2491. Divide Players Into Teams of Equal Skill

Medium Array Hash Table Two Pointers Sorting

Leetcode Link

Problem Description

In this problem, you are tasked with dividing players with certain skills into teams. The players are represented by a positive integer array called `skill`, which has an even number of elements denoted by `n`. Each element in the array represents the skill level of a player, with `skill[i]` corresponding to the `i`-th player's skill.

The players must be divided into `n / 2` teams, with each team consisting of two players. The key condition for forming these teams is that the total skill (sum of the skill levels of both team members) must be the same for every team.

The **chemistry** of a team is defined as the product of the skill levels of the two players in that team. The objective is to calculate and return the sum of the chemistry of all teams. However, if it is impossible to divide the players into teams that all have an equal total skill, you should return `-1`.

Intuition

The intuition behind the solution is to first sort the `skill` array in increasing order. Sorting is useful because it helps in easily pairing players in such a way that might meet the condition of equal total skill for each team.

After sorting, we set a target total skill value, which is the sum of the first and last element in the sorted array. Considering the array is sorted, this target value should be what we aim to match for each team. We set two pointers, one at the start and one at the end of the sorted array, and start pairing players by moving these pointers towards each other.

At each iteration, we check whether the sum of the skills of the two selected players (pointed by `i` and `j`) equals the target skill value (`t`). If it doesn't match the target, then we know immediately that it's impossible to form the teams with an equal total skill and we return `-1`.

If it matches, we compute the chemistry (product of the skills) for this pair of players and add it to a running total (`ans`). We then move our pointers inward (increment `i` and decrement `j`) and continue this process.

We repeat these steps until our pointers meet in the middle. If we are able to pair up all the players in such a way that each team's total skill matches the target, then at the end, we will have successfully found the sum of the chemistry of all teams and return it. If at any point the condition fails, we know that forming the teams as per the condition is not possible, hence the result will be `-1`.

Solution Approach

The solution uses a straightforward approach, leveraging sorting and the two-pointer technique to efficiently pair players into teams with equal total skills.

Here's a breakdown of the algorithm:

1. **Sorting**: The array `skill` is sorted in ascending order. This step ensures that the smallest and largest elements are at the beginning and end of the array, respectively, thus allowing us to easily find pairs that may equal our target total skill.
2. **Setting a target total skill (`t`)**: Once sorted, we take the sum of the first (`skill[0]`) and last (`skill[-1]`) elements to set our target total skill. This forms our baseline which all other pairs must match to ensure all teams have the same total skill.
3. **Two-Pointer Technique**: We use a classic two-pointer approach where one pointer (`i`) starts from the beginning of the array, and the other pointer (`j`) starts from the end. We then iterate through the array, with the following steps in our loop:
   - Check if the sum of the current pair of players (`skill[i] + skill[j]`) equals the target total skill (`t`). If not, we immediately return `-1`, indicating the task is impossible.
   - If the pair matches the target, we calculate the chemistry (product) for this team (`skill[i] * skill[j]`) and add it to our running total (`ans`).
   - Move the pointers closer to each other (`i` increments by 1 and `j` decrements by 1) to form the next team with the subsequent smallest and largest elements.
4. **Loop until pointers meet**: The loop continues until `i` is no longer less than `j`. This means we have checked and formed teams with all players in the sorted array.
5. **Return the result**: If we're able to go through the entire array without finding a mismatch in total skills, we then return the accumulated sum of chemistry (`ans`).

Through this method, we avoid the need for complex data structures and leverage a simple yet effective algorithmic pattern to find the solution. The key to our approach is the greedy strategy of pairing the smallest and largest remaining players to match the target total skill after sorting, ensuring that if there is a possible solution, we will find it.

Example Walkthrough

Let's consider an example to illustrate the solution approach. We have an array `skill = [4, 7, 6, 5]`.

1. **Sorting**: The first step is to sort this array in ascending order. So, after sorting, the array becomes `skill = [4, 5, 6, 7]`.
2. **Setting a target total skill (`t`)**: The target total skill is determined by the sum of the first and last elements in the sorted array. Here, `t = skill[0] + skill[-1]` which is `t = 4 + 7 = 11`.
3. **Two-Pointer Technique**: We set two pointers, `i` at the beginning (`i = 0`) and `j` at the end (`j = 3`). Now, we iterate through the array using these pointers with the following steps:
   - The sum of the current pair is `skill[i] + skill[j]`, which is `4 + 7 = 11`. This equals our target `t`. Since the condition is met, we calculate the chemistry for this team, which is `4 * 7 = 28`.
   - We add this chemistry to our running total (`ans`), so `ans = 28`.
   - Move the pointers: we increment `i` to `1` and decrement `j` to `2`, and then check the next pair.
4. **Next Iteration**: Now `i = 1` and `j = 2`. The sum of this new pair of players is `skill[1] + skill[2]`, which is `5 + 6 = 11`. Again, this matches our target `t`.
   - The chemistry for this team is `5 * 6 = 30`.
   - We add it to our running total, so now `ans = 28 + 30 = 58`.
   - Since we have no more elements left to pair, we have reached the end of our iteration.
5. **Return the result**: As we have successfully formed teams such that each team's total skill equals the target total skill, and there were no mismatches, we return the sum of the chemistry of all teams, which is `ans = 58`.

Therefore, for the given example `skill = [4, 7, 6, 5]`, the function would return `58`.

Python Solution

```python
from typing import List

class Solution:
    def dividePlayers(self, skills: List[int]) -> int:
        # Sort the list of skills in ascending order
        skills.sort()

        # Calculate the target sum of skills for a pair of players
        target_sum = skills[0] + skills[-1]

        # Initialize pointers at the beginning and end of the list
        left, right = 0, len(skills) - 1

        # Initialize variable to store the cumulative product of valid pairs
        cumulative_product = 0

        # Loop through the skills list using two pointers from both ends towards the center
        while left < right:
            # If the current pair of players doesn't have the desired sum, return -1
            if skills[left] + skills[right] != target_sum:
                return -1

            # Calculate the product of the current valid pair's skills
            cumulative_product += skills[left] * skills[right]

            # Move the pointers towards the center for the next potential pair
            left += 1
            right -= 1

        # Return the cumulative product of all valid pairs' skills
        return cumulative_product

# Example usage:
# solution = Solution()
# result = solution.dividePlayers([4, 5, 1, 2, 3, 6])
# print(result)  # Output would depend on the logic correctness and input values
```

Java Solution

```java
class Solution {

    public long dividePlayers(int[] skillLevels) {
        // Sort the skill levels array to organize players by their skill
        Arrays.sort(skillLevels);

        // Get the number of players
        int numOfPlayers = skillLevels.length;

        // Calculate the target sum based on the lowest and highest skill levels
        int targetSum = skillLevels[0] + skillLevels[numOfPlayers - 1];

        // Initialize the answer variable to store the sum of products
        long answer = 0;

        // Use two pointers to iterate from the beginning and end towards the center
        for (int left = 0, right = numOfPlayers - 1; left < right; ++left, --right) {
            // Check if the current pair of players does not meet the target sum
            if (skillLevels[left] + skillLevels[right] != targetSum) {
                return -1; // Return -1 if condition is not met, indicating invalid pairing
            }

            // Calculate the product of the skill levels of the two players and add it to the answer
            answer += (long) skillLevels[left] * skillLevels[right];
        }

        // Return the final answer which is the sum of products of the pairs
        return answer;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for sort

class Solution {
public:
    long long dividePlayers(vector<int>& skillLevels) {
        // Sort the skill levels in ascending order
        sort(skillLevels.begin(), skillLevels.end());

        int numPlayers = skillLevels.size(); // Number of players
        int teamBalancingFactor = skillLevels[0] + skillLevels[numPlayers - 1];
        long long productSum = 0; // Variable to store the sum of product of skills

        // Pair players starting from the weakest and the strongest
        for (int left = 0, right = numPlayers - 1; left < right; ++left, --right) {
            // If the pair doesn't add up to the balancing factor, teams can't be balanced
            if (skillLevels[left] + skillLevels[right] != teamBalancingFactor) {
                return -1;
            }
            // Add up product of the skills of the two players
            productSum += static_cast<long long>skillLevels[left] * skillLevels[right];
        }

        // Return the total sum of products of the skills for all pairs
        return productSum;
    }
};
```

Typescript Solution

```typescript
function dividePlayers(skillLevels: number[]): number {
    // Determine the number of players
    const numOfPlayers = skillLevels.length;

    // Sort the skillLevels in ascending order
    skillLevels.sort((a, b) => a - b);

    // Set target sum as the sum of the lowest and highest skill levels
    const targetSum = skillLevels[0] + skillLevels[numOfPlayers - 1];

    // Initialize answer variable to store the product sum
    let productSum = 0;

    // Iterate over the first half of the sorted array
    for (let i = 0; i < numOfPlayers / 2; i++) {
        // Check if the current pair does not sum up to the target sum
        if (targetSum !== skillLevels[i] + skillLevels[numOfPlayers - 1 - i]) {
            // If they don't, pairing is not possible, return -1
            return -1;
        }

        // Accumulate the products of the pair into the productSum
        productSum += skillLevels[i] * skillLevels[numOfPlayers - 1 - i];
    }

    // Return the final product sum as the answer
    return productSum;
}
```

Time and Space Complexity

The time complexity of the `dividePlayers` function is primarily determined by the sort operation. In Python, the sort method typically uses Timsort, an algorithm with a time complexity of $O(n \log n)$, where n is the number of elements in the input list `skill`. Once the list is sorted, the function enters a while-loop, which iterates approximately $n/2$ times (since each iteration pairs up one element from the start of the list with one from the end). The operations inside the loop are constant time operations, so they do not add more than $O(n)$ to the time complexity. As a result, the overall time complexity of the function is dominated by the sorting step: $O(n \log n)$.

For space complexity, the code does not create any additional data structures that grow with the size of the input list; it only uses a fixed number of variables. Therefore, the space complexity is $O(1)$, implying constant space usage excluding the input list.