Q1:
There are two section of **bounded_knapsack method**:
First section :
Take each of the original items, spilt it into new items, with new quantity and respective weight. Then store then into an items array.
Example: if the item quantity is 10, it will be spilted to 4 new items, as quantity: 1(2^0), 2(2^1), 4(2^2), 2(remaining quantity),
the weight will be the new_quantity * weight.
Since there are N items(N = the number of item of the original input) , and for each item the complexity will be log(max_quantity)
Therefore, overall Time complexity of this section will be **O(N * log(max_quantity))** :
where 'max_quantity' is the max quantity of any item;

Second section:
After splite all the original items, a new items array will be created., with new items_len(the length of items in the new items array).
Then, do a bottom up approach of 0-1 kmapsack problem, which the Time complexity will be **O(items_len * limit):**
where 'limit' is the knapsack's capacity;
'items_len' is the length of the new items array created in first section;

Overall Time Complexity:
Since items_len is created by first section, with time complexity: O(N * log(max_quantity))
Therefore, overall time complexity of the bounded_knapsack method will be **O(N * log(max_quantity) * limit):**
Where 'N' is the number of item of the original input;
'max_quantity' is the max quantity of any item;
'limit' is the knapsack's capacity;

Q2:
longest_bitonic_subsequence method :
As there are two nested for loops, with each based on the length of the input array. So the time complexity will be **O(n^2):**
where 'n' is the length of the input array;

lcs method :
Using memoization approach, to find the lengthe of the longest common subsequence, the time complexity is **O(m * n)**:
where 'm' is the length of the array 1;
'n' is the length of the array 2;

getAllSubsequences :
There are three nested for loops to get all the possible longest common subsequences based on the max length of the LCS, the time complexity of this method will be upper bounded by it. Therefore, the time complexity of this method will be **O(u * m * n)**:
where 'u' is the length of the unique elements, when two input arrays are

combined;
'm' is the length of array 1;
'n' is the length of array 2;

Overall time complexity:
The overall time complexity is upper bounded by the getAllSubsequences method.
Therefore, the overall time complexity will be **O(u * m * n):**
where 'u' is the length of the unique elements, when two input arrays are combined;
'm' is the length of array 1;
'n' is the length of array 2;

Q3:
1)
min_coin_with_plan part is wrong. First error occurred when n = 6006

2)
min_coin_with_plan violated the **overlapping of sub-problem** elements. As the algorithm is tracking all possible combinations that lead to the minimum number of coins for certain n.

Such approach will increase the complexity and memory usage, but in DP what we want
is the value of the optimal sub-problem, not all the possible combinations for the optimal sub-problem.

3)
The first occurred error is when n = 6006.
For denom = [(1, 10), (2, 10), (3, 13), (7, 17), (14, 14), (29, 18), (57, 20), (115, 12), (231, 17), (462, 12)]

When the amount for coin change = **6006 cents**, the corresponding optimal solution should be **14 coins**(12 coins of 462 cents and 2 coins of 231), but the code(min_coin_with_plan) gives the following results : **AttributeError: 'NoneType' object has no attribute 'num_coin'**. When i = 6006

Which means that min_coin_with_plan method does not provide a solution for 6006 cents.
Due to the third for loop, when i=9, it gets the plan for min_coin_with_plan[5544].plan (which will use 12 coins of 462 cents) and then it would not go the if statement, as 5544 cents will used all the supply for 462 cents coin.

4)
To rectify the bug for min_coin_with_plan. For the first for loop, instead of starting i from 0 to m -1(increment), start from m - 1 to 0(decrement),so that when j = 6006, and all the supply for the largest is used.

Code :
Original code(line 29) => **for i in range(m):**

Proposed correction => **for i in range(m - 1, -1, -1):**

As the result, the code will consider the largest coin first and then the smallest coin, which is similar with the greedy approach.
Unlike the current code, when all the supply for the largest is used, that is no way for it to consider the smaller value coin, as i cannot go back, since it is start from 0 to m - 1.