**Name : C.c.Jyoshna**

**Reg.no : 192411098**

**Course Code : CSA0610**

**Course : Design And Analysis of Algorithm**

**1. Aim:**

To find and return the **first palindromic string** in an array of strings. If no such string exists, return an empty string "".

**Algorithm:**

1. Start iterating over each string in the array words.
2. For each string, check if it is a palindrome:
   * A string is a palindrome if it reads the same forward and backward.
   * You can check this by comparing the string with its reverse.
3. If a palindrome is found, return that string immediately.
4. If the loop ends without finding any palindrome, return an empty string "".

**Input:**

* An array of strings words.

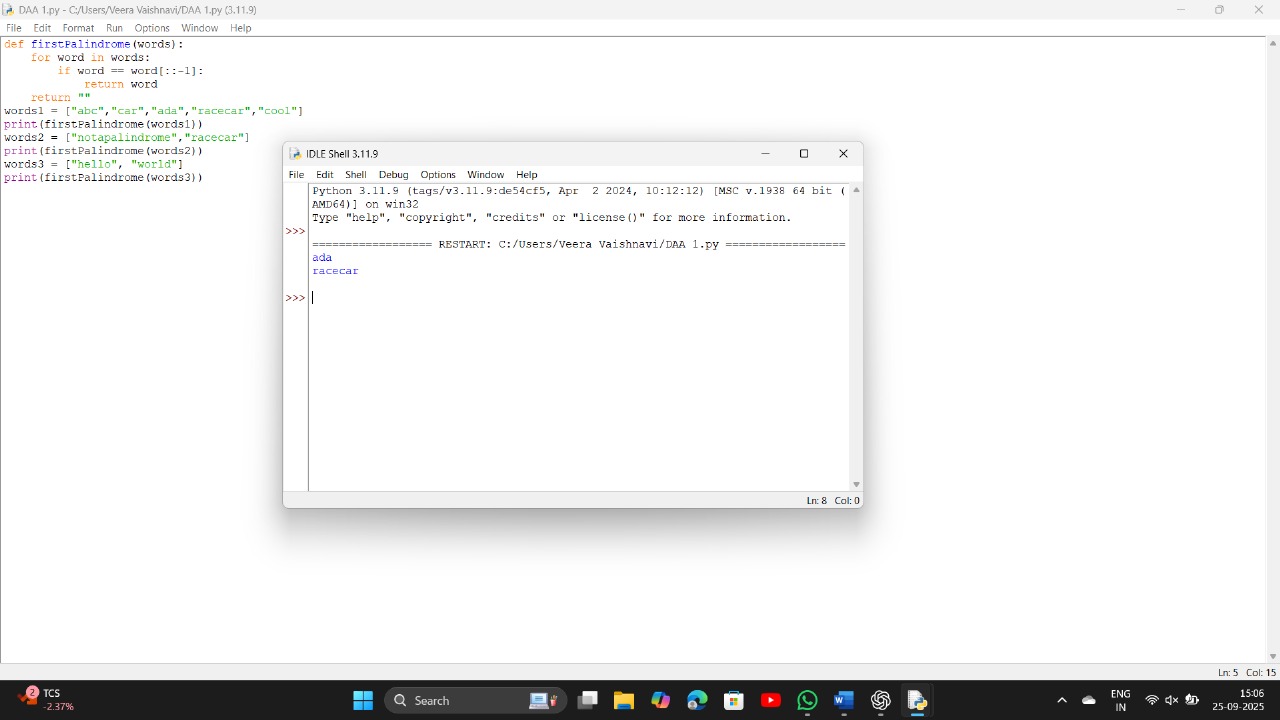
**Output:**

* The first palindromic string found in the array.
* If no palindrome exists, output is an empty string "".

**Result (Example):**

Example 1:  
Input: ["abc","car","ada","racecar","cool"]  
Output: "ada"  
Explanation: "ada" is the first string that is a palindrome.

Example 2:  
Input: ["notapalindrome","racecar"]  
Output: "racecar"



2.

**Aim:**

Count how many elements in nums1 are found in nums2 (answer1), and how many elements in nums2 are found in nums1 (answer2).

**Algorithm:**

1. Make a set from nums2.
2. Count elements in nums1 that are in the nums2 set → answer1.
3. Make a set from nums1.
4. Count elements in nums2 that are in the nums1 set → answer2.
5. Return [answer1, answer2].

**Input:**

nums1 = [4, 3, 2, 3, 1]

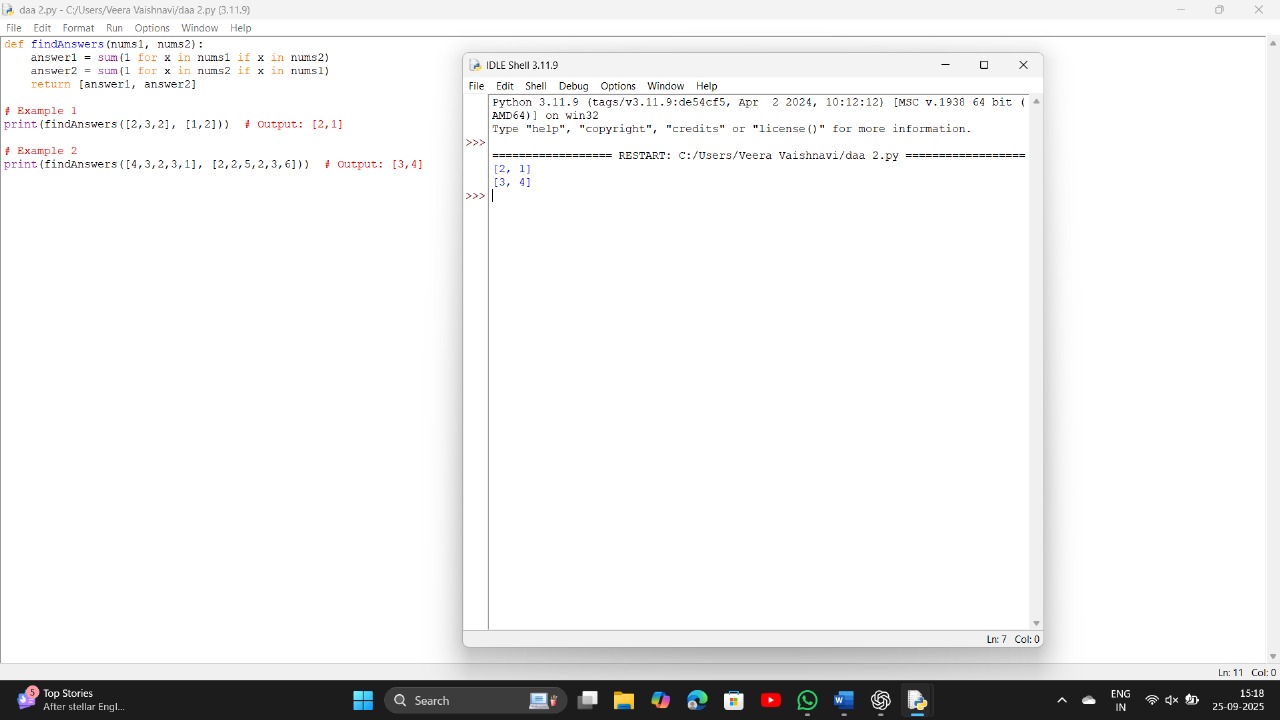
nums2 = [2, 2, 5, 2, 3, 6]

**Output:**

[3, 4]

**Explanation:**

* In nums1, the numbers 3, 2, 3 are found in nums2 → answer1 = 3
* In nums2, the numbers 2, 2, 2, 3 are found in nums1 → answer2 = 4



**3. Sum of Squares of Distinct Counts of Subarrays**

**Aim**

To calculate the sum of the squares of distinct element counts of all subarrays of a given array.

**Algorithm**

1. Initialize total = 0.
2. Loop through all possible starting indices i.
3. For each i, initialize an empty set distinct.
4. For each ending index j, add nums[j] to distinct.
5. Add (len(distinct))² to total.
6. Return total.

**Input**

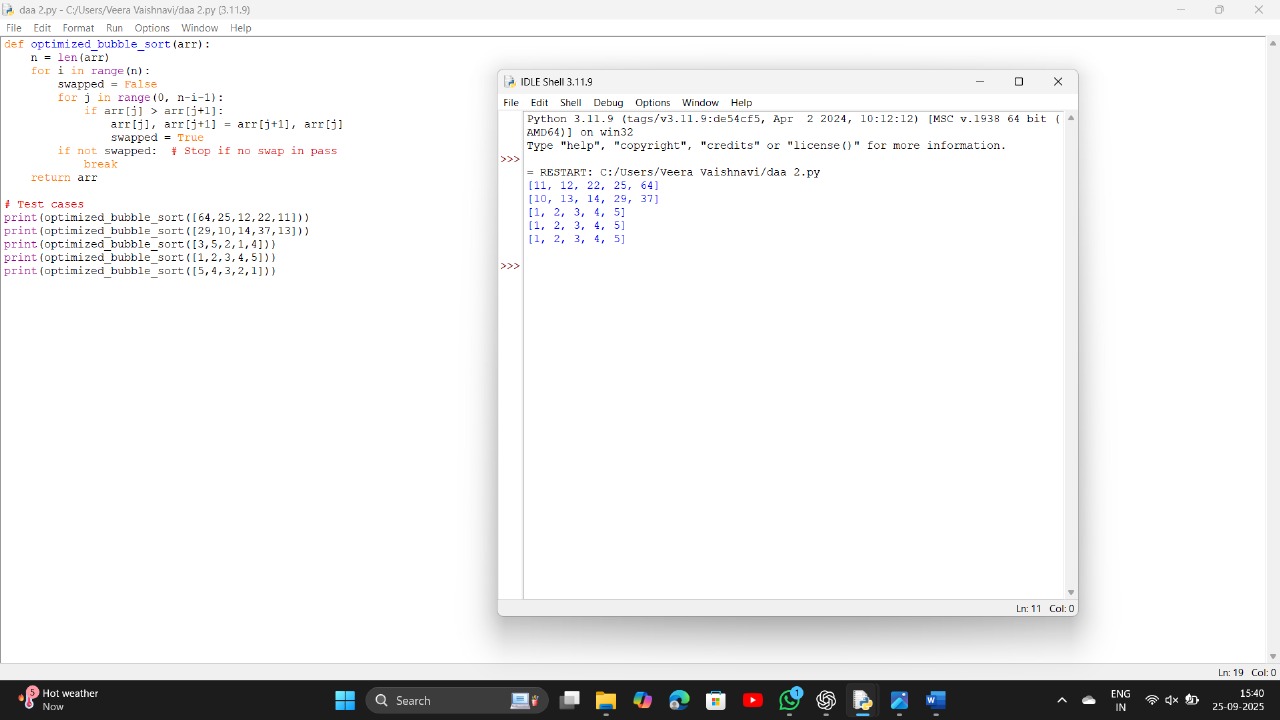
nums = [1,2,1]

**Output**

15

**Result**

All subarrays → [1], [2], [1], [1,2], [2,1], [1,2,1] →  
Distinct counts → 1,1,1,2,2,2 →  
Squares → 1,1,1,4,4,4 → Sum = **15**



**4. Pairs with Product Divisible by k**

**Aim**

Find pairs (i, j) where nums[i] == nums[j] and (i\*j) is divisible by k.

**Algorithm**

1. Initialize count = 0.
2. Loop i from 0 to n-1.
3. Loop j from i+1 to n-1.
4. If nums[i] == nums[j] and (i\*j) % k == 0, increment count.
5. Return count.

**Input**

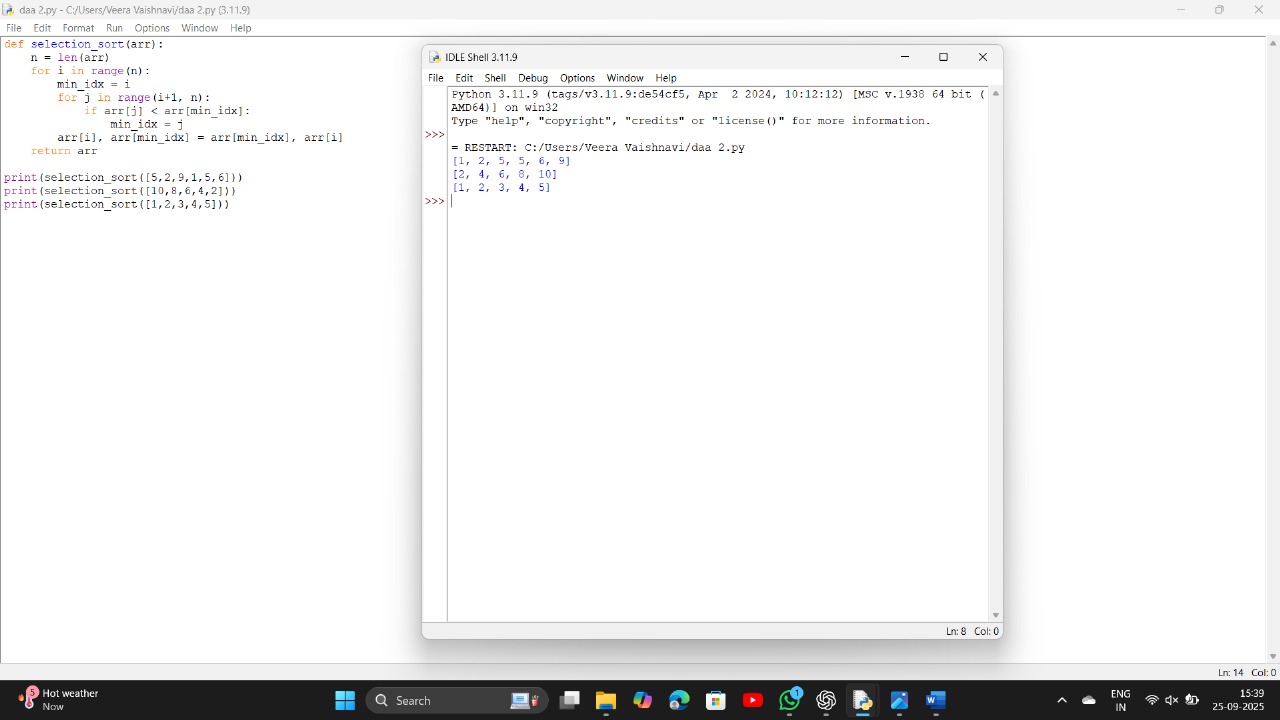
nums = [3,1,2,2,2,1,3], k = 2

**Output**

4

**Result**

Valid pairs: (0,6), (2,3), (2,4), (3,4) → Total = **4**



**5. Find Maximum Element**

**Aim**

Find the maximum element of the array with least time complexity.

**Algorithm**

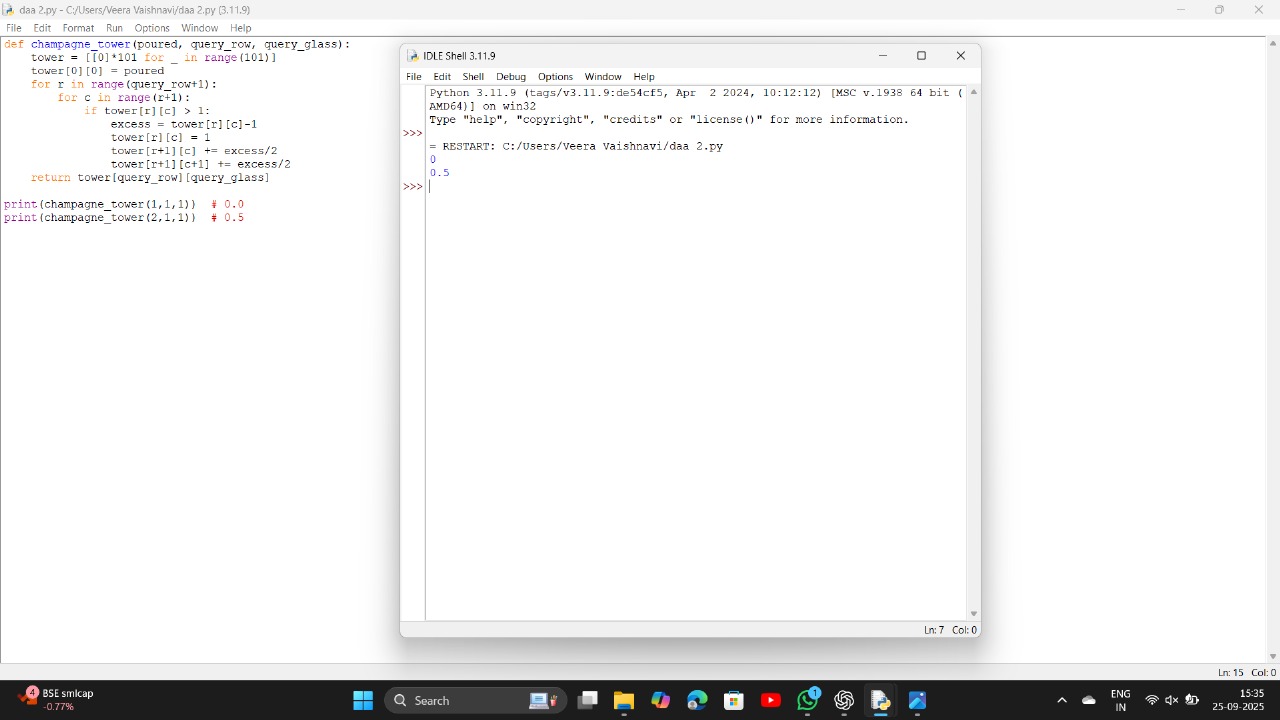
1. Initialize max\_val = arr[0].
2. Loop through array elements.
3. If element > max\_val, update it.
4. Return max\_val.

**Input & Output**

* [1,2,3,4,5] → 5
* [7,7,7,7,7] → 7
* [-10,2,3,-4,5] → 5

**Result**

Max element found in **O(n)** time.



**6. Sort and Find Maximum**

**Aim**

Sort the list using an efficient sorting algorithm and return the maximum element.

**Algorithm**

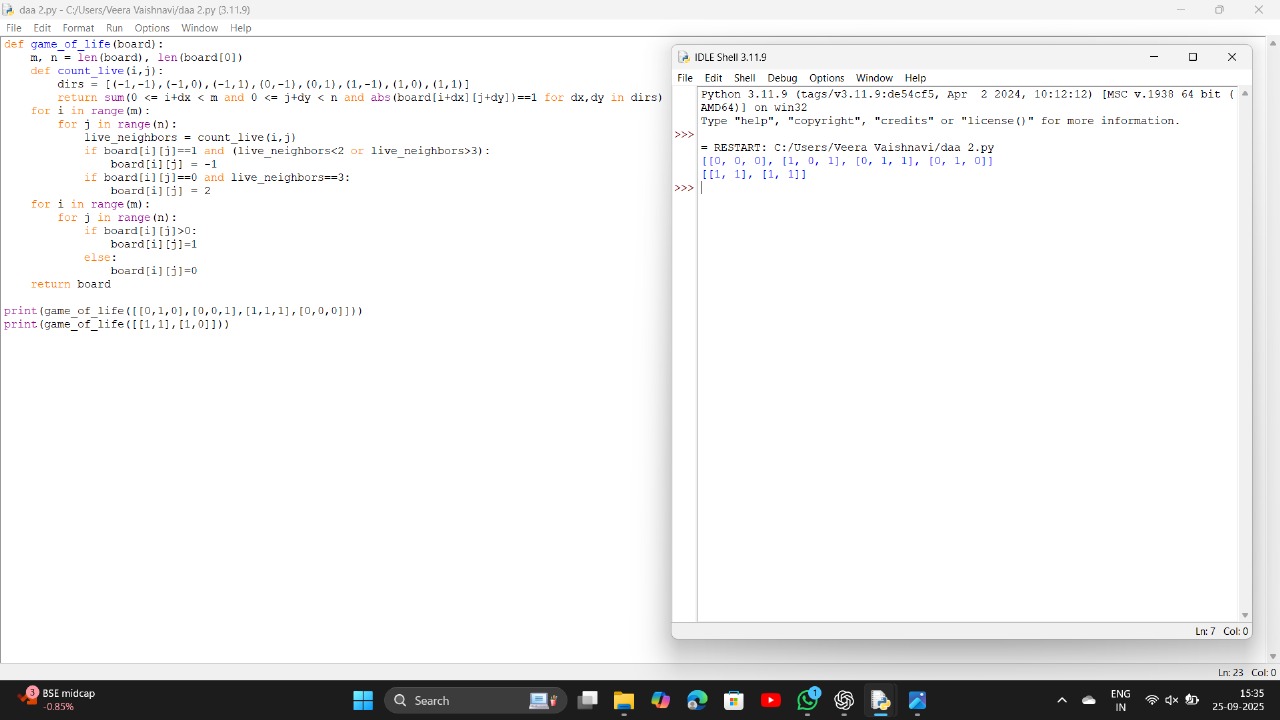
1. If list empty → return None.
2. Sort the list (Timsort in Python).
3. Return last element as maximum.

**Input & Output**

* [] → None
* [5] → 5
* [3,3,3,3,3] → 3

**Result**

Sorting ensures correctness; last element = max.



**7. Unique Elements in List**

**Aim**

Return a list of unique elements from input list.

**Algorithm**

1. Use a set to remove duplicates.
2. Convert set back to list.
3. Return list.

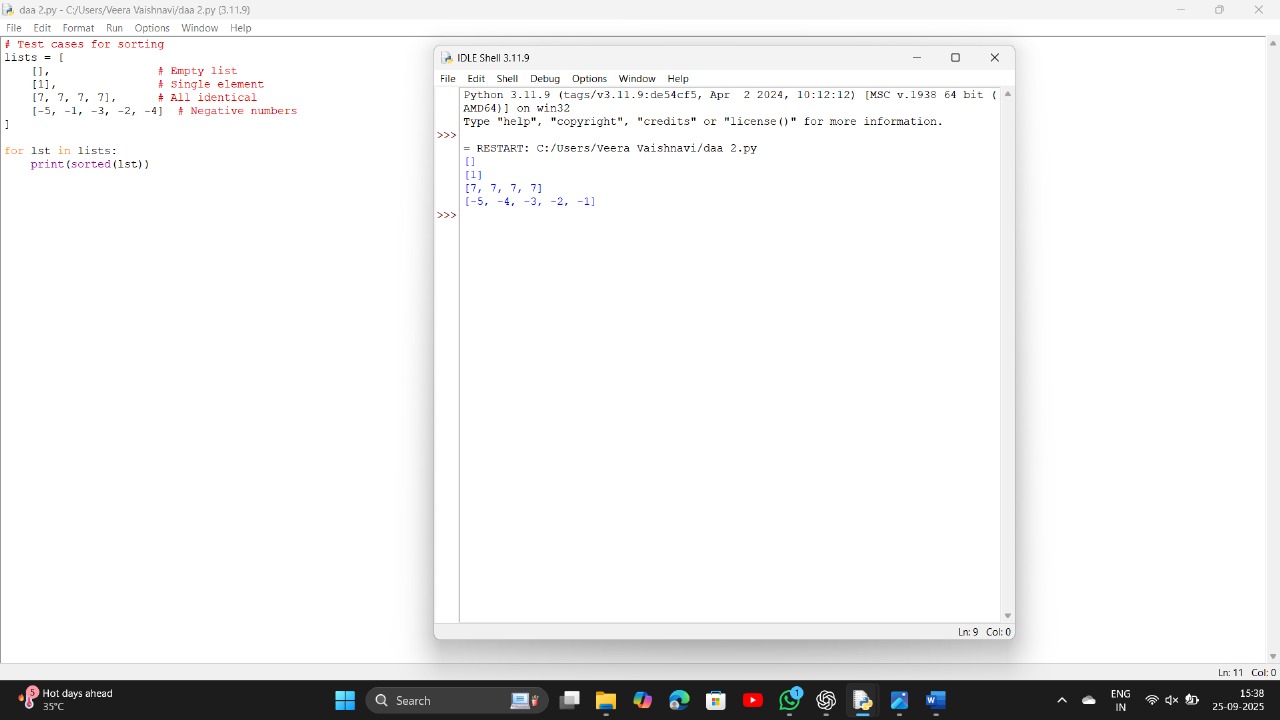
**Space Complexity:**  
O(n) (set stores unique elements).

**Input & Output**

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

**Result**

Duplicates removed; only unique values remain.



**8.Sort an array of integers using the Bubble Sort technique**

**Aim:**

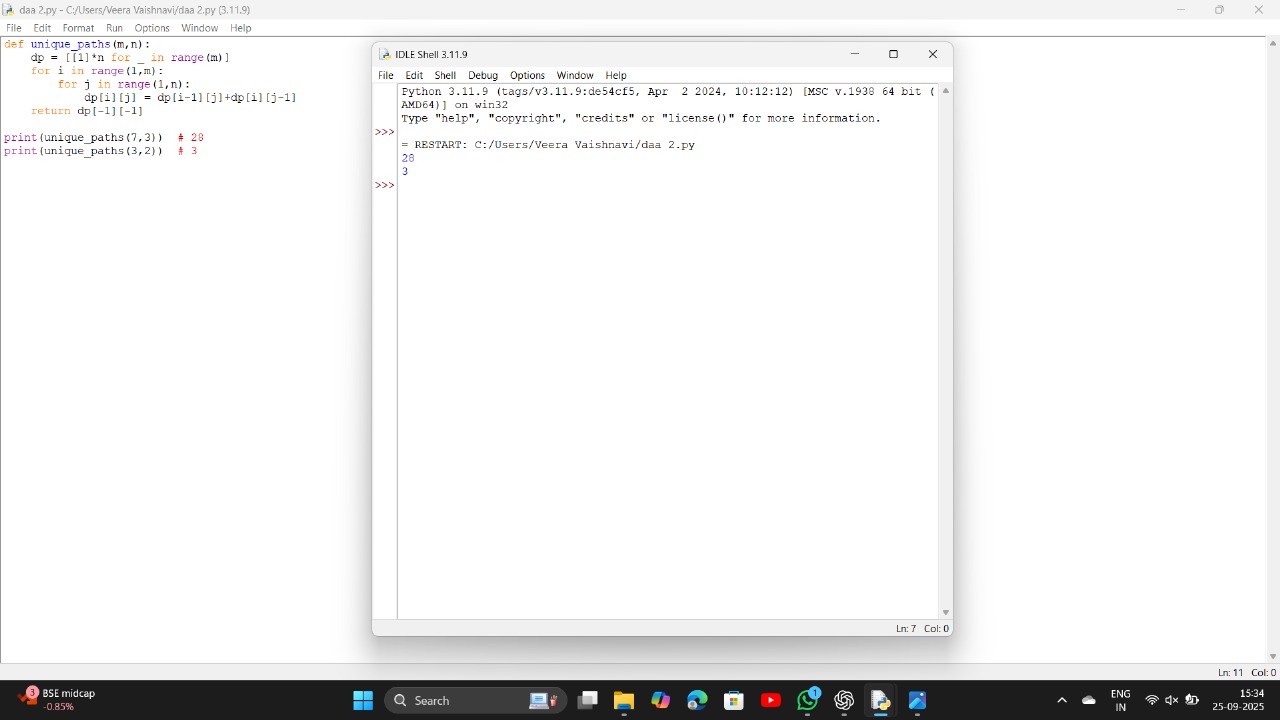
To sort an array of integers in ascending order using the bubble sort algorithm.

**Algorithm:**

1. Start with the first element of the array.
2. Compare the current element with the next element.
3. If the current element is greater than the next element, swap them.
4. Repeat the above process for all elements in the array.
5. After each full pass, the largest element will bubble up to the end.
6. Continue until no swaps are needed.

**Time Complexity:**

* Worst case: **O(n²)**
* Best case: **O(n)** (if already sorted)
* Average case: **O(n²)**



9.

**Binary Search to Check if Number Exists in a Sorted Array**

**Aim:**

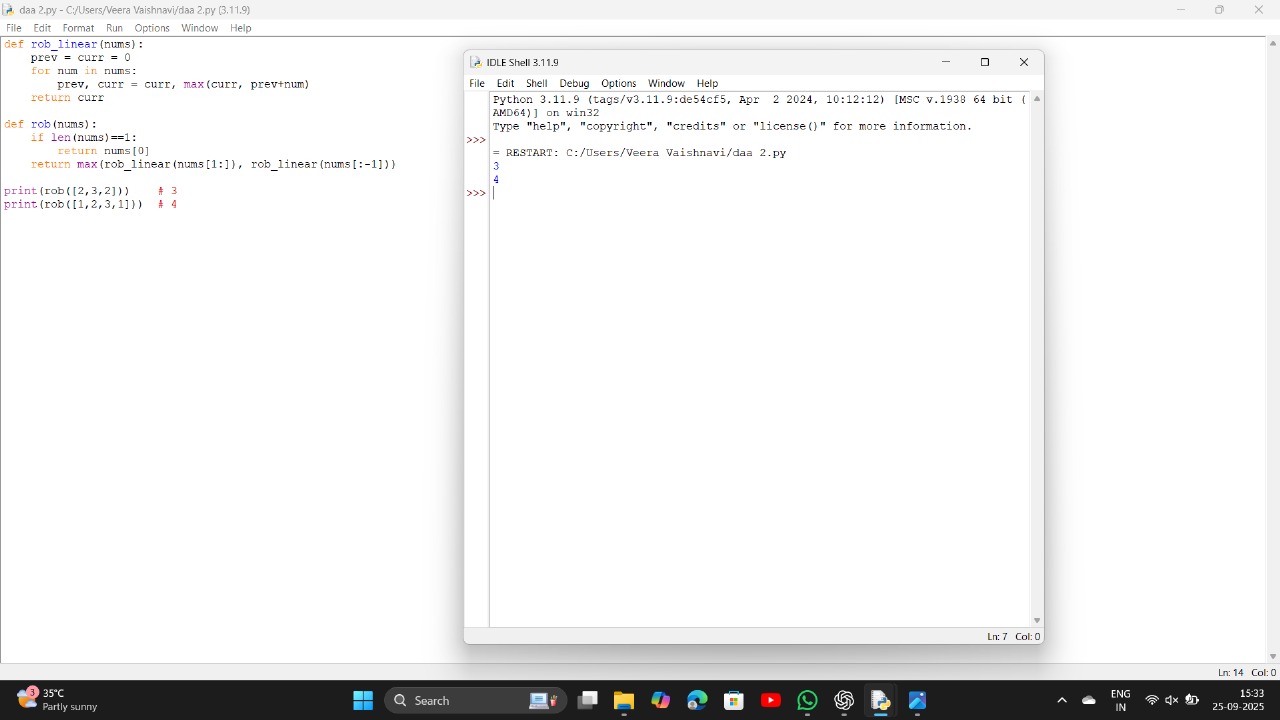
To determine if a given number exists in a sorted array using binary search.

**Algorithm:**

1. Set low = 0, high = n-1.
2. Find the middle index = (low + high) // 2.
3. If the middle element = key, return index.
4. If the key is smaller, search in the left half.
5. If the key is greater, search in the right half.
6. Repeat until low > high.

**Time Complexity:**

* Best case: **O(1)** (if found at first check)
* Worst case: **O(log n)**
* Average case: **O(log n)**



10.

**Sort Array in O(n log n) Time (Merge Sort)**

**Aim:**

To sort an array in ascending order using merge sort with **O(n log n)** time and minimal space.

**Algorithm (Merge Sort):**

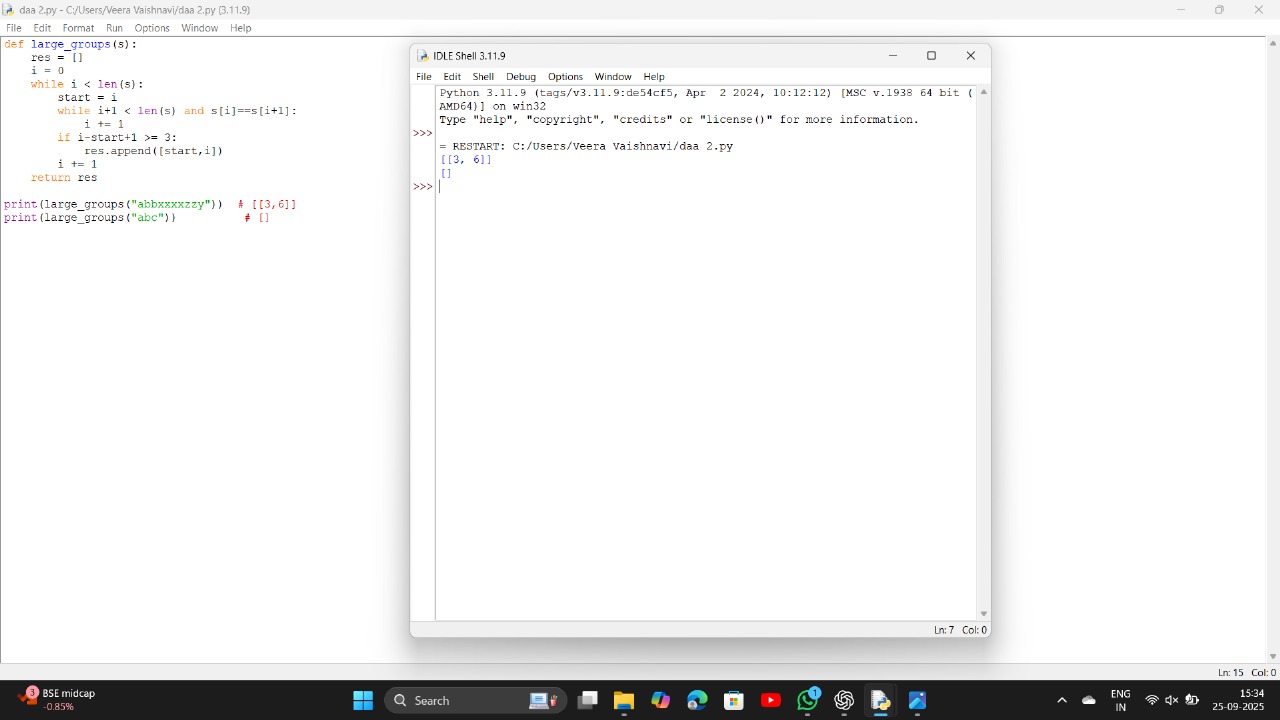
1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves into one sorted array.

**Time Complexity:**

* Worst case: **O(n log n)**
* Best case: **O(n log n)**
* Average case: **O(n log n)**

**Space Complexity:**

* **O(n)** (for temporary arrays during merging)



11.

**Ball Moving Out of Grid in N Steps**

**Aim**

Find the number of ways a ball can move out of a grid in exactly **N steps**.

**Algorithm**

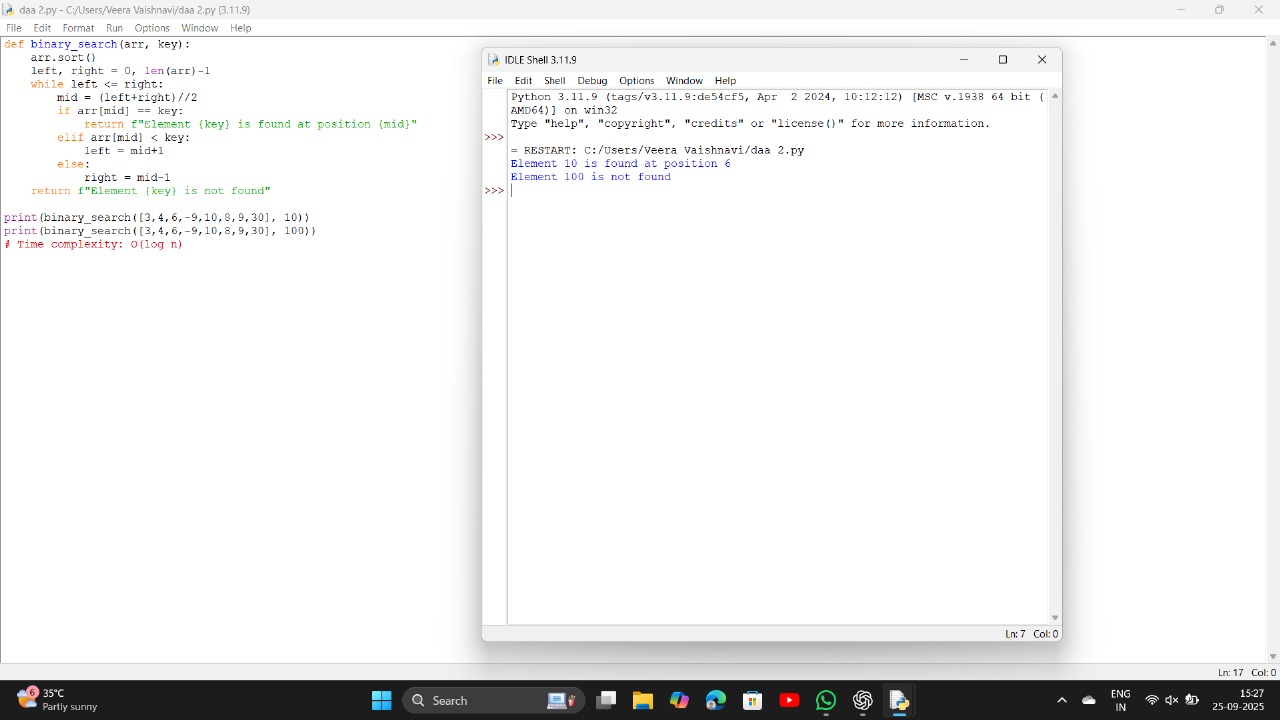
1. Use **DFS with memoization (Dynamic Programming)**.
2. At each step, move in four directions (up, down, left, right).
3. If the ball goes outside the grid, count it as 1.
4. If steps run out inside the grid, return 0.
5. Use memoization (i, j, steps) to avoid recomputation.

**Input**

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

**Output**

6



12.

**House Robber II (Circular Street)**

**Aim**

Find the maximum money that can be robbed without robbing two adjacent houses in a circle.

**Algorithm**

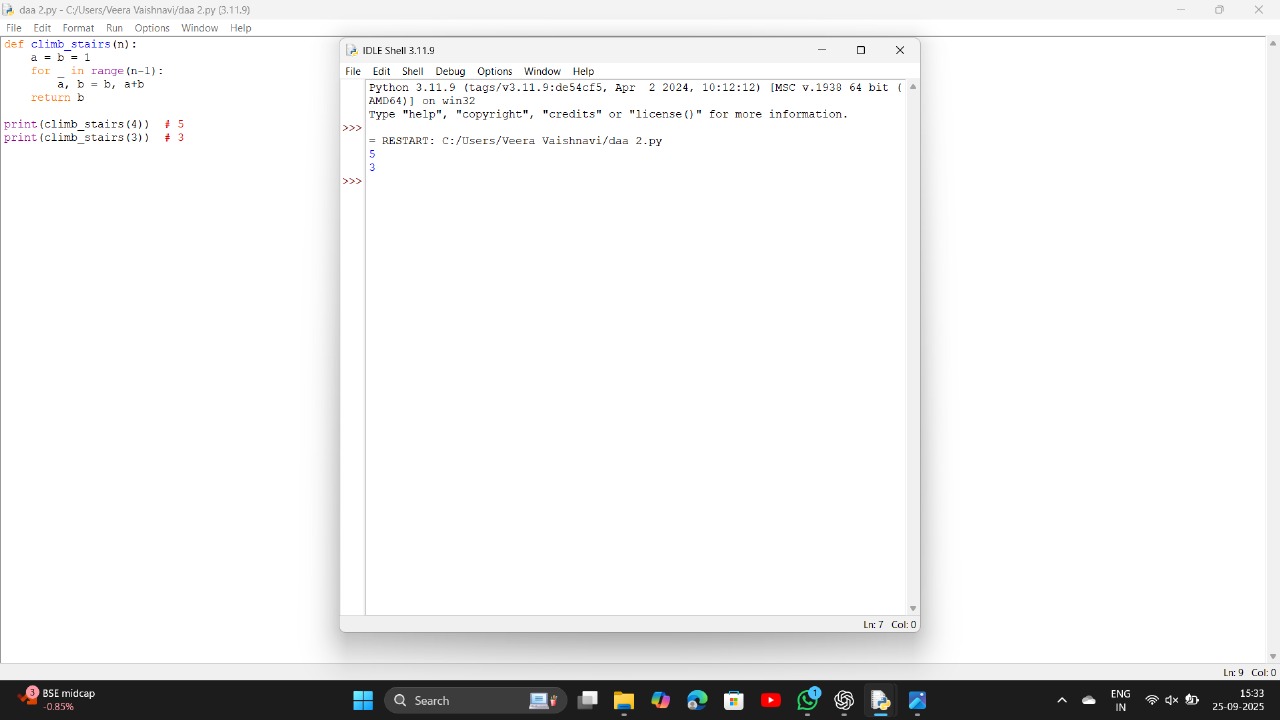
1. Normal House Robber (linear) uses DP.
2. For circular houses → two cases:
   * Exclude first house and rob from 1 to n-1.
   * Exclude last house and rob from 0 to n-2.
3. Take the maximum of both cases.

**Input**

nums = [2, 3, 2]

**Output**

3



13.

**Climbing Stairs**

**Aim**

Find distinct ways to climb n steps, moving either **1** or **2** steps at a time.

**Algorithm**

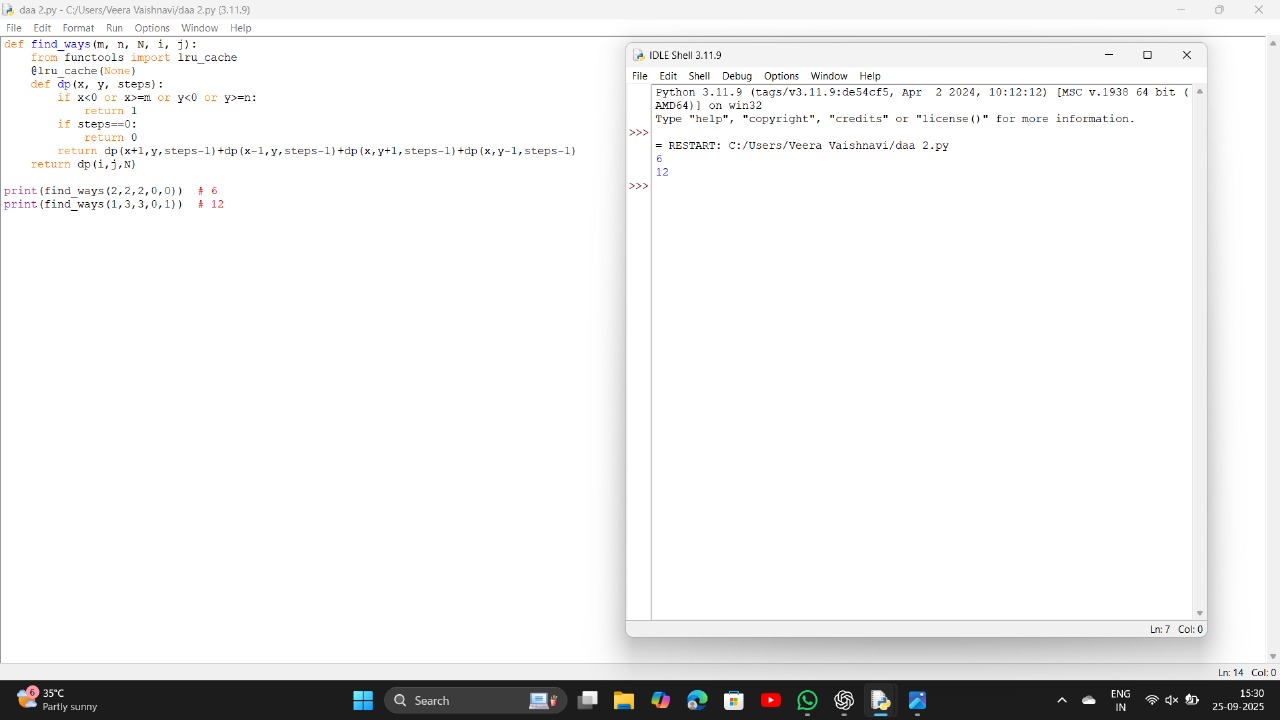
1. This is **Fibonacci sequence**:  
   ways(n) = ways(n-1) + ways(n-2)
2. Use DP to avoid recomputation.

**Input**

n = 4

**Output**

5



14.

**Robot Unique Paths**

**Aim**

Find the number of unique paths from **top-left** to **bottom-right** moving only **down** or **right**.

**Algorithm**

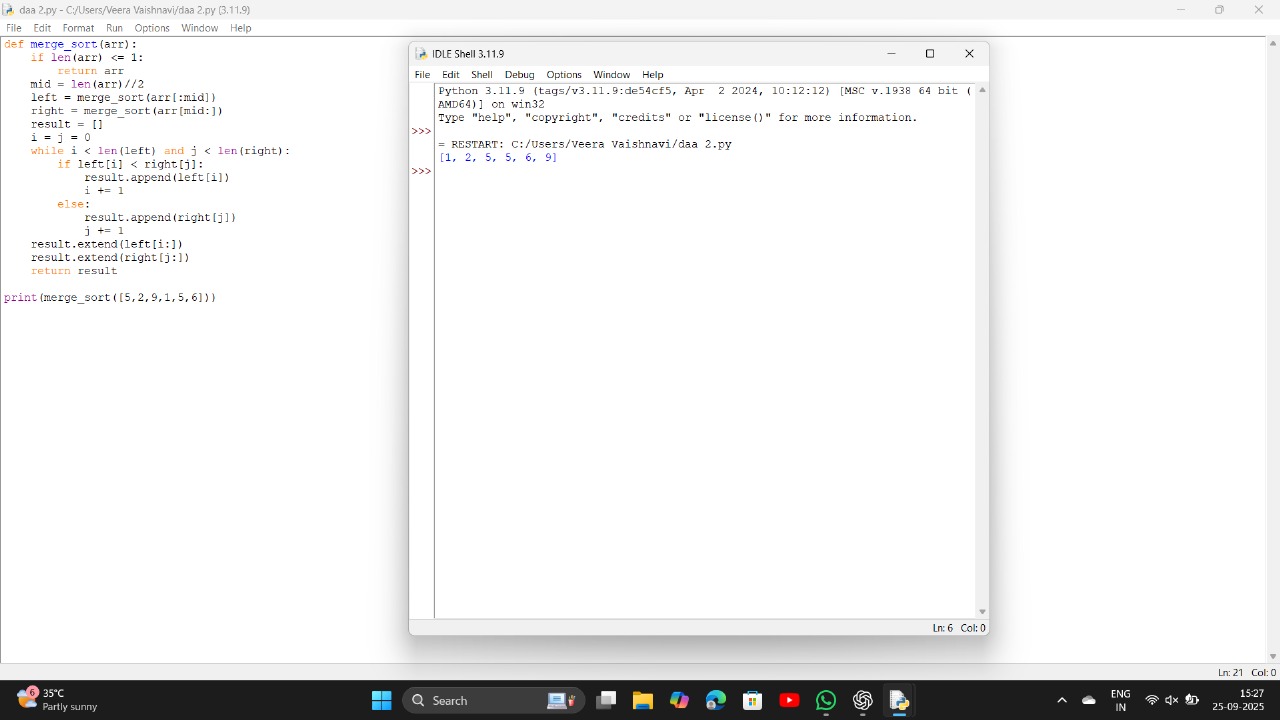
1. Use **DP table**.
2. Each cell = paths from top + paths from left.

**Input**

m=3, n=2

**Output**

3



15.

**Large Groups in String**

**Aim**

Find intervals of large groups (≥3 same consecutive characters).

**Algorithm**

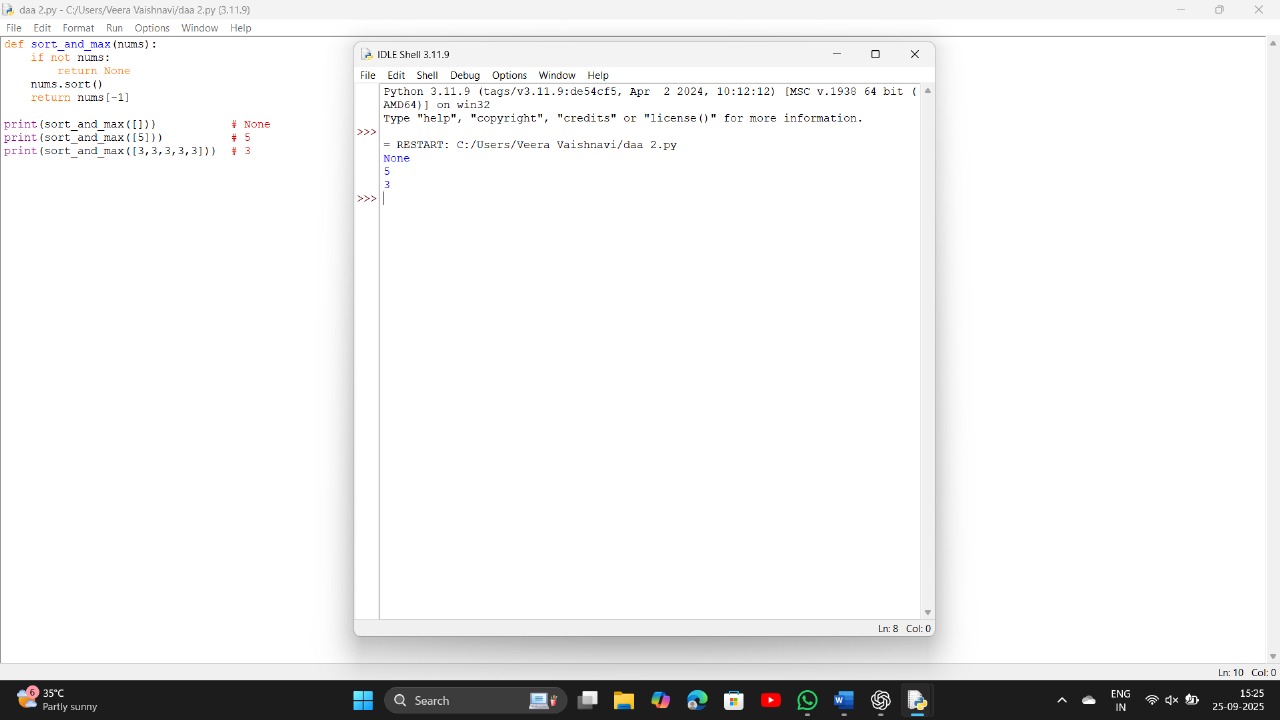
1. Iterate through string with two pointers.
2. If group length ≥ 3, store [start, end].

**Input**

s = "abbxxxxzzy"

**Output**

[[3, 6]]



16.

**Game of Life**

**Aim**

Compute the next state of Conway’s Game of Life grid.

**Algorithm**

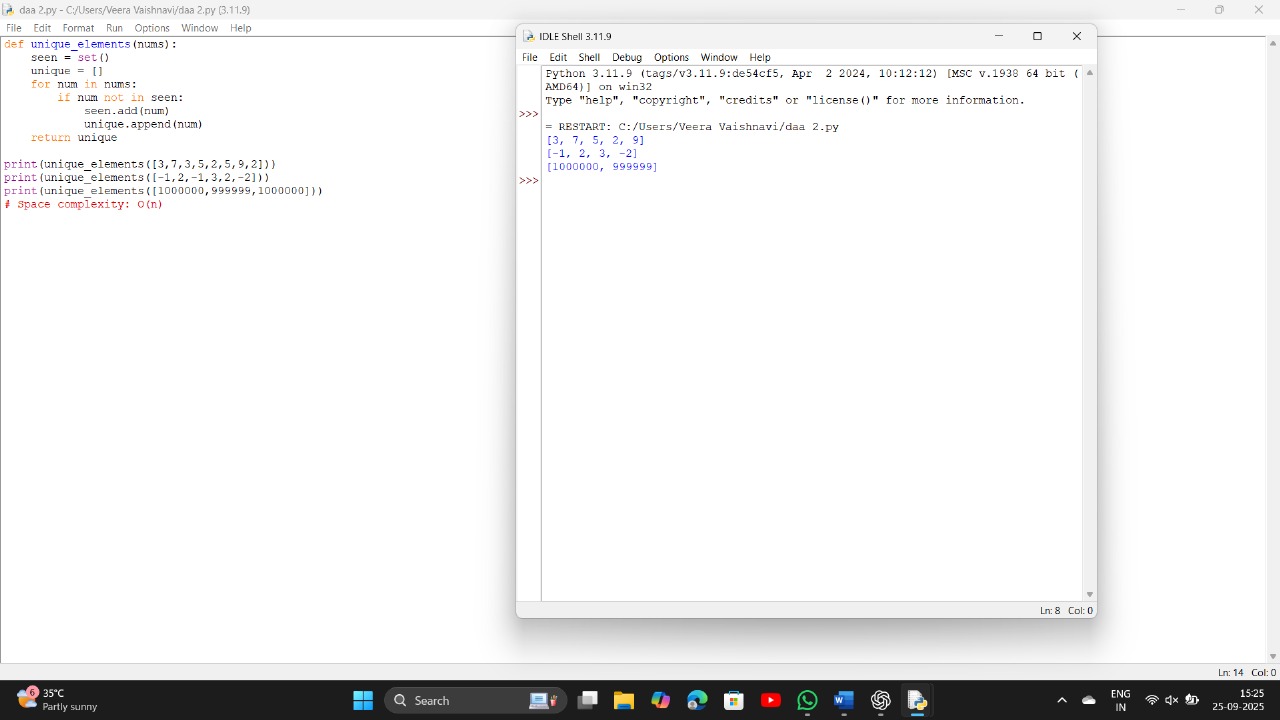
1. For each cell, count live neighbors.
2. Apply rules:
   * <2 → dies
   * 2 or 3 → lives
   * 3 → dies
   * dead + 3 neighbors → live

**Input**

[[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]]



17.

**Champagne Tower**

**Aim**

Find how full the query\_glass is after pouring poured cups.

**Algorithm**

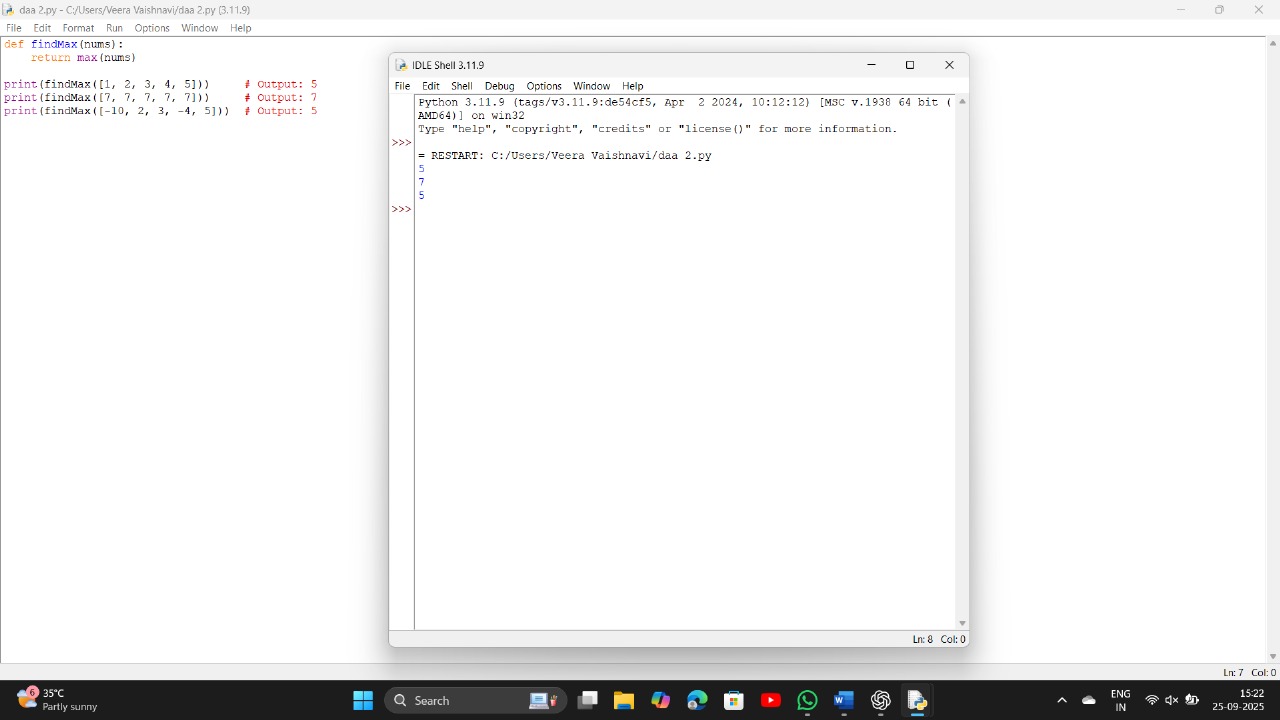
1. Use DP.
2. Each glass spills extra equally to next row (left and right).
3. Stop when reaching query\_row.

**Input**

poured=2, query\_row=1, query\_glass=1

**Output**

0.5



**18. Aim:**

To write a program to perform operations on different types of lists:

* An empty list
* A list with one element
* A list with all identical elements
* A list with negative numbers (sorted in ascending order)

**Algorithm**

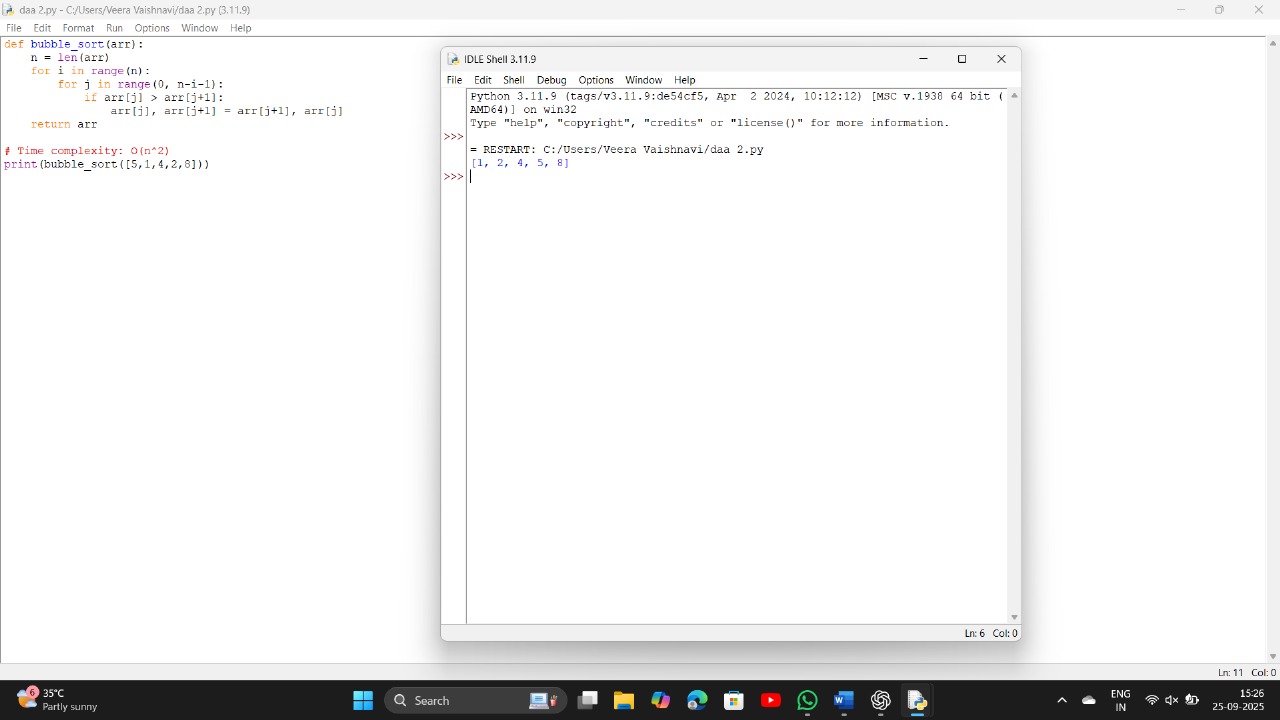
1. Start.
2. Create and display an empty list.
3. Create and display a list with one element.
4. Create and display a list with all identical elements.
5. Create a list with negative numbers.
6. Sort the list with negative numbers in ascending order and display the result.
7. Stop.

Input: [] ∙ Expected Output: []

Input: [1] ∙ Expected Output: [1]

Input: [7, 7, 7, 7] ∙ Expected Output: [7, 7, 7, 7]

Input: [-5, -1, -3, -2, -4] ∙ Expected Output: [-5, -4, -3, -2, -1]



**19. Aim**

To understand and implement the Selection Sort algorithm, which sorts an array by repeatedly selecting the smallest element from the unsorted region and placing it in the correct position.

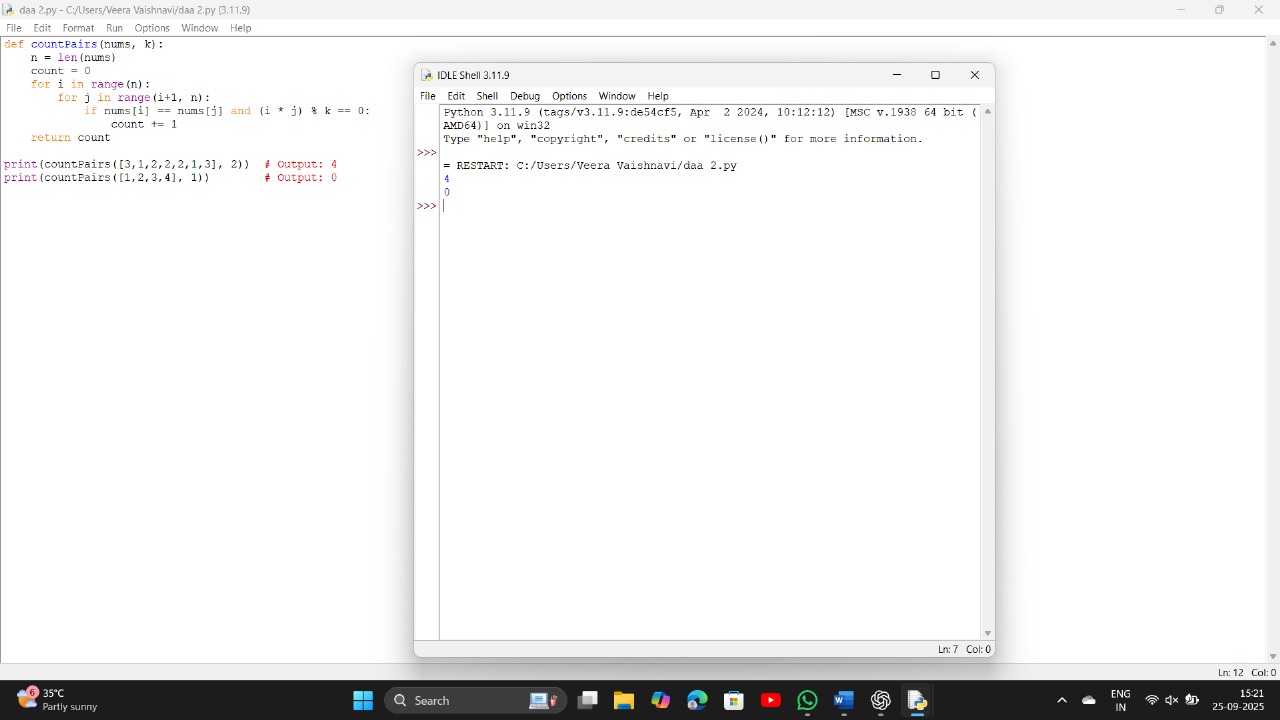
**Algorithm**

1. Start.
2. For i = 0 to n-2:
   * Assume min\_index = i.
   * For j = i+1 to n-1:
     + If arr[j] < arr[min\_index], update min\_index = j.
   * Swap arr[i] and arr[min\_index].
3. Repeat until the entire array is sorted.
4. Stop.

Sorting a Random Array: Input: [5, 2, 9, 1, 5, 6] Output: [1, 2, 5, 5, 6, 9]

Sorting a Reverse Sorted Array: Input: [10, 8, 6, 4, 2] Output: [2, 4, 6, 8, 10]

Sorting an Already Sorted Array: Input: [1, 2, 3, 4, 5] Output: [1, 2, 3, 4, 5]



20. **Aim:**

To write a program to optimize the Bubble Sort algorithm so that it stops early if the list becomes sorted before all passes are completed.

**Algorithm**

1. Start.
2. Read the list of elements.
3. For each pass from 0 to n-1:
   * Set swapped = False.
   * Compare each pair of adjacent elements.
   * If the current element is greater than the next, swap them and set swapped = True.
   * If no elements were swapped during the pass (swapped == False), stop early.
4. Display the sorted list.
5. Stop.

