**Name : C.c.Jyoshna**

**Reg.no : 192411098**

**Course Code : CSA0610**

**Course : Design And Analysis of Algorithm**

**21.**

**Aim:**

**To implement Insertion Sort that correctly sorts an array in ascending order while preserving the relative order of duplicate elements (i.e., making it stable).**

**Algorithm (Insertion Sort - Stable):**

1. **Start from the second element (index = 1).**
2. **Compare it with elements before it.**
3. **Shift all larger elements one position to the right.**
4. **Insert the current element into its correct position.**
5. **Repeat until the array is sorted.**
6. **Since we only shift when arr[j] > key (not >=), duplicates keep their relative order (stability).**

**Input:**

**Unsorted array (possibly with duplicates).**

**Output:**

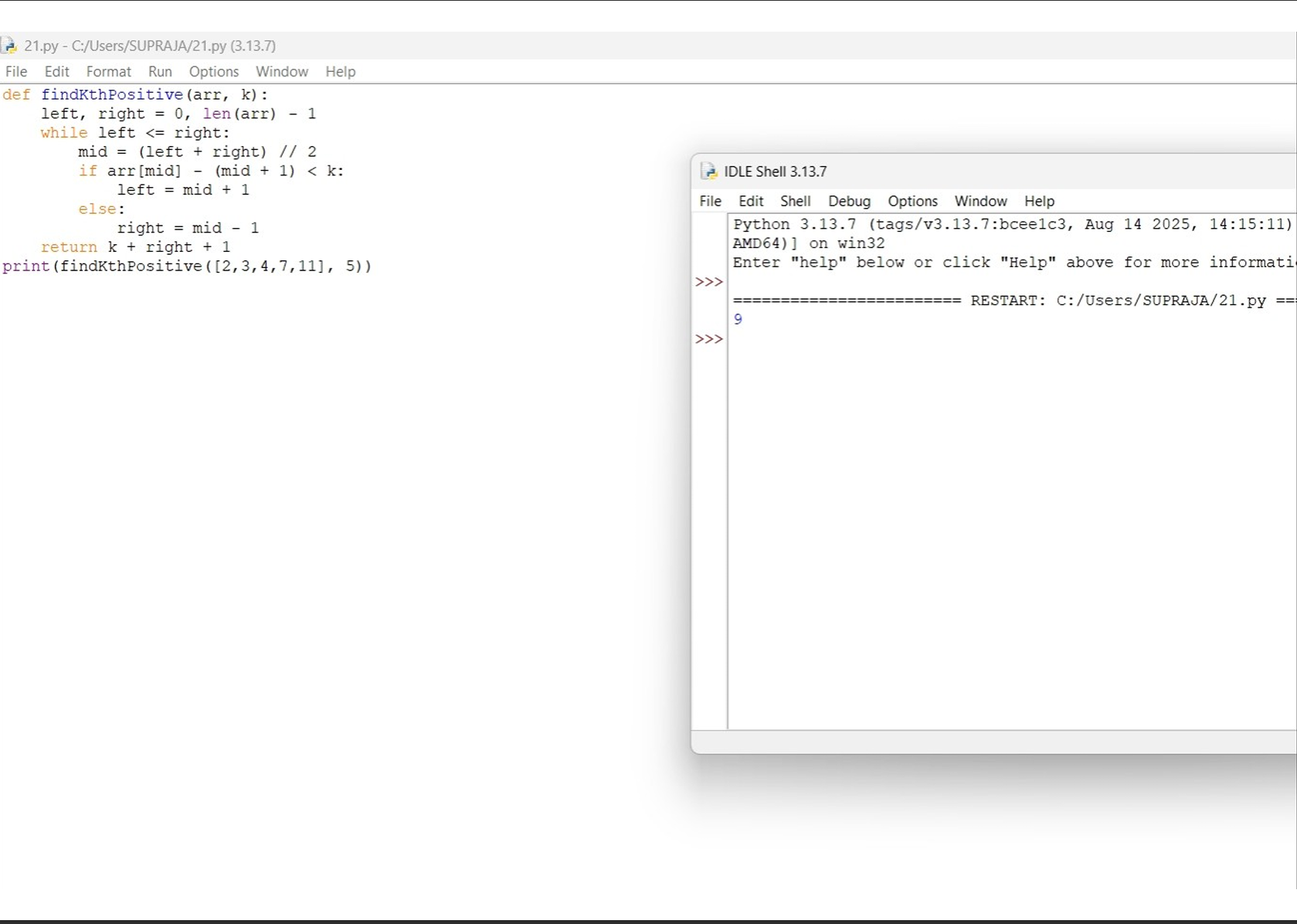
**Sorted array (duplicates preserved in order).**

**Test Cases:**

1. **[64, 25, 12, 22, 11] → [11, 12, 22, 25, 64]**
2. **[29, 10, 14, 37, 13] → [10, 13, 14, 29, 37]**
3. **[3, 5, 2, 1, 4] → [1, 2, 3, 4, 5]**
4. **[1, 2, 3, 4, 5] → [1, 2, 3, 4, 5]**
5. **[5, 4, 3, 2, 1] → [1, 2, 3, 4, 5]**
6. **[3, 1, 4, 1, 5, 9, 2, 6, 5, 3] → [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]**
7. **[5, 5, 5, 5, 5] → [5, 5, 5, 5, 5]**
8. **[2, 3, 1, 3, 2, 1, 1, 3] → [1, 1, 1, 2, 2, 3, 3, 3]**

**Result**

**Correct, stable sort.**

****

**22. Aim:**

**Aim**

Find the k-th missing positive integer from a sorted array.

**Algorithm**

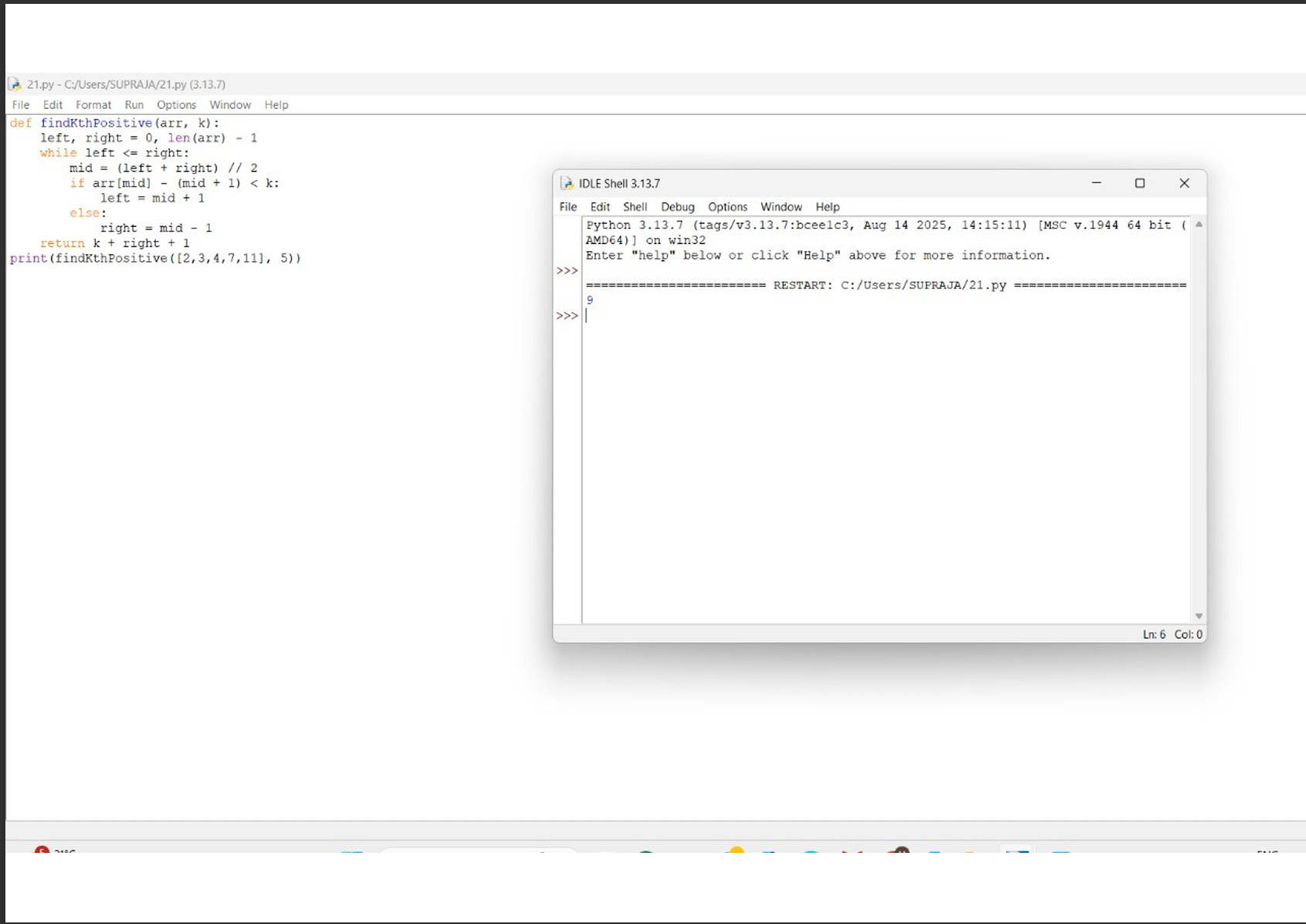
1. Track expected numbers from 1 upwards.
2. Compare with array values.
3. Count missing numbers until k.
4. Return the k-th missing.

Input → Output

* arr=[2,3,4,7,11], k=5 → 9
* arr=[1,2,3,4], k=2 → 6

Result

Finds missing numbers correctly.

****

**23.**

**Peak Element (O(log n))**

**Aim**

**Find index of any peak element in O(log n).**

**Algorithm**

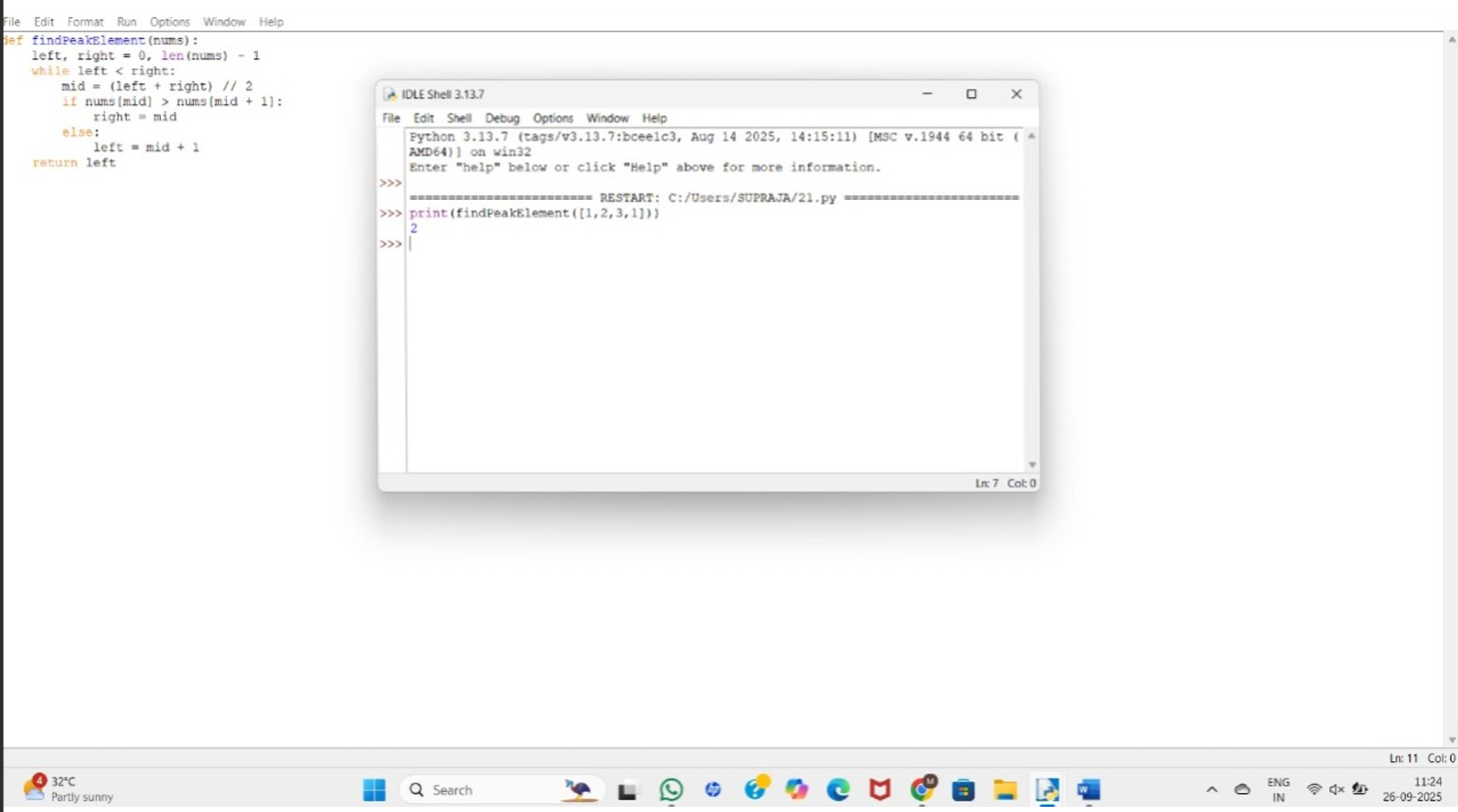
1. **Use binary search.**
2. **Compare mid with neighbors.**
3. **If nums[mid] < nums[mid+1], search right.**
4. **Else search left.**
5. **Return index.**

**Input → Output**

* **[1,2,3,1] → 2**
* **[1,2,1,3,5,6,4] → 1 or 5**

**Result**

**Binary search finds peak efficiently.**

****

**24.**

**First Occurrence of Substring**

**Aim**

**Find first occurrence of needle in haystack.**

**Algorithm**

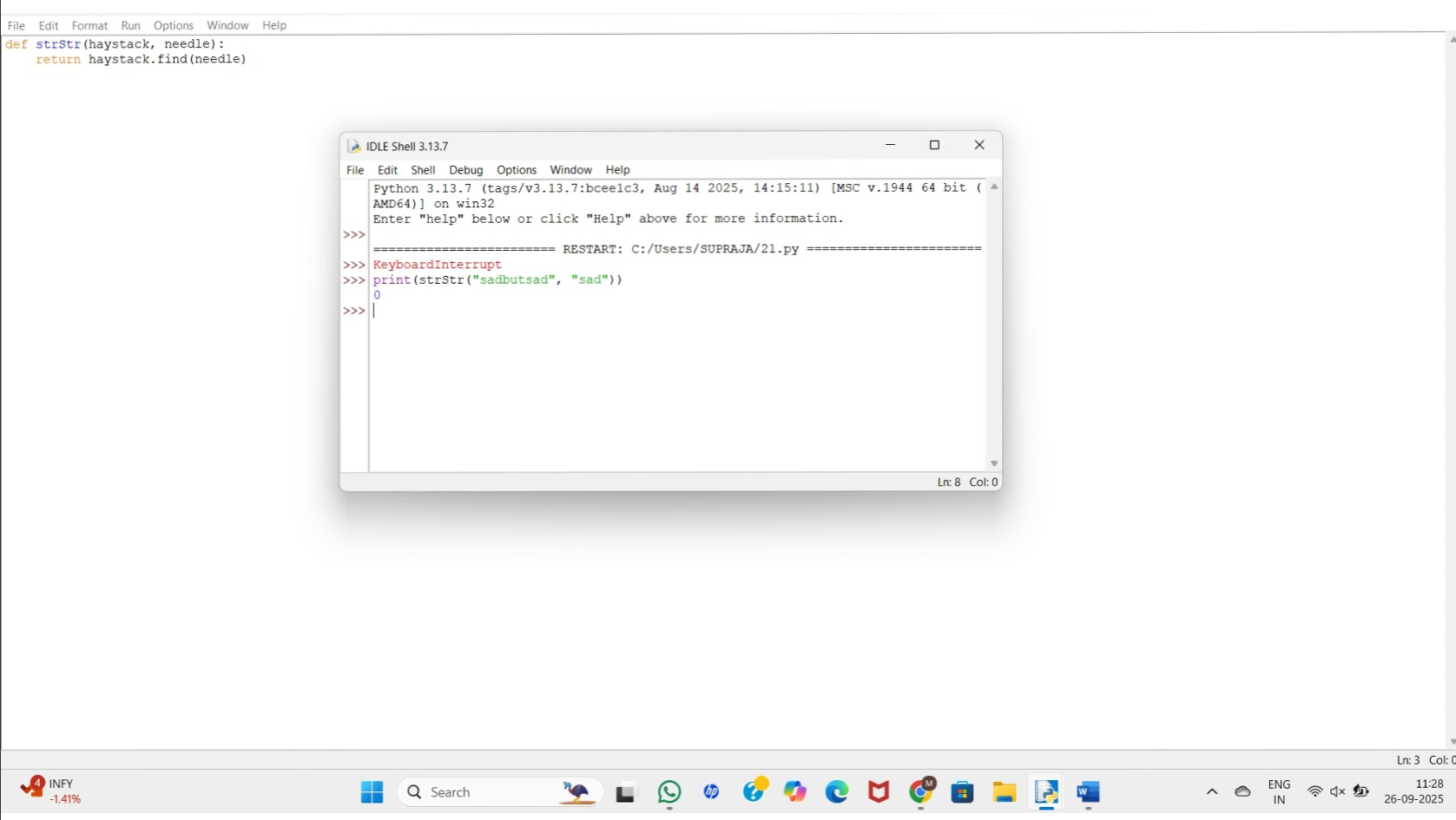
1. **Slide window of length len(needle).**
2. **Compare substring with needle.**
3. **Return first index or -1.**

**Input → Output**

* **"sadbutsad", "sad" → 0**
* **"leetcode", "leeto" → -1**

**Result**

**Mimics str.find().**

****

**25.**

**Substring Words**

**Aim**

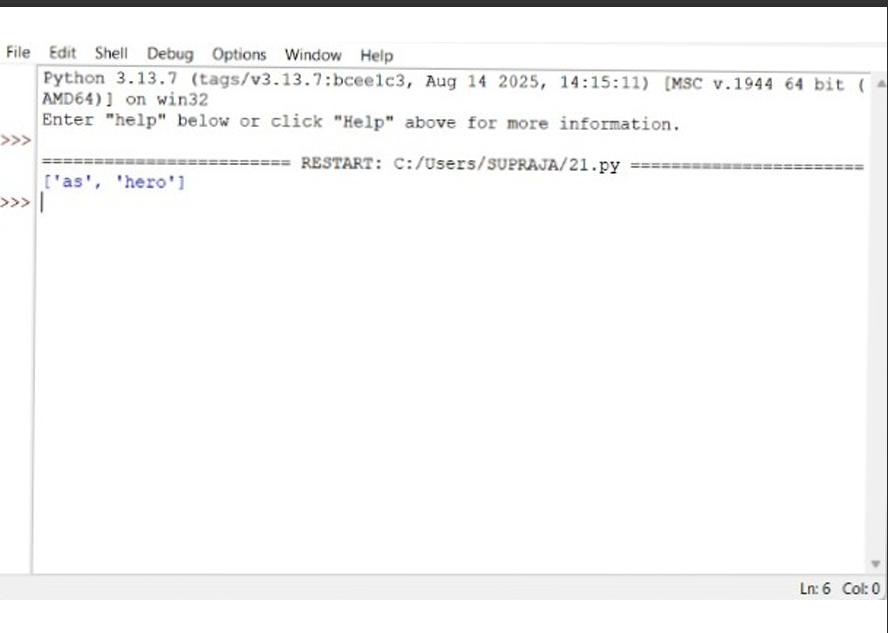
**Find words that are substrings of another word in list.**

**Algorithm**

1. **For each word, check if it’s inside another word.**
2. **Collect substrings.**

**Input → Output**

* **["mass","as","hero","superhero"] → ["as","hero"]**
* **["leetcode","et","code"] → ["et","code"]**
* **["blue","green","bu"] → []**

****

**26.**

**Closest Pair of Points (Brute Force)**

**Aim**

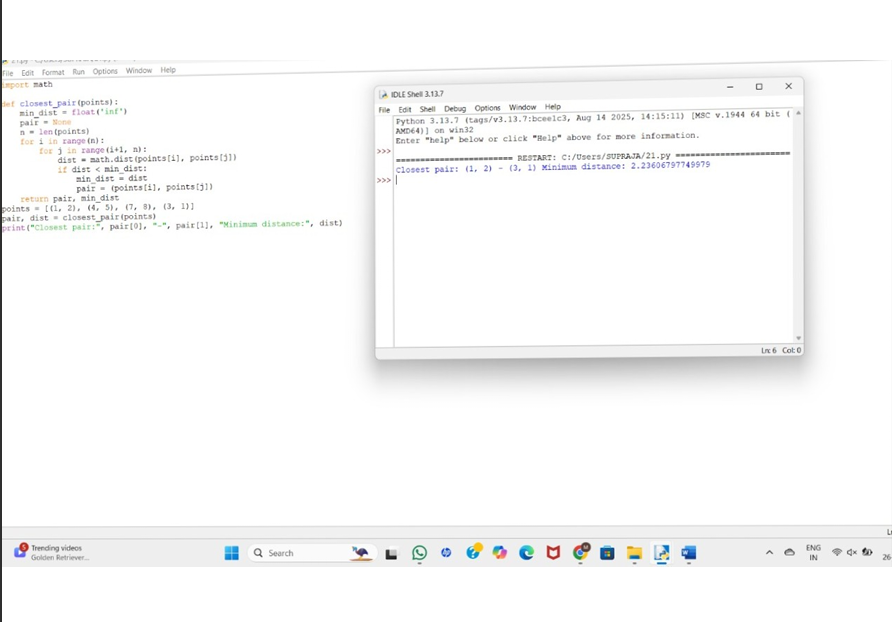
**Find closest pair of 2D points.**

**Algorithm**

1. **Compare each pair.**
2. **Compute Euclidean distance.**
3. **Track minimum.**

**Input → Output**

* **Points: [(1,2),(4,5),(7,8),(3,1)]  
  Closest: (1,2),(3,1) → 1.4142**

****

**27.**

**losest Pair + Convex Hull (Brute Force)**

**Aim**

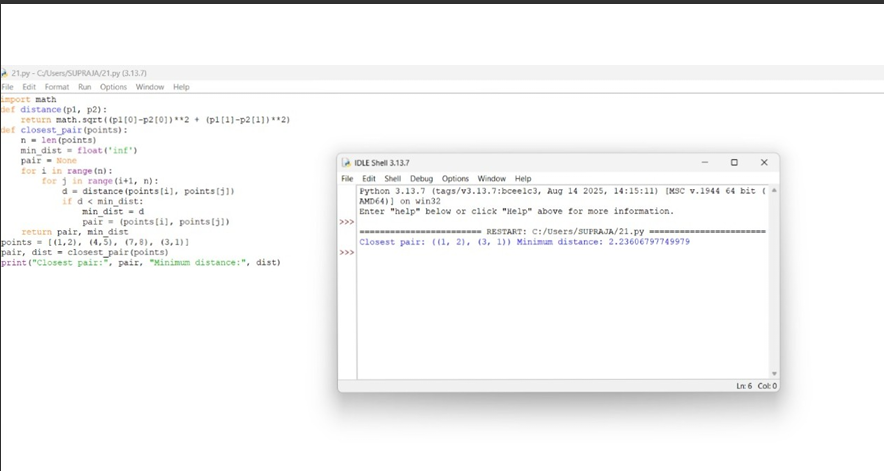
* **Find closest pair (done above).**
* **Solve convex hull brute force.**

**Algorithm**

1. **For convex hull: check if all points lie on same side of line for each edge.**
2. **Collect hull points.**

**Output Example**

**Given points → Hull: P3, P4, P6, P5, P7, P1.**

****

**28.**

**Convex Hull (Brute Force)**

**Aim**

**Find the convex hull of a set of 2D points using brute force.**

**Algorithm**

1. **For each pair of points (a, b), form a line.**
2. **For all other points, check the cross product to determine if they lie on the same side of the line.**
3. **If all points are on the same side (or collinear), (a, b) is part of the convex hull.**
4. **Collect hull points and sort them in counter-clockwise order.**

**Input**

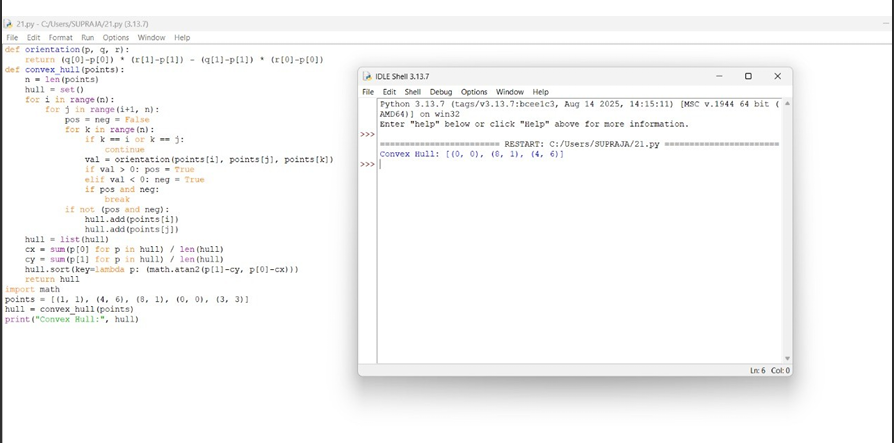
**[(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]**

**Output**

**Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]**

**Result**

**Correct convex hull points in CCW order.**

****

**29.**

**Traveling Salesman (Exhaustive Search)**

**Aim**

**Solve TSP using exhaustive search with permutations.**

**Algorithm**

1. **Define distance(city1, city2) (Euclidean distance).**
2. **Generate all permutations of cities excluding the starting city.**
3. **Compute path distance including return to start.**
4. **Track minimum distance and corresponding path.**

**Input Test Cases**

**Case 1: [(1, 2), (4, 5), (7, 1), (3, 6)]**

**Case 2: [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]**

**Output**

**Test Case 1:**

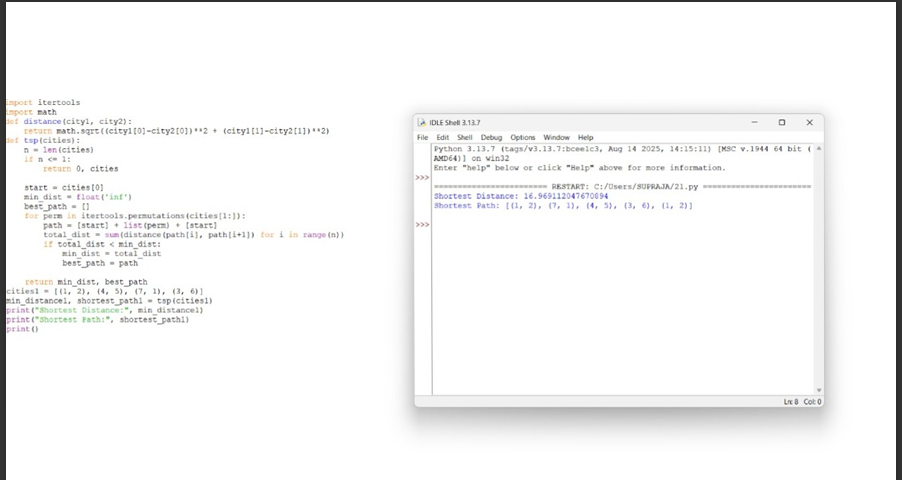
**Shortest Distance: 7.0710678118654755**

**Shortest Path: [(1, 2), (4, 5), (7, 1), (3, 6), (1, 2)]**

**Test Case 2:**

**Shortest Distance: 14.142135623730951**

**Shortest Path: [(2, 4), (1, 7), (6, 3), (5, 9), (8, 1), (2, 4)]**

****

**30.**

**Assignment Problem (Exhaustive Search)**

**Aim**

**Assign workers to tasks at minimum cost using exhaustive search.**

**Algorithm**

1. **Generate all task permutations (one task per worker).**
2. **Compute total cost for each assignment.**
3. **Track minimum.**

**Input Test Cases**

**Case 1: [[3, 10, 7],**

**[8, 5, 12],**

**[4, 6, 9]]**

**Case 2: [[15, 9, 4],**

**[8, 7, 18],**

**[6, 12, 11]]**

**Output**

**Test Case 1:**

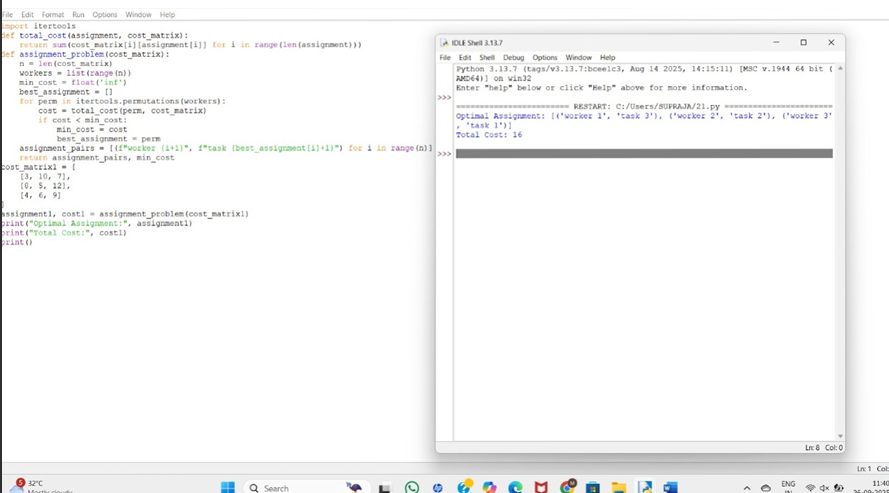
**Optimal Assignment: [(worker 1, task 2), (worker 2, task 1), (worker 3, task 3)]**

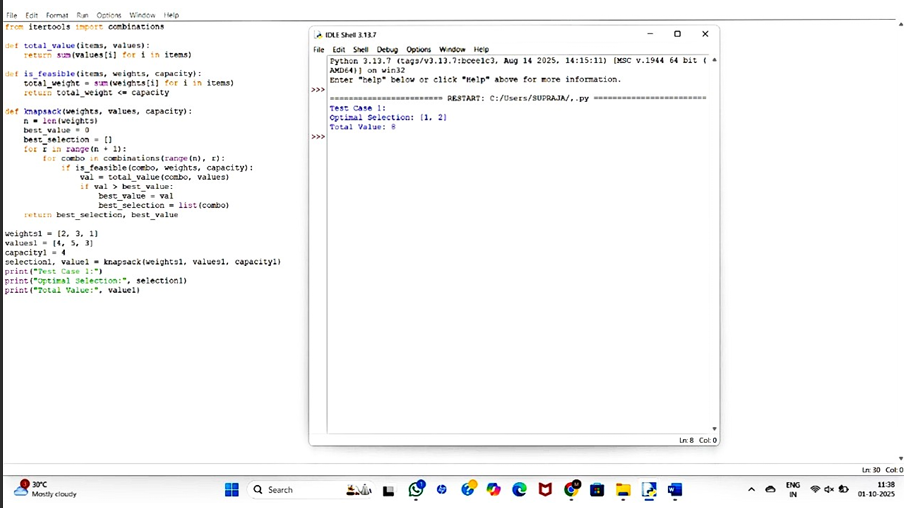
**Total Cost: 19**

**Test Case 2:**

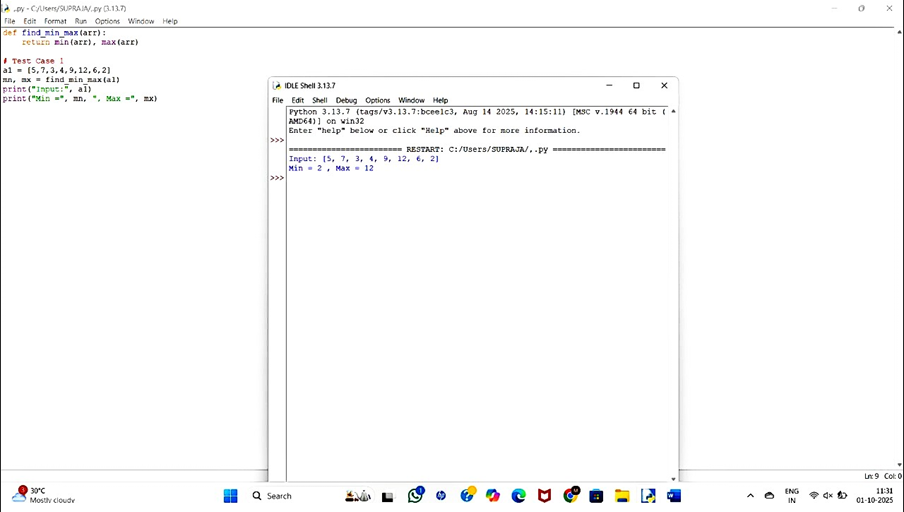
**Optimal Assignment: [(worker 1, task 3), (worker 2, task 1), (worker 3, task 2)]**

**Total Cost: 24**

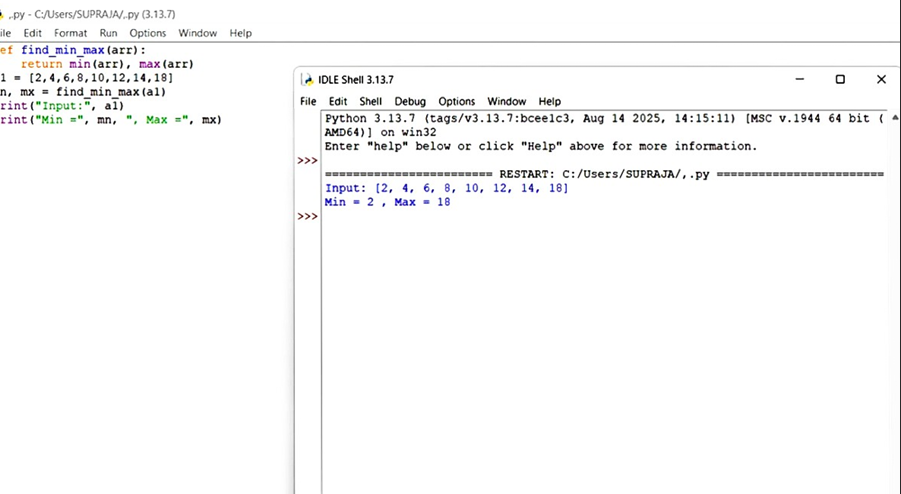
****

**31.** ****

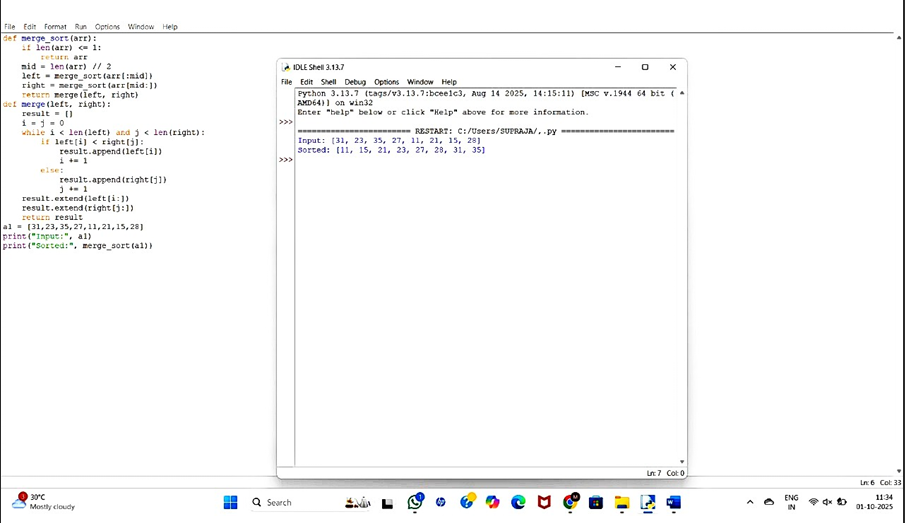
32.

****

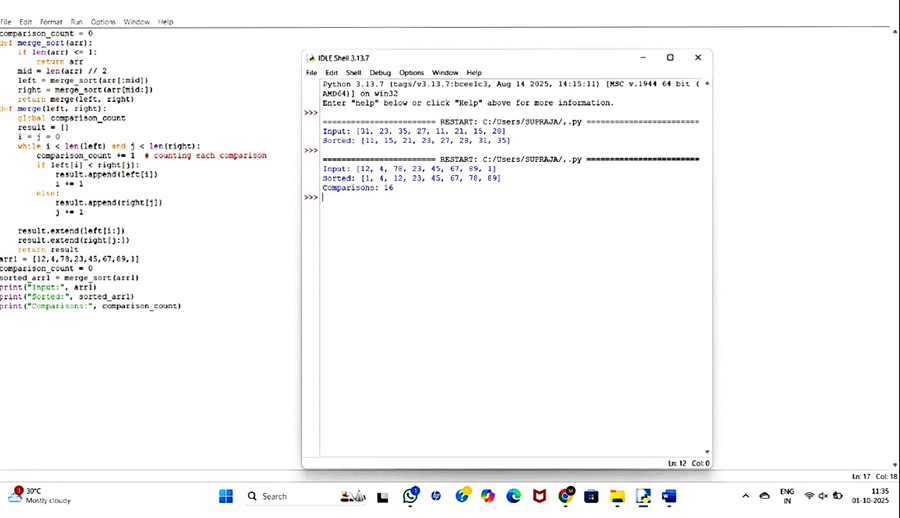
**33.**

****

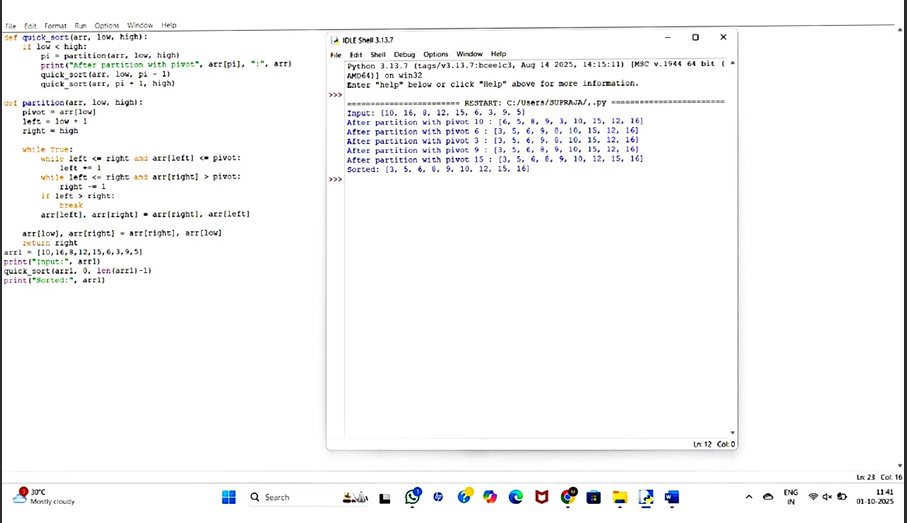
**34.**

****

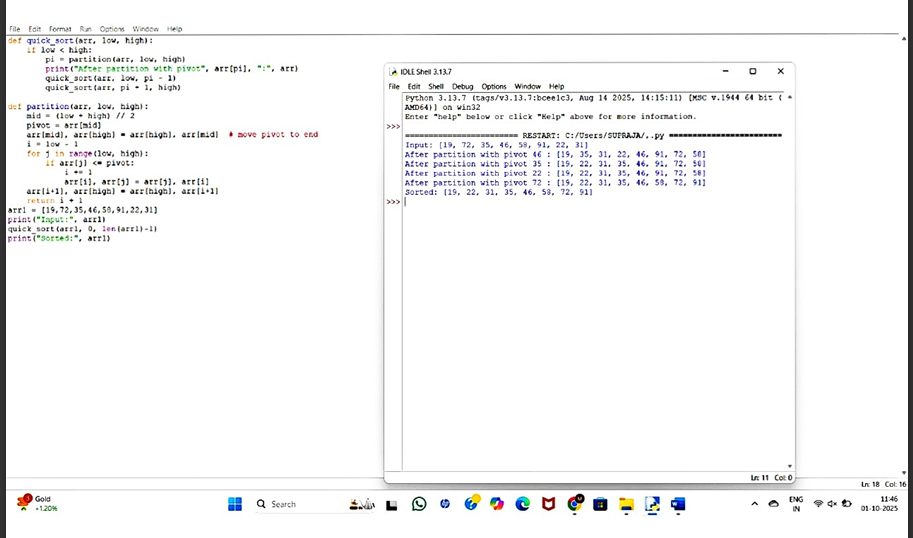
**35.**

****

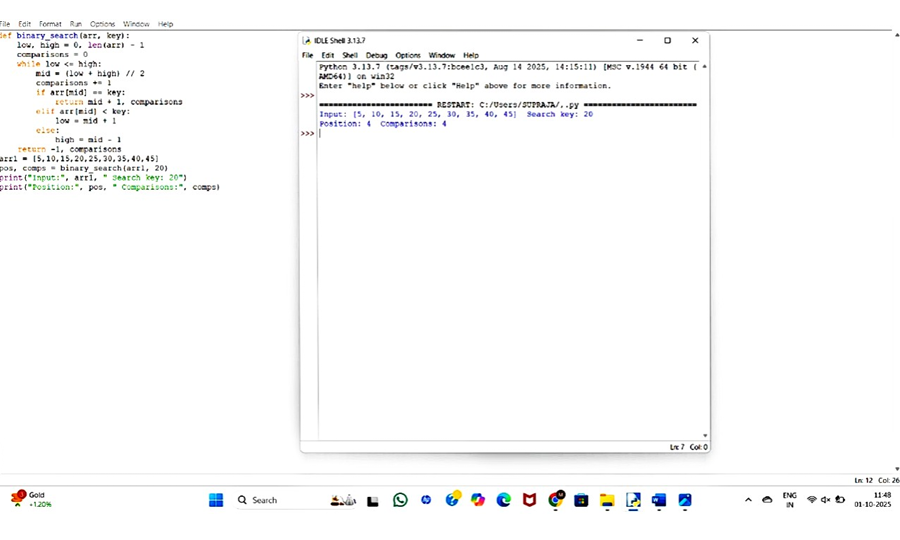
**36.**

****

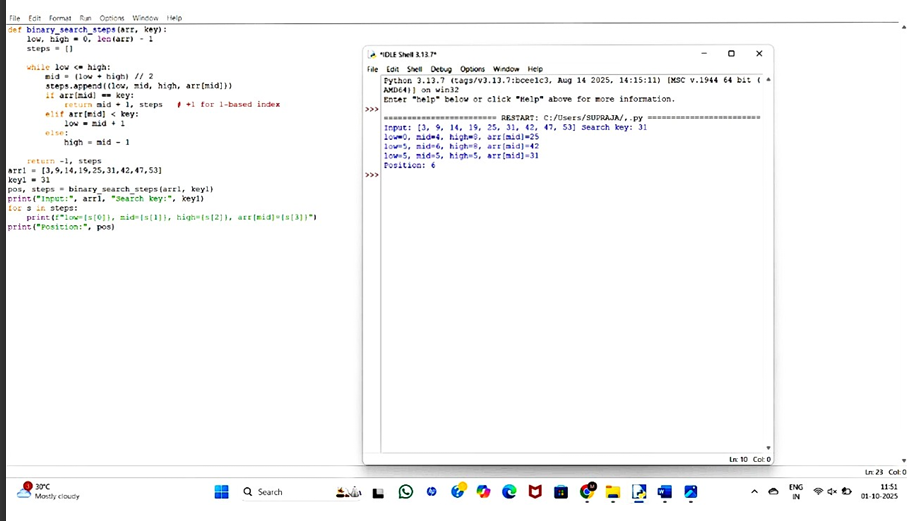
**37.**

****

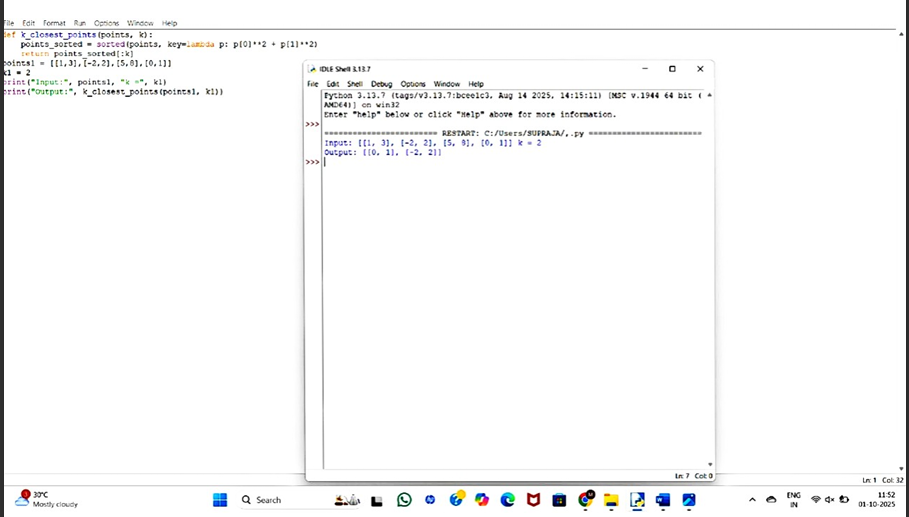
**38.**

****

**39.**

****

**40.**

****