Min Coin With Plan

Q1: (1 mark) Which part is wrong? Is it min_coin_with_plan or min_coin_num_only

The error is with `min_coin_with_plan` . I am currently unable to identify which exact part is wrong but I have an understanding of what and why it goes wrong. The code loops from the smallest denomation, 1, to the largest denomination, 462 for all values from 1 to the target sum.

If we use a larger target sum (eg 6005), when the code reaches the "end", it will have the solution [0, 0, 0, 0, 0, 0, 0, 2, 1, 12]. However, it is clear that it has not used ALL possible coins. This likely stops because it has used up all possible quantity of coins of the denomination 462, which is 12 as well.

When it looks at the sum 6006 after this, it will be considering the last denom (462). Hence when it takes 6006 - 462, it will get 5544 which uses 12 coins of 462 which is the limit. Therefore, it will believe the code cannot continue because the only solution left for 462 uses 1 coin of 462, which is no longer valid.

At that point in time, the algorithm will not consider the possibility of taking 6005 + 1 which is a valid solution.

> Question: Can array min_coin_num_only be changed to a 1-dimensional array?

No, the array used to solve this cannot be changed to a 1-dimensional array. There are two factors to take note of, the quantity as well as the value of the denominations. This will require minimally a 2-dimensional array to account for both factors accordingly.

Q2: (2 marks) Which element of dynamic programming has the wrong part violated? Is it the optimal sub-structure, or the overlapping of sub-problems?

I believe the optimal sub-structure has been violated. The optimal substructure states that if a solution is optimal to a larger problem, it must be optimal to the sub-problem too. In this scenario, I believe the

error here is that the optimal sub-structure is incorrectly defined and does not lead to an optimal solution for the larger problem.

During the process of building the table, the code will always update and replace the `MinCoin` object of target j if the current solution requires a lesser number of coins. While this seems correct at first glance, it means that the as we are building the solution we are losing potential combinations. These potential combinations will be required when the number of coins we are looking for change exceeds a certain value.

A more clear example with numbers is elaborated in Q3.

Q3: (2 marks) Give a counter example that fails this code.

When the amount for coin change = `6006` cents, the corresponding optimal solution `should be using 13 coins, 1x $1, 2x $115, 1x $231, 12x $462` coins, but the code gives the following results `None` .

A second counter example is with the following denom of 2x $1 and 1x $2 coins. The logical solution to n=4 will be to use all the coins, but the `min_coin_with_plan` gives a None solution. This is because when it loops through all possibilities of using getting $1, it will create the and solve the subproblems $1 = 1 x $1, $2 = 2 x $1. When it then looks at the possibilities of getting $2, it will see that using 1 x $2 is a more effective way of getting $2, and override the entry using $1.

When we consider the value $4 with denomination $2, it will take $4 - $2 = $2, and then it will believe that there is no possible way to get $2 anymore. This would be an example of how the sub problem it solves does not solve the actual problem.

Q4: (5 marks) Propose a way to rectify the bug, please highlight your proposed correction in Python code with comments.

My proposed solved solution runs into space complexity issue when solving for the given combination. My proposed solution involves changing the MinCoin datastructure to use a min heap instead to store the possible plans. In theory, when checking through the available plans, the algorithm will be able to find at least 1 plan where the combination of coins used are valid and this will remove the problem where possible combinatons are overwritten using the current `MinCoin` object.

However, this approach resulted in storing all possible combinations for all possible values from 1 to n, where if both n and the number of available denomination and coins are high it runs into a memory error on my 32GB ram desktop.

The code blocks is given below with more comments.

```
import random, time
# random generate 10 denominations
# the next denomination ~= the previous * 2, or plus/minus 1
# for each denomination, generate a quantity between 10 and 20 inclusive

random.seed('coin')
curr = 1
denom = [(curr, random.randint(10, 20))]
for i in range(9):
    curr = 2 * curr + 1 - random.randrange(3)
    denom.append((curr, random.randint(10, 20)))
print(f"The denominations are in denom: quantity \n {denom}")


# class MinCoin:
#     def __init__(self, n, p):
#         self.num_coin = n
#         self.plan = [p]
#     def add_plan(self, p):
#         self.plan.append(p)

# ======================== PROPOSED CHANGES ==============================
"""
I created a new data structure to maintain all possible ways combination you can
get to a certain number of coins. This data structure is a min heap, where the key is
the total sum of the coins in the list aka the plan.

The main motivator of this was I noticed that whenever the previous solution and
MinCoin encountered a solution for a particular solution that uses a smaller number of coins,
it would delete and remove the older, larger solution. This made it problematic to "retrieve"
less optimal solutions where the number of coins had been exhausted.

My initial solution was to create a min heap and store all possible combinations of coins
that makes up the sum, such that when it was looped over the ones with the min coins used
would appear first. There will be some issues faced with larger inputs this way however.

Instead of having a pop method, I tried to implement an iterate method that will
properly iterate through the min heap based on an array via indexing, but was unable to
do that properly. The current implementation will only return the smallest element correct,
aka the one at 0 index but iterates through the index in strictly increasing order instead of
"jumping" from indexes to simulate if we were to pop the min heap.
```

```
This part had the influence of ChatGPT (Prompts provided separately).
"""
class MinSumListHeap:
    def __init__(self):
        self.plan = []
        self.history = set()

    def add_list(self, new_plan):
        if str(new_plan) in self.history:
            print("We saved some space")
            return
        self.history.add(str(new_plan))
        self.plan.append(new_plan)
        self._heapify_up(len(self.plan) - 1)

    def _heapify_up(self,index):
        while index > 0:
            parent = (index - 1) // 2
            if self._get_sum(self.plan[index]) < self._get_sum(self.plan[parent]):
                self.plan[index], self.plan[parent] = self.plan[parent], self.plan[index]
                index = parent
            else:
                break
    def _heapify_down(self, index):
        while True:
            left_child_index = 2 * index + 1
            right_child_index = 2 * index + 2
            smallest = index

            if (left_child_index < len(self.plan) and
                    self._get_sum(self.plan[left_child_index]) < self._get_sum(self.plan[smallest
                smallest = left_child_index

            if (right_child_index < len(self.plan) and
                    self._get_sum(self.plan[right_child_index]) < self._get_sum(self.plan[smalles
                smallest = right_child_index

            if smallest != index:
                self.plan[index], self.plan[smallest] = self.plan[smallest], self.plan[index]
                index = smallest
            else:
                break
    def _get_sum(self, lst):
        return sum(lst)

    def __iter__(self):
        return iter(self.data)
```

```
# print("========Testing minSumListHeap========")
# minSumListHeap = MinSumListHeap()
# minSumListHeap.add_list([1,0,0])
# minSumListHeap.add_list([0,0,2])
# minSumListHeap.add_list([0,0,3])
# minSumListHeap.add_list([0,4,3])
# minSumListHeap.add_list([0,2,0])
# minSumListHeap.add_list([2,0,0])
# minSumListHeap.add_list([5,0,0])

# for plan in minSumListHeap.plan:
#     print(plan)

# maybe we start by modifying minCoin
# I need it to not just store the smallest number of coins, but use a deque based on the number o

# ======================== PROPOSED CHANGES ==============================
n = 6006
n = 12897 # This is the biggest possible number, the total sum of all coins
"""
Uncomment the following to use the smaller test case.
"""
# denom = [(1,2), (2,1)]
# n = 4



m = len(denom)
min_coin_with_plan = [None] * (n + 1)
min_coin_with_plan[0] = MinSumListHeap()
min_coin_with_plan[0].add_list([0] * m)

# ======================== PROPOSED CHANGES ==============================
"""
Beyond initiating the above data structure, I did not change much of the original code.
I understood that the main problem in the original code was how it handled the
situation where the the solution exceeded the current number of coins available for that denomina
This caused incidents where like it would not be able to resolve for 6006, while it had a solutio

I initially believed that by replacing the MinCoin data structure to the one above,
the algorithm below will loop through all the plans and will be able to find one where the
coins used would be a valid combination.

However, the main problem with my proposed change and implementation was the space complexity.
It would quickly come up with a LOT of plans, while they were all feasible it made it impossible
for my computer to get a solution for the proposed denomination and n = 6006.

It does work correctly for smaller amounts. An example the original algorithm failed is for
denoms [(1,2), (2,1)] and n = 4. The original algorithm would not return a solution for 4,
but the solution I implemented will.
```

```
    """

    start = time.time()
    for i in range(m):
        # print(f"Checking for denom {i} which is {denom[i][0]}")
        for j in range(1, n+1):
            # print("Checking for target", j, "and denom", i, "which is", denom[i][0])
            if j >= denom[i][0] and min_coin_with_plan[j - denom[i][0]] is not None:
                # print(f"There are {len(min_coin_with_plan[j - denom[i][0]].plan)} plans for {j - de
                for p in min_coin_with_plan[j - denom[i][0]].plan:
                    if p[i] < denom[i][1]:
                        if min_coin_with_plan[j] is None:
                            min_coin_with_plan[j] = MinSumListHeap()
                        p_new = p.copy()
                        p_new[i] += 1
                        min_coin_with_plan[j].add_list(p_new)

    print(min_coin_with_plan[n].plan)

    print("--- %s seconds ---" % (time.time() - start))

    # MAYBE EACH TIME WE REPLACE, WE ADD THE SUM OF THE NUMBER TOO
```