

# Problem Set 8

CS 5800

Submitted by : Hardik Bishnoi

NUID : 002807991

# Problem 1

(a) Find a sequence with nine distinct integers for which the length of the longest increasing subsequence is 3, and the length of the longest decreasing subsequence is 3. Briefly explain how you constructed your sequence.

## Solution 1(a):

The longest increasing subsequence (LIS) is a problem where we must find the longest number of ordered elements (which may not be consecutive) known as a subsequence in a given array which is in increasing order. The longest decreasing subsequence is the opposite of the LIS, where the elements of the subsequence are in decreasing order.

Hence suppose we have an array  $[1, -1, 2, -2, 4, 3]$ , there is a longest subsequence of 3 which can be found with each element being greater than the next in the subsequence. We can find two such subsequences  $[1, 2, 4]$  and  $[1, 2, 3]$ . This is because when we pick the subsequence, we must make a choice to either pick 3 or 4, hence the longest subsequence can still only be of length three. The same case applies to the longest decreasing subsequence as well.

Hence, to effectively allow for there to be only length three for increasing and decreasing subsequences, we must maximize the choices that we can make. This means we can first put the group into an order. Suppose we have an array:

$$[4, 5, 6, 7, 8, 9, 10, 11, 12]$$

We can then rearrange them such that maximum number of these choices are made. This involves splitting the array into three groups from which we can choose 1 increasing or decreasing subsequence member.

$$[4, 5, 6], [7, 8, 9], [10, 11, 12]$$

We only need to make sure that 1 choice is being made from each triplet.

Now, we will arrange these triplets in decreasing order, this will ensure that the triplets do not interact with each other and the elements are only chosen once from these triplets and not twice.

Hence we land with the final array:

$$[10, 11, 12, 7, 8, 9, 4, 5, 6]$$

This arrangement only has a length of 3 for both LDS and LIS.

---

(b) Let  $S$  be a sequence with ten distinct integers. Prove that there must exist an increasing subsequence of length 4 (or more) or a decreasing subsequence of length 4 (or more).

Solution 1(b):

We can find a solution for with 10 distinct integers through the pigeonhole principle.

The pigeonhole principle states that when we have  $n$  items that are to be put into  $c$  containers, with  $n > c$ , then intuitively, at least one container must have more than 1 item.

Suppose we label each number in the array with  $n = 10$ , we can define ordered pairs  $(x(k), y(k))$ , where  $x(k)$  is the length of the longest increasing subsequence starting with  $k$  and likewise  $y(k)$  is the longest decreasing subsequence.

We know that there are distinct integers in the given array.

Hence according to the pigeonhole principle, all the ordered pairs that are fitting into these pigeonholes must be different since all the integers are distinct. Hence it is impossible for two different numbers in the sequence to be the same  $k$  or hence the same ordered pair.

Hence if we had ordered pairs for  $n = 10$ ,

$(1,1), (1,2), (1,3), (2,1), (2,2) \dots (3,3), (3,4)/(1,4)/..$

We know from part(a) that we can have 3 distinct solutions to LDS and LIS for  $n = 9$ . However, when we have 10 distinct integers, we will be left with one extra ordered pair between any of the 9 original distinct integers and the last integer. Hence if we had 3 original pigeonholes for both LIS and LDS, either LIS or LDS will obtain one extra ordered pair. Hence this will increase the count of LDS or LIS to at least 4.

---

**Solution 2(c):**

Initially we have the array  $S : [10, 11, 12, 7, 8, 9, 4, 5, 6]$ .

We can sort the array  $S$  to be  $S^*$ :

$$[4, 5, 6, 7, 8, 9, 10, 11, 12]$$

Now, we can use the LCS tabulation method to find the sequence:

		10	11	12	7	8	9	4	5	6
4	0	0	0	0	0	0	0	1	1	1
5	0	0	0	0	0	0	0	1	2	2
6	0	0	0	0	0	0	0	1	2	3
7	0	0	0	0	1	1	1	1	2	3
8	0	0	0	0	1	2	2	2	2	3
9	0	0	0	0	1	2	3	3	3	3
10	0	1	1	1	1	2	3	3	3	3
11	0	1	2	2	2	2	3	3	3	3
12	0	1	2	3	3	3	3	3	3	3

Hence we arrive to the combination [4,5,6].

Suppose we have  $n$  distinct integers in an array  $S$ , the longest increasing subsequence of an array  $S$  must be present in its sorted array  $S^*$  because the sequence is in increasing order. If there exists any longer sequence that is a part of both the arrays, it will have to be increasing in nature because  $S^*$  is in increasing order. Hence that subsequence will then become the longest increasing subsequence in  $S$ . Since we have determined the longest possible subsequence in  $S$  already, this is not possible. Hence through contradiction, we have established a general case.

---

(d) Given an input sequence  $S$  with  $n$  distinct integers, design a linearithmic algorithm (i.e., running in  $O(n \log n)$  time) to output the length of the longest increasing subsequence of  $S$ . Clearly explain how your algorithm works, why it produces the correct output, and prove that the running time of your algorithm is  $O(n \log n)$ .

### Solution 1(d):

We can follow a greedy approach which can allow us to gain a faster time on finding the solution:

```
function lengthOfLIS(nums: List[int]) -> int:
    sub = []

    for x in nums:
        if len(sub) == 0 or sub[-1] < x:
            sub.append(x)
        else:
            idx = bisect_left(sub, x)
            sub[idx] = x

    return len(sub)
```

Above is the pseudocode for the solution. The code runs in  $O(n \log n)$  time

The loop iterates through each element in the `nums` array, and each iteration involves a constant amount of work. Therefore, the loop has a time complexity of  $O(n)$ . However, the `bisect_left` function, which performs binary search, has a time complexity of  $O(\log n)$ . Since we perform this operation for each element in the array, the total time complexity contributed by binary search is  $O(n \log n)$ .

The algorithm works by picking the first element and inserting it into `sub`. Then, if the next element is less than the previous number, `sub = [10]`. If the next number is greater than the previous number, add it to `sub`. We keep doing this till we encounter a number less than the previous number, hence `sub = [10,11,12]`. We try to find a number greater than the current number in the subsequence. If we find such a number, then we will replace the number with the current number. Hence the array becomes `[7,11,12]`. We keep doing this till we cannot add any numbers or extend the subarray anymore, and nor can we replace any numbers within the subarray. The final result is `[4,5,6]`. Hence the algorithm works.

# Problem 2

(a) Clearly explain why  $p_1(s) = 0$  for  $s \leq -3$ ,  $p_1(s) = 1/2$  for  $-2 \leq s \leq 1$ , and  $p_1(s) = 1$  for  $s \geq 2$ .

Explain why you must select X if  $s$  is 2 or 3, and you must select Y if  $s$  is -2 or -1.

## Solution 2(a):

Let the probability of flipping the coin X and getting heads is  $p_x = 1/2$  assuming it's a fair coin, and similarly that of Y be  $p_y = 1/2$ . At any point we can choose either X or Y coins to toss.

At our base case, let's say we start with the score zero and we have only one turn:

If we choose to flip coin X, we have :

<b>X</b>	<b>Score</b>
H	1
T	-1

Hence the probability of scoring positive is  $1/2$ .

If we choose to flip coin Y instead,

<b>Y</b>	<b>Score</b>
H	3
T	-3

The probability of scoring positive is  $1/2$

If we have our starting score as  $\leq -3$ ,

<b>X</b>	<b>Score</b>
H	-2
T	-4

<b>Y</b>	<b>Score</b>
H	0
T	-6

Hence there are no scenarios we will win. Hence,  $p_1(-3) = 0$ .

For  $-2 \leq s \leq 1$ ,

When  $s = -2$ ,

<b>X</b>	<b>Score</b>
H	-1
T	-3

<b>Y</b>	<b>Score</b>
H	1
T	-5

Since Y has the maximum probability to win, we will choose Y.

When  $s = 1$ ,

<b>X</b>	<b>Score</b>
H	2
T	0

<b>Y</b>	<b>Score</b>
H	4
T	-2

Hence we can choose either of the coins since both their probabilities to win are the same.

Hence,  $p_1(s) = 1/2, -2 \leq s \leq 1$

Next, when  $s = 2$

<b>X</b>	<b>Score</b>
H	3
T	1

<b>Y</b>	<b>Score</b>
H	5
T	-1

Hence we will choose coin X since it has the maximum chances of winning since it will always win. Therefore,  $p_1(s) = 1$  for  $s \geq 2$ .

---

(b) For each possible value of  $s$ , determine  $p_2(s)$ . Clearly explain how you determined your probabilities, and why your answers are correct. (Hint: each probability will be one of  $0/4$ ,  $1/4$ ,  $2/4$ ,  $3/4$ , or  $4/4$ .)

**Solution 2(b):**

We can observe from part (a) that how the best possible probability can be chosen (by choosing the maximum probability of winning every time), and how the probabilities are impacted by the previously attained score,  $s$ .

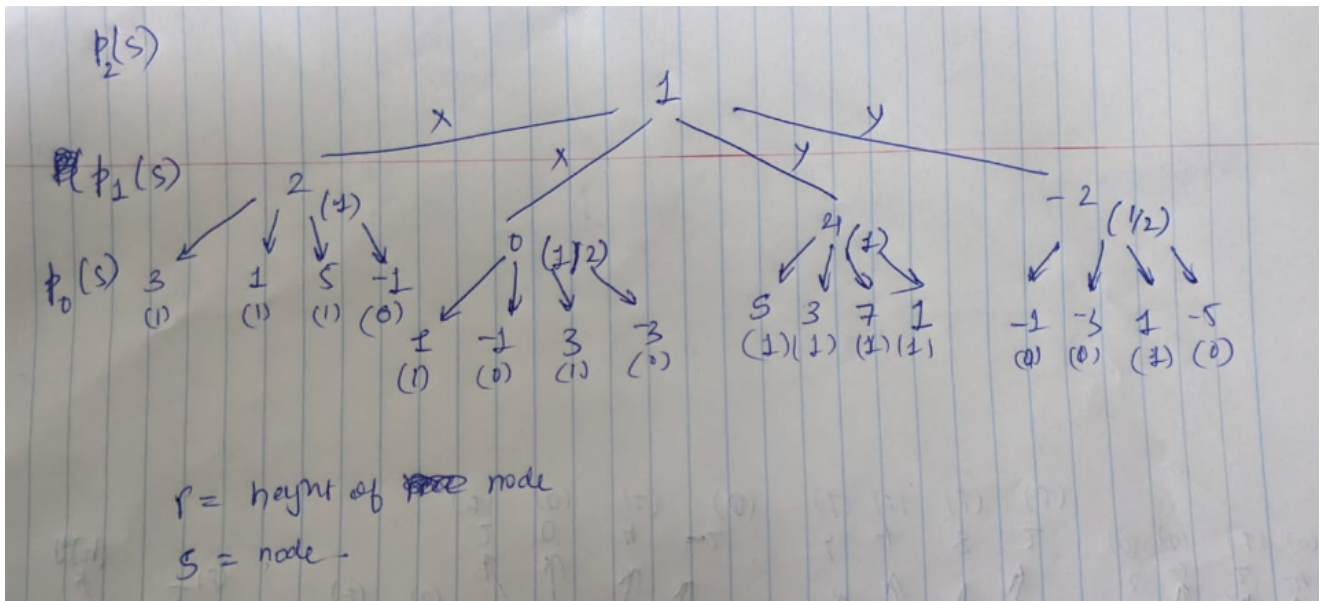
We can represent the probability sets in the form of probability trees or tables.

We also know that there exist 3 cases for how value of the next move can be determined through  $s$  falling into 3 domains in which are:

1.  $p_1(s) = 0, s \leq -3$
2.  $p_1(s) = 1/2, -2 \leq s \leq 1$
3.  $p_1(s) = 1, s \geq 2$

We can draw probability trees (or an event tree) for all of these cases and through these probability trees work on the underlying recursive pattern.

For  $p_1(s) = 1/2, -2 \leq s \leq 1$



We can see that in the tree, the height of the node corresponds to  $r$  in  $p_r(s)$ , or the remaining moves. The nodes themselves are  $s$  or the score events and there are nodes that represent each possible move.

We can hence determine the probability by looking at the tree:

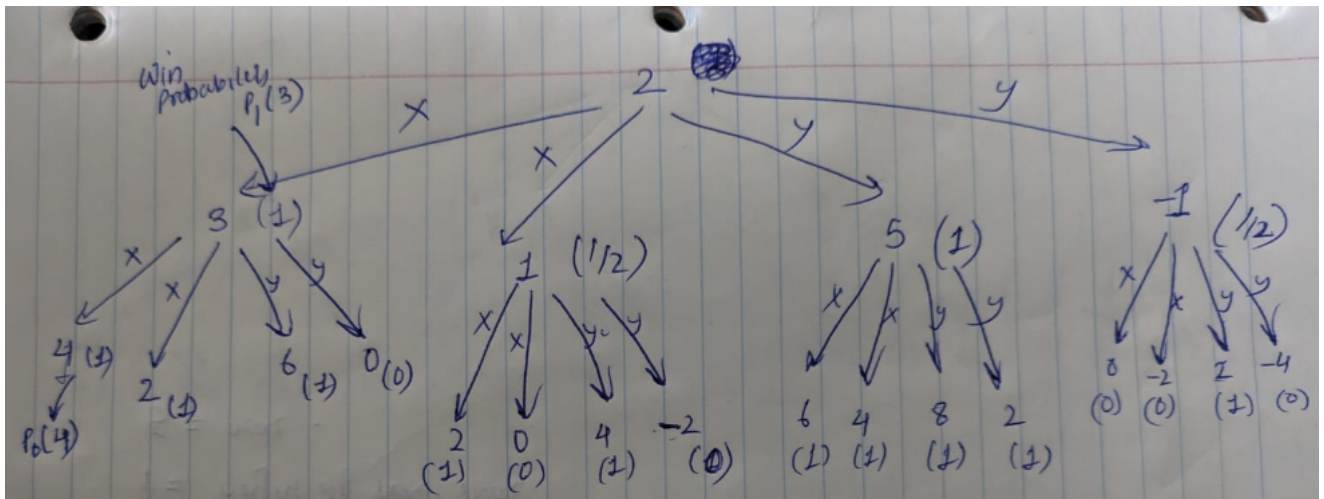


We know that for  $h = r = 2$ , there are 4 possibilities in the next level, 2, 0, 4 and -2. Through our base case we found in (a), we can tell all the scenarios that may take place in the next iteration according to the results of the current iteration. Hence we can use this knowledge to effectively select the outcome with the maximum probability. Here, regardless of if we choose coin X or Y, we will have either 1 or 1/2 probability. We can combine the probability of winning through all outcomes on choosing a particular coin by averaging the probabilities. For coin X it will hence be 3/2 and for coin Y it will also be 3/2. Hence we choose any of the coins.

Suppose we choose coin X. If we win, then there we have half a chance to select X in the next step which means we will certainly win. If we lose, we have 1/2 a chance of winning or losing through any of the coins.

Hence the total probability is :  $3/4$ .

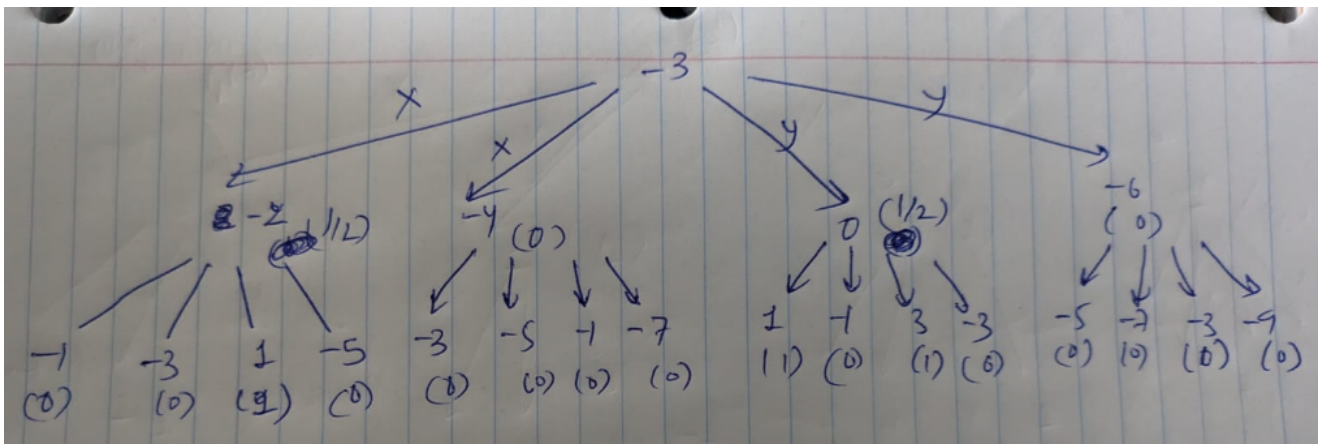
Next, we look at the case  $p_1(s) = 1, s \geq 2$



Here, when we first are to choose the coin from the root node = 2, we land with 4 possibilities at  $h = r = 1$ . We can see that combined probability of winning the game (using  $p_1(s)$ ) for X and Y both is 3/2 again. Hence, we can choose either X and Y. Let us arbitrarily choose X. Since When X is heads and we can choose either X or Y again for the final result, we choose coin X. If X is tails we can choose either X or Y.

Hence our combined probability will be again,  $3/4$ .

Next, we will tackle the case  $p_1(s) = 0, s \leq -3$



Again, in this case we will either pick X or Y since the combined probability of both of them is  $1/2$ .

Hence, according to our previous observations, the combined probability here as well will be  $1/4$

(c) Find a recurrence relation for  $p_r(s)$ , which will be of the form  $p_r(s) = \max(??+??/2, ??+??/2)$ . Clearly justify why this recurrence relation holds. From your recurrence relation, explain why the optimal strategy is to pick X when you have certain positive scores (be conservative) and pick Y when you have certain negative scores (be aggressive).

### Solution 2(c):

In part (b) we observed that how the outcome of a given level is effected by the outcomes of the previous level in the tree. We can see a pattern that arises that the combined probability of winning at the current level is equal to the combined probability of winning at the previous level divided by half.

Since at each level, we can either choose X or Y which can result in the outcomes  $+1$ ,  $-1$  and  $+3$ ,  $-3$  to the score, to represent the winning probability at the current level in terms of the previous level we must factor in the added or subtracted score in the probabilities.

Hence probability of X winning in the previous level becomes:

$$p_{r+1}(s + 1)$$

And for losing it becomes:

$$p_{r+1}(s - 1)$$

Similarly for Y, on winning:

$$p_{r+1}(s + 3)$$

and on losing :

$$p_{r+1}(s - 3)$$

Hence the combined probability for X is :  $p_{r+1}(s + 1) + p_{r+1}(s - 1)$

And for Y :  $p_{r+1}(s + 3) + p_{r+1}(s - 3)$

As we noticed, the combined probability of winning at the current level is equal to the combined probability of winning at the previous level divided by half, the final recurrence becomes:

$$p_r(s) = MAX(\frac{p_{r+1}(s + 1) + p_{r+1}(s - 1)}{2}, \frac{p_{r+1}(s + 3) + p_{r+1}(s - 3)}{2})$$

The best stratagem when we have positive  $s \in 3, 4$  is to pick X because the max function will always pick X because X has a higher probability of 1 compared to  $3/4$  when we pick Y. This is because X will always produce a winning result compared to Y which produces a winning result only half the time when it loses. Hence to ensure victory, we pick X.

### Solution 2(d):

We can use memoization technique to formulate an algorithm:

```
function coinscore(r, s, memo):
    if r == 0:
        return 1 if s >= 1 else 0

    if (r, s) is in memo:
        return memo[(r, s)]

    prob = max(
        (coinscore(r - 1, s - 1, memo) + coinscore(r - 1, s + 1, memo) + coinscore(r - 1, s - 3, memo) + coinscore(r - 1, s + 3, memo)) / 2
    )

    memo[(r, s)] = prob
    return prob
```

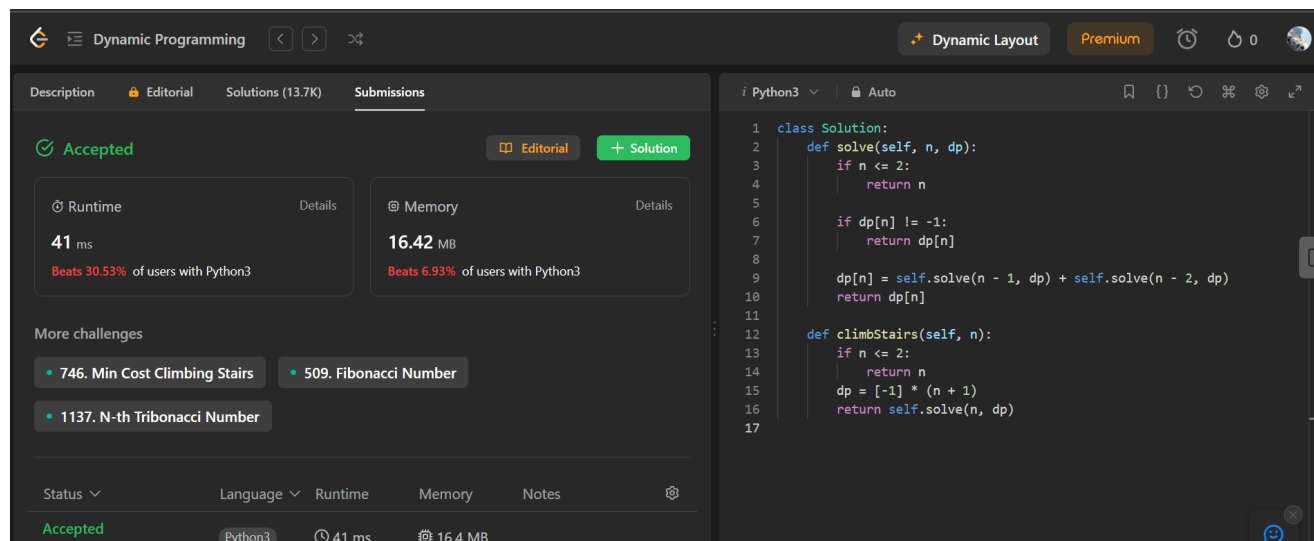
The function calculates a probability based on recursive calls and stores intermediate results in the memo dictionary to avoid redundant computations.

The solution to this is 0.71.

# Problem 3

Problem : [70. Climbing Stairs](#)

Difficulty: Easy



Source:

```
class Solution:
    def solve(self, n, dp):
        if n <= 2:
            return n

        if dp[n] != -1:
            return dp[n]

        dp[n] = self.solve(n - 1, dp) + self.solve(n - 2, dp)
        return dp[n]

    def climbStairs(self, n):
        if n <= 2:
            return n
        dp = [-1] * (n + 1)
        return self.solve(n, dp)
```

Complexity Analysis:

The algorithm uses dynamic programming to store solutions to subproblems in the dp array. The array dp has a length of  $n + 1$ , where each entry in the array corresponds to a subproblem. The solve function is a recursive function that computes the number of ways to climb the stairs for a given step n by summing up the results of subproblems for steps n-1 and n-2. The result for each step is memoized in the dp array, so that if a subproblem has been solved before, its result can be directly retrieved from the array without recomputing. As a result, each step is computed only once, and the time complexity is linear in terms of the input size n. Hence the complexity of my approach is  $O(n)$ .

## Problem : [119. Pascal's Triangle II](#)

Difficulty: Easy

The screenshot shows the LeetCode interface for problem 119. The submission is marked as 'Accepted'. The runtime is 38 ms, which beats 57.87% of users with Python3. The memory usage is 16.30 MB, which beats 8.66% of users with Python3. The code is as follows:

```
1 class Solution:
2     def getRow(self, row):
3         ans = [1]
4
5         temp = 1
6
7         for i in range(row):
8             temp = temp * (row - i)
9             temp = temp // (i + 1)
10            ans.append(temp)
11
12        return ans
13
```

Source:

```
class Solution:
    def getRow(self, row):
        ans = [1]

        temp = 1

        for i in range(row):
            temp = temp * (row - i)
            temp = temp // (i + 1)
            ans.append(temp)

        return ans
```

### Complexity analysis:

The time complexity of the code is  $O(n)$  where  $n = \text{row}$ . This is because the code uses a loop that iterates from 0 to row, performing constant-time operations in each iteration. The dominant factor in the time complexity is the loop

## Problem: [121. Best Time to Buy and Sell Stock](#)

Difficulty: Easy

The screenshot shows the LeetCode interface for problem 121. The submission is marked as 'Accepted'. The runtime is 797 ms, which beats 90.83% of users with Python3. The memory usage is 27.47 MB, which beats 6.57% of users with Python3. The code is as follows:

```
1 class Solution:
2     def maxProfit(self, prices):
3         ans = 0
4         minValue = prices[0]
5
6         for i in range(1, len(prices)):
7             tmpSum = prices[i] - minValue
8             if tmpSum > ans:
9                 ans = tmpSum
10            if prices[i] < minValue:
11                minValue = prices[i]
12
13        return ans
14
```

Source:

```
class Solution:
    def maxProfit(self, prices):
        ans = 0
        minValue = prices[0]

        for i in range(1, len(prices)):
            tmpSum = prices[i] - minValue
            if tmpSum > ans:
                ans = tmpSum
            if prices[i] < minValue:
                minValue = prices[i]

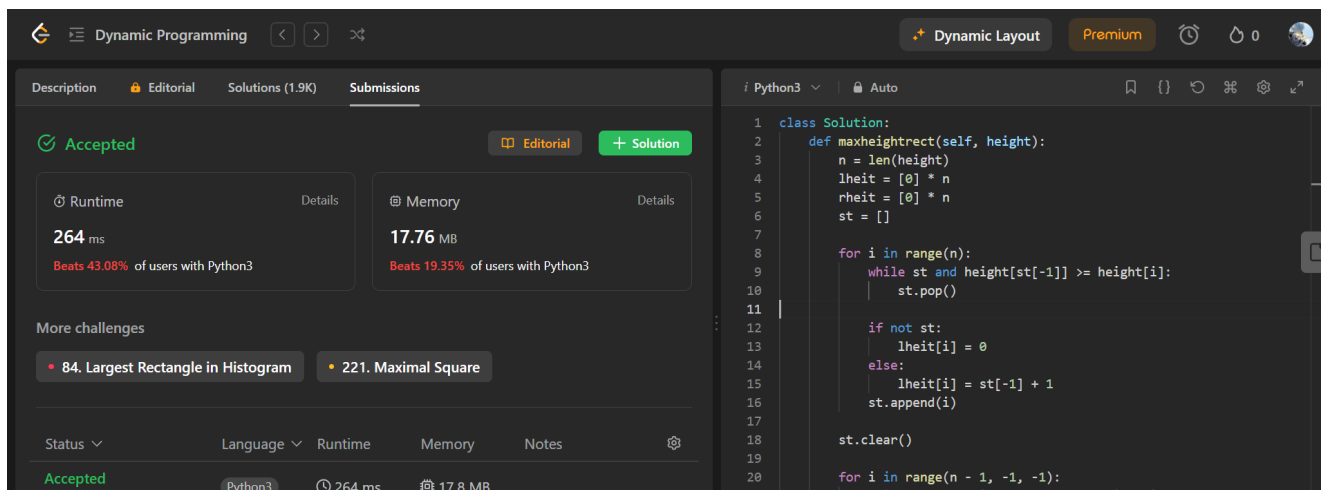
        return ans
```

### Complexity Analysis:

The time complexity of the code is  $O(n)$ , where  $n$  is the length of the input list `prices`. The algorithm iterates through the list once using a loop. In each iteration, it performs constant-time operations. Therefore, the overall time complexity is linear in terms of the input size.

Problem: [85. Maximal Rectangle](#)

Difficulty: Hard



The screenshot displays a LeetCode submission interface. On the left, the 'Submissions' tab is active, showing an 'Accepted' status. Performance metrics indicate a runtime of 264 ms (beating 43.08% of users) and memory usage of 17.76 MB (beating 19.35% of users). Below these metrics, suggested challenges include '84. Largest Rectangle in Histogram' and '221. Maximal Square'. The right side of the interface shows the Python code for the `maxheightrect` method. The code uses a stack to process the histogram heights, maintaining a list of indices for the current stack of bars.

Source:

```
class Solution:
    def maxheightrect(self, height):
        n = len(height)
        lheit = [0] * n
        rheit = [0] * n
        st = []

        for i in range(n):
            while st and height[st[-1]] >= height[i]:
                st.pop()
```

```

        if not st:
            lheit[i] = 0
        else:
            lheit[i] = st[-1] + 1
        st.append(i)

    st.clear()

    for i in range(n - 1, -1, -1):
        while st and height[st[-1]] >= height[i]:
            st.pop()

        if not st:
            rheit[i] = n - 1
        else:
            rheit[i] = st[-1] - 1
        st.append(i)

    maxi = 0
    for i in range(n):
        maxi = max(maxi, height[i] * (rheit[i] - lheit[i] + 1))

    return maxi

def maximalRectangle(self, matrix):
    n = len(matrix)
    m = len(matrix[0])
    maxi = 0
    height = [0] * m

    for i in range(n):
        for j in range(m):
            if matrix[i][j] == '1':
                height[j] += 1
            else:
                height[j] = 0

        longest = self.maxheightrect(height)
        maxi = max(maxi, longest)

    return maxi

```

### Complexity Analysis:

The maximalRectangle function iterates through each element of the input matrix once, updating the height array and computing the maximum rectangle area using the maxheightrect function. Therefore, the overall time complexity is  $O(n * m)$ , where  $n$  is the number of rows in the matrix, and  $m$  is the number of columns.

### 3(b)

The last given problem involves finding the maximal rectangle in a binary matrix, which can be approached by finding the maximum histogram area for each row. I first tried a brute force approach which did not prove to be a correct solution and did not pass many edge cases like when there are no elements in the stack. The complexity also proved to be on the order of  $O(n^3)$ . Hence this made me realise that pre-handling of the various column lengths

is important. This was an important insight since I managed to then lower my time complexity to  $O(MN)$ .