Question 1 (30 points). *A **palindrome** is a non-empty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, civic, racecar, and aibohphobia (fear of palindromes).*
Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string of length n. For example, given the input $\{character\}$, your algorithm should return $\{carac\}$.

(a) (8 points) Please describe your subproblems.

(b) (8 points) Please describe the decisions.

(c) (10 points) Please write down the recursion and base cases.

(d) (4 points) Please describe the running time of your algorithm in terms of the number of subproblems and the running time per subproblem.

(a) Subproblem

Consider the string x of length n and reverse x is called as y. A subproblem can be defined as finding the longest palindromic subsequence in a substring x[1...i] and y[1...j],

Each subproblem involves finding the LCS between the original string and its reverse. This allows us to use dynamic programming to compute the solution.

1. If characters x[i] and y[j] match, then the current character contributes to the palindrome.
2. If they don't match, then you need to decide whether to exclude x[i] or y[j] to maximize the subsequence.

(b) Decisions

Case 1: x[i] == y[j]

dp[i][j] = dp[i-1][j-1] + 1

Case2: x[i] != y[j]

dp[i][j] = max(dp[i-1][j], dp[i][j-1])

(c) Recursion and Base Cases

Recursion:

If x[i] == x[j], then dp[i][j] = dp[i-1][j-1] + 1

If x[i] != x[j], then dp[i][j] = max(dp[i-1][j], dp[i][j-1])

Base Case:

For any i or j where one of the strings is empty (i=0 or j=0), the LCS is 0 because there are no characters to match. Thus:

dp[i][0] = 0 and dp[0][j] = 0

(d) Running Time

The number of subproblems are separately **i and j**. The running time per subproblem is **i*j**. Total Running Time is **n^2**.

Python Code：

```
def lcs(x, y) -> (int, str):
   m = len(x)
   n = len(y)
   dp = [[0] * (n + 1) for _ in range(m + 1)]

   for i in range(1, m + 1):
      for j in range(1, n + 1):
         if x[i - 1] == y[j - 1]:
            dp[i][j] = dp[i - 1][j - 1] + 1
         else:
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
      # Backtrack to get the longest subsequence
   res = []
   i, j = m, n
   while i > 0 and j > 0:
      if x[i - 1] == y[j - 1]:
         res.append(x[i - 1])
         i -= 1
         j -= 1
      elif dp[i - 1][j] >= dp[i][j - 1]:
         i -= 1
      else:
         j -= 1
   return dp[m][n], ''.join(res)

if __name__ == '__main__':
   x = 'character'
   y = x[::-1]
   print(lcs(x, y))
```

Question 2 (30 points). *You have to store n books $(b_1, b_2, \ldots, b_n)$ on shelves in a library. The order of books is fixed by the cataloging system and cannot be rearranged. A book b_i has a thickness t_i and height h_i, where $1 \leq i \leq n$. The length of each bookshelf at the library is L. The values of h_i are not necessarily assumed to be all the same. In this case, we have the freedom to adjust the height of each bookshelf to that of the tallest book on the shelf. Thus, the cost of a particular layout is the sum of the heights of the largest book on each shelf.*

(a) *(10 points) Give an example to show that the greedy algorithm of stuffing each shelf as full as possible does not always give the minimum overall height.*

(b) *(20 points) Describe in English an algorithm based on dynamic programming to achieve the minimum overall height. No need to provide pseudocode, but you must state your subproblems, decisions, recursion, base case(s) and analyze the running time complexity.*

a.

Item	Thickness	Height
book1	2	4
book2	2	2
book3	2	7
book4	2	9

Initial conditions: L = 6

Case1: stuffing each shelf is as full as possible.

The thickness of book1 + book2 + book 3 is 6. The height of first bookshelf is 7;

The height of second bookshelf is 9.

Total height is 16.

Case2: stuffing each shelf is not as full as possible.

Book1 and book2 put in the first bookshelf—The height is 4;

Book3 and book4 put in the second bookshelf – The height is 9;

Total height is 13.

Therefore, stuffing each shelf as full as possible does not always give the minimum overall height.

b.

Subproblem Definition:

dp[i] = minimum total height needed to arrange books from book 1 to book i.
Note: i from 1 to n.

Decision:

Case1: if i = 0:

return 0

Case2: Book i is placed alone on a new shelf:

$$dp[i] = dp[i-1] + h_i$$

Case3: Books j+1 to i are placed together on the same shelf:

If $t_{j+1} + \ldots + t_i \leq L$, then

$$dp[i] = \min(dp[i], dp[j] + \max(h_{j+1}, \ldots, h_i))$$

We try all valid j < i such that the total thickness from j+1 to i does not exceed shelf length L.

Recursion:

To compute dp[i], consider placing book i on the same shelf as some earlier books j+1 to i (i.e., ending the previous shelf at position j). We try all valid j < i such that the total thickness from j+1 to i is ≤ L.

dp[i] = min(dp[j] + max_height(j+1 to i)) for all j where total_thickness(j+1 to i) <= L

Base cases:

dp[0] = 0 — no books → height is 0

Time Complexity:

- For each i, we may check all j < i, and for each such j, compute max height and total thickness of books from j+1 to i.
- This gives $O(n^2)$ time complexity.

Question 3 (40 points). *You are given an unlimited supply of coins of a given denomination S and a target amount N. Your goal is to find the minimum number of coins required to make change for N using the given denominations.*

Example #1. *Suppose the denominations as $S = \{1, 3, 5, 7\}$ and $N = 18$, the minimum number coins is 4 (i.e., $\{7, 7, 3, 1\}$ or $\{5, 5, 5, 3\}$ or $\{7, 5, 5, 1\}$).*

Example #2. *Suppose the denominations are $S = \{25, 10, 1\}$ and $N = 40$, the minimum number of coins is 4 (i.e., $\{10, 10, 10, 10\}$).*

(a) *(6 points) Can you show that the greedy choice (used in HW4) for coin changing is suboptimal for Example #2?*

(b) *(4 = 2 + 2 points) Can you please describe a brute-force (non-DP) approach to solve Example #2? What will be the running time of the brute-force approach?*

(c) *(2 points) Suppose we use dynamic programming, which problem that we have solved in the class is the closest to this problem?*

(d) *(6 points) Please describe your subproblems based on your answer in (c)?*

(e) *(6 points) Please explain the decisions you need to make to solve each of your subproblems.*

(f) *(6 points) Please write down the recursion for this problem and the base case(s).*

(g) *(4 points) Please describe the running time of your algorithm in terms of the number of subproblems and the running time per subproblem.*

(h) *(6 points) For Example #2, please describe the value of each of your subproblems that your dynamic programming algorithm (based on previous steps) will compute in each iteration. In how many iterations will your algorithm terminate?*

a. Firstly, sort coins by their denominations in decreasing order, and the greedy choice will be to select the coin with the largest denomination as long as $N - d > d$, where d is the denomination. Otherwise, pick the next denomination until $N - d = 0$.

S = {25,10,1} , N = 40, count = 0(The number of coins)

Select the largest denomination 25: 40 – 25 = 15, count = 1

Select the denomination 10: 15 – 10 = 5, count = 2

Select the denomination 1: total time is 5, count = 7

The final number of count is 7 and not the best result(actual result is 4).

b. 1. Try all combinations of coins (with repetitions allowed) that sum to 40, and track the one with the minimum number of coins used.

2. **Running Time**: N = target amount, k = number of denominations. Time complexity is O(k^N). About Example #2, time complexity is 3^40

c. P6: Knapsack w/ Repetition

d. dp[i] as the minimum number of coins needed to form amount N (i is from 1 to N). We compute this for all coins get the min dp[i].

e. **Decision:**

If we include coin, we add 1 to the solution of subproblem dp[i - coin]

dp[i] = dp[i - coin] + 1.

f. **Recursion:**

For each coin in the set of denominations S, and for each amount w ≥ coin:

dp[i] = min(dp[i], dp[i - coin] + 1)

Base case:

dp[0] = 0 (zero coins needed to make amount 0)

dp[1:N] = ∞(Because we calculate the minimum number of coins)

g. **Running time:**

N = target amount, k = number of denominations

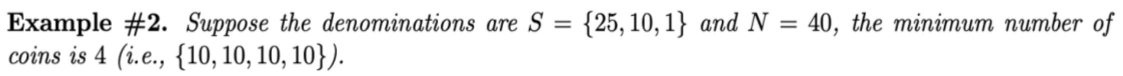
The number of subproblems is N.

The running time per subproblem is k.

The total running time is N*k

h. **Calculation:**

Firstly, we should sort the S so that we can select the largest denominations.



Example #2. *Suppose the denominations are* $S = \{25, 10, 1\}$ *and* $N = 40$, *the minimum number of coins is* 4 *(i.e.,* $\{10, 10, 10, 10\}$*).*

	Res	25	10	1	Iteration
0	**0**				
1	1	-	-	**dp[1-1] + 1 = 1**	**1**
2	2	-	-	**dp[2-1] + 1= 2**	**1**
3	**3**	-	-	**dp[3-1] + 1= 3**	**1**
4	**4**	-	-	**dp[4-1] + 1= 4**	**1**
5	**5**	-	-	**dp[5-1] + 1= 5**	**1**
6	**6**	-	-	**dp[6-1] + 1= 6**	**1**
7	**7**	-	-	**dp[7-1] + 1= 7**	**1**
8	**8**	-	-	**dp[8-1] + 1= 8**	**1**

9	**9**	**-**	**-**	**dp[9-1] + 1= 9**	**1**
10	**1**	**-**	**dp[10-10] + 1= 1**	**dp[10-1] + 1= 10**	**2**
11	**2**	**-**	**dp[11-10] + 1= 1**	**dp[11-1] + 1 = 11**	**2**
12-19	**3-10**	**-**	**Same calculation**	**Same calculation**	**16**
20	**2**	**-**	**dp[20-10] + 1= 2**	**dp[20-1] + 1= 11**	**2**
21-24	**3-6**	**-**	**Same calculation**	**Same calculation**	**8**
25	**1**	**dp[25-25] + 1= 1**	**dp[25-10] + 1= 7**	**dp[25-1] + 1= 7**	**3**
26-29	**2-5**	**Same calculation**	**Same calculation**	**Same calculation**	**12**
30	**3**	**dp[30-25] + 1= 6**	**dp[30-10] + 1= 3**	**dp[30-1] + 1= 6**	**3**
31-34	**4-7**	**Same calculation**	**Same calculation**	**Same calculation**	**12**
35	**2**	**dp[35-25] + 1= 2**	**dp[35-10] + 1= 2**	**dp[35-1] + 1= 11**	**3**
36-39	**3-6**	**Same calculation**	**Same calculation**	**Same calculation**	**12**
40	**4**	**dp[40-25] + 1= 7**	**dp[40-10] + 1= 4**	**dp[40-1] + 1= 7**	**3**

Iterations is 87 if coin <= w(w is from 1 to 40).

Iterations is 120 if we don’t compare the condition of coin and w.