1746. Maximum Subarray Sum After One Operation

Medium Array Dynamic Programming Leetcode Link

Problem Description

You are provided with an integer array `nums`. Your task is to perform exactly one operation in which you choose one element `nums[i]` and replace it with the square of that element, `nums[i] * nums[i]`. The objective is to return the maximum sum of a non-empty subarray after performing this single operation. A subarray is a contiguous part of the array, and you are looking for the subarray which gives you the maximum possible sum after squaring exactly one of its elements.

Intuition

The intuition behind the solution is to use dynamic programming (DP) to keep track of the maximum subarray sums while iterating through the array. For this, you maintain two variables during the iteration.

One (`f`) is to keep track of the maximum sum so far without applying the square operation, and the other (`g`) is to keep track of the maximum sum so far with the square operation applied to one of the elements. As you iterate, for each new element you encounter, you update these two variables.

To update `f`, you add the current element to the maximum sum so far (`f`) if it's positive; otherwise, you start a new subarray sum from the current element.

To update `g`, you have two choices: either you use the square operation on the current element and add it to the maximum sum so far without the operation (`f`), or you add the current element to the maximum sum so far with the operation (`g`).

Finally, `ans` is used to store the maximum of all `f` and `g` encountered so far, which will be your final answer.

The reason for maintaining these two states is because at each step, you have to consider that you can either use your one-time square operation on the current element or on one of the future elements. Therefore, you need to have both scenarios considered in your dynamic programming states.

Solution Approach

The solution uses a simple dynamic programming approach. Two running variables, `f` and `g`, are used to keep track of the current maximum subarray sum with and without using the square operation, respectively.

The algorithm can be broken down into the following steps:

1. Initialize `f` and `g` to `0`, which represent `f[i]` and `g[i]` respectively - the maximum subarray sum ending at index `i`, with `g[i]` considering that the operation has been applied. Also, initialize `ans` with `-inf` to track the overall maximum sum while iterating through the array.
2. Iterate through each number `x` in the `nums` list: a. To update `f` (`ff` in the code), calculate the maximum between `f + x` and `0 + x`. The `max(f, 0)` ensures that if the previous sum `f` is negative, it's better to start a new subarray at the current index. b. To update `g` (`gg` in the code), calculate the maximum between `g + x` (adding the current element to the maximum sum with the operation already used) and `max(f, 0) + x * x` (applying the operation on the current element and adding to the maximum sum without the operation). c. Replace the old `f` and `g` with the new `ff` and `gg`. d. Update `ans` with the maximum of `ans`, `f`, and `g` to ensure it keeps track of the highest sum seen so far.
3. Return `ans`, which contains the maximum subarray sum after exactly one operation.

This approach makes use of the Kadane's algorithm pattern, which is a popular technique for finding the maximum subarray sum. The extension here is the consideration of the additional operation, which is handled by maintaining a parallel running sum that considers the effect of the square operation.

In terms of complexity, the algorithm runs in `O(n)` time where `n` is the size of the input array since it only involves a single pass through the array. The space complexity is `O(1)` as it only uses a fixed number of variables.

Example Walkthrough

Let's go through an example to illustrate the solution approach.

Consider the array `nums = [-2, -1, -3, 4]`. We want to maximize the sum of a subarray after squaring exactly one element from this array. Let's take this step by step:

1. Initialize `f` and `g` to 0, which represent the maximum subarray sum ending at the current index without and with the operation, respectively. Also, initialize `ans` to negative infinity (`-inf`) for tracking the overall maximum.
2. Start iterating through each number in the `nums` list:
   - For the first element -2:
     - Update `ff`: max(f + x, x) = max(0 - 2, -2) = -2
     - Update `gg`: max(g + x, max(f, 0) + x * x) = max(0 - 2, 0 + (-2) * (-2)) = 4
     - Update `f` to `ff`: f = -2
     - Update `g` to `gg`: g = 4
     - Update `ans`: max(-inf, -2, 4) = 4
   - Next element -1:
     - Update `ff`: max(f + x, x) = max(-2 - 1, -1) = -1
     - Update `gg`: max(g + x, max(f, 0) + x * x) = max(4 - 1, 0 + (-1) * (-1)) = 4
     - Update `f` to `ff`: f = -1
     - Update `g` to `gg`: g = 3
     - Update `ans`: max(4, -1, 3) = 4
   - Next element -3:
     - Update `ff`: max(f + x, x) = max(-1 - 3, -3) = -3
     - Update `gg`: max(g + x, max(f, 0) + x * x) = max(3 - 3, 0 + (-3) * (-3)) = 9
     - Update `f` to `ff`: f = -3
     - Update `g` to `gg`: g = 9
     - Update `ans`: max(4, -3, 9) = 9
   - Last element 4:
     - Update `ff`: max(f + x, x) = max(-3 + 4, 4) = 4
     - Update `gg`: max(g + x, max(f, 0) + x * x) = max(9 + 4, 0 + 4 * 4) = 16
     - Update `f` to `ff`: f = 4
     - Update `g` to `gg`: g = 13
     - Update `ans`: max(9, 4, 13) = 13
3. At the end of iteration, the `ans` variable holds the maximum sum possible after squaring exactly one element, which is 13. This is the result of squaring the third element in the original array and adding it to the last element, `(-3) * (-3) + 4` which equals 13.

The solution operates seamlessly by iteratively comparing the consequences of squaring or not squaring the current element against the running sums, which ingeniously captures the essence of Kadane's algorithm while accommodating for the additional operation to eventually arrive at the optimal solution.

Python Solution

```python
from typing import List  # Import List type for type annotations

class Solution:
    def maxSumAfterOperation(self, nums: List[int]) -> int:
        # Initialize current_sum and max_sum, both set to 0.
        current_sum = max_sum_with_operation = 0

        # Initialize result with the smallest number possible.
        result = float('-inf')

        # Iterate over each number in the input list.
        for num in nums:
            # Calculate the max sum of the subarray without operation,
            # by comparing the sum including the current number and dropping to zero when it's negative.
            current_sum = max(current_sum, 0) + num

            # Calculate the max sum of the subarray with one operation applied.
            # Compare the sum with the current number squared (and possibly discard the previous sum),
            # or continue with the previous sum with operation and add the current number.
            max_sum_with_operation = max(max(current_sum + (num * num - num)), max_sum_with_operation + num)

            # Update `current_sum` to the new value calculated.
            # Update `max_sum_with_operation` to the new value calculated.
            current_sum, max_sum_with_operation = current_sum, max_sum_with_operation

            # Record the maximum result found so far by comparing it with `current_sum` and `max_sum_with_operation`.
            result = max(result, current_sum, max_sum_with_operation)

        # Return the maximum result possible after performing the operation exactly once.
        return result

```

Java Solution

```java
class Solution {
    public int maxSumAfterOperation(int[] nums) {
        int maxSumWithoutOp = 0; // Tracks the max sum without any operation
        int maxSumWithOp = 0; // Tracks the max sum with at most one operation (square of an element)
        int maxResult = Integer.MIN_VALUE; // Result variable, starts at the smallest integer as a lower bound

        // Loop through the array
        for (int num : nums) {
            // Calculate new sum without operation, choose between adding current number or starting anew
            int newMaxSumWithoutOp = Math.max(maxSumWithoutOp, 0) + num;

            // Calculate new sum with operation, choose between:
            // 1. Using operation on the current number and adding to maxSumWithoutOp
            // 2. Adding the current number to maxSumWithOp (operation used on a previous number)
            int newMaxSumWithOp = Math.max(maxSumWithoutOp + num * num, maxSumWithOp + num);

            // Move the calculated sums into our tracking variables
            maxSumWithoutOp = newMaxSumWithoutOp;
            maxSumWithOp = newMaxSumWithOp;

            // Update maximum result among all sums with at most one operation
            maxResult = Math.max(maxResult, Math.max(maxSumWithoutOp, maxSumWithOp));
        }

        // Return the maximum result found
        return maxResult;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function finds the maximum sum after performing exactly one operation
    // where the operation is defined as squaring any one element in the array.
    int maxSumAfterOperation(vector<int>& nums) {
        int maxEndHereWithoutOp = 0;  // f: Max sum subarray ending here without using the operation
        int maxEndHereWithOp = 0;     // g: Max sum subarray ending here with using the operation
        int maxResult = INT_MIN;      // ans: Result for the maximum sum after operation

        for (int num : nums) {
            // Update max sum subarray when not using operation
            int newMaxEndHereWithoutOp = max(maxEndHereWithoutOp, 0) + num;

            // Update max sum subarray when using operation, either by using the operation on current
            // number or adding current number to previous subarray where operation was already used.
            int newMaxEndHereWithOp = max(maxEndHereWithoutOp + num * num, maxEndHereWithOp + num);

            // Update variables for the next iteration
            maxEndHereWithoutOp = newMaxEndHereWithoutOp;
            maxEndHereWithOp = newMaxEndHereWithOp;

            // Update the maximum result from three choices: previous max, current subarray without operation,
            // and current subarray with operation.
            maxResult = max({maxResult, maxEndHereWithoutOp, maxEndHereWithOp});
        }

        return maxResult;
    }
};

```

Typescript Solution

```typescript
// Define a function to find the maximum sum after performing exactly one operation
// where the operation is defined as squaring any one element in the array.
function maxSumAfterOperation(nums: number[]): number {
    let maxSumWithoutOperation: number = 0;  // Tracks max sum subarray ending here without the operation
    let maxSumWithOperation: number = 0;     // Tracks max sum subarray ending here with the operation
    let maxResult: number = Number.MIN_SAFE_INTEGER; // Stores the result for the maximum sum after operation

    for (let num of nums) {
        // Update max sum subarray when not using operation
        // If the previous sum is negative, reset it to 0; otherwise, add the current number.
        let newMaxSumWithoutOperation: number = Math.max(maxSumWithoutOperation, 0) + num;

        // Update max sum subarray when using operation. Two choices:
        // 1. Use the operation on the current number and add it to the previous sum without operation.
        // 2. Add the current number to the previous sum where the operation was already used.
        let newMaxSumWithOperation: number = Math.max(maxSumWithoutOperation + num * num, maxSumWithOperation + num);

        // Prepare for the next iteration
        maxSumWithoutOperation = newMaxSumWithoutOperation;
        maxSumWithOperation = newMaxSumWithOperation;

        // Update the maxResult at each step, based on the max sum without operation, the max sum with operation,
        // and the previous maximum result.
        maxResult = Math.max(maxResult, maxSumWithoutOperation, maxSumWithOperation);
    }

    return maxResult; // Return the final result, the maximum sum obtainable after the operation
}

// Usage example
const nums: number[] = [2, 0, -1, 3];
const result: number = maxSumAfterOperation(nums);
console.log(`The maximum sum after operation is: ${result}`);

```

Time and Space Complexity

The given code represents a solution where the objective is to compute the maximum sum of a modified array where exactly one operation of squaring one element is permitted. We use a dynamic programming approach to keep track of the maximum sum we can achieve with and without squaring an element at each step.

Time Complexity:

The algorithm iterates through the `nums` array once, performing a constant amount of work for each element. Specifically, it calculates the values of `ff` and `gg` which involve basic arithmetic operations and comparisons. No nested loops or additional iterations are present. As a result, the time complexity is `O(n)` where `n` is the length of the `nums` array.

Space Complexity:

Since the extra variables `f`, `g`, `ff`, `gg`, and `ans` use a fixed amount of space and no additional data structures dependent on the input size are created, the space complexity is `O(1)`. This constant space is irrespective of the input size.