Question 1: First Palindromic String

Aim

To find the first palindromic string in a given array of strings.

Procedure

1. Read the list of words.
2. For each word, check if it is equal to its reverse.
3. Return the first word that satisfies this condition.
4. If no palindrome is found, return an empty string.

Python Code

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

words = ["abc", "car", "ada", "racecar"]
print(first_palindrome(words))
```

Output

```
ada
```

Result

The program successfully finds the first palindromic string.

Question 2: Intersection Values

Aim

To find how many elements of one array exist in the other array.

Procedure

1. Read two arrays.
2. Count how many elements of the first array exist in the second.
3. Count how many elements of the second array exist in the first.

4. Return both counts.

Python Code

```
def findIntersectionValues(nums1, nums2):
    answer1 = 0
    answer2 = 0

    for num in nums1:
        if num in nums2:
            answer1 += 1

    for num in nums2:
        if num in nums1:
            answer2 += 1

    return [answer1, answer2]

nums1 = [4,3,2,3,1]
nums2 = [2,2,5,2,3,6]
print(findIntersectionValues(nums1, nums2))
```

Output

```
[3, 1]
```

Result

The program correctly calculates the intersection values.

Question 3: Sum of Squares of Distinct Counts

Aim

To calculate the sum of squares of distinct elements in all subarrays.

Procedure

1. Generate all subarrays.
2. Count distinct elements using a set.
3. Square the count and add it to the sum.
4. Return the total sum.

Python Code

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

    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]
print(sumOfSquaresOfDistinctCounts(nums))
```

Output

```
15
```

Result

The program correctly computes the sum of squares of distinct elements.

Question 4: Count Valid Pairs

Aim

To count pairs (i, j) such that nums[i] == nums[j] and (i × j) is divisible by k.

Procedure

1. Use two nested loops to check all pairs.
2. Check if values are equal.
3. Check if (i × j) % k == 0.
4. Increment count if both conditions are true.

Python Code

```
def countPairs(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(countPairs(nums, k))
```

Output

```
4
```

Result

The program correctly counts all valid pairs.

Question 5: Find Maximum Element

Aim

To find the maximum element in an array with least time complexity.

Procedure

1. Initialize max with the first element.
2. Traverse the array.
3. Update max if a larger value is found.
4. Print the max value.

Python Code

```
def find_max(nums):
    max_val = nums[0]
    for num in nums:
        if num > max_val:
            max_val = num
    return max_val

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
```

Result

The program efficiently finds the maximum value.

Question 6: Sort and Find Maximum

Aim

To sort the list using an efficient algorithm and find the maximum element.

Procedure

1. Sort the list using built-in sort.
2. Select the last element as the maximum.
3. Display sorted list and maximum.

Python Code

```python
def process_numbers(nums):
    sorted_nums = sorted(nums)
    max_element = sorted_nums[-1]
    return sorted_nums, max_element

nums = [5, 1, 9, 3, 7]
sorted_list, maximum = process_numbers(nums)
print(sorted_list)
print(maximum)
```

Output

```
[1, 3, 5, 7, 9]
9
```

Result

The program successfully sorts the list and finds the maximum element.

7. Unique Elements from List

Aim

To write a program that extracts only unique elements from a given list.

Procedure

1. Create an empty set.
2. Traverse the list.
3. If element not in set, add it to set and result list.
4. Print result.

Program

```
def unique_elements(arr): seen = set() res = [] for x in arr: if x not in
seen: seen.add(x) res.append(x) return res
print(unique_elements([3,7,3,5,2,5,9,2]))
print(unique_elements([-1,2,-1,3,2,-2]))
print(unique_elements([1000000,999999,1000000]))
```

Output

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

Result

The program successfully extracts unique elements.

8. Bubble Sort

Aim

To sort an array using Bubble Sort and analyze time complexity.

Procedure

1. Compare adjacent elements.
2. Swap if out of order.
3. Repeat for all passes.

Program

```
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



print(bubble_sort([5,2,9,1,5,6]))
```

Output

```
[1, 2, 5, 5, 6, 9]
```

Result

The array is sorted using Bubble Sort.

9. Binary Search

Aim

To check if a number exists using Binary Search.

Procedure

1. Sort the array.
2. Find middle.
3. Compare key.
4. Repeat.

Program

```
def binary_search(arr, key):

    arr.sort()

    low, high = 0, len(arr)-1

    while low <= high:

        mid = (low + high)//2

        if arr[mid] == key:

            return mid

        elif arr[mid] < key:

            low = mid + 1

        else:

            high = mid - 1

    return -1


arr = [3,4,6,-9,10,8,9,30]

key = 10

pos = binary_search(arr, key)

if pos != -1:

    print("Found at position", pos)

else:
    print("Not found")
```

Output

```
Found at position 5
```

Result

Binary search successfully finds the element.

10. Sort Array in O(n log n)

Aim

To sort an array using Merge Sort.

Procedure:

1. Divide the given array into two halves.
2. Recursively sort the left half.
3. Recursively sort the right half.
4. Merge both sorted halves into a single sorted array.
5. Repeat until the entire array is sorted.

Program

```
def merge_sort(arr):

    if len(arr) > 1:

        mid = len(arr)//2

        L = arr[:mid]

        R = arr[mid:]

        merge_sort(L)

        merge_sort(R)

        i=j=k=0

        while i < len(L) and j < len(R):

            if L[i] < R[j]:
                arr[k]=L[i]
                i+=1
            else:
                arr[k]=R[j]
                j+=1
            k+=1
        while i < len(L):
            arr[k]=L[i]
            i+=1
            k+=1
        while j < len(R):
            arr[k]=R[j]
            j+=1
            k+=1


arr = [5,3,8,4,2]
merge_sort(arr)
print(arr)
```

Output

```
[2, 3, 4, 5, 8]
```

Result

The array is sorted in O(n log n).

11. Ball Out of Boundary Paths

Aim

To find number of ways a ball moves out of grid in N steps.

Procedure:

1. Initialize a grid to store the number of ways to reach each cell.
2. Place the ball at the starting position.
3. For each step from 1 to N:
    - Move the ball in all four directions (up, down, left, right).
    - If the ball goes outside the grid, increase the count.
    - Otherwise, update the new position.
4. After N steps, return the total count.

Program

```
def findPaths(m, n, N, i, j):
    MOD = 10**9 + 7
    dp = [[0]*n for _ in range(m)]
    dp[i][j] = 1
    count = 0


    for _ in range(N):
        temp = [[0]*n for _ in range(m)]
        for r in range(m):
            for c in range(n):
                if dp[r][c] > 0:
                    for dr,dc in [(1,0),(-1,0),(0,1),(0,-1)]:
                        nr, nc = r+dr, c+dc
                        if nr<0 or nr>=m or nc<0 or nc>=n:
                            count += dp[r][c]
                        else:
                            temp[nr][nc] += dp[r][c]
        dp = temp

    return count



print(findPaths(2,2,2,0,0))
```

Output

```
6
```

Result

Correct number of paths calculated.

12. House Robber (Circular)

Aim

To find maximum money robbed without alert.

Procedure:

1. If there is only one house, rob it.
2. Since houses are in a circle, consider two cases:
    - Rob from first house to second-last house.
    - Rob from second house to last house.
3. For each case, use dynamic programming:
    - At each house, choose maximum of robbing or skipping.
4. Return the maximum of both cases.

Program

```
def rob(nums):

    if len(nums) == 1:

        return nums[0]



    def helper(arr):

        prev = curr = 0

        for x in arr:

            prev, curr = curr, max(curr, prev + x)

        return curr



    return max(helper(nums[:-1]), helper(nums[1:]))



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

print(rob([1,2,3,1]))
```

Output

```
3
4
```

Result

Maximum money calculated.

13. Climbing Stairs

Aim

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

Procedure

1. If `n` is 1 or 2, return `n`.
2. Use two variables to store previous step counts.
3. Add the last two values to get the next.
4. Repeat until `n` is reached.
5. Print the result.

Program

```
def climbStairs(n):

    if n <= 2:

        return n

    a, b = 1, 2

    for i in range(3, n+1):

        a, b = b, a + b

    return b

print(climbStairs(4))

print(climbStairs(3))
```

Output

```
5
3
```

Result

The program correctly calculates the number of ways to climb the stairs.

14. Unique Paths in Grid

Aim

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

Procedure

1. Create a 2D array.
2. Fill first row and first column with 1.
3. Add values from top and left cells.
4. Store the final answer in the last cell.
5. Print the result.

Program

```
def uniquePaths(m, n):

    dp = [[1]*n for _ in range(m)]

    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(uniquePaths(7, 3))

print(uniquePaths(3, 2))
```

Output

```
28
3
```

Result

The program successfully finds the number of unique paths in the grid.

15. Large Groups in a String

Aim

To find all large groups (3 or more consecutive characters) in a string.

Procedure

1. Traverse the string.
2. Count consecutive characters.
3. If count ≥ 3, store start and end index.
4. Repeat for all characters.
5. Print the result.

Program

```
def largeGroupPositions(s):

    result = []

    i = 0

    while i < len(s):

        j = i

        while j < len(s) and s[j] == s[i]:

            j += 1

        if j - i >= 3:

            result.append([i, j-1])

        i = j

    return result

print(largeGroupPositions("abbxxxxzzy"))

print(largeGroupPositions("abc"))
```

Output

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

Result

The program correctly identifies all large groups in the string.

16. Champagne Tower

Aim

To find how full a specific glass is after pouring champagne into the tower.

Procedure

1. Create a 2D array.
2. Pour champagne into the top glass.
3. If a glass overflows, divide excess into two glasses below.
4. Continue until required row.
5. Print the required glass value.

Program

```
def champagneTower(poured, query_row, query_glass):

    dp = [[0]*101 for _ in range(101)]

    dp[0][0] = poured



    for i in range(100):

        for j in range(i+1):

            excess = max(0, dp[i][j] - 1)

            dp[i][j] = min(1, dp[i][j])

            dp[i+1][j] += excess / 2

            dp[i+1][j+1] += excess / 2

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

print(champagneTower(1, 1, 1))

print(champagneTower(2, 1, 1))
```

Output

```
0.0
0.5
```

Result

The program accurately calculates how full the specified glass is.