**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.  
  
**program:-**

num\_sides = 6

num\_dice = 2

target = 7

dp = [[0] \* (target + 1) for \_ in range(num\_dice + 1)]

dp[0][0] = 1

# Fill the DP table

for dice in range(1, num\_dice + 1):

    for sum\_value in range(1, target + 1):

        for face in range(1, num\_sides + 1):

            if sum\_value - face >= 0:

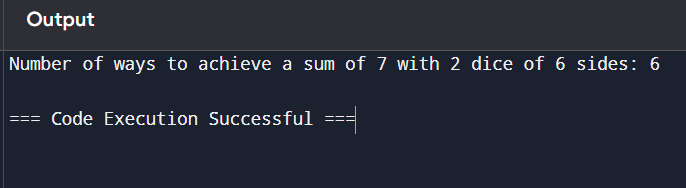
                dp[dice][sum\_value] += dp[dice - 1][sum\_value - face]

# Result

result = dp[num\_dice][target]

print(f"Number of ways to achieve a sum of {target} with {num\_dice} dice of {num\_sides} sides: {result}")

**output:-**



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.

**Program:-**

# Input values

n = 4  # Number of stations

a1 = [7, 9, 3, 4]  # Time taken at each station on assembly line 1

a2 = [8, 5, 6, 4]  # Time taken at each station on assembly line 2

t1 = [2, 1, 3]     # Transfer time from line 1 to line 2 after each station

t2 = [2, 2, 1]     # Transfer time from line 2 to line 1 after each station

e1 = 2             # Entry time to assembly line 1

e2 = 4             # Entry time to assembly line 2

x1 = 3             # Exit time from assembly line 1

x2 = 2             # Exit time from assembly line 2

# Initialize dp arrays

dp1 = [0] \* (n + 1)  # Time for assembly line 1

dp2 = [0] \* (n + 1)  # Time for assembly line 2

# Base case

dp1[0] = e1 + a1[0]  # Time to start on line 1

dp2[0] = e2 + a2[0]  # Time to start on line 2

for i in range(1, n):

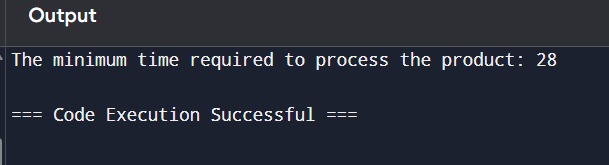
    dp1[i] = min(dp1[i - 1] + a1[i], dp2[i - 1] + t2[i - 1] + a1[i])

    dp2[i] = min(dp2[i - 1] + a2[i], dp1[i - 1] + t1[i - 1] + a2[i])

result = min(dp1[n - 1] + x1, dp2[n - 1] + x2)

print(f"The minimum time required to process the product: {result}")

**output:-**



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

**Program:-**

# Input values

n = 3  # Number of stations

line1\_times = [5, 9, 3]  # Time taken at each station on assembly line 1

line2\_times = [6, 8, 4]  # Time taken at each station on assembly line 2

line3\_times = [7, 6, 5]  # Time taken at each station on assembly line 3

# Transfer times matrix

transfer\_times = [

    [0, 2, 3],  # Transfer from Line 1

    [2, 0, 4],  # Transfer from Line 2

    [3, 4, 0]   # Transfer from Line 3

]

# Initialize DP array

dp = [[0] \* 3 for \_ in range(n)]  # dp[i][j]: time to finish station i on line j

# Base case for the first station

dp[0][0] = line1\_times[0]

dp[0][1] = line2\_times[0]

dp[0][2] = line3\_times[0]

# Fill the DP table

for i in range(1, n):

    dp[i][0] = min(

        dp[i-1][0] + line1\_times[i],  # Continue on Line 1

        dp[i-1][1] + transfer\_times[1][0] + line1\_times[i],  # Transfer from Line 2

        dp[i-1][2] + transfer\_times[2][0] + line1\_times[i]   # Transfer from Line 3

    )

    dp[i][1] = min(

        dp[i-1][0] + transfer\_times[0][1] + line2\_times[i],  # Transfer from Line 1

        dp[i-1][1] + line2\_times[i],  # Continue on Line 2

        dp[i-1][2] + transfer\_times[2][1] + line2\_times[i]   # Transfer from Line 3

    )

    dp[i][2] = min(

        dp[i-1][0] + transfer\_times[0][2] + line3\_times[i],  # Transfer from Line 1

        dp[i-1][1] + transfer\_times[1][2] + line3\_times[i],  # Transfer from Line 2

        dp[i-1][2] + line3\_times[i]   # Continue on Line 3

    )

# Considering dependencies:

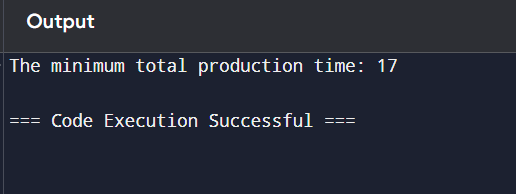
# We need to calculate the total time considering the dependencies as well.

total\_time = min(dp[n-1])  # Time to finish all stations considering the last station on any line

# Output the result

print(f"The minimum total production time: {total\_time}")

**output:-**



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

**Program:-**

import itertools

# Distance matrix

dist = [

    [0, 10, 15, 20],

    [10, 0, 35, 25],

    [15, 35, 0, 30],

    [20, 25, 30, 0]

]

# Number of cities

N = len(dist)

# Function to calculate the minimum path distance

def calculate\_min\_path(dist):

    min\_path = float('inf')  # Start with a very large number

 # Generate all permutations of cities

    for perm in itertools.permutations(range(N)):

        # Calculate the total distance for this permutation

        path\_length = 0

        for i in range(N):

            path\_length += dist[perm[i]][perm[(i + 1) % N]]  # Wrap around to form a loop

       # Update minimum path length if found a new minimum

        if path\_length < min\_path:

            min\_path = path\_length

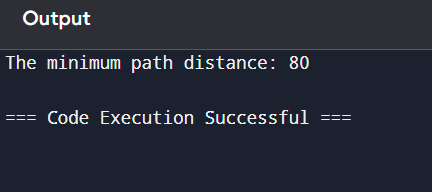
    return min\_path

# Calculate the minimum path distance

min\_distance = calculate\_min\_path(dist)

print(f"The minimum path distance: {min\_distance}")

**output:-**



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.

**Program:-**

import itertools

# Distance matrix for 5 cities: A, B, C, D, E

cities = ['A', 'B', 'C', 'D', 'E']

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,

}

# Fill in the distance matrix for symmetric distances

for city1 in cities:

    for city2 in cities:

        if city1 != city2 and (city2, city1) not in distances:

            distances[(city2, city1)] = distances[(city1, city2)]

# Generate all permutations of cities (except the starting point)

shortest\_route = None

min\_distance = float('inf')

# Calculate distances for each permutation

for perm in itertools.permutations(cities):

    # Start and end at the same city (A)

    if perm[0] == 'A':

        route\_distance = 0

        for i in range(len(perm)):

            route\_distance += distances[(perm[i], perm[(i + 1) % len(perm)])]

        # Check if this route is shorter

        if route\_distance < min\_distance:

            min\_distance = route\_distance

            shortest\_route = perm

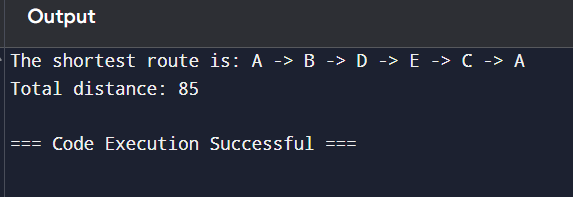
# Output the shortest route and its total distance

route\_str = " -> ".join(shortest\_route) + " -> A"

print(f"The shortest route is: {route\_str}")

print(f"Total distance: {min\_distance}")

**output:-**



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

Example 1:

Input: s = "babad" Output: "bab"

**Program:-**

# Input string

s = "babad"

# Initialize variables to track the start and end of the longest palindrome

start, end = 0, 0

# Expand around centers

for i in range(len(s)):

    # Check for odd-length palindromes (single character center)

    left1, right1 = i, i

    while left1 >= 0 and right1 < len(s) and s[left1] == s[right1]:

        if right1 - left1 > end - start:

            start, end = left1, right1

        left1 -= 1

        right1 += 1

    # Check for even-length palindromes (two character center)

    left2, right2 = i, i + 1

    while left2 >= 0 and right2 < len(s) and s[left2] == s[right2]:

        if right2 - left2 > end - start:

            start, end = left2, right2

        left2 -= 1

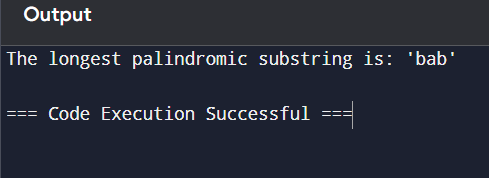
        right2 += 1

# Extract the longest palindromic substring

longest\_palindrome = s[start:end + 1]

print(f"The longest palindromic substring is: '{longest\_palindrome}'")

**output:-**



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

Example 1: Input: s = "abcabcbb" Output: 3

**Program:-**

# Input string

s = "abcabcbb"

# Initialize variables

start = 0  # Start index of the current substring

max\_length = 0  # Maximum length of substring found

char\_index\_map = {}  # Dictionary to store the last index of each character

# Iterate through the string

for end in range(len(s)):

    current\_char = s[end]

    # If the character is already in the map and its index is within the current window

    if current\_char in char\_index\_map and char\_index\_map[current\_char] >= start:

        start = char\_index\_map[current\_char] + 1  # Move the start to the right of the last occurrence

    # Update the last index of the current character

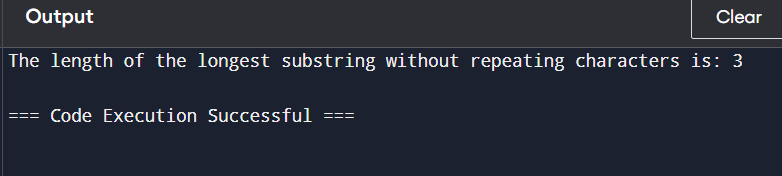
    char\_index\_map[current\_char] = end

    # Update the maximum length

    max\_length = max(max\_length, end - start + 1)

print(f"The length of the longest substring without repeating characters is: {max\_length}")

**output:-**



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"]

**Program:-**

# Input string and word dictionary

s = "leetcode"

wordDict = ["leet", "code"]

# Convert wordDict to a set for faster lookup

word\_set = set(wordDict)

# Initialize the DP array

n = len(s)

dp = [False] \* (n + 1)

dp[0] = True  # Base case: empty string can be segmented

for i in range(1, n + 1):

    for j in range(i):

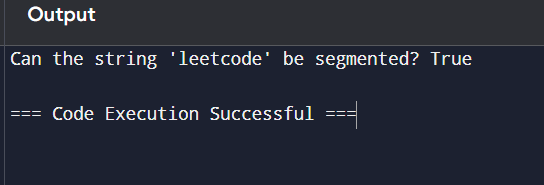
        if dp[j] and s[j:i] in word\_set:

            dp[i] = True

            break

print(f"Can the string '{s}' be segmented? {dp[n]}")

**output:-**



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}

**Program:-**

# Input string and word dictionary

input\_string = "ilike"

word\_dict = {"i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream", "icecream", "man", "go", "mango"}

# Initialize the DP array

n = len(input\_string)

dp = [False] \* (n + 1)

dp[0] = True  # Base case: empty string can be segmented

# Fill the DP array

for i in range(1, n + 1):

    for j in range(i):

        if dp[j] and input\_string[j:i] in word\_dict:

            dp[i] = True

            break

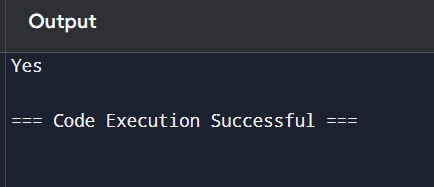
if dp[n]:

    print("Yes")

else:

    print("No")

**output:-**



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.

**Program:-**

# Input words and maximum width

words = ["This", "is", "an", "example", "of", "text", "justification."]

maxWidth = 16

# Initialize variables

result = []

current\_line = []

line\_length = 0

# Process each word

for word in words:

    # Check if adding the next word exceeds the maxWidth

    if line\_length + len(word) + len(current\_line) > maxWidth:

        # Justify the current line

        for i in range(maxWidth - line\_length):

            # Distribute spaces

            current\_line\_index = i % (len(current\_line) - 1 or 1)

            current\_line[current\_line\_index] += ' '

        # Join the line and append to result

        result.append(''.join(current\_line))

        # Reset for the next line

        current\_line = []

        line\_length = 0

    # Add the word to the current line

    current\_line.append(word)

    line\_length += len(word)

# Handle the last line (left-justified)

last\_line = ' '.join(current\_line)

last\_line += ' ' \* (maxWidth - len(last\_line))  # Fill with spaces if needed

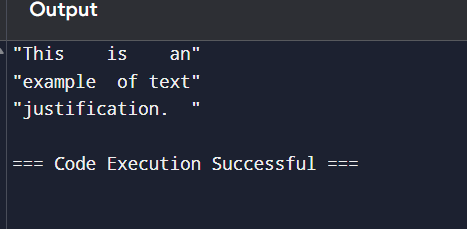
result.append(last\_line)

# Output the result

for line in result:

    print(f'"{line}"')

**output:-**



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.

**Program:-**

class WordFilter:

    def \_\_init\_\_(self, words):

        self.words = words

        self.prefix\_suffix\_map = {}

        # Preprocess the words to create a mapping of (prefix, suffix) to indices

        for index, word in enumerate(words):

            # Create all possible prefixes and suffixes

            for i in range(len(word) + 1):

                prefix = word[:i]

                for j in range(len(word) + 1):

                    suffix = word[j:]

                    # Store the index for this (prefix, suffix) pair

                    if (prefix, suffix) not in self.prefix\_suffix\_map:

                        self.prefix\_suffix\_map[(prefix, suffix)] = index

                    else:

                        # We only want the largest index

                        self.prefix\_suffix\_map[(prefix, suffix)] = max(self.prefix\_suffix\_map[(prefix, suffix)], index)

    def f(self, pref, suff):

        # Return the index of the word with the specified prefix and suffix

        return self.prefix\_suffix\_map.get((pref, suff), -1)

# Example usage:

word\_filter = WordFilter(["apple"])

print(word\_filter.f("a", "e"))  # Output: 0

print(word\_filter.f("a", "l"))  # Output: -1

**output:-**

