ALGORITHM DESIGN TERM PROJECT

Team Members:

1. Min Myint Moh Soe (6530262)
2. Kyaw Ye Lwin @ Anmol (6530377)
3. NOEL PAING OAK SOE (6530183)

Table Of Contents

1. High-Level Explanation of the Problem

The problem "Isolation" presents a partitioning challenge where we are tasked with dividing an array of integers into disjoint, non-empty segments. Each segment must satisfy the constraint that no more than **k** distinct integers appear exactly once. The objective is to count the number of valid ways to divide the array under this constraint. Given that the result can be large, the answer must be returned modulo **998244353**.

This problem can be seen as a variation of the classical **segmentation problem** but with a specific focus on limiting the occurrence of distinct integers within each segment. The complexity arises from ensuring that the segments adhere to this constraint while exploring all possible divisions of the array.

2. Abstract Explanation of Approaches

We explored three primary approaches to solve the problem, each with varying levels of complexity and performance trade-offs:

- **Brute Force Python Approach:** This method involves generating all possible ways to divide the array and checking for each partition if the condition on distinct integers is met. The brute force nature of this approach leads to an exponential time complexity, making it highly inefficient for large inputs. It serves as a proof of concept for smaller datasets but is impractical for larger arrays due to its computational limits.
- **Brute Force Java Approach:** Similar to the Python approach, this method is implemented in Java to leverage Java's more optimized execution engine. While the brute-force algorithm remains inefficient (with the same exponential complexity), Java provides some performance improvements in terms of speed and memory management. This approach serves as a transitional step between the baseline brute force and more advanced optimizations.
- **BucketSort Optimized Approach:** This is the most efficient solution, combining **dynamic programming** with a **bucket sorting technique**. By dividing the array into manageable segments and utilizing buckets to track and count elements, we can efficiently compute the number of valid divisions. The algorithm avoids the inefficiencies of brute-force methods by focusing on smaller, localized operations within each segment, dramatically improving performance. The time complexity of this approach is approximately **O(n log n)**, making it well-suited for larger arrays, up to the problem’s upper limit of 100,000 elements.

3. Performance and Efficiency Comparison

- **Brute Force Python Approach**
    - **Time Complexity**: **O(2^n)**
    - **Memory Usage**: High due to the recursive nature of the implementation.
      This method, while easy to implement, suffers from severe performance issues as the number of possible partitions grows exponentially with the size of the array. It is useful as a baseline for small arrays but becomes infeasible for larger inputs.
- **Brute Force Java Approach**
    - **Time Complexity**: **O(2^n)**
    - **Memory Usage**: Lower compared to Python, due to Java's optimized memory management.
      Java offers improved execution times compared to Python, but the brute-force nature of the algorithm still makes it unsuitable for larger arrays. This approach highlights the limitations of brute-force solutions for problems with large input sizes.
- **BucketSort Optimized Approach**
    - **Time Complexity**: **O(n log n)**
    - **Memory Usage**: Efficient, with the use of buckets to segment the problem.
      This approach significantly improves performance by reducing the problem's complexity. The use of dynamic programming ensures that previous results are reused, and the bucket-based segmentation helps efficiently handle large datasets. By avoiding unnecessary recalculations, the algorithm is able to scale effectively for arrays up to the problem's upper bounds.

Brute Force Java Approach

Code Solution

```java
import java.util.*;

public class Main {
    static final int MOD = 998244353;
    static final int MAX_N = 100005;
    static List<Integer>[] occurrences = new ArrayList[MAX_N];
    static int[] cnt = new int[MAX_N];
    static int[] dp = new int[MAX_N];

    public static void add(int a, int b, int diff) {
        for (int i = a; i <= b; ++i) {
            cnt[i] += diff;
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int k = sc.nextInt();

        for (int i = 1; i <= n; ++i) {
            occurrences[i] = new ArrayList<>();
            occurrences[i].add(-1); // fake position
        }

        dp[0] = 1;

        for (int R = 0; R < n; ++R) {
            int x = sc.nextInt();
            List<Integer> vec = occurrences[x];

            if (vec.size() >= 2) {
                add(vec.get(vec.size() - 2) + 1, vec.get(vec.size() - 1), -1);
            }

            add(vec.get(vec.size() - 1) + 1, R, 1);
            vec.add(R);

            for (int L = 0; L <= R; ++L) {
                if (cnt[L] <= k) {
                    dp[R + 1] += dp[L];
                    if (dp[R + 1] >= MOD) {
                        dp[R + 1] -= MOD;
                    }
                }
            }
        }

        System.out.println(dp[n]);
    }
}
```

Approach

To solve this problem efficiently, we use **dynamic programming (DP)** combined with a **sliding window technique**.

1. **Dynamic Programming (DP)**: We'll use a dp[] array by initializing it with size of original array input where each dp[i] represents the number of ways to divide the array from index 0 to i. Initially, dp[0] = 1 because there's one way to divide an empty array (no segments).

2. **Sliding Window**: At every position R in the array (as we move from left to right), we'll try to determine how many valid segments end at R. For this, we look back to all possible starting points L (from R to 0).
3. **Checking Validity**: For each potential segment [L, R], we check if it's valid meaning it has at most k distinct integers that appear exactly once. If it's valid, we update dp[R+1] by adding dp[L], which means the number of ways to divide the array up to R+1 is increased by the number of ways to divide the array up to L.

How the Occurrences and cnt[] Help

To efficiently check if a segment is valid, we need to know how many distinct integers appear exactly once in the segment [L, R]. This is where **occurrences[]** and **cnt[]** come into play:

- **occurrences[]**: This keeps track of the positions where each element of the array appears. For example, if 1 appears at positions 0 and 3, occurrences[1] = [0, 3].
- **cnt[]**: This array tracks the number of distinct integers that appear exactly once in the current segment [L, R].

The Role of the add() Function

The **add(a, b, diff)** function is crucial for updating the cnt[] array:

- **When an element appears again**: When an element that already appeared in the segment shows up again, it no longer counts as "appearing exactly once." So, we need to **remove its contribution** from cnt[] in the previous segment. This is done by calling add() with -1 to reduce the count.
- **When an element is new in the segment**: When we move the right boundary R of the sliding window, and a new element is added to the segment, we call add() with +1 to increase the count of distinct elements that appear exactly once.

Putting It All Together

1. Start at position R = 0 and expand the segment [L, R].

2. For each position R, look back at all possible starting points L. For each L, check if the segment [L, R] is valid using the cnt[] array (i.e., make sure cnt[L] <= k).
3. If the segment [L, R] is valid, add dp[L] to dp[R+1].
4. Update the occurrences[] and cnt[] arrays using the add() function to account for elements that appear again.

Example Test Case Explained

1. **Initialization**:
    - dp[0] = 1: There is exactly one way to divide an empty array, which is to not divide it at all.
2. **First Element (R = 0)**:
    - The first element of the array is 1. We try to see if we can form segments that end at R = 0.
    - We check all possible starting points L. The only valid segment is [1] (since there is only one element). It's valid because it contains one distinct integer appearing exactly once, which satisfies k = 1.
    - Update dp[1] = dp[0] = 1.
    - Now, dp[] = {1, 1, 0, 0}.
3. **Second Element (R = 1)**:
    - The second element is also 1. We now check all possible starting points L for the segment ending at R = 1.

Possible segments:

- Segment [1, 1] from L = 0 to R = 1: This is **not valid** because 1 appears twice in this segment, violating the condition of having at most k = 1 distinct integers that appear exactly once.
- Segment [1] from L = 1 to R = 1: This segment is **valid** because 1 appears exactly once, satisfying the condition. So we add dp[1] to dp[2].
- Now, dp[2] = dp[1] = 1.
- Updated dp[] = {1, 1, 1, 0}.

4. **Third Element (R = 2)**:
    - The third element is 2. We now check all possible starting points L for the segment ending at R = 2.

Possible segments:

- Segment [1, 1, 2] from L = 0 to R = 2: This is **not valid** because the segment contains two distinct integers, 1 and 2, which both appear exactly once. This violates the condition of having at most k = 1 distinct integers appearing once.
- Segment [1, 2] from L = 1 to R = 2: This is **not valid** because both 1 and 2 appear exactly once.
- Segment [2] from L = 2 to R = 2: This is **valid** because 2 appears exactly once. We add dp[2] to dp[3].
- Now, dp[3] = dp[2] = 1.
- Updated dp[] = {1, 1, 1, 1}.

Conclusion

At the end of the process, the value dp[n] (i.e., dp[3]) tells us the total number of ways to divide the array. In this case, dp[3] = 3.

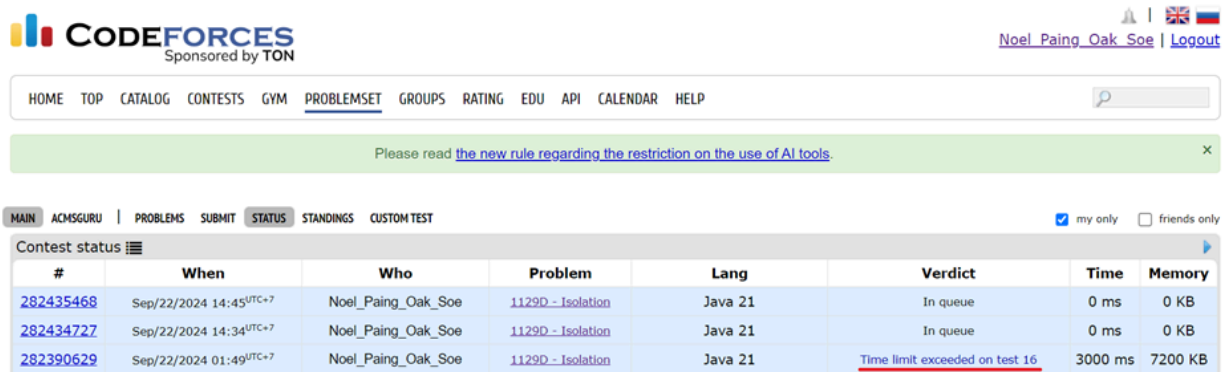
The valid segmentations are:

1. [1], [1], [2]
2. [1, 1], [2]
3. [1, 1, 2]

Notes:

- **occurrences[]**: Tracks where each element appears in the array. For example, occurrences[1] = {0, 1} shows that 1 appears at indices 0 and 1.
- **cnt[]**: Tracks how many distinct elements in the current segment appear exactly once. If cnt[] exceeds k, the segment is invalid.
- **add() function**: Adjusts cnt[] based on element appearances. If an element reappears, reduce its count in cnt[]; if a new element appears, increase its count.

Results



Limitations:

- **Time Complexity**: The solution has a time complexity of **O(N²)** due to the double for loop. As the array size n grows, the performance degrades significantly, making the solution less efficient for larger test cases.
- **Ineffectiveness for Large Cases**: As a result, the solution only passed **16 test cases** on Codeforces. Larger test cases exceed the time limits due to the quadratic nature of the algorithm.

Optimized Solution

The naive brute-force approach is computationally infeasible for large arrays due to its exponential time complexity. Therefore, an optimized solution using dynamic programming combined with a **bucketing algorithm** is employed to handle large test cases efficiently.

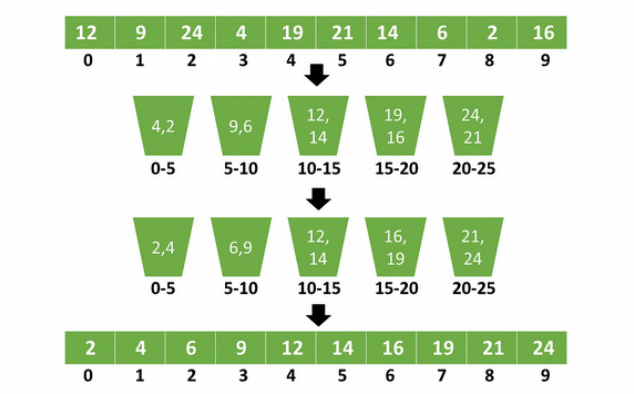
High-Level Overview of the Optimized Algorithm

- **Dynamic Programming (DP):** We use an array **dp[]** where **dp[i]** represents the number of ways to partition the array up to index **i**.
- **Goal:** For each position **R** in the array, we aim to compute **dp[R+1]** efficiently.

The Need for Efficient Counting

- **Inefficiency of Naive Counting:** Checking every possible **L** for each **R** results in a time complexity of **$O(n^2)$** , which is impractical for large n.
- **Solution:** We need a data structure that allows for efficient updates and queries over ranges to maintain counts of elements and compute the required sums quickly.

Bucketing Algorithm



- **Concept:** Divide the array into **buckets** (blocks) of size **B** (e.g., 300 elements per bucket for this code).
- **Purpose of Buckets:**
    - **Efficient Range Updates:** Perform updates on counts over a range of positions efficiently.
    - **Fast Queries:** Quickly compute the sum of **dp[L]** for positions within a bucket that satisfy the condition.

How the Bucketing Algorithm Works :

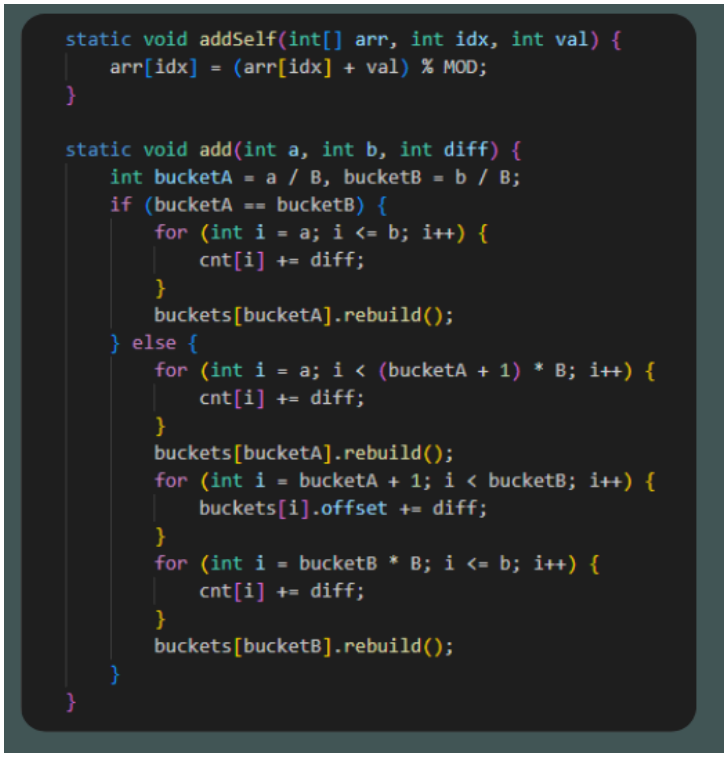
```
static class Bucket {
    int id, offset;
    int[] prefSum = new int[B];

    Bucket(int id) {
        this.id = id;
    }
```

Data Structures

- **buckets[]:** An array of **Bucket** objects, each representing a block of positions in the array.
- **Each Bucket Contains:**
    - **offset:** A value added to all counts within the bucket, allowing efficient bulk updates.

- **prefSum[]:** An array storing prefix sums of **dp[L]** within the bucket, facilitating quick queries.

Operations :



```
static void addSelf(int[] arr, int idx, int val) {
    arr[idx] = (arr[idx] + val) % MOD;
}

static void add(int a, int b, int diff) {
    int bucketA = a / B, bucketB = b / B;
    if (bucketA == bucketB) {
        for (int i = a; i <= b; i++) {
            cnt[i] += diff;
        }
        buckets[bucketA].rebuild();
    } else {
        for (int i = a; i < (bucketA + 1) * B; i++) {
            cnt[i] += diff;
        }
        buckets[bucketA].rebuild();
        for (int i = bucketA + 1; i < bucketB; i++) {
            buckets[i].offset += diff;
        }
        for (int i = bucketB * B; i <= b; i++) {
            cnt[i] += diff;
        }
        buckets[bucketB].rebuild();
    }
}
```

Updating Counts (add() Function):

- Adjusts the counts of elements over a range [a, b] by a diff value.
- Handles cases where elements appear for the first time or reappear in the segment.

First Occurrence of an Element:

- Increase counts in the range from the previous occurrence +1 to the current position R by +1.

Reappearance of an Element:

- Decrease counts in the range from the second-last occurrence +1 to the last occurrence by -1.
- Increase counts in the range from the last occurrence +1 to the current position R by +1.

```
    void rebuild() {
        int first = id * B, last = Math.min((id + 1) * B - 1, MAX_N - 1);
        int smallest = Integer.MAX_VALUE;
        for (int i = first; i <= last; i++) {
            smallest = Math.min(smallest, offset + cnt[i]);
        }
        for (int i = first; i <= last; i++) {
            cnt[i] -= smallest - offset;
        }
        offset = smallest;
        Arrays.fill(prefSum, 0);
        for (int i = first; i <= last; i++) {
            addSelf(prefSum, cnt[i], dp[i]);
        }
        for (int x = 1; x < B; x++) {
            addSelf(prefSum, x, prefSum[x - 1]);
        }
    }
}
```

Rebuilding Buckets (rebuild() Method):

- Recomputes the counts and prefix sums within a bucket after updates.
- Ensures that the **prefSum[]** reflects the current state of counts and offsets.

prefSum and Cumulative Sums:

To efficiently compute **dp[R+1]**, the algorithm uses **prefix sums (prefSum[])** within each bucket. This allows for quick summation of **dp[L]** over valid positions without iterating through every element.

prefSum in Each Bucket:

- Each bucket maintains a **prefSum[]** array of size **B** (the bucket size).
- **prefSum[x]** stores the cumulative sum of **dp[L]** for counts up to **x** within the bucket.
- It accounts for the **offset**, which adjusts counts uniformly within the bucket.

Updating prefSum[]:

- When the counts within a bucket change due to an update, the **rebuild()** method recalculates **prefSum[].**
- It first normalizes counts by subtracting the smallest count **(smallest**) to ensure counts are non-negative.
- Then, it iterates over positions within the bucket to compute prefix sums**: prefSum[cnt[i]] = (prefSum[cnt[i]] + dp[i]) % MOD;**
- Cumulatively adds **dp[i]** to prefSum based on the adjusted count **cnt[i]**.

```
        dp[0] = 1;
        buckets[0].rebuild();

        for (int R = 0; R < n; R++) {
            int x = a[R];
            List<Integer> vec = occurrences[x];
            if (vec.size() >= 2) {
                add(vec.get(vec.size() - 2) + 1, vec.get(vec.size() - 1), -1);
            }
            add(vec.get(vec.size() - 1) + 1, R, 1);
            vec.add(R);

            int total = 0;
            for (int i = 0; i <= R / B; i++) {
                Bucket bucket = buckets[i];
                int atMost = k - bucket.offset;
                if (atMost >= 0) {
                    total = (total + bucket.prefSum[Math.min(atMost, B - 1)]) % MOD;
                }
            }

            dp[R + 1] = total;
            buckets[(R + 1) / B].rebuild();
        }

        System.out.println(dp[n]);
    }
}
```

Computing dp[R+1] with prefSum :

For Each Position R:

- Update counts using **add()** when an element is added to the segment or when it reappears.
- Iterate over all buckets to compute the total number of ways **(total)** to partition the array up to position **R+1** by summing **dp[L]** for valid L.
- **Efficient Summation:**
    - To compute **dp[R+1]**, the algorithm needs to sum **dp[L]** for all positions **L** where the number of distinct integers appearing exactly once is at most **k**.
    - Instead of checking each **L**, it uses **prefSum[]** to get the total sum quickly.
    - For each bucket, it calculates:

      atMost = k - bucket.offset;

      If atMost >= 0, it adds bucket.prefSum[min(atMost, B - 1)] to the total.

Example:

- Suppose in a bucket, after adjustments, the counts range from 0 to 2, and **prefSum[] = [sum0, sum1, sum2].**
- If **k - offset = 1**, we need the cumulative sum for counts up to **1**.
- The total contribution from this bucket is **prefSum[1]**.

Step by Step Walkthrough :

Test Case:

- n = 3, k = 1
- a = [1, 1, 2]

Dynamic Programming Array (dp):

- Initialize **dp[0] = 1** (there is one way to partition an empty array).
- Initial **dp = [1, 0, 0, 0]**

Bucketing Setup:

- Bucket Size (B): Choose **B = 2** for this small example.
- Buckets:
    - Bucket 0: Positions 0 and 1
    - Bucket 1: Position 2

Data Structures:

- **cnt[]:** An array to keep track of the counts of elements appearing exactly once, per position.cnt[i] represents the **number of distinct integers that appear exactly once** in the segments ending at position i
    - **Initialize cnt = [0, 0, 0]**
- **offset[]:** An array to store offsets for each bucket, to allow efficient range updates.
    - **Initialize offset = [0, 0]**
- **prefSum[]:** A 2D array where prefSum[i][x] stores the prefix sums for bucket i up to count x.
    - Each **prefSum[i]** has size **B + 1** (since counts can range from 0 to B).
    - Initialize **prefSum[0] = [0, 0, 0]**, **prefSum[1] = [0, 0, 0]**

We process each position **R** from **0** to **n - 1** and update **dp[R + 1].**

Occurrences:

- **occurrences:** A dictionary to store the positions where each number appears, initialized with a dummy -1 for each number.
    - **occurrences = {1: [-1], 2: [-1]}**

Position R = 0 (Element = 1):

Update Occurrences:

- occurrences[1] = [-1, 0] (initial dummy -1 and current position 0)

Adjust Counts with the Bucketing Algorithm:

- Since 1 appears for the first time:
    - Increase counts from the previous occurrence +1 to current R:
        - Range: **L = occurrences[1][-1] + 1 = 0 to R = 0**
    - Buckets Affected: **Bucket 0** (positions 0 to 0)
    - Update **cnt[0] += 1 ⇒ cnt = [1, 0, 0].**

Compute dp[1]:

- Possible Segments Ending at **R = 0**:
    - **Segment [0, 0]**:
        - **cnt[0] = 1** (number of distinct integers appearing exactly once)
        - Since **cnt[0] ≤ k**, this segment is valid.
        - Update dp[1] by adding dp[0]:
            - **dp[1] += dp[0] ⇒ dp[1] = 0 + 1 = 1**

Updated dp Array:

- **dp = [1, 1, 0, 0]**

Position R = 1 (Element = 1):

Update Occurrences:

- occurrences[1] = [-1, 0, 1]

Adjust Counts with the Bucketing Algorithm:

- Since 1 appears again:
    - Decrease counts from the previous previous occurrence +1 to previous occurrence:
        - Range: **L = occurrences[1][-2] + 1 = 0 to occurrences[1][-1] = 0**
    - Buckets Affected: **Bucket 0**
    - Update **cnt[0] -= 1 ⇒ cnt = [0, 0, 0]**
    - Increase counts from previous occurrence +1 to current R:
        - Range: **L = occurrences[1][-1] + 1 = 1 to R = 1**
    - Buckets Affected: **Bucket 0**
    - **Update cnt[1] += 1 ⇒ cnt = [0, 1, 0]**

Compute dp[2]:

- Possible Segments Ending at **R = 1**:
    - **Segment [0, 1]**:
        - **cnt[0] = 0**
        - Since **cnt[0] ≤ k**, this segment is valid.
        - Update dp[2] by adding dp[0]:
            - **dp[2] += dp[0] ⇒ dp[2] = 0 + 1 = 1**
    - **Segment [1, 1]:**
        - **cnt[1] = 1**
        - Since **cnt[1] ≤ k**, this segment is valid.
        - Update dp[2] by adding dp[1]:
            - **dp[2] += dp[1] ⇒ dp[2] = 1 + 1 = 2**

Updated dp Array:

- **dp = [1, 1, 2, 0]**

Position R = 2 (Element = 2):

Update Occurrences:

- occurrences[2] = [-1, 2]

Adjust Counts with the Bucketing Algorithm:

- Since 2 appears for the first time:
    - Increase counts from the previous occurrence +1 to current R:
        - Range: **L = occurrences[2][-1] + 1 = 0 to R = 2**
    - Buckets Affected:
        - **Bucket 0** (positions 0 and 1)
        - **Bucket 1** (position 2)
    - Update Counts:
        - Bucket 0:
            - **cnt[0] += 1 ⇒ cnt[0] = 1**
            - **cnt[1] += 1 ⇒ cnt[1] = 2**
        - Bucket 1:
            - **cnt[2] += 1 ⇒ cnt[2] = 1**
    - After Update**: cnt = [1, 2, 1]**

Compute dp[3]:

- Possible Segments Ending at **R = 2**:
    - **Segment [0, 2]**:
        - **cnt[0] = 1**
        - Since **cnt[0] ≤ k,** this segment is valid.
        - Update dp[3] by adding dp[0]:
            - **dp[3] += dp[0] ⇒ dp[3] = 0 + 1 = 1**
    - **Segment [1, 2]:**
        - **cnt[1] = 2**
        - Since **cnt[1] > k**, this segment is invalid.
    - **Segment [2, 2]:**
        - **cnt[2] = 1**
        - Since **cnt[2] ≤ k,** this segment is valid.
        - Update dp[3] by adding dp[2]:
            - **dp[3] += dp[2] ⇒ dp[3] = 1 + 2 = 3**

Updated dp Array:

- **dp = [1, 1, 2, 3]**

Conclusion:

- The total number of valid ways to partition the array is dp[n] = dp[3] = 3.

Valid Partitions:

1. [1], [1], [2]
2. [1, 1], [2]
3. [1, 1, 2]

How the Bucketing Algorithm Enhances Efficiency

Reduced Time Complexity:

To efficiently handle large arrays, we divide the array into smaller buckets (blocks) of size B. This allows us to perform range updates and queries quickly, significantly reducing the computational complexity

- **Per Update/Query:** Operations within a bucket are **O(B)**, and there are **O($\frac{n}{B}$)**buckets.
- **Overall:** The algorithm achieves approximately O(n$\sqrt{n}$) time complexity by choosing an optimal bucket size B ≈ $\sqrt{n}$

Efficient Range Updates:

- **Bulk Updates with offset:** Adjust counts over entire buckets without iterating through each element.

Quick Summation of Valid Positions:

- **Prefix Sums:** Avoid the need to check each position individually by using precomputed sums.

Conclusion :

The "Isolation" problem highlights the value of efficient algorithm design in tackling complex array partitioning challenges.

- Brute force approaches in Python and Java, with their O(2^n) time complexity, struggle with large inputs.
- The optimized bucketing algorithm achieves near O(n) time complexity, greatly enhancing performance.

Key improvements include:

1. Dynamic programming utilizing a dp[] array.
2. Efficient range updates through bucket offsets.
3. Fast cumulative sum queries using prefSum[].

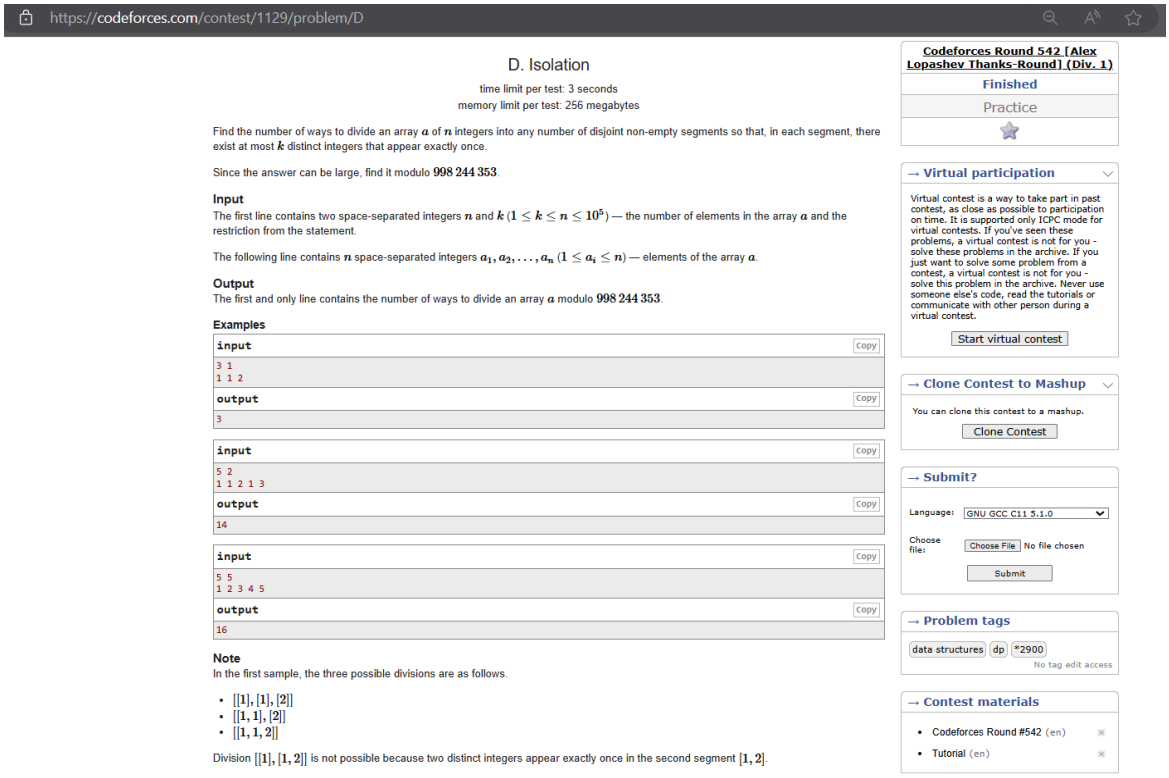
This bucketing approach can manage arrays of up to 10^5 elements and has passed all Codeforces test cases.

279993638	Sep/06/2024 03:27 UTC+7	TrisTheKitten	1129D - Isolation	Java 21	Accepted	2406 ms	17400 KB

References :

Question Reference:

https://codeforces.com/problemset/problem/1129/D



https://codeforces.com/contest/1129/problem/D

D. Isolation

time limit per test: 3 seconds
memory limit per test: 256 megabytes

Find the number of ways to divide an array a of n integers into any number of disjoint non-empty segments so that, in each segment, there exist at most k distinct integers that appear exactly once.

Since the answer can be large, find it modulo $998\,244\,353$.

Input
The first line contains two space-separated integers n and k ($1 \le k \le n \le 10^5$) — the number of elements in the array a and the restriction from the statement.

The following line contains n space-separated integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le n$) — elements of the array a.

Output
The first and only line contains the number of ways to divide an array a modulo $998\,244\,353$.

Examples

input
```
3 1
1 1 2
```
output
```
3
```

input
```
5 2
1 1 2 1 3
```
output
```
14
```

input
```
5 5
1 2 3 4 5
```
output
```
16
```

Note
In the first sample, the three possible divisions are as follows.

- $[[1], [1], [2]]$
- $[[1, 1], [2]]$
- $[[1, 1, 2]]$

Division $[[1], [1, 2]]$ is not possible because two distinct integers appear exactly once in the second segment $[1, 2]$.

Codeforces Round 542 [Alex Lopashev Thanks-Round] (Div. 1)
Finished
Practice

→ Virtual participation
Virtual contest is a way to take part in past contest, as close as possible to participation on time. It is supported only ICPC mode for virtual contests. If you've seen these problems, a virtual contest is not for you - solve these problems in the archive. If you just want to solve some problem from a contest, a virtual contest is not for you - solve this problem in the archive. Never use someone else's code, read the tutorials or communicate with other person during a virtual contest.
Start virtual contest

→ Clone Contest to Mashup
You can clone this contest to a mashup.
Clone Contest

→ Submit?
Language: GNU GCC C11 5.1.0
Choose file: Choose File No file chosen
Submit

→ Problem tags
data structures dp *2900
No tag edit access

→ Contest materials
- Codeforces Round #542 (en)
- Tutorial (en)

Solution Reference:

https://www.youtube.com/watch?v=ldyKbHFjDeI&list=PLl0KD3g-oDOFqBqIfrdV3l68bZbP9VAcC&index=2

Bucketing Algorithm Concept Explanation Reference :

https://www.geeksforgeeks.org/bucket-sort-2/

Submission Reference:

https://codeforces.com/submissions/TrisTheKitten

Contest status

#	When	Who	Problem	Lang	Verdict	Time	Memory
279993638	Sep/06/2024 03:27 UTC+7	TrisTheKitten	1129D - Isolation	Java 21	Accepted	2406 ms	17400 KB
279993564	Sep/06/2024 03:26 UTC+7	TrisTheKitten	1129D - Isolation	PyPy 3-64	Wrong answer on test 1	62 ms	0 KB
279993517	Sep/06/2024 03:25 UTC+7	TrisTheKitten	1129D - Isolation	PyPy 3-64	Time limit exceeded on test 16	3000 ms	30600 KB
279993406	Sep/06/2024 03:23 UTC+7	TrisTheKitten	1129D - Isolation	PyPy 3-64	Time limit exceeded on test 16	3000 ms	48800 KB
279993300	Sep/06/2024 03:22 UTC+7	TrisTheKitten	1129D - Isolation	PyPy 3-64	Time limit exceeded on test 16	3000 ms	49400 KB
279993216	Sep/06/2024 03:20 UTC+7	TrisTheKitten	1129D - Isolation	PyPy 3-64	Time limit exceeded on test 16	3000 ms	49700 KB