2576. Find the Maximum Number of Marked Indices

Medium Greedy Array Two Pointers Binary Search Sorting

Problem Description

In this problem, you have an array of integers called `nums`. The array uses 0-based indexing, which means the first element is at index 0. Your goal is to maximize the number of indices that are marked following a specific rule.

The rule allows you to pick two different unmarked indices `i` and `j` such that the value at `i` is not more than half of the value at `j` (specifically, `2 * nums[i] <= nums[j]`). Once you find such a pair (`i` and `j`), you can mark both indices.

The key task is to keep performing this operation as many times as you want to maximize the count of marked indices. The problem is asking you to return just this maximum count.

Intuition

To solve this problem, we can think of pairing smaller numbers with larger numbers that are at least twice their value. To simplify the process of finding pairs, we can sort the array `nums`. Once the array is sorted, we only need to consider each number once as a potential smaller number in a pair.

The sorted array guarantees that if a number can be paired, it must be paired with a number that comes after it. So, we can use the two-pointer approach to keep track of potential pairs. One pointer (`i`) starts at the beginning of the array, and the other pointer (`j`) starts in the middle (specifically at `(n + 1) // 2`, since this ensures `j` starts from the second half of the array and we can always find a pair such that `2 * nums[i] <= nums[j]` if possible).

Then, we iterate through the array in a while loop, increasing `j` until we find a number that is at least twice nums[i] in value. If we find such a `j`, we increment the count (`ans`) by 2 because we can mark both indices `i` and `j`. Then we move both pointers forward to try to find the next pair.

We continue doing this until `j` reaches the end of the array, and by that time, `ans` will have the maximum number of indices that can be marked.

Solution Approach

The solution involves sorting the input array `nums` to easily find pairs where one number is at least double the other. Sorting is an integral part of the algorithm because it arranges the numbers in ascending order, allowing us to work with the smallest and largest numbers without having to scan the entire array each time.

After sorting, a two-pointer technique is used. This involves initializing two pointers: `i` at the start of the array and `j` at the midpoint of the array. This positioning helps to find pairs'satisfying the `2 * nums[i] <= nums[j]` condition efficiently.

The solution uses a while loop to iterate through the array starting with the pointers at their initial positions:

```
nums.sort()
n = len(nums)
i, j = 0, (n + 1) // 2
ans = 0
while j < n:
```

In this loop, a nested while loop seeks to find a pair satisfying the condition:

```
while j < n and nums[i] * 2 > nums[j]:
    j += 1
```

If such a `j` is found within the array bounds (i.e., `j < n`), it increments `ans` by 2, marking both `i` and `j` :

```
if j < n:
    ans += 2
i, j = i + 1, j + 1
```

After finding a suitable pair, both pointers are moved forward. This is because, once a number has been used in a valid pair, it cannot be used again, and since the array is sorted, we're guaranteed that there are no more numbers before `i` that can be paired with numbers after `j`.

Finally, when the outer loop concludes (meaning `j` has reached the end of the array), `ans` would have been incremented appropriately for every pair found, and the function returns `ans`, which at this point holds the maximum number of indices that can be marked:

```
return ans
```

This algorithm runs in O(n log n) time complexity due to the initial sorting of the array. The rest of the operations consist of single passes through parts of the array, resulting in an O(n) complexity. Thus, sorting is the most time-consuming part of the algorithm. The space complexity is O(1), not counting the space used by the sorting algorithm, which in most implementations would be O(log n), since no extra space is used outside of the sorting operation.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose our input array is `nums = [1, 2, 3, 8, 4]`.

Our goal is to find pairs of indices `i` and `j` where the value at index `i` is no more than half the value at index `j`, and we want to maximize the count of such indices.

1. **Sort the Array:** First, we sort `nums`. After sorting, the array looks like this: `sorted_nums = [1, 2, 3, 4, 8]`.
2. **Initialize Two Pointers:** We have two pointers, `i` starts at index 0 and `j` starts at index `(5 + 1) // 2 = 3` in the array.
3. **Begin the Two-Pointer Approach:**
   - The two pointer values are `nums[i] = 1` and `nums[j] = 4`. Since `2 * nums[i] = 2 * 1 <= 4 = nums[j]`, the pair `(i, j)` meets the condition, so we can mark both indices. We increment `ans` by 2 and move both pointers forward: `i` to 1 and `j` to 4.
   - We now look at `nums[i] = 2` and `nums[j] = 8`. Since `2 * nums[i] = 2 * 2 <= 8 = nums[j]`, we found another pair. We increment `ans` by 2 once again, making `ans` 4, and move both pointers forward: `i` to 2 and `j` beyond the end of the array.
4. **End of the While Loop:** The condition of the outer while loop, `j < n`, is now false because `j` has moved beyond the end of the array.

At this point, we can no longer form any new pairs and the process ends. The result is `ans = 4`, which is the maximum count of indices we can mark based on the given condition. Thus, the example array `nums` can have a maximum of 4 marked indices by performing the operation as described.

Solution Implementation

Python

```python
class Solution:
    def maxNumOfMarkedIndices(self, nums: List[int]) -> int:
        # Sort the array.
        nums.sort()
        # Get the number of elements in the array.
        num_elements = len(nums)

        # Initialize two pointers.
        # 'left' starts from the beginning and 'right' starts from the middle of the array.
        left, right = 0, (num_elements + 1) // 2
        # Initialize the count of marked indices.
        max_marked_indices = 0

        # Iterate over the array until the 'right' pointer reaches the end.
        while right < num_elements:
            # Move the 'right' pointer until we find an element which is more than twice of the 'left' element.
            while right < num_elements and nums[left] * 2 > nums[right]:
                right += 1

            # If we've found a valid pair, increase the count by two.
            # This is because when nums[left] * 2 <= nums[right], both the 'left' and 'right' indices can be marked.
            if right < num_elements:
                max_marked_indices += 2

            # Move both pointers to the next positions.
            left, right = left + 1, right + 1

        # Return the count of the maximum number of marked indices found.
        return max_marked_indices
```

Java

```java
class Solution {

    // Method to determine the maximum number of marked indices.
    public int maxNumOfMarkedIndices(int[] nums) {
        // Sort the input array.
        Arrays.sort(nums);

        // Get the length of the array.
        int n = nums.length;

        // Initialize the count of the maximum number of marked indices.
        int maxMarkedIndices = 0;

        // Use two-pointer technique to traverse the sorted array
        // where 'i' starts from the beginning and 'j' starts from the middle.
        for (int i = 0, j = (n + 1) / 2; j < n; ++i, ++j) {
            // Move the 'j' pointer forward to find a match such that nums[j] >= nums[i] * 2.
            while (j < n && nums[i] * 2 > nums[j]) {
                ++j;
            }

            // If a match is found, increment the maximum marked indices by 2.
            if (j < n) {
                maxMarkedIndices += 2;
            }
        }

        // Return the total count of maximum marked indices.
        return maxMarkedIndices;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Define the function maxNumOfMarkedIndices returning the maximum number
    // of marked indices following the given rules.
    int maxNumOfMarkedIndices(vector<int>& nums) {
        // Sort the nums vector in non-decreasing order.
        sort(nums.begin(), nums.end());

        // Calculate the total number of elements in nums
        int n = nums.size();

        // Initialize the answer to 0
        int answer = 0;

        // Iterate over the elements, i starts from 0 and j starts from the middle extending to the end
        for (int i = 0, j = (n + 1) / 2; j < n; ++i, ++j) {
            // Increment j until the condition nums[i] * 2 <= nums[j] is met
            while (j < n && nums[i] * 2 > nums[j]) {
                ++j;
            }
            // If j is within bounds, increment answer by 2 to count both i and j as marked indices
            if (j < n) {
                answer += 2;
            }
        }

        // Return the final answer
        return answer;
    }
};
```

TypeScript

```typescript
// Function to find the maximum number of marked indices such that
// for every marked index i, there exists a marked index j where nums[i] * 2 <= nums[j].
function maxNumOfMarkedIndices(nums: number[]): number {
    // Sort the array in ascending order.
    nums.sort((a, b) => a - b);

    // Get the length of the array.
    const lengthOfNums = nums.length;

    // Initialize the answer to 0.
    let maxMarkedCount = 0;

    // Loop through the array starting from the first to the middle element (i)
    // and from the middle to the last element (j).
    // Note: Middle is calculated as half the array length plus one, floor-divided.
    for (let i = 0, j = Math.floor((lengthOfNums + 1) / 2); j < lengthOfNums; ++i, ++j) {

        // Increment j until we find an element that is at least twice nums[i] to satisfy the condition.
        while (j < lengthOfNums && nums[i] * 2 > nums[j]) {
            ++j;
        }

        // If we find such an element, increment the maxMarkedCount by 2.
        if (j < lengthOfNums) {
            maxMarkedCount += 2;
        }
    }

    // Return the maximum count of marked indices that satisfy the condition.
    return maxMarkedCount;
}
```

```python
class Solution:
    def maxNumOfMarkedIndices(self, nums: List[int]) -> int:
        # Sort the array.
        nums.sort()
        # Get the number of elements in the array.
        num_elements = len(nums)

        # Initialize two pointers.
        # 'left' starts from the beginning and 'right' starts from the middle of the array.
        left, right = 0, (num_elements + 1) // 2
        # Initialize the count of marked indices.
        max_marked_indices = 0

        # Iterate over the array until the 'right' pointer reaches the end.
        while right < num_elements:
            # Move the 'right' pointer until we find an element which is more than twice of the 'left' element.
            while right < num_elements and nums[left] * 2 > nums[right]:
                right += 1

            # If we've found a valid pair, increase the count by two.
            # This is because when nums[left] * 2 <= nums[right], both the 'left' and 'right' indices can be marked.
            if right < num_elements:
                max_marked_indices += 2

            # Move both pointers to the next positions.
            left, right = left + 1, right + 1

        # Return the count of the maximum number of marked indices found.
        return max_marked_indices
```

Time and Space Complexity

Time Complexity

The given code has two major components contributing to its time complexity:

1. Sorting the `nums` list, which has a time complexity of `O(n log n)` where `n` is the length of the list.
2. A while-loop that iterates over the sorted list to count the "marked" indices. In the worst case, this loop can iterate at most `n` times, resulting in a time complexity of `O(n)`.

Since step 1 is the most significant factor, the overall time complexity of the function `maxNumOfMarkedIndices` is dominated by the sorting step, making it `O(n log n)`.

Space Complexity

The space complexity of the code involves:

1. The sorted `nums` list. If the sorting is done in-place (as is usual with Python's sort methods), there's a constant space complexity of `O(1)`.
2. The use of a fixed number of auxiliary variables like `i`, `j`, `n`, and `ans`, which contribute a constant amount of additional space, also results in `O(1)`.

Hence, the total space complexity of the function `maxNumOfMarkedIndices` is `O(1)`, reflecting that it operates in constant space outside of the input list.