

Problem 2907: Maximum Profitable Triplets With Increasing Prices I

Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

Given the

0-indexed

arrays

prices

and

profits

of length

n

. There are

n

items in a store where the

i

th

item has a price of

$\text{prices}[i]$

and a profit of

$\text{profits}[i]$

We have to pick three items with the following condition:

$\text{prices}[i] < \text{prices}[j] < \text{prices}[k]$

where

$i < j < k$

If we pick items with indices

i

,

j

and

k

satisfying the above condition, the profit would be

$\text{profits}[i] + \text{profits}[j] + \text{profits}[k]$

Return

the

maximum profit

we can get, and

-1

if it's not possible to pick three items with the given condition.

Example 1:

Input:

prices = [10,2,3,4], profits = [100,2,7,10]

Output:

19

Explanation:

We can't pick the item with index $i=0$ since there are no indices j and k such that the condition holds. So the only triplet we can pick, are the items with indices 1, 2 and 3 and it's a valid pick since $\text{prices}[1] < \text{prices}[2] < \text{prices}[3]$. The answer would be sum of their profits which is $2 + 7 + 10 = 19$.

Example 2:

Input:

prices = [1,2,3,4,5], profits = [1,5,3,4,6]

Output:

15

Explanation:

We can select any triplet of items since for each triplet of indices i, j and k such that $i < j < k$, the condition holds. Therefore the maximum profit we can get would be the 3 most profitable items which are indices 1, 3 and 4. The answer would be sum of their profits which is $5 + 4 + 6 = 15$.

Example 3:

Input:

prices = [4,3,2,1], profits = [33,20,19,87]

Output:

-1

Explanation:

We can't select any triplet of indices such that the condition holds, so we return -1.

Constraints:

$3 \leq \text{prices.length} == \text{profits.length} \leq 2000$

$1 \leq \text{prices}[i] \leq 10$

6

$1 \leq \text{profits}[i] \leq 10$

6

Code Snippets

C++:

```
class Solution {  
public:  
    int maxProfit(vector<int>& prices, vector<int>& profits) {  
  
    }  
};
```

Java:

```
class Solution {  
public int maxProfit(int[] prices, int[] profits) {  
  
}  
}
```

Python3:

```
class Solution:  
    def maxProfit(self, prices: List[int], profits: List[int]) -> int:
```

Python:

```
class Solution(object):  
    def maxProfit(self, prices, profits):  
        """  
        :type prices: List[int]  
        :type profits: List[int]  
        :rtype: int  
        """
```

JavaScript:

```
/**  
 * @param {number[]} prices  
 * @param {number[]} profits  
 * @return {number}  
 */  
var maxProfit = function(prices, profits) {  
  
};
```

TypeScript:

```
function maxProfit(prices: number[], profits: number[]): number {  
}  
};
```

C#:

```
public class Solution {  
    public int MaxProfit(int[] prices, int[] profits) {  
        }  
    }  
}
```

C:

```
int maxProfit(int* prices, int pricesSize, int* profits, int profitsSize) {  
}  
}
```

Go:

```
func maxProfit(prices []int, profits []int) int {  
}  
}
```

Kotlin:

```
class Solution {  
    fun maxProfit(prices: IntArray, profits: IntArray): Int {  
        }  
    }  
}
```

Swift:

```
class Solution {  
    func maxProfit(_ prices: [Int], _ profits: [Int]) -> Int {  
        }  
    }  
}
```

Rust:

```
impl Solution {  
    pub fn max_profit(prices: Vec<i32>, profits: Vec<i32>) -> i32 {  
        }  
    }  
}
```

Ruby:

```
# @param {Integer[]} prices  
# @param {Integer[]} profits  
# @return {Integer}  
def max_profit(prices, profits)  
  
end
```

PHP:

```
class Solution {  
  
    /**  
     * @param Integer[] $prices  
     * @param Integer[] $profits  
     * @return Integer  
     */  
    function maxProfit($prices, $profits) {  
  
    }  
}
```

Dart:

```
class Solution {  
    int maxProfit(List<int> prices, List<int> profits) {  
        }  
    }
```

Scala:

```
object Solution {  
    def maxProfit(prices: Array[Int], profits: Array[Int]): Int = {  
        }  
}
```

```
}
```

Elixir:

```
defmodule Solution do
  @spec max_profit(prices :: [integer], profits :: [integer]) :: integer
  def max_profit(prices, profits) do
    end
  end
```

Erlang:

```
-spec max_profit(Prices :: [integer()], Profits :: [integer()]) -> integer().
max_profit(Prices, Profits) ->
  .
```

Racket:

```
(define/contract (max-profit prices profits)
  (-> (listof exact-integer?) (listof exact-integer?) exact-integer?))
```

Solutions

C++ Solution:

```
/*
 * Problem: Maximum Profitable Triplets With Increasing Prices I
 * Difficulty: Medium
 * Tags: array, tree
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

class Solution {
public:
  int maxProfit(vector<int>& prices, vector<int>& profits) {
```

```
}
```

```
} ;
```

Java Solution:

```
/**  
 * Problem: Maximum Profitable Triplets With Increasing Prices I  
 * Difficulty: Medium  
 * Tags: array, tree  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(h) for recursion stack where h is height  
 */  
  
class Solution {  
    public int maxProfit(int[] prices, int[] profits) {  
        return 0;  
    }  
}
```

Python3 Solution:

```
"""  
Problem: Maximum Profitable Triplets With Increasing Prices I  
Difficulty: Medium  
Tags: array, tree  
  
Approach: Use two pointers or sliding window technique  
Time Complexity: O(n) or O(n log n)  
Space Complexity: O(h) for recursion stack where h is height  
"""  
  
class Solution:  
    def maxProfit(self, prices: List[int], profits: List[int]) -> int:  
        # TODO: Implement optimized solution  
        pass
```

Python Solution:

```

class Solution(object):
    def maxProfit(self, prices, profits):
        """
        :type prices: List[int]
        :type profits: List[int]
        :rtype: int
        """

```

JavaScript Solution:

```

/**
 * Problem: Maximum Profitable Triplets With Increasing Prices I
 * Difficulty: Medium
 * Tags: array, tree
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**
 * @param {number[]} prices
 * @param {number[]} profits
 * @return {number}
 */
var maxProfit = function(prices, profits) {
}
```

TypeScript Solution:

```

/**
 * Problem: Maximum Profitable Triplets With Increasing Prices I
 * Difficulty: Medium
 * Tags: array, tree
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

function maxProfit(prices: number[], profits: number[]): number {
```

```
};
```

C# Solution:

```
/*
 * Problem: Maximum Profitable Triplets With Increasing Prices I
 * Difficulty: Medium
 * Tags: array, tree
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

public class Solution {
    public int MaxProfit(int[] prices, int[] profits) {
        return 0;
    }
}
```

C Solution:

```
/*
 * Problem: Maximum Profitable Triplets With Increasing Prices I
 * Difficulty: Medium
 * Tags: array, tree
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

int maxProfit(int* prices, int pricesSize, int* profits, int profitsSize) {
    return 0;
}
```

Go Solution:

```
// Problem: Maximum Profitable Triplets With Increasing Prices I
// Difficulty: Medium
```

```

// Tags: array, tree
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

func maxProfit(prices []int, profits []int) int {

}

```

Kotlin Solution:

```

class Solution {
    fun maxProfit(prices: IntArray, profits: IntArray): Int {
        return 0
    }
}

```

Swift Solution:

```

class Solution {
    func maxProfit(_ prices: [Int], _ profits: [Int]) -> Int {
        return 0
    }
}

```

Rust Solution:

```

// Problem: Maximum Profitable Triplets With Increasing Prices I
// Difficulty: Medium
// Tags: array, tree
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
    pub fn max_profit(prices: Vec<i32>, profits: Vec<i32>) -> i32 {
        return 0
    }
}

```

Ruby Solution:

```
# @param {Integer[]} prices
# @param {Integer[]} profits
# @return {Integer}
def max_profit(prices, profits)

end
```

PHP Solution:

```
class Solution {

    /**
     * @param Integer[] $prices
     * @param Integer[] $profits
     * @return Integer
     */
    function maxProfit($prices, $profits) {

    }
}
```

Dart Solution:

```
class Solution {
  int maxProfit(List<int> prices, List<int> profits) {
    }
}
```

Scala Solution:

```
object Solution {
  def maxProfit(prices: Array[Int], profits: Array[Int]): Int = {
    }
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_profit(prices :: [integer], profits :: [integer]) :: integer
def max_profit(prices, profits) do

end
end
```

Erlang Solution:

```
-spec max_profit(Prices :: [integer()], Profits :: [integer()]) -> integer().
max_profit(Prices, Profits) ->
.
```

Racket Solution:

```
(define/contract (max-profit prices profits)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?))
```