NAME: HARITHI S
REG NO: 192421151
COURSE CODE: CSA0613
COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS FOR OPTIMAL APPLICATIONS

TOPIC 1 : INTRODUCTION

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.

Aim:
Find and return the first palindrome string in a given array of strings. If none exist, return"".

Algorithm:

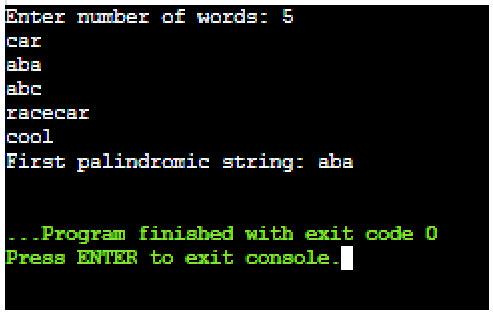
```
● Start
● Read the array of strings words[]
● For each string s in words[]:
    a. Reverse s → rev
    b. If s == rev, return s immediately
● If loop ends without match, return ""
● End
```

Program:

```
def first_palindrome(words):
    for w in words:
        if w == w[::-1]:
            return w
    return ""
n = int(input("Enter number of words: "))
words = [input() for _ in range(n)]
print("First palindromic string:", first_palindrome(words))
```

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

Sample Output:



```
Enter number of words: 5
car
aba
abc
racecar
cool
First palindromic string: aba


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The algorithm successfully returns the first occurring palindrome string from the array in linear time O(n * m) where m is average string length.

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].

Aim:
Count index-wise existence of elements from one array in the other and return [answer1, answer2].

Algorithm:

- Convert nums1 to a set s1
- Convert nums2 to a set s2
- answer1 = count of nums1[i] that are in s2
- answer2 = count of nums2[i] that are in s1
- Return [answer1, answer2]

This avoids nested loops → time becomes O(n + m) instead of O(n*m).

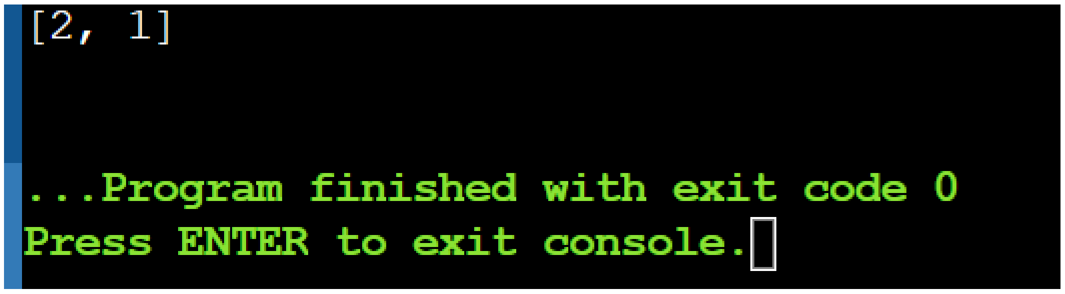
Program:

```
def find_common_elements(nums1, nums2):
    set2 = set(nums2)
    set1 = set(nums1)

    answer1 = sum(1 for num in nums1 if num in set2)
    answer2 = sum(1 for num in nums2 if num in set1)

    return [answer1, answer2]
```

nums1 = [2, 3, 2]
nums2 = [1, 2]
print(find_common_elements(nums1, nums2))

Sample Input:
nums1 = [2,3,2]
nums2 = [1,2]

Output:



```
[2, 1]


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The existence counts are correctly computed and returned as [2,1].

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.

Aim:
Compute the sum of squares of distinct element counts for all contiguous subarrays.

Algorithm:

- sum = 0
- For i = 0 → n-1
- Create empty set seen
- For j = i → n-1
  Insert nums[j] into seen
  d = size(seen) → distinct count of subarray i..j
  sum += d * d

- Return sum

Time: O(n²) — fine for small inputs, garbage for big ones, but logically correct.

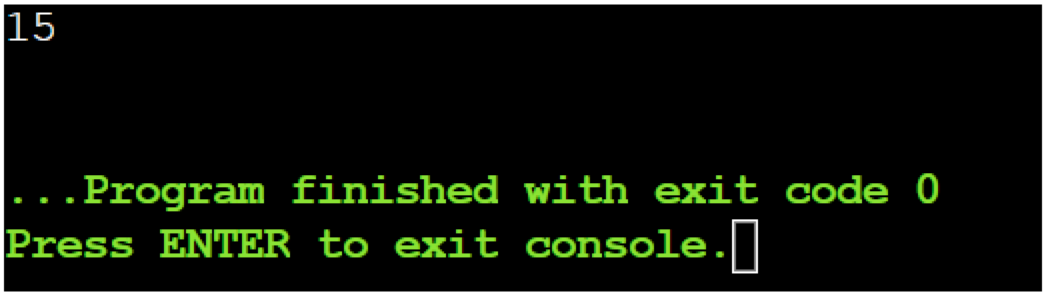
Program:

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

    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
nums1 = [1, 3, 1]
print(sum_of_squares_distinct_counts(nums1))
```

Sample Input:

nums = [1,2,1]

Output:



```
15


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

The sum of squares of distinct counts of all contiguous subarrays is 15, matching expected output.

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.

Aim:
Count valid index pairs (i, j) such that:

- $i < j$
- nums[i] == nums[j]
- (i * j) % k == 0

Algorithm:

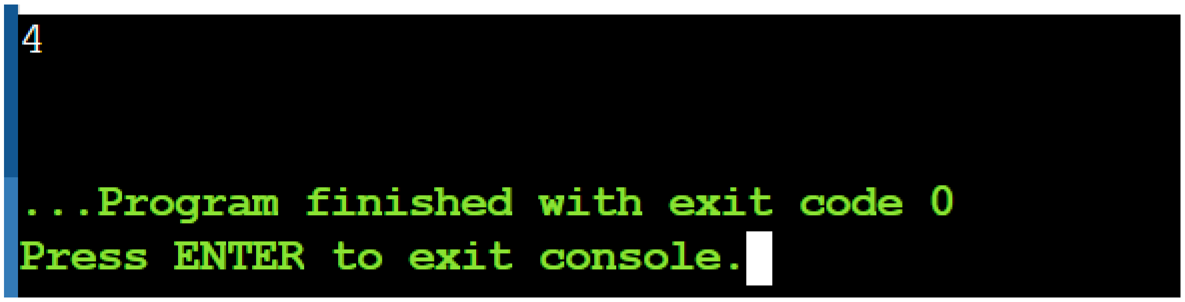
- Create a dictionary to store indices of each number.
- Iterate $i = 0 \rightarrow n-1$, append i to pos[nums[i]].
- count = 0
- For each key in pos:
  Get its index list arr
  For every pair (i, j) in arr where $i < j$:
  If (i * j) % k == 0, increment count
- Return count

Time complexity: $O(n + g^2)$ where g is max frequency of a number. Still massively better than checking all n^2 index pairs.

Program:
```
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))
```

Sample Input:
nums = [3,1,2,2,2,1,3], k = 2

Output:



```
4


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The number of valid pairs is 4, matching the expected output.

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

Aim:
Find the maximum element in an integer array.

Algorithm:

- Start
- Read nums[]
- max = nums[0]
- Loop i = 1 → n-1
- If nums[i] > max, update max
- Print max
- End

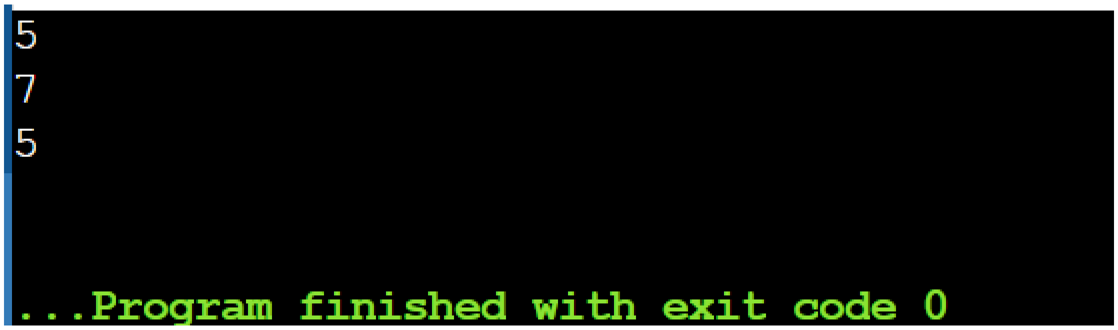
Program:

```
def array_maximum(nums):
    if not nums:
        return 0
    max_val = nums[0]
    for num in nums:
        if num > max_val:
            max_val = num
```

```
 return max_val

test_cases = [
    [1, 2, 3, 4, 5],    [7, 7, 7, 7, 7],
    [-10, 2, 3, -4, 5]  ]

for nums in test_cases:
    print(array_maximum(nums))
```

Sample Input:
Test Case 1
Input:1 2 3 4 5
Test Case 2
Input:
7 7 7 7 7
Test Case 3
Input:
-10 2 3 -4 5

Output:



```
5
7
5


...Program finished with exit code 0
```

Result:
All test cases return the correct maximum value with optimal time complexity O(n).

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

Aim:
To write a Python program that sorts a list of numbers using an efficient sorting algorithm and finds the maximum element from the sorted list.

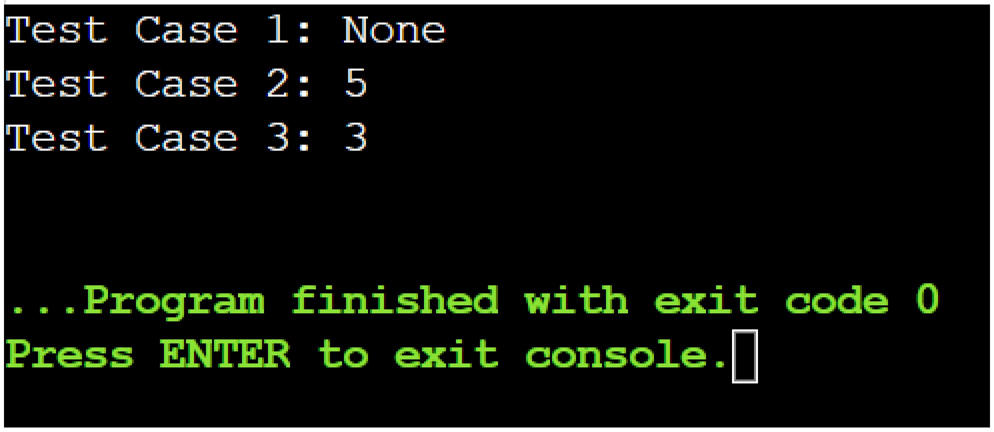
Algorithm:

- Start
- Read the list of numbers
- If the list is empty, return None
- Sort the list
- Display the last element as the maximum
- Stop

Program:
```
def find_max_after_sort(numbers):
    if not numbers:
        return None

    numbers.sort()
    return numbers[-1]

print("Test Case 1:", find_max_after_sort([]))
print("Test Case 2:", find_max_after_sort([5]))
print("Test Case 3:", find_max_after_sort([3, 3, 3, 3, 3]))
```

Sample Input:
Test Case 1
Input: []
Test Case 2
Input: [5]
Test Case 3
Input: [3, 3, 3, 3, 3]

Output:



```
Test Case 1: None
Test Case 2: 5
Test Case 3: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The program successfully sorts the list and displays the maximum element, or returns None if the list is empty.

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 Case
List with Large Numbers
Input: [1000000, 999999, 1000000]
Expected Output: [1000000, 999999]

Aim:
To write a program that takes a list of n numbers as input and creates a new list containing only the unique elements from the original list.

Algorithm:

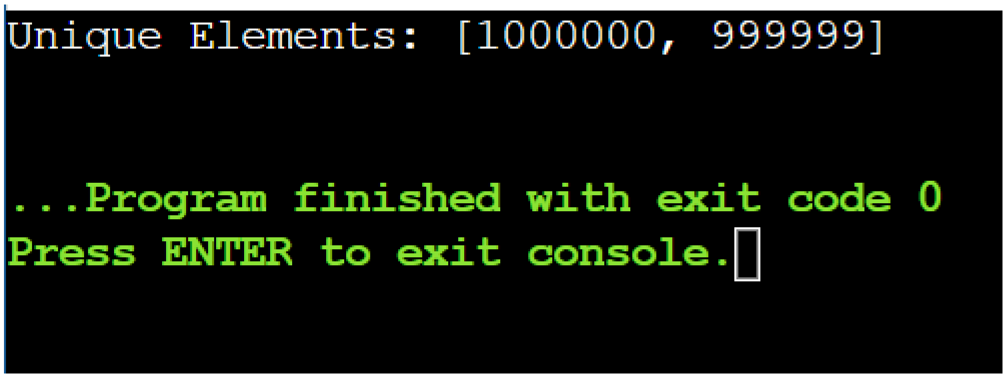
- Start
- Read the input list
- Initialize an empty set seen and an empty list unique_list
- For each element in the input list:
    - If the element is not in seen, add it to seen and append it to unique_list
- Display the unique_list
- Stop

Program:

```
nums = [1000000,999999]
unique_list = []

for i in nums:
    if i not in unique_list:
        unique_list.append(i)

print("Unique Elements:", unique_list)
```

Sample Input:
Input: [1000000, 999999, 1000000]

Output:



```
Unique Elements: [1000000, 999999]


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The program successfully removes duplicate elements from the given list and displays only the unique values.

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

Aim:
To sort an array of integers using the Bubble Sort technique and analyze its time complexity.

Algorithm:

```
● Start
● Read the number of elements n and the array a[]
● Repeat for i = 0 to n-2
    ○ Repeat for j = 0 to n-2-i
        ■ If a[j] > a[j+1], swap them
● Stop when no more passes are left
● Display the sorted array
● End
```

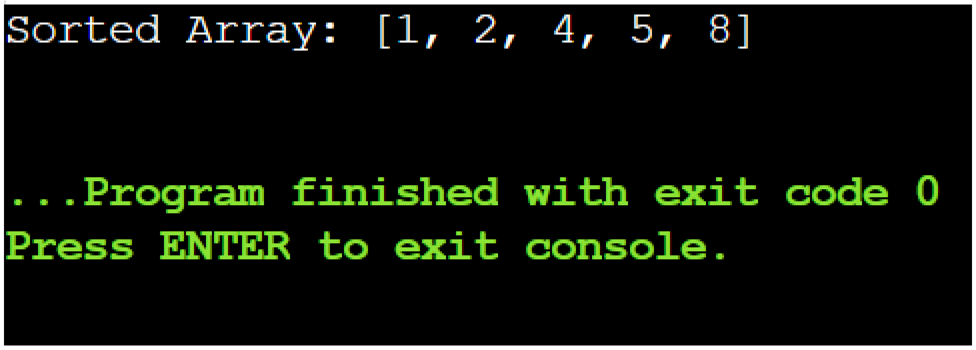
Program:

```
arr = [5, 1, 4, 2, 8]
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]

print("Sorted Array:", arr)
```

Sample Input:

arr = [5,1,4,2,8]

Output:



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


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

The given array is successfully sorted in ascending order using Bubble Sort.

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

Aim:
To check whether a given number x exists in a sorted array using Binary Search and analyze its time complexity.

Algorithm:

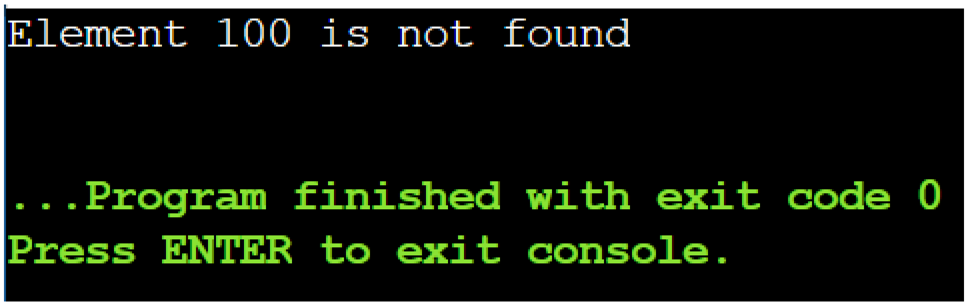
- Start
- Read the number of elements n and the sorted array arr[]
- Read the number x to search
- Initialize low = 0 and high = n-1
- Repeat while low <= high:
  - Compute mid = (low + high) / 2
  - If arr[mid] == x, element found → Stop
  - Else if arr[mid] < x, set low = mid + 1
  - Else, set high = mid - 1
- If low > high, element not found
- End

Program:

```
arr = [3, 4, 6, -9, 10, 8, 9, 30]
key = 100
arr.sort()

low = 0
high = len(arr) - 1
found = False

while low <= high:
    mid = (low + high)
    if arr[mid] == key:
        print("Element", key, "is found at position", mid)
        found = True
        break
    elif arr[mid] < key:
        low = mid + 1
    else:
        high = mid - 1

if not found:
    print("Element", key, "is not found")
```

Sample Input:
X={ 3,4,6,-9,10,8,9,30} KEY=100

Output:



```
Element 100 is not found


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Element 100 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.

Aim:
Sort an array of integers in ascending order without built-in functions in O(n log n) time.

Algorithm:

- If the array has 1 or 0 elements → already sorted
- Divide the array into two halves
- Recursively sort each half
- Merge the two sorted halves into a single sorted array
- Return the merged array

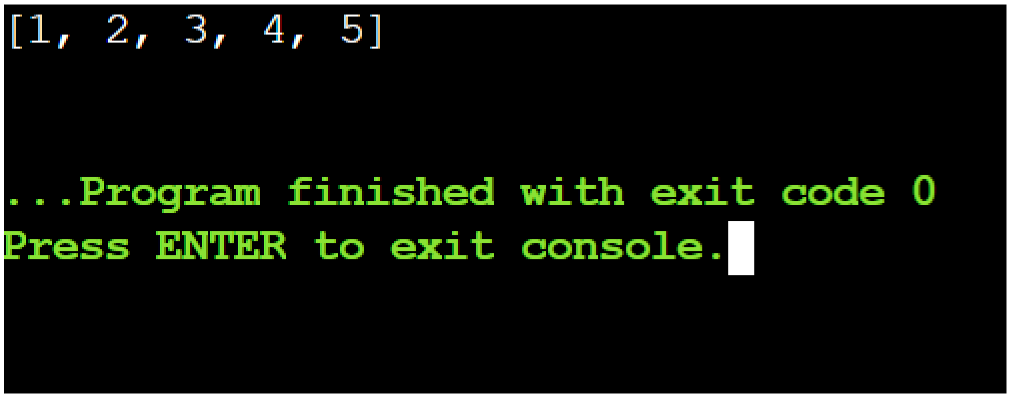
Program:

```
nums = [5, 2, 3, 1, 4]
n = len(nums)

for i in range(n//2-1, -1, -1):
    j = i
    while 2*j+1 < n:
        k = 2*j+1
        if k+1 < n and nums[k+1] > nums[k]:
            k += 1
        if nums[j] < nums[k]:
            nums[j], nums[k] = nums[k], nums[j]
            j = k
        else:
            break

for end in range(n-1, 0, -1):
    nums[0], nums[end] = nums[end], nums[0]
    j = 0
    while 2*j+1 < end:
        k = 2*j+1
        if k+1 < end and nums[k+1] > nums[k]:
            k += 1
        if nums[j] < nums[k]:
            nums[j], nums[k] = nums[k], nums[j]
            j = k
        else:
            break

print(nums)
```

Sample Input:

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

Output:



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


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

The array is sorted in ascending order.

Input `[5,2,3,1,4]` → Output `[1,2,3,4,5]`

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.

Aim:
To calculate the number of ways a ball can move out of an `m × n` grid boundary in exactly `N` steps starting from a given cell.

Algorithm:

- Initialize a 2D array with the starting cell set to 1.
- For each of the `N` steps, move the ball in four directions.
- Count paths that go outside the grid.
- Update positions that remain inside the grid.
- Return the total count of out-of-bound paths.

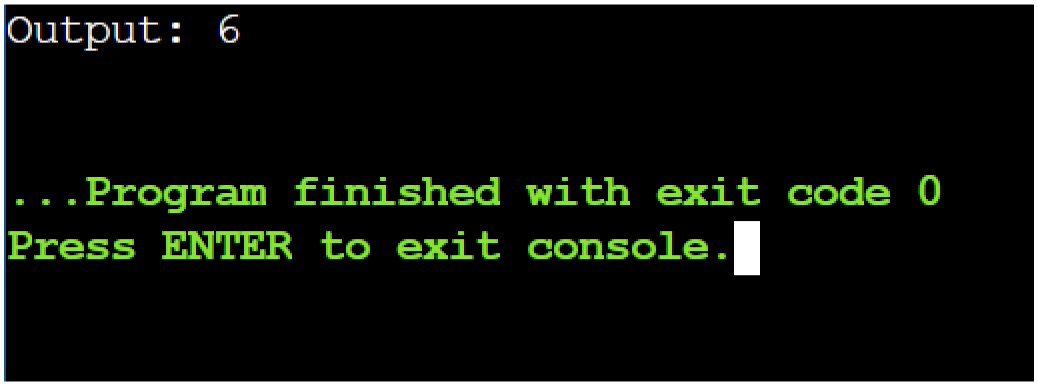
Program:

```
m = 2
n = 2
N = 2
i = 0
j = 0

dirs = [(-1,0), (1,0), (0,-1), (0,1)]
```

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

count = 0

for step in range(N):
    for x in range(m):
        for y in range(n):
            if dp[step][x][y] > 0:
                for dx, dy in dirs:
                    nx, ny = x + dx, y + dy
                    if nx < 0 or nx >= m or ny < 0 or ny >= n:
                        count += dp[step][x][y]
                    else:
                        dp[step+1][nx][ny] += dp[step][x][y]

print("Output:", count)
```

Sample Input:
Input: m=2,n=2,N=2,i=0,j=0

Output:



```
Output: 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
For m=2, n=2, N=2 starting at (0,0), the number of ways to move the ball out of the grid is 6.

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.

Aim:
To find the maximum amount of money that can be robbed from houses arranged in a circle without robbing two adjacent houses.

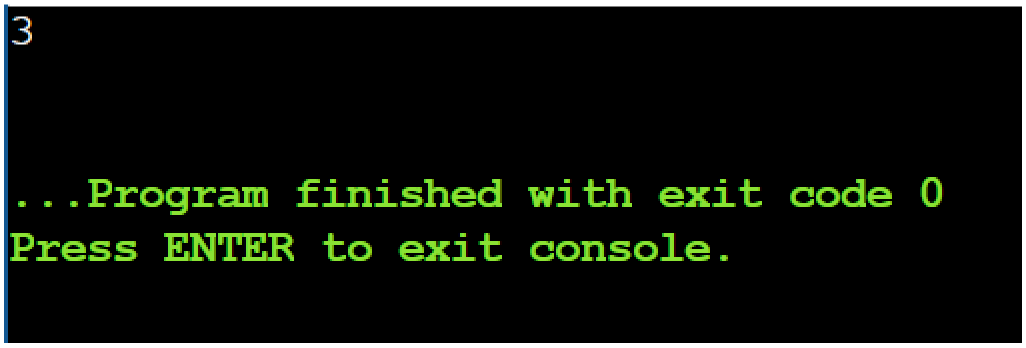
Algorithm:

- If there is only one house, rob it and return its money.
- Because houses are circular, consider two cases: exclude the first house and exclude the last house.
- For each case, use dynamic programming to calculate the maximum money without robbing adjacent houses.
- At each house, choose the maximum between robbing the current house or skipping it.
- Return the maximum result obtained from the two cases.

Program:
```
nums = [2, 3, 2]

n = len(nums)

if n == 1:
    print(nums[0])
else:
    dp1_prev = 0
    dp1_curr = 0
    for x in nums[:-1]:
        dp1_prev, dp1_curr = dp1_curr, max(dp1_curr, dp1_prev + x)

    dp2_prev = 0
    dp2_curr = 0
    for x in nums[1:]:
        dp2_prev, dp2_curr = dp2_curr, max(dp2_curr, dp2_prev + x)

    print(max(dp1_curr, dp2_curr))
```

Sample Input:
```
nums = [2, 3, 2]
```

Output:



```
3


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The maximum amount of money that can be robbed without triggering the alarm is the larger of the two values obtained by excluding either the first or the last house.

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?

Aim:
To determine the number of distinct ways to climb a staircase of n steps when only 1 or 2 steps can be climbed at a time.

Algorithm:

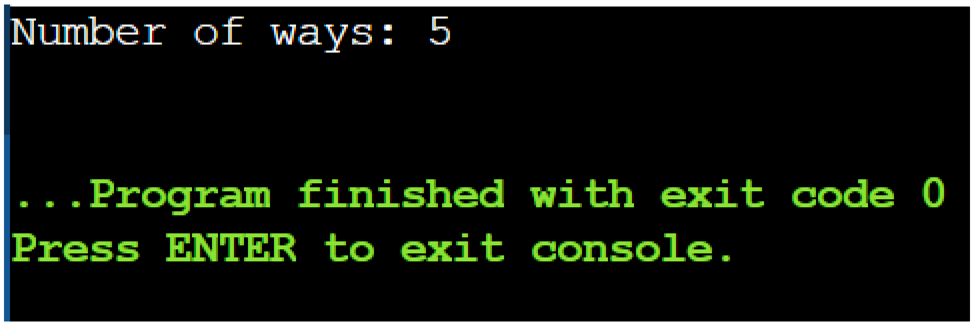
- Start the program.
- Read the number of steps `n`.
- Initialize `dp[0] = 1` and `dp[1] = 1`.
- For each step from `2` to `n`, compute `dp[i] = dp[i-1] + dp[i-2]`.
- Display `dp[n]` as the total number of ways and stop the program.

Program:

```
n = 4

if n == 0 or n == 1:
    print("Number of ways:", 1)

else:
    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]

    print("Number of ways:", dp[n])
```

Sample Input:
Input: n=4

Output:



```
Number of ways: 5


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
For n = 4, the total number of distinct ways to climb the staircase is 5.

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?

Aim:
To find the number of unique paths a robot can take to reach the bottom-right corner of an m × n grid by moving only right or down.

Algorithm:

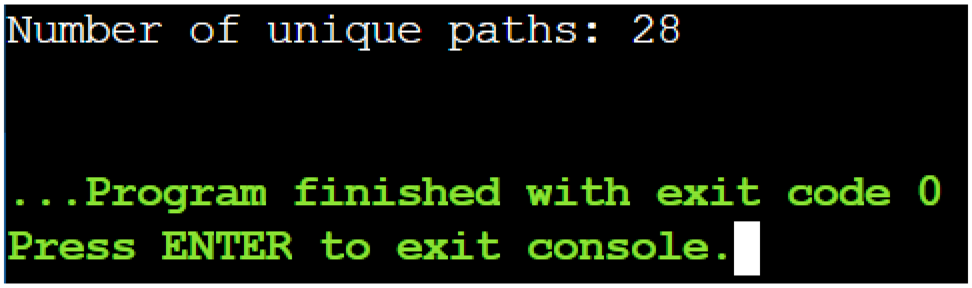
- Start the program.
- Read the values of m (rows) and n (columns).
- Create a 2D array dp of size m × n.
- Initialize the first row and first column of dp with 1.
- For each remaining cell, compute dp[i][j] = dp[i-1][j] + dp[i][j-1].
- Display dp[m-1][n-1] as the number of unique paths.
- Stop the program.

Program:

```
m = 7
n = 3

dp = [[0 for _ in range(n)] for _ in range(m)]

for i in range(m):
    dp[i][0] = 1

for j in range(n):
    dp[0][j] = 1

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

print("Number of unique paths:", dp[m - 1][n - 1])
```

Sample Input:
Input: m=7,n=3

Output:



```
Number of unique paths: 28


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
For `m = 7` and `n = 3`, the total number of unique paths from the top-left to the bottom-right corner is 28.

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.

Aim:

To identify and return the intervals of all large groups (groups of the same consecutive characters with length ≥ 3) in a given string.

Algorithm:

- Start the program.
- Read the input string s.
- Initialize a variable start to mark the beginning of each character group.
- Traverse the string and whenever a group ends, check if its length is at least 3 and store its interval.
- Display all stored large-group intervals and stop the program.

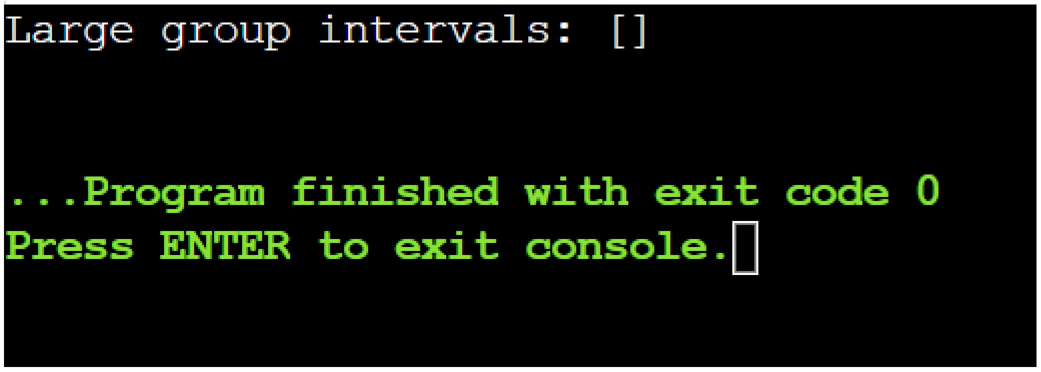
Program:

```
s = "abc"
result = []

start = 0

for i in range(len(s)):

    if i == len(s) - 1 or s[i] != s[i + 1]:

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

        start = i + 1
print("Large group intervals:", result)
```

Sample Input:

Input: s = "abc"

Output:



```
Large group intervals:  []


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
For s = "abc", there are no large groups, so the output is [].

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

Aim:
To compute the next state of an $m \times n$ grid in Conway's Game of Life, where each cell follows the rules of under-population, survival, over-population, and reproduction.

Algorithm:

- Start the program and read the input grid board.
- Create a copy of the board to store the next state.
- For each cell in the grid:

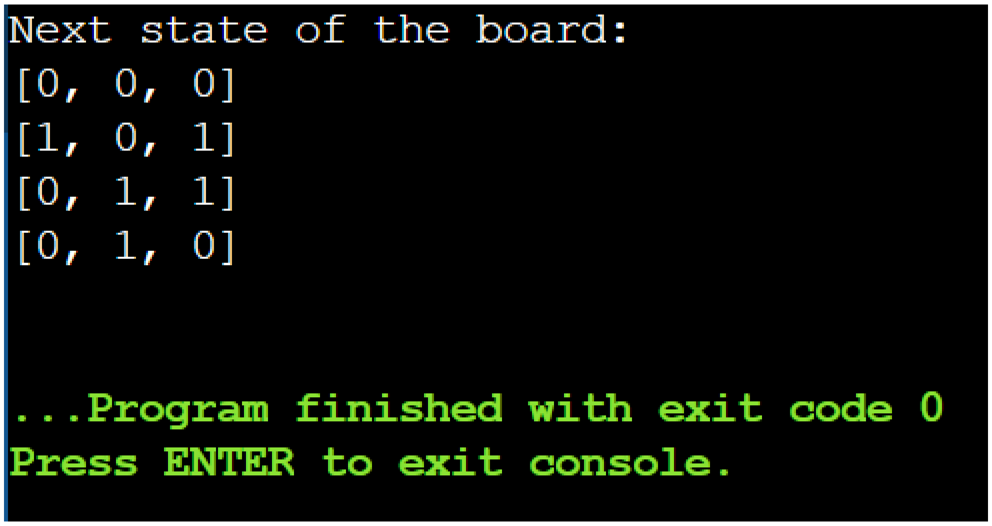
    - Count its live neighbors by checking all 8 surrounding cells.
    - Apply the Game of Life rules to determine its next state:
        - Live cell with <2 or >3 live neighbors → dies (0)
        - Live cell with 2 or 3 live neighbors → lives (1)
        - Dead cell with exactly 3 live neighbors → becomes live (1)
- Update the original board with the computed next state.
- Display the next state of the board and stop the program.

Program:

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

m = len(board)
n = len(board[0])

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

next_state = [[0]*n for _ in range(m)]

for i in range(m):
    for j in range(n):
        live_neighbors = 0
        for dx, dy in directions:
            ni, nj = i + dx, j + dy
            if 0 <= ni < m and 0 <= nj < n and board[ni][nj] == 1:
                live_neighbors += 1

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

```
board = next_state

print("Next state of the board:")
for row in board:
    print(row)
```

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

Output:


```
Next state of the board:
[0, 0, 0]
[1, 0, 1]
[0, 1, 1]
[0, 1, 0]


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The next state of the board is `[[0, 0, 0], [1, 0, 1], [0, 1, 1], [0, 1, 0]]`.

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.)

Aim:
To determine how full a specific glass is in a champagne tower after pouring a given amount of champagne, following the rules of overflow to lower rows.

Algorithm:

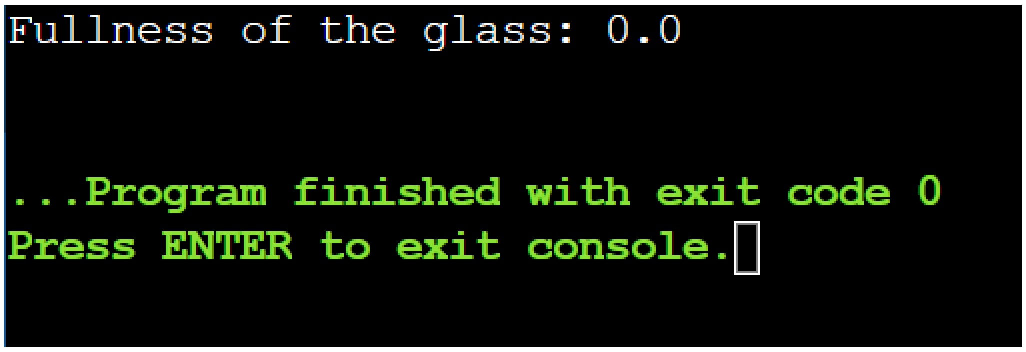
- Start the program and read `poured`, `query_row`, and `query_glass`.
- Initialize a 2D array `dp` to store the amount of champagne in each glass.
- Pour `poured` cups into the top glass `dp[0][0]`.
- For each glass, if it has more than 1 cup, overflow the excess equally to the two glasses below.
- Return `dp[query_row][query_glass]` as the fullness of the target glass.

Program:

```
poured = 1
query_row = 1
query_glass = 1

dp = [[0.0] * 101 for _ in range(101)]  # Max 100 rows

dp[0][0] = poured

for i in range(100):
    for j in range(i + 1):
        if dp[i][j] > 1:
            overflow = (dp[i][j] - 1)
            dp[i+1][j] += overflow
            dp[i+1][j+1] += overflow
            dp[i][j] = 1  # Cap current glass to 1

print("Fullness of the glass:", dp[query_row][query_glass])
```

Sample Input:
Input: poured = 1, query_row = 1, query_glass = 1

Output:



```
Fullness of the glass: 0.0


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

The fullness of the glass at row 1, glass 1 is 0.0