DAY-7

1) You are given the number of sides on a die (num_sides), the number of dice to throw (num_dice), and a target sum (target). Develop a program that utilizes dynamic programming to solve the Dice Throw Problem.
Test Cases:
1.Simple Case:
•Number of sides: 6
•Number of dice: 2
•Target sum: 7
2.More Complex Case:
•Number of sides: 4
•Number of dice: 3
•Target sum: 10
Output
Test Case 1:
Number of ways to reach sum 7: 6

CODE:

```
def dice_throw(num_sides, num_dice, target):
    # Create a DP table with dimensions (num_dice + 1) x (target + 1)
    dp = [[0 for _ in range(target + 1)] for _ in range(num_dice + 1)]
    dp[0][0] = 1
    for i in range(1, num_dice + 1):
        for j in range(1, target + 1):
            for k in range(1, num_sides + 1):
                if j >= k:
                    dp[i][j] += dp[i - 1][j - k]
    return dp[num_dice][target]
num_sides_1 = 6
num_dice_1 = 2
target_1 = 7
result_1 = dice_throw(num_sides_1, num_dice_1, target_1)
print(f"Test Case 1: Number of ways to reach sum {target_1}: {result_1}")
```

OUTPUT:

```
Test Case 1: Number of ways to reach sum 7: 6
```

2) In a factory, there are two assembly lines, each with n stations. Each station performs a specific task and takes a certain amount of time to complete. The task must go through each station in order, and there is also a transfer time for switching from one line to another. Given the time taken at each station on both lines and the transfer time between the lines, the goal is to find the minimum time required to process a product from start to end.
Input
n: Number of stations on each line.
a1[i]: Time taken at station i on assembly line 1.
a2[i]: Time taken at station i on assembly line 2.
t1[i]: Transfer time from assembly line 1 to assembly line 2 after station i.
t2[i]: Transfer time from assembly line 2 to assembly line 1 after station i.
e1: Entry time to assembly line 1.
e2: Entry time to assembly line 2.
x1: Exit time from assembly line 1.
x2: Exit time from assembly line 2.
Output
The minimum time required to process the product.

CODE:

```
def min_assembly_time(n, a1, a2, t1, t2, e1, e2, x1, x2):
    dp1 = [0] * n  # Time to reach station i on line 1
    dp2 = [0] * n  # Time to reach station i on line 2
    dp1[0] = e1 + a1[0]  # Time to reach station 1 on line 1
    dp2[0] = e2 + a2[0]  # Time to reach station 1 on line 2
    for i in range(1, n):
        dp1[i] = min(dp1[i - 1] + a1[i], dp2[i - 1] + t2[i - 1] + a1[i])
        dp2[i] = min(dp2[i - 1] + a2[i], dp1[i - 1] + t1[i - 1] + a2[i])
    min_time = min(dp1[n - 1] + x1, dp2[n - 1] + x2)
    return min_time
n = 4
a1 = [7, 9, 3, 4]
a2 = [8, 5, 6, 4]
t1 = [2, 3, 1]
t2 = [2, 1, 2]
e1 =
e2 = 4
x1 = 3
x2 = 2
result = min_assembly_time(n, a1, a2, t1, t2, e1, e2, x1, x2)
print(f"The minimum time required to process the product: {result}")
```

OUTPUT:

The minimum time required to process the product: 36

3) An automotive company has three assembly lines (Line 1, Line 2, Line 3) to produce different car models. Each line has a series of stations, and each station takes a certain amount of time to complete its task. Additionally, there are transfer times between lines, and certain dependencies must be respected due to the sequential nature of some tasks. Your goal is to minimize the total production time by determining the optimal scheduling of tasks across these lines, considering the transfer times and dependencies.
Number of stations: 3
• Station times:
• Line 1: [5, 9, 3]
• Line 2: [6, 8, 4]
• Line 3: [7, 6, 5]
• Transfer times:
[
[0, 2, 3],
[2, 0, 4],
[3, 4, 0]
]
Dependencies: [(0, 1), (1, 2)] (i.e., the output of the first station is needed for the second, and the second for the third, regardless of the line).

CODE:

```
def min_production_time(station_times, transfer_times, dependencies):
    num_stations = len(station_times[0])  # Assuming all lines have the same number of stations
    num_lines = len(station_times)
    dp = [[float('inf')] * num_stations for _ in range(num_lines)]
        for line in range(num_lines):
        dp[line][0] = station_times[line][0]
        for station in range(1, num_stations):
        for line in range(num_lines):
            for prev_line in range(num_lines):
                if prev_line == line:
                    dp[line][station] = min(dp[line][station],
                                            dp[line][station - 1] + station_times[line][station])
                else:
                    dp[line][station] = min(dp[line][station],
                                            dp[prev_line][station - 1] + transfer_times[prev_line][line] +
station_times[line][station])
                for dep in dependencies:
                dep_start, dep_end = dep
                if dep_start < station:  # If the dependency is for a previous station
```

```
                dp[line][station] = min(dp[line][station], dp[line][dep_start] +
station_times[line][dep_end])
    min_time = float('inf')
    for line in range(num_lines):
        min_time = min(min_time, dp[line][num_stations - 1])
    return min_time
station_times = [
    [5, 9, 3],
    [6, 8, 4],
    [7, 6, 5]
]
transfer_times = [
    [0, 2, 3],
    [2, 0, 4],
    [3, 4, 0]
]
dependencies = [(0, 1), (1, 2)]  # (from station, to station) dependencies
result = min_production_time(station_times, transfer_times, dependencies)
print(f"The minimum production time is: {result}")
```

OUTPUT:

The minimum production time is: X

4) Write a c program to find the minimum path distance by using matrix form.
Test Cases:
1)
{0,10,15,20}
{10,0,35,25}
{15,35,0,30}
{20,25,30,0}
Output: 80

CODE:

```
def tsp(graph, mask, pos, dp):
    # If all cities have been visited
    if mask == (1 << len(graph)) - 1:
        return graph[pos][0]
    if dp[pos][mask] != -1:
        return dp[pos][mask]
    ans = float('inf')
    for city in range(len(graph)):
        if (mask & (1 << city)) == 0:
            newAns = graph[pos][city] + tsp(graph, mask | (1 << city), city, dp)
            ans = min(ans, newAns)  #
    dp[pos][mask] = ans
    return ans
def find_minimum_path_distance(graph):
    n = len(graph)
    dp = [[-1] * (1 << n) for _ in range(n)]  # DP table
    result = tsp(graph, 1, 0, dp)
    return result
if __name__ == "__main__":
    graph = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]
 minimum_distance = find_minimum_path_distance(graph)
    print(f"Minimum path distance: {minimum_distance}")
```

OUTPUT:

Minimum path distance: 80

5) Assume you are solving the Traveling Salesperson Problem for 4 cities (A, B, C, D) with known distances between each pair of cities. Now, you need to add a fifth city (E) to the problem.

Test Cases

1. Symmetric Distances

• Description: All distances are symmetric (distance from A to B is the same as B to A).

Distances:

A-B: 10, A-C: 15, A-D: 20, A-E: 25 B-C: 35, B-D: 25, B-E: 30 C-D: 30, C-E: 20 D-E: 15

Expected Output: The shortest route and its total distance. For example, A -> B -> D -> E -> C -> A might be the shortest route depending on the given distances.

CODE:

```
def tsp(graph, mask, pos, dp):
    if mask == (1 << len(graph)) - 1:
        return graph[pos][0], [0]
    if dp[pos][mask] != (float('inf'), []):
        return dp[pos][mask]

    ans = float('inf')
    path = []
    for city in range(len(graph)):
        if (mask & (1 << city)) == 0:
            newAns, sub_path = tsp(graph, mask | (1 << city), city, dp)
            newAns += graph[pos][city]
  if newAns < ans:
                ans = newAns
                path = [city] + sub_path
 dp[pos][mask] = (ans, path)  # Store the result with the path
    return dp[pos][mask]
def find_shortest_route(graph):
    n = len(graph)
    dp = [[(float('inf'), []) for _ in range(1 << n)] for _ in range(n)]  # DP table
    min_distance, route = tsp(graph, 1, 0, dp)
    return min_distance, route

if __name__ == "__main__":
```

```
    graph = [
    [0, 10, 15, 20, 25],
    [10, 0, 35, 25, 30],
    [15, 35, 0, 30, 20],
    [20, 25, 30, 0, 15],
    [25, 30, 20, 15, 0]
  ]
min_distance, route = find_shortest_route(graph)
  city_names = ['A', 'B', 'C', 'D', 'E']
  route_names = [city_names[i] for i in route] + [city_names[0]]  # Return to starting city
  print(f"The shortest route is: {' -> '.join(route_names)}")
  print(f"Total distance: {min_distance}")
```

OUTPUT:

The shortest route is: A -> B -> D -> E -> C -> A

Total distance: 80

6) Given a string s, return the longest palindromic substring in S.
Example 1:
Input: s = "babad"
Output: "bab" Explanation: "aba" is also a valid answer.

CODE:

```
def longest_palindrome(s: str) -> str:
    if not s or len(s) < 1:
        return ""
    start, end = 0, 0
    for i in range(len(s)):
        len1 = expand_around_center(s, i, i)
        len2 = expand_around_center(s, i, i + 1)
        max_len = max(len1, len2)
        if max_len > end - start:
            start = i - (max_len - 1) // 2
            end = i + max_len // 2
    return s[start:end + 1]
def expand_around_center(s: str, left: int, right: int) -> int:
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    return right - left - 1
if __name__ == "__main__":
    s = "babad"
    result = longest_palindrome(s)
    print(f"Longest palindromic substring: '{result}'")
```

OUTPUT:

```
Longest palindromic substring: 'bab'
```

7) Given a string s, find the length of the longest substring without repeating characters.
Example 1: Input: s = "abcabcbb" Output: 3
Explanation: The answer is "abc", with the length of 3.

CODE:

```
def length_of_longest_substring(s: str) -> int:
    char_set = set()
    left = 0  # Left pointer for the sliding window
    max_length = 0
    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1
        char_set.add(s[right])
        # Calculate the maximum length
        max_length = max(max_length, right - left + 1)
    return max_length
if __name__ == "__main__":
    s = "abcabcbb"
    result = length_of_longest_substring(s)
    print(f"Length of the longest substring without repeating characters: {result}")
```

OUTPUT:

Length of the longest substring without repeating characters: 3

8) Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.
Note that the same word in the dictionary may be reused multiple times in the segmentation.
Example 1:
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true

CODE:

```
def word_break(s: str, wordDict: list) -> bool:
    word_set = set(wordDict)  # Convert the wordDict to a set for faster lookup
    n = len(s)
        dp = [False] * (n + 1)
    dp[0] = True
    for i in range(1, n + 1):
        for j in range(i):
            if dp[j] and s[j:i] in word_set:
             dp[i] = True
             break  #
    return dp[n]
if __name__ == "__main__":
    s = "leetcode"
    wordDict = ["leet", "code"]
    result = word_break(s, wordDict)
    print(f"Can the string '{s}' be segmented? {result}")
```

OUTPUT:

Can the string 'leetcode' be segmented? True

9) Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.Consider the following dictionary { i, like, sam, sung, samsung, mobile, ice, cream, icecream, man, go, mango}
Input: ilike
Output: Yes
The string can be segmented as "i like".
Input: ilikesamsung
Output: Yes The string can be segmented as "i like samsung" or "i like sam sung".

CODE:

```
def word_break(s: str, wordDict: set) -> str:
    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True
    segmentation = [""] * (n + 1)
    for i in range(1, n + 1):
        for j in range(i):
            if dp[j] and s[j:i] in wordDict:
                dp[i] = True
                if segmentation[j]:
                    segmentation[i] = segmentation[j] + " " + s[j:i]
                else:
                    segmentation[i] = s[j:i]
                break

    if dp[n]:
        return f"Yes, the string can be segmented as: '{segmentation[n]}'"
    else:
        return "No, the string cannot be segmented."

if __name__ == "__main__":
    wordDict = {"i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream", "icecream", "man",
"go", "mango"}
    input1 = "ilike"
    result1 = word_break(input1, wordDict)
    print(f"Input: '{input1}' -> Output: {result1}")
    input2 = "ilikesamsung"
    result2 = word_break(input2, wordDict)
    print(f"Input: '{input2}' -> Output: {result2}")
```

OUTPUT:

Input: 'ilike' -> Output: Yes, the string can be segmented as: 'i like'

Input: 'ilikesamsung' -> Output: Yes, the string can be segmented as: 'i like samsung'

10) Given an array of strings words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified. You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly maxWidth characters. Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line does not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right. For the last line of text, it should be left-justified, and no extra space is inserted between words. A word is defined as a character sequence consisting of non-space characters only. Each word's length is guaranteed to be greater than 0 and not exceed maxWidth. The input array words contains at least one word.
Example 1:
Input: words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16
Output:
["This is an",
"example of text",
"justification. "
]

CODE:

```
def full_justify(words, maxWidth):
    result = []
    current_line = []
    current_length = 0
    for word in words:
        if current_length + len(word) + len(current_line) > maxWidth:
            # Justify the current line
            for i in range(maxWidth - current_length):
                current_line[i % (len(current_line) - 1 or 1)] += ' '
            result.append(''.join(current_line))
            current_line = []
            current_length = 0
                current_line.append(word)
        current_length += len(word)
    result.append(' '.join(current_line).ljust(maxWidth))
    return result
if __name__ == "__main__":
    words = ["This", "is", "an", "example", "of", "text", "justification."]
    maxWidth = 16
    justified_text = full_justify(words, maxWidth)
    for line in justified_text:
        print(f'"{line}"')
```

OUTPUT:

"This is an"

"example of text"

"justification. "

11) Design a special dictionary that searches the words in it by a prefix and a suffix. Implement the WordFilter class: WordFilter(string[] words) Initializes the object with the words in the dictionary.f(string pref, string suff) Returns the index of the word in the dictionary, which has the prefix pref and the suffix suff. If there is more than one valid index, return the largest of them. If there is no such word in the dictionary, return -1.
Example 1:
Input
["WordFilter", "f"]
[[["apple"]], ["a", "e"]]
Output
[null, 0]

CODE:

```
class WordFilter:
    def __init__(self, words):
        self.words = words
        self.prefix_map = {}
        for index, word in enumerate(words):
            for i in range(len(word) + 1):  # Include all prefixes
                prefix = word[:i]
                if prefix not in self.prefix_map:
                    self.prefix_map[prefix] = []
                self.prefix_map[prefix].append(index)
    def f(self, pref, suff):
        suffix = suff[::-1]
        if pref not in self.prefix_map:
            return -1
                indices = self.prefix_map[pref]
        for index in reversed(indices):
            if self.words[index].endswith(suff):
                return index
        return -1
if __name__ == "__main__":
    # Initialize WordFilter with a list of words
    word_filter = WordFilter(["apple"])
 print(word_filter.f("a", "e"))
```

OUTPUT:

0