517. Super Washing Machines

Hard Greedy Array

Problem Description

In this problem, you are given `n` super washing machines, each initially containing a certain number of dresses. The machines are aligned in a row, and you can make a move where you choose any `m` machines (where `1 <= m <= n`) and pass one dress from each chosen machine to one of its adjacent machines simultaneously.

Your goal is to find the minimum number of moves required to redistribute the dresses so that all the machines have the same number of dresses. If it's not possible to reach this balance, the function should return `-1`.

To summarize:

- You have an array `machines` representing the number of dresses in each washing machine.
- You need to make all the washing machines have an equal number of dresses using the fewest moves possible.
- A move consists of choosing a subgroup of machines and shifting a dress from each one to an adjacent machine.
- You must determine the minimum moves required, or return `-1` if equal distribution isn't possible.

Intuition

To solve this problem, one must understand that if there is an average number of dresses per machine that isn't a whole number, it's impossible to equally distribute the dresses since you can't split a dress. This is the first check made: if the total number of dresses is not divisible by the number of machines, the method will immediately return `-1`.

Assuming equal distribution is possible, our goal is to identify how far each machine is from the average and to calculate the number of steps needed to equalize the distribution.

The solution does this by:

- Calculating the average number of dresses per machine (`k`).
- Initializing two variables: `s`, which will maintain the cumulative difference from the average at each step, and `ans`, which will hold the maximum number of moves observed so far.

With each iteration over the machines, we:

- Subtract the average `k` from the current number of dresses to find how many dresses need to be passed onto or received from adjacent machines (`x -= k`).
- Add this difference to the cumulative sum `s`, which shows the total imbalance after the current machine.
- Update `ans` with the maximum of its current value, the absolute current imbalance `abs(s)`, and the current machine's specific imbalance `x`.

The absolute current imbalance `abs(s)` represents a sequence of redistributions among machines up to the current point.

Finally, `ans` is the accumulated aggregate of moves required, which gets returned as the minimum number of moves to make all washing machines have the same number of dresses.

Solution Approach

The solution uses a greedy algorithm to redistribute the dresses across the washing machines. The main principle behind the algorithm is that, at each step, the algorithm looks for the best local move that reduces the imbalance without caring for a global plan. To implement this, we use a single pass through the machines array, accumulating the necessary information to determine the minimum number of moves.

Here's a step-by-step breakdown of the solution approach:

1. **Calculate Total and Average**: Determine the total number of dresses (`sum(machines)`) and the average (`k`), which is the target number for each machine to reach. Simultaneously, check if the total sum is divisible by the number of machines (`n`), which is necessary for an equal distribution. If not, we know immediately that the problem has no solution (`return -1`).
2. **Initialization**: Initialize two variables: `s` is set to 0 and will be used to track the running imbalance as we iterate through the machines, and `ans` is also set to 0 and will keep a record of the maximum number of moves needed at any point.
3. **Iterate Through Machines**: Iterate through each washing machine using `x` to represent the number of dresses in the current machine.
4. **Calculate the Local Imbalance**: For each machine, calculate how many dresses it needs to give away or receive to reach the average by subtracting the average `k` from `x` (`x -= k`).
5. **Update the Cumulative Imbalance**: Accumulate the current machine's imbalance to `s` (`s += x`). This gives us a cumulative sum at each point which represents the net dresses that need to be moved after considering the redistribution up to that machine.
6. **Determine Maximum Moves**: After each adjustment, update `ans` to be the maximum of:
   - The previous maximum number of moves `ans`
   - The absolute value of the current cumulative imbalance `abs(s)`, which indicates the minimum moves required for the current and all previous machines to achieve balance.
   - The current machine's specific surplus or deficit `x`, which shows the minimum moves the current machine alone needs irrespective of the cumulative sum.
7. **Return**: After processing all machines, you're left with the maximum value in `ans` that represents the minimum moves required to balance the dresses across all washing machines.

The algorithm uses a very straightforward data structure - a simple array to represent the machines and their respective counts of dresses. The pattern employed here is largely one of accumulation and tracking the maximum requirement, which is typical of problems where you must consider a running total and update an answer based on local constraints (in this case, the machine's individual and cumulative imbalance).

This approach effectively balances the load across all machines in the minimum number of moves, taking into account that multiple redistributions might be happening simultaneously during each move.

Example Walkthrough

Let's assume we have an array of washing machines where `[1, 0, 5]` represents the number of dresses in the respective machines. Let's walk through the solution approach step-by-step.

1. **Calculate Total and Average**:
   - The total sum of dresses is `1 + 0 + 5 = 6`.
   - The number of machines, `n`, is `3`.
   - Calculate the average dresses per machine, `k`, which is `sum / n = 6 / 3 = 2`.
   - The total number of dresses is evenly divisible by the number of machines, so a solution is possible.
2. **Initialization**:
   - We set `s = 0` and `ans = 0`.
3. **Iterate Through Machines**:
   - We will iterate through each machine, starting from the first.
4. **Calculate the Local Imbalance** and **Update the Cumulative Imbalance**:
   - For the first machine, `x = 1`. We need `x` to be equal to the average `k = 2`. Thus, `x -= k` gives us `-1`. The machine needs 1 dress.
   - Update `s` with `x`. Now, `s = -1`.
   - Determine maximum moves and update `ans`. Ans will be the max of `ans`, `abs(s)`, and `abs(x)`, so `ans = max(0, 1, 1) = 1`.
5. **Iterate to the Next Machine**:
   - For the second machine, `x = 0`. To reach the average `k`, it needs 2 dresses. Thus, `x -= k` gives us `-2`.
   - Update `s` with `x`. Now, `s += (-2) => s = -3`.
   - Update `ans`. `ans = max(1, abs(-3), abs(-2)) = 3`.
6. **Iterate to the Third Machine**:
   - For the third machine, `x = 5`. To reach the average `k`, it must give away 3 dresses. Thus, `x -= k` gives us `3`.
   - Update `s` with `x`. Now, `s += 3 => s = 0`. All machines are now balanced.
   - Update `ans`. `ans = max(3, abs(0), abs(3)) = 3`.
7. **Return**:
   - The iteration is complete and the maximum value in `ans` is `3`.
   - The algorithm returns `3` as the minimum number of moves required to balance the dresses across all washing machines.

In summary, the example demonstrates how the algorithm checks for the possibility of equal distribution, iterates to calculate individual and cumulative imbalances and continuously updates the number of moves required based on the maximum imbalance observed at each step. The final answer of `3` moves accounts for the most demanding redistribution that occurs during the process.

Solution Implementation

Python

```python
class Solution:
    def findMinMoves(self, machines: List[int]) -> int:
        # Calculate the length of the machines list.
        machine_count = len(machines)

        # Compute total dresses and the expected dresses per machine, along with a remainder.
        total_dresses, remainder = divmod(sum(machines), machine_count)

        # If there is a remainder, we can't equally distribute dresses to all machines.
        if remainder:
            return -1

        # Initialize variables for the answer and the running balance.
        max_moves = 0
        balance = 0

        # Iterate over each machine.
        for dresses_in_machine in machines:
            # Calculate the number of dresses to move for this machine to reach the expected count.
            dresses_to_move = dresses_in_machine - total_dresses

            # Increment the balance, which represents the ongoing "debt" or "surplus" from left to right.
            balance += dresses_to_move

            # The maximum number of moves required is the maximum of three values:
            # 1. Current max_moves (the max encountered so far)
            # 2. The absolute balance (as we may need to move dresses across multiple machines)
            # 3. Dresses to move for the current machine (as it may require a lot of adding/removing)
            max_moves = max(max_moves, abs(balance), dresses_to_move)

        # Return the maximum number of moves needed to balance all machines.
        return max_moves
```

Java

```java
class Solution {
    public int findMinMoves(int[] machines) {
        int totalDresses = 0; // Sum of all dresses across the machines

        // Calculate the total number of dresses
        for (int dresses : machines) {
            totalDresses += dresses;
        }

        // If total number of dresses is not divisible by the number of machines, there is no solution
        if (totalDresses % machines.length != 0) {
            return -1;
        }

        // Calculate the average number of dresses per machine
        int averageDresses = totalDresses / machines.length;

        // Initialize variables to track the current imbalance and the maximum number of moves required
        int imbalance = 0;
        int maxMoves = 0;

        // Iterate over each machine to calculate the required moves
        for (int dresses : machines) {
            // The number of dresses to be moved from the current machine to reach the average
            dresses -= averageDresses;

            // Update the imbalance of dresses after considering the current machine's contribution
            imbalance += dresses;

            // The maximum number of moves is the maximum of three quantities:
            // 1. Current maximum number of moves
            // 2. The absolute value of the current imbalance
            // 3. The number of dresses to be moved from the current machine
            maxMoves = Math.max(maxMoves, Math.max(Math.abs(imbalance), dresses));
        }

        // Return the maximum number of moves required to balance the machines
        return maxMoves;
    }
}
```

C++

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

class Solution {
public:
    int findMinMoves(vector<int>& machines) {
        int totalDresses = accumulate(machines.begin(), machines.end(), 0); // Sum of all dresses in machines
        int numMachines = machines.size(); // Number of machines

        // If the total number of dresses is not divisible by the number of machines,
        // it is impossible to balance the machines with an equal number of dresses.
        if (totalDresses % numMachines != 0) {
            return -1;
        }

        // Calculate the average number of dresses per machine for balanced state
        int averageDresses = totalDresses / numMachines;

        // Initialize cumulative imbalance and the answer (max number of moves required)
        int cumulativeImbalance = 0;
        int maxMoves = 0;

        // Iterate over each machine
        for (int dressesInMachine : machines) {
            // Calculate the imbalance for the current machine
            int imbalance = dressesInMachine - averageDresses;

            // Update the cumulative imbalance
            cumulativeImbalance += imbalance;

            // The minimum number of moves required is the maximum of three values:
            // 1. The current maxMoves,
            // 2. The absolute value of cumulative imbalance (for adjustments across machines),
            // 3. The current imbalance (if a machine requires more moves on its own).
            maxMoves = max({maxMoves, abs(cumulativeImbalance), imbalance});
        }

        // Return the minimum number of moves required to balance all machines.
        return maxMoves;
    }
};
```

TypeScript

```typescript
function findMinMoves(machines: number[]): number {
    const totalMachines = machines.length;
    let totalDresses = machines.reduce((accumulated, current) => accumulated + current, 0);

    // If the total number of dresses cannot be evenly distributed, return -1.
    if (totalDresses % totalMachines !== 0) {
        return -1;
    }

    const dressesPerMachine = Math.floor(totalDresses / totalMachines);
    let imbalance = 0;  // Used to track the imbalance of dresses during distribution.
    let minMoves = 0;   // The minimum number of moves required to balance.

    // Iterate through each machine.
    for (let dressesInMachine of machines) {
        // Calculate the number of excess dresses in current machine.
        dressesInMachine -= dressesPerMachine;
        // Update the imbalance by adding the excess dresses from the current machine.
        imbalance += dressesInMachine;

        // The minimum moves is the maximum of the current minMoves,
        // the absolute imbalance, and the excess dresses in the current machine.
        minMoves = Math.max(minMoves, Math.abs(imbalance), dressesInMachine);
    }

    // Return the minimum number of moves required to balance the machines.
    return minMoves;
}
```


```python
class Solution:
    def findMinMoves(self, machines: List[int]) -> int:
        # Calculate the length of the machines list.
        machine_count = len(machines)

        # Compute total dresses and the expected dresses per machine, along with a remainder.
        total_dresses, remainder = divmod(sum(machines), machine_count)

        # If there is a remainder, we can't equally distribute dresses to all machines.
        if remainder:
            return -1

        # Initialize variables for the answer and the running balance.
        max_moves = 0
        balance = 0

        # Iterate over each machine.
        for dresses_in_machine in machines:
            # Calculate the number of dresses to move for this machine to reach the expected count.
            dresses_to_move = dresses_in_machine - total_dresses

            # Increment the balance, which represents the ongoing "debt" or "surplus" from left to right.
            balance += dresses_to_move

            # The maximum number of moves required is the maximum of three values:
            # 1. Current max_moves (the max encountered so far)
            # 2. The absolute balance (as we may need to move dresses across multiple machines)
            # 3. Dresses to move for the current machine (as it may require a lot of adding/removing)
            max_moves = max(max_moves, abs(balance), dresses_to_move)

        # Return the maximum number of moves needed to balance all machines.
        return max_moves
```


Time and Space Complexity

The provided code examines an array of `machines` representing the number of dresses in each laundry machine and calculates the minimum number of moves required to equalize the number of dresses in all machines. Here's the computational complexity analysis:

- **Time Complexity:**

  The function `findMinMoves` involves a single `for` loop that iterates over the list of `machines` exactly once. Within this loop, only constant-time operations are performed: subtraction, addition, comparison, and assignment. Therefore, the time complexity is directly proportional to the length of the `machines` list, which is `n`.

  The final time complexity is `O(n)`.

- **Space Complexity:**

  The space complexity is calculated based on the additional space used by the algorithm aside from the input itself. Here, only a few integer variables are used (`n`, `k`, `mod`, `ans`, and `s`). There are no data structures that grow with the size of the input.

  Therefore, the space complexity is `O(1)` for the constant extra space used.