s, e = map(int, input().split())
    for i in range(s, e + 1):
        covered[i] = True

if all(covered[S:E+1]):
    print(1)
else:
    print(0)
```

Subtask 2

In this subtask $N \leq 500$ and the positions are in the range -500 to 500. We can solve this subtask similarly to Subtask 1, but need to be able to handle negative positions, which we can do by adding 500 to all positions so that they become non-negative, and also increasing the size of our array so that it can handle the larger range of positions. The overall time complexity of this solution is $O(N \times L)$ where L is the total length of the box.

Python Solution

```
N, S, E = map(int, input().split())
S += 500
E += 500

covered = [False] * 1001
for _ in range(N):
    s, e = map(int, input().split())
    s += 500
    e += 500
    for i in range(s, e + 1):
        covered[i] = True

if all(covered[S:E+1]):
    print(1)
else:
    print(0)
```

Subtask 3

In this subtask $N \leq 500$. To solve this subtask we can use a technique called **coordinate compression** to extend the solution to Subtask 1. The idea of coordinate compression is to take some set of x integers

and map them into the range [0,x) while maintaining the order of the integers. For example, we can compress $\{17, 4, 6, 2\} \to \{3, 1, 2, 0\}$.

We consider that for each piece of tape the "important" points are $(s_i - 1), s_i, e_i, (e_i + 1)$, since all points between s_i and e_i get covered. Furthermore, S and E are important points. We can use coordinate compression on these important points to give us an array of length at most $4N + 2$. We can adapt the solution to Subtask 1 to use the compressed coordinates. The overall time complexity of this solution is $O(N^2)$.

Python Solution

```
N, S, E = map(int, input().split())
tapes = [tuple(map(int, input().split())) for _ in range(N)]

positions = set((S, E))
for s, e in tapes:
    positions.update((s - 1, s, e, e + 1))
positions = sorted(positions) # Convert set to sorted list

compressed_positions = {}
for i, pos in enumerate(positions):
    compressed_positions[pos] = i

covered = [False] * len(positions)
for s, e in tapes:
    for i in range(compressed_positions[s], compressed_positions[e] + 1):
        covered[i] = True

if all(covered[compressed_positions[S]:compressed_positions[E]+1]):
    print(1)
else:
    print(0)
```

Full Solution

For the full solution we can first sort all pairs of (s_i, e_i), and iterate over them while keeping track of the rightmost covered point, and stop when we encounter a gap. The overall time complexity of this solution is $O(N \log N)$.

Python Solution

```
N, S, E = map(int, input().split())
tapes = [tuple(map(int, input().split())) for _ in range(N)]
tapes.sort()

end = S - 1
for s, e in tapes:
    if end + 1 < s: break
    end = max(end, e)

if E <= end:
    print(1)
else:
    print(0)
```

C++ Solution

The code for this solution can be found at the end of this document .

Mischievous Gnomes

Problem authored by Zalan V
https://train.nzoi.org.nz/problems/1518

Subtask 1

In this subtask there are at most $1,000$ clearings and the clearings are arranged in a line from clearing 0 to clearing $N-1$. To solve this subtask we first need to calculate the distance of each clearing from clearing 0. We can calculate this using a **prefix sum**. Then, for each group of gnomes we can loop over every clearing, and if it is within distance r_i of clearing h_i then we mark the clearing as annoying.

We can observe that to reach clearing $N-1$ from clearing 0 we need to walk through every clearing, so the number of annoying clearings that we need to walk through is the total number of annoying clearings, and the distance we need to walk is simply the distance of clearing $N-1$ from clearing 0. The overall time complexity of this solution is $O(N^2)$.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 2

In this subtask there are no gnomes. Thus, we simply need to find the length of the shortest path from clearing 0 to clearing $N-1$. We can use **Dijkstra's Algorithm** to solve this subtask. The overall time complexity of this solution is $O(M \log M)$.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 3

In this subtask none of the gnomes can leave their home clearing. Thus, the annoying clearings will simply be the home clearings. We will extend the Subtask 2 solution to solve this subtask. As it turns out, we can use pairs of (number of annoying clearings, distance) as distances in Dijkstra's algorithm. We compare two pairs by the number of annoying clearings, with ties broken by distance. The overall time complexity of this solution is $O(M \log M)$.

Python Solution

The code for this solution can be found at the end of this document.

Full Solution

For the full solution, we can use a multi-source Dijkstra traversal where we maximise the remaining distance that we can travel from each clearing to find all the annoying clearings in $O(M \log M)$. We can then use the Dijkstra traversal from the Subtask 3 solution to calculate the minimum number of annoying clearings we must walk through, and the minimum distance. The overall time complexity of this solution is $O(M \log M)$.

C++ Solution

The code for this solution can be found at the end of this document.

Sleep Gardening

Problem authored by Zalan V

https://train.nzoi.org.nz/problems/1363

Subtask 1

In this subtask $M = 1$ and $K = 1$ so there is exactly one tree, and we can perform the sapling planting operation a single time. There are three possible ways to apply the operation: plant saplings to the left of the tree, plant saplings to the right of the tree, or remove the tree and plant saplings in all segments.

Python Solution

```
N, M, K = map(int, input().split())
p, h = map(int, input().split())
print(max(h + (p - 1), h + (N - p), N))
```

Subtask 2

In this subtask $N \leq 100$ and $K = 1$. We can represent the garden as an array, and calculate the result of applying the operation to every single subarray. There are $O(N^2)$ subarrays, and we can calculate the heights of the tree in each given subarray in $O(N)$. This makes the overall time complexity of this solution $O(N^3)$.

Python Solution

```
N, M, K = map(int, input().split())
heights = [0] * N
for _ in range(M):
    p, h = map(int, input().split())
    heights[p - 1] = h

ans = sum(heights)
for i in range(N):
    for j in range(i, N):
        ans = max(ans, sum(heights[:i]) + (j - i + 1) + sum(heights[j + 1:]))
print(ans)
```

Subtask 3

In this subtask $K = 1$. We can think of the total achievable foliage after applying the operation as the sum of the initial foliage and the foliage gained from applying the operation. We can trivially calculate the initial foliage, so it remains to calculate the maximum possible foliage gain. We can observe that the foliage gained from an empty section of the garden is simply the length of the section, and the foliage gained from removing a tree is $1 - h_i$.

We can now reframe the problem as finding the maximum possible subarray sum over these values. We can use **Kadane's Algorithm** to find the maximum possible subarray sum over this new array. The overall time complexity of this solution is $O(N)$.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 4

In this subtask $K \leq 50$. We can use a dynamic programming approach to solve this subtask where our DP state will be (index of tree, operations remaining). Let V be the array we get from transforming each empty section of the garden into its length, and each tree into the value $1 - h_i$.

We define the base cases:

$$\mathrm{dp}_{\mathrm{inc}}(0, j) = \max\begin{cases} V_0 \text{ if } j < K \\ 0 \text{ if } j = K \end{cases}$$

$$\mathrm{dp}_{\mathrm{exc}}(0, j) = 0$$

Then, the recurrences:

$$\mathrm{dp}_{\mathrm{inc}}(i, j) = V_i + \max\begin{cases} \mathrm{dp}_{\mathrm{inc}}(i-1, j) \\ \mathrm{dp}_{\mathrm{exc}}(i-1, j) \end{cases}$$

$$\mathrm{dp}_{\mathrm{exc}}(i, j) = \max\begin{cases} \mathrm{dp}_{\mathrm{inc}}(i-1, j-1) \\ \mathrm{dp}_{\mathrm{exc}}(i-1, j) \end{cases}$$

Our answer will then be $\mathrm{dp}(N-1, K)$ plus the sum of all tree heights. The overall time complexity of this solution is $O(N \times K)$. It should be noted that we can easily apply the **sliding window** DP optimisation technique to reduce the memory usage of this algorithm from $O(N \times K)$ to $O(N)$, although a solution without this optimisation may still pass depending on the language and implementation.

Python Solution

The code for this solution can be found at the end of this document .

Full Solution

We will continue to use the idea from Subtask 4 of transforming the garden into the array V of the gain from applying an operation in each section. The problem that we are solving is formally known as the K-disjoint subarray sum problem, and there are many different algorithms that can be used to solve this problem. We will discuss two approaches here: a greedy approach, and a dynamic programming approach.

Approach 1 (Greedy)

We first consider that we can merge together adjacent negative values in V since it will never be optimal to replace a single tree out of a group of adjacent trees. We can also observe that it will never be optimal to replace a tree at the edge of the garden with a sapling. Thus, we can transform V into an alternating sequence S of positive and negative elements, which starts and ends with a positive element.

Since S both starts and ends with a positive element, its length must be odd, so we can write $|S| = 2x + 1$ for some integer x. We can observe that the sum of the positive elements of S represent the maximum sum of $x + 1$ subarrays, that is, the maximum achievable sum from applying the operation $x + 1$ times.

We can think of S as being partitioned into elements that we take and don't take. The elements that we take are the positive ones, and the elements that we don't take are the negative ones. So we can think of S as (take, don't take, ..., take). We will now consider how we can transform S from a solution for $x + 1$ subarrays into a solution for x subarrays. There are three possible cases: we can either merge three consecutive elements together, we can delete the first two elements, or delete the last two elements.

It can be shown that it is optimal to find the element with the smallest absolute value and either merge it with its two neighbours, or to remove it and its neighbour if it is one of the endpoints. We repeat this until the length of S is $2 \times K - 1$.

We can use a set of elements sorted by index, and a set of elements sorted by absolute value to perform these operations efficiently in $O(\log M)$. The overall time complexity of this solution is then $O(M \log M)$.

C++ Solution

The code for this solution can be found at the end of this document.

Approach 2 (Dynamic Programming)

Another approach that we can use to solve the problem is a DP optimisation technique known as **Lagrangian Relaxation** (sometimes known as Aliens Trick). You can read more about this technique here.

Python Solution

```
N, M, K = map(int, input().split())
tot, last = 0, 0
val = []
for i in range(M):
    p, h = map(int, input().split())
    tot += h
    val.append(p - last - 1)
    val.append(1 - h)
    last = p
val.append(N - last)

def solve(cost):
    inc, exc = (val[0] - cost, 1), (0, 0)
    for i in range(1, len(val)):
        inc, exc = max((exc[0] + val[i] - cost, exc[1] + 1), (inc[0] + val[i],
inc[1])), max(exc, inc)
    return max(inc, exc)

low, upp = 0, 10**15
while low < upp:
    mid = (low + upp + 1) // 2
    res = solve(mid)
    if res[1] >= K: low = mid
    else: upp = mid - 1
print(solve(low)[0] + K * low + tot)
```

C++ Solution

The code for this solution can be found at the end of this document.

Big O Complexity

Computer scientists like to compare programs using something called Big O notation. This works by choosing a parameter, usually one of the inputs, and seeing what happens as this parameter increases in value. For example, let's say we have a list N items long. We often call the measured parameter N. For example, a list of length N.

In contests, problems are often designed with time or memory constraints to make you think of a more efficient algorithm. You can estimate this based on the problem's constraints. It's often reasonable to assume a computer can perform around 100 million (100,000,000) operations per second. For example, if the problem specifies a time limit of 1 second and an input of N as large as 100,000, then you know that an $O(N^2)$ algorithm might be too slow for large N since $100{,}000^2 = 10{,}000{,}000{,}000$, or 10 billion operations.

Time Complexity

The time taken by a program can be estimated by the number of processor operations. For example, an addition $a + b$ or a comparison $a < b$ is one operation.

$O(1)$ time means that the number of operations a computer performs does not increase as N increases (i.e. does not depend on N). For example, say you have a program containing a list of N items and want to access the item at the i-th index. Usually, the computer will simply access the corresponding location in memory. There might be a few calculations to work out which location in memory the entry i corresponds to, but these will take the same amount of computation regardless of N. Note that time complexity does not account for constant factors. For example, if we doubled the number of calculations used to get each item in the list, the time complexity is still $O(1)$ because it is the same for all list lengths. You can't get a better algorithmic complexity than constant time.

$O(\log N)$ time suggests the program takes a constant number of extra operations every time N doubles in size. For example, finding a number in a sorted list using binary search might take 3 operations when $N = 8$, but it will only take one extra operation if we double N to 16. As far as efficiency goes, this is pretty good, since N generally has to get very, very large before a computer starts to struggle.

$O(N)$ time means you have an algorithm where the number of operations is directly proportional to N. For example, a maximum finding algorithm `max()` will need to compare against every item in a list of length N to confirm you have indeed found the maximum. Usually, if you have one loop that iterates N times your algorithm is $O(N)$.

$O(N^2)$ time means the number of operations is proportional to N^2 . For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of N items, each item has to check against the remaining $N - 1$ items. In total, $N(N - 1)$ checks are done. This expands to $N^2 - N$. For Big O, we always take the most significant term as the dominating factor, which gives $O(N^2)$. This is generally not great for large values of N, which can take a very long time to compute. As a general rule of thumb in contests, $O(N^2)$ algorithms are only useful for input sizes of $N \lesssim 10{,}000$. Usually, if you have a nested loop in your program (loop inside a loop) then your solution is $O(N^2)$ if both these loops run about N times.

Additional Example Solutions

Trek 2 Full Solution (C++)

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

int main() {
    int N;
    cin >> N;

    vector<int> heights(N);
    for (int i = 0; i < N; ++i) cin >> heights[i];

    auto ans = minmax_element(heights.begin(), heights.end());
    cout << *ans.second - *ans.first << "\n";

    return 0;
}
```

Mural Subtask 2 Solution (Python)

```python
N = int(input())
mural = list(map(int, input().split()))
paint = list(map(int, input().split()))
mural.sort()

total_inaccuracy = 0
for colour in mural:
    paint_colour = min(paint)
    paint.remove(paint_colour)
    total_inaccuracy += abs(colour - paint_colour)
print(total_inaccuracy)
```

Duck Tape Full Solution (C++)

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

int main() {
    int N, S, E;
    cin >> N >> S >> E;

    vector<pair<int, int>> tapes(N);
    for (int i = 0; i < N; ++i) {
        cin >> tapes[i].first >> tapes[i].second;
    }
    sort(tapes.begin(), tapes.end());

    int end = S - 1;
    for (auto [s, e] : tapes) {
        if (end + 1 < s) break;
        end = max(end, e);
    }

    if (E <= end) cout << "1\n";
    else cout << "0\n";
```

```
    return 0;
}
```

Mischievous Gnomes Subtask 1 Solution (Python)

```
N, M, G = map(int, input().split())
distances = [0] * N
for _ in range(M):
    a, b, c = map(int, input().split())
    distances[b] = distances[a] + c

annoying = [False] * N
for _ in range(G):
    h, r = map(int, input().split())
    for i in range(N):
        if distances[h] - r <= distances[i] <= distances[h] + r:
        annoying[i] = True

print(sum(annoying))
print(distances[N - 1])
```

Mischievous Gnomes Subtask 2 Solution (Python)

```
import heapq

N, M, G = map(int, input().split())
adj = [[] for _ in range(N)]
distances = [0] * N
for _ in range(M):
    a, b, c = map(int, input().split())
    adj[a].append((b, c))
    adj[b].append((a, c))

best_distances = [10**10] * N
best_distances[0] = 0
distance_queue = [(0, 0)]
while distance_queue:
    dist, node = heapq.heappop(distance_queue)
    if node == N - 1:
        print(0)
        print(dist)
        break
    if best_distances[node] < dist: continue
    for child, child_dist in adj[node]:
        new_child_dist = dist + child_dist
        if new_child_dist < best_distances[child]:
            best_distances[child] = new_child_dist
            heapq.heappush(distance_queue, (new_child_dist, child))
```

Mischievous Gnomes Subtask 3 Solution (Python)

```
import heapq

N, M, G = map(int, input().split())
adj = [[] for _ in range(N)]
distances = [0] * N
for _ in range(M):
    a, b, c = map(int, input().split())
    adj[a].append((b, c))
```

```
    adj[b].append((a, c))

annoying = [False] * N
for _ in range(G):
    h, r = map(int, input().split())
    annoying[h] = True

best_distances = [(10**10, 10**10)] * N
best_distances[0] = (int(annoying[0]), 0)
distance_queue = [((int(annoying[0]), 0), 0)]
while distance_queue:
    dist, node = heapq.heappop(distance_queue)
    if node == N - 1:
        print(dist[0])
        print(dist[1])
        break
    if best_distances[node] < dist: continue
    for child, child_dist in adj[node]:
        new_child_dist = (dist[0] + annoying[child], dist[1] + child_dist)
        if new_child_dist < best_distances[child]:
            best_distances[child] = new_child_dist
            heapq.heappush(distance_queue, (new_child_dist, child))
```

Mischievous Gnomes Full Solution (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> ii;
typedef vector<int> vi;

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

    int N, M, G;
    cin >> N >> M >> G;

    vector<vector<ii>> adj(N);
    for (int i = 0; i < M; ++i) {
        int a, b, c;
        cin >> a >> b >> c;
        adj[a].push_back({b, c});
        adj[b].push_back({a, c});
    }

    vi remainingDist(N, -1);
    priority_queue<ii> que1;
    vector<bool> annoying(N);
    for (int i = 0; i < G; ++i) {
        int h, r;
        cin >> h >> r;
        remainingDist[h] = r;
        que1.push({r, h});
    }
    while (que1.size()) {
        auto [r, n] = que1.top();
        que1.pop();
```

```
        if (remainingDist[n] < r) continue;
        annoying[n] = true;
        for (auto [c, d] : adj[n]) {
            int nr = r - d;
            if (nr > remainingDist[c]) {
                remainingDist[c] = nr;
                que1.push({nr, c});
            }
        }
    }

    vector<ii> dist(N, {INT_MAX / 2, 0});
    priority_queue<pair<ii, int>, vector<pair<ii, int>>, greater<>> que2;
    que2.push({{0, 0}, 0});
    while (que2.size()) {
        auto [d, n] = que2.top();
        que2.pop();
        if (dist[n] < d) continue;
        d.first += annoying[n];
        if (n == N - 1) {
            cout << d.first << endl;
            cout << d.second << endl;
            break;
        }
        for (auto [c, cd] : adj[n]) {
            ii nd = {d.first, d.second + cd};
            if (nd < dist[c]) {
                dist[c] = nd;
                que2.push({nd, c});
            }
        }
    }
    return 0;
}
```

Sleep Gardening Subtask 3 Solution (Python)

```
N, M, K = map(int, input().split())

total_heights = 0
sections = []
last_pos = 0
for i in range(M):
    p, h = map(int, input().split())
    total_heights += h
    sections.append(p - last_pos - 1)
    sections.append(1 - h)
    last_pos = p
sections.append(N - last_pos)

min_prefix = 0
prefix_sum = 0
max_subarray = 0
for x in sections:
    prefix_sum += x
    min_prefix = min(min_prefix, prefix_sum)
```

```
    max_subarray = max(max_subarray, prefix_sum - min_prefix)
print(total_heights + max_subarray)
```

Sleep Gardening Subtask 4 Solution (C++)

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

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

    int N, M, K;
    cin >> N >> M >> K;

    vector<ll> sections;
    ll last_pos = 0;
    ll total_heights = 0;
    for (int i = 0; i < M; ++i) {
        int a, b;
        cin >> a >> b;
        total_heights += b;
        sections.push_back(a - last_pos - 1);
        sections.push_back(1 - b);
        last_pos = a;
    }
    sections.push_back(N - last_pos);

    int L = sections.size();
    vector<vector<ll>> dp_inc(2, vector<ll>(K + 1)), dp_exc(2, vector<ll>(K + 1));
    for (int i = 0; i < K; ++i) dp_inc[0][i] = sections[0];
    for (int i = 1; i < L; ++i) {
        int idx = i & 1;
        for (int j = 1; j <= K; ++j) dp_exc[idx][j] = max(dp_exc[idx ^ 1][j],
dp_inc[idx ^ 1][j - 1]);
        for (int j = 0; j < K; ++j) dp_inc[idx][j] = sections[i] + max(dp_inc[idx ^
1][j], dp_exc[idx ^ 1][j]);
    }
    ll ans = 0;
    for (int i = 0; i < K + 1; ++i) ans = max(ans, max(dp_inc[(L - 1) & 1][i],
dp_exc[(L - 1) & 1][i]));

    cout << ans + total_heights << endl;

    return 0;
}
```

Sleep Gardening Full Solution 1 (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<ll, ll> pll;

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

```cpp
ll N, M, K;
cin >> N >> M >> K;

vector<ll> val;
ll last_pos = 0, total_heights = 0;
for (int i = 0; i < M; ++i) {
    ll p, h;
    cin >> p >> h;
    total_heights += h;
    if (last_pos < p - 1) {
        val.push_back(p - last_pos - 1);
        val.push_back(1 - h);
    }
    else if (val.size()) val.back() += 1 - h;
    last_pos = p;
}
if (last_pos < N) val.push_back(N - last_pos);
else if (val.size()) val.pop_back();

if (val.empty()) {
    cout << total_heights << endl;
    return 0;
}

// Use a custom comparator to sort the {position, value} pairs by absolute value
struct cmp {
    bool operator() (pll a, pll b) const {
        if (abs(a.second) != abs(b.second)) return abs(a.second) < abs(b.second);
        return a.first < b.first;
    };
};
set<pll, cmp> by_abs;
set<pll> by_idx;

for (int i = 0; i < val.size(); ++i) {
    by_idx.insert({i, val[i]});
    by_abs.insert({i, val[i]});
}

while (by_abs.size() / 2 >= K) {
    auto a = *by_abs.begin();
    auto it = by_idx.find(a);

    if (it == by_idx.begin()) {
        by_abs.erase(*next(it));
        by_idx.erase(next(it));
        by_abs.erase(*it);
        by_idx.erase(it);
    }
    else if (it == prev(by_idx.end())) {
        by_abs.erase(*prev(it));
        by_idx.erase(prev(it));
        by_abs.erase(*it);
        by_idx.erase(it);
    }
```

```
        else {
            a.second += next(it)->second + prev(it)->second;

            by_abs.erase(*next(it));
            by_idx.erase(next(it));
            by_abs.erase(*prev(it));
            by_idx.erase(prev(it));
            by_abs.erase(*it);
            by_idx.erase(it);

            by_abs.insert(a);
            by_idx.insert(a);
        }
    }

    ll ans = 0;
    for (auto [i, v] : by_idx) {
        if (v > 0) ans += v;
    }
    cout << total_heights + ans << "\n";

    return 0;
}
```

Sleep Gardening Full Solution 2 (C++)

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<ll, ll> pll;

pll solve_relaxed(vector<ll>& val, ll cost) {
    pll inc{val[0] - cost, 1}, exc{};
    for (int i = 1; i < val.size(); ++i) {
        pll ninc = max(
            pll{exc.first + val[i] - cost, exc.second + 1},
            pll{inc.first + val[i], inc.second}
        );
        exc = max(exc, inc);
        inc = ninc;
    }
    return max(inc, exc);
}

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

    int N, M, K;
    cin >> N >> M >> K;

    vector<ll> val;
    ll last_pos = 0;
    ll total_heights = 0;
    for (int i = 0; i < M; ++i) {
        ll a, b;
        cin >> a >> b;
```

```
        total_heights += b;
        val.push_back(a - last_pos - 1);
        val.push_back(1 - b);
        last_pos = a;
    }
    if (last_pos < N) val.push_back(N - last_pos);

    ll low = 0;
    ll upp = 1e15;
    while (low < upp) {
        ll mid = (low + upp + 1) / 2;
        auto res = solve_relaxed(val, mid);
        if (res.second >= K) low = mid;
        else upp = mid - 1;
    }

    auto ans = solve_relaxed(val, low);
    cout << ans.first + K * low + total_heights << endl;

    return 0;
}
```