2244. Minimum Rounds to Complete All Tasks

Medium Greedy Array Hash Table Counting

Leetcode Link

Problem Description

You are given an array of integers called `tasks`, where each integer in the array represents the difficulty level of a specific task. The index of the array starts from 0. The challenge requires you to find a way to complete all the given tasks by grouping them into rounds. In each round, you have the option to complete 2 or 3 tasks, but there's a catch: all tasks in a round must have the same difficulty level.

The primary goal is to figure out the minimum number of rounds you need to complete all the tasks. However, if you find that it's not possible to complete all of them under the given constraints (for example, if there is a task that cannot be grouped into 2 or 3 because it is the only one of its difficulty level), the answer should be -1. The task is essentially asking for an efficient way to group these tasks into rounds of 2 or 3 such that no tasks are left ungrouped.

Intuition

To solve this problem, we need to focus on how to efficiently group the tasks. It's apparent that a single task cannot be grouped by itself (since we need at least 2), so any task with a frequency of 1 immediately makes the problem impossible, and we return -1.

If there's more than one task of the same difficulty, we should complete them in as few rounds as possible. Naturally, we should try to group them in threes first, since that will require fewer rounds than grouping in twos.

The intuition behind the solution is that we'll count the frequency of each task's difficulty level first. Then for each task difficulty, if it occurs only once, we cannot complete it under the given constraints and return -1. If the task occurs more than once, we calculate the number of rounds needed by first doing as many groups of three as possible (by taking the integer division of the count by three), and then check if there's a remainder. If there's a remainder after dividing by three (meaning one or two tasks are left), we'll need one more round to complete those.

This approach ensures we do the minimum number of rounds, since we prioritize completing tasks in groups of three before resorting to groups of two.

Solution Approach

The implementation of the solution in Python uses a `Counter` from the `collections` module to determine the frequency of each difficulty level in the `tasks` array. The `Counter` object will give us a dictionary where each key is a unique task difficulty level and the corresponding value is the number of times this difficulty appears in the `tasks` list.

Here's a breakdown of the algorithm and data structures used in the solution:

1. **Counter to record frequencies**: We use `Counter(tasks)` to count the occurrences of each difficulty level. This is crucial because the approach to solve this problem depends heavily on how many times each task difficulty appears.
2. **Looping through the counts**: We then iterate over the values of this `Counter` object using `for v in cnt.values()`. Here `v` will be the number of times a particular difficulty level occurs.
3. **Checking for impossibility**: Inside the loop, we immediately check if `v == 1` meaning the task cannot be completed in 2 or 3 rounds because there is only one task of this difficulty. We return `-1` because it's not possible to complete all tasks under the given constraints.
4. **Calculating rounds**: If there are two or more tasks of the same difficulty level, we calculate the number of rounds required to complete all tasks of this difficulty. First, we do `v // 3` which gives us the number of rounds where we can complete 3 tasks at a time. Because it's integer division, any leftover tasks (either 1 or 2) would not be counted in this division.
5. **Accounting for leftovers**: After that, we have `v % 3 != 0` which checks whether there's a remainder when dividing the count by three. If there's a remainder, we need an additional round to complete the leftover tasks. Note that the leftover tasks can only be 1 or 2 since any count of 3 or more would have been taken care of by the previous division.
6. **Summing up the rounds**: The total rounds for a difficulty level are the sum of rounds with 3 tasks and, if required, one additional round for leftovers. This total number of rounds is accumulated in the `ans` variable by `ans += v // 3 + (v % 3 != 0)`.
7. **Returning the result**: After iterating through all task difficulties and accumulating the rounds required for each, we return the sum stored in `ans`.

The algorithm efficiently solves the problem by maximizing the number of tasks completed in each round, ensuring that the answer reflects the minimum number of rounds needed. It also correctly identifies and handles the case when the problem constraints make it impossible to complete the tasks.

Example Walkthrough

Let's say we have the array `tasks = [4, 3, 3, 4, 3]`, which contains the difficulty levels of the tasks that need to be completed.

First, we count the frequencies of each task difficulty using a `Counter`. So after applying `Counter(tasks)`, we get `{4:2, 3:3}`. This tells us that we have two tasks with difficulty 4 and three tasks with difficulty 3.

Now let's follow the algorithm with these counts:

1. For difficulty 4, the count is 2. It perfectly fits in one round where we can complete 2 tasks. So no tasks are left over, and we set `ans = 1` for this difficulty.
2. For difficulty 3, the count is 3. We can group these tasks into one round of 3 tasks. Since there's no remainder, all tasks of difficulty 3 are completed in `3 // 3 = 1` round, and we update `ans = ans + 1 = 2`.

Thus, for the input `tasks = [4, 3, 3, 4, 3]`, the minimum number of rounds needed to complete all tasks is `2`.

By following the algorithm, we efficiently grouped the tasks into the minimum number of rounds (2 in this case), ensuring that no tasks are left ungrouped. If at any point we had encountered a task count of `1`, we would have returned `-1`, indicating it's not possible to complete all tasks under the given constraints. However, in this example, we were able to group all tasks into rounds of 2 or 3 with no tasks left over, thus successfully completing the challenge.

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def minimumRounds(self, tasks: List[int]) -> int:
        # Count the frequency of each task using Counter
        task_count = Counter(tasks)

        # Initialize the number of minimum rounds required to 0
        minimum_rounds = 0

        # Iterate over the frequency of each unique task
        for frequency in task_count.values():

            # If there's exactly one task, it cannot be completed in rounds of 2 or 3
            if frequency == 1:
                return -1  # Return -1 as it's not possible to complete the task

            # Calculate the full rounds of 3's that can be completed
            full_rounds = frequency // 3

            # Check if there's a remainder when the frequency is divided by 3
            has_remainder_round = (frequency % 3 != 0)

            # Increment the minimum rounds by full rounds:
            # add 1 more round if there's a remainder (since remainder can be either 1 or 2)
            minimum_rounds += full_rounds + has_remainder_round

        # Return the calculated minimum rounds needed to complete the tasks
        return minimum_rounds

# Example usage:
# sol = Solution()
# print(sol.minimumRounds([2, 2, 3, 3, 2, 4, 4, 4, 4, 4]))  # Should return 4
```

Java Solution

```java
class Solution {
    public int minimumRounds(int[] tasks) {
        // Hash map to store the frequency of each task
        Map<Integer, Integer> taskCounts = new HashMap<>();

        // Populate the map with the frequency of tasks
        for (int task : tasks) {
            taskCounts.merge(task, 1, Integer::sum);
        }

        // Variable to hold the minimum number of rounds needed
        int minRounds = 0;

        // Iterate through the values of the map, which are the frequencies
        for (int count : taskCounts.values()) {
            // If any task is only done once, it's not possible to form a round, hence return -1
            if (count == 1) {
                return -1;
            }

            // Rounds are formed in groups of 3 or 2, so we divide by 3 and then add 1 if there's a remainder (2 or 1)
            // This is the least number of rounds to complete the tasks
            minRounds += count / 3 + (count % 3 == 0 ? 0 : 1);
        }

        // Return the calculated minimum rounds
        return minRounds;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    int minimumRounds(vector<int>& tasks) {
        // Create a hash map (unordered_map) to count occurrences of each task
        unordered_map<int, int> taskCounts;
        for (const auto& task : tasks) {
            ++taskCounts[task];
        }

        int totalRounds = 0; // Initialize total rounds needed to 0

        // Iterate through the map to calculate rounds required for each unique task
        for (const auto& [task, count] : taskCounts) {
            // If a task occurs only once, it is not possible to complete it, return -1
            if (count == 1) {
                return -1;
            }

            // Calculate the rounds required for the current task
            // The task can be completed in 3-round sets or a 2-round set if necessary
            int roundsForTask = count / 3 + (count % 3 != 0);
            totalRounds += roundsForTask; // Add to the total rounds
        }

        return totalRounds; // Return the total number of rounds needed to complete all tasks
    }
};
```

Typescript Solution

```typescript
// TypeScript code to calculate the minimum number of rounds required
// to complete each task at least twice, given some conditions.

// Import the necessary types from 'types' module, if TypeScript type definitions are necessary.

// Function that computes the minimum number of rounds needed to complete tasks
function minimumRounds(tasks: number[]): number {
    // Use a Map to count occurrences of each task
    const taskCounts = new Map<number, number>();
    tasks.forEach((task) => {
        taskCounts.set(task, (taskCounts.get(task) || 0) + 1);
    });

    let totalRounds = 0; // Initialize the total number of rounds needed to 0

    // Iterate through the task counts to calculate the required number of rounds
    taskCounts.forEach((count, task) => {
        // If a task occurs only once, it's not possible to complete it twice
        if (count === 1) {
            // Signal failure as it's impossible to complete this task in any round
            totalRounds = -1;
            return;
        }

        // Calculate the number of rounds for the current task count
        // A task can be completed in either 3-round sets or 2-round sets if needed
        const roundsForTask = Math.floor(count / 3) + (count % 3 > 0 ? 1 : 0);
        totalRounds += roundsForTask; // Accumulate the total rounds needed
    });

    // Return the total number of rounds, or -1 if any task is not completable
    return totalRounds;
}

// Example usage:
// console.log(minimumRounds([1, 2, 3, 1, 2, 3, 2, 2]));  // Output might be 6 rounds
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the following parts:

1. The creation of the counter `cnt` from the list of tasks, which involves iterating over all elements of the list to count the occurrences of each unique task. This process has a complexity of $O(n)$, where n is the number of tasks in the input list.
2. The iteration over the values of the counter `cnt`. In the worst case, this is also $O(n)$ since each task might be unique. However, typically the number of unique tasks will be less than the total number of tasks.
3. The calculation of the minimum number of rounds for each task, which is constant time $O(1)$ per task.

Therefore, the overall time complexity of the given code is $O(n)$.

Space Complexity

The space complexity is determined by:

1. The space required to store the counter `cnt`, which will have as many entries as there are unique tasks. In the worst case, each task is unique; hence the space complexity is $O(n)$.
2. The additional space used by the variable `ans`, which is $O(1)$.

Combining these considerations, the overall space complexity of the code is $O(n)$.