**C - Not All Covered -** [**https://atcoder.jp/contests/abc408/tasks/abc408\_c?lang=en**](https://atcoder.jp/contests/abc408/tasks/abc408_c?lang=en)

**🧩 Problem Statement**

We have:

* **N castle walls**, numbered from 1 to N.
* **M turrets**, each turret i guards a continuous range of walls [Li, Ri].

We can **destroy turrets**.  
If we destroy turret i, then walls [Li, Ri] are **no longer guarded by it**.

Your task:

Find the **minimum number of turrets to destroy** so that **at least one wall** is **not guarded by any turret**.

**Example 1**

**Input**

10 4

1 6

4 5

5 10

7 10

**Visualization**

| **Turret** | **Guards Walls** |
| --- | --- |
| 1 | [1–6] |
| 2 | [4–5] |
| 3 | [5–10] |
| 4 | [7–10] |

If we check:

* Wall 1 → guarded by turret 1
* Wall 3 → only turret 1 guards it  
  So if we **destroy turret 1**, wall 3 becomes **unguarded**.

✅ Minimum turrets to destroy = **1**

**Example 2**

5 2

1 2

3 4

Walls guarded:

* 1–2 by turret 1
* 3–4 by turret 2
* Wall 5 is **not covered** by any turret.

✅ Already unguarded → answer = **0**

**Example 3**

5 10

2 5

1 5

1 2

2 4

2 2

5 5

2 4

1 2

2 2

2 3

Here, **all walls 1–5** are guarded by **multiple turrets**.

We need to **remove enough turrets** so that **at least one wall** becomes unguarded.

The answer is 3.

**🧠 How to Think About It**

We want **at least one position (wall)** that becomes unguarded.

That means we must find a wall that is **covered by the fewest number of turrets**, because removing those is the easiest way to make it unguarded.

So the **key idea** is:

Find the **minimum number of overlapping turrets** covering any single wall.

That’s exactly how many turrets must be destroyed to make that wall unguarded.

**⚙️ Step-by-Step Solution**

**Step 1: Use a Difference Array to Count Coverage**

We can efficiently find **how many turrets cover each wall** using the **prefix sum technique** (like in “Karen and Coffee”).

For each turret [Li, Ri]:

diff[Li] += 1

diff[Ri + 1] -= 1

Then, take prefix sums:

cover[i] = cover[i - 1] + diff[i]

This gives the number of turrets guarding wall i.

**Step 2: Check if Some Wall is Already Unguarded**

If any cover[i] == 0, answer = 0.

Otherwise, we must destroy enough turrets to make **some** cover[i] drop to zero.

**Step 3: The Minimum Number to Destroy**

For any wall i, cover[i] means:

* That wall is guarded by cover[i] turrets.
* We must destroy all those turrets to make it unguarded.

So, the **minimum number of turrets to destroy** is:

min\_cover = min(cover[i]) over all i from 1 to N

That’s the answer.

**✅ Final C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

const int MAXN = 1e6 + 5;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int N, M;

cin >> N >> M;

vector<int> diff(N + 2, 0);

for (int i = 0; i < M; i++) {

int L, R;

cin >> L >> R;

diff[L] += 1;

if (R + 1 <= N) diff[R + 1] -= 1;

}

vector<int> cover(N + 2, 0);

for (int i = 1; i <= N; i++) {

cover[i] = cover[i - 1] + diff[i];

}

int min\_cover = INT\_MAX;

for (int i = 1; i <= N; i++) {

min\_cover = min(min\_cover, cover[i]);

}

// If some wall is already unguarded

if (min\_cover == 0) cout << 0 << "\n";

else cout << min\_cover << "\n";

return 0;

}

**🧮 Complexity Analysis**

| **Step** | **Time** | **Space** |
| --- | --- | --- |
| Reading input + updating diff | O(M) | O(N) |
| Building prefix sum | O(N) | O(N) |
| Finding min coverage | O(N) | O(1) |
| **Total** | **O(N + M)** | **O(N)** |

✅ Fits within the limits easily.

**🔍 Summary**

| **Concept** | **Explanation** |
| --- | --- |
| What we want | At least one wall not guarded |
| Key observation | Minimum number of turrets covering any wall = answer |
| Technique used | Prefix sum (difference array) |
| Edge case | Already unguarded → answer = 0 |
| Time complexity | O(N + M) |
| Space complexity | O(N) |

**B. Karen and Coffee -** [**https://codeforces.com/problemset/problem/816/B**](https://codeforces.com/problemset/problem/816/B)

**🧩 Problem Understanding**

Karen has **n recipes**, and each recipe gives a **temperature range** [li, ri] where coffee tastes good.

A temperature T is **admissible** if **at least k recipes** recommend brewing coffee at that temperature (i.e., T lies inside at least k of the [li, ri] intervals).

Karen then asks **q queries**, each query gives a range [a, b], and you must find **how many admissible integer temperatures** exist within [a, b].

**Example Intuition**

Input:

3 2 4

91 94

92 97

97 99

92 94

93 97

95 96

90 100

* Recipe 1 → [91, 94]
* Recipe 2 → [92, 97]
* Recipe 3 → [97, 99]
* k = 2 → temperature must appear in **at least 2 intervals**

Let’s find how many recipes recommend each temperature:

| **Temperature** | **Recipes Count** | **Admissible?** |
| --- | --- | --- |
| 91 | 1 | No |
| 92 | 2 | ✅ |
| 93 | 2 | ✅ |
| 94 | 2 | ✅ |
| 95 | 1 | No |
| 96 | 1 | No |
| 97 | 2 | ✅ |
| 98 | 1 | No |
| 99 | 1 | No |

→ So admissible temperatures are {92, 93, 94, 97}

Now queries:

1. [92,94] → 3 admissible
2. [93,97] → 3 admissible
3. [95,96] → 0 admissible
4. [90,100] → 4 admissible ✅

**🧠 How to Think About the Problem**

We have up to **200,000 recipes** and **200,000 queries**, and the temperature range can go up to **200,000** — meaning a brute force solution (checking each temperature for each recipe) would be **O(n × 200000)** → far too slow.

We need an **O(n + max\_temp + q)** solution.

**⚙️ Efficient Approach — Prefix Sum + Difference Array**

**Step 1: Use a Difference Array**

We can count how many intervals cover each temperature efficiently using a **difference array technique**.

* For each interval [l, r]:
* diff[l] += 1
* diff[r + 1] -= 1
* Then, take a prefix sum of this array → gives us the **number of recipes recommending each temperature**.

**Step 2: Mark Admissible Temperatures**

Create another array ok[temp] = 1 if count[temp] >= k, otherwise 0.

**Step 3: Prefix Sum for Queries**

Now, make a **prefix sum** array of ok[] called pref[], where:

pref[i] = pref[i-1] + ok[i]

Then for any query [a, b]:

answer = pref[b] - pref[a-1]

✅ Constant-time per query.

**🧮 Complexity Analysis**

| **Operation** | **Complexity** |
| --- | --- |
| Building diff array | O(n) |
| Prefix sum to get counts | O(max\_temp) |
| Building admissible prefix | O(max\_temp) |
| Answering q queries | O(q) |
| **Total** | **O(n + q + max\_temp)** |
| **Memory** | O(max\_temp) ≈ 200k |

Efficient and fits limits easily.

**✅ C++ Implementation**

#include <bits/stdc++.h>

using namespace std;

const int MAX = 200000 + 5;

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

int n, k, q;

cin >> n >> k >> q;

vector<int> diff(MAX, 0);

// Step 1: Build difference array

for (int i = 0; i < n; i++) {

int l, r;

cin >> l >> r;

diff[l] += 1;

if (r + 1 < MAX) diff[r + 1] -= 1;

}

// Step 2: Build prefix sum to get count per temperature

vector<int> count(MAX, 0);

count[0] = diff[0];

for (int i = 1; i < MAX; i++)

count[i] = count[i - 1] + diff[i];

// Step 3: Mark admissible temperatures (count >= k)

vector<int> ok(MAX, 0);

for (int i = 1; i < MAX; i++) {

ok[i] = (count[i] >= k ? 1 : 0);

}

// Step 4: Build prefix sum for admissible counts

vector<int> pref(MAX, 0);

for (int i = 1; i < MAX; i++)

pref[i] = pref[i - 1] + ok[i];

// Step 5: Answer queries in O(1)

while (q--) {

int a, b;

cin >> a >> b;

cout << pref[b] - pref[a - 1] << "\n";

}

return 0;

}

**🧾 Summary**

| **Step** | **Description** |
| --- | --- |
| 1️⃣ | Use difference array to track how many intervals cover each temperature |
| 2️⃣ | Prefix sum → count of recipes per temperature |
| 3️⃣ | Mark admissible temperatures (count ≥ k) |
| 4️⃣ | Prefix sum again → answer queries instantly |
| ✅ | Time: O(n + q + 200000), Memory: O(200000) |