**D. Odd Queries -** [**https://codeforces.com/problemset/problem/1807/D**](https://codeforces.com/problemset/problem/1807/D)

**C. Little Girl and Maximum Sum-** [**https://codeforces.com/problemset/problem/276/C**](https://codeforces.com/problemset/problem/276/C)

**🧩 Problem Statement**

You are given:

* An array a of n integers.
* q queries, each query gives a **range [l, r]** (1-indexed).

For each position i (1 ≤ i ≤ n), count **how many queries cover index i** — that is, in how many of the given ranges [l, r] does the index i appear.

After counting this frequency for each position, you must:

* Rearrange the array a in any order you want.
* Rearrange the frequency array (number of times each index is covered) in any order you want.

Your goal is to **maximize the sum**:

[  
\text{Sum} = \sum\_{i=1}^{n} a\_i \times \text{freq}\_i  
]

and print that maximum possible sum.

**🧠 How to Think (Step-by-Step Intuition)**

Let’s break down the problem logically:

**Step 1: What affects the sum?**

The sum is formed by multiplying a[i] with how many times index i is covered by queries (freq[i]).

So, if some positions are covered more often (higher freq[i]), you’d ideally want **larger values of a[i]** to be multiplied with them — because that maximizes the product.

**Step 2: How to find freq[i] efficiently?**

A naive approach would loop through each query and increment all indices from l to r.  
That would be **O(n \* q)** — too slow when both are large.

👉 Instead, use a **difference array technique**:

* For each query [l, r], do:
* diff[l] += 1
* diff[r+1] -= 1
* Then, compute prefix sums to get how many queries cover each index.

This runs in **O(n + q)** — very efficient.

**Step 3: How to maximize the total sum?**

Once we know the frequency of each position (freq[i]), we can sort both arrays:

* Sort a in **ascending order**.
* Sort freq in **ascending order**.

Then multiply a[i] \* freq[i] for all i.

Why does this work?  
Because the **Rearrangement Inequality** tells us that:

The sum of products is maximized when both sequences are sorted in the same order.

**💻 Final Solution (C++)**

#include <bits/stdc++.h>

using namespace std;

using ll = long long;

int main() {

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

cin.tie(nullptr);

int n, q;

cin >> n >> q;

vector<ll> a(n);

for (int i = 0; i < n; ++i) cin >> a[i];

// Step 1: Build difference array

vector<ll> diff(n + 2, 0);

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

int l, r;

cin >> l >> r; // 1-based

diff[l] += 1;

diff[r + 1] -= 1;

}

// Step 2: Build frequency array

vector<ll> freq(n);

ll cur = 0;

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

cur += diff[i];

freq[i - 1] = cur; // store 0-based

}

// Step 3: Sort and compute maximum sum

sort(a.begin(), a.end());

sort(freq.begin(), freq.end());

ll ans = 0;

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

ans += a[i] \* freq[i];

cout << ans << '\n';

return 0;

}

**⏱️ Complexity Analysis**

| **Operation** | **Complexity** |
| --- | --- |
| Building difference array | O(q) |
| Prefix sum for freq | O(n) |
| Sorting arrays a and freq | O(n log n) |
| Final summation | O(n) |
| **Total Time Complexity** | **O(n log n + q)** |
| **Space Complexity** | **O(n)** |

**✅ Summary**

| **Concept** | **Description** |
| --- | --- |
| **Problem Type** | Range frequency counting + maximizing sum |
| **Key Idea** | Larger values should multiply with higher frequencies |
| **Techniques Used** | Difference array, prefix sum, sorting |
| **Core Formula** | Sort both arrays → multiply element-wise → sum up |
| **Efficiency** | O(n log n + q), works for large constraints |

**C. Two TVs -** [**https://codeforces.com/problemset/problem/845/C**](https://codeforces.com/problemset/problem/845/C)

**🧩 Problem Statement (Rephrased)**

Polycarp wants to **watch n TV shows**, each with a time interval:

show i → starts at li and ends at ri

He owns **2 TVs**.

* A single TV **cannot watch two shows that overlap in time**.
* Even if one show ends **exactly when another starts**, they are **considered overlapping**, so they **cannot** be watched on the same TV.

We must determine whether **2 TVs** are **enough** to watch all n shows — i.e., can we assign each show to one of the two TVs so that no TV’s assigned shows overlap in time?

**Output:**

* "YES" — if two TVs are enough.
* "NO" — otherwise.

**📘 Example**

**Example 1**

Input:

3

1 2

2 3

4 5

Output:

YES

✅ Explanation:

* Show 1: [1, 2]
* Show 2: [2, 3]
* Show 3: [4, 5]

Shows 1 and 2 **touch at time 2**, so they overlap and need two TVs.  
Show 3 doesn’t overlap with either, so 2 TVs are enough.

**Example 2**

Input:

4

1 2

2 3

2 3

1 2

Output:

NO

🚫 Explanation:

* Show 1: [1, 2]
* Show 2: [2, 3]
* Show 3: [2, 3]
* Show 4: [1, 2]

At time 2, all four shows overlap — that requires 4 TVs.  
Hence, **“NO”**.

**💡 How to Think (Intuition)**

We need to **check if at any point more than two shows overlap**.

* If **3 or more** shows overlap → **“NO”**
* Otherwise → **“YES”**

Instead of checking every second (impossible, since times go up to 1e9), we can use a **sweep line algorithm**.

**⚙️ Solution Approach (Sweep Line Algorithm)**

**Step 1️⃣ — Represent events**

For each show [li, ri]:

* When the show **starts** → we **add +1 active show**.
* When the show **ends** → we **subtract -1 active show**.

So we store:

(l, +1) → start event

(r, -1) → end event

**Step 2️⃣ — Sort events properly**

We sort all events by time.  
But the tricky part is **when two events happen at the same time**.

**Rule (important):**

If one show **ends at time t**, and another **starts at time t**, they **overlap**.  
So we must process **starts before ends**.

That’s why:

if (a.first != b.first)

return a.first < b.first;

return a.second > b.second; // +1 (start) before -1 (end)

**Step 3️⃣ — Sweep through events**

Keep a counter active = number of currently overlapping shows.

For each event:

* active += e.second
* If active > 2 → print "NO" (since 3 overlaps)
* After processing all → print "YES"

**✅ Final Code (Explained)**

#include <bits/stdc++.h>

using namespace std;

#define ll long long

int main() {

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

cin.tie(nullptr);

int n;

cin >> n;

vector<pair<ll,int>> events;

events.reserve(2\*n);

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

ll l, r;

cin >> l >> r;

events.emplace\_back(l, +1); // start

events.emplace\_back(r, -1); // end

}

// Sort by time; start before end if equal

sort(events.begin(), events.end(), [](const pair<ll,int> &a, const pair<ll,int> &b){

if (a.first != b.first)

return a.first < b.first;

return a.second > b.second;

});

int active = 0;

for (auto &e : events) {

active += e.second;

if (active > 2) { // more than two overlapping shows

cout << "NO\n";

return 0;

}

}

cout << "YES\n";

return 0;

}

**🧮 Complexity Analysis**

| **Step** | **Operation** | **Complexity** |
| --- | --- | --- |
| Input reading | O(n) |  |
| Sorting events | O(n log n) |  |
| Sweep line | O(n) |  |
| **Total** | **O(n log n)** |  |
| Memory | **O(n)** |  |

**🏁 Summary**

| **Concept** | **Description** |
| --- | --- |
| Problem type | Interval overlap checking |
| Key idea | Sweep line algorithm |
| Sorting rule | Start events before end events |
| Condition | If more than 2 shows overlap → NO |
| Time complexity | O(n log n) |
| Space complexity | O(n) |

**A. Greg and Array-** [**https://codeforces.com/problemset/problem/295/A**](https://codeforces.com/problemset/problem/295/A)

**🧩 Problem Statement**

We have:

* An array a[1..n]
* m **operations**:  
  Each operation i is defined by 3 numbers → (li, ri, di)  
  Meaning:

Add di to each element a[li], a[li+1], ..., a[ri]

* k **queries**:  
  Each query (xi, yi) means:

Apply all operations from xi to yi (inclusive).

We must compute the **final array** after applying all queries (and hence all operations).

**🧠 Understanding with an Example**

**Example**

n=3, m=3, k=3

a = [1, 2, 3]

Operations:

1. (1, 2, +1)

2. (1, 3, +2)

3. (2, 3, +4)

Queries:

(1, 2)

(1, 3)

(2, 3)

**Step 1: What does each query mean?**

Each query tells us to **apply a range of operations**.

| **Query** | **Affects operations** | **Meaning** |
| --- | --- | --- |
| (1, 2) | op1, op2 | both ops applied once |
| (1, 3) | op1, op2, op3 | all ops applied once |
| (2, 3) | op2, op3 | last two ops applied once |

So:

* op1 is used in queries 1 and 2 → used **2 times**
* op2 is used in all queries → used **3 times**
* op3 is used in queries 2 and 3 → used **2 times**

**Step 2: Apply operations with frequency**

Each operation (li, ri, di) adds di to [li, ri].  
If operation i is applied cnt[i] times, total effect = di \* cnt[i].

| **Operation** | **li** | **ri** | **di** | **Count** | **Total Added** |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | 2 | 1 | 2 | +2 on [1,2] |
| 2 | 1 | 3 | 2 | 3 | +6 on [1,3] |
| 3 | 2 | 3 | 4 | 2 | +8 on [2,3] |

Now we add these totals to a.

**Step 3: Efficient Range Updates (Prefix Sum)**

Instead of updating [li, ri] one by one, we use **difference array** logic.

For each operation:

diff[li] += (di \* cnt[i])

diff[ri + 1] -= (di \* cnt[i])

Then compute prefix sum on diff to get total increment per index.

**Step 4: Apply to Original Array**

After computing total increments:

a[i] += total\_added[i]

That gives the final result.

**✅ Step-by-Step Implementation Plan**

1. **Read input**
2. **Store all operations**
3. **Compute how many times each operation is used**
   * Use prefix sum on queries (xi, yi)
4. **Compute total effect on array**
   * Use prefix sum again for operations
5. **Add results to original array and print**

**💡 Key Observations**

* 2 layers of prefix sum are required:
  1. For counting how often each operation is used
  2. For applying cumulative additions to the array efficiently

**⚙️ Final C++ Code**

#include <bits/stdc++.h>

using namespace std;

#define int long long

int32\_t main() {

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

cin.tie(nullptr);

int n, m, k;

cin >> n >> m >> k;

vector<int> a(n + 1);

for (int i = 1; i <= n; i++) cin >> a[i];

vector<int> l(m + 1), r(m + 1), d(m + 1);

for (int i = 1; i <= m; i++) cin >> l[i] >> r[i] >> d[i];

// Step 1: Count how many times each operation is applied

vector<int> op\_count(m + 2, 0);

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

int x, y; cin >> x >> y;

op\_count[x]++;

op\_count[y + 1]--;

}

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

op\_count[i] += op\_count[i - 1]; // prefix sum

}

// Step 2: Apply operations to array using prefix sum again

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

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

long long total = (long long)d[i] \* op\_count[i];

diff[l[i]] += total;

diff[r[i] + 1] -= total;

}

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

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

a[i] += diff[i];

}

for (int i = 1; i <= n; i++) cout << a[i] << " ";

cout << "\n";

return 0;

}

**⏱️ Complexity Analysis**

| **Step** | **Time** | **Space** |
| --- | --- | --- |
| Counting operation usage | O(k + m) | O(m) |
| Applying operations | O(m + n) | O(n) |
| **Total** | **O(n + m + k)** | **O(n + m)** |

✅ Fits perfectly within limits (≤ 10⁵).

**🧮 Example Revisited**

Input:

3 3 3

1 2 3

1 2 1

1 3 2

2 3 4

1 2

1 3

2 3

Output:

9 18 17

Everything matches!

**🧠 Summary**

| **Concept** | **Explanation** |
| --- | --- |
| What’s asked | Apply groups of range updates efficiently |
| Trick used | **Two-layer prefix sum** |
| First prefix | Count how many times each operation is used |
| Second prefix | Apply operation effects to array |
| Time complexity | O(n + m + k) |
| Space complexity | O(n + m) |

**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) |