DAA 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.

```
def num_ways_to_roll(num_sides, num_dice, target):
    dp = [[0] * (target + 1) for _ in range(num_dice + 1)]
    dp[0][0] = 1
    for dice in range(1, num_dice + 1):
        for sum_value in range(1, target + 1):
            dp[dice][sum_value] = 0
            for face in range(1, num_sides + 1):
                if sum_value - face >= 0:
                    dp[dice][sum_value] += dp[dice - 1][sum_value - face]
    return dp[num_dice][target]
num_sides = 6
num_dice = 2
target = 7
print(f"Number of ways to roll {num_dice} dice with {num_sides} sides to get a sum of {target} is:", num_ways_to_roll(num_sides, num_dice, target))
```

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.

```
def min_time_to_process(n, time1, time2, transfer):
    dp = [[0] * 2 for _ in range(n)]
    dp[0][0] = time1[0]
    dp[0][1] = time2[0]
    for i in range(1, n):
        dp[i][0] = min(dp[i-1][0] + time1[i], dp[i-1][1] + transfer[i-1] + time1[i])
        dp[i][1] = min(dp[i-1][1] + time2[i], dp[i-1][0] + transfer[i-1] + time2[i])
    return min(dp[n-1][0], dp[n-1][1])
```

```
n = 4

time1 = [4, 5, 3, 2]

time2 = [2, 10, 1, 4]

transfer = [7, 9, 3]

print(f"Minimum time to process the product is: {min_time_to_process(n, time1, time2, transfer)}")
```

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

```
def min_production_time(num_stations, time, transfer):
    dp = [[float('inf')] * 3 for _ in range(num_stations)]
    for line in range(3):
        dp[0][line] = time[line][0]
    for i in range(1, num_stations):
        for current_line in range(3):
            min_time = float('inf')
            for previous_line in range(3):
                if previous_line == current_line:
                    min_time = min(min_time, dp[i-1][previous_line] + time[current_line][i])
                else:
                    min_time = min(min_time, dp[i-1][previous_line] +
transfer[previous_line][current_line] + time[current_line][i])
            dp[i][current_line] = min_time
    return min(dp[num_stations-1])
num_stations = 3
time = [
    [4, 5, 3],  # Processing times for Line 1
    [2, 10, 1], # Processing times for Line 2
    [3, 6, 7]   # Processing times for Line 3
]
```

```
transfer = [
    [0, 7, 9],  # Transfer times from Line 1 to Line 2 and Line 3
    [7, 0, 8],  # Transfer times from Line 2 to Line 1 and Line 3
    [9, 8, 0]   # Transfer times from Line 3 to Line 1 and Line 2
]
print(f"Minimum production time is: {min_production_time(num_stations, time, transfer)}")
```

4. Write a c program to find the minimum path distance by using matrix form.

```
#include <stdio.h>
#include <limits.h>
#define INF 99999
#define V 4
void floydWarshall(int graph[][V]) {
    int dist[V][V];
    int i, j, k;
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
        }
    }
    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
    printf("Shortest distances between every pair of vertices:\n");
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
```

```
            if (dist[i][j] == INF) {
                printf("INF ");
            } else {
                printf("%d ", dist[i][j]);
            }
        }
        printf("\n");
    }
}

int main()
    int graph[V][V] = {
        {0, 3, INF, 7},
        {8, 0, 2, INF},
        {5, INF, 0, 1},
        {2, INF, INF, 0}
    };
    floydWarshall(graph);
    return 0;
}
```

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.

```
import itertools
def tsp(dist):
    n = len(dist)
    # dp[mask][i] = minimum cost to visit all cities in `mask` ending at city `i`
    dp = [[float('inf')] * n for _ in range(1 << n)]
    dp[1][0] = 0  # Start at city 0
    for mask in range(1 << n):
        for u in range(n):
            if mask & (1 << u):
```

```
            for v in range(n):
                if not (mask & (1 << v)):
                    new_mask = mask | (1 << v)
                    dp[new_mask][v] = min(dp[new_mask][v], dp[mask][u] + dist[u][v])
    min_cost = float('inf')
    for i in range(1, n):
        min_cost = min(min_cost, dp[(1 << n) - 1][i] + dist[i][0])
    return min_cost
def main():
    dist = [
        [0, 10, 15, 20, 25],  # Distances from A to other cities
        [10, 0, 35, 25, 30],  # Distances from B to other cities
        [15, 35, 0, 30, 20],  # Distances from C to other cities
        [20, 25, 30, 0, 15],  # Distances from D to other cities
        [25, 30, 20, 15, 0]   # Distances from E to other cities
    ]
    result = tsp(dist)
    print(f"The minimum cost to visit all cities is: {result}")
if __name__ == "__main__":
    main()
```

6. Given a string s, return the longest palindromic substring in S.

```
def longest_palindromic_substring(s: str) -> str:
    if not s:
        return ""
    def expand_around_center(left: int, right: int) -> str:
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return s[left + 1:right]
    longest = ""
    for i in range(len(s)):
```

```
        odd_palindrome = expand_around_center(i, i)
        even_palindrome = expand_around_center(i, i + 1)
        longest = max(longest, odd_palindrome, even_palindrome, key=len)
    return longest
s = "babad"
print(f"Longest palindromic substring is: {longest_palindromic_substring(s)}")
```

7. Given a string s, find the length of the longest substring without repeating characters.

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

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.

```
def word_break(s: str, wordDict: list[str]) -> bool:
    word_set = set(wordDict)  # Convert list to set for O(1) lookups
    dp = [False] * (len(s) + 1)  # DP array
    dp[0] = True  # Base case: empty string can be segmented
    for i in range(1, len(s) + 1):
        for j in range(i):
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
```

```
            break  # No need to check further if we found a valid segmentation
    return dp[len(s)]
s = "leetcode"
wordDict = ["leet", "code"]
print(f"Can the string '{s}' be segmented? {word_break(s, wordDict)}")
```

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}

```
def word_break(s: str, wordDict: set) -> bool:
    n = len(s)
    dp = [False] * (n + 1)
    dp[0] = True  # Base case: an empty string can always be segmented
    for i in range(1, n + 1):
        for j in range(i):
            if dp[j] and s[j:i] in wordDict:
                dp[i] = True
                break  # No need to check further if we found a valid segmentation
    return dp[n]
s = "ilikeicecream"
wordDict = {"i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream", "icecream", "man", "go", "mango"}
print(f"Can the string '{s}' be segmented? {word_break(s, wordDict)}")
```

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.

```
def full_justify(words, maxWidth):
    def justify_line(line, width, is_last_line=False):
        if is_last_line or len(line) == 1:
```

```
            return ' '.join(line).ljust(width)
        total_chars = sum(len(word) for word in line)
        spaces = width - total_chars
        gaps = len(line) - 1
        if gaps > 0:
            even_space = spaces // gaps
            extra_space = spaces % gaps
            line_str = ''
            for i in range(gaps):
                line_str += line[i] + ' ' * (even_space + (1 if i < extra_space else 0))
            line_str += line[-1]
            return line_str
    result = []
    current_line = []
    current_length = 0
    for word in words:
        if current_length + len(word) + len(current_line) > maxWidth:
            result.append(justify_line(current_line, maxWidth))
            current_line = []
            current_length = 0
        current_line.append(word)
        current_length += len(word)
    if current_line:
        result.append(justify_line(current_line, maxWidth, is_last_line=True))
    return result
words = ["This", "is", "an", "example", "of", "text", "justification."]
maxWidth = 16
formatted_text = full_justify(words, maxWidth)
for line in formatted_text:
    print(f"'{line}'")
```

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.

```
class WordFilter:
    def __init__(self, words):
        self.word_map = {}
        for index, word in enumerate(words):
            for i in range(len(word) + 1):
                for j in range(len(word) + 1):
                    prefix = word[:i]
                    suffix = word[j:]
                    self.word_map[(prefix, suffix)] = index
    def f(self, pref, suff):
        return self.word_map.get((pref, suff), -1)
words = ["apple", "banana", "carrot"]
wf = WordFilter(words)
print(wf.f("a", "e"))
print(wf.f("b", "a"))
print(wf.f("c", "t"))
print(wf.f("b", "c"))
```