2398. Maximum Number of Robots Within Budget

Hard · Queue · Array · Binary Search · Prefix Sum · Sliding Window · Heap (Priority Queue) · Leetcode Link

Problem Description

You are given `n` robots, each with a specific time it takes to charge (`chargeTimes[i]`) and a cost that it incurs when running (`runningCosts[i]`). Your goal is to determine the maximum number of consecutive robots that can be active without exceeding a given budget. The total cost to run `k` robots is calculated by adding the maximum charge time from the selected `k` robots to the product of `k` and the sum of their running costs.

The problem is essentially asking you to find the longest subsequence of robots that can operate concurrently within the constraints of your budget, taking into consideration both the upfront charge cost and the ongoing running costs.

Intuition

The solution utilizes the sliding window technique to find the longest subsequence of consecutive robots that can be run within the budget. To efficiently manage the running sum of the running costs and the maximum charge time within the current window, we can use a deque (double-ended queue) and maintain the sum of the running costs.

Sliding Window Technique

We iterate through the robots using a sliding window defined by two pointers, `j` (the start) and `i` (the end). For each position `i`:

1. A deque is used to keep track of the indices of the robots in the current window, maintaining the indices in decreasing order of their `chargeTimes`.
2. We add the robot at position `i` to the window. If it has a charge time greater than some robots already in the window, those robots are removed from the end of the deque because they are rendered irrelevant (a larger charge time has been found).
3. We add the running cost of the current robot to a running sum `s`.
4. We check if the current window exceeds the budget by calculating the total cost using the front of the deque (which has the robot with the maximum charge time) and the running sum `s`. If it does exceed the budget, we shrink the window from the left by increasing `j` and adjusting the sum and deque accordingly.
5. The answer (`ans`) is updated as we go, to be the maximum window size found that satisfies the budget constraint.

By the end of the iteration, `ans` holds the length of the longest subsequence of robots we can operate without exceeding the budget.

Solution Approach

The implementation uses a greedy approach combined with a sliding window technique to determine the maximum number of consecutive robots that can be run within the budget. Here's a step-by-step breakdown:

1. **Initialization**: A deque `q` is declared to keep track of the robots' indices in the window while ensuring access to the largest `chargeTime` in constant time. Variable `s` is used to store the running sum of `runningCosts`, `j` is the start of the current window, `ans` is the variable that will store the final result, and `n` holds the length of the arrays.

2. **Sliding Window**: This implementation uses a single loop that iterates over all robots' indices `i` from `0` to `n-1`. For each index `i`:

   - Insert the current robot into the deque:
     - If there are robots in the deque with `chargeTimes` less than or equal to the current robot's `chargeTime`, they are removed from the end since they do not affect the max anymore.
     - The current index `i` is added to the end of the deque.
   - The running cost of the robot `i` is added to the running sum `s`.

3. **Maintaining the Budget Constraint**: The algorithm checks if the total cost of running robots from the current window exceeds the budget:

   - While the sum of the maximum `chargeTime` (from the front of the deque) and the product of `s` and the window size (`i - j + 1`) is greater than `budget`, the window needs to be shrunk from the left. This includes:
     - If the robot at the start of the window is also at the front of the deque, it is removed.
     - The running cost of the robot at `j` is subtracted from `s`.
     - The start index `j` is incremented to shrink the window.

4. **Update the Result**: After fixing the window by ensuring it's within the budget, update the maximum number of robots `ans` by comparing it with the size of the current window.

5. **Return the Result**: After the loop finishes, the variable `ans` holds the length of the longest segment of consecutive robots that can be run without exceeding the budget, according to the specified cost function.

The algorithm's time complexity is O(n), where n is the number of robots, since each element is inserted and removed from the deque at most once. The usage of a deque enables the algorithm to determine the maximum `chargeTime` in O(1) time while keeping the ability to insert and delete elements from both ends efficiently. The running sum `s` is also updated in constant time, making this approach efficient for large inputs.

Example Walkthrough

Let's go through a small example to illustrate the solution approach.

Suppose we have `n = 4` robots with the following `chargeTimes` and `runningCosts`:

```
chargeTimes = [3, 6, 1, 4]
runningCosts = [2, 1, 3, 2]
```

And let's say our budget is `16`.

Now let's apply our sliding window technique:

1. Initialize our variable `s` to 0, our deque `q` to empty, `ans` to 0, and our window start `j` to 0.

2. Start iterating with the sliding window:

   i. Start with the first robot `i = 0`: - `chargeTimes[i]` is 3; we add it to our deque `q` (which is now [0]). - We add `runningCosts[i]` to `s` (which is now 2). - The window size is `i - j + 1` which is `1` at this point. - The total cost calculation is `max(chargeTimes)` (which is 3 from the deque) + `s * window size` = 3 + 2 * 1 = 5, which is within the budget.

   ii. Move to the next robot `i = 1`: - `chargeTimes[i]` is 6; since it's larger than the last element in the deque, we push it to the deque `q` (which is now [0,1]). - Add `runningCosts[i]` to `s` (which is now 3). - The calculation now is `max(chargeTimes)` (which is 6) + `s * window size` (2 * 2) = 6 + 4 = 10, still within the budget.

   iii. Move to robot `i = 2`: - `chargeTimes[i]` is 1, it's less than the elements in the deque, so nothing is removed and it's added to `q` (now [0,1,2]). - Add `runningCosts[i]` to `s` (which is now 6). - The total cost is `max(chargeTimes)` (which is still 6) + `s * window size` (3 * 3) = 6 + 9 = 15, within the budget.

   iv. Move to robot `i = 3`: - `chargeTimes[i]` is 4, which is less than 6 but more than 3, so we pop from the end of the deque until the condition is satisfied and then add index 3. Deque q is now [0,1,3]. - Add `runningCosts[i]` to `s` (which is now 8). - The total cost now is `max(chargeTimes)` (which is still 6) + `s * window size` (4 * 4) = 6 + 16 = 22, which exceeds the budget.

3. To maintain the budget constraint:

   i. When the cost exceeds the budget (which is the case now), we need to shrink our window from the left: - Since `j = 0` is at the front of the deque, remove it from the deque `q` (now [1,3]). - Subtract `runningCosts[j]` from `s` to update the running sum (now 6). - Increment `j` to 1 to shrink the window. - Now, the window size is 3 (from index 1 to 3), and the cost calculation is `max(chargeTimes)` (which is 6) + `s * window size` = 6 + 6 * 3 = 24, still over budget.

   ii. Repeat the process by incrementing `j` to 2 and adjust:`q` (now [3]), `s` to 3, and `ans` remains 2 which was the biggest window size that was in budget before.

   iii. The current window size is now `i - j + 1` which is 2, and the total cost is `max(chargeTimes)` (which is 4) + `s * window size` = 4 + 3 * 2 = 10, within the budget. - We update `ans` to the current window size, but since it's not bigger than the previous `ans` (which was 2), `ans` remains 2.

4. After iterating through all robots, our answer `ans` is `2` which indicates we can run at most 2 consecutive robots within the given budget.

By following this approach, we manage to calculate the maximum number of consecutive robots that can be active within the allocated budget in an efficient manner, with a loop that runs linearly relative to the number of robots.

Python Solution

```python
from collections import deque

class Solution:
    def maximum_robots(self, charge_times, running_costs, budget):
        # Deque to store indices of robots with charge_times in non-increasing order
        max_charge_deque = deque()
        # Holds the sum of the running costs of the current set of robots
        running_cost_sum = 0
        # Stores the maximum number of robots that can be activated
        max_robots = 0
        # Starting index of the current window of robots
        start_idx = 0

        # Iterate over all the robots
        for i in range(len(charge_times)):
            current_charge = charge_times[i]
            current_running_cost = running_costs[i]

            # Remove robots from the back of the deque if their charge time is less than
            # or equal to the current robot's charge time
            while max_charge_deque and charge_times[max_charge_deque[-1]] <= current_charge:
                max_charge_deque.pop()
            # Add the current robot's index to the deque
            max_charge_deque.append(i)

            # Add the current robot's running cost to the sum
            running_cost_sum += current_running_cost

            # Ensure the sum of the max charge time in the window and the total running cost
            # for the robots in the window does not exceed the budget
            while (max_charge_deque and
                    charge_times[max_charge_deque[0]] + (i - start_idx + 1) * running_cost_sum > budget):
                # If the robot at the front of the deque is the robot at start_idx, remove it
                if max_charge_deque[0] == start_idx:
                    max_charge_deque.popleft()
                # Remove the robot's running cost at start_idx from the sum and move start_idx forward
                running_cost_sum -= running_costs[start_idx]
                start_idx += 1

            # Update the maximum number of robots that can be activated
            max_robots = max(max_robots, i - start_idx + 1)

        return max_robots
```

Java Solution

```java
class Solution {

    // Method to calculate the maximum number of robots that can be activated within the budget
    public int maximumRobots(int[] chargeTimes, int[] runningCosts, long budget) {
        // Queue to store indices of robots to ensure chargeTimes are in non-increasing order from front to back
        Deque<Integer> queue = new ArrayDeque<>();

        // Total number of robots
        int numOfRobots = chargeTimes.length;

        // Running sum of the costs
        long runningSum = 0;

        // Starting index for the current window
        int windowStart = 0;

        // Result, i.e., the maximum number of robots that can be activated
        int maxRobots = 0;

        // Loop over each robot
        for (int i = 0; i < numOfRobots; ++i) {
            // Current robot's charge time and running cost
            int currentChargeTime = chargeTimes[i];
            int currentRunningCost = runningCosts[i];

            // Remove robots from the back of the queue whose charge time is less than or equal to the current one
            while (!queue.isEmpty() && chargeTimes[queue.getLast()] <= currentChargeTime) {
                queue.pollLast();
            }

            // Add the current robot to the queue
            queue.offer(i);

            // Update the running sum with the current robot's running cost
            runningSum += currentRunningCost;

            // If the total cost exceeds the budget, remove robots from the front of the queue
            while (!queue.isEmpty() && chargeTimes[queue.getFirst()] + (i - windowStart + 1) * runningSum > budget) {
                if (queue.getFirst() == windowStart) {
                    queue.pollFirst(); // Remove the robot at the start of the window if it is at the front of the queue
                }
                runningSum -= runningCosts[windowStart++]; // Reduce the running sum and move the window start forward
            }

            // Update the result with the maximum number of robots
            maxRobots = Math.max(maxRobots, i - windowStart + 1);
        }

        // Return the maximum number of robots
        return maxRobots;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <deque>
using std::vector;
using std::deque;
using std::max;

class Solution {
public:
    // Finds the maximum number of robots that can be activated within a given budget
    int maximumRobots(vector<int>& chargeTimes, vector<int>& runningCosts, long long budget) {
        // Deque to store indices of robots with chargeTimes in non-increasing order
        deque<int> maxChargeDeque;
        // Holds the sum of the running costs of the current set of robots
        long long runningCostSum = 0;
        // Stores the maximum number of robots that can be activated
        int maxRobots = 0;
        // Starting index of the current window of robots
        int startIdx = 0;
        int numRobots = chargeTimes.size();

        // Iterate over all the robots
        for (int i = 0; i < numRobots; ++i) {
            int currentCharge = chargeTimes[i];
            int currentRunningCost = runningCosts[i];

            // Remove robots from the back of the deque if their charge time is less than
            // or equal to the current robot's charge time
            while (!maxChargeDeque.empty() && chargeTimes[maxChargeDeque.back()] <= currentCharge) {
                maxChargeDeque.pop_back();
            }
            // Add the current robot's index to the deque
            maxChargeDeque.push_back(i);

            // Add the current robot's running cost to the sum
            runningCostSum += currentRunningCost;

            // Ensure the sum of the max charge time in the window and the total running cost
            // for the robots in the window does not exceed the budget
            while (!maxChargeDeque.empty() && chargeTimes[maxChargeDeque.front()] + (i - startIdx + 1) * runningCostSum > budget) {
                // If the robot at the front of the deque is the robot at the startIdx, remove it
                if (maxChargeDeque.front() == startIdx) {
                    maxChargeDeque.pop_front();
                }
                // Remove the robot's running cost at startIdx from the sum and move the startIdx forward
                runningCostSum -= runningCosts[startIdx++];
            }

            // Update the maximum number of robots that can be activated
            maxRobots = max(maxRobots, i - startIdx + 1);
        }

        return maxRobots;
    }
};
```

Typescript Solution

```typescript
function maximumRobots(chargeTimes: number[], runningCosts: number[], budget: number): number {
    // Deque to store indices of robots with chargeTimes in non-increasing order
    const maxChargeDeque: number[] = [];
    // Holds the sum of the running costs of the current set of robots
    let runningCostSum = 0;
    // Stores the maximum number of robots that can be activated
    let maxRobots = 0;
    // Starting index of the current window of robots
    let startIdx = 0;
    const numRobots = chargeTimes.length;

    // Iterate over all the robots
    for (let i = 0; i < numRobots; ++i) {
        const currentCharge = chargeTimes[i];
        const currentRunningCost = runningCosts[i];

        // Remove robots from the back of the deque if their charge time is less than
        // or equal to the current robot's charge time
        while (maxChargeDeque.length > 0 && chargeTimes[maxChargeDeque[maxChargeDeque.length - 1]] <= currentCharge) {
            maxChargeDeque.pop();
        }
        // Add the current robot's index to the deque
        maxChargeDeque.push(i);

        // Add the current robot's running cost to the sum
        runningCostSum += currentRunningCost;

        // Ensure the sum of the max charge time in the window and the total running cost
        // for the robots in the window does not exceed the budget
        while (maxChargeDeque.length > 0 && chargeTimes[maxChargeDeque[0]] + (i - startIdx + 1) * runningCostSum > budget) {
            // If the robot at the front of the deque is the robot at startIdx, remove it
            if (maxChargeDeque[0] === startIdx) {
                maxChargeDeque.shift();
            }
            // Remove the robot's running cost at startIdx from the sum and move the startIdx forward
            runningCostSum -= runningCosts[startIdx++];
        }

        // Update the maximum number of robots that can be activated
        maxRobots = Math.max(maxRobots, i - startIdx + 1);
    }

    return maxRobots;
}
```

Time and Space Complexity

Time Complexity

The given code maintains a deque (`q`) and iterates over the `chargeTimes` array once. The primary operations within the loop are:

- Adding elements to the deque which takes `O(1)` time per operation, but elements are only added when they are larger than the last element. Since elements are removed from the deque if they are not greater, each element is added and removed at most once.

- Removing elements from the dequeue from both ends also takes `O(1)` time per operation, ensuring that no element is processed more than once.

- The while loop inside the for loop is executed to ensure that the current maximum charge time and total cost do not exceed the budget. Although it seems nested, it does not make the overall algorithm exceed `O(n)` because each element is added to the deque only once, and hence can be removed only once.

Given these observations, each operation is in constant time regarding the current index, and since we only iterate over the array once, the time complexity is `O(n)` where `n` is the length of the `chargeTimes` array.

Space Complexity

The space complexity is primarily determined by the deque `q`, which in the worst case might hold all elements if the `chargeTimes` are in non-decreasing order. Thus, in the worst-case scenario, the space complexity is `O(n)` where `n` is the length of the `chargeTimes` array.

Other variables used (such as `s`, `j`, `a`, `b`, and `ans`) only require constant space (`O(1)`), so do not affect the overall space complexity.