Problem 2234: Maximum Total Beauty of the Gardens

Problem Information

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

Problem Description

Alice is a caretaker of

n

gardens and she wants to plant flowers to maximize the total beauty of all her gardens.

You are given a

0-indexed

integer array

flowers

of size

n

, where

flowers[i]

is the number of flowers already planted in the

i

th

garden. Flowers that are already planted

cannot

be removed. You are then given another integer

newFlowers

, which is the

maximum

number of flowers that Alice can additionally plant. You are also given the integers

target

,

full

, and

partial

.

A garden is considered

complete

if it has

at least

target

flowers. The

total beauty

of the gardens is then determined as the

sum

of the following:

The number of

complete

gardens multiplied by

full

.

The

minimum

number of flowers in any of the

incomplete

gardens multiplied by

partial

. If there are no incomplete gardens, then this value will be

0

.

Return

the

maximum

total beauty that Alice can obtain after planting at most

newFlowers

flowers.

Example 1:

Input:

flowers = [1,3,1,1], newFlowers = 7, target = 6, full = 12, partial = 1

Output:

14

Explanation:

Alice can plant - 2 flowers in the 0

th

garden - 3 flowers in the 1

st

garden - 1 flower in the 2

nd

garden - 1 flower in the 3

rd

garden The gardens will then be [3,6,2,2]. She planted a total of 2 + 3 + 1 + 1 = 7 flowers. There is 1 garden that is complete. The minimum number of flowers in the incomplete gardens is 2. Thus, the total beauty is 1 * 12 + 2 * 1 = 12 + 2 = 14. No other way of planting flowers can obtain a total beauty higher than 14.

Example 2:

Input:

flowers = [2,4,5,3], newFlowers = 10, target = 5, full = 2, partial = 6

Output:

30

Explanation:

Alice can plant - 3 flowers in the 0

th

garden - 0 flowers in the 1

st

garden - 0 flowers in the 2

nd

garden - 2 flowers in the 3

rd

garden The gardens will then be [5,4,5,5]. She planted a total of 3 + 0 + 0 + 2 = 5 flowers. There are 3 gardens that are complete. The minimum number of flowers in the incomplete gardens is 4. Thus, the total beauty is 3 * 2 + 4 * 6 = 6 + 24 = 30. No other way of planting flowers can obtain a total beauty higher than 30. Note that Alice could make all the gardens complete but in this case, she would obtain a lower total beauty.

Constraints:

1 <= flowers.length <= 10

5

1 <= flowers[i], target <= 10

5

1 <= newFlowers <= 10

10

1 <= full, partial <= 10

5

Code Snippets

C++:

```
class Solution {
public:
long long maximumBeauty(vector<int>& flowers, long long newFlowers, int
target, int full, int partial) {

}
};
```

Java:

```
class Solution {
public long maximumBeauty(int[] flowers, long newFlowers, int target, int
full, int partial) {

}
}
```

Python3:

```
class Solution:
def maximumBeauty(self, flowers: List[int], newFlowers: int, target: int,
full: int, partial: int) -> int:
```

Python:

```
class Solution(object):
def maximumBeauty(self, flowers, newFlowers, target, full, partial):
"""
:type flowers: List[int]
:type newFlowers: int
:type target: int
:type full: int
:type partial: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} flowers
* @param {number} newFlowers
* @param {number} target
* @param {number} full
* @param {number} partial
* @return {number}
*/
var maximumBeauty = function(flowers, newFlowers, target, full, partial) {

};
```

TypeScript:

```
function maximumBeauty(flowers: number[], newFlowers: number, target: number,
full: number, partial: number): number {

};
```

C#:

```
public class Solution {
public long MaximumBeauty(int[] flowers, long newFlowers, int target, int
full, int partial) {

}
}
```

C:

```
long long maximumBeauty(int* flowers, int flowersSize, long long newFlowers,
int target, int full, int partial) {

}
```

Go:

```
func maximumBeauty(flowers []int, newFlowers int64, target int, full int,
partial int) int64 {

}
```

Kotlin:

```
class Solution {
fun maximumBeauty(flowers: IntArray, newFlowers: Long, target: Int, full:
Int, partial: Int): Long {

}
}
```

Swift:

```
class Solution {
func maximumBeauty(_ flowers: [Int], _ newFlowers: Int, _ target: Int, _
full: Int, _ partial: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn maximum_beauty(flowers: Vec<i32>, new_flowers: i64, target: i32, full:
```

```
i32, partial: i32) -> i64 {

}
}
```

Ruby:

```
# @param {Integer[]} flowers
# @param {Integer} new_flowers
# @param {Integer} target
# @param {Integer} full
# @param {Integer} partial
# @return {Integer}
def maximum_beauty(flowers, new_flowers, target, full, partial)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $flowers
* @param Integer $newFlowers
* @param Integer $target
* @param Integer $full
* @param Integer $partial
* @return Integer
*/
function maximumBeauty($flowers, $newFlowers, $target, $full, $partial) {

}
}
```

Dart:

```
class Solution {
int maximumBeauty(List<int> flowers, int newFlowers, int target, int full,
int partial) {

}
}
```

Scala:

```
object Solution {
def maximumBeauty(flowers: Array[Int], newFlowers: Long, target: Int, full:
Int, partial: Int): Long = {

}
}
```

Elixir:

```
defmodule Solution do
@spec maximum_beauty(flowers :: [integer], new_flowers :: integer, target ::
integer, full :: integer, partial :: integer) :: integer
def maximum_beauty(flowers, new_flowers, target, full, partial) do

end
end
```

Erlang:

```
-spec maximum_beauty(Flowers :: [integer()], NewFlowers :: integer(), Target
:: integer(), Full :: integer(), Partial :: integer()) -> integer().
maximum_beauty(Flowers, NewFlowers, Target, Full, Partial) ->
.
```

Racket:

```
(define/contract (maximum-beauty flowers newFlowers target full partial)
(-> (listof exact-integer?) exact-integer? exact-integer? exact-integer?
exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Total Beauty of the Gardens
* Difficulty: Hard
* Tags: array, greedy, sort, search
*
```

```
* 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:
long long maximumBeauty(vector<int>& flowers, long long newFlowers, int
target, int full, int partial) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Total Beauty of the Gardens
* Difficulty: Hard
* Tags: array, greedy, sort, search
*
* 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 long maximumBeauty(int[] flowers, long newFlowers, int target, int
full, int partial) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Total Beauty of the Gardens
Difficulty: Hard
Tags: array, greedy, sort, search

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 maximumBeauty(self, flowers: List[int], newFlowers: int, target: int,
full: int, partial: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def maximumBeauty(self, flowers, newFlowers, target, full, partial):
"""
:type flowers: List[int]
:type newFlowers: int
:type target: int
:type full: int
:type partial: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Maximum Total Beauty of the Gardens
* Difficulty: Hard
* Tags: array, greedy, sort, search
*
* 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[]} flowers
* @param {number} newFlowers
* @param {number} target
* @param {number} full
* @param {number} partial
* @return {number}
```

```
*/
var maximumBeauty = function(flowers, newFlowers, target, full, partial) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Total Beauty of the Gardens
* Difficulty: Hard
* Tags: array, greedy, sort, search
*
* 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 maximumBeauty(flowers: number[], newFlowers: number, target: number,
full: number, partial: number): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Total Beauty of the Gardens
* Difficulty: Hard
* Tags: array, greedy, sort, search
*
* 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 long MaximumBeauty(int[] flowers, long newFlowers, int target, int
full, int partial) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Total Beauty of the Gardens
* Difficulty: Hard
* Tags: array, greedy, sort, search
*
* 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
*/

long long maximumBeauty(int* flowers, int flowersSize, long long newFlowers,
int target, int full, int partial) {

}
```

Go Solution:

```
// Problem: Maximum Total Beauty of the Gardens
// Difficulty: Hard
// Tags: array, greedy, sort, search
//
// 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 maximumBeauty(flowers []int, newFlowers int64, target int, full int,
partial int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun maximumBeauty(flowers: IntArray, newFlowers: Long, target: Int, full:
Int, partial: Int): Long {

}
}
```

Swift Solution:

```
class Solution {
func maximumBeauty(_ flowers: [Int], _ newFlowers: Int, _ target: Int, _
full: Int, _ partial: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Maximum Total Beauty of the Gardens
// Difficulty: Hard
// Tags: array, greedy, sort, search
//
// 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_beauty(flowers: Vec<i32>, new_flowers: i64, target: i32, full:
i32, partial: i32) -> i64 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} flowers
# @param {Integer} new_flowers
# @param {Integer} target
# @param {Integer} full
# @param {Integer} partial
# @return {Integer}
def maximum_beauty(flowers, new_flowers, target, full, partial)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $flowers
```

```
* @param Integer $newFlowers
* @param Integer $target
* @param Integer $full
* @param Integer $partial
* @return Integer
*/
function maximumBeauty($flowers, $newFlowers, $target, $full, $partial) {

}
}
```

Dart Solution:

```
class Solution {
int maximumBeauty(List<int> flowers, int newFlowers, int target, int full,
int partial) {

}
}
```

Scala Solution:

```
object Solution {
def maximumBeauty(flowers: Array[Int], newFlowers: Long, target: Int, full:
Int, partial: Int): Long = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec maximum_beauty(flowers :: [integer], new_flowers :: integer, target ::
integer, full :: integer, partial :: integer) :: integer
def maximum_beauty(flowers, new_flowers, target, full, partial) do

end
end
```

Erlang Solution:

```
-spec maximum_beauty(Flowers :: [integer()], NewFlowers :: integer(), Target
:: integer(), Full :: integer(), Partial :: integer()) -> integer().
maximum_beauty(Flowers, NewFlowers, Target, Full, Partial) ->
.
```

Racket Solution:

```
(define/contract (maximum-beauty flowers newFlowers target full partial)
(-> (listof exact-integer?) exact-integer? exact-integer? exact-integer?
exact-integer? exact-integer?)
)
```