881. Boats to Save People

Medium Greedy Array Two Pointers Sorting

Leetcode Link

Problem Description

In this problem, we are provided with an array called `people`, where each element represents the weight of a person. We also have an infinite number of boats that can carry a maximum weight of `limit`. Each boat can carry at most two people at the same time, as long as the total weight of these people doesn't exceed the `limit`. Our objective is to determine the minimum number of boats required to carry every person.

Intuition

To solve this problem, we can use a two-pointer technique. The main idea is to pair the heaviest person left with the lightest person that can fit in the same boat with them, thus optimizing the usage of boat space. We start by sorting the `people` array to be able to efficiently pair people.

After sorting, we set two pointers: one at the start (`i`) and one at the end (`j`) of the array. The `i` pointer represents the lightest person left to place on a boat, while `j` represents the heaviest. We loop until our two pointers meet. In each iteration, we try to place the heaviest person (`people[j]`) in a new boat and see if we can also fit the lightest person (`people[i]`) with them without exceeding the `limit`.

- If we can fit both the `i` person and the `j` person on the same boat, we increment the `i` pointer (meaning the `i` person is now on a boat).
- Regardless of whether the `i` person got on the boat or not, we decrement the `j` pointer, indicating that the `j` person is now on a boat, and increment the counter `ans`, which tracks the number of boats used.

We continue this process until every person has been placed on a boat. The counter `ans` gives us the minimum number of boats needed to carry everyone.

Solution Approach

The solution approach for this problem hinges on the efficient use of a two-pointer technique and the algorithm that supports it. Here's a walkthrough of the implementation details:

1. **Sorting the Array**: We begin by sorting the `people` array. Sorting is crucial because it allows us to consider the lightest and heaviest weights that can possibly be paired together. In Python, this is done with the `sort()` method, which sorts the array in increasing order. Sorting is a common preprocessing step in many algorithms to facilitate ordered operations.

2. **Two-Pointer Technique**: After sorting, we initialize two pointers: `i`, pointing to the start, and `j`, pointing to the end of the array. These pointers will move towards each other as we pair up people into boats. The two-pointer technique is a common pattern used to solve problems that involve arrays, especially when looking for pairs that meet a certain criteria, as it is both space efficient and has a linear time complexity in this context.

3. **Iterating and Pairing**: As we iterate through the array, we check if the person at the `i`-th position (lightest remaining) can be paired with the person at the `j`-th position (heaviest remaining) without exceeding the weight `limit`.

   - If the sum of weights `people[i] + people[j]` is less than or equal to `limit`, it means both can share one boat. We then move the `i` pointer up (incrementing it by 1) to consider the next lightest person for the subsequent iterations.
   - The `j` pointer is moved down (decremented by 1) after every iteration since the `j`-th person is always placed in a boat (alone if not with the `i`-th person).

4. **Counting Boats**: We have a counter, `ans`, which is incremented in every iteration of the loop. This counter represents the number of boats used.

5. **Loop Termination**: The loop continues until `i > j`, which means all persons have been paired up and placed on boats. At this point, the value of `ans` represents the minimum number of boats required to carry all persons.

The algorithm uses no additional data structures and operates directly on the input array. Therefore, the space complexity is O(1), as it only uses a fixed amount of extra space. The time complexity, after sorting, is O(n), where n is the number of people, since every person is considered exactly once when pairing them.

Example Walkthrough

Let us consider a small example to illustrate the solution approach. Suppose we have an array `people` given as `[3, 5, 3, 4]` and the `limit` of the boats is `5`. We want to find out the minimum number of boats needed to carry everyone following the solution approach detailed above.

Step 1: Sorting the Array

First, we sort the array, which becomes `[3, 3, 4, 5]`.

Step 2: Initializing Two Pointers

We set our two pointers `i` and `j`. The pointer `i` starts at index `0` and `j` starts at index `3`, the last index in the sorted array.

Step 3: Iterating and Pairing Persons to Boats

1. We compare the weights at `i` and `j`, which are `people[i] = 3` and `people[j] = 5`. Since their sum is `8` which is greater than `limit`, they cannot go together. We need a boat for `people[j]`, so we decrement `j` to `2`.
2. Our pointers are now at `people[i] = 3` and `people[j] = 4`. Their sum is `7`, which again exceeds the `limit`. So, we place `people[j]` in a new boat and decrement `j` to `1`.
3. Now `people[i] = 3` and `people[j] = 3`. Their sum is `6`, still exceeding the limit. The third boat is used for `people[j]`, and we decrement `j` to `0`.
4. Finally, the pointers `i` and `j` both point to the first element, which is `3`. We have no choice but to use a fourth boat for `people[i]`.

Step 4: Counting Boats

We needed one boat for each step, so the total number of boats used is `4`.

Step 5: Loop Termination

The loop terminates when `i` is greater than `j`. This happens after the fourth iteration in our example.

Using the approach, we determined that the minimum number of boats required to carry everybody in the array `[3, 5, 3, 4]` with a limit of `5` is `4`. This example demonstrates the efficiency of the two-pointer technique in minimizing the number of boats used by attempting to pair the lightest and heaviest people on the same boat without surpassing the weight limit.

Python Solution

```python
from typing import List

class Solution:
    def num_rescue_boats(self, people: List[int], limit: int) -> int:
        # Sort the list of people to organize by weight for optimal pairing.
        people.sort()
        # Initialize the count of rescue boats needed.
        num_boats = 0
        # Two pointers to keep track of the lightest and heaviest person not yet on a boat.
        lightest_index, heaviest_index = 0, len(people) - 1

        # Continue until all people have been assigned boats.
        while lightest_index <= heaviest_index:
            # If the lightest and heaviest person can share a boat, increase the lightest pointer.
            if people[lightest_index] + people[heaviest_index] <= limit:
                lightest_index += 1
            # Always decrease the heaviest pointer since the heaviest person gets on a boat.
            heaviest_index -= 1
            # Increment the boat count as either one or two people have been assigned to a boat.
            num_boats += 1

        # Return the total number of boats needed.
        return num_boats

# Example usage:
# solution = Solution()
# print(solution.num_rescue_boats([3, 2, 2, 1], 3)) # Output: 3
```

Java Solution

```java
import java.util.Arrays; // Import Arrays class to use the sort method

class Solution {

    /**
     * Returns the minimum number of boats required to rescue people.
     *
     * @param people An array representing the weight of each person.
     * @param limit Maximum weight capacity a boat can carry.
     * @return The minimum number of boats required.
     */
    public int numRescueBoats(int[] people, int limit) {
        // Sort the array of people to organize weights in ascending order
        Arrays.sort(people);

        // Initialize the count of boats to 0
        int boatCount = 0;

        // Use two-pointer technique to pair the lightest and heaviest people
        for (int lighter = 0, heavier = people.length - 1; lighter <= heavier; --heavier) {
            // If the lightest and heaviest persons can share a boat, increment lighter pointer
            if (people[lighter] + people[heavier] <= limit) {
                ++lighter;
            }
            // A boat is used, increment boat count
            ++boatCount;
        }

        // Return the total number of boats required after going through all the people
        return boatCount;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Method to find out the number of boats necessary to rescue all people.
    // "people" is the list of weights of the people, and "limit" is the weight limit of the boat.
    int numRescueBoats(vector<int>& people, int limit) {
        // First, sort the list of people by their weights.
        sort(people.begin(), people.end());

        // Initialize the counter for the number of boats needed.
        int numBoats = 0;

        // Use two pointers, one at the beginning (lightest person) and one at the end (heaviest person).
        int i = 0;
        int j = people.size() - 1;

        // Iterate until all people have been considered.
        while (i <= j) {
            // If the lightest and the heaviest person can share a boat, increment the pointer to the next lightest person.
            if (people[i] + people[j] <= limit) {
                ++i;
            }
            // A boat is used for the heaviest person, decrement the pointer to the next heaviest person.
            --j;

            // Increment the counter for boats as either one or two people have used a boat.
            numBoats++;
        }

        // Return the total number of boats needed.
        return numBoats;
    }
};
```

Typescript Solution

```typescript
// Function to find out the number of boats necessary to rescue all people.
// `people` is the list of weights of the people, and `limit` is the weight limit of the boat.
function numRescueBoats(people: Array<number>, limit: number): number {
    // First, sort the list of people by their weights in non-decreasing order.
    people.sort((a, b) => a - b);

    // Initialize the counter for the number of boats needed.
    let numBoats: number = 0;

    // Use two pointers, one at the beginning (lightest person) and one at the end (heaviest person).
    let i: number = 0;
    let j: number = people.length - 1;

    // Iterate until all people have been considered.
    while (i <= j) {
        // If the lightest and the heaviest person can share a boat, increment the pointer to the next lightest person.
        if (people[i] + people[j] <= limit) {
            i++;
        }
        // A boat is used for the heaviest person, decrement the pointer to the next heaviest person.
        j--;

        // Increment the counter for boats as either one or two people have used a boat.
        numBoats++;
    }

    // Return the total number of boats needed.
    return numBoats;
}
```

Time and Space Complexity

The given Python code implements an efficient algorithm to find the minimum number of boats required to rescue people based on their weights and the limit of a boat's capacity.

Time Complexity

The time complexity of the code is determined by the sorting operation and the while loop that is used to pair people onto boats.

- The `people.sort()` operation has a time complexity of $O(n \log n)$, where n is the length of the `people` list. Sorting is typically achieved using algorithms like Timsort in Python, which has this complexity.
- The while loop runs in $O(n)$ time since in the worst case, it could iterate over all elements once (where n is the number of people). Each operation inside the loop (checking condition and incrementing/decrementing counters) is $O(1)$.

Combining both operations, the overall time complexity of the code is $O(n \log n) + O(n)$, which simplifies to $O(n \log n)$ as the $O(n \log n)$ term dominates $O(n)$ when n is large.

Space Complexity

The space complexity of the algorithm is mainly due to the additional space required for sorting the `people` list.

- The `people.sort()` operation is typically done in-place in Python, meaning that it doesn't require additional space proportional to its input, so it's $O(1)$ space complexity.
- The variables `i`, `j`, and `ans` use a constant amount of extra space $O(1)$.

Thus, the overall space complexity of the code is $O(1)$, indicating that it uses a constant amount of space regardless of the input size.