1599. Maximum Profit of Operating a Centennial Wheel

Medium Array Simulation Leetcode Link

Problem Description

You're in charge of a Ferris wheel (the Centennial Wheel) that has four gondolas, and each gondola can hold up to four people. You can rotate the wheel counterclockwise to bring one gondola down to the ground to let people on and off, but each rotation costs some amount of money (`runningCost`). There are `n` groups of customers, represented by an array `customers`, where `customers[i]` indicates how many new customers are ready to board the Ferris wheel just before the `i`th rotation. You must rotate the wheel to serve these customers, but no new customer should wait if there's room available on the gondola. The customers also pay you a certain amount (`boardingCost`) every time they board the gondola. Your goal is to figure out the minimum number of rotations you need to perform to maximize your profit from operating the Ferris wheel. If you cannot make a profit, the answer should be `-1`.

Intuition

To solve the problem, we need to consider the cost of running a rotation against the revenue from boarding customers. We calculate profit incrementally with each customer and each rotation, taking into account the existing customers waiting to board. The profit at any step is the total money earned from boarding passengers minus the total cost of rotations so far.

We continue rotating the wheel, serving customers, and updating the total profit until there are no customers waiting or no more groups arriving. We also keep track of the maximum profit encountered and the number of rotations it took to reach that profit.

To arrive at the solution, we need to maintain a few variables as we process each group of customers: the current waiting number of customers (`wait`), the total profit (`t`), and the current rotation index (`i`). For each group, we serve up to 4 customers (or fewer if we have fewer waiting). We then calculate the profit and update the maximum profit (`mx`) if the current profit exceeds it. We remember the rotation at which the maximum profit was achieved (`ans`).

After processing all customers, we return the rotation count when the maximum profit was achieved. If there was never a positive profit, we return `-1`.

Solution Approach

The implementation of the solution is quite straightforward and iterative. It hinges on a loop that simulates the operation of the wheel, serving customers group by group and calculating profits.

- We initiate a while loop that continues as long as there are waiting customers (`wait`) or there are still groups of customers we haven't processed (controlled by checking if index `i` is less than the length of `customers`).
- On each iteration, if the index `i` is within bounds of the array, we add the number of new customers arriving (`customers[i]`) to the number of waiting customers (`wait`).
- We simulate boarding customers onto the gondola by taking the smaller of the number of waiting customers (`wait`) or the maximum capacity of a gondola (`4`). This number is stored in the variable `up`.
- We then subtract the number of customers who have boarded from the waiting queue (`wait`).
- The total profit (`t`) is updated by adding the revenue from boarding customers (`up * boardingCost`) and subtracting the running cost (`runningCost`).
- The rotation counter (`i`) is incremented as we have completed a rotation.
- If the total profit after this rotation exceeds the maximum recorded profit (`mx`), we update `mx` and also record the number of rotations it took to reach this new maximum profit with `ans = i`.
- Lastly, after the loop finishes executing, we return the `ans`. If `mx` (maximum profit) has never been positive, then `ans` would still be `-1`, indicating no profitable scenario was encountered.

Variables used:

- `wait`: an integer to keep track of the number of waiting customers.
- `up`: an integer representing the number of customers that board the wheel in each rotation.
- `t`: an integer to keep the running total of profit.
- `mx`: an integer to store the maximum profit achieved at any point.
- `ans`: an integer to store the number of rotations at which `mx` was achieved.
- `i`: an integer used as an index to iterate over the array of customers and also to count the number of rotations.

Algo/Pattern used:

- The pattern used here is a simple iteration over the customer's array and updating states(`wait`, `t`, `mx`, `ans`) based on the customers served and the cost incurred in each rotation.
- The algorithm continuously compares the running profit with the maximum observed profit to identify the optimal stopping point.
- The loop ensures that all customers are served if profitable by not terminating until `wait` is empty, meaning no customers are left waiting.

This approach requires no complex data structures and is efficient, with a time complexity of `O(n)` where `n` is the number of customer groups, because it passes through the customer list only once.

Example Walkthrough

Let's consider an example where the `runningCost` is `5`, the `boardingCost` is `6`, and we have `customers` array `[8, 1]`.

Following the solution approach:

1. Initially, we set `wait = 0`, `t = 0`, `mx = -1`, `ans = -1`, and `i = 0`.
2. In the first rotation (i = 0), we start with `customers[0] = 8`. The `wait` gets updated to `8`. Since the capacity of a gondola is `4`, `up = min(wait, 4) = 4`.
3. Now, `4` customers board, so `wait` becomes `8-4 = 4`. We calculate profit by `t = t + (up * boardingCost) - runningCost` which becomes `0 + (4 * 6) - 5 = 19`. Since `19 > mx`, we update `mx = 19` and `ans = i + 1 = 1`.
4. Rotation i is incremented, `i = 1`.
5. In the second rotation (i = 1), the `customers[1] = 1` is added to `wait`, so the `wait` becomes `4 + 1 = 5`. Again, `up = min(wait, 4) = 4`.
6. `4` customers board, so `wait` becomes `5 - 4 = 1`. Update the profit: `t = 19 + (4 * 6) - 5 = 38`. Now `38 > mx`, so we update `mx = 38` and `ans = i + 1 = 2`.
7. Increase rotation index, `i = 2`.
8. In the third rotation (i = 2), there are no more customers in the array, but we still have `1` customer waiting. `up = min(wait, 4) = 1`.
9. `1` customer boards, `wait` becomes `1 - 1 = 0`. Update the profit: `t = 38 + (1 * 6) - 5 = 39`. The profit `mx` remains unchanged as `39` is not greater than `38`, so no updates to `mx` or `ans`.
10. Since there are no more customers waiting or in the array, we stop.

At the end of these rotations, the maximum profit `mx` was `38`, which occurred after `2` rotations. Hence, the answer is `ans = 2`.

Python Solution

```python
from typing import List

class Solution:
    def min_operations_max_profit(self, customers: List[int], boarding_cost: int, running_cost: int) -> int:
        # Initialize necessary variables
        max_profit = total_profit = 0
        waiting_customers = 0
        rotations = 0
        best_rotation = -1  # Set to -1 as default when no solution is found

        # Iterate over the customers list and continue as long as there are waiting customers or unprocessed days
        while waiting_customers > 0 or rotations < len(customers):
            # Add customers from the current day if it exists
            if rotations < len(customers):
                waiting_customers += customers[rotations]

            # Calculate the number of customers that can board (up to 4)
            boarding_customers = min(waiting_customers, 4)
            waiting_customers -= boarding_customers

            # Update total profit
            total_profit += boarding_customers * boarding_cost - running_cost

            # Check and store the maximum profit and corresponding rotation
            if total_profit > max_profit:
                max_profit = total_profit
                best_rotation = rotations + 1  # Add 1 for the 1-indexed result

            # Move to the next rotation
            rotations += 1

        # Return the best rotation for maximum profit or -1 if no profit is ever reached
        return best_rotation

```

Java Solution

```java
class Solution {

    public int minOperationsMaxProfit(int[] customers, int boardingCost, int runningCost) {
        // Initialize variables to store:
        // 'maxProfit': the current maximum profit seen (initialized to 0).
        // 'totalOperations': the total number of operations to achieve the 'maxProfit'.
        // 'waitingCustomers': the number of customers waiting to board.
        // 'currentRotation': the current rotation/round of the gondola.
        int maxProfit = 0;
        int totalOperations = -1; // Begins at -1 to handle cases when no profit can be made.
        int waitingCustomers = 0;
        int currentRotation = 0;

        // Loop through all the customers or until there are no more waiting customers.
        while (waitingCustomers > 0 || currentRotation < customers.length) {
            // Add the customers arriving in the current rotation to 'waitingCustomers'.
            if (currentRotation < customers.length) {
                waitingCustomers += customers[currentRotation];
            }

            // Calculate the number of people to board in this rotation.
            // It should not exceed 4, which is the gondola's capacity.
            int boardingCustomers = Math.min(4, waitingCustomers);

            // Decrease the count of 'waitingCustomers' by the number of people who just boarded.
            waitingCustomers -= boardingCustomers;

            // Move to the next rotation.
            currentRotation++;

            // Calculate the total profit after this rotation.
            int profitThisRotation = boardingCustomers * boardingCost - runningCost;

            // Add the profit from this rotation to the total profit.
            maxProfit += profitThisRotation;

            // Check if the total profit we just calculated is greater than the previously recorded maximum profit.
            // If it is, update the 'maxProfit' and 'totalOperations'.
            if (maxProfit > 0 && maxProfit > maxProfit) {
                maxProfit = maxProfit;
                totalOperations = currentRotation;
            }
        }

        // Return the total number of operations needed to reach maximum profit.
        // If a profit cannot be made, 'totalOperations' would be -1.
        return totalOperations;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include necessary library for vector usage
#include <algorithm> // Include necessary library for using min function

class Solution {
public:
    int minOperationsMaxProfit(std::vector<int>& customers, int boardingCost, int runningCost) {
        int maxProfit = -1; // Initialize variable to store the maximum profit index
        int currentProfit = 0; // Current profit initialized to zero
        int totalWaitingCustomers = 0; // Counter for customers waiting
        int rotations = 0; // Counter for the number of rotations
        int currentlyBoarded; // Customers that can board the ride on current rotation

        // Loop until we've processed all customers or until there are no more waiting customers
        while (totalWaitingCustomers > 0 || rotations < customers.size()) {
            // Add customers from the current rotation if it exists
            if (rotations < customers.size()) {
                totalWaitingCustomers += customers[rotations];
            }

            // Board up to 4 customers or the number of waiting customers, whichever is lower
            currentlyBoarded = std::min(4, totalWaitingCustomers);
            // Subtract the boarded customers from the waiting queue
            totalWaitingCustomers -= currentlyBoarded;
            // Move to the next rotation
            rotations++;

            // Update the current profit by adding profit from boarding customers and subtracting the running cost
            currentProfit += currentlyBoarded * boardingCost - runningCost;

            // If the current profit is greater than the max profit, update maxProfit and store the current rotation
            if (currentProfit > maxProfit) {
                maxProfit = currentProfit;
                maxProfit = rotations; // Store the most profitable rotation
            }
        }

        // If maxProfit remains -1, it means that running the ride was never profitable; otherwise, return the number of rotations
        return (maxProfit > 0 ? maxProfit : -1);
    }
};

```

Typescript Solution

```typescript
import { min } from "lodash"; // Assuming lodash is used for the `min` function, otherwise native Math.min can be used.

// This variable holds the maximum profit calculated, initialized with -1 indicating no profit has been made.
let maxProfit: number = -1;

// Function to calculate number of operations needed to reach maximum profit from the given boarding and running costs.
// Takes in an array of customers, boarding cost per person, and running cost per ride.
function minOperationsMaxProfit(customers: number[], boardingCost: number, runningCost: number): number {
    // Stores the current profit made, initialized to 0.
    let currentProfit: number = 0;
    // Counts the total number of customers waiting in line.
    let totalWaitingCustomers: number = 0;
    // Counts how many rotations the ride has been through.
    let rotations: number = 0;
    // Number of customers that can board the ride in the current rotation.
    let currentlyBoarded: number;

    // Continue looping until all customers are processed or there are no more waiting customers.
    while (totalWaitingCustomers > 0 || rotations < customers.length) {
        // If there are remaining rotations, add customers to the waiting total.
        if (rotations < customers.length) {
            totalWaitingCustomers += customers[rotations];
        }

        // Board the number of customers that is the lesser of available seats (4) or waiting customers.
        currentlyBoarded = min([4, totalWaitingCustomers])!;
        // Reduce the number of total waiting customers by the amount that just boarded the ride.
        totalWaitingCustomers -= currentlyBoarded;
        // Increment rotation count, moving to the next cycle.
        rotations++;

        // Update the running total of current profit by adding profit from customers times the boarding
        // cost and subtracting the cost of running the ride.
        currentProfit += currentlyBoarded * boardingCost - runningCost;

        // If current profit is greater than what's recorded in maxProfit, update maxProfit to
        // the newest profit and set the count of rotations at which max profit occurred.
        if (currentProfit > maxProfit) {
            maxProfit = currentProfit; // Update the max profit to the new profit.
            maxProfit = rotations; // This should actually store the rotation count, needs correction.
        }
    }

    // If maxProfit is still -1, it means running the ride never turned profitable.
    // Otherwise, return the optimized number of rotations to reach max profit.
    return (maxProfit > 0 ? maxProfit : -1);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of elements in the `customers` list. This is because the loop runs for each customer in the `customers` list plus additional iterations for any remaining waiting customers after the end of the list. The operations inside the loop are constant time, which means they do not depend on the size of the input list, so the time complexity is linear with regard to the number of elements in the input.

Space Complexity

The space complexity of the code is `O(1)`. The function uses a fixed number of variables (`ans`, `mx`, `t`, `wait`, `i`, and `up`) whose space requirement does not scale with the input size (the number of customers). Hence, the space used by the algorithm is constant.