**B. Sereja and Suffixes -** [**https://codeforces.com/problemset/problem/368/B**](https://codeforces.com/problemset/problem/368/B)

**🔍 Problem Understanding**

We’re given:

* An integer array a of size n.
* m queries — each query gives an index li.

For each li, we need to find **how many distinct numbers** appear in the suffix starting from index li to the end of the array (n).

Formally:  
We want the count of unique elements in a[li...n].

**Example**

n = 10, m = 10

a = [1,2,3,4,1,2,3,4,100000,99999]

queries: 1 2 3 4 5 6 7 8 9 10

Let’s analyze:

| **l** | **subarray (a[l...n])** | **distinct count** |
| --- | --- | --- |
| 1 | [1,2,3,4,1,2,3,4,100000,99999] | 6 |
| 2 | [2,3,4,1,2,3,4,100000,99999] | 6 |
| 3 | [3,4,1,2,3,4,100000,99999] | 6 |
| 4 | [4,1,2,3,4,100000,99999] | 6 |
| 5 | [1,2,3,4,100000,99999] | 6 |
| 6 | [2,3,4,100000,99999] | 5 |
| 7 | [3,4,100000,99999] | 4 |
| 8 | [4,100000,99999] | 3 |
| 9 | [100000,99999] | 2 |
| 10 | [99999] | 1 |

**🧠 Naive Idea (TLE ❌)**

For each query li, scan from li to n and count distinct numbers using a set.

for each query li:

set<int> st;

for i from li to n:

st.insert(a[i]);

print st.size();

➡️ **Time complexity:** O(m \* n) = up to 10¹⁰ operations (too slow)

**⚡ Efficient Approach — Reverse Traversal (O(n + m))**

**Observation:**

If we know the number of distinct elements in suffix starting at index i+1,  
then we can compute for index i easily:

* If a[i] has not appeared before (in suffix after i), then distinct count increases by 1.
* Otherwise, it remains the same.

**✅ Step-by-Step Logic**

1. Create an array suffixDistinct[n+1] where:
2. suffixDistinct[i] = number of distinct elements in a[i...n]
3. Maintain a frequency array or boolean visited array for elements a[i].
4. Traverse from **right to left (n → 1)**:
   * If element a[i] was **not seen before**, increment current distinct count.
   * Store that count in suffixDistinct[i].
5. Answer each query li instantly by outputting suffixDistinct[li].

**🧩 Example Walkthrough**

a = [1,2,3,4,1,2,3,4,100000,99999]

n = 10

Start from right:

| **i** | **a[i]** | **seen before?** | **count** | **suffixDistinct[i]** |
| --- | --- | --- | --- | --- |
| 10 | 99999 | no | 1 | 1 |
| 9 | 100000 | no | 2 | 2 |
| 8 | 4 | no | 3 | 3 |
| 7 | 3 | no | 4 | 4 |
| 6 | 2 | no | 5 | 5 |
| 5 | 1 | no | 6 | 6 |
| 4 | 4 | yes | 6 | 6 |
| 3 | 3 | yes | 6 | 6 |
| 2 | 2 | yes | 6 | 6 |
| 1 | 1 | yes | 6 | 6 |

✅ Works perfectly.

**✅ Final C++ Solution**

#include <bits/stdc++.h>

using namespace std;

int main() {

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

cin.tie(nullptr);

int n, m;

cin >> n >> m;

vector<int> a(n + 1);

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

cin >> a[i];

vector<int> distinctSuffix(n + 1);

vector<bool> seen(100001, false); // since ai ≤ 10^5

int distinctCount = 0;

// Traverse from end to start

for (int i = n; i >= 1; --i) {

if (!seen[a[i]]) {

seen[a[i]] = true;

distinctCount++;

}

distinctSuffix[i] = distinctCount;

}

// Answer queries

while (m--) {

int l;

cin >> l;

cout << distinctSuffix[l] << '\n';

}

return 0;

}

**🧮 Complexity Analysis**

| **Operation** | **Complexity** |
| --- | --- |
| Build suffix counts | **O(n)** |
| Each query answer | **O(1)** |
| **Total** | **O(n + m)** |
| Space | **O(n + max(a[i])) ≈ O(10⁵)** |

✅ Fits easily within 1 second.

**🏁 Summary**

| **Step** | **Concept** |
| --- | --- |
| 1️⃣ | Traverse from right to left |
| 2️⃣ | Track seen numbers |
| 3️⃣ | Store running distinct count in suffixDistinct[i] |
| 4️⃣ | Answer queries in O(1) |

**B. Polycarp Training -** [**https://codeforces.com/problemset/problem/1165/B**](https://codeforces.com/problemset/problem/1165/B)

**🔍 Problem Understanding**

Polycarp trains by solving contests.

* On **day 1**, he must solve **1** problem.
* On **day 2**, he must solve **2** problems.
* On **day 3**, he must solve **3** problems, etc.

He has a list of contests — each contest i has a[i] problems.

He can choose **one unused contest per day**.

But he can only train on that day if there is **a contest with at least k problems** (where k is the day number).

Once he uses a contest, it’s gone.

He wants to train for as **many consecutive days as possible**.

We must find:  
👉 The **maximum number of days** Polycarp can train if he chooses contests optimally.

**🧩 Example 1**

Input:

4

3 1 4 1

Contests = [3, 1, 4, 1]

Let’s sort the contests: [1, 1, 3, 4]

Day 1 → needs ≥ 1 → pick contest with 1 problem (remaining: [1, 3, 4]) ✅  
Day 2 → needs ≥ 2 → pick contest with 3 problems (remaining: [1, 4]) ✅  
Day 3 → needs ≥ 3 → pick contest with 4 problems (remaining: [1]) ✅  
Day 4 → needs ≥ 4 → only 1 left ❌ (not enough)

✅ Total days = 3

**🧩 Example 2**

Input:

3

1 1 1

Sorted: [1, 1, 1]

Day 1 → needs ≥1 → OK  
Day 2 → needs ≥2 → ❌ none

✅ Total days = 1

**🧩 Example 3**

Input:

5

1 1 1 2 2

Sorted: [1, 1, 1, 2, 2]

Day 1 → pick contest with 1 problem ✅  
Day 2 → pick contest with 2 problems ✅  
Day 3 → needs ≥3 → ❌ none left

✅ Total = 2 days.

**💡 Key Insight**

If you sort the contests ascending by the number of problems,  
you can **greedily pick contests** from smallest to largest.

We maintain a counter day = current training day.

For each contest in sorted order:

* If a[i] ≥ day, then Polycarp can use this contest for the current day, and we increment day.
* Otherwise, skip that contest (it’s too small).

**✅ Algorithm (Greedy)**

1. Sort the array a.
2. Initialize day = 1 (first day).
3. Traverse sorted contests:
   * If contest has a[i] >= day → Polycarp uses it, increment day.
4. The final answer is day - 1.

**🧠 Example Walkthrough**

a = [3,1,4,1]

Sorted = [1,1,3,4]

| **Contest** | **Needed problems** | **Can use?** | **Days trained** |
| --- | --- | --- | --- |
| 1 | ≥1 | ✅ yes | 1 |
| 1 | ≥2 | ❌ no | still 1 |
| 3 | ≥2 | ✅ yes | 2 |
| 4 | ≥3 | ✅ yes | 3 |

✅ Answer = 3

**🧮 Complexity**

| **Step** | **Complexity** |
| --- | --- |
| Sorting | O(n log n) |
| Single pass | O(n) |
| **Total** | **O(n log n)** |
| Memory | O(1) extra |

Works perfectly for n ≤ 2×10⁵.

**✅ Final C++ Code**

#include <bits/stdc++.h>

using namespace std;

int main() {

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

cin.tie(nullptr);

int n;

cin >> n;

vector<int> a(n);

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

cin >> a[i];

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

int day = 1;

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

if (a[i] >= day) {

day++;

}

}

cout << day - 1 << "\n";

return 0;

}

**🏁 Summary**

| **Concept** | **Description** |
| --- | --- |
| Approach | Greedy + Sorting |
| Sorting by | Contest problems |
| Condition | If a[i] >= current\_day → train |
| Complexity | O(n log n) |
| Output | Maximum days Polycarp can train |