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.**

Example 1:

Input: words = ["abc","car","ada","racecar","cool"]

Output: "ada"

Explanation: The first string that is palindromic is "ada".

Note that "racecar" is also palindromic, but it is not the first.

ALGORITHM:

1. Start with two pointers, one at the beginning of the string and one at the end.
2. Compare the characters at these pointers.
3. If the characters are equal, move the pointers towards the center.
4. If the characters ever differ, the string is not a palindrome.
5. Repeat this process until the pointers meet or cross each other.
6. If the pointers meet or cross without finding unequal characters, the string is a palindrome

CODE:

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


# Example
words = ["abc", "car", "ada", "racecar", "cool"]
print(first_palindromic_string(words))
```

OUTPUT: “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].

Example 1:
Input: nums1 = [2,3,2], nums2 = [1,2]
Output: [2,1]
Explanation:

Example 2:

Input: nums1 = [4,3,2,3,1], nums2 = [2,2,5,2,3,6]
Output: [3,4]
Explanation:
The elements at indices 1, 2, and 3 in nums1 exist in nums2 as well. So answer1 is 3.
The elements at indices 0, 1, 3, and 4 in nums2 exist in nums1. So answer2 is 4.

ALGORITHM:

1. Define a function calculate_answers that takes two lists of numbers, nums1 and nums2, as input.
2. Convert nums2 into a set set_nums2 for efficient lookups.
3. Count common elements in nums1 and nums2 by iterating through each list and checking for common elements using set intersection.
4. Return a list containing the counts of common elements in both lists.

CODE:

```
def count_indices(nums1, nums2):
    set_nums1 = set(nums1)
    set_nums2 = set(nums2)

    answer1 = sum(1 for num in nums1 if num in set_nums2)
    answer2 = sum(1 for num in nums2 if num in set_nums1)

    return [answer1, answer2]

# Example usage
nums1 = [1, 2, 2, 3]
nums2 = [2, 3, 4, 5]
result = count_indices(nums1, nums2)
print(result)  # Output: [3, 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.

Example 1:
Input: nums = [1,2,1]
Output: 15
Explanation: Six possible subarrays are:
[1]: 1 distinct value
[2]: 1 distinct value
[1]: 1 distinct value
[1,2]: 2 distinct values
[2,1]: 2 distinct values
[1,2,1]: 2 distinct values
The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 + 22 + 22 + 22 = 15.

Example 2:
Input: nums = [1,1]
Output: 3
Explanation: Three possible subarrays are:
[1]: 1 distinct value
[1]: 1 distinct value
[1,1]: 1 distinct value
The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 = 3.

ALGORITHM:

1. Define a function sum_of_squares_distinct_counts that takes a list of numbers, nums, as input.
2. Initialize a variable result to 0.
3. Iterate through the indices of the input list using a loop with index i.
4. Create an empty dictionary count to store the count of distinct elements.
5. Iterate through the elements of the input list starting from index i.
6. Update the count of each element in the dictionary count.
7. Calculate the sum of squares of the values in the count dictionary and add it to the result.
8. Return the final result.

CODE:

```
def sum_of_distinct_counts(nums):
    result = 0
    for i in range(len(nums)):
        for j in range(i, len(nums)):
            distinct_count = len(set(nums[i:j+1]))
            result += distinct_count ** 2
    return result

# Example 1
nums1 = [1, 2, 1]
print(sum_of_distinct_counts(nums1))  # Output: 15

# Example 2
nums2 = [1, 1]
print(sum_of_distinct_counts(nums2))  # Output: 3
```

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.
Example 1:
Input: nums = [3,1,2,2,2,1,3], k = 2
Output: 4
Explanation:
There are 4 pairs that meet all the requirements:
- nums[0] == nums[6], and 0 * 6 == 0, which is divisible by 2.
- nums[2] == nums[3], and 2 * 3 == 6, which is divisible by 2.
- nums[2] == nums[4], and 2 * 4 == 8, which is divisible by 2.
- nums[3] == nums[4], and 3 * 4 == 12, which is divisible by 2.
Example 2:
Input: nums = [1,2,3,4], k = 1
Output: 0
Explanation: Since no value in nums is repeated, there are no pairs (i,j) that meet all the requirements.

ALGORITHM:

1. Define a function count_pairs that takes a list of numbers, nums, and an integer k as input.
2. Initialize a variable count to 0.
3. Iterate through the indices of the input list using a loop with index i.
4. Within the first loop, iterate through the indices starting from i+1.
5. Check if elements at indices i and j are equal and if (i * j) % k == 0.
6. If conditions are met, increment count.
7. Return the final count.

CODE:

```
def count_pairs(nums, k):
    count = 0
    for i in range(len(nums)):
        for j in range(i+1, len(nums)):
            if nums[i] == nums[j] and (i * j) % k == 0:
                count += 1
    return count

# Example 1
nums1 = [3, 1, 2, 2, 2, 1, 3]
k1 = 2
output1 = count_pairs(nums1, k1)
print(output1)  # Output: 4

# Example 2
nums2 = [1, 2, 3, 4]
k2 = 1
output2 = count_pairs(nums2, k2)
print(output2)  # Output: 0
```

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

ALGORITHM:

```
Define a function find_duplicates that takes a list of numbers, nums, as input.
2. Initialize empty sets seen and duplicates.
3. Iterate through each number in nums.
4. Add the number to duplicates if it's in seen; otherwise, add it to seen.
5. Return a list of elements in duplicates.
```

CODE:

```
def find_max_element(input_list):
    return max(input_list)

# Test Cases
test_case_1 = [1, 2, 3, 4, 5]
test_case_2 = [7, 7, 7, 7, 7]
test_case_3 = [-10, 2, 3, -4, 5]

# Expected Outputs
output_1 = find_max_element(test_case_1)
output_2 = find_max_element(test_case_2)
output_3 = find_max_element(test_case_3)

print(output_1)
print(output_2)
print(output_3)
```

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.

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**

ALGORITHM:

1. Define a function sort_and_find_max that takes a list of numbers, input_list, as input.
2. Sort the input list to get sorted_list.
3. Retrieve the last element of sorted_list as max_element.
4. Return max_element as the output.

CODE:

```
def find_max_in_sorted_list(input_list):
    if not input_list:
        return None  # or appropriate message indicating empty list

    sorted_list = sorted(input_list)
    return sorted_list[-1]

assert find_max_in_sorted_list([]) == None

assert find_max_in_sorted_list([5]) == 5

assert find_max_in_sorted_list([3, 3, 3, 3, 3]) == 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?

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]**

ALGORITHM:

Define a function get_unique_elements that takes a list of numbers, input_list, as input.
2. Convert the input list to a set to remove duplicates.
3. Convert the set back to a list to maintain the original order of unique elements.
4. Return the list of unique elements

CODE:

```
def get_unique_elements(input_list):
    return list(set(input_list))

# Test Cases
input1 = [3, 7, 3, 5, 2, 5, 9, 2]
input2 = [-1, 2, -1, 3, 2, -2]
input3 = [1000000, 999999, 1000000]

output1 = get_unique_elements(input1)
output2 = get_unique_elements(input2)
output3 = get_unique_elements(input3)
```

```
print("Output 1:", output1)
print("Output 2:", output2)
print("Output 3:", output3)
```

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

ALGORITHM:
1. Define a function bubble_sort that takes an array of integers as input.
2. Implement nested loops to compare and swap adjacent elements.
3. Repeat the process until the array is sorted.
4. Return the sorted array.
5. The time complexity of bubble sort is O(n^2) in the worst case.

CODE:
```
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

# Test the bubble sort function
arr = [64, 34, 25, 12, 22, 11, 90]
sorted_arr = bubble_sort(arr)
print("Sorted array:", sorted_arr)
```

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

Test Case:
Example X={ 3,4,6,-9,10,8,9,30} KEY=10
Output: Element 10 is found at position 5

Example X={ 3,4,6,-9,10,8,9,30} KEY=100
Output : Element 100 is not found

ALGORITHM:
```
1. Define a function binary_search(arr, x, n) where arr is the sorted array, x is the number to search, and n is the size of the array.
2. Initialize start to 0 and end to n-1.
3. While start is less than or equal to end:
  a. Calculate the middle index as (start + end) // 2.
  b. If the middle element is equal to x, return its position.
  c. If x is greater than the middle element, update start to mid + 1.
  d. If x is less than the middle element, update end to mid - 1.
4. If the element is not found after the loop, return it was not found.
```

CODE:
```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] < x:
            low = mid + 1
        elif arr[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1

# Test Case
arr = [3, 4, 6, -9, 10, 8, 9, 30]
x = 10
result = binary_search(arr, x)
if result != -1:
    print(f"Element {x} is found at position {result}")
else:
    print(f"Element {x} is not found")
```

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

ALGORITHM:

1. Implement Merge Sort algorithm.
2. Define a function merge_sort(nums, left, right).
3. Recursively call merge_sort for left and right halves.
4. Merge the sorted halves using a merge function.
5. Copy the sorted elements back to the original array.

CODE:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[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

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

    return result

# Example Usage
nums = [12, 11, 13, 5, 6, 7]
sorted_nums = merge_sort(nums)
print(sorted_nums)
```

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.

Example:

· Input: **m=2,n=2,N=2,i=0,j=0** · Output: **6**

·Input: **m=1,n=3,N=3,i=0,j=1** · Output: **12**

ALGORITHM:

```
1. Create a function findPaths(m, n, N, i, j).
2. Initialize a 3D array dp of dimensions (m x n x N+1).
3. Set base case: dp[x][y][k] = 1 if out of bounds.
4. Use dynamic programming to calculate ways to reach each cell in N steps.
5. Iterate steps 1 to N and update dp based on neighbors.
6. Sum up ways to move the ball out of the grid from (i, j) in N steps.
```

CODE:

```
def findPaths(m, n, N, i, j):
    MOD = 10**9 + 7
    dp = [[[0] * n for _ in range(m)] for _ in range(N + 1)]
    for step in range(1, N + 1):
        for x in range(m):
            for y in range(n):
                for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                    nx, ny = x + dx, y + dy
                    if nx < 0 or nx >= m or ny < 0 or ny >= n:
                        dp[step][x][y] += 1
                    else:
                        dp[step][x][y] = (dp[step][x][y] + dp[step - 1][nx][ny]) % MOD
    return dp[N][i][j]

# Test Cases
print(findPaths(2, 2, 2, 0, 0))  # Output: 6
print(findPaths(1, 3, 3, 0, 1))  # Output: 12
```

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

ALGORITHM:

```
1. Define a function rob(nums) to find the maximum money to rob.
2. If the number of houses is <= 3, return the max amount in the array.
3. Calculate two scenarios:
   - Rob from the first house to the second-to-last house.
   - Rob from the second house to the last house.
4. Return the max amount from the two scenarios.
```

CODE:

```
def rob(nums):
    def rob_range(start, end):
        rob_next, rob_curr = 0, 0
        for i in range(start, end):
            rob_next, rob_curr = max(rob_curr + nums[i], rob_next), rob_next
        return rob_next

    if len(nums) == 1:
        return nums[0]
    return max(rob_range(0, len(nums) - 1), rob_range(1, len(nums)))

# Examples
nums1 = [2, 3, 2]
```

```
print("The maximum money you can rob without alerting the police is", rob(nums1))

nums2 = [1, 2, 3, 1]
print("The maximum money you can rob without alerting the police is", rob(nums2))
```

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?
Examples:
(i) Input: n=4   Output: 5
(ii) Input: n=3 Output: 3

ALGORITHM:
1. Create a function climbStairs(n) to find the distinct ways to climb the staircase.
2. Initialize an array dp to store the ways to reach each step.
3. Set base cases: dp[0] = 1 and dp[1] = 1.
4. Iterate from 2 to n and calculate dp[i] = dp[i-1] + dp[i-2].
5. Return dp[n] as the total distinct ways to climb to the top.

CODE:

```
def climbStairs(n):
    if n == 1:
        return 1
    first, second = 1, 2
    for i in range(3, n + 1):
        third = first + second
        first = second
        second = third
    return second

# Examples
print(climbStairs(4))  # Output: 5
print(climbStairs(3))  # Output: 3
```

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 in time. The robot is trying to reach the bottom-right corner of the grid. How many possible unique paths are there?
Examples:
(i) Input: m=7,n=3   Output: 28
(ii) Input: m=3,n=2   Output: 3

ALGORITHM:
1. Create a function uniquePaths(m, n) to find the unique paths.
2. Initialize a 2D array dp to store the number of paths for each cell.
3. Set base cases: dp[0][0] = 1, initialize the first row and column with 1.
4. Iterate over the grid updating dp[i][j] = dp[i-1][j] + dp[i][j-1].
5. Return dp[m-1][n-1] as the total unique paths to reach the bottom-right corner.

CODE:

```
def unique_paths(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]
```

```
# Example 1
m1, n1 = 7, 3
output1 = unique_paths(m1, n1)
print(output1)  # Output: 28

# Example 2
m2, n2 = 3, 2
output2 = unique_paths(m2, n2)
print(output2)  # Output: 3
```

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.

Example 1:
Input: s = "abbxxxxzzy"
Output: [[3,6]]
Explanation: "xxxx" is the only large group with start index 3 and end index 6.

Example 2:
Input: s = "abc"
Output: []
Explanation: We have groups "a", "b", and "c", none of which are large groups

ALGORITHM:
1. Create a function largeGroupPositions(s) to find the intervals of large groups.
2. Iterate through the string s to track the start and end indices of each group.
3. Check if the group is large (3 or more characters) and add its interval [start, end] to the result list.
4. Return the list of intervals of large groups sorted by the start index.

CODE:

```
def large_group_positions(s):
    result = []
    i = 0
    n = len(s)

    while i < n:
        j = i
        while j < n and s[j] == s[i]:
            j += 1
        if j - i >= 3:
            result.append([i, j - 1])
        i = j

    return result

# Example usage
s1 = "abbxxxxzzy"
s2 = "abc"

print(large_group_positions(s1))  # Output: [[3, 6]]
print(large_group_positions(s2))  # Output: []
```

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.

1. **Any live cell with two or three live neighbors lives on to the next generation.**
2. **Any live cell with more than three live neighbors dies, as if by over-population.**
3. **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*.

Example 1:

0	1	0
0	0	1
1	1	1
0	0	0

⇨

0	0	0
1	0	1
0	1	1
0	1	0

ALGORITHM:

161. Create a function gameOfLife(board) to calculate the next state of the board.
2. Iterate through each cell in the board and count live neighbors.
3. Apply the rules to determine the next state of each cell based on live neighbor count.
4. Update the board with the next state.
5. Return the updated board as the next state of the Game of Life.

CODE:

```
def game_of_life(board):
    # Neighbors array to find 8 neighboring cells for a given cell
    neighbors = [(1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1), (0, -1), (1, -1)]

    rows = len(board)
    cols = len(board[0])

    # Create a copy of the original board to keep track of the updates
    copy_board = [[board[row][col] for col in range(cols)] for row in range(rows)]

    for row in range(rows):
        for col in range(cols):
            live_neighbors = 0
            for neighbor in neighbors:
                r = row + neighbor[0]
                c = col + neighbor[1]
                if (0 <= r < rows) and (0 <= c < cols) and (copy_board[r][c] == 1):
                    live_neighbors += 1

            # Apply the Game of Life rules
            if copy_board[row][col] == 1 and (live_neighbors < 2 or live_neighbors > 3):
                board[row][col] = 0
            if copy_board[row][col] == 0 and live_neighbors == 3:
                board[row][col] = 1

    return board

# Example usage
board = [
    [0, 1, 0],
    [0, 0, 1],
    [1, 1, 1],
    [0, 0, 0]
]

new_state = game_of_life(board)
print(new_state)
```

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.

ALGORITHM:

1. Create a 2D array to represent the champagne distribution in the glasses.

2. Simulate pouring champagne into the glasses, distributing excess equally to adjacent glasses.

3. Iterate through each row and glass to determine the fill level of the specified glass.

4. Return the fill level of the jth glass in the ith row after pouring the desired amount of champagne.

CODE:

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

    glasses = [[0] * k for k in range(1, 102)]

    glasses[0][0] = poured

    for i in range(100):

        for j in range(i + 1):

            q = (glasses[i][j] - 1) / 2.0

            if q > 0:

                glasses[i + 1][j] += q

                glasses[i + 1][j + 1] += q

    return min(1, glasses[query_row][query_glass])
```