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, take the first element in , and put all numbers greater than to a subsequence and those lower than to a subsequence .

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

By pigeonhole principle, either or must have at least 5 elements.

Let us assume has 5 elements.

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).

Similarly, if we assume has 5 elements, then:

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

To prove above statements, let us assume we have 5 elements and none of them form an increasing/decreasing subsequence of length 3. We will assign each element a pair of numbers , where is the length of the longest increasing subsequence ending at this element, and is the length of the longest decreasing subsequence ending at this element.

There is no increasing or decreasing subsequence of length 3 with 5 elements,

and .

The 5 elements must be assigned one of these pairs: .

By pigeonhole principles, we know that at least 2 elements must be assigned the same pair.

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

.

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.

Our supposition must be false, and among any 5 distinct elements, there must be an increasing or decreasing subsequence of length 3.

In either case, we are guaranteed to have either an increasing or decreasing subsequence of length 4 (or more).

(c)

The length of the longest common subsequence of and should be 3, and the proof is as follows:

Let , and be the sorted version of . Then the longest increasing subsequence must satisfy that , the longest common subsequence must appear in both and in the same order.

Any increasing subsequence of will also be a subsequence of since is sorted in ascending order,

is a common subsequence of and .

Any common subsequence of and will also be an increasing subsequence, since is sorted in ascending order,

is an increasing subsequence of .

The length of and is equal.

(d)

To tackle this question, I have referred to [Geeksforgeeks](https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/).

The overall idea of this algorithm is: for each number in the given array, we would perform the following steps:

1. If the number is greater than the last element of the result array, we would append the number to the end of the list. This indicates that we have found a greater element in the array, which lead to a longer subsequence.
2. Otherwise, we would use a binary search on the result array, to find the smallest element that is greater than or equal to the current number. Then, we replace that element with the current number. Although this step will not form a correct increasing subsequence, this will keep the result array sorted and ensures that we have the potential for a longer subsequence in the future.

For example, assume the input array is . Through above steps, we would firstly have the result array as , then since should be placed at where is, we replace with , so that the result array is . If the input array were , then even if is not a correct increasing subsequence, the length of it is the same as , so the result is correct. Similarly, if the input array is , then our result array should be , an incorrect yet the same length with the correct subsequence. Meanwhile, the next step assures that we update the result array with , the length of which is 4 as the correct answer. With that said, the code for such algorithm is as follows:

# HW3 Problem 1: Longest Increasing Subsequence(d)

def length\_of\_longest\_increasing\_subsequence(arr: list) -> int:

'''

Given an array arr of n elements,

find the length of the longest increasing subsequence with O(n \* logn) time.

'''

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

while left <= right:

mid = left + (right - left) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return left

# Initialize an empty list to store the LIS candidates

lis = []

for num in arr:

# Find the position to replace or append the current number

pos = binary\_search(lis, num)

# If pos is equal to the length of lis, append the number

if pos == len(lis):

lis.append(num)

else:

# Otherwise, replace the element at the found position

lis[pos] = num

# The length of lis is the length of the longest increasing subsequence

return len(lis)

Proof of correctness:

As mentioned above, this algorithm always contains the smallest possible tail values for increasing subsequences of different lengths. This ensures that we can extend the subsequences optimally, while in the meantime main the correct length.

Time complexity:

We iterate through all elements of the input array and use the binary search, which takes time, to determine each element’s position in the result array. So apparently the overall time complexity is .

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:

Wake-Up-One (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)

To wake all cats up, the idea is similar to (a). That is, every time when an awake cat awakes a sleeping cat, we compare the current maximum time with that sleeping cat’s time and update the maximum time. The running time of this algorithm is , so does the space complexity, and the pseudocode is as follows:

Wake-Up-All (matrix):

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

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

queue = collections.deque()

awake = set()

max\_time = 0

# Initialize the queue with all awake cats

for i in range(m):

for j in range(n):

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

queue.append((i, j, 0)) # (row, col, time)

awake.add((i, j))

# Perform BFS

while queue:

current\_row, current\_col, time = queue.popleft()

max\_time = max(max\_time, time)

for direction in directions:

new\_row = current\_row + direction[0]

new\_col = current\_col + direction[1]

if (new\_row in range(0, m) and new\_col in range(0, n)

and (new\_row, new\_col) not in awake

and matrix[new\_row][new\_col] == -1):

queue.append((new\_row, new\_col, time + 1))

awake.add((new\_row, new\_col))

# Check if all the cats are awake

for row in matrix:

if -1 in row:

# There is at least one cat that cannot be woken up

return -1

return max\_time

The proof of running time as is as follows:

Firstly, as for the initialization of the queue, it takes in the worst case. Secondly, by performing BFS, each cell of the matrix is approached once. Thus, the running time of this step should be as well. Finally, it takes time to check the entire matrix if there is still a sleeping cat. Therefore, overall, the running time of this algorithm is .

(c)

The idea to find the target cat is the same as (a) and (b). On the top of that, we need to keep track of parent cat as the cat who awakes the current cat, then return the reversed list. My pseudocode is as follows:

Wake-Up-All (matrix, target\_x, target\_y):

if matrix[target\_x][target\_y] != -1:

# If the target cat is not asleep, then return an empty list.

return []

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

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

queue = collections.deque()

awake = set()

path = {}

for i in range(m):

for j in range(n):

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

queue.append((i, j))

awake.add((i, j))

path[(i, j)] = None

while queue:

current\_x, current\_y = queue.popleft()

for direction in directions:

new\_x = current\_x + direction[0]

new\_y = current\_y + direction[1]

if (new\_x in range(0, m) and new\_y in range(0, n)

and (new\_x, new\_y) not in awake

and matrix[new\_x][new\_y] == -1):

path[(new\_x, new\_y)] = (current\_x, current\_y)

awake.add((new\_x, new\_y))

queue.append((new\_x, new\_y))

if new\_x = target\_x and new\_y = target\_y:

return Construct-Path(path, (new\_x, new\_y))

# If no path is returned above, it means the target cat cannot be reached.

return []

Construct-Path (path: list, target: tuple):

result = []

current = target

while current is not None:

result.append(current)

current = path[current]

return result[::-1]

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.