<https://leetcode.com/problems/minimum-number-of-operations-to-make-array-continuous/description/>

You are given an integer array nums. In one operation, you can replace any element in nums with any integer.

nums is considered continuous if both of the following conditions are fulfilled:

All elements in nums are unique.

The difference between the maximum element and the minimum element in nums equals nums.length - 1.

For example, nums = [4, 2, 5, 3] is continuous, but nums = [1, 2, 3, 5, 6] is not continuous.

Return the minimum number of operations to make nums continuous.

**Example 1:**

**Input:** nums = [4,2,5,3]

**Output:** 0

**Explanation:** nums is already continuous.

**Example 2:**

**Input:** nums = [1,2,3,5,6]

**Output:** 1

**Explanation:** One possible solution is to change the last element to 4.The resulting array is [1,2,3,5,4], which is continuous.

**Example 3:**

**Input:** nums = [1,10,100,1000]

**Output:** 3

**Explanation:** One possible solution is to:- Change the second element to 2.- Change the third element to 3.- Change the fourth element to 4.The resulting array is [1,2,3,4], which is continuous.

**Constraints:**

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^9

**Attempt 1: 2024-12-11**

**Solution 1: Not fixed length Sliding Window (180 min)**

**Style 1: Srink the left side**

class Solution {

    public int minOperations(int[] nums) {

        int n = nums.length;

        // Remove duplicates and sort the array

        Arrays.sort(nums);

        int uniqueNumbers = 1;

        for(int i = 1; i < n; i++) {

            if(nums[i] != nums[i - 1]) {

                nums[uniqueNumbers++] = nums[i];

            }

        }

        int maxContinuousSubarrayLen = 0;

        int i = 0;

        for(int j = 0; j < uniqueNumbers; j++) {

            // The difference in sorted array between the maximum element(nums[j])

// and the minimum element(nums[i]) in nums equals nums.length - 1,

            // if the difference more than nums.length - 1, it violate the definition

// of 'continuous' array (between nums[i] till nums[j]), we have to shrink

            // the window size by moving forward left index 'i' to find another window

            while(nums[j] - nums[i] > n - 1) {

                i++;

            }

            // Update the maximum continuous subarray length

            maxContinuousSubarrayLen = Math.max(maxContinuousSubarrayLen, j - i + 1);

        }

        // Calculate the minimum operations, besides maximum continuous

        // subarray length, all remain chars have to replace

        return n - maxContinuousSubarrayLen;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Style 2: Expand the right side**

class Solution {

    public int minOperations(int[] nums) {

        int n = nums.length;

        // Remove duplicates and sort the array

        Arrays.sort(nums);

        int uniqueNumbers = 1;

        for(int i = 1; i < n; i++) {

            if(nums[i] != nums[i - 1]) {

                nums[uniqueNumbers++] = nums[i];

            }

        }

        int maxContinuousSubarrayLen = 0;

        int j = 0;

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

            // The difference in sorted array between the maximum element(nums[j])

            // and the minimum element(nums[i]) in nums equals nums.length - 1,

            // if the difference less than nums.length - 1, it matches the definition

            // of 'continuous' array (between nums[i] till nums[j]), we can continue

            // expanding the window size by moving forward right index 'j' to find

            // another window

            while(j < uniqueNumbers && nums[j] - nums[i] <= n - 1) {

                j++;

            }

            // Update the maximum continuous subarray length

            maxContinuousSubarrayLen = Math.max(maxContinuousSubarrayLen, j - i);

        }

        // Calculate the minimum operations, besides maximum continuous

        // subarray length, all remain chars have to replace

        return n - maxContinuousSubarrayLen;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/2009>

**Problem Description**

The problem presents an integer array nums. The goal is to make the given array nums continuous by performing a certain operation. The operation consists of replacing any element in nums with any integer. An array is defined as continuous if it satisfies the following two conditions:

All elements in the array are unique.

The difference between the maximum element and the minimum element in the array is equal to the length of the array minus one.

The requirement is to return the minimum number of operations needed to make the array nums continuous.

**Intuition**

To find the minimum number of operations to make the nums array continuous, we use a two-pointer approach. Here's the general intuition behind this approach:

Removing Duplicates: Since all numbers must be unique in a continuous array, we first remove duplicates by converting the array to a set and then back to a sorted list.

Finding Subarrays with Potential: **We iterate through the sorted and deduplicated nums to look for subarrays that could potentially be continuous with minimal changes. Each subarray is characterized by a fixed starting point (i) and a dynamically found endpoint (j), where the difference between the maximum and minimum element (which is the first and last in the sorted subarray) is not greater than the length of the array minus one.**

Greedy Selection: **For each starting point i, we increment the endpoint j until the next element would break the continuity criterion. The size of the subarray between points i and j represents a potential continuous subarray.**

Calculating Operations: For each of these subarrays, we calculate the number of operations needed by subtracting the number of elements in the subarray from the total number of elements in nums. The rationale is that elements not in the subarray would need to be replaced to make the entire array continuous.

Finding the Minimum: As we want the minimum number of operations, we track the smallest number of operations needed throughout the iteration by using the min() function, updating the ans variable accordingly.

The loop efficiently finds the largest subarray that can be made continuous without adding any additional elements (since adding elements is not an option as per problem constraints). The remaining elements—those not included in this largest subarray—are the ones that would need to be replaced. The count of these elements gives us the minimum operations required.

**Solution Approach**

The solution uses a sorted array without duplicates and a sliding window to find the minimum number of operations. The steps involved in the implementation are as follows:

Sorting and Deduplication: The input array nums is first converted to a set to remove any duplicates since our final array needs to have all unique elements. This set is then converted back into a sorted list to allow for easy identification of subarrays with potential to be continuous. This is important for step 2, the sliding window approach.

nums = sorted(set(nums))

Initial Variables: The variable n stores the length of the original array. The variable ans is initialized to n, representing the worst-case scenario where all elements need to be replaced. We also initialize a variable j to 0, which will serve as our sliding window’s endpoint.

Sliding Window: We then use a sliding window to find the largest subarray where the elements can remain unchanged. To do this, we iterate over the sorted array with a variable i that represents the starting point of our subarray.

for i, v in enumerate(nums):

Inside the loop, j is incremented until the condition nums[j] - v <= n - 1 is no longer valid. This condition checks whether the subarray starting from i up to j can remain continuous if we were to fill in the numbers between nums[i] and nums[j].

while j < len(nums) and nums[j] - v <= n - 1:

j += 1

Calculating the Minimum: For each valid subarray, we calculate the number of elements that need to be replaced:

ans = min(ans, n - (j - i))

This calculates how many elements are not included in the largest potential continuous subarray and takes the minimum of the current answer and the number of elements outside the subarray. The difference n - (j - i) gives us the number of operations needed to fill in the missing numbers, since we skipped over n - (j - i) numbers to achieve the length n.

By the end of the loop, ans contains the minimum number of operations required to make the array continuous, which is then returned.

This implementation efficiently solves the problem using a sorted set for uniqueness and a sliding window to find the best subarray. The selected subarray has the most elements that are already part of a hypothetical continuous array, thus minimizing the required operations.

**Example Walkthrough**

Let's take the array nums = [4, 2, 5, 3, 5, 7, 6] as an example to illustrate the solution approach.

Sorting and Deduplication:

nums = sorted(set(nums)) // nums = [2, 3, 4, 5, 6, 7]

We first remove the duplicate number 5 and then sort the array. The array becomes [2, 3, 4, 5, 6, 7].

Initial Variables:

n = len(nums) // n = 7 (original array length)

ans = n // ans = 7

j = 0

Sliding Window: We iterate through the sorted and deduplicated array using a sliding window technique. The sliding window starts at each element i in nums and we try to expand the window by increasing j.

a. When i = 0 (nums[i] = 2):

nums[j] - nums[i] <= n - 1

nums[0] - 2 <= 6 // 0 - 2 <= 6, condition is true, try next

nums[1] - 2 <= 6 // 3 - 2 <= 6, condition is true, try next

nums[2] - 2 <= 6 // 4 - 2 <= 6, condition is true, try next

nums[3] - 2 <= 6 // 5 - 2 <= 6, condition is true, try next

nums[4] - 2 <= 6 // 6 - 2 <= 6, condition is true, try next

nums[5] - 2 <= 6 // 7 - 2 <= 6, condition is true

At this point, the subarray [2, 3, 4, 5, 6, 7] is the largest we can get starting from i = 0, without needing addition.

We calculate the operations needed for this subarray:

ans = min(ans, n - (j - i)) // ans = min(7, 7 - (5 - 0)) = 2

So, we need to replace 2 elements in the original array to make the subarray from 2 to 7 continuous.

b. The loop continues for i = 1 to i = 5, with the window size becoming smaller each time because the maximum possible value a continuous array can have also decreases.

By the end of the loop, we find that the minimum number of operations required is 2, which is the case when we consider the subarray [2, 3, 4, 5, 6, 7]. The two operations would involve replacing the two remaining numbers 4 and 5 (from the original nums) to get a continuous range that includes the largest possible number of the original elements.

Therefore, the answer for the example array is 2. This demonstrates how the approach uses a sliding window to minimize the number of operations needed to make the array continuous.

**Solution Implementation**

class Solution {

public int minOperations(int[] nums) {

// Sort the array to bring duplicates together and ease the operation count process

Arrays.sort(nums);

// Start uniqueNumbers counter at 1 since the first number is always unique

int uniqueNumbers = 1;

// Step through the sorted array and remove duplicates

for (int i = 1; i < nums.length; ++i) {

if (nums[i] != nums[i - 1]) {

nums[uniqueNumbers++] = nums[i];

}

}

// Initialize variable to track the minimum number of operations

int minOperations = nums.length;

// Use a sliding window to count the number of operations

for (int i = 0, j = 0; i < uniqueNumbers; ++i) {

// Expand the window to the right as long as the condition is met

while (j < uniqueNumbers && nums[j] - nums[i] <= nums.length - 1) {

++j;

}

// Calculate the minimum operations needed and store the result

minOperations = Math.min(minOperations, nums.length - (j - i));

}

// Return the minimum number of operations found

return minOperations;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the given code snippet involves several operations:

Sorting the unique elements in the array: This operation has a time complexity of O(k log k), where k is the number of unique elements in the array.

The for-loop runs k times, where k is the number of unique elements after removing duplicates.

Inside the for-loop, we have a while-loop; but notice that each element is visited at most once by the while-loop because j only increases. This implies the while-loop total times through all for-loop iterations is O(k).

Combining these complexities, we have a total time complexity of O(k log k + k), which simplifies to O(k log k) because k log k will dominate for larger k.

**Space Complexity**

The space complexity is determined by:

Storing the sorted unique elements, which takes O(k) space.

Miscellaneous variables (ans, j, n), which use constant space O(1).

Hence, the total space complexity is O(k) for storing the unique elements set. Note that k here represents the count of unique elements in the original nums list.

**Refer to chatGPT**

**Approach:**

The problem can be broken into the following steps:

Key Observations:

To make the array continuous, all numbers should fall within the smallest possible range of size equal to the length of the array (n).

Removing duplicates and using sorting can help identify the minimum range.

Sliding Window with Sorted Unique Values:

Extract unique numbers from nums, sort them, and treat them as potential candidates for forming a continuous sequence.

Use a sliding window to find the maximum number of unique elements that can fit in a contiguous range of size n.

Operations:

The operations needed are calculated as:
operations = n − maxElementsInRange

This involves replacing or removing elements outside the range.

class Solution {

public int minOperations(int[] nums) {

int n = nums.length;

// Step 1: Remove duplicates and sort the array

TreeSet<Integer> uniqueSet = new TreeSet<>();

for (int num : nums) {

uniqueSet.add(num);

}

List<Integer> uniqueNums = new ArrayList<>(uniqueSet);

// Step 2: Sliding window to find the maximum valid range

int maxElementsInRange = 0;

int left = 0;

for (int right = 0; right < uniqueNums.size(); right++) {

// Check if the range is valid

while (uniqueNums.get(right) - uniqueNums.get(left) >= n) {

left++;

}

// Update the maximum elements in a valid range

maxElementsInRange = Math.max(maxElementsInRange, right - left + 1);

}

// Step 3: Calculate the minimum operations

return n - maxElementsInRange;

}

}

**Explanation of Steps:**

Remove Duplicates and Sort:

Use a TreeSet to extract unique elements and sort them.

Sliding Window:

Traverse the sorted unique elements.

Maintain a sliding window where the range difference (uniqueNums[right] - uniqueNums[left]) is less than n.

This window gives the maximum number of elements that can be retained in a contiguous range.

Calculate Operations:

Subtract the size of the largest valid range from the array length to get the required operations.

**Need clarification on why removing duplicates not impact the minimum required operation number ?**

Removing duplicates does not impact the minimum required number of operations because duplicates in the array do not contribute to extending the contiguous range of distinct values. Here's a detailed explanation:

**Key Observations:**

Goal:

The goal is to form a contiguous range of distinct values of size nnn (the original array length).

Duplicates in the array are redundant and cannot help in achieving a larger contiguous range of unique values.

Impact of Duplicates:

Removing duplicates reduces the size of the candidate set to consider but does not change the minimum operations needed to achieve the desired result.

For any duplicate value, either:

It lies inside the final contiguous range: Its presence is unnecessary because only one instance of it is required.

It lies outside the final contiguous range: It will be replaced or removed anyway during the operation.

Example to Illustrate:

Suppose nums=[4,2,5,3,2,5]

After removing duplicates, we get [2,3,4,5]. This still captures all the unique values in nums.

The contiguous range of size 4 is already achievable, and duplicates like the second 2 and 5 add no value to this range.

The operations needed for numsnumsnums and the deduplicated version are the same.

**Why Removing Duplicates Is Safe:**

If duplicates were retained, they would not impact the result because:

They don't expand the range of unique values.

They are simply ignored when calculating how many elements fit into a valid range.

If duplicates were removed, the computation focuses on the unique numbers, which are sufficient to determine the operations needed.

**Conclusion:**

Removing duplicates does not change the result because duplicates do not contribute to extending the contiguous range of unique values. It only simplifies the problem, ensuring a more efficient calculation.

**Based on definition of 'continuous' nums**

**An array 'nums' is considered continuous if both of the following conditions are fulfilled:**

**(1) All elements in nums are unique.**

**(2) The difference between the maximum element and the minimum element in nums equals nums.length - 1.**

**Is that means the 'nums' array after sorting must be like below pattern ?**

**continuous nums = {a, a + 1, a + 2, ...., a + n - 1}**

Yes, based on the given definition of a "continuous" array, after sorting, the array must indeed follow the pattern:

{a, a + 1, a + 2,…, a + (n − 1)}

Here’s why:

**Key Points from the Definition:**

Unique Elements:

All elements in the array must be unique, meaning no duplicates are allowed.

Range Condition:

The difference between the maximum and minimum elements equals nums.length − 1.

This implies that the array spans exactly nnn consecutive integers, where nnn is the length of the array.

**Why This Implies the Pattern:**

Sorting the Array:

After sorting, the minimum element will be at the beginning (a) and the maximum element will be at the end (a + (n − 1).

Consecutive Numbers:

Since there are no duplicates and the range matches n−1, the sorted array must consist of consecutive integers:
a, a + 1, a + 2,…,a + (n − 1)

No Gaps:

The array contains exactly nnn elements, so there cannot be any gaps between the integers.

**Example:**

nums = [3,5,4,2]:

After sorting: [2,3,4,5]

Minimum: 2, Maximum: 5

Difference: 5 − 2 = 3

Length: n = 4, and difference = n − 1, so the array is continuous.

nums = [1,3,4,6]:

After sorting: [1,3,4,6]

Minimum: 1, Maximum: 6

Difference: 6 − 1 = 5

Length: n = 4, but difference ≠ n − 1, so the array is **not continuous**.

**Conclusion:**

Yes, for the array to be "continuous," it must follow the exact pattern {a, a + 1, a + 2, …, a + (n − 1)} after sorting.

**Solution 2: Binary Search + Find Target Last Occurrence (180 min)**

class Solution {

    public int minOperations(int[] nums) {

        int n = nums.length;

        // Remove duplicates and sort the array

        Arrays.sort(nums);

        int uniqueNumbers = 1;

        for(int i = 1; i < n; i++) {

            if(nums[i] != nums[i - 1]) {

                nums[uniqueNumbers++] = nums[i];

            }

        }

        int maxContinuousSubarrayLen = 0;

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

            int left = nums[i];

            int right = left + n - 1;

            // 1.After removing the duplicates the new right

            // boundary limitation is at index 'uniqueNumbers - 1'

            // 2.We actually want to find maximum 'right' because

            // maximum 'right' will result maximum 'j' and 'j - i'

            // will also be maximum and then n - (j - i) which

            // is the final result will be minimum, ideally we

            // should use 'Find Target Last Occurrence' to get 'right',

            // and since its all unique elements sorted array

            // 'nums' now, we either able to find the target value

            // directly or if not able to find only the insertion

            // position available, refer to L704

            int j = findTargetLastOccurrence(nums, uniqueNumbers - 1, right);

            maxContinuousSubarrayLen = Math.max(maxContinuousSubarrayLen, j - i);

        }

        // Calculate the minimum operations, besides maximum continuous

        // subarray length, all remain chars have to replace

        return n - maxContinuousSubarrayLen;

    }

    // Find Target Last Occurrence

    // The 'findTargetLastOccurrence' method try to find the right most

    // position or the insert position of given 'target', hence we use

    // 'boundary' parameter here limit the Binary Search applicable

    // range on sorted array 'nums' should between nums[0, boundary]

    private int findTargetLastOccurrence(int[] nums, int boundary, int target) {

        int lo = 0;

        int hi = boundary;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(nums[mid] > target) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // If not able to find 'target', then index 'lo'

        // is the insert position

        if(lo >= boundary || nums[lo] != target) {

            return lo;

        }

        // If able to find 'target', then index 'lo - 1'

        // is the last occurrence of the 'target', or just

        // return as 'hi'

        return lo - 1;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/minimum-number-of-operations-to-make-array-continuous/solutions/1470853/python-binary-search-clean-concise/>

**Idea**

Store the original length, n = len(nums).

Firstly, make elements in nums unique and sort nums array.

Try elements in nums as the start of the continuous array, let say start.

Elements in the continuous array must in range [start, end], where end = start + n - 1.

Binary search to find the index of the right insert position of end in nums, let say idx.

Then we can calculate the number of unique numbers in range [start, end] by uniqueLen = n - idx.

The cost to make coninuous array is cost = n - uniqueLen.

We update the best answer so far, ans = min(ans, cost)

Then return the best ans we have.

class Solution:

def minOperations(self, nums: List[int]) -> int:

n = len(nums)

nums = sorted(set(nums)) # Make `nums` as unique numbers and sort `nums`.

ans = n

for i, start in enumerate(nums):

end = start + n - 1 # We expect elements of continuous array must in range [start..end]

idx = bisect\_right(nums, end) # Find right insert position

uniqueLen = idx - i

ans = min(ans, n - uniqueLen)

return ans

**Refer to chatGPT**

**Key Idea:**

The goal is to find the smallest number of operations to transform the array into a "continuous" form, as defined earlier.

Instead of directly iterating through all possible transformations, we can use binary search to determine the largest number of unique elements that can fit within a valid range of size nnn, where nnn is the length of the array.

**Steps to Solve Using Binary Search:**

Sort and Remove Duplicates:

First, remove duplicates from the array to focus only on unique values. Sort the resulting array.

Binary Search for a Valid Range:

Use binary search on the sorted, unique array to find the maximum range of unique elements that fit into a continuous window of size nnn.

Count Operations:

Calculate the number of operations required to replace or remove the elements outside the valid range.

class Solution {

    public int minOperations(int[] nums) {

        int n = nums.length;

        // Step 1: Remove duplicates and sort

        TreeSet<Integer> uniqueSet = new TreeSet<>();

        for (int num : nums) {

            uniqueSet.add(num);

        }

        List<Integer> uniqueNums = new ArrayList<>(uniqueSet);

        // Step 2: Initialize binary search bounds

        int left = 0, right = uniqueNums.size() - 1, maxInRange = 0;

        // Step 3: Binary search to find the largest valid range

        while (left < uniqueNums.size()) {

            // Use binary search to find the farthest index with valid range

            int low = left, high = uniqueNums.size() - 1;

            while (low <= high) {

                int mid = low + (high - low) / 2;

                if (uniqueNums.get(mid) - uniqueNums.get(left) < n) {

                    maxInRange = Math.max(maxInRange, mid - left + 1);

                    low = mid + 1;

                } else {

                    high = mid - 1;

                }

            }

            left++;

        }

        // Step 4: Minimum operations = length of nums - maximum in range

        return n - maxInRange;

    }

}

**Explanation of Steps:**

Remove Duplicates and Sort:

Since duplicates do not contribute to expanding the range, removing them simplifies the problem. Sorting ensures the values are in increasing order.

Binary Search for Farthest Valid Index:

For each starting point (left), use binary search to find the farthest index (right) such that:
uniqueNums[right] − uniqueNums[left] < n

Update the maxInRange to store the maximum size of valid ranges found.

Count Operations:

The number of operations needed to make the array continuous is:
operations = n − maxInRange

**Refer to**

[L424.P2.6.Longest Repeating Character Replacement (Ref.L340)](note://1581773836A946879EB5237C85BFF6C2)

[L410.Split Array Largest Sum (Ref.L1011,L704,L1482)](note://WEB4d9d6f26a965381b137615128e2648be)

[L704.Binary Search](note://484532A5D2CB4B98A28218674438D5D8)