<https://leetcode.com/problems/maximum-number-of-robots-within-budget/description/>

You have n robots. You are given two 0-indexed integer arrays, chargeTimes and runningCosts, both of length n. The ith robot costs chargeTimes[i] units to charge and costs runningCosts[i] units to run. You are also given an integer budget.

The total cost of running k chosen robots is equal to max(chargeTimes) + k \* sum(runningCosts), where max(chargeTimes) is the largest charge cost among the k robots and sum(runningCosts) is the sum of running costs among the k robots.

Return the maximum number of consecutive robots you can run such that the total cost does not exceed budget.

**Example 1:**

**Input:** chargeTimes = [3,6,1,3,4], runningCosts = [2,1,3,4,5], budget = 25

**Output:** 3

**Explanation:**

It is possible to run all individual and consecutive pairs of robots within budget.

To obtain answer 3, consider the first 3 robots. The total cost will be max(3,6,1) + 3 \* sum(2,1,3) = 6 + 3 \* 6 = 24 which is less than 25.It can be shown that it is not possible to run more than 3 consecutive robots within budget, so we return 3.

**Example 2:**

**Input:** chargeTimes = [11,12,19], runningCosts = [10,8,7], budget = 19

**Output:** 0

**Explanation:** No robot can be run that does not exceed the budget, so we return 0.

**Constraints:**

chargeTimes.length == runningCosts.length == n

1 <= n <= 5 \* 10^4

1 <= chargeTimes[i], runningCosts[i] <= 10^5

1 <= budget <= 10^15

**Attempt 1: 2025-01-19**

**Solution 1: Not fixed length Sliding Window + Decreasing Monotonic Deque (60 min)**

**想到用 Decreasing Monotonic Deque 来保证仅用 O(n) 时间复杂度随时更新和保存 Sliding Window 中的最大值是关键，Deque 的前端 (deque.peekFirst()) 存储着当前 Sliding Window 中最大 chargeTime 所对应的坐标，Deque 的后端 (deque.peekLast()) 会随时更新，以保证从 first 到 last 一直满足 monotonically decreasing.**

class Solution {

    public int maximumRobots(int[] chargeTimes, int[] runningCosts, long budget) {

        int maxRobots = 0;

        long runningSum = 0;

        Deque<Integer> deque = new ArrayDeque<>();

        int i = 0;

        for(int j = 0; j < chargeTimes.length; j++) {

            runningSum += runningCosts[j];

            // Maintain monotonic decreasing deque to make sure

            // the deque.peekFirst() always store maximum charge

            // time's array index of the current Sliding Window

            // Note: Include '=' for update case since we always

            // want to keep the most recent index, when we start

            // shrink the window size, the fresh tailing index

            // stored on deque will help ?

            while(!deque.isEmpty() && chargeTimes[deque.peekLast()] <= chargeTimes[j]) {

                deque.pollLast();

            }

            deque.addLast(j);

            long maxChargeTimeInWindow = (long) chargeTimes[deque.peekFirst()];

            long runningCost = (long) (j - i + 1) \* runningSum;

            // Have to shrink the window since over budget

            if(maxChargeTimeInWindow + runningCost > budget) {

                // Remove running cost at index 'i' since we are

                // shrink window and remove index 'i'

                runningSum -= runningCosts[i];

                // Only update (remove) the maximum charge time's

                // array index of current Sliding Window stored

                // at deque.peekFirst() when pending removal index

                // 'i' equals to the stored index, otherwise no

                // need update deque

                if(deque.peekFirst() == i) {

                    deque.pollFirst();

                }

                i++;

            }

            maxRobots = Math.max(maxRobots, j - i + 1);

        }

        return maxRobots;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/maximum-number-of-robots-within-budget/solutions/2524838/java-c-python-sliding-window-o-n-solution/>

**Solution 1: Sliding Window + TreeMap**

Use a sorted data structure to find the maximum value in a sliding window.

Time O(nlogn)

Space O(n)

int maximumRobots(vector<int>& times, vector<int>& costs, long long budget) {

long long i = 0, j, sum = 0, n = times.size();

multiset<int> s;

for (int j = 0; j < n; ++j) {

sum += costs[j];

s.insert(times[j]);

if (\*rbegin(s) + sum \* (j - i + 1) > budget) {

sum -= costs[i];

s.erase(s.find(times[i++]));

}

}

return n - i;

}

**Solution 2: Sliding Window + Mono Deque**

Use a mono deque to find the maximum value in a sliding window.

Time O(n)

Space O(n)

public int maximumRobots(int[] times, int[] costs, long budget) {

long sum = 0;

int i = 0, n = times.length;

Deque<Integer> d = new LinkedList<Integer>();

for (int j = 0; j < n; ++j) {

sum += costs[j];

while (!d.isEmpty() && times[d.peekLast()] <= times[j])

d.pollLast();

d.addLast(j);

if (times[d.getFirst()] + (j - i + 1) \* sum > budget) {

if (d.getFirst() == i)

d.pollFirst();

sum -= costs[i++];

}

}

return n - i;

}

**Refer to**

<https://algo.monster/liteproblems/2398>

**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](https://algo.monster/problems/sliding_window_maximum)**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:

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.

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).

We add the running cost of the current robot to a running sum s.

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.

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](https://algo.monster/problems/sliding_window_maximum) technique to determine the maximum number of consecutive robots that can be run within the budget. Here's a step-by-step breakdown:

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.

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.

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.

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.

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.

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;

}

}

**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.

**Refer to chatGPT**

import java.util.Deque;

import java.util.LinkedList;

class Solution {

    public int maximumRobots(int[] chargeTimes, int[] runningCosts, long budget) {

        int n = chargeTimes.length;

        Deque<Integer> deque = new LinkedList<>(); // Monotonic deque to store indices of `chargeTimes`

        long runningSum = 0; // Sum of runningCosts in the current window

        int left = 0; // Left pointer of the sliding window

        int maxRobots = 0; // Maximum number of robots we can activate

        for (int right = 0; right < n; right++) {

            // Add the current robot's runningCost to the running sum

            runningSum += runningCosts[right];

            // Maintain the monotonic deque for `chargeTimes`

            while (!deque.isEmpty() && chargeTimes[deque.peekLast()] <= chargeTimes[right]) {

                deque.pollLast();

            }

            deque.addLast(right);

            // Calculate the total cost for the current window

            long maxChargeTime = chargeTimes[deque.peekFirst()];

            long totalCost = maxChargeTime + (right - left + 1) \* runningSum;

            // Shrink the window if the total cost exceeds the budget

            if (totalCost > budget) {

                // Remove the leftmost element's runningCost from the running sum

                runningSum -= runningCosts[left];

                // Remove the leftmost element from the deque if it is no longer in the window

                if (deque.peekFirst() == left) {

                    deque.pollFirst();

                }

                left++; // Shrink the window

            }

            // Update the maximum number of robots

            maxRobots = Math.max(maxRobots, right - left + 1);

        }

        return maxRobots;

    }

}

**Explanation:**

Sliding Window:

The sliding window [left, right] tracks the range of consecutive robots being considered.

The size of the window represents the number of robots being activated.

Monotonic Deque for Charge Times:

The deque maintains the indices of chargeTimes in decreasing order.

The maximum chargeTimes in the current window can be retrieved in O(1) time by accessing deque.peekFirst().

Total Cost Calculation:

The total cost for the current window is calculated as:
totalCost=maxChargeTime+(windowSize×runningSum)
where:

maxChargeTime is the maximum value of chargeTimes in the window.

runningSum is the sum of runningCosts in the window.

windowSize is the number of robots in the current window.

Shrinking the Window:

If the totalCost exceeds the budget, shrink the window by moving the left pointer forward and updating the runningSum and deque.

Updating the Result:

After ensuring the window satisfies the budget constraint, update the maximum number of robots (maxRobots) with the current window size.

**Refer to**

[L239.P2.1.Sliding Window Maximum (Ref.L1425)](note://6738BA39EFAD452FA70CC8B64512437D)

[L739.Daily Temperatures](note://6FB3A11E2B7746DCA411301F941DD948)

[L2040.Kth Smallest Product of Two Sorted Arrays](note://WEB66e3c42d583656a15cac16c720603209)

[L2064.Minimized Maximum of Products Distributed to Any Store (Ref.L410)](note://WEB2ca95ff77a9fee4bcced70f3e62a41df)

[L2187.Minimum Time to Complete Trips (Ref.L410,L1011,L1283)](note://WEB723d73111b17d88843c05d9aa91556cb)