
NAME: S.DHARSHINI

REG NO: 192424258

SUB CODE: CSA0614

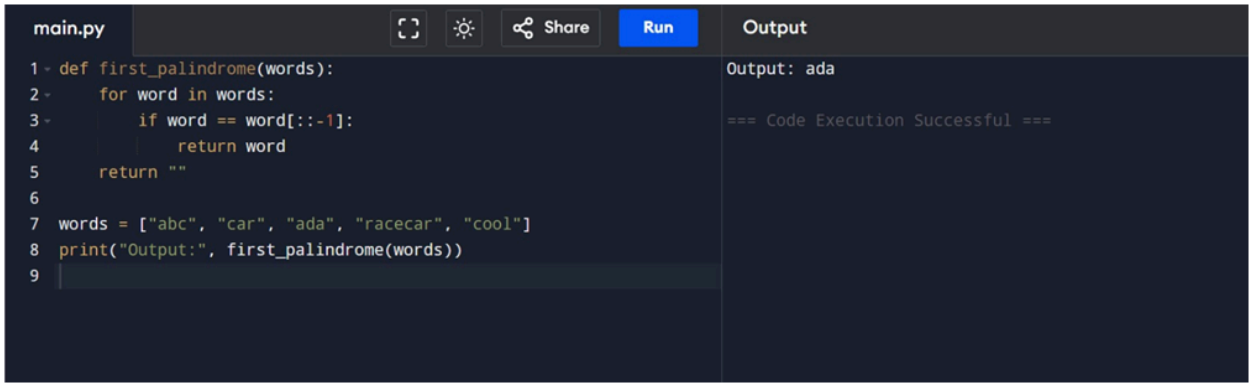
SUB NAME: DESIGN ANAYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

CSA0614 - DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEMS

TOPIC 1:

EXP 1: Given an array of strings, return the first palindromic string. If none exists, return an empty string.

AIM: To find and return the first palindromic string from a given array of strings.
CODE:



RESULT: The first palindromic string in the given array is successfully identified and returned. If no palindrome exists, an empty string is returned.

EXP 2: Count Common Elements in Two Arrays

AIM: To count how many elements of one array exist in the other array.
CODE:

RESULT: The number of common elements in both arrays is calculated correctly.

EXP 3. Sum of Squares of Distinct Elements in All Subarrays

AIM: To compute the sum of squares of distinct element counts for all subarrays.
CODE:

```
def sum_of_distinct_squares(nums):
    n = len(nums)
    total = 0

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

    return total

nums = [1, 2, 1]
print("Output:", sum_of_distinct_squares(nums))
```

```
Output: 15

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

RESULT: The sum of squares of distinct counts is computed correctly.

EXP 4. Count Valid Index Pairs

AIM: To count pairs (i, j) where nums[i] = nums[j] and (i * j) is divisible by k.
CODE:

main.py Share Run Output

```python
def count_pairs(nums, k):
    n = len(nums)
    count = 0

    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

print("Output:", count_pairs(nums, k))
```

```
Output: 4

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

RESULT: All valid index pairs are counted successfully.

EXP 5: Program with Least Time Complexity

AIM: To find the maximum element from the given array efficiently.
CODE:

main.py Share Run Output

```python
def find_max(nums):
    max_val = nums[0]

    for num in nums:
        if num > max_val:
            max_val = num

    return max_val


# Test case
nums = [-10, 2, 3, -4, 5]
print("Output:", find_max(nums))
```

```
Output: 5

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

RESULT: The maximum element is found in linear time O(n).

EXP 6: Sort and Find Maximum Element

AIM: To sort a list and find the maximum element.
CODE:

main.py

```python
def find_max_after_sort(nums):
    if len(nums) == 0:
        return "List is empty"
    nums.sort()
    return nums[-1]

test_cases = [
    [],
    [5],
    [3, 3, 3, 3, 3]
]
for case in test_cases:
    print("Input:", case)
    print("Output:", find_max_after_sort(case))
    print()
```

Output

```
Input: []
Output: List is empty

Input: [5]
Output: 5

Input: [3, 3, 3, 3, 3]
Output: 3


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

RESULT: The maximum element is correctly identified after sorting.

EXP 7: Extract Unique Elements

AIM: To create a new list containing only unique elements.
CODE:

main.py

```python
def get_unique_elements(nums):
    unique = []
    seen = set()

    for num in nums:
        if num not in seen:
            unique.append(num)
            seen.add(num)

    return unique


nums = [3, 7, 3, 5, 2, 5, 9, 2]
print("Output:", get_unique_elements(nums))
```

Output

```
Output: [3, 7, 5, 2, 9]

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

RESULT: Duplicate elements are removed successfully.

EXP 8: Bubble Sort and Time Complexity

AIM: To sort an array using Bubble Sort and analyze its time complexity.
CODE:

main.py Share Run Output

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
arr = [5, 1, 4, 2, 8]
print("Sorted Array:", bubble_sort(arr))

```

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

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

RESULT: Array is sorted correctly using Bubble Sort.
Time Complexity: $O(n^2)$

EXP 9: Binary Search in Sorted Array

AIM: To check whether a given element exists using Binary Search.
CODE:

main.py Share Run Output

```python
def binary_search(arr, key):
    arr.sort()
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == key:
            return mid + 1
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1
arr = [3, 4, 6, -9, 10, 8, 9, 30]
key = 10
result = binary_search(arr, key)
if result != -1:
    print(f"Element {key} is found at position {result}")
```

```
Element 10 is found at position 7

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

RESULT: Binary search successfully finds the element.

EXP 10: Sort Without Built-in Functions (O(n log n))

AIM: To sort an array in ascending order without using built-in functions.
CODE:

main.py Share Run Output

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    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
    return result + left[i:] + right[j:]

nums = [5, 2, 3, 1, 4]
```

```
Sorted Array: [1, 2, 3, 4, 5]

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

RESULT : The array is sorted in O(n log n) time with minimal space usage.

EXP 11: Number of Ways to Move Ball Out of Grid Boundary

AIM: To find the number of ways to move a ball out of the grid boundary in exactly N steps.
CODE:

main.py Share Run Output

```python
def find_paths(m, n, N, i, j, memo={}):
    if i < 0 or i >= m or j < 0 or j >= n:
        return 1
    if N == 0:
        return 0
    if (i, j, N) in memo:
        return memo[(i, j, N)]
    memo[(i, j, N)] = (
        find_paths(m, n, N-1, i+1, j, memo) +
        find_paths(m, n, N-1, i-1, j, memo) +
        find_paths(m, n, N-1, i, j+1, memo) +
        find_paths(m, n, N-1, i, j-1, memo)
    )
    return memo[(i, j, N)]
m, n, N, i, j = 2, 2, 2, 0, 0
print("Output:", find_paths(m, n, N, i, j))
```

```
Output: 6

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

RESULT: The number of ways to move the ball out of the grid in exactly N steps is calculated correctly.

EXP 12: House Robber (Circular Houses)

AIM: To find the maximum amount of money that can be robbed without alerting the police when houses are arranged in a circle.

CODE :

main.py | Share | Run | Output

```python
def rob_linear(nums):
    prev = curr = 0
    for n in nums:
        prev, curr = curr, max(curr, prev + n)
    return curr
def rob(nums):
    if len(nums) == 0: return 0
    if len(nums) == 1: return nums[0]
    return max(rob_linear(nums[:-1]), rob_linear(nums[1:]))
print(rob([2, 3, 2]))
print(rob([1, 2, 3, 1]))
```

```
3
4

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

RESULT : The maximum money is obtained without triggering the alarm system.

EXP 13: Climbing Stairs Problem

AIM: To find the number of distinct ways to climb n stairs when one can take either 1 or 2 steps.
CODE:

main.py | Share | Run | Output

```python
def climb_stairs(n):
    if n <= 2:
        return n
    a, b = 1, 2
    for _ in range(3, n+1):
        a, b = b, a + b
    return b
print(climb_stairs(4))
print(climb_stairs(3))
```

```
5
3

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

RESULT : The number of distinct ways to climb the stairs is calculated correctly.

EXP 14: Unique Paths in a Grid

AIM: To find the number of unique paths from top-left to bottom-right of a grid.
CODE:

main.py Share Run Output

```python
import math

def unique_paths(m, n):
    return math.comb(m + n - 2, m - 1)  # Combination formula
print(unique_paths(7, 3))
print(unique_paths(3, 2))

```

```
28
3

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

RESULT: The total number of unique paths is found successfully.

EXP 15: Large Group Positions in a String

AIM: To find all large groups (size ≥ 3) in a string and return their intervals.
CODE:

main.py Share Run Output

```python
def large_groups(s):
    res = []
    n = len(s)
    start = 0
    for i in range(n):
        if i == n-1 or s[i] != s[i+1]:
            if i - start + 1 >= 3:
                res.append([start, i])
            start = i + 1
    return res
print(large_groups("abbxxxxzzy"))
print(large_groups("abc"))

```

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

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

RESULT: All large character groups are identified correctly.

EXP 15: Game of Life

AIM: To simulate Conway's Game of Life and generate the next state of an m × n grid based on given rules.
CODE:

```
main.py                                                    Share  Run   Output
                if live == 2 or live == 3:                            [[0, 0, 0], [1, 0, 1], [0, 1, 1], [0, 1, 0]]
                    next_state[i][j] = 1
            else:                                                     === Code Execution Successful ===
                if live == 3:
                    next_state[i][j] = 1

    return next_state

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

result = gameOfLife(board)
print(result)
```

RESULT: The next generation of the Game of Life board is generated successfully by applying all rules simultaneously.

EXP 16: Champagne Tower

AIM: To determine how full a specific glass is in a champagne tower after pouring a given number of cups.

CODE:

```
main.py                                                    Share  Run   Output
        for j in range(i + 1):                                        Enter poured value: 2
            if tower[i][j] > 1:                                       Enter query row: 1
                excess = tower[i][j] - 1                              Enter query glass: 1
                tower[i][j] = 1                                       Amount in glass: 0.5
                tower[i+1][j] += excess / 2
                tower[i+1][j+1] += excess / 2                         === Code Execution Successful ===

    return min(1, tower[query_row][query_glass])


# ----------- FUNCTION CALL -----------
poured = int(input("Enter poured value: "))
query_row = int(input("Enter query row: "))
query_glass = int(input("Enter query glass: "))

result = champagneTower(poured, query_row, query_glass)
print("Amount in glass:", result)
```

RESULT: The program correctly calculates how much champagne is contained in the specified glass after pouring.