Problem 502: IPO

Problem Information

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

Problem Description

Suppose LeetCode will start its

IPO

soon. In order to sell a good price of its shares to Venture Capital, LeetCode would like to work on some projects to increase its capital before the

IPO

. Since it has limited resources, it can only finish at most

k

distinct projects before the

IPO

. Help LeetCode design the best way to maximize its total capital after finishing at most

k

distinct projects.

You are given

n

projects where the

i

th

project has a pure profit

profits[i]

and a minimum capital of

capital[i]

is needed to start it.

Initially, you have

w

capital. When you finish a project, you will obtain its pure profit and the profit will be added to your total capital.

Pick a list of

at most

k

distinct projects from given projects to

maximize your final capital

, and return

the final maximized capital

.

The answer is guaranteed to fit in a 32-bit signed integer.

Example 1:

Input:

k = 2, w = 0, profits = [1,2,3], capital = [0,1,1]

Output:

4

Explanation:

Since your initial capital is 0, you can only start the project indexed 0. After finishing it you will obtain profit 1 and your capital becomes 1. With capital 1, you can either start the project indexed 1 or the project indexed 2. Since you can choose at most 2 projects, you need to finish the project indexed 2 to get the maximum capital. Therefore, output the final maximized capital, which is 0 + 1 + 3 = 4.

Example 2:

Input:

k = 3, w = 0, profits = [1,2,3], capital = [0,1,2]

Output:

6

Constraints:

1 <= k <= 10

5

0 <= w <= 10

9

n == profits.length

n == capital.length

1 <= n <= 10

5

0 <= profits[i] <= 10

4

0 <= capital[i] <= 10

9

Code Snippets

C++:

```
class Solution {
public:
int findMaximizedCapital(int k, int w, vector<int>& profits, vector<int>&
capital) {

}
};
```

Java:

```
class Solution {
public int findMaximizedCapital(int k, int w, int[] profits, int[] capital) {

}
}
```

Python3:

```
class Solution:
def findMaximizedCapital(self, k: int, w: int, profits: List[int], capital:
List[int]) -> int:
```

Python:

```
class Solution(object):
def findMaximizedCapital(self, k, w, profits, capital):
"""
:type k: int
:type w: int
:type profits: List[int]
:type capital: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} k
* @param {number} w
* @param {number[]} profits
* @param {number[]} capital
* @return {number}
*/
var findMaximizedCapital = function(k, w, profits, capital) {

};
```

TypeScript:

```
function findMaximizedCapital(k: number, w: number, profits: number[],
capital: number[]): number {

};
```

C#:

```
public class Solution {
public int FindMaximizedCapital(int k, int w, int[] profits, int[] capital) {

}
}
```

C:

```
int findMaximizedCapital(int k, int w, int* profits, int profitsSize, int*
capital, int capitalSize) {

}
```

Go:

```
func findMaximizedCapital(k int, w int, profits []int, capital []int) int {

}
```

Kotlin:

```
class Solution {
fun findMaximizedCapital(k: Int, w: Int, profits: IntArray, capital:
IntArray): Int {

}
}
```

Swift:

```
class Solution {
func findMaximizedCapital(_ k: Int, _ w: Int, _ profits: [Int], _ capital:
[Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn find_maximized_capital(k: i32, w: i32, profits: Vec<i32>, capital:
Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer} k
# @param {Integer} w
# @param {Integer[]} profits
# @param {Integer[]} capital
# @return {Integer}
def find_maximized_capital(k, w, profits, capital)

end
```

PHP:

```
class Solution {

/**
* @param Integer $k
* @param Integer $w
* @param Integer[] $profits
* @param Integer[] $capital
* @return Integer
*/
function findMaximizedCapital($k, $w, $profits, $capital) {

}
}
```

Dart:

```
class Solution {
int findMaximizedCapital(int k, int w, List<int> profits, List<int> capital)
{

}
}
```

Scala:

```
object Solution {
def findMaximizedCapital(k: Int, w: Int, profits: Array[Int], capital:
Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec find_maximized_capital(k :: integer, w :: integer, profits ::
[integer], capital :: [integer]) :: integer
def find_maximized_capital(k, w, profits, capital) do

end
end
```

Erlang:

```
-spec find_maximized_capital(K :: integer(), W :: integer(), Profits ::
[integer()], Capital :: [integer()]) -> integer().
find_maximized_capital(K, W, Profits, Capital) ->
.
```

Racket:

```
(define/contract (find-maximized-capital k w profits capital)
(-> exact-integer? exact-integer? (listof exact-integer?) (listof
exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: IPO
* Difficulty: Hard
* Tags: array, greedy, sort, 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 findMaximizedCapital(int k, int w, vector<int>& profits, vector<int>&
capital) {
}
};
```

Java Solution:

```
/**
* Problem: IPO
* Difficulty: Hard
* Tags: array, greedy, sort, 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 findMaximizedCapital(int k, int w, int[] profits, int[] capital) {

}
}
```

Python3 Solution:

```
"""
Problem: IPO
Difficulty: Hard
Tags: array, greedy, sort, 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 findMaximizedCapital(self, k: int, w: int, profits: List[int], capital:
List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def findMaximizedCapital(self, k, w, profits, capital):
"""
:type k: int
:type w: int
:type profits: List[int]
:type capital: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: IPO
* Difficulty: Hard
* Tags: array, greedy, sort, 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} k
* @param {number} w
* @param {number[]} profits
* @param {number[]} capital
* @return {number}
*/
var findMaximizedCapital = function(k, w, profits, capital) {

};
```

TypeScript Solution:

```
/**
* Problem: IPO
* Difficulty: Hard
* Tags: array, greedy, sort, 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 findMaximizedCapital(k: number, w: number, profits: number[], capital: number[]): number {

};
```

C# Solution:

```
/*
* Problem: IPO
* Difficulty: Hard
* Tags: array, greedy, sort, 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 FindMaximizedCapital(int k, int w, int[] profits, int[] capital) {

}
}
```

C Solution:

```
/*
* Problem: IPO
* Difficulty: Hard
* Tags: array, greedy, sort, 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 findMaximizedCapital(int k, int w, int* profits, int profitsSize, int* capital, int capitalSize) {


}
```

Go Solution:

```
// Problem: IPO
// Difficulty: Hard
// Tags: array, greedy, sort, 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 findMaximizedCapital(k int, w int, profits []int, capital []int) int {


}
```

Kotlin Solution:

```
class Solution {
fun findMaximizedCapital(k: Int, w: Int, profits: IntArray, capital:
IntArray): Int {


}
}
```

Swift Solution:

```
class Solution {
func findMaximizedCapital(_ k: Int, _ w: Int, _ profits: [Int], _ capital:
[Int]) -> Int {


}
}
```

Rust Solution:

```
// Problem: IPO
// Difficulty: Hard
// Tags: array, greedy, sort, 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 find_maximized_capital(k: i32, w: i32, profits: Vec<i32>, capital:
Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer} k
# @param {Integer} w
# @param {Integer[]} profits
# @param {Integer[]} capital
# @return {Integer}
def find_maximized_capital(k, w, profits, capital)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $k
* @param Integer $w
* @param Integer[] $profits
* @param Integer[] $capital
* @return Integer
*/
function findMaximizedCapital($k, $w, $profits, $capital) {

}
}
```

Dart Solution:

```
class Solution {
int findMaximizedCapital(int k, int w, List<int> profits, List<int> capital)
{

}
}
```

Scala Solution:

```
object Solution {
def findMaximizedCapital(k: Int, w: Int, profits: Array[Int], capital:
Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec find_maximized_capital(k :: integer, w :: integer, profits ::
[integer], capital :: [integer]) :: integer
def find_maximized_capital(k, w, profits, capital) do

end
end
```

Erlang Solution:

```
-spec find_maximized_capital(K :: integer(), W :: integer(), Profits ::
[integer()], Capital :: [integer()]) -> integer().
find_maximized_capital(K, W, Profits, Capital) ->
.
```

Racket Solution:

```
(define/contract (find-maximized-capital k w profits capital)
(-> exact-integer? exact-integer? (listof exact-integer?) (listof
exact-integer?) exact-integer?)
)
```