**Problem Statement:** Stepwise Execution Analysis of Sorting Algorithms I

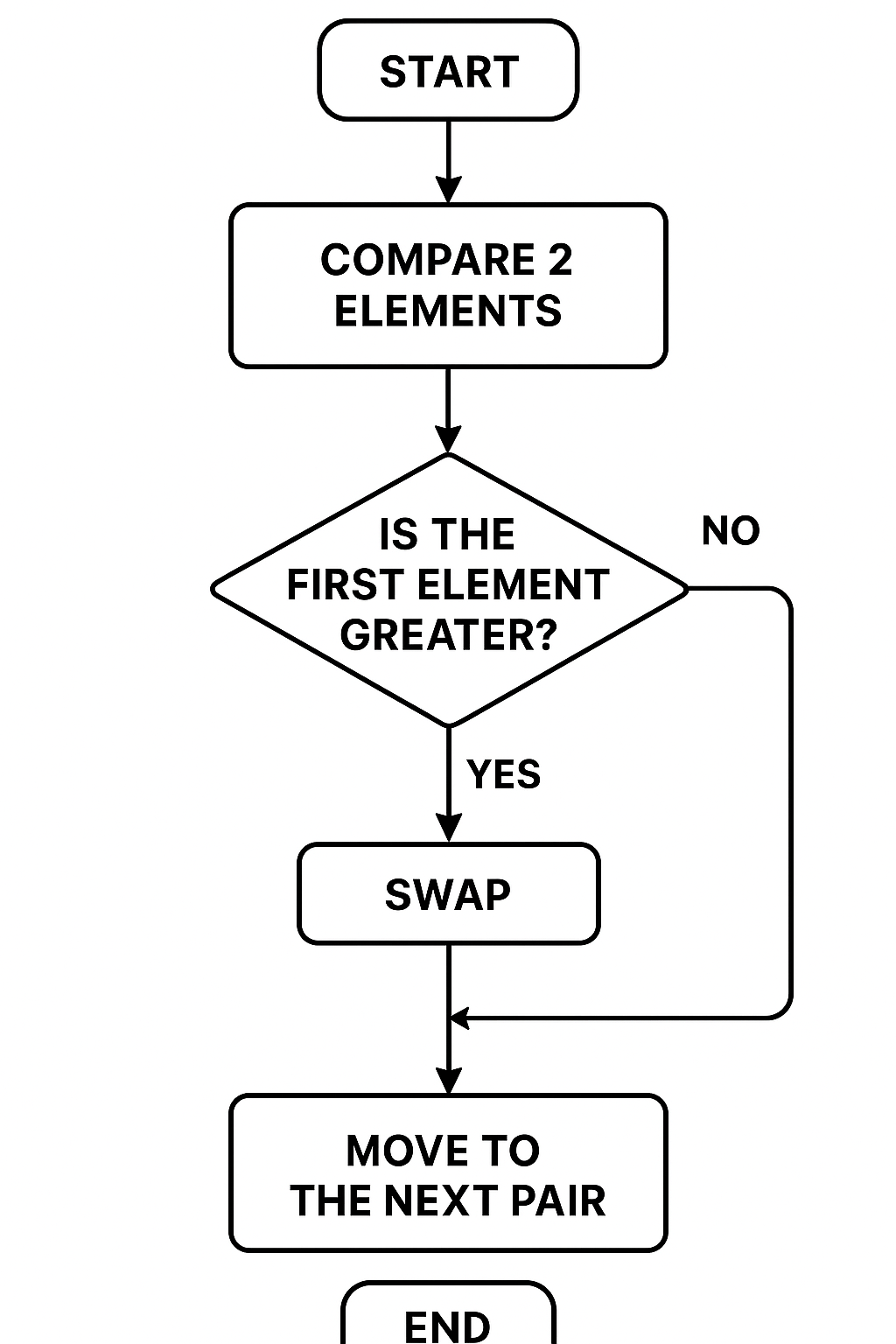
**Bubble Sort**

**Theory:**

**Algorithm:**

1. Look at the first two elements; if the first one is bigger, **switch their positions**.
2. Go to the next pair and do the same comparison and switch if needed.
3. Keep doing this until you reach the end of the list — now the **biggest value has moved to the last position.**
4. Repeat the process for the rest of the list, leaving out the part that is already sorted at the end.
5. Stop when a full round happens with **no swaps,** meaning the list is completely sorted.

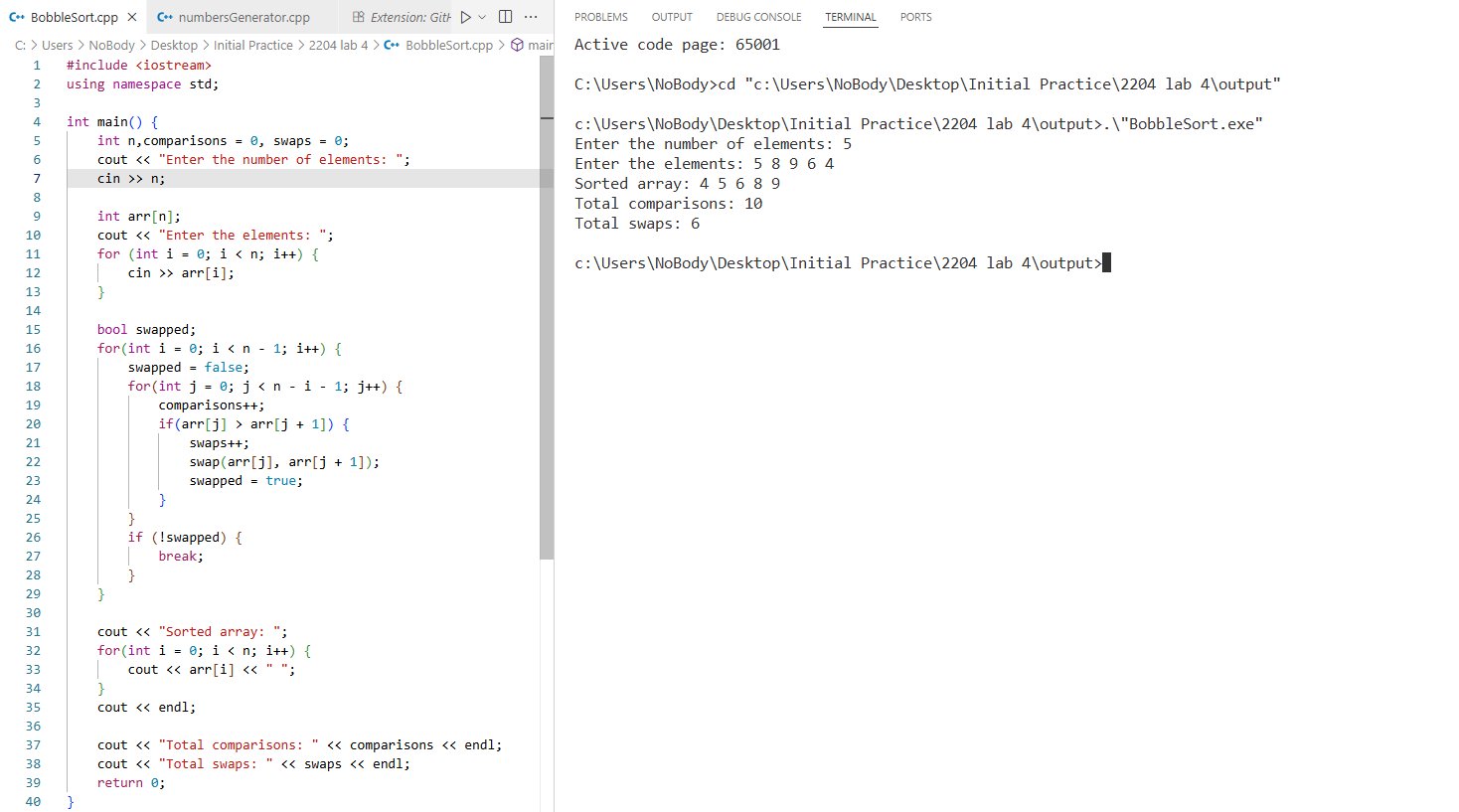
**Flowchart:**



**Code:**

1. #include <iostream>
2. **using** **namespace** std;
4. **int** main() {
5. **int** n,comparisons = 0, swaps = 0;
6. cout << "Enter the number of elements: ";
7. cin >> n;
9. **int** arr[n];
10. cout << "Enter the elements: ";
11. **for** (**int** i = 0; i < n; i++) {
12. cin >> arr[i];
13. }
15. **bool** swapped;
16. **for**(**int** i = 0; i < n - 1; i++) {
17. swapped = **false**;
18. **for**(**int** j = 0; j < n - i - 1; j++) {
19. comparisons++;
20. **if**(arr[j] > arr[j + 1]) {
21. swaps++;
22. swap(arr[j], arr[j + 1]);
23. swapped = **true**;
24. }
25. }
26. **if** (!swapped) {
27. **break**;
28. }
29. }
31. cout << "Sorted array: ";
32. **for**(**int** i = 0; i < n; i++) {
33. cout << arr[i] << " ";
34. }
35. cout << endl;
37. cout << "Total comparisons: " << comparisons << endl;
38. cout << "Total swaps: " << swaps << endl;
39. **return** 0;
40. }

**Screenshots:**



**Analysis:**

Input Array: A = [5, 8, 9, 6, 4]

Stepwise Execution (Pass by Pass)

Pass 1 (i = 1):

Compare 5 and 8 → no swap → [5, 8, 9, 6, 4]

Compare 8 and 9 → no swap → [5, 8, 9, 6, 4]

Compare 9 and 6 → swap → [5, 8, 6, 9, 4]

Compare 9 and 4 → swap → [5, 8, 6, 4, 9]

Pass 2 (i = 2):

Compare 5 and 8 → no swap → [5, 8, 6, 4, 9]

Compare 8 and 6 → swap → [5, 6, 8, 4, 9]

Compare 8 and 4 → swap → [5, 6, 4, 8, 9]

Pass 3 (i = 3):

Compare 5 and 6 → no swap → [5, 6, 4, 8, 9]

Compare 6 and 4 → swap → [5, 4, 6, 8, 9]

Pass 4 (i = 4):

Compare 5 and 4 → swap → [4, 5, 6, 8, 9]

Final Sorted Array: [4, 5, 6, 8, 9]

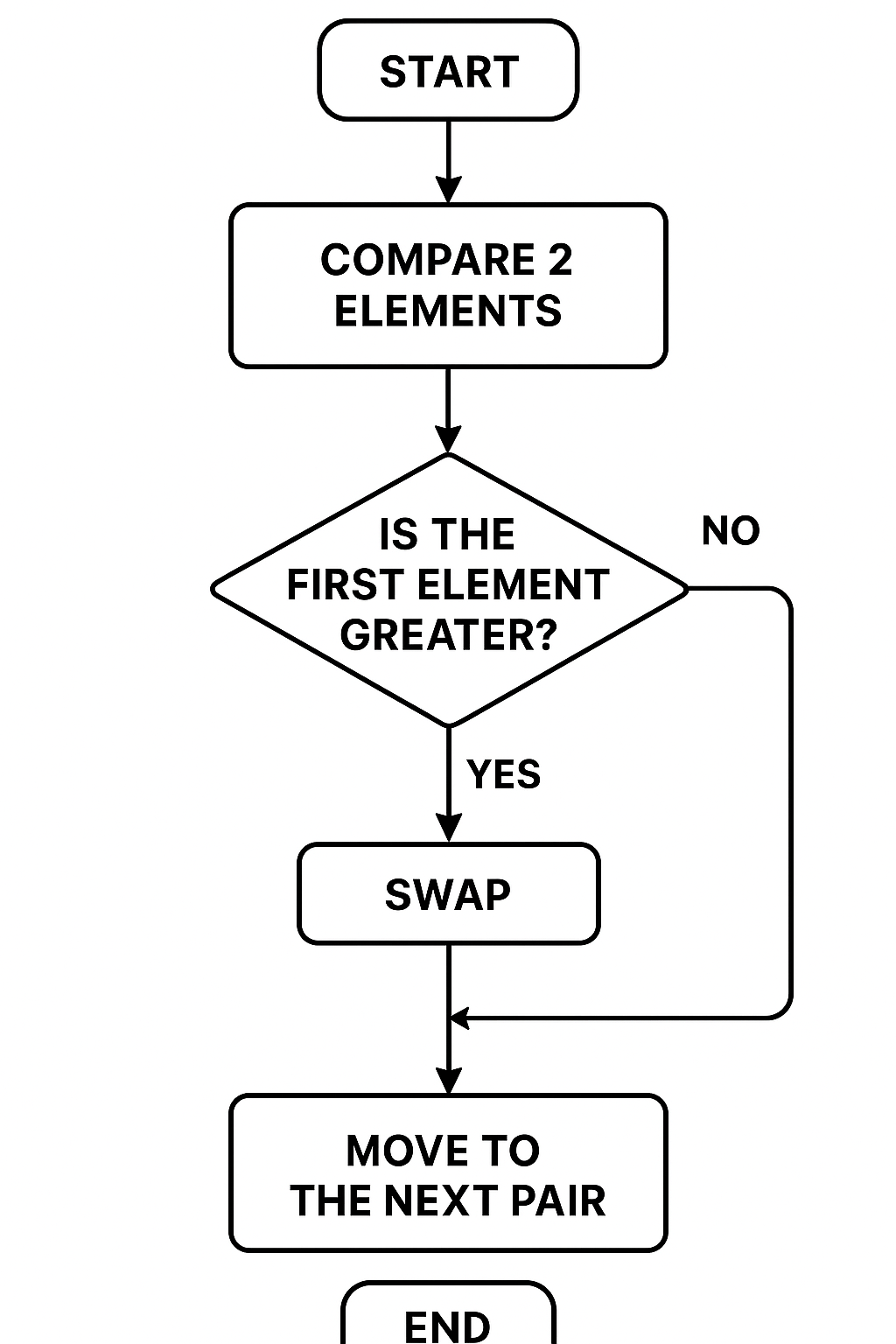
**Selection Sort**

**Theory:**

**Algorithm:**

1. Begin with the first element and scan the entire list to locate the smallest value.
2. Swap that smallest value with the first element.
3. Shift your focus to the second element and repeat the search within the remaining unsorted part of the list.
4. Keep repeating this process until every element is in its correct position and the list is fully sorted.

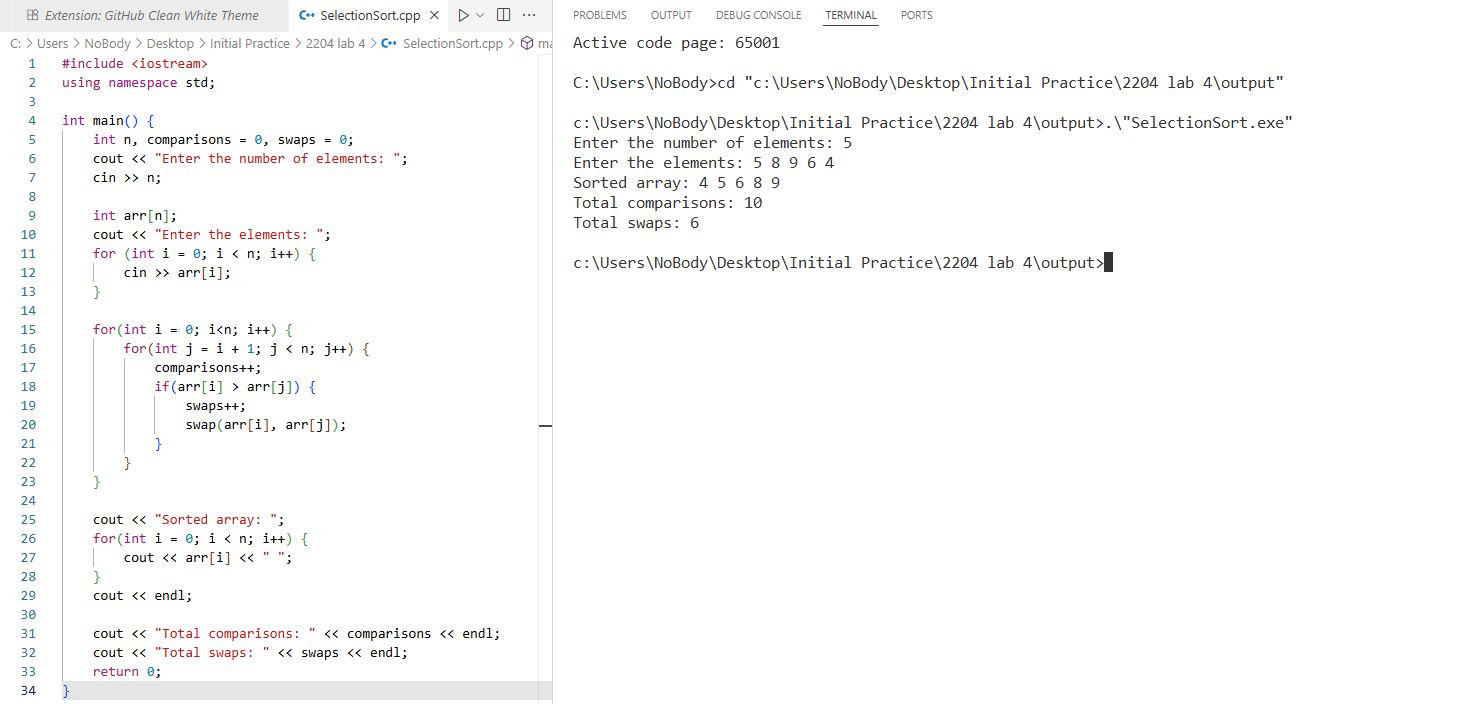
**Flowchart:**



**Code:**

1. #include <iostream>
2. **using** **namespace** std;
4. **int** main() {
5. **int** n, comparisons = 0, swaps = 0;
6. cout << "Enter the number of elements: ";
7. cin >> n;
9. **int** arr[n];
10. cout << "Enter the elements: ";
11. **for** (**int** i = 0; i < n; i++) {
12. cin >> arr[i];
13. }
15. **for**(**int** i = 0; i<n; i++) {
16. **for**(**int** j = i + 1; j < n; j++) {
17. comparisons++;
18. **if**(arr[i] > arr[j]) {
19. swaps++;
20. swap(arr[i], arr[j]);
21. }
22. }
23. }
25. cout << "Sorted array: ";
26. **for**(**int** i = 0; i < n; i++) {
27. cout << arr[i] << " ";
28. }
29. cout << endl;
31. cout << "Total comparisons: " << comparisons << endl;
32. cout << "Total swaps: " << swaps << endl;
33. **return** 0;
34. }

**Screenshots:**



**Analysis:**

Initial Array:

[5, 8, 9, 6, 4]

Pass 1 (First position)

* Find the smallest element in the entire array → 4
* Swap 4 with the first element 5

Array after Pass 1: [4, 8, 9, 6, 5]

Pass 2 (Second position)

* Look through the remaining array [8, 9, 6, 5]
* Smallest element is 5
* Swap 5 with 8

Array after Pass 2: [4, 5, 9, 6, 8]

Pass 3 (Third position)

* Look through the remaining [9, 6, 8]
* Smallest element is 6
* Swap 6 with 9

Array after Pass 3: [4, 5, 6, 9, 8]

Pass 4 (Fourth position)

* Look through the remaining [9, 8]
* Smallest element is 8
* Swap 8 with 9

Array after Pass 4: [4, 5, 6, 8, 9]

Pass 5 (Last position)

* Only one element left, no action needed.

Final Sorted Array: [4, 5, 6, 8, 9]

**Observation Table**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Test Case No. | Input Size (n) | Number of Comparisons | | Number of Swap/Shift | | Execution Time | | |
| Bubble Sort | Selection Sort | Bubble Sort | Selection Sort | Bubble Sort | Selection Sort |
| 01 | 10 (Best) | 9 | 45 | 0 | 0 | 4.012 s | 9.454 s |
| 02 | 10 (Average) | 44 | 45 | 19 | 15 | 6.629 s | 7.907 s |
| 03 | 10 (Worst) | 45 | 45 | 45 | 45 | 2.387 s | 7.548 s |
| 04 | 100 (Best) | 99 | 4950 | 0 | 0 | 2.936 s | 7.325 s |
| 05 | 100 (Average) | 4814 | 4950 | 2404 | 1490 | 7.265 s | 7.465 s |
| 06 | 100 (Worst) | 4950 | 4950 | 4950 | 4950 | 4.136 s | 8.598 s |
| 07 | 1000 (Best) | 999 | 499500 | 0 | 0 | 5.044 s | 6.935 s |
| 08 | 1000 (Average) | 498465 | 499500 | 241331 | 171500 | 4.063 s | 8.547 s |
| 09 | 1000 (Worst) | 499500 | 499500 | 499500 | 499500 | 3.203 s | 8.487 s |

Conclusion:

For small datasets, both Bubble Sort and Selection Sort can be used, but their efficiency varies with data conditions. Bubble Sort performs well when the data is nearly sorted (best case) due to minimal comparisons and zero swaps, but it becomes inefficient as data size grows or in worst-case scenarios. Selection Sort, on the other hand, performs a fixed number of comparisons regardless of the data’s order, making it predictable but generally slower for small, nearly sorted datasets. For large datasets, neither algorithm is practical due to their O(n²) complexity, but Selection Sort may be slightly more consistent than Bubble Sort.