Problem #1

(a)

The sequence I found is , with following steps:

Firstly, I chose nine distinct integers from 1 to 9, and group them into 3 sets: .

9

6 8

3 5 7

2 4

1

Secondly, I would arrange in its decreasing order respectively, as pointed by the green arrow. And this will make sure the length of any longest decreasing subsequence is 3.

Since every element in its subsequence is in decreasing order, if we put the subsequence whose first integer is the smallest, which is , at the beginning of the final sequence, then every rest element in would form an increasing subsequence with the first integer of other subsequences, as pointed by the orange arrow.

Lastly, since we group these nine elements into 3 sets, it is assured that the length of longest increasing subsequence is 3.

(b)

Proof by contradiction.

Let us assume that there exist two different numbers in the sequence to be represented by the same ordered pair.

.

and is distinct,

Either is larger than or is larger than . Let us assume that .

Case 1, assume is at the left side of in the sequence:

There are decreasing numbers at the right of , and ,

is one of the decreasing numbers at the right of ,

. This is a contradiction since we assume .

Case 2, assume is at the right side of in the sequence:

There are increasing numbers at the right of , and ,

is one of the increasing numbers at the right of ,

. This is a contradiction since we assume .

It is impossible for two distinct numbers in the sequence to have the identical pair.

(c)

The length should be 3, and the proof is as follows:

From previous homework and learning, we know that the time complexity of insertion sort, merge sort, heapsort and quicksort in the worst case is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Insertion Sort | Merge Sort | Heap Sort | Quick Sort |
| Time Complexity |  |  |  |  |

Problem #2

(a)

There does not exist an algorithm that is guaranteed to solve this problem in 15 or

fewer matches, since any comparison sort algorithm requires comparisons in the worst case.

Since it is a problem that we sort 8 students with a comparison sort algorithm, we could formulate this problem into sorting an array with length 8 using a comparison sort algorithm. Then assume a decision tree of height with reachable leaves corresponding to such comparison sort on 8 elements.

Each of the permutations appear as one or more leaves,

.

A binary tree of height has no more than leaves,

,

,

.

,

The height will be at least 16,

There does not exist an algorithm that is guaranteed to solve this problem in 15 or

fewer matches.

(b)

The simultaneous minimum and maximum algorithm could be applied to solve the problem here. There does not exist an algorithm that is guaranteed to solve this problem in 9 or fewer matches, since at least  comparisons are necessary to find both the maximum and minimum of  numbers in the worst case.

First of all, the pseudocode for the simultaneous minimum and maximum algorithm is as follows:

Simultaneous-Min-Max(arr):

if arr is empty:

return None, None

if the length of arr is even:

min = min(arr[0], arr[1])

max = max(arr[0], arr[1])

start\_index = 2

else:

min = max = arr[0]

start\_index = 1

for i = every other number from start\_index to length of arr – 1:

if arr[i] < arr[i + 1]:

min = min(min, arr[i])

max = max(max, arr[i + 1])

else:

min = min(min, arr[i + 1])

max = max(max, arr[i])

return min, max

Secondly, from above implementation, we know that this algorithm takes every other element as a pair and compares the two elements in that pair and compares the smaller one with the current minimum and the larger one with the current maximum, which leads to 3-time comparison.

One pair consists of two elements,

This algorithm would group the array into pairs when the length is even, and pairs and 1 element left when the length is odd.

When the length is even, the number of comparisons executed by this algorithm would be 1 comparison of the first two elements, then comparisons for all the pairs, which would be in total.

When the length is odd, the number of comparisons executed by this algorithm would be 2 more comparisons among 1 element and the minimum as well as the maximum of the n – 1 array, which is .

At least  comparisons are necessary to find both the maximum and minimum of  numbers in the worst case.

The number of the students is 8,

There should be at least comparisons to determine the highest and the lowest chess rating, namely there does not exist an algorithm that is guaranteed to solve this problem in 9 or fewer matches.

(c)

A similar algorithm to above simultaneous minimum and maximum algorithm could be applied to solve the problem here. There does not exist an algorithm that is guaranteed to solve this problem in 8 or fewer matches, since comparisons are necessary to find the second maximum of  numbers in the worst case.

First, the algorithm splits all elements into pairs of 2 elements and compares them. Then, every winner from the last round would be split into pairs again and be compared with each other.

The total number of comparisons executed by this algorithm would be .

Every element except the overall winner must lose exactly 1 comparison,

There must be comparisons in total.

This algorithm will take comparisons to find the largest of elements.

There would be rounds of comparisons, and the second largest element would be compared to some element once in every round. It must be the last element that lost to the largest as well,

Except for the last comparison where the second largest lost the largest, there should be comparisons to determine the second largest element.

In total, it takes comparisons to find the second largest of elements in the worst case.

It takes at least comparisons to find the highest chess rating player and the second highest chess rating player, and there does not exist an algorithm that could solve this problem in 8 or fewer matches.

Problem #3

(a)

, counts the occurrences of elements in ,

Let be an array of length ,

0 1 2 3 4 5

.

After calculating the running time, would be:

0 1 2 3 4 5

Let be the output array. According to the counting sort we know that:

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

,

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

,

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

,

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

,

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

,

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

,

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

,

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

,

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

,

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

,

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

,

The out put .

(b)

Since is the array of the occurrences of elements in , we know that the value in represents the last index of the same element. So similarly, we need to start backwards to be consistent with that logic. By employing that logic, the counting sort is stable.

Let us use an example to elaborate above explanation.

Let , from (a) we know , namely, the last 0 should be put at index 0 (0-index), the last 1 should be put at index 2, and the last 2 which is the blue 2 should be put at index 5. So, the output array , which provides the stability. In controversy, if we start put elements from onwards, this algorithm will become unstable.

(c)

Neither Quick sort nor Heap sort is stable. The counterexample is as follows:

Let , steps of Quick sort are as follows:

Let us pick the middle element as the pivot every time. Then . Since

the last is smaller than 2, we swap with . Then , is split into two lists

and . Repeat above procedure, we have , which,

after partition would be . The next partition would split it into two already sorted arrays so we would save that step. And is already sorted. Thus, the sorted array would be . However, originally in is at the left of and , but now the order changed. Therefore, quick sort is not stable.

Steps of Heap sort are as follows:

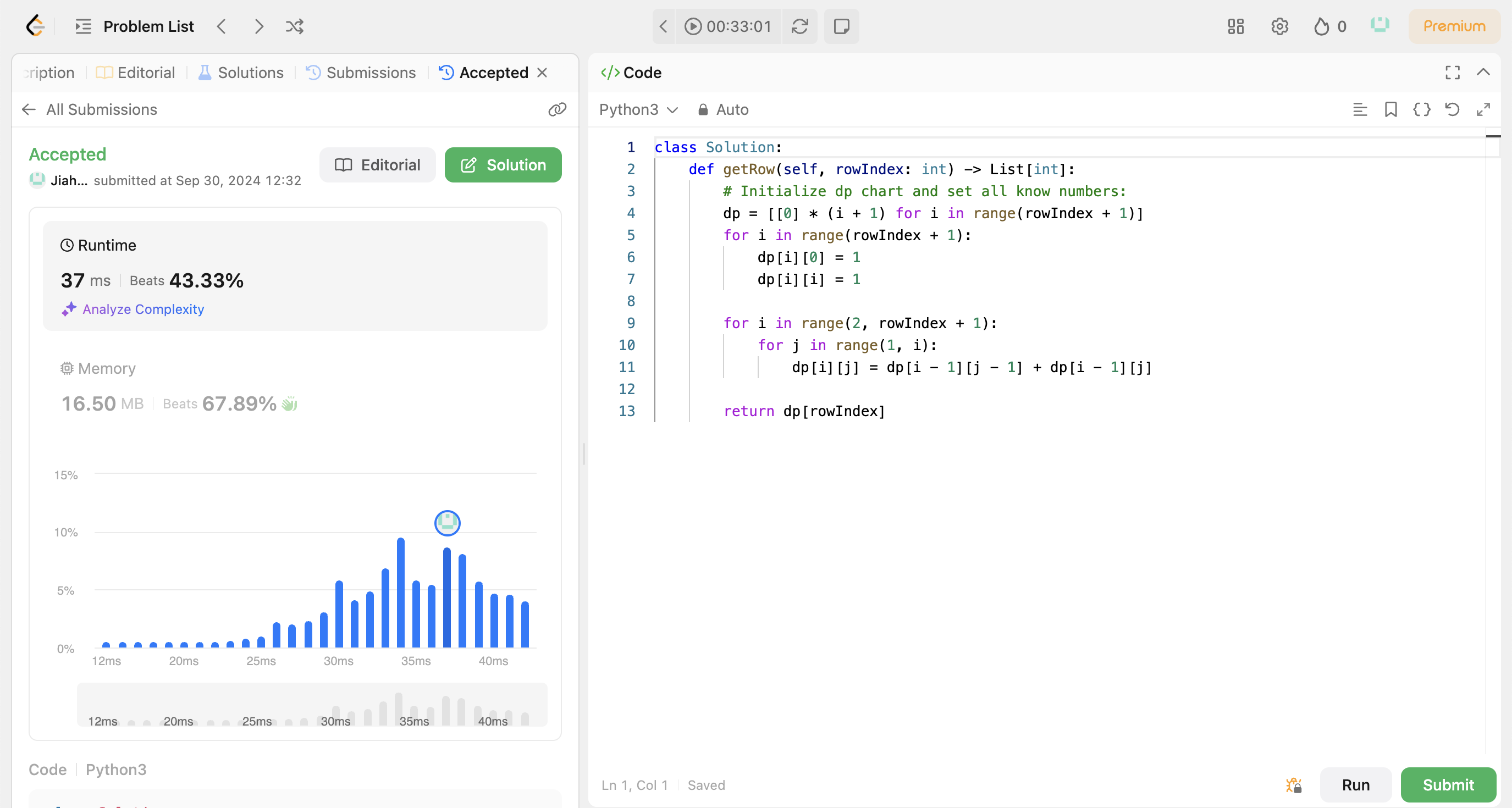
Firstly, the procedure of heapifying as a min heap is as follows:

Therefore, after Heap sort, , which also messed with the original order. Thus, Heap sort is not stable either.

Problem #4

(a)

119. Pascal’s Triangle II (Easy), I used 33:01 minutes to solve it.



894. All Possible Full Binary Trees (Medium), I used 33:50 minutes to solve it.

A screenshot of a computer

Description automatically generated

(b)

119. Pascal’s Triangle II (Easy) is the very first Dynamic Programming problem I solved. I have practiced several other types of problems before, such as maps/dictionaries, linked lists, stack/queue/heaps, and DFS/BFS. But I haven’t got a chance to dive deep into DP until this problem. Although it was marked as easy, I spent nearly half an hour to understand and try. Speaking of specific ways I tried, firstly I was thinking about calculating the -th array directly. That is, since each row of Pascal’s Triangle can be represented using the sequence of combinations, the -th array of Pascal’s Triangle should be . However, implementing the calculation of binomial coefficient seems difficult. So secondly, I tried to visualize Pascal’s Triangle as what the GIF indicates: the -th element in the -th array is the sum of -th and -th element in the -th array. At the time I started to implement that logic, I noticed that I need to initialize the 2D array, namely the DP chart first. Then I realized that I did not remember how to do so in Python. After using to debug and checking if the DP chart is initialized correctly, I reached my solution.

Before this problem, I assume DP problems are all difficult to tackle. But thanks to this problem, I felt proud of myself because I tried hard and figured out one working solution on my own. I will no longer consider DP problems as unsolvable in the future, which is the most significate insight I gained. Other than that, my lack of familiarity with nested lists and collections library in Python also made it hard for me to solve the problem at first. I will keep on learning related syntaxes.