In [2]:

```
#1. Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an
def firstPalindrome(words):
    # Helper function to check if a string is a palindrome
    def isPalindrome(word):
        return word == word[::-1]

    for word in words:
        if isPalindrome(word):
            return word

    return ""

words = ["abc", "car", "ada", "racecar", "cool"]
print(firstPalindrome(words))  # Output: "ada"
```

```
ada
```

In [3]:

```
#2. You are given two integer arrays nums1 and nums2 of sizes n and m, respectively. Calculate the following values: answer1
def calculateIndices(nums1, nums2):
    set_nums2 = set(nums2)

    answer1 = sum(1 for num in nums1 if num in set_nums2)

    set_nums1 = set(nums1)

    answer2 = sum(1 for num in nums2 if num in set_nums1)

    return [answer1, answer2]

nums1 = [1, 2, 3, 4]
nums2 = [3, 4, 5, 6]
print(calculateIndices(nums1, nums2))  # Output: [2, 2]
```

```
[2, 2]
```

In [23]:

```
#3
def sum_of_squares_of_distinct_counts(nums):
    from collections import defaultdict

    n = len(nums)
    result = 0

    for start in range(n):
        count_map = defaultdict(int)
        for end in range(start, n):
            count_map[nums[end]] += 1
            distinct_count = len(count_map)
            result += distinct_count ** 2

    return result

# Example usage:
nums = [1, 2, 1]
print(sum_of_squares_of_distinct_counts(nums))  # Output: 15
```

```
15
```

In [6]:

```
#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 <
def countPairs(nums, k):
    n = len(nums)
    count = 0
    value_indices = {}

    for i in range(n):
        if nums[i] in value_indices:
            value_indices[nums[i]].append(i)
        else:
            value_indices[nums[i]] = [i]

    for indices in value_indices.values():
        size = len(indices)
        for i in range(size):
            for j in range(i + 1, size):
                if (indices[i] * indices[j]) % k == 0:
                    count += 1

    return count

nums = [3, 1, 2, 2, 3, 3]
k = 2
print(countPairs(nums, k))  # Output: number of valid pairs
```

```
4
```

In [8]:

```
#5. Write a  program FOR THE BELOW TEST CASES with least time complexity         Test Cases: -
#1) Input: {1, 2, 3, 4, 5} Expected Output: 5
#2) Input: {7, 7, 7, 7, 7} Expected Output: 7
#3) Input: {-10, 2, 3, -4, 5} Expected Output: 5


def findMax(nums):
    max_value = nums[0]

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

    return max_value

print(findMax([1, 2, 3, 4, 5]))  # Expected Output: 5
print(findMax([7, 7, 7, 7, 7]))  # Expected Output: 7
print(findMax([-10, 2, 3, -4, 5]))  # Expected Output: 5
```

```
5
7
5
```

In [10]:

```
#6. You have an algorithm that process a list of numbers. It firsts sorts the list using an efficient sorting algorithm and
#Test Cases
#1. Empty List
#1. Input: []
#2. Expected Output: None or an appropriate message indicating that the list is empty.
#2. Single Element List
#1. Input: [5]
#2. Expected Output: 5
#3. All Elements are the Same
#1. Input: [3, 3, 3, 3, 3]
#2. Expected Output: 3

def processList(nums):
    # Check for the empty list case
    if not nums:
        return "List is empty"

    nums.sort()

    max_value = nums[-1]

    return max_value

print(processList([]))  # Expected Output: "List is empty"
print(processList([5]))  # Expected Output: 5
print(processList([3, 3, 3, 3, 3]))  # Expected Output: 3
print(processList([1, 2, 3, 4, 5]))  # Expected Output: 5
print(processList([-10, 2, 3, -4, 5]))  # Expected Output: 5
```

```
List is empty
5
3
5
5
```

In [11]:

```
#7. Write a program that takes an input list of n numbers and  creates a new list containing only the unique elements from th
#Test Cases
#Some Duplicate Elements
#•  Input: [3, 7, 3, 5, 2, 5, 9, 2]
#•  Expected Output: [3, 7, 5, 2, 9] (Order may vary based on the algorithm used)
#Negative and Positive Numbers
#•  Input: [-1, 2, -1, 3, 2, -2]
#•  Expected Output: [-1, 2, 3, -2] (Order may vary)
#List with Large Numbers
#•  Input: [1000000, 999999, 1000000]
#•  Expected Output: [1000000, 999999]


def uniqueElements(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(uniqueElements([3, 7, 3, 5, 2, 5, 9, 2]))  # Expected Output: [3, 7, 5, 2, 9]
print(uniqueElements([-1, 2, -1, 3, 2, -2]))  # Expected Output: [-1, 2, 3, -2]
print(uniqueElements([1000000, 999999, 1000000]))  # Expected Output: [1000000, 999999]
```

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

In [12]:

```
#8.Sort an array of integers using the bubble sort technique. Analyze its time complexity using Big-O notation. Write the cod
def bubbleSort(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]

arr = [64, 34, 25, 12, 22, 11, 90]
bubbleSort(arr)
print("Sorted array is:", arr)
```

```
Sorted array is: [11, 12, 22, 25, 34, 64, 90]
```

In [13]:

```
#9.Checks if a given number x exists in a sorted array arr using binary search. Analyze its time complexity using Big-O notat
def binarySearch(arr, key):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == key:
            return mid

        elif arr[mid] < key:
            left = mid + 1

        else:
            right = mid - 1

    return -1

arr = sorted([3, 4, 6, -9, 10, 8, 9, 30])  # The array should be sorted for binary search
key = 10
index = binarySearch(arr, key)

if index != -1:
    print(f"Element {key} is found at position {index}")
else:
    print(f"Element {key} is not found in the array")

print("Sorted array is:", arr)
print(f"Index of element {key}:", index)
```

```
Element 10 is found at position 6
Sorted array is: [-9, 3, 4, 6, 8, 9, 10, 30]
Index of element 10: 6
```

In [15]:

```
#10.Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without usi
def mergeSort(nums):
    if len(nums) <= 1:
        return nums

    mid = len(nums) // 2
    left_half = nums[:mid]
    right_half = nums[mid:]

    left_half = mergeSort(left_half)
    right_half = mergeSort(right_half)

    return merge(left_half, right_half)

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

    result.extend(left[i:])
    result.extend(right[j:])

    return result

nums = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
sorted_nums = mergeSort(nums)
print("Sorted array:", sorted_nums)
```

```
Sorted array: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

In [16]:

```
#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
def findPaths(m, n, N, startRow, startCol):
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    dp = [[[0] * (N + 1) for _ in range(n)] for _ in range(m)]

    dp[startRow][startCol][0] = 1

    for step in range(1, N + 1):
        for r in range(m):
            for c in range(n):
                for dr, dc in directions:
                    nr, nc = r + dr, c + dc
                    if 0 <= nr < m and 0 <= nc < n:
                        dp[r][c][step] += dp[nr][nc][step - 1]

    total_ways = 0
    for r in range(m):
        for c in range(n):
            if r == 0 or r == m - 1 or c == 0 or c == n - 1:
                total_ways += dp[r][c][N]

    return total_ways

m = 3
n = 3
N = 4
startRow = 0
startCol = 0
print(findPaths(m, n, N, startRow, startCol))  # Output: Number of ways to move out of grid boundary in exactly N steps
```

```
32
```

In [17]:

```
#12.    You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed
def rob(nums):
    def rob_linear(nums):
        n = len(nums)
        if n == 0:
            return 0
        if n == 1:
            return nums[0]

        dp = [0] * n
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        for i in range(2, n):
            dp[i] = max(dp[i-1], nums[i] + dp[i-2])

        return dp[-1]

    n = len(nums)
    if n == 0:
        return 0
    if n == 1:
        return nums[0]
    if n == 2:
        return max(nums[0], nums[1])

    max1 = rob_linear(nums[:-1])

    max2 = rob_linear(nums[1:])

    return max(max1, max2)

nums = [2, 3, 2]
print(rob(nums))  # Output: 3

nums = [1, 2, 3, 1]
print(rob(nums))  # Output: 4
```

```
3
4
```

In [18]:

```
#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
def climbStairs(n):
    if n == 0:
        return 1
    if 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]

# Example usage:
n = 2
print(climbStairs(n))
n = 3
print(climbStairs(n))

n = 4
print(climbStairs(n))
```

```
2
3
5
```

In [19]:

```
#14.    A robot is located at the top-left corner of a m×n grid .The robot can only move either down or right at any point i
def uniquePaths(m, n):
    dp = [[0] * n for _ in range(m)]

    dp[0][0] = 1

    for i in range(m):
        for j in range(n):
            if i > 0:
                dp[i][j] += dp[i-1][j]
            if j > 0:
                dp[i][j] += dp[i][j-1]

    return dp[m-1][n-1]


m = 3
n = 7
print(uniquePaths(m, n))  # Output: 28

m = 3
n = 2
print(uniquePaths(m, n))  # Output: 3
```

```
28
3
```

In [20]:

```
#15.    In a string S of lowercase letters, these letters form consecutive groups of the same character. For example, a stri
def largeGroupPositions(S):
    n = len(S)
    if n == 0:
        return []

    result = []
    start = 0

    while start < n:
        end = start

        while end < n and S[end] == S[start]:
            end += 1

        if end - start >= 3:
            result.append([start, end - 1])

        start = end

    return result

S = "abbxxxxzyy"
print(largeGroupPositions(S))  # Output: [[3, 6]]

S = "abc"
print(largeGroupPositions(S))  # Output: []

S = "aaa"
print(largeGroupPositions(S))  # Output: [[0, 2]]
```

```
[[3, 6]]
[]
[[0, 2]]
```

In [24]:

```
#17.
def champagneTower(poured, query_row, query_glass):
    dp = [[0.0] * (r + 1) for r in range(query_row + 1)]
    dp[0][0] = poured  # Pour the initial amount into the top glass

    for i in range(query_row):
        for j in range(i + 1):
            # Calculate overflow amount
            overflow = (dp[i][j] - 1.0) / 2.0
            if overflow > 0:
                dp[i + 1][j] += overflow
                dp[i + 1][j + 1] += overflow

    return min(dp[query_row][query_glass], 1.0)

poured = 4
query_row = 2
query_glass = 1
print(champagneTower(poured, query_row, query_glass))  # Output: 0.5
```

```
0.5
```

In [22]:

```
#16.
def gameOfLife(board):
    if not board:
        return

    m, n = len(board), len(board[0])
    next_board = [[0] * n for _ in range(m)]

    directions = [(-1, -1), (-1, 0), (-1, 1),
                  (0, -1),          (0, 1),
                  (1, -1), (1, 0), (1, 1)]

    def count_live_neighbors(x, y):
        count = 0
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < m and 0 <= ny < n and board[nx][ny] == 1:
                count += 1
        return count

    for i in range(m):
        for j in range(n):
            live_neighbors = count_live_neighbors(i, j)

            if board[i][j] == 1:
                if live_neighbors < 2 or live_neighbors > 3:
                    next_board[i][j] = 0
                else:
                    next_board[i][j] = 1
            else:
                if live_neighbors == 3:
                    next_board[i][j] = 1
                else:
                    next_board[i][j] = 0

    for i in range(m):
        for j in range(n):
            board[i][j] = next_board[i][j]

board = [
    [0, 1, 0],
    [0, 0, 1],
    [1, 1, 1],
    [0, 0, 0]
]
gameOfLife(board)
print(board)
```

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

In [ ]: