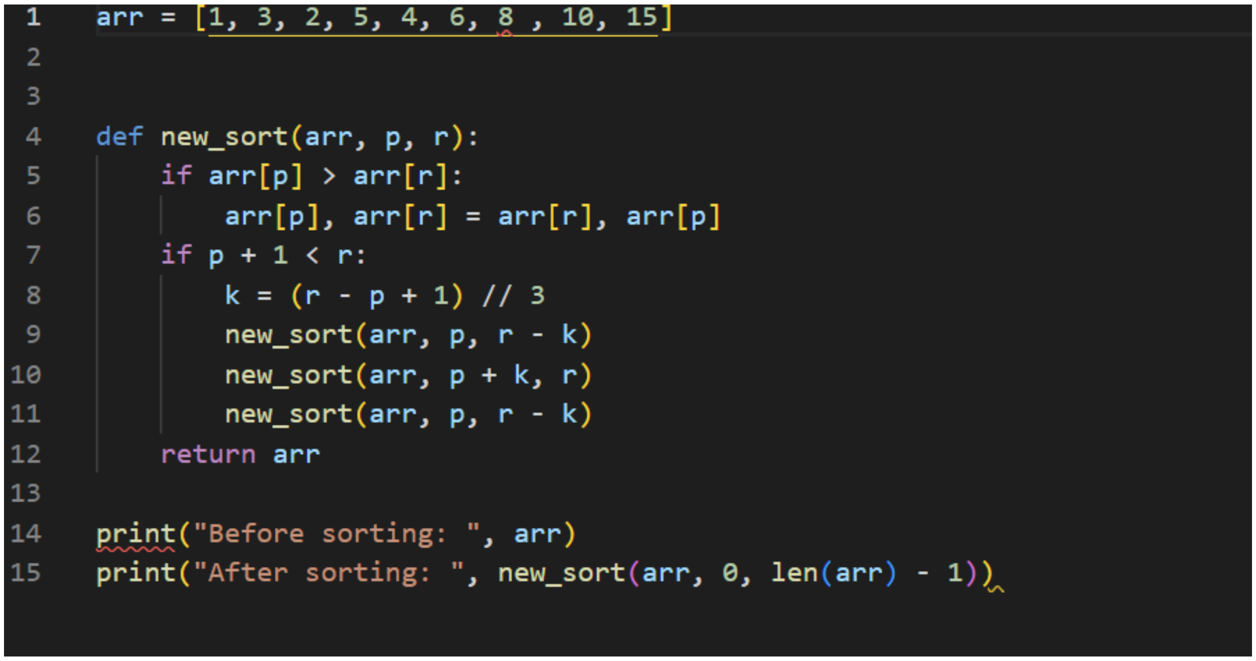
P1

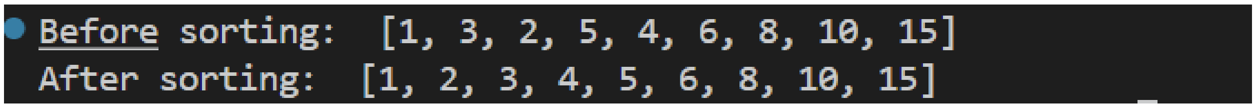
a)



```python
arr = [1, 3, 2, 5, 4, 6, 8 , 10, 15]


def new_sort(arr, p, r):
    if arr[p] > arr[r]:
        arr[p], arr[r] = arr[r], arr[p]
    if p + 1 < r:
        k = (r - p + 1) // 3
        new_sort(arr, p, r - k)
        new_sort(arr, p + k, r)
        new_sort(arr, p, r - k)
    return arr

print("Before sorting: ", arr)
print("After sorting: ", new_sort(arr, 0, len(arr) - 1))
```



```
Before sorting:  [1, 3, 2, 5, 4, 6, 8, 10, 15]
After sorting:  [1, 2, 3, 4, 5, 6, 8, 10, 15]
```

For base case, n =1, if array has one element, it is sorted. This algorithms works because it ensures that the smallest value will be at far left. Largest value will be set to the far right. The first recursive call will sort the first two thirds of the array. The second recursive call will sort last two thirds of the array. And the last recursive call will make sure first two thirds of the array and last two thirds of the array are sorted. Hence, we can prove that the NEW-SORT (A, 1, n) correctly sorts the input array A [1 : n]

b)

T(n) = 3T (2n/3)+ O(1)

$$T(n) = 3T\left(\frac{2n}{3}\right) + O(1)$$

$$a = 3, \ b = \frac{3}{2}$$

$$\log_{\frac{3}{2}} 3 = 2.71, \text{ Case1}$$

$$O(n^{2.71})$$

c)

compare with Insertion Sort(O(n2) at worst), Merge Sort(O(nlogn)at worst),Heap Sort(O(nlogn) at worst),, Quick sort(O(n2) at worst), it is a very inefficient sorting algorithm. The student does not deserve a straight A

P2

a)

$\underbrace{A \quad B \quad C \quad D \quad E \quad F \quad G \quad H}_{\text{compare each one of them}}$

$\downarrow$

8! ways = 40320

each comparison is
decision tree

$\downarrow$

which means
after n matches,
there will be 2^n
outcomes and at
least 40320 ending
is needed

$2^n \geq 40320$

$n \geq \log_2(40320)$

$n \geq 15.3$

So we will need
at least 16 matches
to sort all 8 players

b)

A B C D E F G H   8

max = 4 matches to play against among 8 players, winner will enter next until find the best player

Min = 2 matches to play against among 8 players, after first match loser will enter next until find the worst.

max = 4 + 2 + 1 = 7

min = 2 + 1 = 3

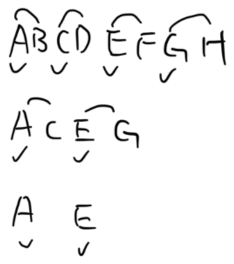
7 + 3 = 10 matches

So, at least 10 matches needed

c)

$$\underbrace{A\ B\ C\ D}\ \underbrace{E\ F\ G\ H}$$

$$4 + 2 + 1 + 2 = 9$$



make 8 players play against each other in pair of 2.

4 matches in 8 players
2 matches for 4 winners
1 match for selecting best player. so we have 7 matches to find highest rating player. For second best, we know it must within the first 4 winners we find for best player. we know one of them is the best play. so, it must within the 3 winners. 2 matches needed to find the second best. (4+2+1+2=9)
Hence, there does not exist a solution in 8 or fewer matches

P3

a)

[4, 5, 0, 1, 3, 4, 3, 4, 3, 0, 3]

① find max

max = 5

② count [$\overset{0}{0}$, $\overset{1}{0}$, $\overset{2}{0}$, $\overset{3}{0}$, $\overset{4}{0}$, $\overset{5}{0}$] max+1 = 6

③ count occurence

↓

index	0	1	2	3	4	5
count [	2	1	0	4	3	1]

↓

④ modify count array

index	0	1	2	3	4	5
count [	2	3	3	7	10	11]

⑤ Place into order

[0 0 1 3 3 3 3 4 4 4 5]

1. 3 → count [3] = 7
2. 0 → count [0] = 2
3. 3 → count [3] = 6
4. 4 → count [4] = 10
5. 3 → count [3] = 5
6. 4 → count [4] = 9
7. 3 → count [3] = 4
8. 1 → count [1] = 3
9. 0 → count [0] = 1
10. 5 → count [5] = 10
11. 4 → count [4] = 8

b)

The algorithm was process from right to left, which means last occurrence of an element gets the highest available position. So, the array maintained their relative order. Also, the placement order is fixed based on the count array. Plus, counting sort doesn't require comparison. So no swap is needed.

c)

No, quicksort and heapsort are not stable.

For quick sort,

The partition of quicksort would require swapping of elements. The order may gets changed when elements are on opposite sides of the pivot.

Ex.

5(red), 3, 5(blue), 2

After quick sort:

[2, 3, 5(blue), 5(red)]

For heapsort,

Heapsort builds a heap and then extract max or min. Heapify operations can move elements out of order.

Ex.

[4(red), 2, 4(blue), 3]

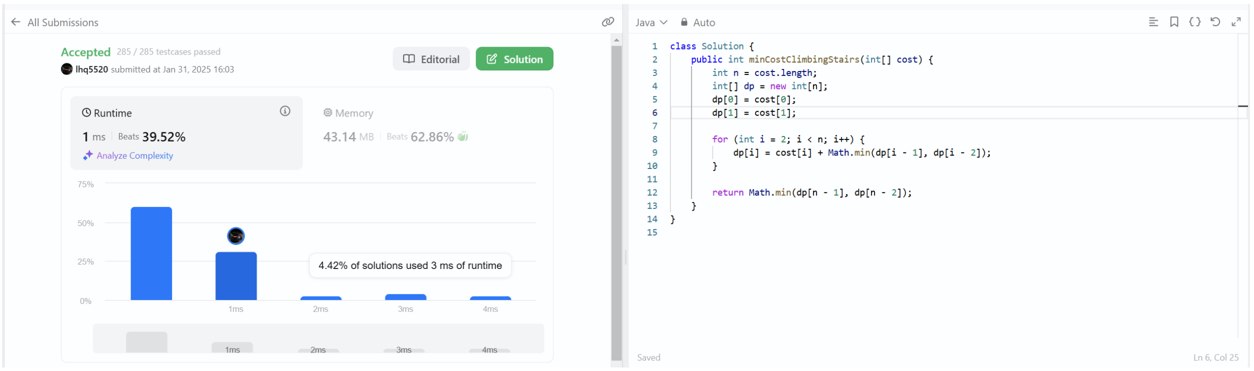
After heap sort:

[2, 3, 4(red), 4(blue)]

P4

a)

LeetCode 746: Min Cost Climbing Stairs, Easy



All Submissions

Accepted 285 / 285 testcases passed
lhq5520 submitted at Jan 31, 2025 16:03

Runtime: 1 ms | Beats 39.52%
Memory: 43.14 MB | Beats 62.86%

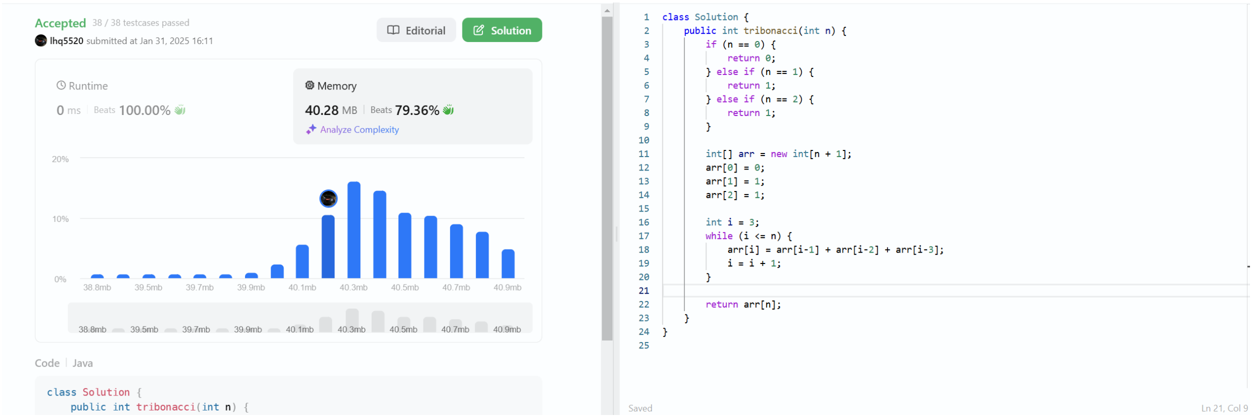
4.42% of solutions used 3 ms of runtime

```java
class Solution {
    public int minCostClimbingStairs(int[] cost) {
        int n = cost.length;
        int[] dp = new int[n];
        dp[0] = cost[0];
        dp[1] = cost[1];

        for (int i = 2; i < n; i++) {
            dp[i] = cost[i] + Math.min(dp[i - 1], dp[i - 2]);
        }

        return Math.min(dp[n - 1], dp[n - 2]);
    }
}
```

LeetCode 1137: N-th Tribonacci Number, Easy



Accepted 38 / 38 testcases passed
lhq5520 submitted at Jan 31, 2025 16:11

Runtime: 0 ms | Beats 100.00%
Memory: 40.28 MB | Beats 79.36%

```java
class Solution {
    public int tribonacci(int n) {
        if (n == 0) {
            return 0;
        } else if (n == 1) {
            return 1;
        } else if (n == 2) {
            return 1;
        }

        int[] arr = new int[n + 1];
        arr[0] = 0;
        arr[1] = 1;
        arr[2] = 1;

        int i = 3;
        while (i <= n) {
            arr[i] = arr[i-1] + arr[i-2] + arr[i-3];
            i = i + 1;
        }

        return arr[n];
    }
}
```

b)

I used a brute-force recursive method when I originally tried to solve the Tribonacci problem. I created a function that recursively called itself for n-1, n-2, and n-3, with base cases for n = 0, n = 1, and n = 2, because the sequence follows a distinct pattern in which each integer is the sum of the preceding three. At first, I felt this method made sense because recursion matched the issue statement exactly, and I assumed it would be a simple method to obtain the right answer. But when I went through the various test cases, I saw that the code was repeatedly recalculating the same values. In order to avoid having to recalculate the findings, this made me think about saving previously calculated results.

I used an array to implement memoization in order to better my strategy. I made sure that every number in the sequence was only calculated once by keeping the outcomes of each function call. I was able to access values instantly rather than recalculating them, which greatly reduced the number of unnecessary calculations. Although this was an improvement, I discovered that even though I only ever used the last three numbers at a time, I was still storing all values up to n in an array.

With this realization, I further improved my strategy by tracking the final three digits in the sequence using an iterative solution with just three variables. This kept the results accurate while doing away with the necessity for an array. The final approach was more memory and compute efficient because it employed a loop rather than recursive logic, which it closely mimicked.

I was able to comprehend how many strategies built upon one another because to this challenge. I gained a thorough grasp of the problem's structure by beginning with a straight recursive solution, and then I worked to optimize it gradually. I learned from the process how crucial it is to know when it makes sense to store intermediate findings and when it is not. It also reaffirmed that there are frequently several methods to address a problem and that trying out various strategies might yield deeper insights into enhancing coding clarity and efficiency.