Problem 1774: Closest Dessert Cost

Problem Information

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

Problem Description

You would like to make dessert and are preparing to buy the ingredients. You have

n

ice cream base flavors and

m

types of toppings to choose from. You must follow these rules when making your dessert:

There must be

exactly one

ice cream base.

You can add

one or more

types of topping or have no toppings at all.

There are

at most two

of

each type

of topping.

You are given three inputs:

baseCosts

, an integer array of length

n

, where each

baseCosts[i]

represents the price of the

i

th

ice cream base flavor.

toppingCosts

, an integer array of length

m

, where each

toppingCosts[i]

is the price of

one

of the

i

th

topping.

target

, an integer representing your target price for dessert.

You want to make a dessert with a total cost as close to

target

as possible.

Return

the closest possible cost of the dessert to

target

. If there are multiple, return

the

lower

one.

Example 1:

Input:

baseCosts = [1,7], toppingCosts = [3,4], target = 10

Output:

10

Explanation:

Consider the following combination (all 0-indexed): - Choose base 1: cost 7 - Take 1 of topping 0: cost 1 x 3 = 3 - Take 0 of topping 1: cost 0 x 4 = 0 Total: 7 + 3 + 0 = 10.

Example 2:

Input:

baseCosts = [2,3], toppingCosts = [4,5,100], target = 18

Output:

17

Explanation:

Consider the following combination (all 0-indexed): - Choose base 1: cost 3 - Take 1 of topping 0: cost 1 x 4 = 4 - Take 2 of topping 1: cost 2 x 5 = 10 - Take 0 of topping 2: cost 0 x 100 = 0 Total: 3 + 4 + 10 + 0 = 17. You cannot make a dessert with a total cost of 18.

Example 3:

Input:

baseCosts = [3,10], toppingCosts = [2,5], target = 9

Output:

8

Explanation:

It is possible to make desserts with cost 8 and 10. Return 8 as it is the lower cost.

Constraints:

n == baseCosts.length

m == toppingCosts.length

1 <= n, m <= 10

1 <= baseCosts[i], toppingCosts[i] <= 10

4

1 <= target <= 10

4

Code Snippets

C++:

```
class Solution {
public:
int closestCost(vector<int>& baseCosts, vector<int>& toppingCosts, int
target) {

}
};
```

Java:

```
class Solution {
public int closestCost(int[] baseCosts, int[] toppingCosts, int target) {

}
}
```

Python3:

```
class Solution:
def closestCost(self, baseCosts: List[int], toppingCosts: List[int], target:
```

```
int) -> int:
```

Python:

```
class Solution(object):
def closestCost(self, baseCosts, toppingCosts, target):
"""
:type baseCosts: List[int]
:type toppingCosts: List[int]
:type target: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} baseCosts
* @param {number[]} toppingCosts
* @param {number} target
* @return {number}
*/
var closestCost = function(baseCosts, toppingCosts, target) {

};
```

TypeScript:

```
function closestCost(baseCosts: number[], toppingCosts: number[], target:
number): number {

};
```

C#:

```
public class Solution {
public int ClosestCost(int[] baseCosts, int[] toppingCosts, int target) {

}
}
```

C:

```
int closestCost(int* baseCosts, int baseCostsSize, int* toppingCosts, int
toppingCostsSize, int target) {

}
```

Go:

```
func closestCost(baseCosts []int, toppingCosts []int, target int) int {

}
```

Kotlin:

```
class Solution {
fun closestCost(baseCosts: IntArray, toppingCosts: IntArray, target: Int):
Int {

}
}
```

Swift:

```
class Solution {
func closestCost(_ baseCosts: [Int], _ toppingCosts: [Int], _ target: Int) ->
Int {

}
}
```

Rust:

```
impl Solution {
pub fn closest_cost(base_costs: Vec<i32>, topping_costs: Vec<i32>, target:
i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} base_costs
# @param {Integer[]} topping_costs
# @param {Integer} target
```

```
# @return {Integer}
def closest_cost(base_costs, topping_costs, target)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $baseCosts
* @param Integer[] $toppingCosts
* @param Integer $target
* @return Integer
*/
function closestCost($baseCosts, $toppingCosts, $target) {

}
}
```

Dart:

```
class Solution {
int closestCost(List<int> baseCosts, List<int> toppingCosts, int target) {

}
}
```

Scala:

```
object Solution {
def closestCost(baseCosts: Array[Int], toppingCosts: Array[Int], target:
Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec closest_cost(base_costs :: [integer], topping_costs :: [integer],
target :: integer) :: integer
```

```
def closest_cost(base_costs, topping_costs, target) do

end
end
```

Erlang:

```
-spec closest_cost(BaseCosts :: [integer()], ToppingCosts :: [integer()],
Target :: integer()) -> integer().
closest_cost(BaseCosts, ToppingCosts, Target) ->
.
```

Racket:

```
(define/contract (closest-cost baseCosts toppingCosts target)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Closest Dessert Cost
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int closestCost(vector<int>& baseCosts, vector<int>& toppingCosts, int
target) {

}
};
```

Java Solution:

```
/**
* Problem: Closest Dessert Cost
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int closestCost(int[] baseCosts, int[] toppingCosts, int target) {

}
}
```

Python3 Solution:

```
"""
Problem: Closest Dessert Cost
Difficulty: Medium
Tags: array, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def closestCost(self, baseCosts: List[int], toppingCosts: List[int], target:
int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def closestCost(self, baseCosts, toppingCosts, target):
"""
:type baseCosts: List[int]
```

```
:type toppingCosts: List[int]
:type target: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Closest Dessert Cost
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number[]} baseCosts
* @param {number[]} toppingCosts
* @param {number} target
* @return {number}
*/
var closestCost = function(baseCosts, toppingCosts, target) {

};
```

TypeScript Solution:

```
/**
* Problem: Closest Dessert Cost
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function closestCost(baseCosts: number[], toppingCosts: number[], target:
number): number {
```

```
};
```

C# Solution:

```
/*
* Problem: Closest Dessert Cost
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int ClosestCost(int[] baseCosts, int[] toppingCosts, int target) {

}
}
```

C Solution:

```
/*
* Problem: Closest Dessert Cost
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int closestCost(int* baseCosts, int baseCostsSize, int* toppingCosts, int
toppingCostsSize, int target) {

}
```

Go Solution:

```
// Problem: Closest Dessert Cost
// Difficulty: Medium
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func closestCost(baseCosts []int, toppingCosts []int, target int) int {

}
```

Kotlin Solution:

```
class Solution {
fun closestCost(baseCosts: IntArray, toppingCosts: IntArray, target: Int):
Int {

}
}
```

Swift Solution:

```
class Solution {
func closestCost(_ baseCosts: [Int], _ toppingCosts: [Int], _ target: Int) ->
Int {

}
}
```

Rust Solution:

```
// Problem: Closest Dessert Cost
// Difficulty: Medium
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn closest_cost(base_costs: Vec<i32>, topping_costs: Vec<i32>, target:
```

```
i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} base_costs
# @param {Integer[]} topping_costs
# @param {Integer} target
# @return {Integer}
def closest_cost(base_costs, topping_costs, target)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $baseCosts
* @param Integer[] $toppingCosts
* @param Integer $target
* @return Integer
*/
function closestCost($baseCosts, $toppingCosts, $target) {

}
}
```

Dart Solution:

```
class Solution {
int closestCost(List<int> baseCosts, List<int> toppingCosts, int target) {

}
}
```

Scala Solution:

```
object Solution {
def closestCost(baseCosts: Array[Int], toppingCosts: Array[Int], target:
```

```
Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec closest_cost(base_costs :: [integer], topping_costs :: [integer],
target :: integer) :: integer
def closest_cost(base_costs, topping_costs, target) do

end
end
```

Erlang Solution:

```
-spec closest_cost(BaseCosts :: [integer()], ToppingCosts :: [integer()],
Target :: integer()) -> integer().
closest_cost(BaseCosts, ToppingCosts, Target) ->
.
```

Racket Solution:

```
(define/contract (closest-cost baseCosts toppingCosts target)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?
exact-integer?)
)
```