2593. Find Score of an Array After Marking All Elements

Medium Array Sorting Simulation Heap (Priority Queue)

Leetcode Link

Problem Description

In this problem, you are given an array called `nums` which contains positive integers. The task is to implement an algorithm to calculate a score, beginning with a score of 0. The algorithm consists of the following steps:

1. Identify the smallest unmarked integer in the array. If there's a tie for the smallest number, choose the one that comes first in the array.
2. Add the value of this chosen integer to a running total score.
3. Mark the chosen integer as well as its neighbors - the numbers immediately to its left and right in the array, if they exist.
4. Repeat this process until all elements in the array have been marked.

The goal is to return the final score after all elements in the array have been processed according to the algorithm described.

Intuition

To efficiently solve this problem, we must strategically pick the smallest unmarked integers while keeping track of which elements are marked. Using a min heap (or a priority queue in terms of some programming languages) can help us to maintain a sorted set of unmarked elements to choose from. Each element we're considering is a pair consisting of the integer's value and its index in the array.

The intuition for using a min heap is due to its property of always giving us the smallest element when we pop from it. This helps us to follow the rule of choosing the smallest unmarked integer at each step without having to sort the array or search for the minimum value each time we make a choice.

When we choose an integer from the min heap, we add its value to the score and mark it along with its adjacent indices. Once an index is marked, we ensure it doesn't come back into consideration by popping out all marked elements from the top of the min heap before making the next choice.

The main challenges we need to address in our algorithm are:

- Efficiently finding and choosing the smallest unmarked numbers.
- Keeping track of which elements have been marked to avoid selecting them again.
- Handling the situation when there are multiple smallest numbers (resolving ties by index).
- Maintaining the min heap property by popping out marked elements before each new selection.

To achieve the above challenges, we initialize a priority queue to store pairs of numbers and their indices from the given array, and we mark elements inside a boolean visitation array. The workflow consists of choosing and popping the smallest number from the heap, marking the chosen numbers, and popping out all marked elements from the top of the heap before proceeding to the next step.

With this approach, we can assure that we're always operating on the right elements in a time-efficient manner. The process goes on until the heap is empty, which means all elements have been marked, and we then return the accumulated score.

Solution Approach

The implementation of the solution to this problem involves a couple of critical concepts from computer science, primarily from the field of data structures and algorithms.

Solution 1: Priority Queue (Min Heap)

Since we need to repeatedly select the smallest unmarked element from our list, a **Min Heap** is used, which is a binary tree where the parent is always smaller than its children. This guarantees that the smallest element will always be at the root of the tree, hence accessible in constant time $O(1)$. To ensure that we are always working with the smallest unmarked items, we perform the following steps:

- Introduce a priority queue and fill it with pairs `(value, index)` for all elements in the array.
- Use a `visited` array of boolean values initially set to `False`, which will mark whether elements in `nums` have been considered or not.
- While the priority queue is not empty:
  1. Extract the smallest element from the heap which is unmarked. If it is already marked, continue extracting until an unmarked one is found.
  2. Once found, add its value to the `score`, and mark this element as visited.
  3. Also mark the adjacent elements (left and right neighbors) as visited.
  4. Then, remove all top elements in the heap that are marked as visited to keep the heap clean for the next iteration.
- After all elements have been marked and processed, the running score is returned as the final answer.

The **time complexity** of this solution is $O(n \log n)$ because we perform at most n insertions to the heap and n deletions from the heap, both of which are $O(\log n)$ operations. The **space complexity** is $O(n)$ due to the storage required for the heap and the visited array.

Solution 2: Sorting (Alternate Approach)

An alternate solution involves sorting. Here's the workflow:

- Create an Index array `idx` where $idx[i] = i$ for each element in `nums`.
- Sort the index array `idx` so that its indices correspond to the elements in `nums` sorted first by value and then by their index, should values be equal.
- Create an array `vis` of length `n+2` initialized to `false`, which will keep track of the marked elements.
- Iterate through the index array `idx`, for each element `i`:
  1. If `vis[i+1]` is `false`, it means the element `nums[i]` is not marked, add it to the score.
  2. Mark this element as visited along with its left and right adjacent elements by setting the corresponding `vis` entries to `true`.
- Continue this process for the entire array `idx`.
- Finally, return the accumulated score as the result.

In this approach, the initial sorting has a **time complexity** of $O(n \log n)$, and the iteration has a **time complexity** of $O(n)$, resulting in a total complexity of $O(n \log n)$. The **space complexity** is $O(n)$ due to the additional index and visited arrays.

Both methods leverage efficient data structures to systematically reduce our problem into smaller sub-problems that can be solved in a stepwise fashion.

Example Walkthrough

Let's consider a simple example to demonstrate how the solution approach works. Suppose we have the following array `nums: [4, 1, 3, 2]`. We want to calculate the score based on the rules specified.

Here's how we would apply the **Priority Queue (Min Heap)** approach step by step:

1. Initialize our priority queue with pairs `(value, index)` and the `visited` array:
   - Priority Queue: `[(4, 0), (1, 1), (3, 2), (2, 3)]` (will be represented as a heap internally).
   - Visited: `[False, False, False, False]`
2. The priority queue works as a min heap and will sort the pairs to `[(1, 1), (2, 3), (3, 2), (4, 0)]` with the smallest pair `(1, 1)` at the top.
3. Pop `(1, 1)` from the heap since `1` is the smallest unmarked element and its index is `1`. Add its value to the score (initially `0`), so score = `1`.
4. Mark element at index `1` and its neighbors as visited:
   - Visited: `[False, True, True, False]`
5. Pop all marked elements from the top of the heap. In this case, the next element `(3, 2)` would be popped out, as it's marked visited.
6. The heap now effectively has `[(2, 3), (4, 0)]`. Next, we pop `(2, 3)` because it is the smallest unmarked element. Add its value to the score, so score = `1 + 2 = 3`.
7. Mark element at index `3` and its neighbors as visited. Since there's no element to the right of index `3`, only index `3` is marked:
   - Visited: `[False, True, True, True]`
8. Again, we pop all marked elements from the top of the heap. Since `(4, 0)` is at the top and is the last unmarked element, we process it next.
9. Add the value of the last element to the score, so score = `3 + 4 = 7`.
10. Mark the last element as visited:
    - Visited: `[True, True, True, True]`
11. The priority queue is now empty, so we exit the loop and return the final score `7`.

Following the above steps, we would be able to find the result using the Priority Queue (Min Heap) approach. The final score as per our steps for the given array `[4, 1, 3, 2]` is `7`.

Python Solution

```python
from heapq import heapify, heappop # We need these functions to work with a priority queue (min-heap).

class Solution:
    def findScore(self, nums: List[int]) -> int:
        # Calculate the length of the input list.
        length = len(nums)

        # Initialize a visited list to keep track of elements that have been added to the score.
        visited = [False] * length

        # Create a priority queue with pairs (value, index) for each number in nums.
        priority_queue = [(value, index) for index, value in enumerate(nums)]
        heapify(priority_queue)  # Transform list into a heap, in-place, in linear time.

        # Initialize the answer to 0.
        score = 0

        # Process the queue until it's empty.
        while priority_queue:
            # Pop the smallest item from the heap, increasing the score by its value.
            value, index = heappop(priority_queue)
            score += value
            visited[index] = True  # Mark the current index as visited.

            # Mark the neighbors of the current index as visited.
            for neighbor in (index - 1, index + 1):
                if 0 <= neighbor < length:
                    visited[neighbor] = True

            # Remove all elements from the heap whose indices have been visited.
            while priority_queue and visited[priority_queue[0][1]]:
                heappop(priority_queue)

        # Return the calculated score.
        return score
```

Java Solution

```java
class Solution {
    public long findScore(int[] nums) {
        int length = nums.length;
        // Create an array to keep track of visited indices
        boolean[] visited = new boolean[length];
        // Create a priority queue to store the pairs of number and index,
        // sorted by the number value, and in case of a tie, by the index
        PriorityQueue<int[]> priorityQueue = new PriorityQueue<>(
            (a, b) -> a[0] == b[0] ? a[1] - b[1] : a[0] - b[0]);

        // Add all numbers along with their indices to the priority queue
        for (int i = 0; i < length; ++i) {
            priorityQueue.offer(new int[] {nums[i], i});
        }

        long totalScore = 0; // This will hold the final score

        // Process the queue until it's empty
        while (!priorityQueue.isEmpty()) {
            var currentPair = priorityQueue.poll(); // Get the pair with the smallest number
            totalScore += currentPair[0]; // Add the number to the score
            visited[currentPair[1]] = true; // Mark the index as visited

            // Visit the neighbors of the current index and mark them as visited
            for (int neighborIndex : List.of(currentPair[1] - 1, currentPair[1] + 1)) {
                if (neighborIndex >= 0 && neighborIndex < length) {
                    visited[neighborIndex] = true;
                }
            }

            // Discard all pairs from the queue where the index has been visited
            while (!priorityQueue.isEmpty() && visited[priorityQueue.peek()[1]]) {
                priorityQueue.poll();
            }
        }

        // Return the total score calculated
        return totalScore;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    long long findScore(vector<int>& nums) {
        int size = nums.size();
        // 'visited' vector keeps track of whether a number at an index has been added to the score.
        vector<bool> visited(size, false);
        // Pair stores the number and its index in the heap.
        using ValueIndexPair = pair<int, int>;
        // Min-heap to store numbers and their indices based on their value, in ascending order.
        priority_queue<ValueIndexPair, vector<ValueIndexPair>, greater<ValueIndexPair>> min_heap;

        // Populate the heap with the numbers and their indices.
        for (int i = 0; i < size; ++i) {
            min_heap.emplace(nums[i], i);
        }

        long long score = 0;
        // Process the numbers until the heap is empty.
        while (!min_heap.empty()) {
            auto [value, index] = min_heap.top();
            min_heap.pop();
            // Add the value from the top of the min-heap to the score.
            score += value;
            // Mark the current index as visited.
            visited[index] = true;
            // Mark the adjacent numbers as visited as well, as per the problem statement.
            if (index - 1 >= 0) {
                visited[index - 1] = true;
            }
            if (index - 1 >= 0) {
                visited[index - 1] = true;
            }
            // Remove the top element of the heap if it's been visited.
            while (!min_heap.empty() && visited[min_heap.top()[1]]) {
                min_heap.pop();
            }
        }
        // Return the final calculated score.
        return score;
    }
};
```

Typescript Solution

```typescript
interface Pair {
    numberValue: number;
    index: number;
}

function findScore(numbers: number[]): number {
    // Initialize a priority queue with a custom compare function.
    // The priority is based on the number value first, then the index.
    const priorityQueue = new PriorityQueue({
        compare: (a: Pair, b: Pair) =>
            (a.numberValue === b.numberValue ? a.index - b.index : a.numberValue - b.numberValue),
    });

    // Get the length of the input array.
    const length = numbers.length;

    // Create an array to mark visited elements.
    const visited: boolean[] = new Array(length).fill(false);

    // Enqueue all elements along with their index into the priority queue.
    for (let i = 0; i < length; ++i) {
        priorityQueue.enqueue({ numberValue: numbers[i], index: i });
    }

    // Initialize the score to zero.
    let score = 0;

    // Dequeue elements from the priority queue until it's empty.
    while (!priorityQueue.isEmpty()) {
        const { numberValue, index } = priorityQueue.dequeue()!;

        // Skip already visited elements.
        if (visited[index]) {
            continue;
        }

        // Accumulate the score with the selected number's value.
        score += numberValue;

        // Mark the current, previous, and next elements as visited.
        visited[index] = true;
        if (index - 1 >= 0) {
            visited[index - 1] = true;
        }
        if (index + 1 < length) {
            visited[index + 1] = true;
        }

        // Remove all front elements in the queue that have been visited.
        while (!priorityQueue.isEmpty() && visited[priorityQueue.front()!.index]) {
            priorityQueue.dequeue();
        }
    }

    // Return the final score.
    return score;
}
```

Time and Space Complexity

The time complexity of the given code is $O(n * \log n)$, as the algorithm requires inserting all n elements into a min-heap and each insertion has a complexity of $O(\log n)$. Additionally, each element is removed exactly once, which also requires $O(\log n)$ time per element. Hence, combined, the overall complexity is $O(n * \log n)$.

The space complexity is $O(n)$ because we allocate an array `vis` of size n and a heap `q` that also could contain up to n elements at the beginning. Although elements are popped from the heap, it does not grow to more than n entries.