Question #1

(a)

A screenshot of a computer

Description automatically generated

A screenshot of a email

Description automatically generated

Question #2

(a)

The Union Find/ Disjoint Set Algorithm. The main idea is to split nodes into different unions, and check if some nodes are connected or not. If and is complementary to each other, then any node in the union must not relate to any node in . The pseudocode is as follows. Here, n represents the length of elements to consider.

Can-Satisfy-Constraints (n, m1, m2):

# Initialize Union-Find data structures, by:

# Firstly, setting the parent of every element to itself.

parent = [0, 1, …, n - 1]

# Then initialize every element’s rank as 0.

rank = [0, 0, …, 0]

# Define helper functions for Union-Find

# The find function is to find the top-parent of one element.

Find(x):

# If the parent of x is not x itself,

if parent[x] != x:

# then set x’s parent as recursively found parent, the top-parent.

parent[x] = find(parent[x])

# If we found the element whose parent is itself, it must be the top-  
# parent, so return it. Then every child of x will have their parent as x.

return parent[x]

# The union function is to link two elements based on their rank.

Union (u, v):

# Find two elements’ parents.

root\_u = find(u)

root\_v = find(v)

# If their parents are not the same:

if root\_u != root\_v:

# Case 1, u’s parent rank is greater than v’s.

if rank[root\_u] > rank[root\_v]:

# In that case, we put v as u’s child.

parent[root\_v] = root\_u

# Case 2, v’s parent rank is greater than u’s.

elif rank[root\_u] < rank[root\_v]:

# Then we do vice versa.

parent[root\_u] = root\_v

# Case 3, u’s parent is the same rank as v’s.

else:

# Then it’s the same. We just pick u as the parent and   
# update its rank.

parent[root\_v] = root\_u

rank[root\_u] += 1

# Process equality constraints:

for (xi, xj) in m1:

# Link all elements existed in equalities.

union(xi, xj)

# Check inequality constraints

for (xi, xj) in m2:

# Check inequalities. If two elements have the same parent, it means   
# they must be in the same union. Then it is impossible to satisfy both   
# equalities and inequalities.

if find(xi) == find(xj):

return False

return True

(b)

For answer this question, I referred to a [YouTube video](https://www.youtube.com/watch?v=aBxjDBC4M1U).

The total running time of above algorithm is .

Firstly, initializing parent and rank arrays takes time.

Secondly, the Fund function takes time. What we did in Fund function is called path compression, which compresses the graph of a long linear node link into a dense 2-layer graph. So theoretically speaking we would not encounter a long linear node link graph. In practice, can be considered nearly as constant time.

Thirdly, the Union function takes time as well, since this function calls Fund function, and other operations are constant time.

Last, iterating with Union function will take time and iterating with Find function to check if there is any contradiction with will take time, since we need to iterate all conditions in them.

So, combining all parts, the total running time of the algorithm is:. Since is considered nearly constant, we could say the running time of this algorithm is .

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

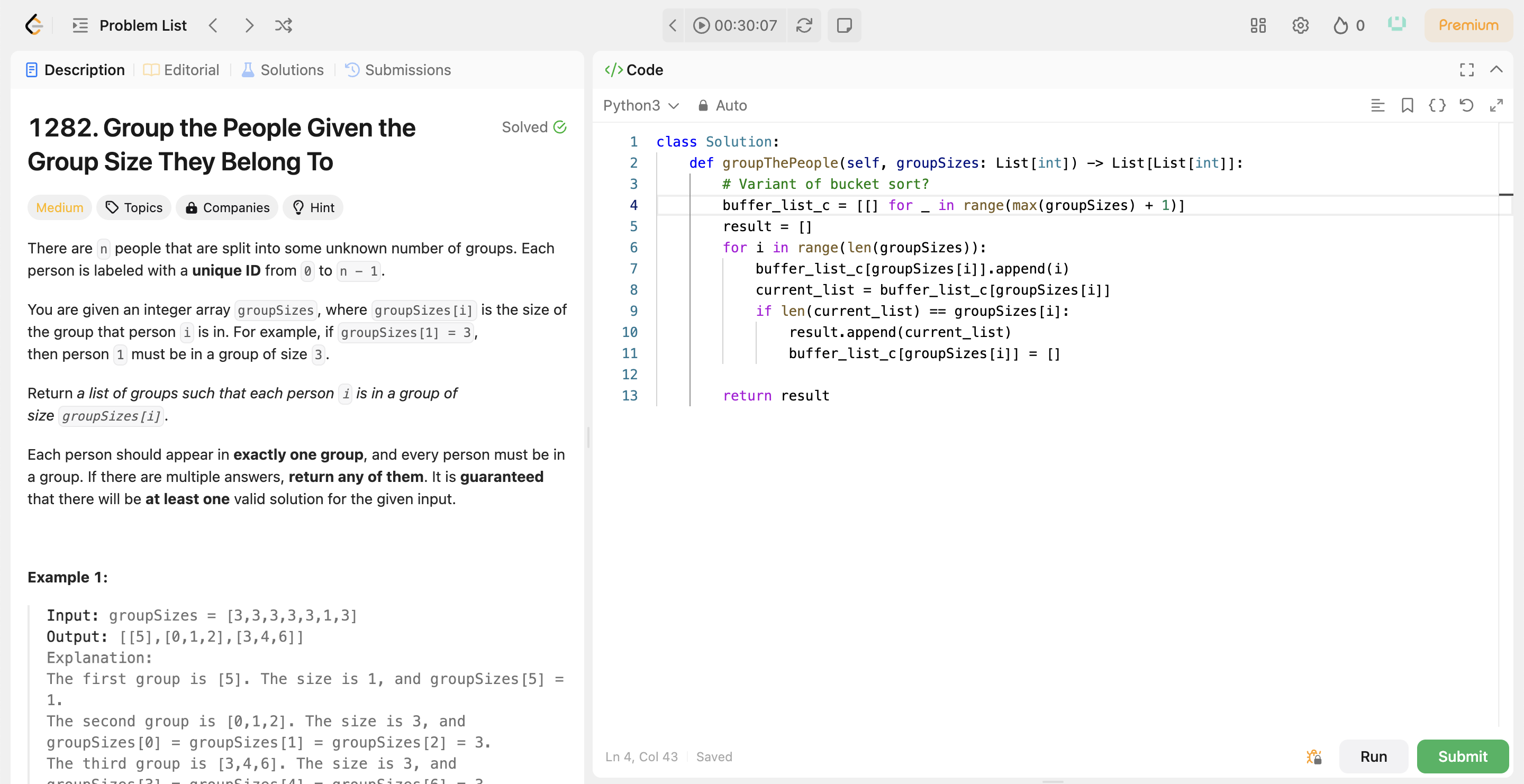
(a)

[**2160. Minimum Sum of Four Digit Number After Splitting Digits**](https://leetcode.com/problems/minimum-sum-of-four-digit-number-after-splitting-digits/) (Easy), I used 34:44 minutes to solve it.

A screenshot of a computer

Description automatically generated

[**1282. Group the People Given the Group Size They Belong To**](https://leetcode.com/problems/group-the-people-given-the-group-size-they-belong-to/) (Medium), I used 30:07 minutes to solve it.



(b)

I tried another problem before submitting these two, which is [**1689. Partitioning Into Minimum Number Of Deci-Binary Numbers**](https://leetcode.com/problems/partitioning-into-minimum-number-of-deci-binary-numbers/). The solution to this problem is incredibly easy: just return the length of the maximum number in the string. With hints, this problem only took me around 5 minutes, so I didn’t include it in Problem 4. However, this problem has provided me with a general taste of greedy algorithm: mainly focus on the maximum or the minimum, ignore noises. I am inspired by this principle and would like to pick [**1282. Group the People Given the Group Size They Belong To**](https://leetcode.com/problems/group-the-people-given-the-group-size-they-belong-to/) and share my thoughts.

At the first glance of this problem, I thought this might be a variant of bucket sort, since it requires “sorting” to some degree. So, I drew a similar sketch to bucket sort, using A, C to represent the input. For test case 1, since the input is , the maximum of which is 3, then I should have a list of 4 elements. Since this problem needs the index of each number, I decided to have a nested list. While iterating all numbers in the input, I will append the index of each number in the corresponding result list. That is, the 4th list of the result list grows into after iterating the first 3 ‘3’s, and I will check if the length of the current list is the same as the number itself. If it is, then append that list to the result list. Similarly, I will find because of ‘1’, and finally the .

The struggle I faced was the time complexity. The bucket sort has a time complexity of , which in my idea should be an optimal solution. Yet my solution only beats less than 15% of solutions. In conclusion, I need to research for faster algorithm in terms of greedy solutions.