



Problem Description

You are provided with an array nums, which contains integer elements and is indexed starting from 0. Along with this, you are given an integer k. The goal is to maximize your score through a specific operation that you can perform exactly k times.

The operation consists of the following steps:

- Pick an element m from the array nums.
 Pomove this element from the array.
- 2. Remove this element from the array.
- Insert a new element into the array that has a value of m + 1.
 Increase your score by m.
- After performing this operation exactly k times, you need to determine the maximum score you can achieve.

Intuition

choose the largest element, which we will denote as x, you ensure that your score increments by the maximum possible value each time.

Here's the thinking process step by step:

To maximize the score from each operation, it's logical to always select the largest element available in the array. If you always

• In the first operation, choose the maximum element in the array, x. Your score increases by x and now the array has a new

- element x + 1.
 For the second operation, since you now have x + 1 in the array (which is greater than the original x), you select and remove x +
- 1, and then add x + 2. Your score increases by x + 1 this time.
 You continue this process, each time selecting the current maximum, which continues to increase by 1 with each operation.
- After k operations, your total score will be an accumulated sum of all selected elements: x from the first operation, x + 1 from the

So, you can calculate the total score by taking the sum of the arithmetic sequence that starts with x, has k terms, and a common difference of 1.

Thus, the formula to calculate the final score is:

We can simplify the summation using the formula for the sum of the first n natural numbers:

Score = k * x + (1 + 2 + ... + (k - 1))

second, and so on, up to x + k - 1 for the kth operation.

```
Sum = n * (n + 1) / 2
```

Therefore, you can express the score as:

```
Score = k * x + (k - 1) * k / 2
```

Applying this formula grants us the maximum score after k operations without the need to simulate the operations step by step.

or relying on particular data structures. Here's how we translate the mathematical solution into a concrete implementation:

Solution Approach

First, we need to determine the maximum element from the array nums, which is denoted as x. In Python, this can be easily done
by using the built-in max() function.

The solution outlined above hinges on understanding and applying a mathematical concept rather than running complex algorithms

1 x = max(nums)
 Now that we have the maximum number (x), we must calculate the score based on the formula derived earlier:

```
Score = k * x + (k - 1) * k / 2
```

In the formula, k * x accounts for selecting the maximum element x k times, while the second part of the formula (k - 1) * k / 2 is the sum of the arithmetic series contributing to the incrementality of the score with each operation.

```
• The implementation of this calculation is straightforward and directly derived from the formula. It can be written in a single line of
```

code in Python:

1 return k * x + k * (k - 1) // 2

Here, the // operator is used to perform integer division to avoid any floating-point result, which is suitable since we are dealing with integers only.

```
By abstracting the problem into a formula, the time complexity of the solution is O(n), where n is the length of the array, coming from
```

the time complexity of finding the maximum element. The space complexity is 0(1), as we only store the maximum element and use no additional data structures to hold intermediate values.

Example Walkthrough

Let's illustrate the solution approach with a small example:

We first find the maximum element in the array, which is x = 5.

Then, we calculate the maximum score by using the earlier derived formula:
 Score = k * x + (k - 1) * k / 2

Plugging in the values:

Suppose our input array nums is [3, 5, 2] and we are allowed to perform the operation k = 2 times.

```
So, without physically removing elements and adding them back to the array, we can directly compute that the maximum score after performing the operation twice is 11.
```

1. From array nums = [3, 5, 2], find the maximum element x, which is 5.

2. Calculate the maximum score using the given values k = 2 and x = 5.

Score = 2 * 5 + (2 - 1) * 2 / 2 Score = 10 + 1 Score = 11

This hypothetical set of operations confirms the correctness of our direct calculation using the formula. The formula saves time by avoiding multiple iterations for selecting and replacing elements with their increments.

To break it down further, after the first operation, we would have selected element 5, removed it, added 6 to the array, and increased

our score by 5. After the second operation, we would select the new maximum element, which is 6, remove it, add 7 to the array, and

increase our score by another 6. Therefore, the total score would be 5 (initial score) + 6 (second operation) = 11.

3. Substitute into the formula to get the result: Score = 2 * 5 + 1 * (2 - 1) = 11.
A Python implementation of this example would be:

4 x = max(nums)
5 score = k * x + k * (k - 1) // 2
6
7 print(score) # Output should be 11

from typing import List

 $max_num = max(nums)$

return maximized_sum

class Solution:

6

9

14

15

16

17

18

1 nums = [3, 5, 2]

Implementing this step-by-step:

```
This program would output 11, which represents the maximum score achieved after performing the operation k times.

Python Solution
```

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

 $maximized_sum = k * max_num + k * (k - 1) // 2$

#include <algorithm> // Include algorithm for std::max_element

int maximizeSum(std::vector<int>& nums, int k) {

// Find the maximum element in the vector

// Function to maximize sum by performing k increments on the maximum element

// The formula used is the sum of the first k elements of an arithmetic series

// starting with maxValue and decrementing by 1 each time.

const sum = k * maxValue + (k * (k - 1)) / 2;

int maxElement = *std::max_element(nums.begin(), nums.end());

int totalIncrement = k * maxElement + k * (k - 1) / 2;

Return the calculated maximized sum

Find the maximum number in the given list of numbers

And for each iteration, we also add the i-th increment (0 to k-1).

This is equivalent to max_num * k (adding max_num k times), plus the sum

of the first k natural numbers, which is k * (k - 1) // 2 using the formula for the sum

of the first 'n' natural numbers.

The strategy to maximize the sum is to repeatedly increment the maximum number.

We will do this k times, each time adding the maximum number to the sum.

```
Java Solution
```

```
class Solution {
       public int maximizeSum(int[] nums, int k) {
           // Initialize a variable to store the maximum value found in the array
           int maxVal = 0;
           // Iterate through all elements in the array to find the maximum value
           for (int value : nums) {
               maxVal = Math.max(maxVal, value);
9
10
11
           // Calculate and return the sum of the largest k numbers in an arithmetic progression
12
           // starting from maxVal and with a common difference of 1.
           return k * maxVal + k * (k - 1) / 2;
13
16
C++ Solution
```

// Calculate the total increment on the max element after k operations // This is done by multiplying the maximum element with k // And then adding the sum of first k natural numbers (k*(k-1)/2)

#include <vector>

class Solution {

public:

```
// Return the sum after all increments are done
16
17
           return totalIncrement;
18
19 };
20
Typescript Solution
   /**
    * Maximizes the sum by adding the top k numbers where the
    * first number is the maximum in the array and every subsequent
    * number is one less than the previous.
    * @param {number[]} nums - The array of numbers to search through.
    * @param {number} k - The number of top elements to consider for maximizing the sum.
    * @returns {number} The maximized sum according to the described formula.
   function maximizeSum(nums: number[], k: number): number {
       // Find the maximum value in the nums array.
       const maxValue = Math.max(...nums);
12
       // Calculate the sum of the top k numbers starting from the maxValue.
13
```

// Return the calculated sum. return sum; }

Time and Space Complexity

14

15

16

The given Python code consists of two main operations: finding the maximum value in the list nums, and performing arithmetic calculations to obtain the result.

Time Complexity The time complexity of finding the maximum element in a list of n elements using max(nums) is O(n), as it requires checking each

element exactly once. The arithmetic operations k * x + k * (k - 1) // 2 involve constant time calculations, hence their complexity is 0(1).

Overall, the time complexity of the function is 0(n) + 0(1), which simplifies to 0(n).

Space Complexity

In terms of space complexity, there are no additional data structures used that grow with the input size. Only a constant amount of additional space is used for variables like x. Therefore, the space complexity of the function is 0(1).

additional space is used for variables like x. Therefore, the space complexity of the function is 0(1). In conclusion, the time complexity of the provided code is 0(n) and the space complexity is 0(1).