STUDENT NAME: M.RAVI SAI VINAY

REG.NO:192311035.

COURSE CODE:CSA0689.

SUB.NAME:DESIGN AND ANALYSIS OF ALGORITHM.

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 Test Case 2: Number of ways to reach sum 10: 27.

```python
def dice_throw(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_value in range(1, num_sides + 1):
                if sum_value - face_value >= 0:
                    dp[dice][sum_value] += dp[dice - 1][sum_value - face_value]
    return dp[num_dice][target]

print(f"Test Case 1: Number of ways to reach sum 7: {dice_throw(6, 2, 7)}")
print(f"Test Case 2: Number of ways to reach sum 10: {dice_throw(4, 3, 10)}")
```

```
Test Case 1: Number of ways to reach sum 7: 6
Test Case 2: Number of ways to reach sum 10: 6

=== Code Execution Successful ===
```

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.

```python
def minimum_time(n, a1, a2, t1, t2, e1, e2, x1, x2):
    dp1 = [0] * n
    dp2 = [0] * n
    dp1[0] = e1 + a1[0]
    dp2[0] = e2 + a2[0]
    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])

    final_time_line_1 = dp1[-1] + x1
    final_time_line_2 = dp2[-1] + x2
    return min(final_time_line_1, final_time_line_2)


n = 4
a1 = [4, 5, 3, 2]
a2 = [2, 10, 1, 4]
t1 = [0, 7, 4, 5]
t2 = [0, 9, 2, 8]
e1 = 10
e2 = 12
x1 = 18
x2 = 7
result = minimum_time(n, a1, a2, t1, t2, e1, e2, x1, x2)
print("Minimum time required to process the product:", result)
```

```
Minimum time required to process the product: 36

=== Code Execution Successful ===
```

3. 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).

```python
def minimum_production_time(L1, L2, L3, T, dependencies):
    n = len(L1)
    dp = [[0] * n for _ in range(3)]

    dp[0][0] = L1[0]
    dp[1][0] = L2[0]
    dp[2][0] = L3[0]
    for i in range(1, n):
        dp[0][i] = min(dp[0][i-1] + L1[i],
                       dp[1][i-1] + T[1][0] + L1[i],
                       dp[2][i-1] + T[2][0] + L1[i])

        dp[1][i] = min(dp[1][i-1] + L2[i],
                       dp[0][i-1] + T[0][1] + L2[i],
                       dp[2][i-1] + T[2][1] + L2[i])

        dp[2][i] = min(dp[2][i-1] + L3[i],
                       dp[0][i-1] + T[0][2] + L3[i],
                       dp[1][i-1] + T[1][2] + L3[i])

    result = min(dp[0][-1], dp[1][-1], dp[2][-1])
    return result

L1 = [5, 9, 3]
L2 = [6, 8, 4]
L3 = [7, 6, 5]
T = [
    [0, 2, 3],
    [2, 0, 4],
    [3, 4, 0]
]
dependencies = [(0, 1), (1, 2)]
result = minimum_production_time(L1, L2, L3, T, dependencies)
print("Minimum production time:", result)
```

Output
```
Minimum production time: 17

=== Code Execution Successful ===
```

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 2) {0,10,10,10} {10,0,10,10} {10,10,0,10} {10,10,10,0} Output: 40 3) {0,1,2,3} {1,0,4,5} {2,4,0,6} {3,5,6,0} Output: 12

```python
1  import numpy as np
2  def floyd_warshall(graph):
3      V = len(graph)
4      dist = np.array(graph)
5      for k in range(V):
6          for i in range(V):
7              for j in range(V):
8                  if dist[i][k] != float('inf') and dist[k][j] != float('inf'):
9                      dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
10     def tsp(dp, mask, pos):
11         if mask == (1 << V) - 1:
12             return dp[pos][0] if dp[pos][0] != float('inf') else float('inf')
13
14         if dp[mask][pos] != float('inf'):
15             return dp[mask][pos]
16
17         ans = float('inf')
18         for i in range(V):
19             if mask & (1 << i) == 0:
20                 ans = min(ans, dist[pos][i] + tsp(dp, mask | (1 << i), i))
21
22         dp[mask][pos] = ans
23         return ans
24
25     dp = [[float('inf')] * V for _ in range(1 << V)]
26     dp[1][0] = 0
27
28     min_path_distance = tsp(dp, 1, 0)
29     return min_path_distance
30  test_case_1 = [
31      [0, 10, 15, 20],
32      [10, 0, 35, 25],
33      [15, 35, 0, 30],
34      [20, 25, 30, 0]
35  ]
36
37  test_case_2 = [
38      [0, 10, 10, 10],
39      [10, 0, 10, 10],
40      [10, 10, 0, 10],
41      [10, 10, 10, 0]
42  ]
43
44  test_case_3 = [
45      [0, 1, 2, 3],
46      [1, 0, 4, 5],
47      [2, 4, 0, 6],
48      [3, 5, 6, 0]
49  ]
50
51  print("Test Case 1:")
52  print("Minimum path distance:", floyd_warshall(test_case_1))
53
54  print("Test Case 2:")
55  print("Minimum path distance:", floyd_warshall(test_case_2))
56
57  print("Test Case 3:")
58  print("Minimum path distance:", floyd_warshall(test_case_3))
59
```

Output:
```
Test Case 1:
Minimum path distance: 0
Test Case 2:
Minimum path distance: 0
Test Case 3:
Minimum path distance: 0

=== Code Execution Successful ===
```

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.

```
from itertools import permutations

def calculate_distance(permutation, distances):
    total_distance = 0
    for i in range(len(permutation)):
        total_distance += distances[permutation[i]][permutation[(i + 1) % len(permutation)]]
    return total_distance

def traveling_salesperson_problem(distances):
    cities = list(distances.keys())
    min_distance = float('inf')
    best_route = []

    for perm in permutations(cities):
        current_distance = calculate_distance(perm, distances)
        if current_distance < min_distance:
            min_distance = current_distance
            best_route = perm

    return best_route, min_distance


distances = {
    'A': {'B': 10, 'C': 15, 'D': 20, 'E': 25},
    'B': {'A': 10, 'C': 35, 'D': 25, 'E': 30},
    'C': {'A': 15, 'B': 35, 'D': 30, 'E': 20},
    'D': {'A': 20, 'B': 25, 'C': 30, 'E': 15},
    'E': {'A': 25, 'B': 30, 'C': 20, 'D': 15}
}

best_route, min_distance = traveling_salesperson_problem(distances)
print(f"Shortest route: {' -> '.join(best_route)}")
print(f"Total distance: {min_distance}")
```

Output:
```
Shortest route: A -> B -> D -> E -> C
Total distance: 85

=== Code Execution Successful ===
```

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. Example 2: Input: s = "cbbd" Output: "bb" Constraints: ● 1 <= s.length <= 1000 ● s consist of only digits and English letters.

```
1  def longest_palindromic_substring(s):
2      def expand_around_center(left, right):
3          while left >= 0 and right < len(s) and s[left] == s[right]:
4              left -= 1
5              right += 1
6          return s[left + 1:right]
7
8      longest = ""
9
10     for i in range(len(s)):
11         odd_palindrome = expand_around_center(i, i)
12         if len(odd_palindrome) > len(longest):
13             longest = odd_palindrome
14
15         even_palindrome = expand_around_center(i, i + 1)
16         if len(even_palindrome) > len(longest):
17             longest = even_palindrome
18     return longest
19
20 print("Example 1:")
21 print("Input: s = 'babad'")
22 print("Output:", longest_palindromic_substring("babad"))
23
24 print("Example 2:")
25 print("Input: s = 'cbbd'")
26 print("Output:", longest_palindromic_substring("cbbd"))
27
```

Output:
```
Example 1:
Input: s = 'babad'
Output: bab
Example 2:
Input: s = 'cbbd'
Output: bb

=== Code Execution Successful ===
```

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. Example 2: Input: s = "bbbbb" Output: 1 Explanation: The answer is "b", with the length of 1. Example 3: Input: s = "pwwkew" Output: 3 Explanation: The answer is "wke", with the length of 3. Notice that the answer must be a substring, "pwke" is a subsequence and not a substring. Constraints: ● 0 <= s.length <= 5 * 104 ● s consists of English letters, digits, symbols and spaces.

```python
def length_of_longest_substring(s):
    char_set = set()
    start = 0
    max_length = 0

    for end in range(len(s)):
        while s[end] in char_set:
            char_set.remove(s[start])
            start += 1
        char_set.add(s[end])
        max_length = max(max_length, end - start + 1)

    return max_length

print("Example 1:")
print("Input: s = 'abcabcbb'")
print("Output:", length_of_longest_substring("abcabcbb"))

print("Example 2:")
print("Input: s = 'bbbbb'")
print("Output:", length_of_longest_substring("bbbbb"))

print("Example 3:")
print("Input: s = 'pwwkew'")
print("Output:", length_of_longest_substring("pwwkew"))
```

```
Example 1:
Input: s = 'abcabcbb'
Output: 3
Example 2:
Input: s = 'bbbbb'
Output: 1
Example 3:
Input: s = 'pwwkew'
Output: 3

=== Code Execution Successful ===
```

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 Explanation: Return true because "leetcode" can be segmented as "leet code". Example 2: Input: s = "applepenapple", wordDict = ["apple","pen"] Output: true Explanation: Return true because "applepenapple" can be segmented as "apple pen apple". Note that you are

allowed to reuse a dictionary word. Example 3: Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"] Output: false.

```python
1  def wordBreak(s, wordDict):
2      word_set = set(wordDict)
3      dp = [False] * (len(s) + 1)
4      dp[0] = True
5      for i in range(1, len(s) + 1):
6          for j in range(i):
7              if dp[j] and s[j:i] in word_set:
8                  dp[i] = True
9                  break
10     return dp[len(s)]
11
12 print(wordBreak("leetcode", ["leet","code"]))
13 print(wordBreak("applepenapple", ["apple","pen"]))
14 print(wordBreak("catsandog", ["cats","dog","sand","and","cat"]))
```

Output:
```
True
True
False

=== Code Execution Successful ===
```

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".

```python
1  def wordBreak(s, wordDict):
2      word_set = set(wordDict)
3      dp = [False] * (len(s) + 1)
4      dp[0] = True
5
6      for i in range(1, len(s) + 1):
7          for j in range(i):
8              if dp[j] and s[j:i] in word_set:
9                  dp[i] = True
10                 break
11
12     return "Yes" if dp[len(s)] else "No"
13
14 print(wordBreak("ilike", ["i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream", "icecream", "man", "go", "mango"]))
15 print(wordBreak("ilikesamsung", ["i", "like", "sam", "sung", "samsung", "mobile", "ice", "cream", "icecream", "man", "go",
       "mango"]))
```

Output:
```
Yes
Yes

=== Code Execution Successful ===
```

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. "] Example 2: Input: words = ["What","must","be","acknowledgment","shall","be"], maxWidth = 16 Output: ["What must be", "acknowledgment ", "shall be "] Explanation: Note that the last line is "shall be " instead of "shall be", because the last line must be left-justified instead of fully-justified. Note that the second line is also left-justified because it contains only one word.

```python
def fullJustify(words, maxWidth):
    def justify_line(line_words, line_length, is_last_line=False):
        if len(line_words) == 1 or is_last_line:
            return ' '.join(line_words).ljust(maxWidth)
        total_spaces = maxWidth - line_length
        spaces_between_words = len(line_words) - 1
        even_space = total_spaces // spaces_between_words
        extra_space = total_spaces % spaces_between_words
        line = ''
        for i in range(spaces_between_words):
            line += line_words[i]
            line += ' ' * (even_space + (1 if i < extra_space else 0))
        line += line_words[-1]
        return line

    res = []
    current_words = []
    current_length = 0

    for word in words:
        if current_length + len(word) + len(current_words) > maxWidth:
            res.append(justify_line(current_words, current_length))
            current_words = [word]
            current_length = len(word)
        else:
            current_words.append(word)
            current_length += len(word)

    res.append(justify_line(current_words, current_length, is_last_line=True))
    return res

print(fullJustify(["This", "is", "an", "example", "of", "text", "justification."], 16))
print(fullJustify(["What", "must", "be", "acknowledgment", "shall", "be"], 16))
```

```
['This    is    an', 'example of text', 'justification.  ']
['What    must    be', 'acknowledgment ', 'shall be        ']

=== Code Execution Successful ===
```

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] Explanation WordFilter wordFilter = new WordFilter(["apple"]); wordFilter.f("a", "e"); // return 0, because the word at index 0 has prefix = "a" and suffix = "e".

```python
class WordFilter:
    def __init__(self, words):
        self.prefix_suffix_map = {}
        for index, word in enumerate(words):
            word_length = len(word)
            for i in range(word_length):
                prefix = word[:i + 1]
                for j in range(word_length):
                    suffix = word[j:]
                    self.prefix_suffix_map[(prefix, suffix)] = index

    def f(self, pref, suff):
        return self.prefix_suffix_map.get((pref, suff), -1)


wordFilter = WordFilter(["apple"])
print(wordFilter.f("a", "e"))
```

Output

```
0

=== Code Execution Successful ===
```