NAME : TEJASWINI M

REG NO. : 192411079

COURSE CODE : CSA0613

COURSE NAME : DESIGN AND ANALYSIS OF ALGORITHMS FOR OPTIMAL APPLICATIONS

SLOT : A

TOPIC 1 INTRODUCTION

1. **Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an empty string "". A string is palindromic if it reads the same forward and backward.**

AIM:

To find and display the first palindromic string in a given array of strings. If no palindrome exists, display an empty string.

ALGORITHM:

1. Start
2. Read the list of strings words
3. For each string in the list:
   - Reverse the string
   - Compare the original string with its reverse
   - If both are equal, it is a palindrome
   - Print the string and stop the program
4. If no palindromic string is found after checking all elements, print an empty string ""
5. Stop

CODE:

```
def first_palindrome(words):
    for word in words:
        if word == word[::-1]:
```

```
            return word
    return ""
words = ["abc", "car", "ada", "racecar", "cool"]
result = first_palindrome(words)
print("First Palindromic String:", result)
```

INPUT:

```
words = ["abc", "car", "ada", "racecar", "cool"]
```

OUTPUT:

```
First Palindromic String: ada
```

2. **You are given two integer arrays nums1 and nums2 of sizes n and m, respectively. Calculate the following values: answer1 : the number of indices i such that nums1[i] exists in nums2. answer2 : the number of indices i such that nums2[i] exists in nums1 Return [answer1,answer2].**

AIM:

To count how many elements of one array exist in the other array and return the counts as a list.

ALGORITHM:

1. Start
2. Read two integer arrays nums1 and nums2
3. Convert both arrays into sets for fast lookup
4. Initialize answer1 = 0 and answer2 = 0
5. For each element in nums1:
   - If the element exists in nums2, increment answer1
6. For each element in nums2:
   - If the element exists in nums1, increment answer2
7. Print [answer1, answer2]
8. Stop

CODE:

```
def count_common_indices(nums1, nums2):
```

```
    set1 = set(nums1)
    set2 = set(nums2)

    answer1 = 0
    answer2 = 0

    for num in nums1:
        if num in set2:
            answer1 += 1

    for num in nums2:
        if num in set1:
            answer2 += 1

    return [answer1, answer2]
nums1 = [1, 2, 3]
nums2 = [2, 3, 4]
result = count_common_indices(nums1, nums2)
print("Result:", result)
```

INPUT:

```
nums1 = [1, 2, 3]
nums2 = [2, 3, 4]
```

OUTPUT:

```
Result: [2, 2]
```

3. **You are given a 0-indexed integer array nums. The distinct count of a subarray of nums is defined as: Let nums[i..j] be a subarray of nums consisting of all the indices from i to j such that 0 <= i <= j < nums.length. Then the number of distinct values in nums[i..j] is called the distinct count of nums[i..j]. Return the sum of the squares of distinct counts of**

all subarrays of nums. A subarray is a contiguous non-empty sequence of elements within an array.

AIM:

To calculate the sum of squares of distinct element counts for all possible non-empty subarrays of a given integer array.

ALGORITHM:

1. Start
2. Read the integer array nums
3. Initialize total_sum = 0
4. For each starting index i from 0 to n-1:
   - Create an empty set distinct_set
5. For each ending index j from i to n-1:
   - Add nums[j] to distinct_set
   - Find the number of distinct elements using len(distinct_set)
   - Square this value and add it to total_sum
6. After checking all subarrays, print total_sum
7. Stop

CODE:

```
def sum_of_squares_of_distinct(nums):
    total_sum = 0
    n = len(nums)

    for i in range(n):
        distinct_set = set()
        for j in range(i, n):
            distinct_set.add(nums[j])
            count = len(distinct_set)
            total_sum += count * count

    return total_sum
nums = [1, 2, 1]
```

```
result = sum_of_squares_of_distinct(nums)
print("Sum of squares of distinct counts:", result)
```

INPUT:

nums = [1, 2, 1]

OUTPUT:

Sum of squares of distinct counts: 15

4. **Given a 0-indexed integer array nums of length n and an integer k, return the number of pairs (i, j) where 0 <= i < j < n, such that nums[i] == nums[j] and (i * j) is divisible by k.**

AIM:

To count the number of index pairs (i, j) such that nums[i] == nums[j] and (i * j) is divisible by k.

ALGORITHM:

1. Start
2. Read the integer array nums and integer k
3. Initialize count = 0
4. Use two loops:
   - Outer loop from i = 0 to n-1
   - Inner loop from j = i+1 to n-1
5. For each pair (i, j):
   - Check if nums[i] == nums[j]
   - Check if (i * j) % k == 0
   - If both conditions are true, increment count
6. After checking all pairs, print count
7. Stop

CODE:

```
def count_pairs(nums, k):
    count = 0
    n = len(nums)
    for i in range(n):
```

```
        for j in range(i + 1, n):
            if nums[i] == nums[j] and (i * j) % k == 0:
                count += 1
    return count
nums = [3, 1, 2, 2, 2, 1, 3]
k = 2
result = count_pairs(nums, k)
print("Number of valid pairs:", result)
```

INPUT:

```
nums = [3, 1, 2, 2, 2, 1, 3]
k = 2
```

OUTPUT:

```
Number of valid pairs: 4
```

5. **Write a program FOR THE BELOW TEST CASES with least time complexity Test Cases: Input: {1, 2, 3, 4, 5} Expected Output: 5**

AIM:

To find the maximum value in an integer array using minimum time complexity.

ALGORITHM:

1. Start
2. Read the array
3. Assume the first element is the maximum
4. Traverse the array once
5. If any element is greater than the current maximum, update it
6. Print the maximum value
7. Stop

CODE:

```
def find_max(nums):
    maximum = nums[0]
```

```
    for num in nums:
        if num > maximum:
            maximum = num
    return maximum
print(find_max([1, 2, 3, 4, 5]))
print(find_max([7, 7, 7, 7, 7]))
print(find_max([-10, 2, 3, -4, 5]))
```

INPUT:

```
print(find_max([1, 2, 3, 4, 5]))
print(find_max([7, 7, 7, 7, 7]))
print(find_max([-10, 2, 3, -4, 5]))
```

OUTPUT:

```
5
7
5
```

6. **You have an algorithm that process a list of numbers. It firsts sorts the list using an efficient sorting algorithm and then finds the maximum element in sorted list. Write the code for the same.**

AIM:

To sort a list of numbers and find the maximum element from the sorted list, handling edge cases like an empty list.

ALGORITHM:

1. Start
2. Read the list of numbers
3. If the list is empty:
   - Print an appropriate message and stop

4. Sort the list using an efficient sorting method
5. The maximum element will be the **last element** of the sorted list
6. Print the maximum element
7. Stop

CODE:

```
def find_max_after_sort(nums):
    # Check for empty list
    if len(nums) == 0:
        return "List is empty"
    nums.sort()
    return nums[-1]
print(find_max_after_sort([]))            # Empty list
print(find_max_after_sort([5]))           # Single element
print(find_max_after_sort([3, 3, 3, 3, 3])) # All elements same
```

INPUT:

```
print(find_max_after_sort([]))
print(find_max_after_sort([5]))
print(find_max_after_sort([3, 3, 3, 3, 3]))
```

OUTPUT:

```
List is empty
5
3
```

7. Write a program that takes an input list of n numbers and creates a new list containing only the unique elements from the original list. What is the space complexity of the algorithm?

AIM:

To create a new list containing only the unique elements from a given list of numbers.

ALGORITHM:

1. Start
2. Read the input list of numbers
3. Create an empty set seen to store elements already encountered
4. Create an empty list unique_list
5. Traverse each element in the input list:
   - If the element is not in seen, add it to seen and append it to unique_list
6. Print unique_list
7. Stop

CODE:

```
def get_unique_elements(nums):
    seen = set()
    unique_list = []
    for num in nums:
        if num not in seen:
            seen.add(num)
            unique_list.append(num)
    return unique_list
print(get_unique_elements([3, 7, 3, 5, 2, 5, 9, 2]))
print(get_unique_elements([-1, 2, -1, 3, 2, -2]))
print(get_unique_elements([1000000, 999999, 1000000]))
```

INPUT:

```
print(get_unique_elements([3, 7, 3, 5, 2, 5, 9, 2]))
print(get_unique_elements([-1, 2, -1, 3, 2, -2]))
print(get_unique_elements([1000000, 999999, 1000000]))
```

OUTPUT:

```
[3, 7, 5, 2, 9]
[-1, 2, 3, -2]
[1000000, 999999]
```

8. **Sort an array of integers using the bubble sort technique. Analyze its time complexity using Big-O notation. Write the code**

AIM:

To sort an array of integers using the Bubble Sort technique and analyze its time complexity using Big-O notation.

ALGORITHM:

1. Start
2. Read the array of integers
3. Repeat for each element of the array:
   - Compare adjacent elements
   - If the current element is greater than the next, swap them
4. After each pass, the largest element moves to the end
5. Continue until the array is sorted
6. Print the sorted array
7. Stop

CODE:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
```

```
            # Swap elements
            arr[j], arr[j + 1] = arr[j + 1], arr[j]

    return arr
nums = [5, 1, 4, 2, 8]
sorted_nums = bubble_sort(nums)
print("Sorted Array:", sorted_nums)
```

INPUT:

nums = [5, 1, 4, 2, 8]

OUTPUT:

Sorted Array: [1, 2, 4, 5, 8]

9. Checks if a given number x exists in a sorted array arr using binary search. Analyze its time complexity using Big-O notation.

AIM:

To check whether a given number exists in an array using Binary Search and analyze its time complexity using Big-O notation.

ALGORITHM:

1. Start
2. Read the array arr and the key x
3. Sort the array (since Binary Search works only on sorted arrays)
4. Set low = 0 and high = len(arr) - 1
5. While low <= high
6. Find mid = (low + high) // 2
7. If arr[mid] == x, print the position and stop
8. If arr[mid] < x, set low = mid + 1
9. Else, set high = mid - 1

10. If the loop ends, print that the element is not found

11. Stop

CODE:

```
def binary_search(arr, x):
    arr.sort()  # Sorting the array first
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return f"Element {x} is found at position {mid + 1}"
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return f"Element {x} is not found"
arr1 = [3, 4, 6, -9, 10, 8, 9, 30]
key1 = 10
print(binary_search(arr1, key1))
```

INPUT:

```
arr1 = [3, 4, 6, -9, 10, 8, 9, 30]
key1 = 10
print(binary_search(arr1, key1))
```

OUTPUT:

Element 10 is found at position 7

10. Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without using any built-in functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

AIM:

To sort an array of integers in ascending order without using any built-in functions, achieving a time complexity of O(n log n) and minimum possible space complexity.

ALGORITHM:

1. Start
2. Read the input array nums
3. If the length of the array is less than or equal to 1, return the array
4. Divide the array into two halves
5. Recursively apply merge sort on both halves
6. Merge the two sorted halves into a single sorted array
7. Return the sorted array
8. Stop

CODE:

```
def merge_sort(nums):
    if len(nums) <= 1:
        return nums
    mid = len(nums) // 2
    left = merge_sort(nums[:mid])
    right = merge_sort(nums[mid:])
    return merge(left, right)
```

```
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while i < len(left):
        result.append(left[i])
        i += 1
    while j < len(right):
        result.append(right[j])
        j += 1
    return result
nums = [5, 2, 3, 1]
sorted_nums = merge_sort(nums)
print(sorted_nums)
```

INPUT:

```
nums = [5, 2, 3, 1]
sorted_nums = merge_sort(nums)
print(sorted_nums)
```

OUTPUT:

[1, 2, 3, 5]

11. Given an m x n grid and a ball at a starting cell, find the number of ways to move the ball out of the grid boundary in exactly N steps.

AIM:

To find the number of ways to move a ball out of an m × n grid boundary in exactly N steps starting from a given cell (i, j).

ALGORITHM:

1. Start
2. Read values m, n, N, i, j
3. Create a 2D DP array dp of size m × n initialized to 0
4. Set dp[i][j] = 1 (starting position)
5. Initialize count = 0
6. For each step from 1 to N
7. Create a new 2D array temp of size m × n initialized to 0
8. For each cell (r, c) in the grid
9. If the move goes out of the grid, add dp[r][c] to count
10. Else, add dp[r][c] to the corresponding new position in temp
11. After N steps, return count
12. Stop

CODE:

```
def find_paths(m, n, N, i, j):
    dp = [[0 for _ in range(n)] for _ in range(m)]
    dp[i][j] = 1
    count = 0
```

```
for _ in range(N):
    temp = [[0 for _ in range(n)] for _ in range(m)]
    for r in range(m):
        for c in range(n):
            if dp[r][c] > 0:
                val = dp[r][c]
                if r - 1 < 0:
                    count += val
                else:
                    temp[r - 1][c] += val
                if r + 1 >= m:
                    count += val
                else:
                    temp[r + 1][c] += val
                if c - 1 < 0:
                    count += val
                else:
                    temp[r][c - 1] += val
                if c + 1 >= n:
                    count += val
                else:
                    temp[r][c + 1] += val
    dp = temp
return count
```

print(find_paths(2, 2, 2, 0, 0))

INPUT:

```
print(find_paths(2, 2, 2, 0, 0))
```

OUTPUT:

```
6
```

12. You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security systems connected, and it will automatically contact the police if two adjacent houses were broken into on the same night.

AIM:

To determine the maximum amount of money that can be robbed from houses arranged in a circular manner without robbing two adjacent houses.

ALGORITHM:

1. Start
2. Read the input array nums
3. If there is only one house, return its value
4. Since houses are in a circle, solve two cases:
    - Case 1: Rob houses from index 0 to n-2
    - Case 2: Rob houses from index 1 to n-1
5. Use a helper function to find the maximum sum for each case using DP
6. Return the maximum of the two cases
7. Stop

CODE:

```
def rob(nums):
```

```
    if len(nums) == 1:
        return nums[0]
    return max(rob_linear(nums[:-1]), rob_linear(nums[1:]))
def rob_linear(arr):
    prev1 = 0
    prev2 = 0
    for num in arr:
        temp = prev1
        prev1 = max(prev2 + num, prev1)
        prev2 = temp
    return prev1
print(rob([2, 3, 2]))
```

INPUT:

```
print(rob([2, 3, 2]))
```

OUTPUT:

```
3
```

13. **You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?**

AIM:

To find the number of distinct ways to climb a staircase of n steps when you can take either 1 step or 2 steps at a time.

ALGORITHM:

1. Start
2. Read the value of n
3. If n is 0 or 1, return 1

4. Create an array dp of size n+1
5. Set dp[0] = 1 and dp[1] = 1
6. For i from 2 to n
7. Set dp[i] = dp[i-1] + dp[i-2]
8. Return dp[n]
9. Stop

CODE:

```
def climb_stairs(n):
    if n == 0 or n == 1:
        return 1
    dp = [0] * (n + 1)
    dp[0] = 1
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
print(climb_stairs(4))
print(climb_stairs(3))
```

INPUT:

```
print(climb_stairs(4))
```

OUTPUT:

```
5
```

14. A robot is located at the top-left corner of a m×n grid .The robot can only move either dowor right at any point in time. The robot is trying to reach the bottom-right corner of the grid. How many possible unique paths are there?

AIM:

To find the number of unique paths a robot can take to move from the top-left corner to the bottom-right corner of an m × n grid, moving only right or down.

ALGORITHM:

1. Start
2. Read the values m and n
3. Create a 2D array dp of size m × n
4. Initialize the first row and first column with 1 (only one way to move)
5. For each cell dp[i][j], calculate dp[i][j] = dp[i-1][j] + dp[i][j-1]
6. Continue until the bottom-right corner is reached
7. Return dp[m-1][n-1]
8. Stop

CODE:

```
def unique_paths(m, n):
    dp = [[0 for _ in range(n)] for _ in range(m)]
    for i in range(m):
        dp[i][0] = 1
    for j in range(n):
        dp[0][j] = 1
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
    return dp[m - 1][n - 1]
print(unique_paths(7, 3))
```

INPUT:

```
print(unique_paths(7, 3))
```

OUTPUT:

```
28
```

15. **In a string S of lowercase letters, these letters form consecutive groups of the same character. For example, a string like s = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z", and "yy". A group is identified by an interval [start, end], where start and end denote the start and end indices (inclusive) of the group. In the above example, "xxxx" has the interval [3,6]. A group is considered large if it has 3 or more characters. Return the intervals of every large group sorted in increasing order by start index.**

AIM:

To identify all large groups (3 or more consecutive identical characters) in a string and return their intervals [start, end] sorted by start index.

ALGORITHM:

1. Start
2. Read the input string s
3. Initialize result as an empty list
4. Initialize start = 0
5. Traverse the string with index i from 1 to len(s)
6. If s[i] != s[i-1] or i == len(s)
    - Check if the group length i - start >= 3
    - If yes, append [start, i-1] to result
    - Update start = i
7. Continue until the end of the string
8. Return result
9. Stop

CODE:

```
def large_group_positions(s):
    result = []
    start = 0
    for i in range(1, len(s) + 1):
        if i == len(s) or s[i] != s[i - 1]:
            if i - start >= 3:
```

```
            result.append([start, i - 1])
        start = i
    return result
print(large_group_positions("abbxxxxzzy"))
```

INPUT:

```
print(large_group_positions("abbxxxxzzy"))
```

OUTPUT:

```
[[3, 6]]
```

16. **"The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970." The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules**
    Any live cell with fewer than two live neighbors dies as if caused by under-population.
    Any live cell with two or three live neighbors lives on to the next generation.
    Any live cell with more than three live neighbors dies, as if by over-population.
    Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.
    The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return *the next state*.

AIM:

To compute the next state of a given m × n grid for Conway's Game of Life by applying the rules of under-population, survival, over-population, and reproduction simultaneously to all cells.

ALGORITHM:

1. Start
2. Read the input grid board of size m × n
3. Create a copy of the board or mark intermediate states to avoid overwriting

4. For each cell at position (i, j)
    - Count the number of live neighbors (8 directions: horizontal, vertical, diagonal)
    - Apply the rules:
        - If live and neighbors < 2 → dies
        - If live and neighbors 2 or 3 → lives
        - If live and neighbors > 3 → dies
        - If dead and neighbors == 3 → becomes live
5. Update the board simultaneously based on the computed next state
6. Return the updated board
7. Stop

CODE:

```
def game_of_life(board):
    m, n = len(board), len(board[0])
    directions = [(-1, -1), (-1, 0), (-1, 1),
                  (0, -1),          (0, 1),
                  (1, -1), (1, 0), (1, 1)]
    copy_board = [[board[i][j] for j in range(n)] for i in range(m)]
    for i in range(m):
        for j in range(n):
            live_neighbors = 0
            for dx, dy in directions:
                ni, nj = i + dx, j + dy
                if 0 <= ni < m and 0 <= nj < n and copy_board[ni][nj] == 1:
                    live_neighbors += 1
```

```
            if copy_board[i][j] == 1:
                if live_neighbors < 2 or live_neighbors > 3:
                    board[i][j] = 0
            else:
                if live_neighbors == 3:
                    board[i][j] = 1
    return board
board = [[0,1,0],
         [0,0,1],
         [1,1,1],
         [0,0,0]]
next_state = game_of_life(board)
for row in next_state:
    print(row)
```

INPUT:

```
board = [[0,1,0],
         [0,0,1],
         [1,1,1],
         [0,0,0]]
```

OUTPUT:

```
[0, 0, 0]
[1, 0, 1]
[0, 1, 1]
[0, 1, 0]
```

17. **We stack glasses in a pyramid, where the first row has 1 glass, the second row has 2 glasses, and so on until the 100th row. Each glass holds one cup of champagne. Then, some**

champagne is poured into the first glass at the top. When the topmost glass is full, any excess liquid poured will fall equally to the glass immediately to the left and right of it. When those glasses become full, any excess champagne will fall equally to the left and right of those glasses, and so on. (A glass at the bottom row has its excess champagne fall on the floor.) For example, after one cup of champagne is poured, the top most glass is full. After two cups of champagne are poured, the two glasses on the second row are half full. After three cups of champagne are poured, those two cups become full - there are 3 full glasses total now. After four cups of champagne are poured, the third row has the middle glass half full, and the two outside glasses are a quarter full, as pictured below.

Now after pouring some non-negative integer cups of champagne, return how full the jth glass in the ith row is (both i and j are 0-indexed.)

AIM:

To determine how full a specific glass is in a champagne tower after pouring a given number of cups, following the rule that overflow from a glass is equally split to the glasses immediately below it.

ALGORITHM:

1. Start
2. Read input values: poured, query_row, and query_glass
3. Create a 2D array dp with dimensions (query_row+2) × (query_row+2) initialized to 0
4. Pour the champagne into the top glass: dp[0][0] = poured
5. For each row i from 0 to query_row
6. For each glass j in row i
    - If dp[i][j] > 1 (overflow occurs)
    - Calculate excess = dp[i][j] - 1
    - Add half of excess to the glass below-left: dp[i+1][j] += excess / 2
    - Add half of excess to the glass below-right: dp[i+1][j+1] += excess / 2
    - Set dp[i][j] = 1 (since a glass can hold at most 1 cup)

7. Return min(1, dp[query_row][query_glass]) as the fullness of the requested glass
8. Stop

CODE:

```
def champagneTower(poured, query_row, query_glass):
    dp = [[0] * (query_row + 2) for _ in range(query_row + 2)]
    dp[0][0] = poured
    for i in range(query_row + 1):
        for j in range(i + 1):
            if dp[i][j] > 1:
                excess = dp[i][j] - 1
                dp[i+1][j] += excess / 2
                dp[i+1][j+1] += excess / 2
                dp[i][j] = 1
    return dp[query_row][query_glass]
print(champagneTower(2, 1, 1))
```

INPUT:

```
print(champagneTower(1, 1, 1))
```

OUTPUT:

```
0.5
```