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 and pigeonhole principle.

Let us use to represent a set of 10 distinct integers and take one element in , where the longest increasing subsequence and the longest decreasing subsequence both starts at . Let us assume the length of and is at most 3.

If both and contains at most 4 numbers, then the total number of elements is at most 9 (, numbers in and ), which is impossible,

Either or must have at least 5 elements.

Let us assume

1. If any of these 5 elements form an increasing subsequence of length 3, then together with , we have an increasing subsequence of length 4.
2. If not, then these 5 elements must form a decreasing subsequence of length 4 (or more).

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)

Fractional-Knapsack-Problem (item\_list, max\_weight):

sort the item\_list by the value per weight in descending order

initialize a result\_list and total\_value

for each\_item in item\_list:

if max\_weight == 0:

break

if max\_weight >= each\_item.weight:

result\_list.append(each\_item)

total\_value += each\_item.value

max\_weight -= each\_item.weight

else:

result\_list.append((max\_weight / each\_item.weight, each\_item))

total\_value += (max\_weight / each\_item) \* each\_item.value

return result\_list, total\_value

The logic above is, if the weight of a new item is not exceeding the maximum weight, then put the whole of it into the backpack. Otherwise just put the part of that item into the pack.

(b)

The time complexity of above algorithm is and the space complexity is .

For time complexity, since I firstly sort the input list by the value per weight, it would at least cost . Then if the backpack has enough capacity, I will loop every item in the list, which requires time. So overall the time complexity is .

For space complexity, the sorting step requires space. Similarly, if the total weight permits the result list could contain every item with the original shape, which will lead to space. Having a total value variable typically requires space, therefore, the space complexity of this algorithm is .

(c)

A greedy approach works for the fractional knapsack but not for the 0-1 knapsack, due to following reasons:

1. Key differences between the two problems:

1. The fractional knapsack can take part of the item, while 0-1 knapsack can either take one item, or not take it. Therefore, the fractional knapsack will always get filled full, but never will the 0-1 knapsack.
2. The greedy approach works for the fractional knapsack situation, but not the 0-1 knapsack situation.

2. Counterexample:

Let us refer to textbook’s example. Suppose there are 3 items as follows and the capacity of the knapsack is 50:

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Value | Weight | Value Per Weight |
| 1 | 60 | 10 | 6 |
| 2 | 100 | 20 | 5 |
| 3 | 120 | 30 | 4 |

And with greedy approach we always take the item with highest value per weight, which in this case is item 1. After adding item 1 the capacity is changed to 40, which only allows either item 2 or 3, not both. The greedy approach would pick item 2, which has the second highest value per weight ratio. Then the total value of the knapsack is 160. However, the optimal way to fill the knapsack is to put item 2 and 3, which will make the total value of the knapsack 220. Thus, we showed that the greedy approach does not lead to the optimal solution using above example.

3. Proof on why greedy approach is optimal for the fractional knapsack problem:

Let be a list of items. The greedy approach will sort by the value per weight of each element first, and then pick the item with the highest ratio. Suppose after sorting the items in are , and there is one way to fill the knapsack with different item selection.

The greedy approach will fill the knapsack with , where is the number of the last item.

Any other item number greater than has a lower value per weight than , there is no way a lower value per weight item will bring higher value than the greedy approach does.

The assumption is a contradiction.

 The greedy approach must be optimal in the fractional knapsack problem.

Problem #3

(a)

The overall idea is to add all awake cats into a queue, then pop one cat every time and check if the target cat is next to the current cat. If not, then set that cat as awake with a waking time and put the cat to the queue, until we found the target cat. My pseudocode is as follows:

When-To-Wake-Up (matrix, x, y):

# If cat at (x, y) is not sleeping, then it needs 0 second to get it up.

if matrix[x][y] != -1:

return 0

# Use a set to store awake cats, so we don’t need to update the matrix

m, n = len(matrix), len(matrix[0])

directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

q = collections.deque([])

awake = set()

# Put all awake cats to q with awaking time 0 second, and to awake set.

for i in range(m):

for j in range(n):

if matrix[i][j] == 1:

q.append((i, j, 0))

awake.add((i, j))

# While there are awaking cats,

while q:

# pop the first cat in q,

current\_i, current\_j, time = q.popleft()

# loop the four directions of that cat,

for direction\_i, direction\_j in directions:

new\_i = current\_i + direction\_i

new\_j = current\_j + direction\_j

# if the adjacent cat is within matrix’s range, and is not a visited awake   
 cat, and is sleeping:

if (new\_i in range (0, m) and new\_j in range(0, n)

and (new\_i, new\_j) not in awake

and matrix[new\_i][new\_j] == -1):

# If that cat happens to be the target, return time + 1;

if new\_i == x and new\_j == j:

return time + 1

# if that cat is not the target, then append this cat to the q since it   
 is awake now, as well as the awake set.

q.append((new\_i, new\_j, time + 1))

awake.add((new\_i, new\_j))

# If there is no time + 1 returned above, then the cat cannot be reached.

return -1

(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)

994. Rotting Oranges (Medium), I used 58:49 minutes to solve it. Since the screenshot doesn’t include all my code, I also pasted the code below.

A screenshot of a computer

Description automatically generated

class Solution:

def orangesRotting(self, grid: List[List[int]]) -> int:

# Use m, n to represent length

m = len(grid)

n = len(grid[0])

# Find all rotten oranges

fresh\_orange = 0

q = collections.deque([])

for i in range(m):

for j in range(n):

if grid[i][j] == 2:

q.append((i, j))

elif grid[i][j] == 1:

fresh\_orange += 1

# If no fresh oranges:

if fresh\_orange == 0:

return 0

# Set 4 directions

directions = [[-1, 0], [0, -1], [1, 0], [0, 1]]

time = 0

# Use BFS to track all cells

while q and fresh\_orange > 0:

time += 1

# Check one rotten orange and its adjeceant ones:

for \_ in range(len(q)):

current\_i, current\_j = q.popleft()

for direc\_i, direc\_j in directions:

next\_i = current\_i + direc\_i

next\_j = current\_j + direc\_j

if next\_i in range(0, m) and next\_j in range(0, n) and grid[next\_i][next\_j] == 1:

fresh\_orange -= 1

grid[next\_i][next\_j] = 2

q.append((next\_i, next\_j))

return time if fresh\_orange == 0 else -1

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.