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

**Bubble Sort**

**Theory:**

* Initialize an integer array with user input
* Sort the array using Bubble Sort:
* Initialize two counters: cntSwap (to count swaps) and cntCompare (to count comparisons).
* For each pass from the first element to the second last element:
* Set a boolean flag swapped to false.
* For each element from the start up to the last unsorted element (which decreases with each

pass):

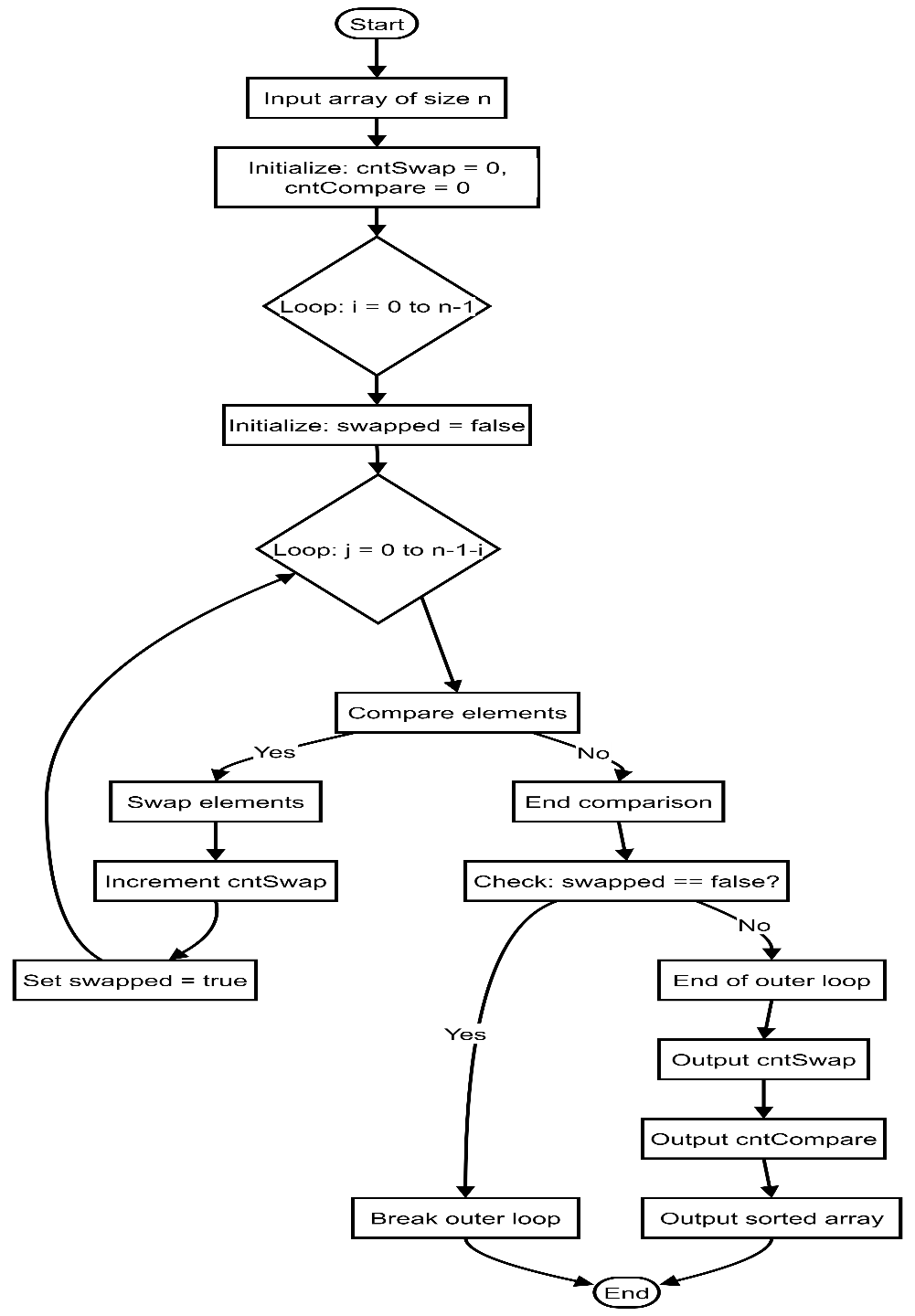
* Compare the current element with the next element.
* Increment the comparison counter cntCompare.
* If the current element is greater than the next element:

Swap the two elements.

Set swapped to true.

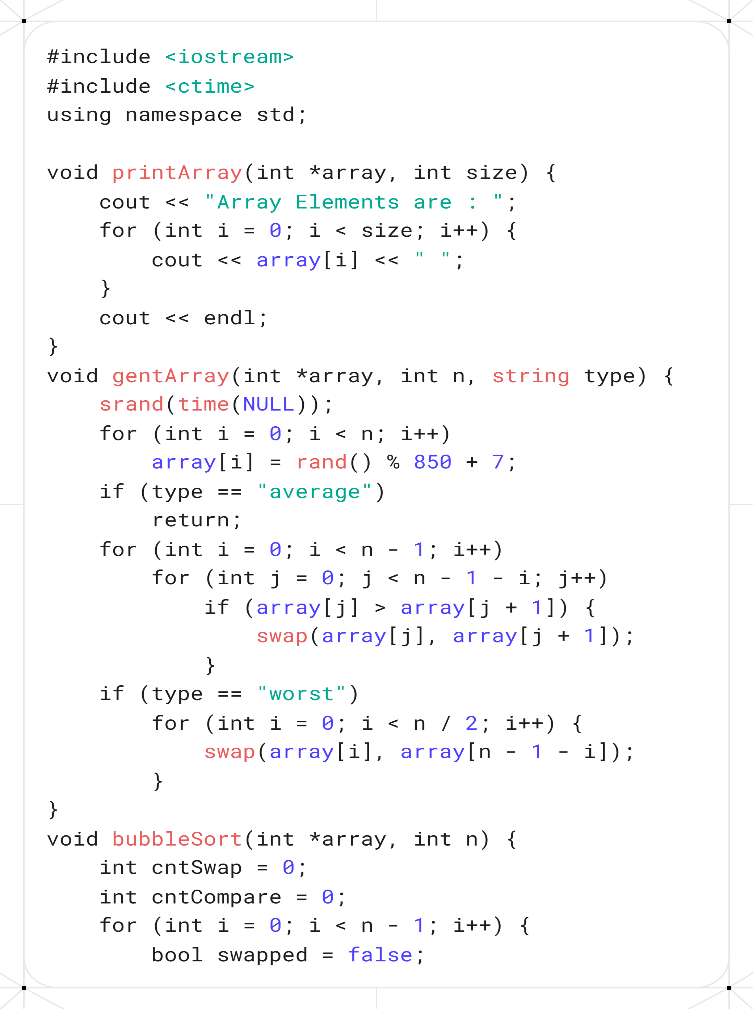
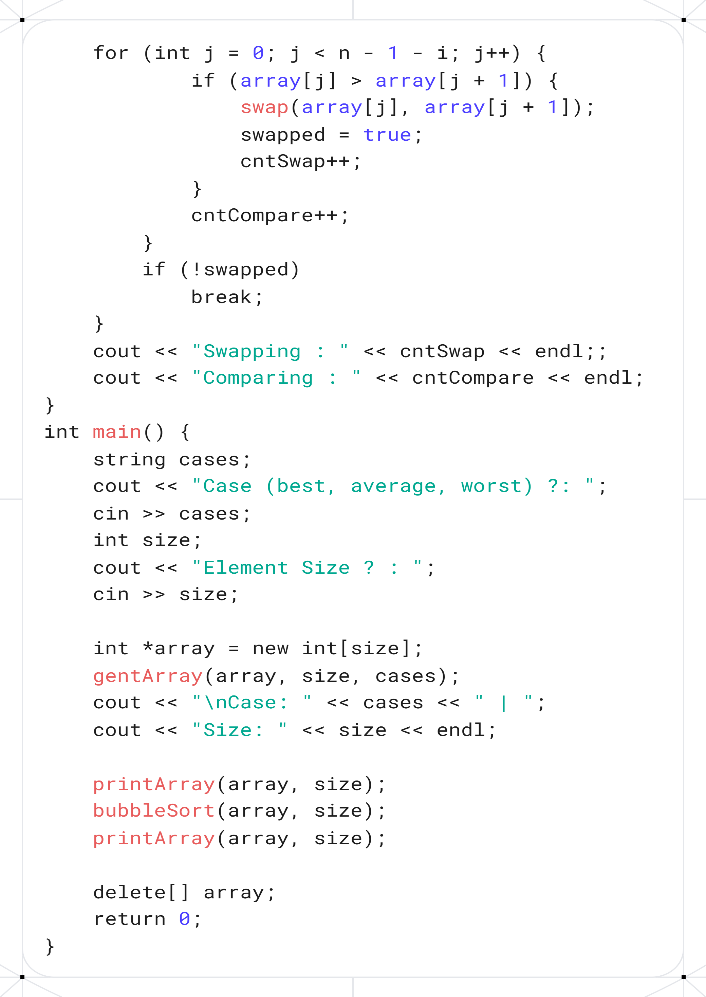
* Increment the swap counter cntSwap.
* If no swaps occurred during this pass (swapped is false), the array is sorted; break out of

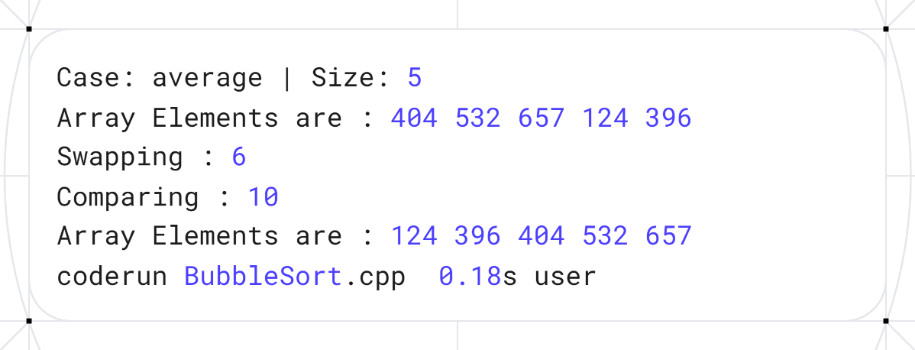
the loop early.

* Print the total number of swaps and comparisons performed during the sorting process.

**Code:**

|  |
| --- |
| 1. #include <iostream> 2. #include <ctime> 3. **using** **namespace** std; 4. **void** printArray(**int** \*array, **int** size) { 5. cout << "Array Elements are : "; 6. **for** (**int** i = 0; i < size; i++) { 7. cout << array[i] << " "; 8. } 9. cout << endl; 10. } 11. **void** gentArray(**int** \*array, **int** n, string type) { 12. **srand**(**time**(NULL)); 13. **for** (**int** i = 0; i < n; i++) 14. array[i] = **rand**() % 850 + 7; 15. **if** (type == "average") 16. **return**; 17. **for** (**int** i = 0; i < n - 1; i++) 18. **for** (**int** j = 0; j < n - 1 - i; j++) 19. **if** (array[j] > array[j + 1]) { 20. swap(array[j], array[j + 1]); 21. } 22. **if** (type == "worst") 23. **for** (**int** i = 0; i < n / 2; i++) { 24. swap(array[i], array[n - 1 - i]); 25. } 26. } 27. **void** bubbleSort(**int** \*array, **int** n) { 28. **int** cntSwap = 0; 29. **int** cntCompare = 0; 30. **for** (**int** i = 0; i < n - 1; i++) { 31. **bool** swapped = **false**; 32. **for** (**int** j = 0; j < n - 1 - i; j++) { 33. **if** (array[j] > array[j + 1]) { 34. swap(array[j], array[j + 1]); 35. swapped = **true**; 36. cntSwap++; 37. } 38. cntCompare++; 39. } 40. **if** (!swapped) 41. **break**; 42. } 43. cout << "Swapping : " << cntSwap << endl;; 44. cout << "Comparing : " << cntCompare << endl; 45. } 46. **int** main() { 47. string cases; 48. cout << "Case (best, average, worst) ?: "; 49. cin >> cases; 50. **int** size; 51. cout << "Element Size ? : "; 52. cin >> size; 54. **int** \*array = **new** **int**[size]; 55. gentArray(array, size, cases); 56. cout << "\nCase: " << cases << " | "; 57. cout << "Size: " << size << endl; 59. printArray(array, size); 60. bubbleSort(array, size); 61. printArray(array, size); 62. **delete**[] array; 63. **return** 0; 64. } |
|  |

**Screenshots:**

****

**Analysis:** Input Array**: A = [404, 532, 657, 124, 396]**

**Stepwise Execution (Pass by Pass):**

**Pass 1 (i = 1):**

* Compare 404 and 532 → no swap → [404, 532, 657, 124, 396]
* Compare 532 and 657 → no swap → [404, 532, 657, 124, 396]
* Compare 657 and 124 → swap → [404, 532, 124, 657, 396]
* Compare 657 and 396 → swap → [404, 532, 124, 396, 657]

**Pass 2 (i = 2):**

* Compare 404 and 532 → no swap → [404, 532, 124, 396, 657]
* Compare 532 and 124 → swap → [404, 124, 532, 396, 657]
* Compare 532 and 396 → swap → [404, 124, 396, 532, 657]

**Pass 3 (i = 3):**

* Compare 404 and 124 → swap → [124, 404, 396, 532, 657]
* Compare 404 and 396 → swap → [124, 396, 404, 532, 657]

**Pass 4 (i = 4):**

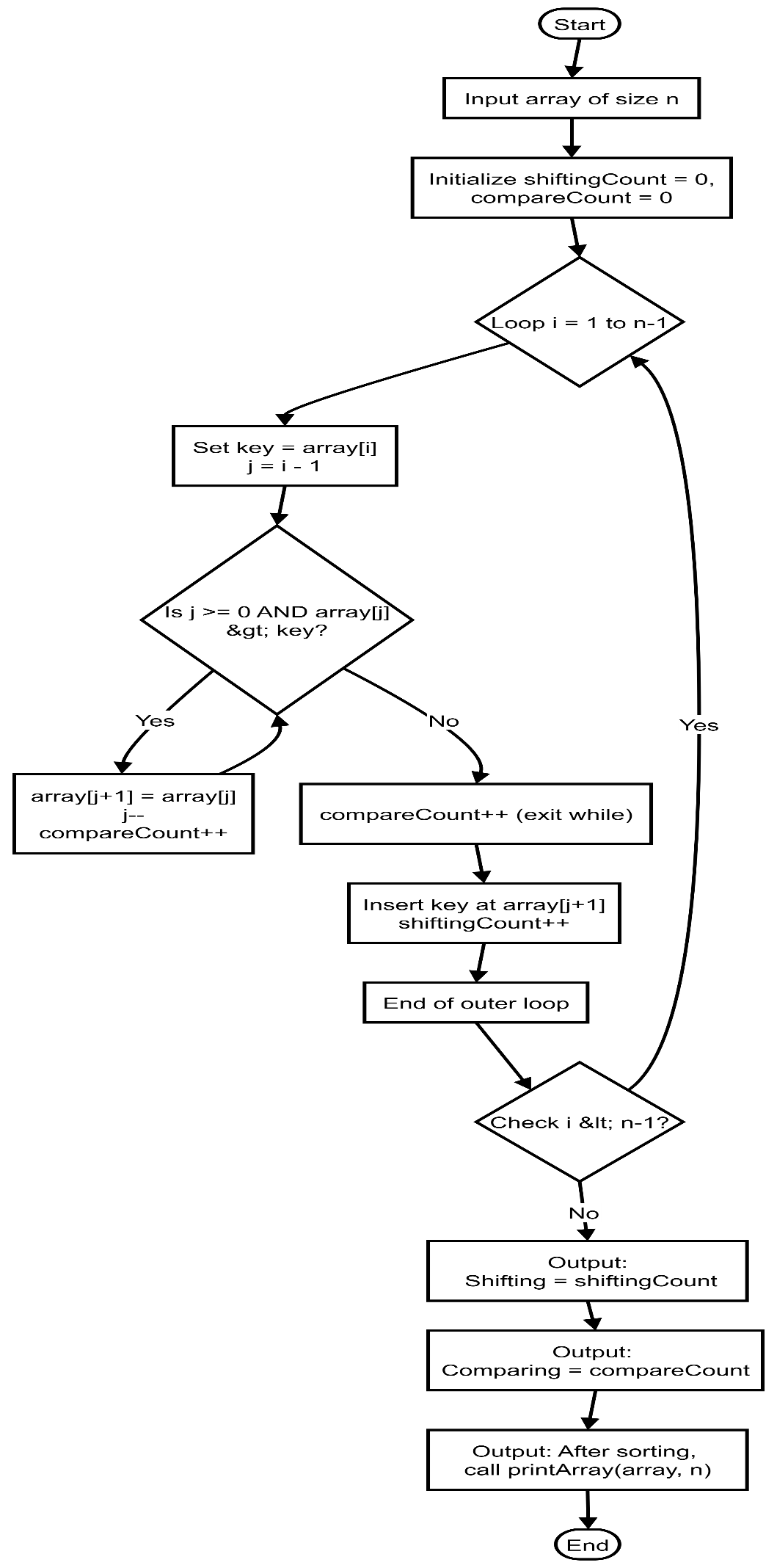
* Compare 124 and 396 → no swap → [124, 396, 404, 532, 657]

**Final Sorted Array**: [124, 396, 404, 532, 657]

**Insertion Sort**

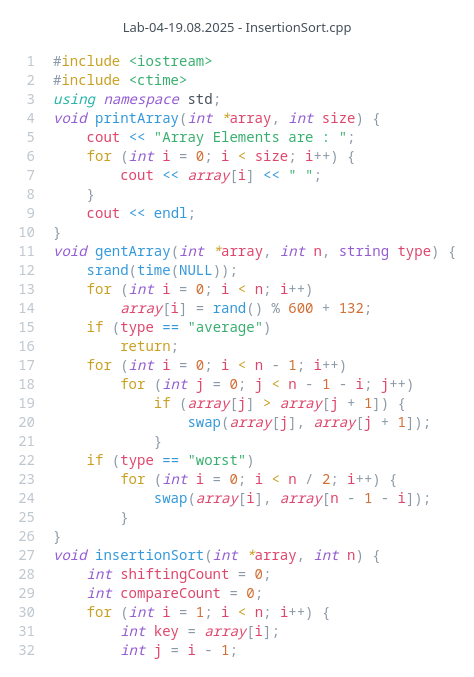
**Theory:**

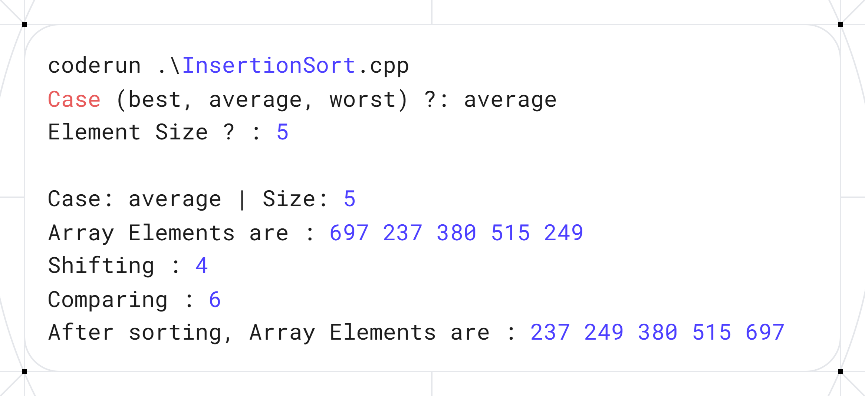
1. Initialize an integer array with user input.
2. Print the array elements:
   * Iterate through the array from the first to the last element.
   * Print each element followed by a space.
3. Sort the array using Insertion Sort:
   * Initialize two counters: shiftingCount (to count the number of shifts) and compareCount (to count the number of comparisons).
   * For each element from the second element (index 1) to the last:
     + Store the current element as key.
     + Initialize j as the index before the current element.
     + While j is non-negative and the element at j is greater than key:
       - Shift the element at j one position to the right (to j + 1).
       - Decrement j.
       - Increment the comparison counter compareCount.
     + Insert the key at position j + 1.
     + Increment the shifting counter shiftingCount.
4. Print the total number of shifts and comparisons performed during sorting.
5. Print the sorted array elements:
   * Iterate through the array and print each element followed by a space.



**Code:**

1. #include <iostream>
2. #include <ctime>
3. **using** **namespace** std;
4. **void** printArray(**int** \*array, **int** size) {
5. cout << "Array Elements are : ";
6. **for** (**int** i = 0; i < size; i++) {
7. cout << array[i] << " ";
8. }
9. cout << endl;
10. }
11. **void** gentArray(**int** \*array, **int** n, string type) {
12. **srand**(**time**(NULL));
13. **for** (**int** i = 0; i < n; i++)
14. array[i] = **rand**() % 600 + 132;
15. **if** (type == "average")
16. **return**;
17. **for** (**int** i = 0; i < n - 1; i++)
18. **for** (**int** j = 0; j < n - 1 - i; j++)
19. **if** (array[j] > array[j + 1]) {
20. swap(array[j], array[j + 1]);
21. }
22. **if** (type == "worst")
23. **for** (**int** i = 0; i < n / 2; i++) {
24. swap(array[i], array[n - 1 - i]);
25. }
26. }
27. **void** insertionSort(**int** \*array, **int** n) {
28. **int** shiftingCount = 0;
29. **int** compareCount = 0;
30. **for** (**int** i = 1; i < n; i++) {
31. **int** key = array[i];
32. **int** j = i - 1;
33. **while** (j >= 0) {
34. compareCount++;
35. if(array[j] > key) {
36. array[j + 1] = array[j];
37. j--;
38. shiftingCount++;
39. } else {
40. break;
41. }
42. }
43. array[j + 1] = key;
44. }
45. cout << "Shifting : " << shiftingCount << endl;
46. cout << "Comparing : " << compareCount << endl;
47. cout << "After sorting, ";
48. printArray(array, n);
49. }
50. **int** main() {
51. string cases;
52. cout << "Case (best, average, worst) ?: ";
53. cin >> cases;
54. **int** size;
55. cout << "Element Size ? : ";
56. cin >> size;
57. **int** \*array = **new** **int**[size];
58. gentArray(array, size, cases);
59. cout << "\nCase: " << cases << " | ";
60. cout << "Size: " << size << endl;
62. printArray(array, size);
63. insertionSort(array, size);
64. **delete**[] array;
65. **return** 0;
66. }

**** **Screenshots:**

****

**Analysis:** Input Array: **A = [697, 237, 380, 515, 249]**

#### Stepwise Execution (Pass by Pass):

**Pass 1 (i = 1):**

* Key = 237
* Compare 697 and 237 → 697 > 237 → shift 697 right → [697, 697, 380, 515, 249]
* Insert 237 at position 0 → [237, 697, 380, 515, 249]

**Pass 2 (i = 2):**

* Key = 380
* Compare 697 and 380 → 697 > 380 → shift 697 right → [237, 697, 697, 515, 249]
* Compare 237 and 380 → 237 < 380 → stop shifting
* Insert 380 at position 1 → [237, 380, 697, 515, 249]

**Pass 3 (i = 3):**

* Key = 515
* Compare 697 and 515 → 697 > 515 → shift 697 right → [237, 380, 697, 697, 249]
* Compare 380 and 515 → 380 < 515 → stop shifting
* Insert 515 at position 2 → [237, 380, 515, 697, 249]

**Pass 4 (i = 4):**

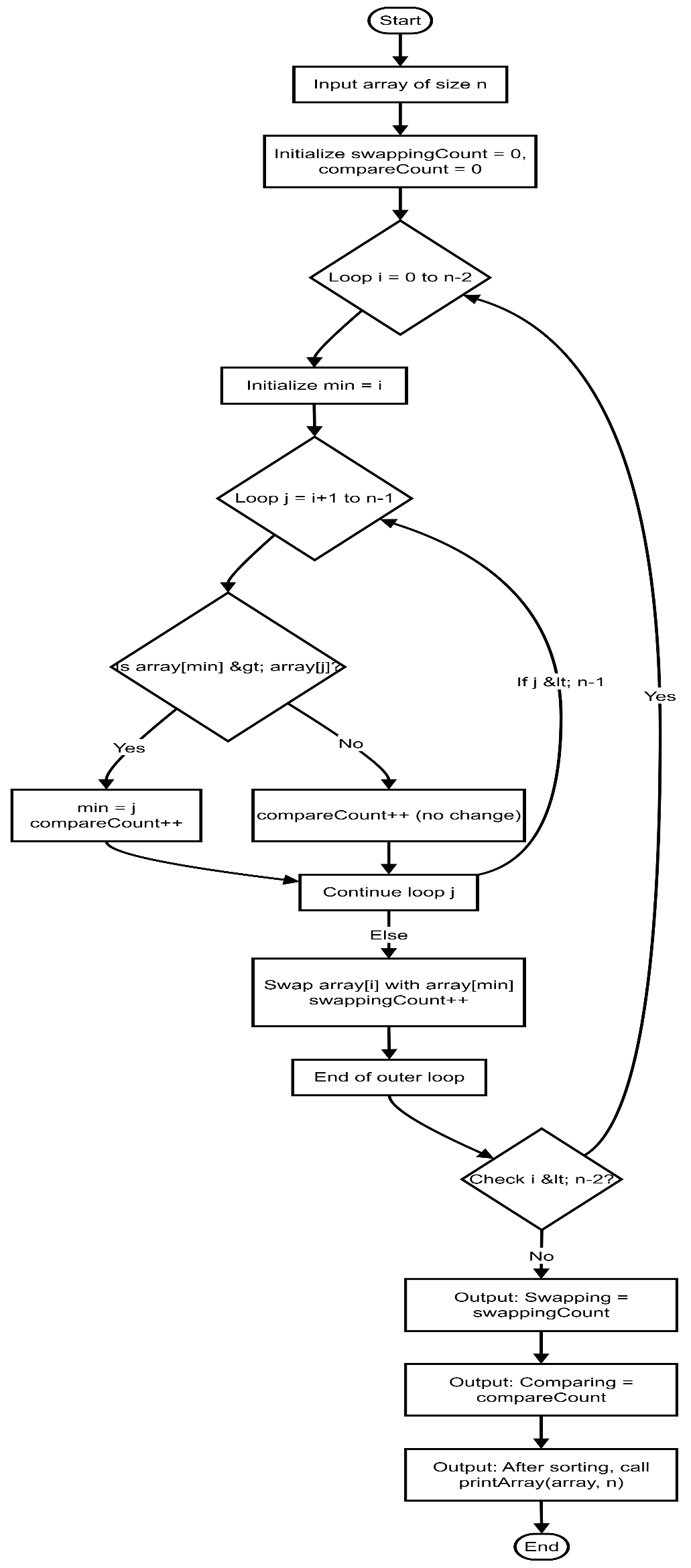
* Key = 249
* Compare 697 and 249 → 697 > 249 → shift 697 right → [237, 380, 515, 697, 697]
* Compare 515 and 249 → 515 > 249 → shift 515 right → [237, 380, 515, 515, 697]
* Compare 380 and 249 → 380 > 249 → shift 380 right → [237, 380, 380, 515, 697]
* Compare 237 and 249 → 237 < 249 → stop shifting
* Insert 249 at position 1 → [237, 249, 380, 515, 697]

### **Final Sorted Array:** [237, 249, 380, 515, 697]

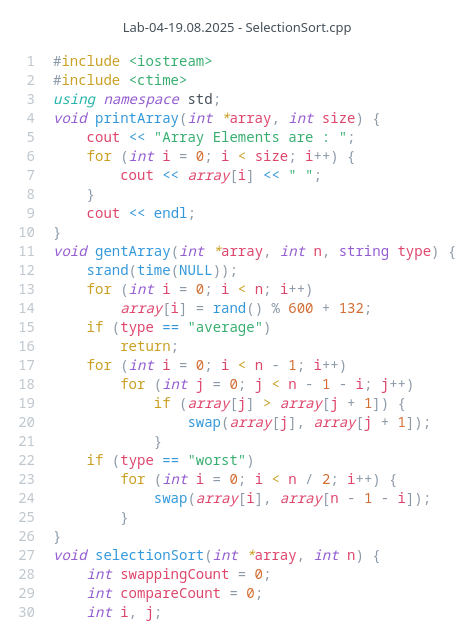
**Selection Sort**

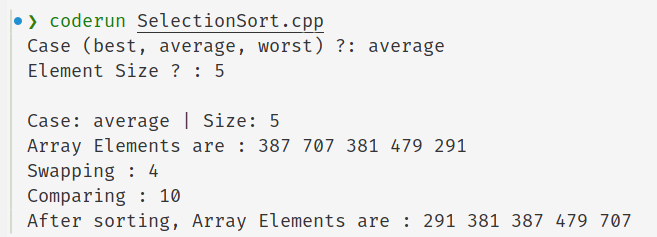
**Theory:**

1. **Initialize an integer array**.
2. **Print the array elements**:
   * Iterate through the array from the first element to the last.
   * Print each element followed by a space.
3. **Sort the array using Selection Sort**:
   * Initialize two counters: swappingCount (to count swaps) and compareCount (to count comparisons).
   * For each index i from 0 to n - 2:
     + Set min to i (assume the current element is the minimum).
     + For each index j from i + 1 to n - 1:
       - Compare the element at min with the element at j.
       - Increment compareCount.
       - If the element at j is smaller than the element at min, update min to j.
     + Swap the element at i with the element at min.
     + Increment swappingCount.
4. **Print the total number of swaps and comparisons** performed during sorting.
5. **Print the sorted array elements**:
   * Iterate through the array and print each element followed by a space.

**Code:**

1. #include <iostream>
2. #include <ctime>
3. **using** **namespace** std;
4. **void** printArray(**int** \*array, **int** size) {
5. cout << "Array Elements are : ";
6. **for** (**int** i = 0; i < size; i++) {
7. cout << array[i] << " ";
8. }
9. cout << endl;
10. }
11. **void** gentArray(**int** \*array, **int** n, string type) {
12. **srand**(**time**(NULL));
13. **for** (**int** i = 0; i < n; i++)
14. array[i] = **rand**() % 600 + 132;
15. **if** (type == "average")
16. **return**;
17. **for** (**int** i = 0; i < n - 1; i++)
18. **for** (**int** j = 0; j < n - 1 - i; j++)
19. **if** (array[j] > array[j + 1]) {
20. swap(array[j], array[j + 1]);
21. }
22. **if** (type == "worst")
23. **for** (**int** i = 0; i < n / 2; i++) {
24. swap(array[i], array[n - 1 - i]);
25. }
26. }
27. **void** selectionSort(**int** \*array, **int** n) {
28. **int** swappingCount = 0;
29. **int** compareCount = 0;
30. **int** i, j;
31. **for** (i = 0; i < n - 1; i++) {
32. **int** min = i;
33. **for** (j = i + 1; j < n; j++) {
34. compareCount++;
35. **if** (array[min] > array[j])
36. min = j;
37. }
38. if (min != 1) {
39. swap(array[i], array[min]);
40. swappingCount++;
41. }
42. }
43. }
44. cout << "Swapping : " << swappingCount << endl;
45. cout << "Comparing : " << compareCount << endl;
46. cout << "After sorting, ";
47. printArray(array, n);
48. }
49. **int** main() {
50. string cases;
51. cout << "Case (best, average, worst) ?: ";
52. cin >> cases;
53. **int** size;
54. cout << "Element Size ? : ";
55. cin >> size;
56. **int** \*array = **new** **int**[size];
57. gentArray(array, size, cases);
58. cout << "\nCase: " << cases << " | ";
59. cout << "Size: " << size << endl;
61. printArray(array, size);
62. selectionSort(array, size);
64. **delete**[] array;
65. **return** 0;
66. }

**Screenshots:**

****

**Analysis:** Input Array**: A = [387, 707, 381, 479, 291]**

**Stepwise Execution (Pass by Pass)**

**Pass 1 (i = 0):**

* Initial minimum index = 0 (value 387)
* Compare 707 with 387 → 707 > 387 → minimum stays at index 0
* Compare 381 with 387 → 381 < 387 → update minimum index to 2
* Compare 479 with 381 → 479 > 381 → minimum stays at index 2
* Compare 291 with 381 → 291 < 381 → update minimum index to 4
* Swap element at index 0 (387) with element at index 4 (291)
* Array after **pass 1:** **[291, 707, 381, 479, 387]**

**Pass 2 (i = 1):**

* Initial minimum index = 1 (value 707)
* Compare 381 with 707 → 381 < 707 → update minimum index to 2
* Compare 479 with 381 → 479 > 381 → minimum stays at index 2
* Compare 387 with 381 → 387 > 381 → minimum stays at index 2
* Swap element at index 1 (707) with element at index 2 (381)
* Array after **pass 2: [291, 381, 707, 479, 387]**

**Pass 3 (i = 2):**

* Initial minimum index = 2 (value 707)
* Compare 479 with 707 → 479 < 707 → update minimum index to 3
* Compare 387 with 479 → 387 < 479 → update minimum index to 4
* Swap element at index 2 (707) with element at index 4 (387)
* Array after **pass 3: [291, 381, 387, 479, 707]**

**Pass 4 (i = 3):**

* Initial minimum index = 3 (value 479)
* Compare 707 with 479 → 707 > 479 → minimum stays at index 3
* Swap element at index 3 with itself (no change)
* Array after **pass 4: [291, 381, 387, 479, 707]**

**Final Sorted Array: [291, 381, 387, 479, 707]**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Test Case No. | Input Size (n) | Number of Comparison | | | Number of Swap/Shift | | | Execution Time | | |
| Bubble Sort | Insertion Sort | Selection Sort | Bubble Sort | Insertion Sort | Selection Sort | Bubble Sort | Insertion Sort | Selection Sort |
| 01 | 10 (Best) | 9 | 9 | 45 | 0 | 0 | 0 | 0.21s | 0.21s | 0.21s |
| 02 | 10 (Average) | 45 | 34 | 45 | 26 | 27 | 6 | 0.20s | 0.20s | 0.20s |
| 03 | 10 (Worst) | 45 | 45 | 45 | 45 | 45 | 5 | 0.20s | 0.22s | 0.20s |
| 04 | 100 (Best) | 99 | 99 | 4950 | 0 | 0 | 0 | 0.21s | 0.21s | 0.20s |
| 05 | 100 (Average) | 4884 | 2864 | 4950 | 2465 | 2768 | 94 | 0.21s | 0.20s | 0.21s |
| 06 | 100 (Worst) | 4950 | 4950 | 4950 | 4944 | 4943 | 53 | 0.21s | 0.20s | 0.21s |
| 07 | 1000 (Best) | 999 | 999 | 499500 | 0 | 0 | 0 | 0.20s | 0.20s | 0.22s |
| 08 | 1000 (Average) | 499224 | 248094 | 499500 | 251345 | 247103 | 993 | 0.21s | 0.20s | 0.20s |
| 09 | 1000 (Worst) | 499499 | 499146 | 499500 | 498928 | 498633 | 634 | 0.21s | 0.21s | 0.21s |

**Observation Table**

**Conclusion:**

Bubble Sort is the simplest to understand but the least efficient. It is mainly appropriate for very small or nearly sorted datasets, and is often used for educational purposes rather than practical applications. Selection Sort is most suitable when minimizing the number of swaps is more important than speed. Although it performs poorly on large datasets, it can be useful in cases where writing to memory is costly. Insertion Sort is the most efficient of the three for small or nearly sorted datasets. Because it adapts quickly to partially ordered data, it is often used as a helper algorithm inside more advanced sorting techniques. In short, Bubble Sort is chosen mainly for teaching, not real-world use. Insertion Sort is chosen for small or nearly sorted data and Selection Sort is chosen when swap cost matters.