Question 1

Solution to (a)

Consider price table below:

Length	1	2	3	4
Price	1	5	8	9
Density	1	2.5	2.667	2.25

The greedy approach will cut rope into 3 and 1 with $r = 9$, where the optimal solution is to cut into 2 and 2 with $r = 10$. So the optimal solution is larger in value than the greedy solution.

Solution to (b)

```
define CUT-ROD(p, n, c)

 initialize an array r[0..n] to 0

 for i = 1 to n (inclusive)
   q = p[i]
   for j = 1 to i-1 (inclusive)
     q = max(q, p[j-1] + r[i-j]-c)
   r[i] = q
 return r[n]
```

Question 2

Solution to (a)

Denote pebble in row i as P_i,

$$P_i = 1 \text{ if there is a pebble}$$
$$P_i = 0 \text{ if there is no pebble}$$

	Case 0	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7
R1	0	1	1	1	0	0	0	0
R2	0	0	0	0	1	1	0	0
R3	0	0	1	0	0	0	1	0
R4	0	0	0	1	0	1	0	1
S	0	1	2	2	1	2	1	1

Since there's no vertically adjacent pebble, based on the table above, the maximum number of pebbles that can be placed is 2.

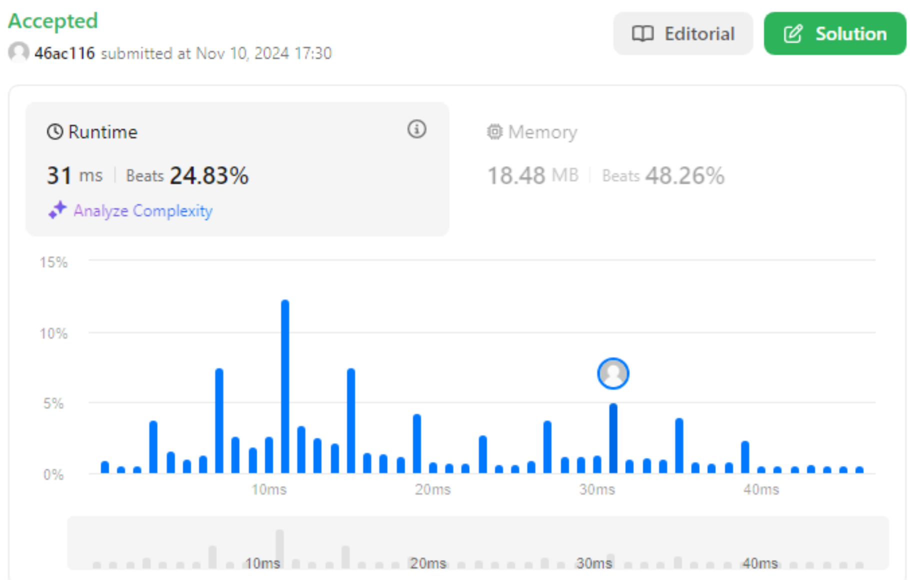
Solution to (b)

```
def print_pebble_game(placement):
    for x in placement[1:]:
        print(bin(x)[2:].zfill(4))

cases = [0, 0b1000, 0b1010, 0b1001, 0b0100, 0b0101, 0b0010, 0b0001]
def sum_selected_by_binary(column, binary_number):
    total_sum = 0
    for i in range(len(column)):
        if binary_number & (1 << i):
            total_sum += column[i]
    return total_sum

def pebble_game(table, n):
    placement = [0] * (n + 1)
    for i in range(1, n + 1):
        max_sum = 0
        for case in cases:
            temp_sum = sum_selected_by_binary(table[i-1], case)
            if placement[i-1] & case == 0 and temp_sum > max_sum:
                max_sum = temp_sum
                placement[i] = case
    return placement
```

We uses bit operation here to save the inconvenience of compatibility check. The AND operation is used to check if the current case is compatible with the previous column. Since the number of cases is 8 and the maximum bits is 4, the inner loops can be considered as having constant time, the time complexity of this algorithm is $O(n)$.

Question 3

42. Trapping Rain Water (Hard)



Code | Python3

```python
class Solution:
    def trap(self, height: List[int]) -> int:
        l = len(height)
        left = [0] * l
        right = left.copy()

        left[0] = height[0]
        right[-1] = height[-1]

        for i in range(1, l):
            left[i] = max(height[i], left[i-1])

        for i in range(l-2, -1, -1):
            right[i] = max(height[i], right[i+1])

        s = 0
        for i in range(1, l-1):
            s += min(left[i], right[i]) - height[i]

        return s
```

View less

The algorithm iterate through the height 3 times with each time have a cost of $O(n)$, so the total is still $O(n)$

1526. Minimum Number of Increments on Subarrays to Form a Target Array (Hard)

Accepted

46ac116 submitted at Nov 10, 2024 20:44

Solution

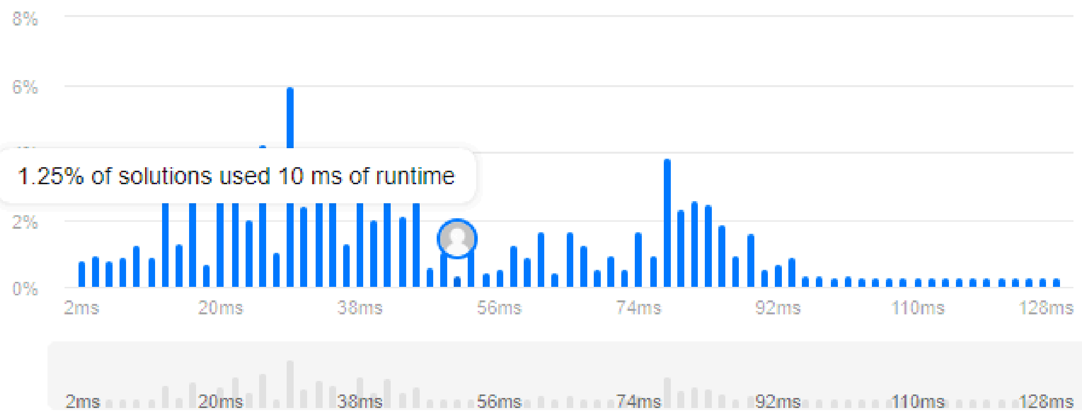
Runtime

51 ms | Beats 40.82%

Analyze Complexity

Memory

26.28 MB | Beats 66.80%



Code | Python3

```python
class Solution:
    def minNumberOperations(self, target: List[int]) -> int:
        n = len(target)
        dp = [0] * n
        dp[0] = target[0]
        for i in range(1, n):
            if target[i] <= target[i-1]:
                dp[i] = dp[i-1]
            else:
                dp[i] = dp[i-1] + (target[i] - target[i-1])
        return dp[-1]
```

View less

The algorithm iterate over the size of the target so the time complexity is $O(n)$