<https://leetcode.com/problems/frequency-of-the-most-frequent-element/description/>

The **frequency** of an element is the number of times it occurs in an array.

You are given an integer array nums and an integer k. In one operation, you can choose an index of nums and increment the element at that index by 1.

Return the **maximum possible frequency** of an element after performing **at most** k operations.

**Example 1:**

**Input:** nums = [1,2,4], k = 5

**Output:** 3

**Explanation:** Increment the first element three times and the second element two times to make nums = [4,4,4].4 has a frequency of 3.

**Example 2:**

**Input:** nums = [1,4,8,13], k = 5

**Output:** 2

**Explanation:** There are multiple optimal solutions:

- Increment the first element three times to make nums = [4,4,8,13]. 4 has a frequency of 2.

- Increment the second element four times to make nums = [1,8,8,13]. 8 has a frequency of 2.

- Increment the third element five times to make nums = [1,4,13,13]. 13 has a frequency of 2.

**Example 3:**

**Input:** nums = [3,9,6], k = 2

**Output:** 1

**Constraints:**

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^5

1 <= k <= 10^5

**Attempt 1: 2024-12-29**

**Solution 1: Not fixed length Sliding Window (10 min)**

**Style 1: Use 'while' loop**

class Solution {

    public int maxFrequency(int[] nums, int k) {

        int max = 0;

        Arrays.sort(nums);

        long sum = 0;

        int i = 0;

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

            sum += nums[j];

            while((long) nums[j] \* (j - i + 1) - sum > k) {

                sum -= nums[i];

                i++;

            }

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

        }

        return max;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Style 2: Use 'if' instead of 'while' loop**

class Solution {

    public int maxFrequency(int[] nums, int k) {

        int max = 0;

        Arrays.sort(nums);

        long sum = 0;

        int i = 0;

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

            sum += nums[j];

            if((long) nums[j] \* (j - i + 1) - sum > k) {

                sum -= nums[i];

                i++;

            }

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

        }

        return max;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/frequency-of-the-most-frequent-element/solutions/1175090/java-c-python-sliding-window/>

**Intuition**

Sort the input array A

Sliding window prolem actually, the key is to find out the valid condition:

k + sum >= size \* max

which is

k + sum >= (j - i + 1) \* A[j]

**Explanation**

For every new element A[j] to the sliding window,

Add it to the sum by sum += A[j].

Check if it'a valid window by

sum + k < (long)A[j] \* (j - i + 1)

If not, removing A[i] from the window by

sum -= A[i] and i += 1.

Then update the res by res = max(res, j - i + 1).

I added solution 1 for clearly expain the process above.

Don't forget to handle the overflow cases by using long

**Solution 1: Use while loop**

public int maxFrequency(int[] A, int k) {

int res = 1, i = 0, j;

long sum = 0;

Arrays.sort(A);

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

sum += A[j];

while (sum + k < (long)A[j] \* (j - i + 1)) {

sum -= A[i];

i += 1;

}

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

}

return res;

}

**Solution 2: Use if clause**

**Just save some lines and improve a little time.**

public int maxFrequency(int[] A, long k) {

int i = 0, j;

Arrays.sort(A);

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

k += A[j];

if (k < (long)A[j] \* (j - i + 1))

k -= A[i++];

}

return j - i;

}

**Refer to**

<https://leetcode.com/problems/frequency-of-the-most-frequent-element/editorial/>

**Approach 1: Sliding Window**

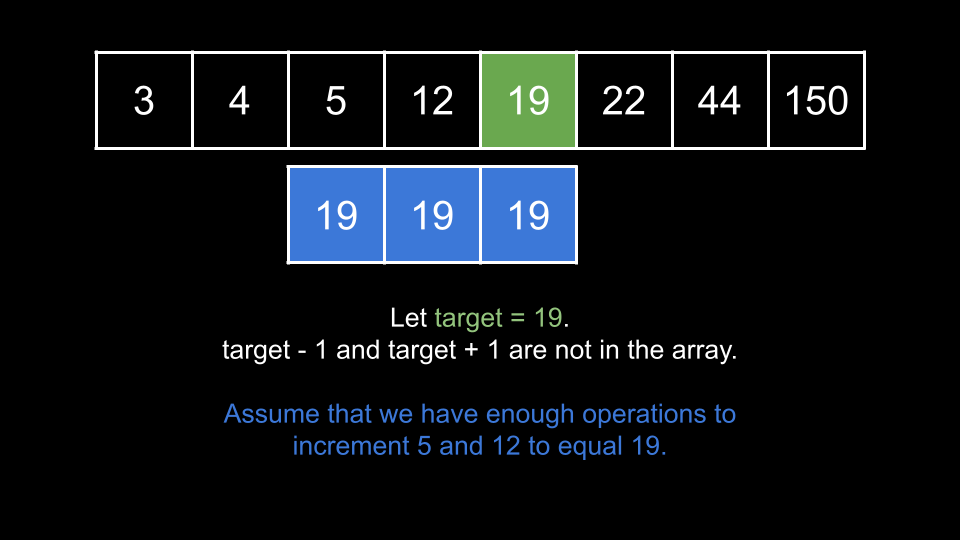
**Intuition**

In this problem, we want to make as many elements as we can equal using k increments.

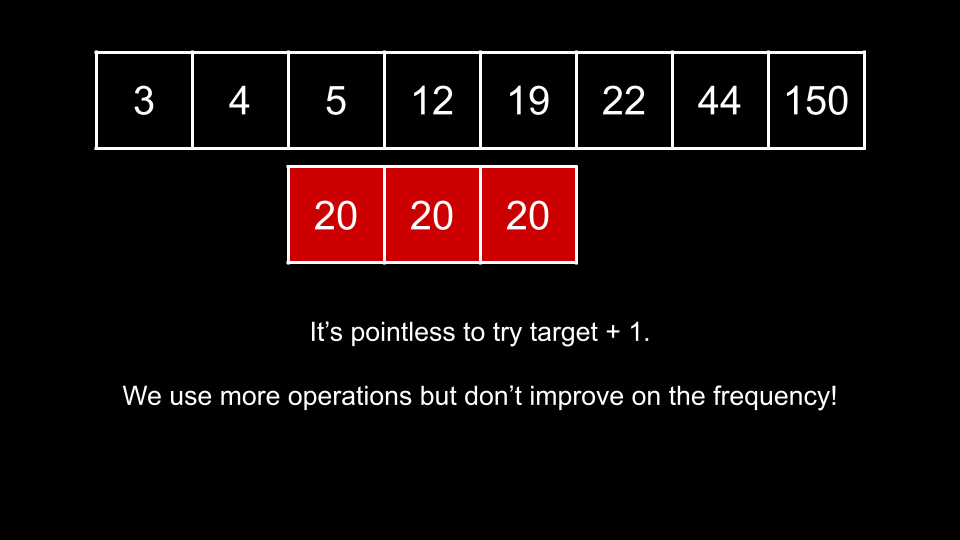
Let's say that we choose a number target and want to maximize its frequency. Intuitively, the elements that we would increment would be the elements that are closest to target (and less than target, since we can only increment).

So what number should we choose for target? The optimal target will already exist in the array. Why?

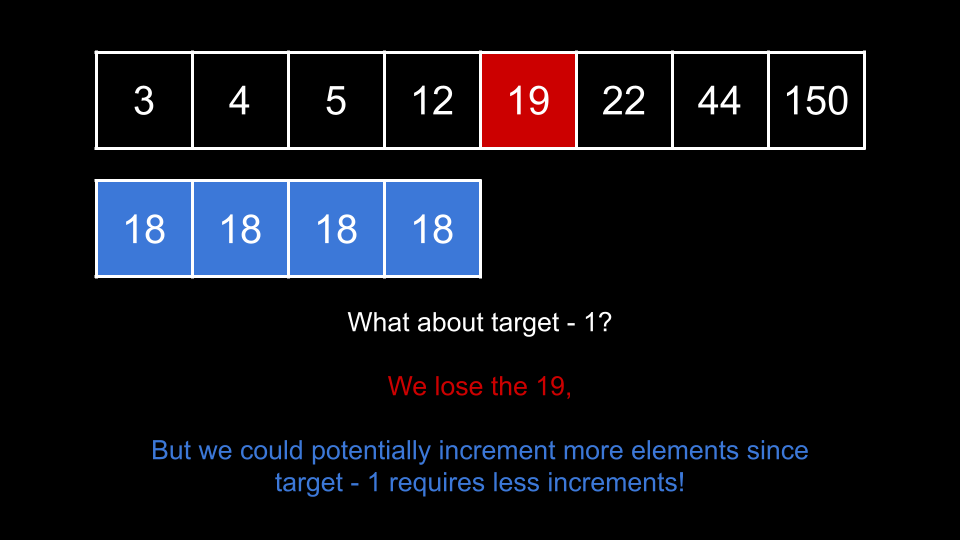
Assume target is in nums, but target - 1 and target + 1 are not in nums. Let's say that we can increment x elements to be equal to target using at most k operations. We will prove that making target - 1 or target + 1 the most frequent element does not lead to better results.



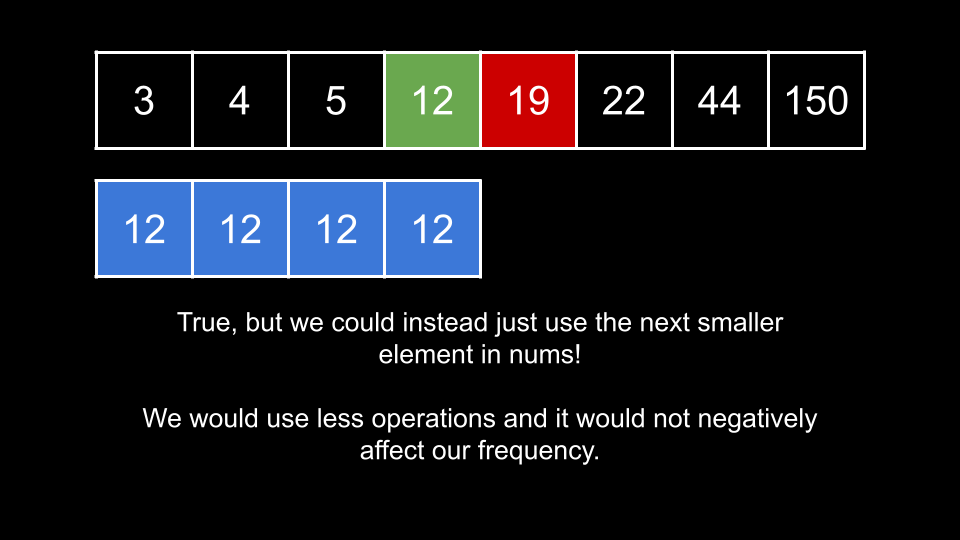
It would be pointless to instead try to make target + 1 the most frequent element, since this would cost us x extra operations and we would not improve on our answer. The same goes for even larger elements target + 2 and etc.



What about target - 1? Compared with making target the most frequent element, we would lose the values representing these targets from our max frequency, but we would save x operations which we could potentially use to increment more than one extra element and thus improve our answer.



The above statement is true, but meaningless! Consider the greatest element in nums that is less than target. That is, if we were to sort nums, consider the element that comes right before target. If we were to instead consider this element as the target, we would save more than x operations without negatively affecting the frequency relative to considering target - 1.



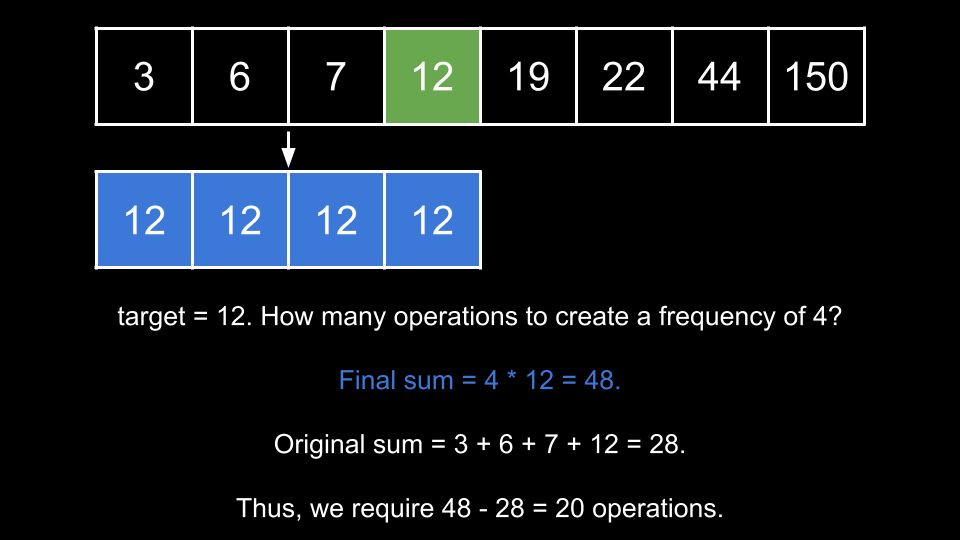
**In summary, for any given number absent that is not in nums, consider the greatest number in nums smaller than absent as smaller Target. The number of operations to raise some number of elements to smaller Target will always be less than the number of steps needed to raise them to absent.**

**Thus, the optimal value of target must exist in nums. We can iterate over nums and consider each element as target.**

For a given value of target, how can we efficiently check the frequency we could achieve? As we mentioned at the start, we would want to increment elements that are closest to target. As such, we will start by sorting nums so that as we iterate over the elements, we know the elements closest to target are just to the left of target.

Now that nums is sorted, consider the first element to the left of target as smaller. As smaller is the closest element to target, we want to increment it to equal target. This will cost us target - smaller operations. Now, consider the next element to the left as smaller2. Now this is the element closest to target, so we increment it using target - smaller2 operations. We continue this process until we run out of operations.

As you can see, the number of operations required is simply the difference between target and the numbers we are incrementing. Let's say that the final frequency of target was 4. We would have a sum of 4 \* target. The number of operations would be this sum minus the sum of the elements before we incremented them. Consider the following example:



This brings us to our solution. We will use a sliding window over the sorted nums. For each element nums[right], we will treat target as this element and try to make every element in our window equal to target.

The size of the window is right - left + 1. That means we would have a final sum of (right - left + 1) \* target. If we track the sum of our window in a variable curr, then we can calculate the required operations as (right - left + 1) \* target - curr. If it requires more than k operations, we must shrink our window. Like in all sliding window problems, we will use a while loop to shrink our window by incrementing left until k operations are sufficient.

Once the while loop ends, we know that we can make all elements in the window equal to target. We can now update our answer with the current window size. The final answer will be the largest valid window we find after iterating right over the entire input.

**Algorithm**

**Algorithm**

Sort nums.

Initialize the following integers:

left = 0, the left pointer.

ans = 0, the best answer we have seen so far.

curr = 0, the sum of the elements currently in our window.

Iterate right over the indices of nums:

Consider target = nums[right].

Add target to curr.

While the size of the window right - left + 1 multiplied by target, minus curr is greater than k:

Subtract nums[left] from curr.

Increment left.

Update ans with the current window size if it is larger.

Return ans.

**Implementation**

Be careful! Given the constraints, we may run into integer overflow. Use long accordingly in Java and C++ (Python doesn't have overflow).

class Solution {

public int maxFrequency(int[] nums, int k) {

Arrays.sort(nums);

int left = 0;

int ans = 0;

long curr = 0;

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

long target = nums[right];

curr += target;

while ((right - left + 1) \* target - curr > k) {

curr -= nums[left];

left++;

}

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

}

return ans;

}

}

**Complexity Analysis**

Given n as the length of nums,

**Time complexity: O(n⋅logn)**

Despite the while loop, each iteration of the for loop is amortized O(1). The while loop only runs O(n) times across all iterations. This is because each iteration of the while loop increments left. As left can only increase and cannot exceed n, the while loop never performs more than n iterations total. This means the sliding window process runs in O(n).

However, we need to sort the array, which costs O(n⋅logn).

**Space Complexity: O(logn) or O(n)**

We only use a few integer variables, but some space is used to sort.

The space complexity of the sorting algorithm depends on the implementation of each programming language:

In Java, Arrays.sort() for primitives is implemented using a variant of the Quick Sort algorithm, which has a space complexity of O(logn)

In C++, the sort() function provided by STL uses a hybrid of Quick Sort, Heap Sort and Insertion Sort, with a worst case space complexity of O(logn)

In Python, the sort() function is implemented using the Timsort algorithm, which has a worst-case space complexity of O(n)

**Approach 2: Advanced Sliding Window**

**Intuition**

This approach is an extension of the previous one.

Notice that the only thing we care about is the length of the longest window. We don't need to know what the window itself is. As we slide the window over the array, let's say we find a valid window with a length of len. **We no longer care about any windows with lengths less than len, because they could not possibly improve on our answer.**

**The purpose of the while loop in the previous approach is to shrink the window until it is valid again. In this approach, we will not shrink the window - we will just try to grow it as large as we can.**

We will keep the same condition in the while loop that checks if the current window [left, right] is valid, but instead of using a while loop, we will just use an if statement. This means left never increases by more than 1 per iteration. Because right also increases by 1 per iteration, if we cannot find a valid window, we will simply be sliding a window with static size across the array.

However, if we add an element nums[right] to the window and the window is valid, then the if statement will not trigger, and left will not be incremented. Thus, we will increase our window size by 1. In this scenario, it implies the current window [left, right] is the best window we have seen so far.

As you can see, it is actually impossible for our window size to decrease, since each iteration increases right by 1 and left by either 0 or 1.

Because our window size cannot decrease, it also means that the size of the window always represents the length of the best window we have found so far - analogous to ans from the previous approach.

At the end of the iteration, the size of our window is n - left. We return this as the answer.

**Algorithm**

Sort nums.

Initialize the following integers:

left = 0, the left pointer.

curr = 0, the sum of the elements currently in our window.

Iterate right over the indices of nums:

Consider target = nums[right].

Add target to curr.

If the size of the window right - left + 1 multiplied by target, minus curr is greater than k:

Subtract nums[left] from curr.

Increment left.

Return nums.length - left.

class Solution {

public int maxFrequency(int[] nums, int k) {

Arrays.sort(nums);

int left = 0;

long curr = 0;

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

long target = nums[right];

curr += target;

if ((right - left + 1) \* target - curr > k) {

curr -= nums[left];

left++;

}

}

return nums.length - left;

}

}

**Complexity Analysis**

Given n as the length of nums,

**Time complexity: O(n⋅logn)**

Each iteration of the for loop costs O(1). This means the sliding window process runs in O(n).

However, we need to sort the array, which costs O(n⋅logn).

**Space Complexity: O(logn) or O(n)**

We only use a few integer variables, but some space is used to sort.

The space complexity of the sorting algorithm depends on the implementation of each programming language:

In Java, Arrays.sort() for primitives is implemented using a variant of the Quick Sort algorithm, which has a space complexity of O(logn)

In C++, the sort() function provided by STL uses a hybrid of Quick Sort, Heap Sort and Insertion Sort, with a worst case space complexity of O(logn)

In Python, the sort() function is implemented using the Timsort algorithm, which has a worst-case space complexity of O(n)

**Solution 2: Binary Search (60 min, similar to** [L2968.Apply Operations to Maximize Frequency Score](note://WEB45984b791ce0bb0f0a1224f1eb84df43)**)**

**Wrong Solution**

**Test by**

**nums = [1,2,4], k = 5**

**Output = 2, Expected = 3**

class Solution {

    public int maxFrequency(int[] nums, int 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];

        }

        int max = 0;

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

            max = Math.max(max, binarySearch(nums, presum, k, i));

        }

        return max;

    }

    private int binarySearch(int[] nums, long[] presum, int k, int rightBoundary) {

        int lo = 0;

        int hi = rightBoundary;

        // We want maximum 'rightBoundary - lo + 1' hence we want minimum 'lo'

        // Find lower boundary

        while(lo <= hi) {

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

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

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // Here we will get 'lo' as minimum value since using

        // Find lower boundary strategy, then the maximum

        // subarray length (aka 'frequency') we can get equals

        // to 'rightBoundary - lo + 1'

        return rightBoundary - lo + 1;

    }

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

        long total\_ops = (long) nums[rightBoundary] \* (rightBoundary - leftBoundary + 1);

        long original\_ops = presum[rightBoundary] - presum[leftBoundary + 1];

        return total\_ops - original\_ops <= k;

    }

}

**Correct Solution**

class Solution {

    public int maxFrequency(int[] nums, int 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];

        }

        int max = 0;

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

            max = Math.max(max, binarySearch(nums, presum, k, i));

        }

        return max;

    }

    private int binarySearch(int[] nums, long[] presum, int k, int rightBoundary) {

        int lo = 0;

        int hi = rightBoundary;

        // We want maximum 'rightBoundary - lo + 1' hence we want minimum 'lo'

        // Find lower boundary

        while(lo <= hi) {

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

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

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // Here we will get 'lo' as minimum value since using

        // Find lower boundary strategy, then the maximum

        // subarray length (aka 'frequency') we can get equals

        // to 'rightBoundary - lo + 1'

        return rightBoundary - lo + 1;

    }

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

        long total\_ops = (long) nums[rightBoundary] \* (rightBoundary - leftBoundary + 1);

        // Indexing Misalignment:

        // The passed in 'rightBoundary' originally define range as [0, len - 1],

        // the index system is based on 'nums', but here we have to work with 'presum'

        // which has valid index range 1 more than 'nums' as [0, len]

        // If we only following index system based on 'nums' and pass in index as

        // 'rightBoundary' (based on 'nums' index system) for 'presum',

        // presum[rightBoundary] only gives the sum of elements up to

        // nums[rightBoundary - 1], but we need the sum up to nums[rightBoundary],

        // hence we have to manually increase 1 on this 'nums' based index to

        // align with the 'presum' index system

        // Mapping relation between 'presum' and 'nums' if keep original define range:

        // presum[1] = nums[0]

        // presum[2] = nums[0] + nums[1]

        // ...

        // presum[rightBoundary] = nums[0] + nums[1] + .. + nums[rightBoundary - 1]

        // ... etc.

        // Only '+ 1' will include nums[rightBoundary]:

        // presum[rightBoundary + 1] = nums[0] + nums[1] + .. + nums[rightBoundary - 1] + nums[rightBoundary]

        // Now

        // If we want sum up of 'nums[leftBoundary] + ... + nums[rightBoundary]'

        // it mapping to 'presum[rightBoundary + 1] - presum[leftBoundary]',

        // The prefix sum presum[rightBoundary + 1] gives the sum up to and including

        // nums[rightBoundary], and presum[leftBoundary] correctly excludes elements

        // before nums[leftBoundary]

        long original\_ops = presum[rightBoundary + 1] - presum[leftBoundary];

        return total\_ops - original\_ops <= k;

    }

}

Time Complexity: O(n \* logn)

Space Complexity: O(n)

**How the correct solution works ?**

**Refer to chatGPT**

The difference between the incorrect and correct versions of the isPossible() method lies in how the prefix sum (presum) is indexed and how the range of numbers is summed.

**Issue in the Incorrect Solution**

long original\_ops = presum[rightBoundary] - presum[leftBoundary + 1];

**Indexing Misalignment:** presum[rightBoundary] gives the sum of elements up to nums[rightBoundary - 1], but you need the sum up to nums[rightBoundary].

Similarly, presum[leftBoundary + 1] gives the sum up to nums[leftBoundary] rather than starting from it.

This misalignment causes the range sum calculation to be incorrect, leading to the wrong result.

**Fix in the Correct Solution**

long original\_ops = presum[rightBoundary + 1] - presum[leftBoundary];

**Correct Range Calculation:** The prefix sum presum[rightBoundary + 1] gives the sum up to and including nums[rightBoundary], and presum[leftBoundary] correctly excludes elements before nums[leftBoundary].

This ensures that the range [leftBoundary, rightBoundary] is summed correctly.

**Why This Fix Works**

In the correct formula:

total\_ops calculates the total operations needed to make all elements in the range [leftBoundary, rightBoundary] equal to nums[rightBoundary].

This assumes you are incrementing smaller numbers in the range to match nums[rightBoundary].

original\_ops calculates the current sum of elements in the range [leftBoundary, rightBoundary] using the corrected prefix sum indices.

Subtracting original\_ops from total\_ops gives the operations needed to make all elements in the range equal. If this is ≤ k, the subarray can be transformed within the allowed operation cost.

**Concrete Example**

**Input:**

nums = [1, 2, 4], k = 5

**Steps**:

Sorted nums: [1, 2, 4]

Prefix Sum (presum): [0, 1, 3, 7]

**Incorrect Calculation** for Subarray [1, 2] (leftBoundary = 0, rightBoundary = 1):

total\_ops = nums[1] \* (1 - 0 + 1) = 2 \* 2 = 4

original\_ops = presum[1] - presum[1] = 1 - 3 = -2 (incorrect due to indexing)

total\_ops - original\_ops = 4 - (-2) = 6 → Exceeds k = 5.

**Correct Calculation** for Subarray [1, 2]:

total\_ops = nums[1] \* (1 - 0 + 1) = 2 \* 2 = 4

original\_ops = presum[2] - presum[0] = 3 - 0 = 3 (correct indexing)

total\_ops - original\_ops = 4 - 3 = 1 → Valid as 1 ≤ k.

**Summary**

The fix ensures the prefix sum indices correctly map to the subarray range, leading to accurate calculations of original\_ops. Without the fix, the range sums are misaligned, causing incorrect results.

**Refer to**

<https://leetcode.com/problems/frequency-of-the-most-frequent-element/editorial/>

**Approach 3: Binary Search**

**Intuition**

Note: the previous two approaches are the optimal solutions and are sufficient to solve the problem. Here, we will look at another unique way to approach the problem for the sake of completeness.

Given an index i, if we treat nums[i] as target, we are concerned with how many elements on the left we can take. In the earlier approaches, we used a sliding window. In this approach, we will directly find the left-most index of these elements using binary search.

Let's say that best is the index of the furthest element to the left that we could increment to target = nums[i]. Note that here, best is analogous to what left was after the while loop finished in the first approach. How do we find best?

**The value of best must be in the range [0, i]. We will perform a binary search on this range**. For a given index mid:

The number of elements in the window would be count = i - mid + 1.

Thus, the final sum after making every element in the window equal to target would be finalSum = count \* target.

The original sum of the elements is the sum of the elements from index mid to index i. We can use a prefix sum to find this originalSum.

Thus, the number of operations we need is operationsRequired = finalSum - originalSum.

If operationsRequired > k, it's impossible to include the index mid. We update left = mid + 1.

Otherwise, the task is possible and we should look for a better index. We update best = mid and right = mid - 1.

Essentially, we are binary searching the left bound from the first approach for a given right bound i. If we pre-process a prefix sum, then for each mid, we have all the necessary information to find operationsRequired.

**Algorithm**

Define a function check(i):

Initialize the following integers:

target = nums[i], the current target.

left = 0, the left bound of the binary search.

right = i, the right bound of the binary search.

best = i, the best (furthest left) index that we can increment to target.

While left <= right

Calculate mid = (left + right) / 2.

Calculate count = i - mid + 1.

Calculate finalSum = count \* target.

Calculate originalSum = prefix[i] - prefix[mid] + nums[mid].

Calculate operationsRequired = finalSum - originalSum.

If operationsRequired > k, move left = mid + 1.

Otherwise, update best = mid and right = mid - 1.

Return i - best + 1.

Sort nums.

Create a prefix sum of nums.

Initialize ans = 0.

Iterate i over the indices of nums:

Update ans with check(i) if it is larger.

Return ans

**Implementation**

Be careful! Given the constraints, we may run into integer overflow. Use long accordingly in Java and C++ (Python doesn't have overflow).

class Solution {

public int check(int i, int k, int[] nums, long[] prefix) {

int target = nums[i];

int left = 0;

int right = i;

int best = i;

while (left <= right) {

int mid = (left + right) / 2;

long count = i - mid + 1;

long finalSum = count \* target;

long originalSum = prefix[i] - prefix[mid] + nums[mid];

long operationsRequired = finalSum - originalSum;

if (operationsRequired > k) {

left = mid + 1;

} else {

best = mid;

right = mid - 1;

}

}

return i - best + 1;

}

public int maxFrequency(int[] nums, int k) {

Arrays.sort(nums);

long[] prefix = new long[nums.length];

prefix[0] = nums[0];

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

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

}

int ans = 0;

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

ans = Math.max(ans, check(i, k, nums, prefix));

}

return ans;

}

}

**Complexity Analysis**

Given *n* as the length of nums,

Time complexity: *O*(*n*⋅log*n*)

First, we sort nums which costs *O*(*n*⋅log*n*).

Next, we iterate over the indices of nums. For each of the *O*(*n*) indices, we call check, which costs up to *O*(log*n*) as its a binary search over the array's elements. The total cost is *O*(*n*⋅log*n*).

Space complexity: *O*(*n*)

The prefix array uses *O*(*n*) space.

**Refer to**

[L2968.Apply Operations to Maximize Frequency Score](note://WEB45984b791ce0bb0f0a1224f1eb84df43)