CS202 Assignment 2 – Shin Minchul

Q1. Study of A2Q1_coin_change_limited.py

1.1 Provide an example input that causes the code to fail

```
root@DESKTOP-2H9Q6MQ:/mnt/c/Users/kmta1/OneDrive/Documents/2025 Y2T2/CS202 Design and Analysis o
[(1, 10), (2, 1), (3, 1), (7, 2), (14, 2), (29, 1), (59, 1), (118, 2), (237, 1), (473, 1)]
1 1 [[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
2 1 [[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]]
3 1 [[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]]
4 2 [[1, 0, 1, 0, 0, 0, 0, 0, 0, 0]]
5 2 [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0]]
6 5 [[4, 1, 0, 0, 0, 0, 0, 0, 0, 0]]
7 1 [[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]]
8 2 [[1, 0, 0, 1, 0, 0, 0, 0, 0, 0]]
9 2 [[0, 1, 0, 1, 0, 0, 0, 0, 0, 0]]
10 2 [[0, 0, 1, 1, 0, 0, 0, 0, 0, 0]]
inconsistent result at 6 cents, min_coin_plan give minimum 5 coins, but MinCoin = 3
```

After changing code to denom.append(curr, random.randint(1, 2)), inputs [(1,10), (2, 1), (3, 1), (7, 2), (14, 2), (29, 1), (59, 1), (118, 2), (237, 1), (473, 1)] causes the code to fail at 6 cents.

1.2 Generalize the issue by identifying the common characteristic(s) of input examples that lead to failure.

The issue is generated when the coin supply is tight, such as 1 or 2 coins per denomination as written in 1.1. The dynamic programming table (min_coin_plan) propagates suboptimal plans due to processing coins in ascending order (It should be descending order). When smaller coins are considered first, early non-optimal plans are stored, hence, the better combinations that use larger coins being not considered. Therefore, it will result in inconsistencies between min_coin_plan and MinCoin which print:

e.g “inconsistent result at 997 cents, no min_coin_plan, but MinCoin = 8” or

“Inconsistent result at 368 cents, min_coin_plan give minimum 8 coins, but MinCoin = 6”

1.3 Debug this code and justify the changes you made

```
denom = [(curr, random.randint(10, 20))]
for i in range(9):
    curr = 2 * curr + 1 - random.randrange(3)
    denom.append((curr, random.randint(1, 2)))
    # what if you change it to denom.append((curr, random.randint(1, 2)))
print(denom)

denom.sort(reverse=True, key=lambda x: x[0])
```

To solve the order-dependency problem, I modified coin by putting 1 more extra line which prioritize larger coin denominations during processing. This will ensure that the algorithm attempts to use higher denominations first which has the higher chance to obtain fewer total coins to make a specific coin. It also helps avoid cases where smaller coin paths are picked too early and mess up the future better answers. This change makes the result from DP more accurate and consistent with the actual minimum coin count we expect.

Q2. Complexity of LCVS Algorithm

```
Justification of the Time Complexity of the LCVS Algorithm:
    - m = len(A)  (length of sequence A)
    - n = len(B)  (length of sequence B)

1. Building a_positions map:
    - Iterates through all elements in A once.
    - Time complexity: O(m)

2. First Loop (for LDS calculation):
    - Outer loop runs for each element in B -> O(n)
    - For each B element found in A, we iterate through indices of A -> worst
case O(m)
    - Inner loop over j from 0 to i (on average O(m))
    - Total worst-case complexity for LDS part: O(n * m)

3. Second Loop (for LIS calculation):
    - Symmetric to LDS loop, just reversed -> also O(n * m)

4. Final loop to combine LDS and LIS:
    - Iterates over A once -> O(m)

Total Time Complexity:
    = 1 + 2 + 3 + 4
    = O(m) + O(2nm) + O(m)
    = O(2nm + 2m)
    = O(n * m)

Therefore, the overall worst-case time complexity is:
    = O(n * m)
```

Q3. Asymptotic Analysis

3.1 Part 1

Using the recursion tree method, we now calculate Asymptotic bound or the recurrence:

$$T(n) = 3 \cdot T\left(\frac{n}{4}\right) + \frac{n}{\log n}$$

First, at level i, there are 3^i subproblems of size $n/4^i$ and each cost is:

$$\frac{n/4^i}{\log(n/4^i)} = \frac{n}{4^i(\log n - i\log 4)}$$

Total cost at level i is:

$$3^i \cdot \frac{n}{4^i(\log n - i\log 4)} = \frac{n \cdot (3/4)^i}{\log n - i\log 4}$$

As it stops when $h = \log_4 n$, T(n) can be expressed as:

$$T(n) = \sum_{i=0}^{\log_4 n} \frac{n \cdot (3/4)^i}{\log n - i\log 4}$$

From here we can approximate $\log n - i\log 4 \approx \log n$ hence T(n) will be:

$$T(n) \approx \frac{n}{\log n}\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i = \frac{n}{\log n} \cdot \frac{1}{1-\frac{3}{4}} = \frac{n}{\log n} \cdot 4$$

Therefore, the total time complexity is: $T(n) = \Theta\left(\frac{n}{\log n}\right)$

3.2 Part 2

Using the recursion tree method, we now calculate Asymptotic bound for the recurrence:

$$T(n) = 2 \cdot T(n-1) + n$$

From here, we can get:

$$T(n) = 2 \cdot T(n-1) + n = 2 \cdot T(2 \cdot T(n-2) + n - 1) + n$$
$$= 2^2 \cdot T(n-2) + 2(n-1) + n$$
$$= 2^3 \cdot T(n-3) + 2^2 \cdot (n-2) + 2(n-1) + n$$

From here we can observe that after k steps, $T(n) = 2^k \cdot T(n-k) + \sum_{i=0}^{k-1} 2^i \cdot (n-i)$

Let k = n we can get $T(n) = 2^n \cdot T(0) + \sum_{i=0}^{n-1} 2^i \cdot (n-i)$

Here, the sum $\sum_{i=0}^{n-1} 2^i \cdot (n-i)$ contains terms where 2^i grows exponentially and $n-i$ decreases linearly.

Even though $(n-i)$ shrinks, the exponential growth of 2^i dominates.

We can upper bound each term as:

$$2^i \cdot (n-i) \le 2^i \cdot n$$

Hence $\sum_{i=0}^{n-1} 2^i \cdot (n-i) \le n \cdot \sum_{i=0}^{n-1} 2^i = n \cdot (2^n - 1) = O(n \cdot 2^n)$

Since $T(0)$ is a constant, the term $2^n \cdot T(0)$ is asymptotically smaller than $O(n \cdot 2^n)$ Therefore, the total time complexity of this is $T(n) = O(n \cdot 2^n)$