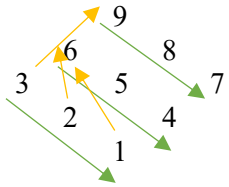
Problem #1

(a)

The sequence I found is $S = [3, 2, 1, 6, 5, 4, 9, 8, 7]$, with following steps:

Firstly, I chose nine distinct integers from 1 to 9, and group them into 3 sets: $S_1 = \{1, 2, 3\}, S_2 = \{4, 5, 6\}, and\ S_3 = \{7, 8, 9\}$.



Secondly, I would arrange S_1, S_2, S_3 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 S_1, at the beginning of the final sequence, then every rest element in S_1 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 S to represent a set of 10 distinct integers, take the first element a_1 in S, and put all numbers greater than a_1 to a subsequence In and those lower than a_1 to a subsequence De.

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

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

Let us assume In has 5 elements.

1. If any of these 5 elements form an increasing subsequence of length 3, then together with a_i, 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 De has 5 elements, then:

1. If any of these 5 elements form a decreasing subsequence of length 3, then together with a_i, 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 (x, y), where x is the length of the longest increasing subsequence ending at this element, and y 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,

∴ $x \leq 2$ and $y \leq 2$.

∴ The 5 elements must be assigned one of these pairs: $(1,1), (1,2), (2,1), (2,2)$.

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

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

∴ $x(i) = x(j), y(i) = y(j)$.

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

∵ There are $y(i) - 1$ decreasing numbers at the right of i, and $i > j$,

∴ j is one of the decreasing numbers at the right of i,

∴ $y(j) \leq y(i) - 1$. This is a contradiction since we assume $y(i) = y(j)$.

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

∵ There are $x(j) - 1$ increasing numbers at the right of j, and $i > j$,

∴ i is one of the increasing numbers at the right of j,

∴ $x(j) - 1 \geq x(i)$. This is a contradiction since we assume $x(j) = x(i)$.

∴ It is impossible for two distinct numbers in the sequence to have the identical $(x(k), y(k))$ 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 S and S^* should be 3, and the proof is as follows:

Let $S = [a_1, a_2, \dots, a_n]$, and $S^* = [a_1^*, a_2^*, \dots, a_n^*]$ be the sorted version of S. Then the longest increasing subsequence $S_{increasing} = [a_{i_1}, a_{i_2}, \dots, a_{i_n}]$ must satisfy that $a_{i_1} < a_{i_2} < \cdots < a_{i_n}$, the longest common subsequence $S_{common} = [a_{j_1}, a_{j_2}, \dots, a_{j_m}]$ must appear in both S and S^* in the same order.

∵ Any increasing subsequence of S will also be a subsequence of S^* since S^* is sorted in ascending order,

∴ $S_{increasing}$ is a common subsequence of S and S^*.

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

∴ S_{common} is an increasing subsequence of S.

∴ The length of $S_{increasing}$ and S_{common} is equal.

(d)

To tackle this question, I have referred to Geeksforgeeks.

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 $[2,7,8,3,4,5]$. Through above steps, we would firstly have the result array as $[2,7,8]$, then since 3 should be placed at where 7 is, we replace 7 with 3, so that the result array is $[2,3,8]$. If the input array were $[2,7,8,3]$, then even if $[2,3,8]$ is not a correct increasing subsequence, the length of it is the same as $[2,7,8]$, so the result is correct. Similarly, if the input array is $[2,7,8,3,4]$, then our result array should be $[2,3,4]$, an incorrect yet the same length with the correct subsequence. Meanwhile, the next step assures that we update the result array with $[2,3,4,5]$, the length of which is 4 as the correct answer. With that said, the code for such algorithm is as follows:

```python
# 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 n elements of the input array and use the binary search, which takes $O(\log n)$ time, to determine each element's position in the result array. So apparently the overall time complexity is $O(n \log n)$.

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 $O(n \log n)$ and the space complexity is $O(n)$.

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

For space complexity, the sorting step requires $O(n)$ space. Similarly, if the total weight permits the result list could contain every item with the original shape, which will lead to $O(n)$ space. Having a total value variable typically requires $O(1)$ space, therefore, the space complexity of this algorithm is $O(n)$.

(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:
   i. 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.
   ii. 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 L be a list of items. The greedy approach will sort L by the value per weight of each element first, and then pick the item with the highest ratio. Suppose after sorting the items in L are $a_1, a_2, a_3, \ldots . a_n$, and there is one way to fill the knapsack with different item selection.

$\because$ The greedy approach will fill the knapsack with $a_1, a_2, a_3, \ldots\ a_j$, where j is the number of the last item.

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

$\therefore$ The assumption is a contradiction.

$\therefore$ 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 $O(m \times n)$, 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 $O(m \times n)$ 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 app $O(m \times n)$roached once. Thus, the running time of this step should be $O(m \times n)$ as well. Finally, it takes $O(m \times n)$ time to check the entire matrix if there is still a sleeping cat. Therefore, overall, the running time of this algorithm is $O(m \times n)$.

(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:

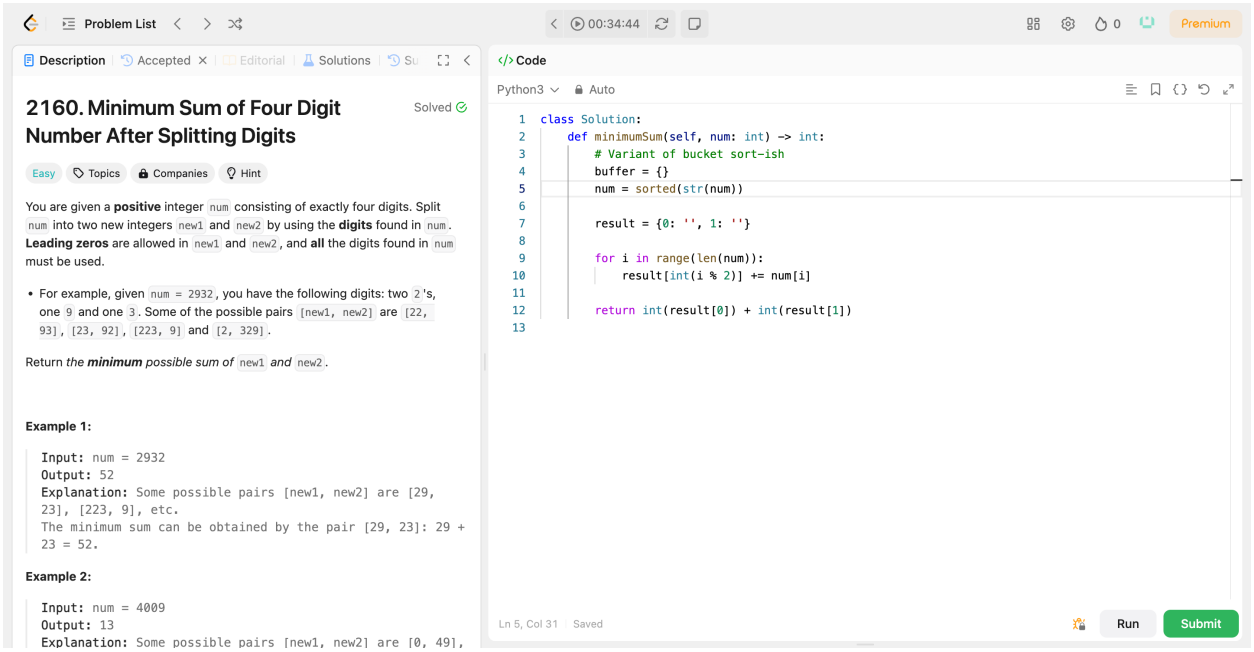
```
Wake-Up-All (matrix, target_x, target_y):
    if matrix[target_x][target_y] != -1:
    # If the target cat is not asleep, then return an empty list.
        return []

    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 (Easy), I used 34:44 minutes to solve it.



Problem List | 00:34:44 | Premium

Description | Accepted | Editorial | Solutions

2160. Minimum Sum of Four Digit Number After Splitting Digits — Solved

Easy | Topics | Companies | Hint

You are given a **positive** integer `num` consisting of exactly four digits. Split `num` into two new integers `new1` and `new2` by using the **digits** found in `num`. **Leading zeros** are allowed in `new1` and `new2`, and **all** the digits found in `num` must be used.

- For example, given `num = 2932`, you have the following digits: two `2`'s, one `9` and one `3`. Some of the possible pairs `[new1, new2]` are `[22, 93]`, `[23, 92]`, `[223, 9]` and `[2, 329]`.

Return *the **minimum** possible sum of* `new1` *and* `new2`.

Example 1:

```
Input: num = 2932
Output: 52
Explanation: Some possible pairs [new1, new2] are [29, 23], [223, 9], etc.
The minimum sum can be obtained by the pair [29, 23]: 29 + 23 = 52.
```

Example 2:

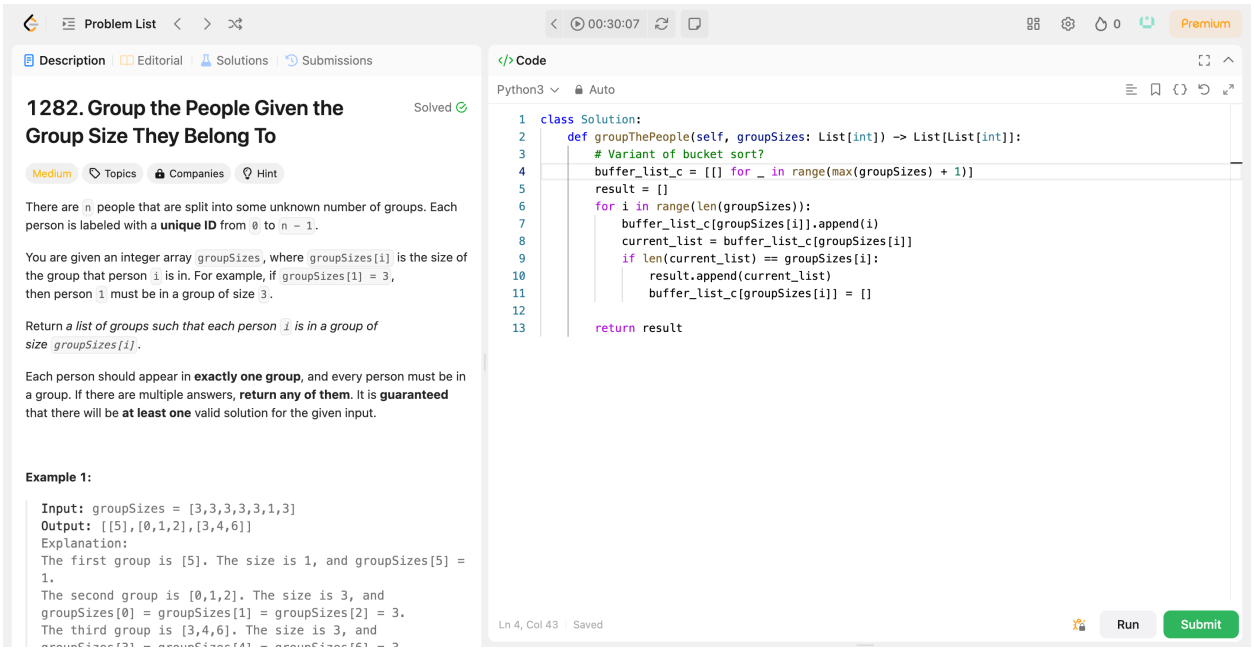
```
Input: num = 4009
Output: 13
Explanation: Some possible pairs [new1, new2] are [0, 49],
```

Code | Python3 | Auto

```python
class Solution:
    def minimumSum(self, num: int) -> int:
        # Variant of bucket sort-ish
        buffer = {}
        num = sorted(str(num))

        result = {0: '', 1: ''}

        for i in range(len(num)):
            result[int(i % 2)] += num[i]

        return int(result[0]) + int(result[1])
```

Ln 5, Col 31 | Saved | Run | Submit

1282. Group the People Given the Group Size They Belong To (Medium), I used 30:07 minutes to solve it.



Problem List | 00:30:07 | Premium

Description | Editorial | Solutions | Submissions

1282. Group the People Given the Group Size They Belong To — Solved

Medium | Topics | Companies | Hint

There are `n` people that are split into some unknown number of groups. Each person is labeled with a **unique ID** from `0` to `n - 1`.

You are given an integer array `groupSizes`, where `groupSizes[i]` is the size of the group that person `i` is in. For example, if `groupSizes[1] = 3`, then person `1` must be in a group of size `3`.

Return *a list of groups such that each person `i` is in a group of size `groupSizes[i]`.*

Each person should appear in **exactly one group**, and every person must be in a group. If there are multiple answers, **return any of them**. It is **guaranteed** that there will be **at least one** valid solution for the given input.

Example 1:

```
Input: groupSizes = [3,3,3,3,3,1,3]
Output: [[5],[0,1,2],[3,4,6]]
Explanation:
The first group is [5]. The size is 1, and groupSizes[5] = 1.
The second group is [0,1,2]. The size is 3, and groupSizes[0] = groupSizes[1] = groupSizes[2] = 3.
The third group is [3,4,6]. The size is 3, and
```

Code | Python3 | Auto

```python
class Solution:
    def groupThePeople(self, groupSizes: List[int]) -> List[List[int]]:
        # Variant of bucket sort?
        buffer_list_c = [[] for _ in range(max(groupSizes) + 1)]
        result = []
        for i in range(len(groupSizes)):
            buffer_list_c[groupSizes[i]].append(i)
            current_list = buffer_list_c[groupSizes[i]]
            if len(current_list) == groupSizes[i]:
                result.append(current_list)
                buffer_list_c[groupSizes[i]] = []

        return result
```

Ln 4, Col 43 | Saved | Run | Submit

(b)

I tried another problem before submitting these two, which is **1689. 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** 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 $[3,3,3,3,3,1,3]$, 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 4^{th} list of the result list grows into $[0,1,2]$ 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 $[5]$ because of '1', and finally the $[3,4,6]$.

The struggle I faced was the time complexity. The bucket sort has a time complexity of n, 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.