Problem 2398: Maximum Number of Robots Within Budget

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You have

n

robots. You are given two

0-indexed

integer arrays,

chargeTimes

and

runningCosts

, both of length

n

. The

i

th

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

Code Snippets

C++:

```
class Solution {
public:
int maximumRobots(vector<int>& chargeTimes, vector<int>& runningCosts, long
long budget) {

}
};
```

Java:

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

}
}
```

Python3:

```
class Solution:
def maximumRobots(self, chargeTimes: List[int], runningCosts: List[int],
budget: int) -> int:
```

Python:

```
class Solution(object):
def maximumRobots(self, chargeTimes, runningCosts, budget):
"""
:type chargeTimes: List[int]
:type runningCosts: List[int]
:type budget: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} chargeTimes
* @param {number[]} runningCosts
* @param {number} budget
* @return {number}
*/
var maximumRobots = function(chargeTimes, runningCosts, budget) {
};
```

TypeScript:

```
function maximumRobots(chargeTimes: number[], runningCosts: number[], budget:
number): number {

};
```

C#:

```
public class Solution {
public int MaximumRobots(int[] chargeTimes, int[] runningCosts, long budget)
{

}
}
```

C:

```
int maximumRobots(int* chargeTimes, int chargeTimesSize, int* runningCosts,
int runningCostsSize, long long budget) {

}
```

Go:

```
func maximumRobots(chargeTimes []int, runningCosts []int, budget int64) int {

}
```

Kotlin:

```
class Solution {
fun maximumRobots(chargeTimes: IntArray, runningCosts: IntArray, budget:
Long): Int {

}
}
```

Swift:

```
class Solution {
func maximumRobots(_ chargeTimes: [Int], _ runningCosts: [Int], _ budget:
Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn maximum_robots(charge_times: Vec<i32>, running_costs: Vec<i32>,
budget: i64) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} charge_times
# @param {Integer[]} running_costs
# @param {Integer} budget
# @return {Integer}
def maximum_robots(charge_times, running_costs, budget)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $chargeTimes
* @param Integer[] $runningCosts
* @param Integer $budget
* @return Integer
*/
function maximumRobots($chargeTimes, $runningCosts, $budget) {

}
}
```

Dart:

```
class Solution {
int maximumRobots(List<int> chargeTimes, List<int> runningCosts, int budget)
{

}
}
```

Scala:

```
object Solution {
def maximumRobots(chargeTimes: Array[Int], runningCosts: Array[Int], budget:
Long): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec maximum_robots(charge_times :: [integer], running_costs :: [integer],
budget :: integer) :: integer
def maximum_robots(charge_times, running_costs, budget) do

end
end
```

Erlang:

```
-spec maximum_robots(ChargeTimes :: [integer()], RunningCosts :: [integer()],
Budget :: integer()) -> integer().
maximum_robots(ChargeTimes, RunningCosts, Budget) ->
.
```

Racket:

```
(define/contract (maximum-robots chargeTimes runningCosts budget)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Number of Robots Within Budget
* Difficulty: Hard
* Tags: array, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
int maximumRobots(vector<int>& chargeTimes, vector<int>& runningCosts, long
long budget) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Number of Robots Within Budget
* Difficulty: Hard
* Tags: array, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Number of Robots Within Budget
```

```
Difficulty: Hard
Tags: array, search, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def maximumRobots(self, chargeTimes: List[int], runningCosts: List[int],
budget: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def maximumRobots(self, chargeTimes, runningCosts, budget):
"""
:type chargeTimes: List[int]
:type runningCosts: List[int]
:type budget: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Maximum Number of Robots Within Budget
* Difficulty: Hard
* Tags: array, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[]} chargeTimes
* @param {number[]} runningCosts
* @param {number} budget
* @return {number}
*/
var maximumRobots = function(chargeTimes, runningCosts, budget) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Number of Robots Within Budget
* Difficulty: Hard
* Tags: array, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function maximumRobots(chargeTimes: number[], runningCosts: number[], budget:
number): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Number of Robots Within Budget
* Difficulty: Hard
* Tags: array, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int MaximumRobots(int[] chargeTimes, int[] runningCosts, long budget)
{

}
}
```

C Solution:

```
/*
* Problem: Maximum Number of Robots Within Budget
* Difficulty: Hard
* Tags: array, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

int maximumRobots(int* chargeTimes, int chargeTimesSize, int* runningCosts,
int runningCostsSize, long long budget) {

}
```

Go Solution:

```
// Problem: Maximum Number of Robots Within Budget
// Difficulty: Hard
// Tags: array, search, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func maximumRobots(chargeTimes []int, runningCosts []int, budget int64) int {

}
```

Kotlin Solution:

```
class Solution {
fun maximumRobots(chargeTimes: IntArray, runningCosts: IntArray, budget:
Long): Int {

}
}
```

Swift Solution:

```
class Solution {
func maximumRobots(_ chargeTimes: [Int], _ runningCosts: [Int], _ budget:
Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Maximum Number of Robots Within Budget
// Difficulty: Hard
// Tags: array, search, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn maximum_robots(charge_times: Vec<i32>, running_costs: Vec<i32>,
budget: i64) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} charge_times
# @param {Integer[]} running_costs
# @param {Integer} budget
# @return {Integer}
def maximum_robots(charge_times, running_costs, budget)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $chargeTimes
* @param Integer[] $runningCosts
* @param Integer $budget
* @return Integer
*/
function maximumRobots($chargeTimes, $runningCosts, $budget) {

}
}
```

Dart Solution:

```
class Solution {
int maximumRobots(List<int> chargeTimes, List<int> runningCosts, int budget)
{

}
}
```

Scala Solution:

```
object Solution {
def maximumRobots(chargeTimes: Array[Int], runningCosts: Array[Int], budget:
Long): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec maximum_robots(charge_times :: [integer], running_costs :: [integer],
budget :: integer) :: integer
def maximum_robots(charge_times, running_costs, budget) do

end
end
```

Erlang Solution:

```
-spec maximum_robots(ChargeTimes :: [integer()], RunningCosts :: [integer()],
Budget :: integer()) -> integer().
maximum_robots(ChargeTimes, RunningCosts, Budget) ->
.
```

Racket Solution:

```
(define/contract (maximum-robots chargeTimes runningCosts budget)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?
exact-integer?)
)
```