<https://leetcode.com/problems/apply-operations-to-maximize-frequency-score/description/>

You are given a **0-indexed** integer array nums and an integer k.

You can perform the following operation on the array **at most** k times:

Choose any index i from the array and **increase** or **decrease** nums[i] by 1.

The score of the final array is the **frequency** of the most frequent element in the array.

Return the **maximum** score you can achieve.

The frequency of an element is the number of occurences of that element in the array.

**Example 1:**

**Input:** nums = [1,2,6,4], k = 3

**Output:** 3

**Explanation:** We can do the following operations on the array:

- Choose i = 0, and increase the value of nums[0] by 1. The resulting array is [2,2,6,4].

- Choose i = 3, and decrease the value of nums[3] by 1. The resulting array is [2,2,6,3].

- Choose i = 3, and decrease the value of nums[3] by 1. The resulting array is [2,2,6,2].

The element 2 is the most frequent in the final array so our score is 3.

It can be shown that we cannot achieve a better score.

**Example 2:**

**Input:** nums = [1,4,4,2,4], k = 0

**Output:** 3

**Explanation:** We cannot apply any operations so our score will be the frequency of the most frequent element in the original array, which is 3.

**Constraints:**

1 <= nums.length <= 10^5

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

0 <= k <= 10^14

**Attempt 1: 2024-12-28**

**Solution 1: Math + Sorting + Binary Search (180 min)**

class Solution {

    public int maxFrequencyScore(int[] nums, long k) {

        Arrays.sort(nums);

        int len = nums.length;

        long[] presum = new long[len + 1];

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

            presum[i] = presum[i - 1] + (long) nums[i - 1];

        }

        // 'lo' and 'hi' represent target frequency of the

        // most frequent element in the array, the minimum

        // potential target frequency is 0, the maximum

        // potential target frequency is array length, we

        // keep searching for the middle value based on

        // minimum and maximum target frequency boundary

        // and check if a subarray of length equal to this

        // middle value can be transformed such that all of

        // its elements become the median of this subarray

        // with a total operation cost that doesn't exceed k

        int lo = 0;

        int hi = len;

        // Find upper boundary

        while(lo <= hi) {

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

            // Since we try to find the maximum score you can achieve,

            // if able to find with element frequency(subarray length)

            // equal to 'mid', then we move forward 'lo' to 'mid + 1'

            // attempt on larger element frequency, otherwise move

            // backward 'hi' to 'mid - 1' attempt on smaller element

            // frequency

            if(isPossible(nums, presum, k, mid)) {

                lo = mid + 1;

            } else {

                hi = mid - 1;

            }

        }

        return lo - 1;

    }

    private boolean isPossible(int[] nums, long[] presum, long k, int subarrayLen) {

        boolean possible = false;

        // Try for all subarrays of size 'subarrayLen'

        for(int i = 0; i <= nums.length - subarrayLen; i++) {

            int j = i + subarrayLen;

            // The target number is the median number of the current subarray(sorted)

            int targetNum = nums[(i + j) / 2];

            // Calculate how much we need to add to the left half of the subarray

            // to make it all equal to targetNum

            long leftCost = ((i + j) / 2 - i) \* (long) targetNum - (presum[(i + j) / 2] - presum[i]);

            // Similarly, calculate the cost for right half of the subarray to make

            // it all equal to targetNum

            long rightCost = (presum[j] - presum[(i + j) / 2]) - ((j - (i + j) / 2) \* (long) targetNum);

            // If total operations needed are less or equal to k, it's possible to make all elements

            // of the subarray equal to targetNum

            if(leftCost + rightCost <= k) {

                return true;

            }

        }

        return false;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(n)

**Refer to**

<https://algo.monster/liteproblems/2968>

**Problem Description**

You are given an array of integers nums and an integer k. The array is 0-indexed, meaning that the first element is at position 0. You have the opportunity to perform an operation up to k times on the array. In each operation, you can choose any element in the array and either increase or decrease its value by 1. The goal is to maximize the "score" of the final array, where the score is defined as the frequency of the most frequent element within the array.

To clarify, the frequency of an element is how many times that element appears in the array. You want to make adjustments to the array such that the most frequent element appears as many times as possible, without exceeding the limit of k operations.

The problem asks you to find out the maximum score that could be achieved under these conditions.

**Intuition**

To solve this problem, one needs to grasp the idea that if we organize the numbers in a particular order, we can efficiently change a segment of numbers to be the same value, which is a strategic way to maximize the frequency of a single number. [Sorting](https://algo.monster/problems/sorting_summary)**the array nums is a good start because it allows us to focus on transforming a contiguous subsequence into a single, repeated number with fewer operations.**

**Since increasing or decreasing elements by 1 count as a single operation each time, it's preferable to turn a sequence of consecutive numbers into one number—the median of the chosen sequence. This is because the median minimizes the total distance (and thus the total number of operations) needed to make all elements in the range equal to it.**

Considering this, if achieving a frequency of x is possible within the k operations, then any frequency less than x should also be possible. This suggests that the relationship between frequency and feasibility is monotonic, which lends itself nicely to a [binary search](https://algo.monster/problems/binary-search-speedrun) approach to find the maximum feasible frequency.

For the [binary search](https://algo.monster/problems/binary-search-speedrun), we define the low boundary l as 0 and the high boundary r as the length of the array n. **We keep searching for the middle value mid and check if a subarray of length mid can be transformed such that all of its elements become the median of this subarray with a total operation cost that doesn't exceed k.**

To quickly calculate the operation cost for transforming a subarray into a single number, we take advantage of prefix sums—a cumulative sum of the elements in the array that allows us to compute the sum of a subarray in constant time. With two pointers marking the boundaries of the candidate subarray and the [prefix sum](https://algo.monster/problems/subarray_sum) array, we can efficiently compute the operation cost to transform the subarray elements to the median and check against our operations budget k.

In summary, by [sorting](https://algo.monster/problems/sorting_summary) the array, leveraging the concept of the median, and applying [binary search](https://algo.monster/problems/binary-search-speedrun) alongside prefix sums, we can pinpoint the maximum frequency (score) that we can achieve within our operational constraints.

**Solution Approach**

The solution implements a combination of [sorting](https://algo.monster/problems/sorting_summary), [prefix sum](https://algo.monster/problems/subarray_sum) computation, and [binary search](https://algo.monster/problems/binary-search-speedrun) techniques to achieve the desired outcome. Here's how each component contributes to the final solution:

**Sorting:** To deal with a range of elements efficiently, the array nums is first sorted. By sorting the array, we ensure that any segment we pick can be changed to the median value of that segment with minimal operations. This is because the median value minimizes the sum of absolute differences to all other elements in a sorted sequence.

**Prefix Sum:** We employ a prefix sum array s to enable quick calculations of the sum of numbers within any subarray [i, j] of our sorted array. The prefix sum aids in computing the operation cost for two halves of the segment: the left half where we subtract the median, and the right half where we add to the median. It's crucial for evaluating whether we can convert a subarray of a certain size to the same number within k operations.

**Binary Search:** Knowing that the problem has a monotonic property (if a certain frequency is possible then all lesser frequencies are also possible), we use binary search to find the maximum frequency that is feasible. We define the low boundary l and the high boundary r and search in the middle mid. If a subarray of length mid can be converted to a single number with less than k operations, we can potentially achieve a higher frequency and thus move our lower boundary up; otherwise, we lower our upper boundary.

Here's the practical application of these concepts in the code:

We start with a sorted array and initiate a [prefix sum](https://algo.monster/problems/subarray_sum) s.

**We then use a** [binary search](https://algo.monster/problems/binary-search-speedrun), **where within each iteration we set a mid value. For each potential mid (which denotes a subarray size), we iterate through the array to check if there's a segment of size mid that can be turned into a single valued sequence, with the median being the target value.**

For each subarray [i, j] considered, left is calculated as the sum of operations needed to decrease each element to the median, and right is calculated as the sum of operations needed to increase each element to the median.

The left sum is calculated using the formula: ((i + j) / 2 - i) \* nums[(i + j) / 2] - (s[(i + j) / 2] - s[i]).

The right sum is calculated using the formula: (s[j] - s[(i + j) / 2]) - ((j - (i + j) / 2) \* nums[(i + j) / 2]).

If the sum of left and right is within the k operation limit, it means that a subarray of length mid can indeed be transformed to have the same value; thus, we know we can achieve at least this frequency—and maybe more. So, the [binary search](https://algo.monster/problems/binary-search-speedrun) will continue to look for a larger subarray that also satisfies the condition, until it finds the largest feasible subarray size before exceeding the k operations limit.

The result l, which is maintained by the [binary search](https://algo.monster/problems/binary-search-speedrun), will hold the maximum frequency score that could be achieved once the search completes.

By intertwining these algorithms and computational strategies, the solution efficiently navigates through the possibilities of subarray sizes to find the one that yields the maximum frequency within the provided constraints.

**Solution Implementation**

class Solution {

public int maxFrequencyScore(int[] nums, long k) {

// Sort the array first.

Arrays.sort(nums);

int length = nums.length;

// Initialize a prefix sum array with an extra space for ease of calculations.

long[] prefixSum = new long[length + 1];

// Populate the prefix sum array.

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

prefixSum[i] = prefixSum[i - 1] + nums[i - 1];

}

// Initialize the binary search boundaries.

int left = 0, right = length;

// Perform binary search.

while (left < right) {

int middle = (left + right + 1) >> 1;

boolean isPossible = false;

// Try for all subarrays of size 'middle'

for (int i = 0; i <= length - middle; i++) {

int j = i + middle;

// Choose the middle element as the target number to which other elements will be increased.

int targetNum = nums[(i + j) / 2];

// Calculate how much we need to add to the left half of the subarray to make it all equal to targetNum.

long costLeft = ((i + j) / 2 - i) \* (long) targetNum - (prefixSum[(i + j) / 2] - prefixSum[i]);

// Similarly, calculate the cost for right half of the subarray.

long costRight = (prefixSum[j] - prefixSum[(i + j) / 2]) - ((j - (i + j) / 2) \* (long) targetNum);

// If total operations needed are less or equal to k, it's possible to make all elements of the subarray equal.

if (costLeft + costRight <= k) {

isPossible = true;

break;

}

}

// If it's possible to make the elements of a subarray equal with at most k operations, move left boundary up.

if (isPossible) {

left = middle;

} else {

// Otherwise, reduce the size of the subarray.

right = middle - 1;

}

}

// The maximum frequency is the size of the largest subarray for which

// we can make all the elements equal with at most k operations.

return left;

}

}

**Time and Space Complexity**

The time complexity of the code provided is O(n log n + n log n). This is because there are two main parts to the algorithm contributing to the time complexity:

nums.sort() takes O(n log n) time to sort the array.

The while loop runs for O(log n) iterations (binary search on the size of the frequency), and within each iteration, there is a for loop that could run up to n times dependent on the value of mid. The operations inside the for loop are constant time, therefore the time complexity for this part is O(n log n).

The space complexity of the code is O(n). This stems from the auxiliary space used to store cumulative sums of the array nums in the list s. Since s has the same length as nums, it takes up O(n) space. No other data structures in the code use space that scales with the input size.

**Refer to**

<https://leetcode.com/problems/apply-operations-to-maximize-frequency-score/solutions/4414735/sorting-binary-search-sliding-window-prefix-sum/>

**Intuition**

Simple Sliding Window + Binary Search

Apply the Binary Search on Answer technique.

Minimum possible freq: 1 and maxPossible Freq: nums.length

leftFreq = 1, rightFreq = nums.length;

Apply binary search on this range and check the feasibility of frequency equal to midFreq where midFreq is leftFreq + (rightFreq - leftFreq) / 2;

if(possible(midFreq)) {

// try to increase the frequency by moving leftFreq to midFreq + 1

leftFreq = midFreq - 1;

} else {

rightFreq = midFreq - 1;

}

In order to check the feasibility, apply the fixed size sliding window of length midFreq to check whether numbers in that window can be converted to the median

of that fixed range or not while maxOperations allowed are k?

In order to do this efficiently we can apply the concept of prefix sum to quickly calculate the number of operations required to convert every number to the median element.

class Solution {

private boolean isPossible(int length, int[] nums, long k, long[] prefixSum) {

int leftIdx = 0, rightIdx = 0;

while(rightIdx < nums.length) {

if(rightIdx - leftIdx + 1 < length) {

rightIdx++;

continue;

}

int medianIdx = leftIdx + ((rightIdx - leftIdx) / 2);

long curOperations = (prefixSum[rightIdx + 1] - prefixSum[medianIdx + 1]);

if(medianIdx > leftIdx) {

curOperations -= prefixSum[medianIdx] - prefixSum[leftIdx];

}

if((rightIdx - leftIdx + 1) % 2 == 0) {

curOperations -= nums[medianIdx];

}

if(curOperations <= k) {

return true;

}

leftIdx++;

rightIdx++;

}

return false;

}

public int maxFrequencyScore(int[] nums, long k) {

if(nums.length == 1) {

return 1;

}

Arrays.sort(nums);

long[] prefixSum = new long[nums.length + 1];

for(int idx = 0; idx < nums.length; idx++) {

prefixSum[idx + 1] = prefixSum[idx] + nums[idx];

}

int leftFreq = 2, rightFreq = nums.length;

while(leftFreq <= rightFreq) {

int midFreq = leftFreq + ((rightFreq - leftFreq) >> 1);

if(isPossible(midFreq, nums, k, prefixSum)) {

leftFreq = midFreq + 1;

} else {

rightFreq = midFreq - 1;

}

}

return rightFreq;

}

}

**Solution 2: Sliding Window (60 min)**

class Solution {

    public int maxFrequencyScore(int[] nums, long k) {

        Arrays.sort(nums);

        int result = 0;

        int i = 0;

        long cur = 0;

        for(int j = 0; j < nums.length; j++) {

            // Add the cost of making the new element nums[j] equal

            // to the median of the current window (nums[(i + j) / 2]).

            cur += nums[j] - nums[(i + j) / 2];

            while(cur > k) {

                // Subtract the contribution of nums[i] from the cost, and adjust

                // the median to the next valid value (nums[(i + j + 1) / 2]).

                cur -= nums[(i + j + 1) / 2] - nums[i];

                i++;

            }

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

        }

        return result;

    }

}

**Refer to chatGPT**

**Key Observations:**

Sorting the Array:

The array A is sorted to ensure that smaller numbers are incremented to match larger ones. This minimizes the cost of operations.

Using the Median:

For any window, the **median** minimizes the sum of absolute differences between the elements and a chosen value. By centering the cost calculation around the median, the implementation reduces the complexity of cost management.

Cost Calculation:

The cur variable keeps track of the "cost" to make all elements in the current window equal to the median.

For the window A[i:j+1], the **median** is:

A[(i+j)/2] when j - i is even.

A[(i+j+1)/2] when j - i is odd.

**Sliding Window Logic:**

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

cur += A[j] - A[(i + j) / 2];

Expand the window by adding A[j] to the window.

Cost Update: Add the cost of making the new element A[j] equal to the median of the current window (A[(i + j) / 2]).

**Shrinking the Window:**

while (cur > k) {

cur -= A[(i + j + 1) / 2] - A[i];

i++;

}

If the current cost exceeds k, shrink the window by incrementing the left pointer i.

Cost Adjustment: Subtract the contribution of A[i] from the cost, and adjust the median to the next valid value (A[(i + j + 1) / 2]).

**Key Insights:**

Median-Based Cost Management:

**The choice of the median as the target value for all elements in the window minimizes the total cost of adjustments. This is why the cost is computed as:
cur+=A[j]−A[median]**

Dynamic Median Adjustment:

**When the left pointer i is moved, the median shifts dynamically to the next element. The adjustment for cur ensures that the cost is recalculated based on the new median:
cur−=A[new median]−A[i]**

**Refer to**

[L1703.Minimum Adjacent Swaps for K Consecutive Ones (Ref.L2968)](note://WEB69292b54d5c37f5290b7b5196a3c8139)

[L1838.Frequency of the Most Frequent Element (Ref.L2968)](note://WEBe75023aadd133f6c8d91ff09f6ec8794)

[L2448.Minimum Cost to Make Array Equal (Ref.L2968)](note://WEBf2a30b725c21a5e1b234b381454bc744)

[L2967.Minimum Cost to Make Array Equalindromic (Ref.L2968)](note://WEBc3d8f1268b7c828f000f00d80a79ef9c)