2202. Maximize the Topmost Element After K Moves

Medium Greedy Array

Leetcode Link

Problem Description

In this problem, we have an array `nums` that represents the contents of a pile with an index starting at 0. The element at index 0 (`nums[0]`) is at the top of the pile. There are two types of moves we can perform on this pile:

1. Remove the topmost element of the pile, if the pile is not empty.
2. Add any one of the previously removed elements back onto the top of the pile, assuming there's at least one removed element available.

We are also given an integer `k`, which represents the total number of moves that have to be made.

Objective: Our task is to find out the maximum possible value of the element at the top of the pile after making exactly `k` moves. If it's not possible to have any elements left in the pile after `k` moves, we should return `-1`.

This problem requires us to manipulate the topmost element of a pile with some constraints and to plan the moves to maximize the value of the topmost element taking into consideration the number of available moves.

Intuition

To find the solution, we need to consider the number of elements in the pile and the number of available moves, `k`. We can break this down into different scenarios:

1. If `k` is 0, we do not need to make any moves, and we simply return the current topmost element, `nums[0]`.
2. If there is only one element in the pile (`len(nums) == 1`) and `k` is odd, we will inevitably end up with an empty pile after these moves since each odd move removes the last element and each even move can't put it back (as there's no other to add). Hence, in this case, the answer is `-1`.
3. If `k` is less than the number of elements in the pile (`k < len(nums)`), we need to get the maximum between two potential topmost elements:
   - The largest element from the range `nums[0]` to `nums[k-1]`, acknowledging that we can make `k-1` removals and then put the maximum number back on the pile.
   - The element at index `k` (`nums[k]`), since it can be the top of the pile if we remove `k` elements (one of them being this `nums[k]`) and then put `nums[k]` back.
4. If `k` is greater or equal to the number of elements (`k >= len(nums)`), we only look at the maximum element within the bounds of `nums` in the first `k-1` moves. The `k-th` move could be adding an element back if there are still remaining moves.

By considering these scenarios, we can design the solution which ensures that we return the maximum topmost value after `k` moves, or `-1` if it's not possible to perform `k` moves and still have elements left in the pile.

Solution Approach

The provided solution is straightforward and makes use of conditional checks and the `max` function to compare elements within the `nums` array based on different scenarios discussed in the intuition section.

Here's a step-by-step explanation of the implementation:

1. **Check for No Moves (`k == 0`)**: If the number of moves `k` is `0`, we simply return the topmost element as no moves are made.

```
if k == 0:
    return nums[0]
```

2. **Check for Single Element Piles (`n == 1`)**: If the pile has only one element, we need to check if `k` is an odd number. If `k` is odd, we will end up with no element at the top, so return `-1`. If `k` is even, we can keep removing and adding the same element, effectively making no changes, and we return the topmost element.

```
n = len(nums)
if n == 1:
    if k % 2:
        return -1
    return nums[0]
```

3. **Find Maximum with Constraints**: We then find the maximum value from the array up to the `k-1` index. This is because after `k-1` removals, we could replace the topmost element with one of the removed elements. We use the `max` function over the slice `nums[: k - 1]`, and use `default=-1` to handle the edge case where the slice could be empty.

```
ans = max(nums[: k - 1], default=-1)
```

4. **Consider Element at Index `k`**: If `k` is less than the length of the array (`k < n`), it means the element at the index `k` could be a candidate for the maximum topmost element after `k` moves. We should choose the larger between the current `ans` and `nums[k]`.

```
if k < n:
    ans = max(ans, nums[k])
```

5. **Return the Result**: Finally, we return `ans` as the maximum possible value for the top of the pile after exactly `k` moves.

In terms of data structures, the solution simply uses the list (array) given, without the need for additional data structures. The key algorithmic pattern here is the greedy approach, selecting the largest value at each step with consideration to the constraints given by `k` and the number of elements `n`. The implementation uses basic array operations and conditions to achieve the desired outcome.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose we are given the following array of integers representing our pile:

```
nums = [4, 1, 5, 6]
```

and we have `k = 3` moves to make.

Following the solution steps:

1. Since `k` is not `0`, we do not return `nums[0]` right away. We proceed with the next steps.
2. There is more than one element in the pile (`len(nums) = 4`), so we do not need to consider the scenario where `k` is odd, and we only have one element.
3. We find the maximum value from the array up to the `k-1` index, which is `2` in this case, so we consider `nums` up to and including index `1`: `nums[:k-1] = [4, 1]`. We find the maximum value in this range:

```
ans = max(nums[:2])  # The maximum value from [4, 1] which is 4.
```

   So, `ans` is now `4`.

4. Since `k` is less than the length of the array (`k < n`), we consider the element at index `k`, which is `nums[3] = 6`. We compare it with our current `ans` and select the larger value:

```
ans = max(ans, nums[3])  # We compare 4 and 6.
```

   So, `ans` is updated to `6`.

5. After evaluating the given conditions, we have determined that the maximum value we can have at the top of the pile after exactly `3` moves is `6`.

We can now visualize the moves:

- First move: we remove the topmost element `4` from the pile, now the pile is `[1, 5, 6]`.
- Second move: we remove the topmost element `1` from the pile, now the pile is `[5, 6]`.
- Third move: we add an element back to the pile, and we have the option to add either `1` or `4`. We choose `4` as it is the larger value. However, we also realize that the remaining element `6` can be placed on top as it is the largest of all the elements we have, even larger than `4`. Thus, we opt to place `6` on top instead, leading to our answer.

Therefore, the maximum possible value for the top of the pile after exactly `3` moves is `6`.

Python Solution

```
class Solution:
    def maximumTopiself, nums: List[int], k: int) -> int:
        # If we cannot perform any operations, return the top element immediately.
        if k == 0:
            return nums[0]

        # Get the number of elements in the stack.
        num_elements = len(nums)

        # If there's only one element in the stack and k is odd,
        # we will never be able to put any element at the top after removal.
        if num_elements == 1:
            # If k is odd, it means we end up with an empty stack and nothing to replace the top
            return -1 if k % 2 == 1 else nums[0]

        # Find the maximum value in the stack within the range of available operations.
        # We exclude the k-th element because if we are able to reach it,
        # it means we have removed the original top and haven't replaced it yet.
        max_value = max(nums[:k - 1], default=-1)

        # If k is less than the number of elements, we can consider
        # the k-th element as a candidate for the maximum top element
        # after performing the operations.
        if k < num_elements:
            # "nums[k]" is the element just after 'k - 1' operations, it might be the largest.
            max_value = max(max_value, nums[k])

        return max_value
```

Java Solution

```
class Solution {
    public int maximumTop(int[] nums, int k) {
        // If no moves are allowed (k is zero), we return the top number on the stack
        if (k == 0) {
            return nums[0];
        }

        // Get the number of elements in the stack
        int stackSize = nums.length;

        // Special case when there's only one number in the stack
        if (stackSize == 1) {
            // If k is odd, we end up with an empty stack
            if (k % 2 == 1) {
                return -1;
            }
            // If k is even, we can restore the same number back to the stack
            return nums[0];
        }

        // Initialize the answer to an impossible value, signaling not found yet
        int maximumValue = -1;

        // Look for the maximum number that can be the top of the stack
        // either by removing the top k-1 numbers or by not changing the stack.
        for (int index = 0; index < Math.min(k - 1, stackSize); ++index) {
            maximumValue = Math.max(maximumValue, nums[index]);
        }

        // If k is less than the stack size, we must also consider the k-th number in the stack,
        // since it can be on top after k-1 operations.
        if (k < stackSize) {
            maximumValue = Math.max(maximumValue, nums[k]);
        }

        // Return the maximum number that can be on top of the stack
        return maximumValue;
    }
}
```

C++ Solution

```
class Solution {
public:
    int maximumTop(vector<int>& nums, int k) {
        // If no operation is to be performed, simply return the top element.
        if (k == 0) return nums[0];

        // The variable 'n' holds the size of the vector 'nums'.
        int n = nums.size();

        // If we only have one number and an odd number of moves, we can never have a top.
        if (n == 1) {
            if (k % 2) return -1;
            return nums[0];
        }

        // Initialize the maximum number we can get to negative, since we can't get a negative index.
        int maxNumber = -1;

        // Check until lesser of k-1 or n for the possible maximum number on top.
        for (int i = 0; i < min(k - 1, n); ++i) {
            maxNumber = max(maxNumber, nums[i]);
        }

        // If we have more than k elements, check if the kth element can be the top.
        if (k < n) {
            maxNumber = max(maxNumber, nums[k]);
        }

        // Return the maximum number that could be on top after k moves.
        return maxNumber;
    }
};
```

Typescript Solution

```
function maximumTop(nums: number[], k: number): number {
    // If no operation is to be performed, simply return the topmost element.
    if (k === 0) return nums[0];

    // Variable 'n' holds the size of the array 'nums'.
    let n: number = nums.length;

    // If there is only one number and an odd number of moves,
    // we can never have a top, so return -1.
    if (n === 1) {
        if (k % 2) return -1;
        return nums[0];
    }

    // Initialize 'maxNumber' as negative, to represent that we have not yet found a feasible number.
    let maxNumber: number = -1;

    // Search for the maximum number within the bounds of 'k - 1' or 'n'
    // which could be the top element after performing 'k - 1' moves.
    for (let i: number = 0; i < Math.min(k - 1, n); ++i) {
        maxNumber = Math.max(maxNumber, nums[i]);
    }

    // If there are more than 'k' elements, it is possible to place
    // the 'k'-th element on top after performing 'k' operations.
    if (k < n) {
        maxNumber = Math.max(maxNumber, nums[k]);
    }

    // Return the maximum number that could be on top after performing 'k' moves.
    return maxNumber;
}
```

Time and Space Complexity

The time complexity of the given code is $O(k)$, where `k` is the number of operations to be performed on the `nums` array. This is because the code potentially examines elements up to the `k-1` position in the `nums` array to find the maximum value, and the number of elements looked at is limited by `k`. When `k < n`, it also compares the value at the `k`-th position once, which does not affect the overall time complexity.

The space complexity of the code is $O(1)$ since the code only uses a constant amount of additional space. The variables `n` and `ans` are used to store the length of the `nums` array and the interim maximum values, respectively, and they do not depend on the size of the input array.