HW2, Session 19
Jiahuan He

Problem #1

(a)

Proof by induction.

∵ The condition in line 3 is $p + 1 < r \Rightarrow r - p > 1$,

∴ When the index of the last element is at least 2 greater than the first element, namely at least there are 3 elements in the array would we call the function recursively.

∴ The base case is when the array contains only 1 or 2 elements.

Base case 1, when $n = 2$:

∵ The subarray $A[1{:}\,n]$ only contains 1 element,

∴ The array $A[1{:}\,n]$ is sorted.

Base case 2, when $n = 3$:

∵ The algorithm checks if $A[1] > A[n]$ and swaps them if necessary.

∴ After above step, the array $A[1{:}\,n]$ is sorted.

Assume New-Sort correctly sorts $A[p{:}\,n - k]$ and $A[p + k{:}\,n]$ where $k = \left\lfloor\frac{n-p+1}{3}\right\rfloor$, namely one third of the array's length, and p is the index of the first element. Now we only need to prove this algorithm correctly sorts $A[p{:}\,n]$ as well.

∵ $A[p + k{:}\,n]$ is sorted,

∴ $A[p + 2k{:}\,n]$ is sorted too.

∵ $A[p{:}\,n - k]$ is sorted, $n - k = p + 2k$,

∴ $A[p{:}\,p + 2k]$ is sorted too.

∵ $A[p + 2k{:}\,n]$ is sorted, $A[p{:}\,p + 2k]$ is sorted,

∴ $A[p{:}\,n]$ is sorted.

Combined with base cases, we have proved that New-Sort correctly sorts the input array $A[1{:}\,n]$. ■

(b)

The recurrence for the worst-case running time is $T(n) = 3T\left(\frac{2}{3}n\right) + O(1)$ where $n > 2$, and $T(n) = \Theta(n^{\log_{\frac{3}{2}} 3}) \approx \Theta(n^{2.71})$.

From the pseudocode we know that the New-Sort divides the array A into three parts and recursively sorts the first two-thirds, the last two-thirds, and then the first two-thirds again. This gives us the following recurrence relation:

$T(n) = 3T\left(\frac{2}{3}n\right) + O(1)$, where $O(1)$ stands for the efforts we spend on initializing variables, etc. According to the Master Theorem, we know that the watershed function is $n^{\log_b a} = n^{\log_{\frac{3}{2}} 3}$, and $f(n) = O(1)$.

∵ $n^{\log_{\frac{3}{2}} 3}$ grows asymptotically and polynomially faster than $O(1)$,

∴ Case 1 applies in this case.

∴ $T(n) = \Theta(n^{\log_{\frac{3}{2}} 3}) \approx \Theta(n^{2.71})$.

(c)

No, the students do not deserve an A in the course, since this New-Sort is slower than insertion sort, merge sort, heapsort and quicksort.

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	$\mathrm{O}(n^2)$	$\mathrm{O}(n\log n)$	$\mathrm{O}(n\log n)$	$\mathrm{O}(n^2)$

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 $\Omega(n \log n)$ 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 h with l reachable leaves corresponding to such comparison sort on 8 elements.

$\because$ Each of the $8!$ permutations appear as one or more leaves,
$\therefore 8! \leq l$.
$\because$ A binary tree of height h has no more than 2^h leaves,
$\therefore 8! \leq l \leq 2^h$,
$\therefore \log 8! \leq \log l \leq \log 2^h$,
$\therefore \log 8! \leq h$.
$\therefore 15.299 \leq h$,
$\therefore$ The height h will be at least 16,
$\therefore$ 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 $\left\lceil 3\frac{n}{2} \right\rceil - 2$ comparisons are necessary to find both the maximum and minimum of n 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 $\frac{n}{2}$ pairs when the length is even, and $\frac{n}{2}$ 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 $3 \times \frac{n-2}{2} = \frac{3}{2}n - 3$ comparisons for all the pairs, which would be $\frac{3}{2}n - 2 = \left\lceil \frac{3}{2}n \right\rceil - 2$ 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 $2 + \frac{3}{2}(n-1) - 2 = \frac{3}{2}n - \frac{3}{2} = \left(\left\lceil \frac{3}{2}n \right\rceil - \frac{1}{2}\right) - \frac{3}{2} = \left\lceil \frac{3}{2}n \right\rceil - 2$.

∴ At least $\left\lceil 3\frac{n}{2} \right\rceil - 2$ comparisons are necessary to find both the maximum and minimum of n numbers in the worst case.

∵ The number of the students is 8,

∴ There should be at least $\left\lceil 3\frac{n}{2} \right\rceil - 2 = 12 - 2 = 10$ 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 $n + \lceil \log n \rceil - 2$ comparisons are necessary to find the second maximum of n 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 $\frac{n}{2} + \frac{n}{4} + \cdots + 1$.

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

∴ There must be $n - 1$ comparisons in total.

∴ This algorithm will take $n - 1$ comparisons to find the largest of n elements.

∵ There would be $\lceil \log n \rceil$ 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,

$\therefore$ Except for the last comparison where the second largest lost the largest, there should be $\lceil \log n \rceil - 1$ comparisons to determine the second largest element.

$\therefore$ In total, it takes $n - 1 + \lceil \log n \rceil - 1 = n + \lceil \log n \rceil - 2$ comparisons to find the second largest of n elements in the worst case.

$\therefore$ It takes at least $n + \lceil \log n \rceil - 2 = 8 + \lceil \log 8 \rceil - 2 = 9$ 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)

$\because A = [4, 5, 0, 1, 3, 4, 3, 4, 3, 0, 3]$, C counts the occurrences of elements in A,

$\therefore$ Let C be an array of length $\max(A) + 1 = 5 + 1 = 6$,

0 1 2 3 4 5

$\therefore C = [2, 1, 0, 4, 3, 1]$.

After calculating the running time, C would be:

$C = [2, 1 + 2, 1 + 2, 1 + 2 + 4, 1 + 2 + 4 + 3, 1 + 2 + 4 + 3 + 1]$

0 1 2 3 4 5

$C = [2, 3, 3, 7, 10, 11]$

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

$A[10] = 3 \Rightarrow B[C[3] - 1](since\ B\ is\ 0\ index) = B[7 - 1] = B[6] = 3$

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

$\therefore B = [0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0]$, $C = [2, 3, 3, 6, 10, 11]$

$A[9] = 0 \Rightarrow B[C[0] - 1] = B[2 - 1] = B[1] = 0$

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

$\therefore B = [0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0]$, $C = [1, 3, 3, 6, 10, 11]$

$A[8] = 3 \Rightarrow B[C[3] - 1] = B[5] = 3$

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

$\therefore B = [0, 0, 0, 0, 0, 3, 3, 0, 0, 0, 0]$, $C = [1, 3, 3, 5, 10, 11]$

$A[7] = 4 \Rightarrow B[C[4] - 1] = B[9] = 4$

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

$\therefore B = [0, 0, 0, 0, 0, 3, 3, 0, 0, 4, 0]$, $C = [1, 3, 3, 5, 9, 11]$

$A[6] = 3 \Rightarrow B[C[3] - 1] = B[4] = 3$

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

$\therefore B = [0, 0, 0, 0, 3, 3, 3, 0, 0, 4, 0]$, $C = [1, 3, 3, 4, 9, 11]$

$A[5] = 4 \Rightarrow B[C[4] - 1] = B[8] = 4$

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

$\therefore B = [0, 0, 0, 0, 3, 3, 3, 0, 4, 4, 0]$, $C = [1, 3, 3, 4, 8, 11]$

$A[4] = 3 \Rightarrow B[C[3] - 1] = B[3] = 3$

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

$\therefore B = [0, 0, 0, 3, 3, 3, 3, 0, 4, 4, 0]$, $C = [1, 3, 3, 3, 8, 11]$

$A[3] = 1 \Rightarrow B[C[1] - 1] = B[2] = 1$

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

$\therefore B = [0, 0, 1, 3, 3, 3, 3, 0, 4, 4, 0]$, $C = [1, 2, 3, 3, 8, 11]$

$A[2] = 0 \Rightarrow B[C[0] - 1] = B[0] = 0$

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

$\therefore B = [0, 0, 1, 3, 3, 3, 3, 0, 4, 4, 0]$, $C = [0, 2, 3, 3, 8, 11]$

$A[1] = 5 \Rightarrow B[C[5] - 1] = B[10] = 5$

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

$\therefore B = [0, 0, 1, 3, 3, 3, 3, 0, 4, 4, 5]$, $C = [0, 2, 3, 3, 8, 10]$

$A[0] = 4 \Rightarrow B[C[4] - 1] = B[7] = 4$

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

$\therefore B = [0, 0, 1, 3, 3, 3, 3, 4, 4, 4, 5]$, $C = [0, 2, 3, 3, 7, 10]$

$\therefore$ The out put $B = [0, 0, 1, 3, 3, 3, 3, 4, 4, 4, 5]$.

(b)

Since C is the array of the occurrences of elements in A, we know that the value in C 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 $A = [1, 0, 2, 2, 2, 1]$, from (a) we know $C = [1, 3, 6]$, 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 $B = [0, 1, 1, 2, 2, 2]$, which provides the stability. In controversy, if we start put elements from A onwards, this algorithm will become unstable.

(c)

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

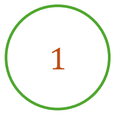
Let $A = [1, 0, 2, 2, 2, 1]$, steps of Quick sort are as follows:

Let us pick the middle element as the pivot every time. Then $A = [1, 0, \overset{\downarrow}{2}, 2, 2, 1]$. Since the last 1 is smaller than 2, we swap 2 with 1. Then $A = [1, 0, \overset{\downarrow}{1}, 2, 2, 2]$, A is split into two lists $A_{left} = [1, 0, 1]$ and $A_{right}[2, 2, 2]$. Repeat above procedure, we have $A_{left} = [1, \overset{\downarrow}{0}, 1]$, which, after partition would be $A_{left} = [0, \overset{\downarrow}{1}, 1]$. The next partition would split it into two already sorted arrays so we would save that step. And $A_{right}[2, 2, 2]$ is already sorted. Thus, the sorted array would be $A = [0, 1, 1, 2, 2, 2]$. However, originally in A 2 is at the left of 2 and 2, but now the order changed. Therefore, quick sort is not stable.

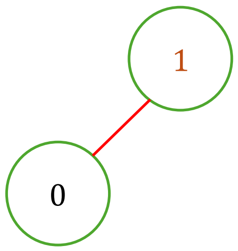
Steps of Heap sort are as follows:

Firstly, the procedure of heapifying $A = [1, 0, 2, 2, 2, 1]$ as a min heap is as follows:

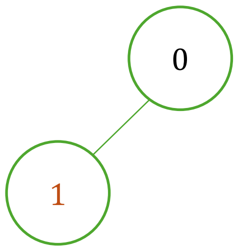
1.



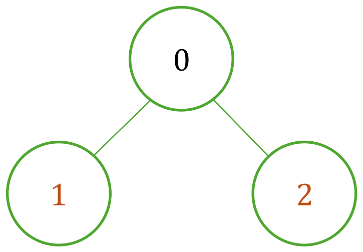
2.



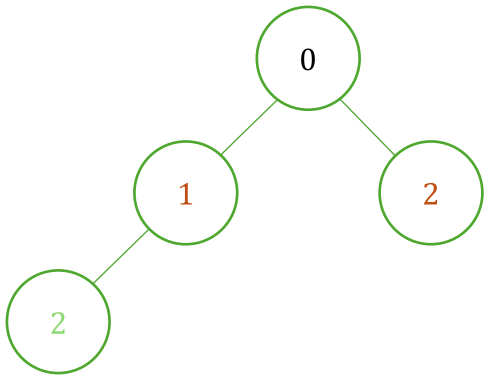
3.



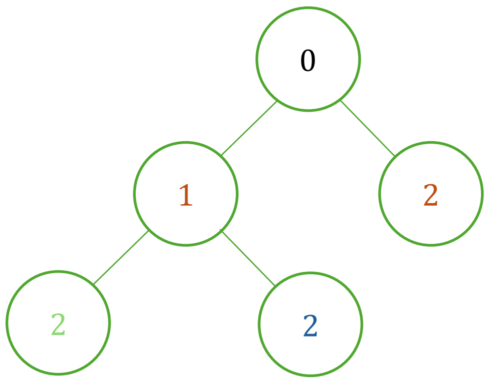
4.



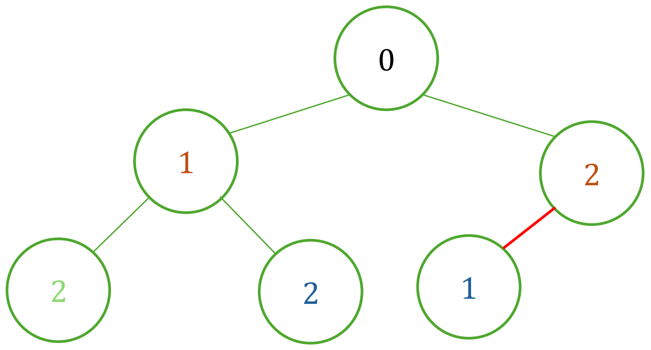
5.



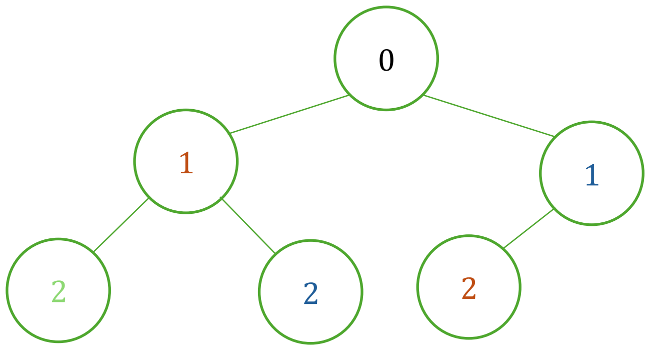
6.



7.



8.

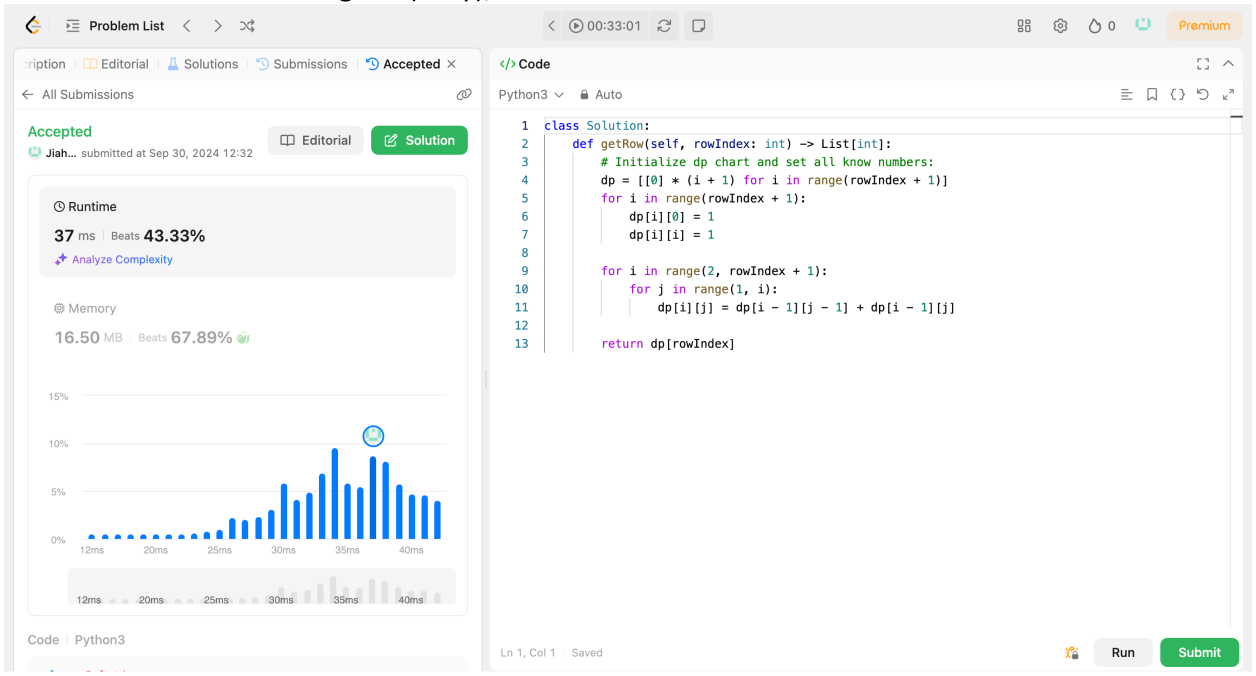


Therefore, after Heap sort, $A = [0, 1, 1, 2, 2, 2]$, which also messed with the original order. Thus, Heap sort is not stable either.

Problem #4

(a)

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

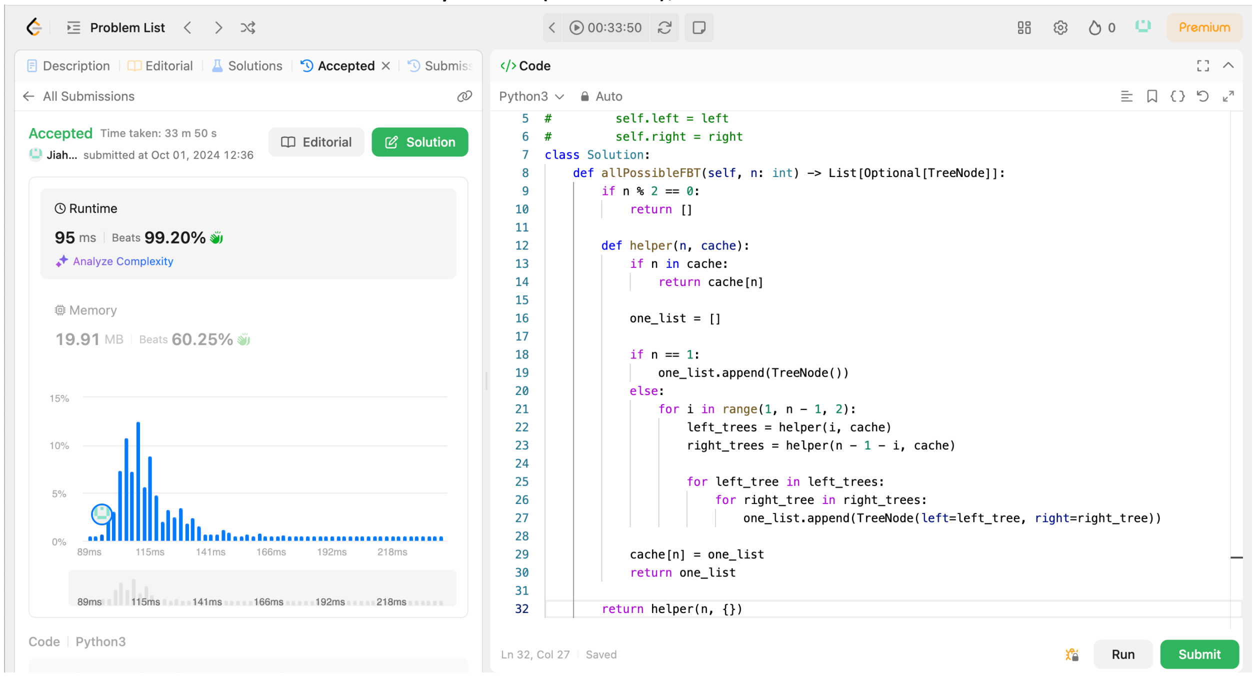


```python
class Solution:
    def getRow(self, rowIndex: int) -> List[int]:
        # Initialize dp chart and set all know numbers:
        dp = [[0] * (i + 1) for i in range(rowIndex + 1)]
        for i in range(rowIndex + 1):
            dp[i][0] = 1
            dp[i][i] = 1

        for i in range(2, rowIndex + 1):
            for j in range(1, i):
                dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]

        return dp[rowIndex]
```

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



```python
#         self.left = left
#         self.right = right
class Solution:
    def allPossibleFBT(self, n: int) -> List[Optional[TreeNode]]:
        if n % 2 == 0:
            return []

        def helper(n, cache):
            if n in cache:
                return cache[n]

            one_list = []

            if n == 1:
                one_list.append(TreeNode())
            else:
                for i in range(1, n - 1, 2):
                    left_trees = helper(i, cache)
                    right_trees = helper(n - 1 - i, cache)

                    for left_tree in left_trees:
                        for right_tree in right_trees:
                            one_list.append(TreeNode(left=left_tree, right=right_tree))

            cache[n] = one_list
            return one_list

        return helper(n, {})
```

(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 $rowIndex$-th array directly. That is, since each row of Pascal's Triangle can be represented using the sequence of combinations, the k-th array of Pascal's Triangle should be $C(k,0), C(k,1), \dots, C(k,k)$. However, implementing the calculation of binomial coefficient seems difficult. So secondly, I tried to visualize Pascal's Triangle as what the GIF indicates: the m-th element in the n-th array is the sum of $(m-1)$-th and m-th element in the $(n-1)$-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 $print()$ 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.