Maximum Profits

We'll discuss how to write a function that can reap the maximum profits from a day's stock data. We'll discuss both the simple brute-force solution and an elegant solution to this problem.

Maximum Profits

Instructions

Suppose we could access yesterday's prices for a certain stock as a chronological list. Write a function that takes the list and returns the highest profit possible from one purchase and one sale of the stock yesterday.

For example, a stock price list of `[10, 7, 5, 8, 11, 9]` shows that the stock started at `10` and ended at `9`, going through the numbers chronologically. There is at least a 1-minute difference between the stock prices.

Taking that array as input, our function should return `6`, the maximum possible profit from buying when the price was `5` and selling when the price was `11`.

If no profit can be made, return `0`.

No "shorting" — you must buy before you sell. You may not buy and sell in the same time step.

Input: Array of Numbers

Output: Number

Hints

- We'll have to convert this array of stock prices into possible profits.
- Think about the fact that this is a chronologically ordered list.

```
1  function getMaxProfit(prices) {
2      // Your code here
3  }
```

Solution 1

```
1  function getMaxProfit(prices) {
2      const possibleProfits = [0];
3
4      for(let i = 0; i < prices.length; i++) {
5          for(let j = i + 1; j < prices.length; j++) {
6              possibleProfits.push(prices[j] - prices[i]);
7          }
8      }
9
10     return Math.max(...possibleProfits);
11 }
```

Show Results Show Console

2.14s

4 of 4 Public Tests Passed

Result	Input	Expected Output	Actual Output	Reason

Result	Input	Expected Output	Actual Output	Reason
✓	[10, 7, 5, 8, 11, 9]	6	6	Succeeded
✓	[1, 2, 3, 4, 5]	4	4	Succeeded
✓	[5, 4, 3, 2, 1]	0	0	Succeeded
✓	[5, 20, 4, 10, 1]	15	15	Succeeded

How it Works

In the double for-loop, we're calculating every buy-and-sell profit possible and throwing each value into an array.

The return statement goes through each of these and returns the largest one.

Time

Double For-Loop

We have a standard double for-loop, suggesting `O(n^2)` efficiency. Indeed, we'll see that this is the overall time complexity of the function.

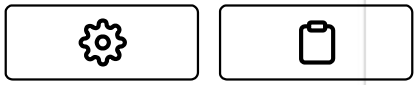
To be sure, we can count the number of times the inner for-loop runs with the following code.

```
let count = 0;

function getMaxProfit(prices) {
    const possibleProfits = [];

    for(let i = 0; i < prices.length; i++) {
        for(let j = i; j < prices.length; j++) {
            possibleProfits.push(prices[j] - prices[i]);
            count++;
```

```
10          }
11      }
12
13      return Math.max(...possibleProfits);
14  }
15
16  getMaxProfit([1, 2, 3]);
17  console.log(count);
```



In the above code, try changing the size of the array on line 16. Using this code, we get the following table.

Array Length	Count
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

Array Length	Count
11	66
12	78
...	...

The table above follows this formula, where `c` is count and `n` is array length:

```
c = (n^2 + n) / 2
```

So, the time complexity of our double for-loop is `O((n^2 + n) / 2)`.

Math.max

The time complexity of the call to `Math.max` is exactly the same - the double for-loop pushes a certain number of items into the array and then `Math.max` must process them all again. So that statement's time complexity is also `(n^2 + n) / 2`.

Entire Function

Since the Math.max call happens after the double for-loop, we can add the two time complexities together. We get a final value of `O(n^2 + n)`.

Dropping all but the highest-order term, we get a time complexity of:

O(n^2).

Space

Our array must store every item the double for-loop pushes into it, so we have the same space and time complexity:

O(n^2).

We can do better. It's tricky, but I recommend giving it a try. Here are some hints for Solution 2.

Hints

- We must limit ourselves to a single loop, a single pass through the array.
- We'll have to keep track of a couple of variables and update them in our loop.
- Think about the relationship between a number and the numbers that come after it.
- We'll have to keep track of the maximum profit throughout the function.
- We'll have to keep track of the minimum stock value throughout our loop.

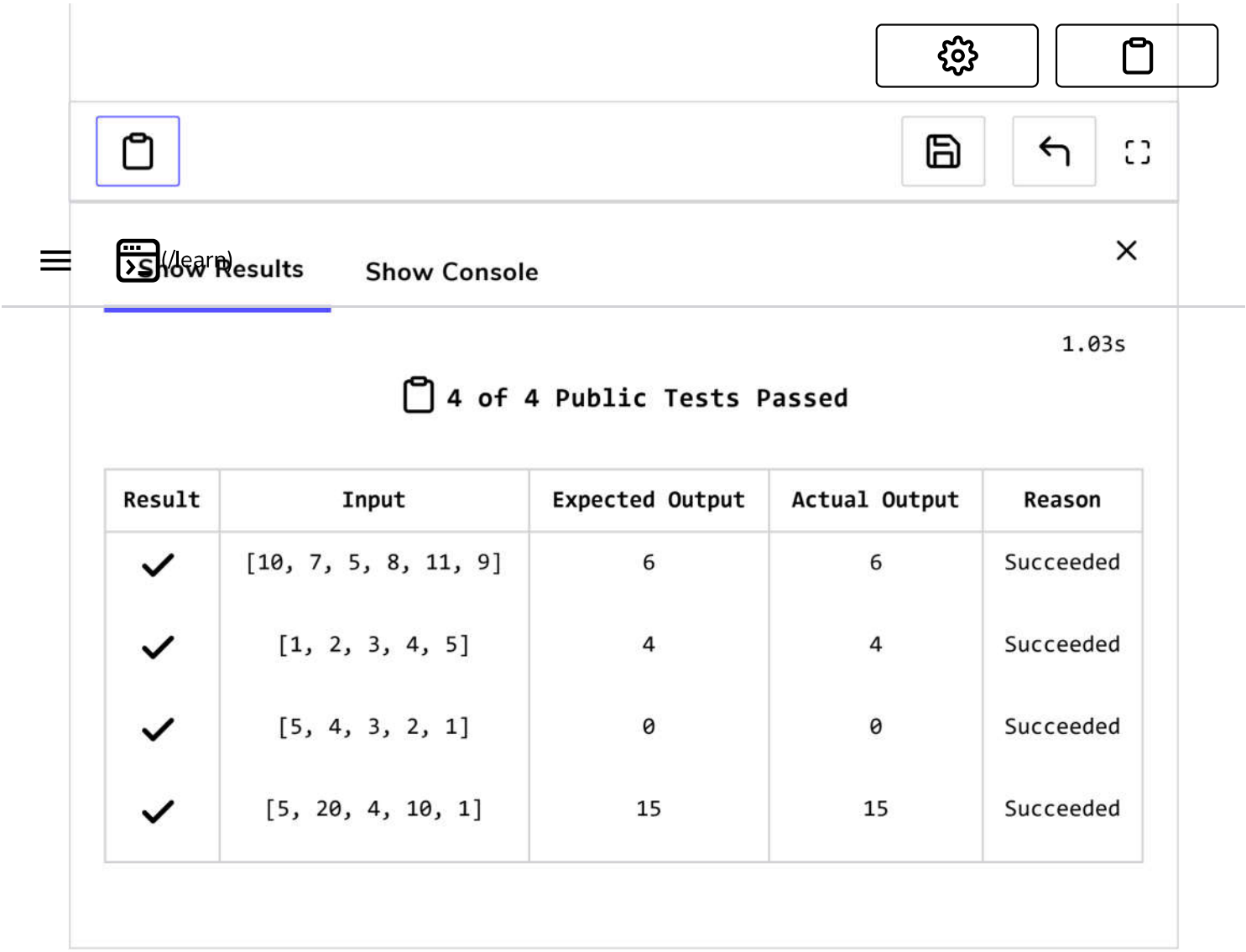
Good luck!

Solution 2

```
function getMaxProfit(prices) {
    let minPrice = Infinity;
    let maxProfit = 0;

    for(let i = 0; i < prices.length; i++) {
        const currentPrice = prices[i];
        minPrice = Math.min(minPrice, currentPrice);
        maxProfit = Math.max(maxProfit, currentPrice - minPrice);
    }

    return maxProfit;
}
```



Result	Input	Expected Output	Actual Output	Reason
✓	[10, 7, 5, 8, 11, 9]	6	6	Succeeded
✓	[1, 2, 3, 4, 5]	4	4	Succeeded
✓	[5, 4, 3, 2, 1]	0	0	Succeeded
✓	[5, 20, 4, 10, 1]	15	15	Succeeded

How it Works

We start by assuming the maximum profit we can make is `0`, the minimum value we can accept. We also start by assuming the minimum stock price is the highest value we can use, `Infinity`.

Throughout the function, we'll keep track of the minimum stock price and the maximum possible profit we can make, changing them whenever we need to as we go through the loop.

The Loop

The second line in the loop updates our minimum price if necessary.

The third line updates the maximum profit if the profit from buying at our current minimum price and selling at the current number is greater than the current max profit.

At the end, we return that maximum profit.

Time

This time, we go through the array only once, giving us an efficiency of:

O(n)

Much better than `O(n^2)`.

Space

We don't create any arrays or store values across loop iterations. All we track is the two variables declared at the beginning. We maintain the same amount of information regardless of array size, so we have a space complexity of:

O(1)

It's the best we can get.

Conclusion

This is the solution that would win the interviewer over. It's difficult and it shows the ability to logically deal with both numbers and time. It's elegant, short and sweet. It has the best time and space complexities imaginable for this type of problem.

Take away the idea that there are times when you can turn an `O(n^2)` solution into an `O(n)` solution with some critical thinking. Occasionally, keeping track of important values and updating them in the loop will vastly speed up a function.

← Back

Array Subset

Next →

Single Mutation

Mark as Completed

Report an Issue