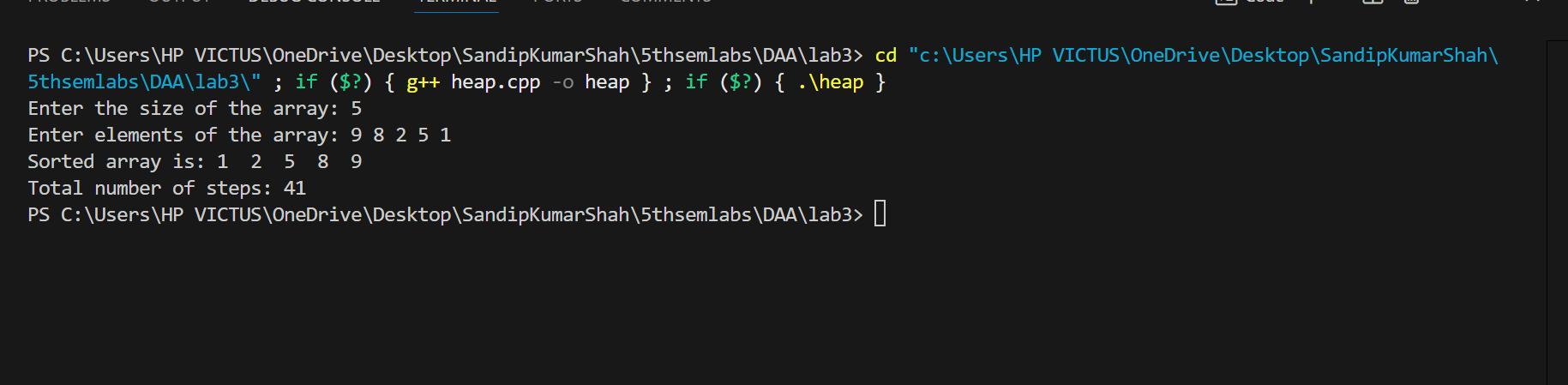
**// Heap:**

**Analysis of Heap sort:**

* **Time Complexity:**
  + Heapify: O(logn) per node.
  + Building the heap: O(n).
  + Sorting: O(nlogn).   
    **Overall: O(nlogn).**
* **Space Complexity:**
  + O(1), as sorting is done in place.

**Output:**

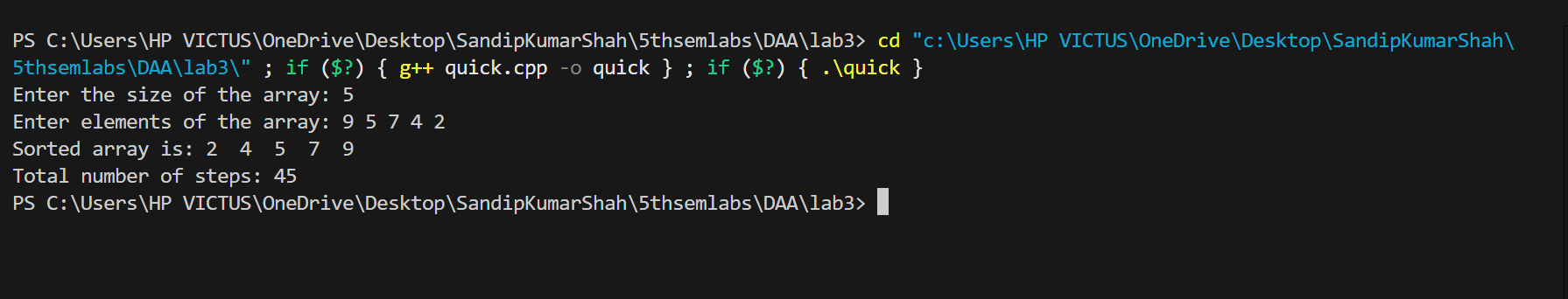
****

**// Quick Sort:**

**Analysis:**

* **Time Complexity:**
  + Best/Average Case: O(nlogn).
  + Worst Case (skewed partition): O(n2).
* **Space Complexity:**
  + O(logn) for recursion stack in the best case; O(n) in the worst case.

**Output:**

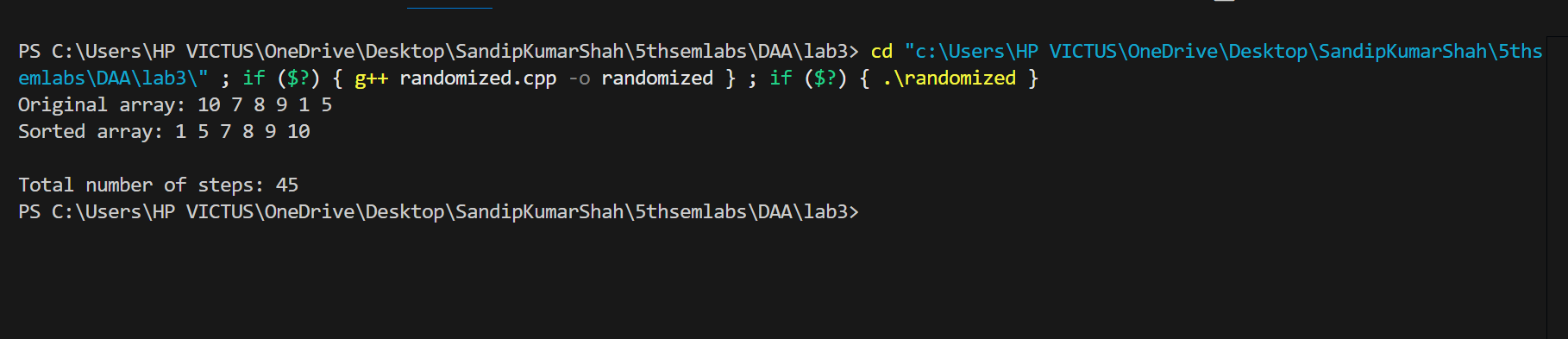
****

**// Randomized Quick sort:**

**Analysis:**

* **Time Complexity:**
  + Best/Average Case: O(nlogn).
  + Worst Case (rare due to randomization): O(n2).
* **Space Complexity:**
  + O(logn) on average for recursion stack.

**Output:**

****

**Conclusion:**

Heap Sort, Quick Sort, and Randomized Quick Sort are efficient sorting algorithms, each with distinct characteristics. Heap Sort uses a binary heap data structure to repeatedly extract the maximum element, providing a time complexity of O(n log n) and being reliable for large datasets. Quick Sort, a divide-and-conquer algorithm, partitions the array around a pivot, with an average time complexity of O(n log n) but a worst-case time complexity of O(n^2), which can be mitigated by Randomized Quick Sort. The latter improves performance by randomly selecting pivots, reducing the likelihood of encountering the worst-case scenario, making it more efficient for varied datasets.