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"

```
def palindromic(words):
   for word in words:
      if word==word[::-1]:
         return word
   return ""
words=["abc","car","ada","rececar"]
print(palindromic(words))
```

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

```
def count(num1,num2):
   a1=sum(1 for num in num1 if num in num2)
   a2=sum(1 for num in num2 if num in num1)
   return[a1,a2]
num1=[2,3,2]
num2=[1,2]
print(count(num1,num2))
```

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.
Input: nums = [1,2,1]
Output: 15

```
def sum_of_squares_of_distinct_counts(nums):
   n = len(nums)
```

```
    result = 0
    for i in range(n):
        distinct = set()
        for j in range(i, n):
            distinct.add(nums[j])
            result += len(distinct) ** 2
    return result
nums1 = [1, 2, 1]
print(sum_of_squares_of_distinct_counts(nums1))
```

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.
Input: nums = [3,1,2,2,2,1,3], k = 2
Output: 4

```
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
nums1 = [3, 1, 2, 2, 2, 1, 3]
k1 = 2
print(count_pairs(nums1, k1))
```

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

```
def find(nums):
    if not nums:
        return None
    return max(nums)
```

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

print(find(nums))
```

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**

```
l=[5]
l1=sorted(l)
print(l1[-1])
```

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]**

```
def unique(num):
   return list(set(num))

num=[3,7,3,5,2,5,9,2]

print(unique(num))
```

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

```
def bubble(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=[64,34,25,12,22,11,90]
print(bubble(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

```
def binary(arr,x,low,high):
    while low<=high:
        mid=low+(high-low)//2
        if arr[mid]==x:
            return mid
        elif arr[mid]<x:
            low=mid+1
        else:
            high=mid-1
    return -1
arr=[3,4,6,9,10,8,9,30]
x=10
result=binary(arr,x,0,len(arr)-1)
if result!=-1:
    print("element is found at index"+str(result))
else:
    print("element 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.

```
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
    return arr
arr=[12,11,13,5,6,7]
sorted=merge_sort(arr)
print(sorted)
```

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**
- **I**nput**: m=1,n=3,N=3,i=0,j=1** - Output**: 12**

```
def find_paths(m, n, N, i, j):
    memo = {}
    def dp(x, y, remaining_steps):
        if x < 0 or x >= m or y < 0 or y >= n:
            return 1
        if remaining_steps == 0:
            return 0
        if (x, y, remaining_steps) in memo:
            return memo[(x, y, remaining_steps)]
        ways = (dp(x + 1, y, remaining_steps - 1) +
                dp(x - 1, y, remaining_steps - 1) +
                dp(x, y + 1, remaining_steps - 1) +
                dp(x, y - 1, remaining_steps - 1))
        memo[(x, y, remaining_steps)] = ways
        return ways
    return dp(i, j, N)
print(find_paths(2, 2, 2, 0, 0))
```

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.

Examples:

(i) Input : nums = [2, 3, 2]
Output : The maximum money you can rob without alerting the police is 3(robbing house 1).

```
def rob(nums):
    def rob_linear(houses):
        prev, curr = 0, 0
```

```
        for money in houses:
            prev, curr = curr, max(curr, prev + money)
        return curr

    if len(nums) == 1:
        return nums[0]
    return max(rob_linear(nums[1:]), rob_linear(nums[:-1]))
print(rob([2, 3, 2]))
```

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

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

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

```
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))
```

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

```
def largeGroupPositions(s):

    result = []

    i = 0

    while i < len(s):

        start = i

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

            i += 1

        if i - start >= 3:

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

    return result

print(largeGroupPositions("abbxxxxzzy"))
```

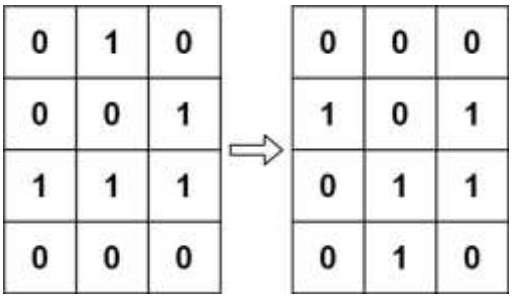
16."The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970." The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts 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.**

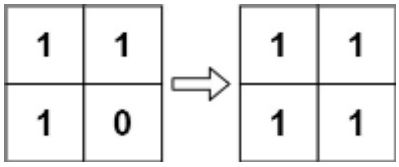
The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return *the next state*.

Example 1:



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]]

Example 2:



Input: board = [[1,1],[1,0]]
Output: [[1,1],[1,1]]

```
def game_of_life(board):
    def count_live_neighbors(board, i, j):
        count = 0
        for x in range(-1, 2):
            for y in range(-1, 2):
                if x == 0 and y == 0:
                    continue
                if 0 <= i + x < len(board) and 0 <= j + y < len(board[0]):
                    count += board[i + x][j + y] & 1
        return count
    m, n = len(board), len(board[0])
    for i in range(m):
        for j in range(n):
            live_neighbors = count_live_neighbors(board, i, j)
```

```
            if board[i][j] == 1 and (live_neighbors < 2 or live_neighbors > 3):
                board[i][j] = 2
            if board[i][j] == 0 and live_neighbors == 3:
                board[i][j] = -1
    for i in range(m):
        for j in range(n):
            board[i][j] >>= 1
    return board
board1 = [[0, 1, 0], [0, 0, 1], [1, 1, 1], [0, 0, 0]]
print(game_of_life(board1))
```

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.)

Example 1:

Input: poured = 1, query_row = 1, query_glass = 1
Output: 0.00000
Explanation: We poured 1 cup of champange to the top glass of the tower (which is indexed as (0, 0)). There will be no excess liquid so all the glasses under the top glass will remain empty.

Example 2:

Input: poured = 2, query_row = 1, query_glass = 1
Output: 0.50000
Explanation: We poured 2 cups of champange to the top glass of the tower (which is indexed as (0, 0)). There is one cup of excess liquid. The glass

indexed as (1, 0) and the glass indexed as (1, 1) will share the excess liquid equally, and each will get half cup of champange.

```
def champagneTower(poured, query_row, query_glass):
    A = [[0] * k for k in range(1, 102)]
    A[0][0] = poured
    for i in range(query_row + 1):
        for j in range(i + 1):
            q = (A[i][j] - 1.0) / 2.0
            if q > 0:
                A[i + 1][j] += q
                A[i + 1][j + 1] += q
    return min(1, A[query_row][query_glass])
```