3010. Divide an Array Into Subarrays With Minimum Cost I

Easy Array Enumeration Sorting

Problem Description

In this problem, you are given an array of integers `nums` which has a length of `n`. Your task is to split this array into three disjoint contiguous subarrays. The term 'disjoint' means the subarrays should not overlap, and 'contiguous' indicates that the elements in each subarray should be next to each other in the original array `nums`.

The unique aspect of this problem is how the *cost* of a subarray is defined. The cost is determined by the first element of each subarray. So, if you create a subarray `[2,5,6]`, the cost is `2` because it is the first element.

Your objective is to find a way to divide the array `nums` that results in the minimum possible sum of the costs of these three subarrays. Put simply, you want to select the first elements of three subarrays such that their sum is as small as possible.

Intuition

The solution draws on a straightforward idea: we need to find the smallest possible values for the first elements of three subarrays since these elements determine the cost of the subarrays.

Therefore, one element will be the first element of the whole array `nums`, which is unavoidable as it will always be the first element of the first subarray. Let's denote this element with the variable `a`.

For the other two subarrays, we need to find the smallest and second smallest values among the remaining elements of `nums`. We don't worry about the specific position within the array of these elements because our objective is to minimize the sum of the three costs, not to arrange subarrays in any particular order.

The solution's approach begins by initializing `a` to the first element of `nums`. We then iterate through the rest of the array looking for the smallest (`b`) and second smallest (`c`) elements. If we encounter a value less than `b`, we shift the old `b` to `c` and update `b` with this new smallest value. Otherwise, if we find a value that is less than `c` but greater than or equal to `b`, we just update `c` since we have found a new second smallest value.

Finally, we add `a`, `b`, and `c` together to get the minimum possible sum of the costs, as this sum represents the cost of the first element of each of the three needed subarrays.

Solution Approach

The algorithm relies on the notion of finding the minimum and second minimum values in an array. This is a common pattern when dealing with problems that require optimization based on certain criteria—in this case, the minimum sum of specific elements.

In the reference solution approach, the variables `a`, `b`, and `c` represent the first element of the array, the smallest value, and the second smallest value among the remaining elements of the array, respectively. The `inf` or infinity value used to initialize `b` and `c` ensures that any real number encountered in the `nums` array will be less than `inf` and appropriately assigned to `b` or `c`.

The process to achieve the solution can be broken into the following steps:

1. Initialize variables `a`, `b`, and `c`:
   - `a` is set to `nums[0]`, because it will be the first element of the first subarray and its cost is unavoidable.
   - `b` and `c` are set to infinity (`inf`).
2. Loop through the `nums` array starting from the second element, as the first is already used for `a`.
3. For each element `x` in `nums`:
   - If `x` is less than `b`, it means we have found a new minimum. Hence, we update `c` to the old `b` (the previous minimum), then update `b` to `x` (the new minimum).
   - Otherwise, if `x` is greater than or equal to `b` but less than `c`, we just need to update `c` to `x`, as we have found a new second smallest value.
4. Once the loop is completed, `a`, `b`, and `c` will hold the cost of the first elements of the three subarrays, with `a` being the cost of the unavoidable first subarray, and `b` and `c` being the minimum and second minimum costs we could find for the other two subarrays.

By the end of the iteration, adding `a`, `b`, and `c` gives us the minimum possible sum for the costs of the three subarrays.

The beauty of this solution lies in its simplicity and efficiency. It allows us to find the minimum sum of costs with just one pass through the array and without any need for complex data structures or sorting algorithms. This single-pass scan of the array also means that the solution has a time complexity of O(n), where n is the number of elements in `nums`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the array `nums = [4, 1, 3, 2, 5]`.

1. Initialize `a`, `b`, `c`:
   - `a` is set to `nums[0]`, so `a = 4`.
   - `b` and `c` are set to infinity, and thus any number from the array will replace these values.
2. Loop through the `nums` array starting from the second element, `nums[1]`.
3. Iterating through `nums`:
   - Current element is `1`. Since `1 < inf`, we update `b`:
     - `b` now becomes `4` (old `b`) and we update `b` to `1`.
   - Now `a = 4`, `b = 1`, `c = 4`.
   - Next element is `3`. Since `3 > b` (`1`) but `3 < c` (`4`), we update `c`:
     - Hence, `c` is now `3`.
   - Now `a = 4`, `b = 1`, `c = 3`.
   - Next comes `2`. It is less than `c` (`3`) and also less than `b` (`1`):
     - Since `2` is not less than `b`, we only update `c` with `2`.
   - Now `a = 4`, `b = 1`, `c = 2`.
   - The last element is `5`. It is greater than both `b` and `c`, so no update is made.
   - The values remain `a = 4`, `b = 1`, `c = 2`.
4. By the end of the iteration, we have `a = 4` (the cost of the first subarray), `b = 1`, and `c = 2`. The latter two are the minimum and second minimum values that could be the costs of the other two subarrays.

The minimum possible sum of the costs for the three subarrays is the sum of `a`, `b`, and `c`, which in this case is (4 + 1 + 2 = 7).

So by employing a single pass through the array and maintaining two variables to keep track of the smallest and second smallest elements after the first, we are able to efficiently compute the minimum sum of subarray costs.

Solution Implementation

Python

```python
from typing import List
from math import inf

class Solution:
    def minimumCost(self, nums: List[int]) -> int:
        # Initialize variables. 'a' will hold the smallest number,
        # 'b' will be second smallest. 'c' will be the third smallest.
        # 'inf' is used to represent an infinitely large number.
        a, b, c = nums[0], inf, inf

        # Iterate over the nums list starting from the second element.
        for num in nums[1:]:
            # If the current number is smaller than 'b',
            # then we update 'b' and 'c' accordingly.
            if num < b:
                # Before updating 'b', shift its value to 'c'.
                c = b
                b = num
            # If 'num' is not smaller than 'b', but is less than 'c',
            # then we just update 'c'.
            elif num < c:
                c = num

        # Return the sum of the three smallest numbers.
        return a + b + c

# Note: Calling the method minimumCost will require an instance of the Solution class.
# For example: Solution().minimumCost([your_list_of_numbers])
```

Java

```java
class Solution {
    public int minimumCost(int[] prices) {
        // Initialize variables for the smallest, second smallest, and third smallest prices
        int smallest = prices[0], secondSmallest = Integer.MAX_VALUE, thirdSmallest = Integer.MAX_VALUE;

        // Iterate through the prices starting from the second element
        for (int i = 1; i < prices.length; ++i) {
            // Check if the current price is less than the second smallest
            if (prices[i] < secondSmallest) {
                // Update the third smallest to be the previous second smallest
                thirdSmallest = secondSmallest;
                // Update the second smallest to be the current price
                secondSmallest = prices[i];
            } else if (prices[i] < thirdSmallest) { // Otherwise, check if the current price is less than the third smallest
                // Update the third smallest to be the current price
                thirdSmallest = prices[i];
            }
        }

        // Return the sum of smallest, second smallest, and third smallest prices
        return smallest + secondSmallest + thirdSmallest;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // For sort

class Solution {
public:
    int minimumCost(std::vector<int>& nums) {
        // Check if the vector has less than three numbers;
        // if it does, one cannot select three numbers, so return 0.
        if(nums.size() < 3) {
            return 0;
        }

        // Sort the nums vector in ascending order.
        std::sort(nums.begin(), nums.end());

        // Initialize the sum of the first three smallest numbers.
        int minimumSum = nums[0] + nums[1] + nums[2];

        // Return the sum, which is the cost of purchasing the three cheapest items.
        return minimumSum;
    }
};
```

TypeScript

```typescript
function minimumCost(nums: number[]): number {
    // Initialize the smallest elements a, b, and c with the first element and two placeholders (100).
    let smallest = nums[0];
    let secondSmallest = 100;
    let thirdSmallest = 100;

    // Iterate over the array of numbers starting from the second element.
    for (const current of nums.slice(1)) {
        // If the current element is smaller than the second smallest, update the second and third smallest.
        if (current < secondSmallest) {
            [secondSmallest, thirdSmallest] = [current, secondSmallest];
        } else if (current < thirdSmallest) {
            // If the current element is only smaller than the third smallest, only update the third smallest.
            thirdSmallest = current;
        }
    }

    // Sum the three smallest elements found.
    return smallest + secondSmallest + thirdSmallest;
}
```

```python
from typing import List
from math import inf

class Solution:
    def minimumCost(self, nums: List[int]) -> int:
        # Initialize variables. 'a' will hold the smallest number,
        # 'b' will be second smallest. 'c' will be the third smallest.
        # 'inf' is used to represent an infinitely large number.
        a, b, c = nums[0], inf, inf

        # Iterate over the nums list starting from the second element.
        for num in nums[1:]:
            # If the current number is smaller than 'b',
            # then we update 'b' and 'c' accordingly.
            if num < b:
                # Before updating 'b', shift its value to 'c'.
                c = b
                b = num
            # If 'num' is not smaller than 'b', but is less than 'c',
            # then we just update 'c'.
            elif num < c:
                c = num

        # Return the sum of the three smallest numbers.
        return a + b + c

# Note: Calling the method minimumCost will require an instance of the Solution class.
# For example: Solution().minimumCost([your_list_of_numbers])
```

Time and Space Complexity

The provided Python code aims to calculate the minimum cost by finding the three smallest numbers in the given list `nums` and summing them up. It initializes `a` with the first element of the list and sets `b` and `c` to infinity (`inf`). The loop then iterates through the remaining elements, updating `b` and `c` with the next smallest values.

The time complexity of this code is indeed `O(n)` where `n` is the length of the array `nums`. This is because the code goes through the elements of `nums` exactly once in a single loop, performing a constant number of operations per element (comparisons and assignments).

The space complexity of the code is `O(1)` as the space used does not grow with the size of the input list `nums`. Only a fixed number of variables (`a`, `b`, `c`) are used regardless of the input size.