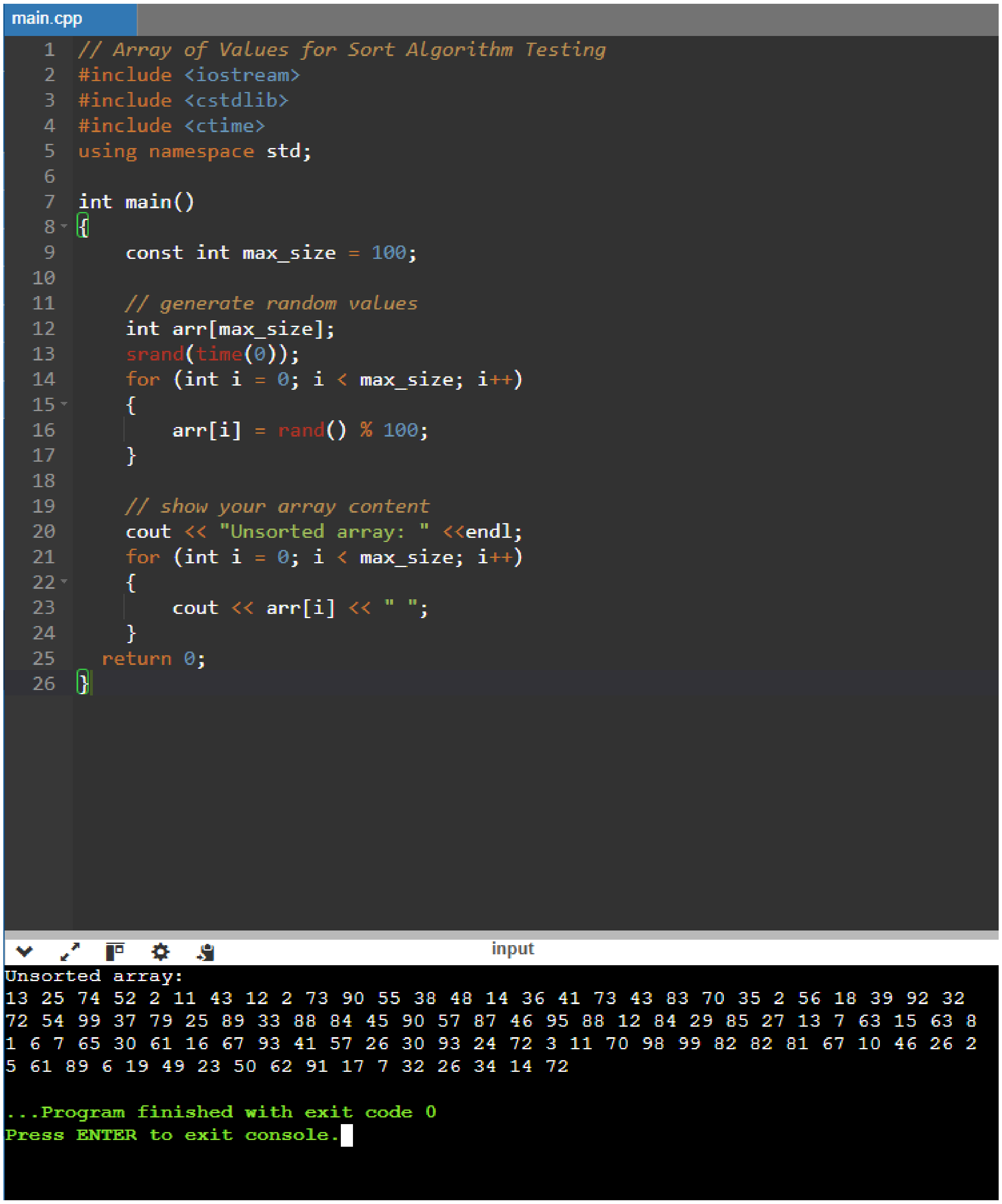
Activity No. 8	
SORTING ALGORITHMS: SHELL, MERGE, AND QUICK SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** October 21, 2024
Section: CPE21S4	**Date Submitted:** October 23, 2024
Name(s): Bona, Andrei Nycole So	**Instructor:** Prof. Maria Rizetter Sayo

6. Output

Code + Console Screenshot



main.cpp

```cpp
// Array of Values for Sort Algorithm Testing
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    const int max_size = 100;

    // generate random values
    int arr[max_size];
    srand(time(0));
    for (int i = 0; i < max_size; i++)
    {
        arr[i] = rand() % 100;
    }

    // show your array content
    cout << "Unsorted array: " <<endl;
    for (int i = 0; i < max_size; i++)
    {
        cout << arr[i] << " ";
    }
  return 0;
}
```

input

```
Unsorted array:
13 25 74 52 2 11 43 12 2 73 90 55 38 48 14 36 41 73 43 83 70 35 2 56 18 39 92 32
72 54 99 37 79 25 89 33 88 84 45 90 57 87 46 95 88 12 84 29 85 27 13 7 63 15 63 8
1 6 7 65 30 61 16 67 93 41 57 26 30 93 24 72 3 11 70 98 99 82 82 81 67 10 46 26 2
5 61 89 6 19 49 23 50 62 91 17 7 32 26 34 14 72

...Program finished with exit code 0
Press ENTER to exit console.
```

Observations	In this C++ code, there are two for loops. One, to generate 100 random numbers to put inside an array and Two, to display the generated random numbers.

Table 8-1. Array of Values for Sort Algorithm Testing

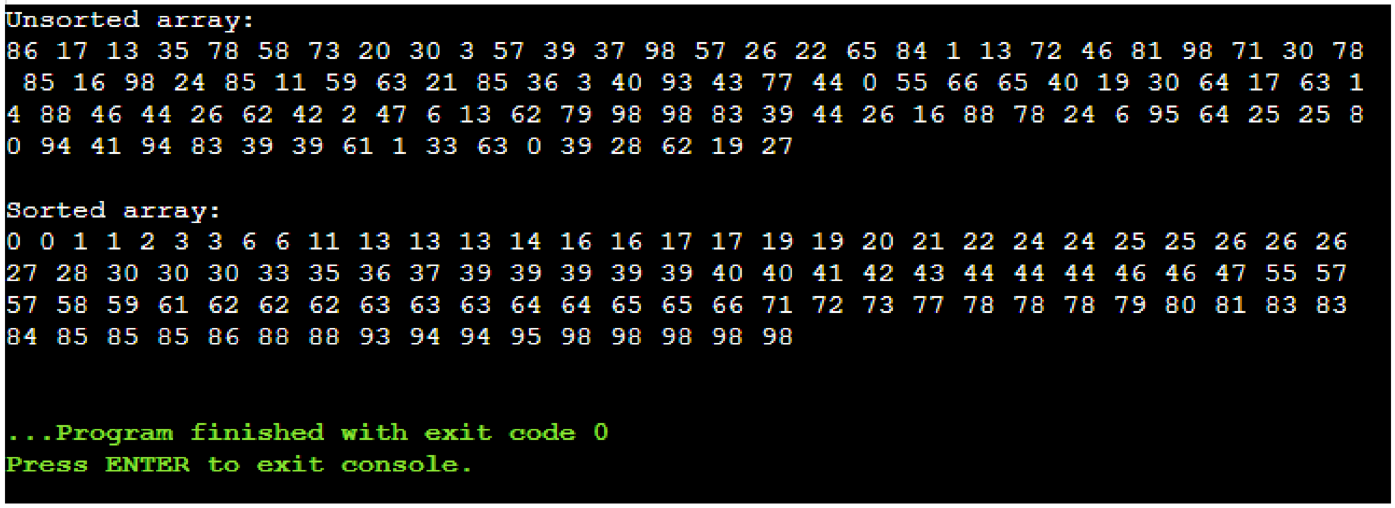
Code + Console Screenshot

main.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "sort.h"
using namespace std;

int main()
{
    const int max_size = 5;

    // generate random values
    int arr[max_size];
    srand(time(0));
    for (int i = 0; i < max_size; i++)
            {
        arr[i] = rand() % 100;
    }

    // show your array content
    cout << "Unsorted array: " <<endl;
    for (int i = 0; i < max_size; i++)
            {
        cout << arr[i] << " ";
    }

    size_t arrSize = sizeof(arr) / sizeof(arr[0]);

    cout <<endl;
    cout <<endl;


    shellSort(arr, arrSize);



    cout << "Sorted array: " << endl;
    for (size_t i = 0; i < arrSize; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

sort.h

```
void shellSort(int arr[], int size)
{

        for (int interval = size / 2; interval > 0; interval /=
2)
        {
            for (int i = interval; i < size; i++) {
                int temp = arr[i];
                int j;

                for (j = i; j >= interval && arr[j - interval] >
temp; j -= interval) {
                    arr[j] = arr[j - interval];
                }
                arr[j] = temp;
            }
        }
    }
```

```
Unsorted array:
86 17 13 35 78 58 73 20 30 3 57 39 37 98 57 26 22 65 84 1 13 72 46 81 98 71 30 78
 85 16 98 24 85 11 59 63 21 85 36 3 40 93 43 77 44 0 55 66 65 40 19 30 64 17 63 1
4 88 46 44 26 62 42 2 47 6 13 62 79 98 98 83 39 44 26 16 88 78 24 6 95 64 25 25 8
0 94 41 94 83 39 39 61 1 33 63 0 39 28 62 19 27

Sorted array:
0 0 1 1 2 3 3 6 6 11 13 13 13 14 16 16 17 17 19 19 20 21 22 24 24 25 25 26 26 26
27 28 30 30 30 33 35 36 37 39 39 39 39 39 40 40 41 42 43 44 44 44 46 46 47 55 57
57 58 59 61 62 62 62 63 63 63 64 64 65 65 66 71 72 73 77 78 78 78 79 80 81 83 83
84 85 85 85 86 88 88 93 94 94 95 98 98 98 98 98


...Program finished with exit code 0
Press ENTER to exit console.
```

Observations	The Shell Sort in sort.h works by breaking the array into smaller groups based on a gap, starting at half the array size. As the gap gets smaller with each step, the elements are compared and rearranged if needed.

Table 8-2. Shell Sort Technique

Code + Console Screenshot

main.cpp

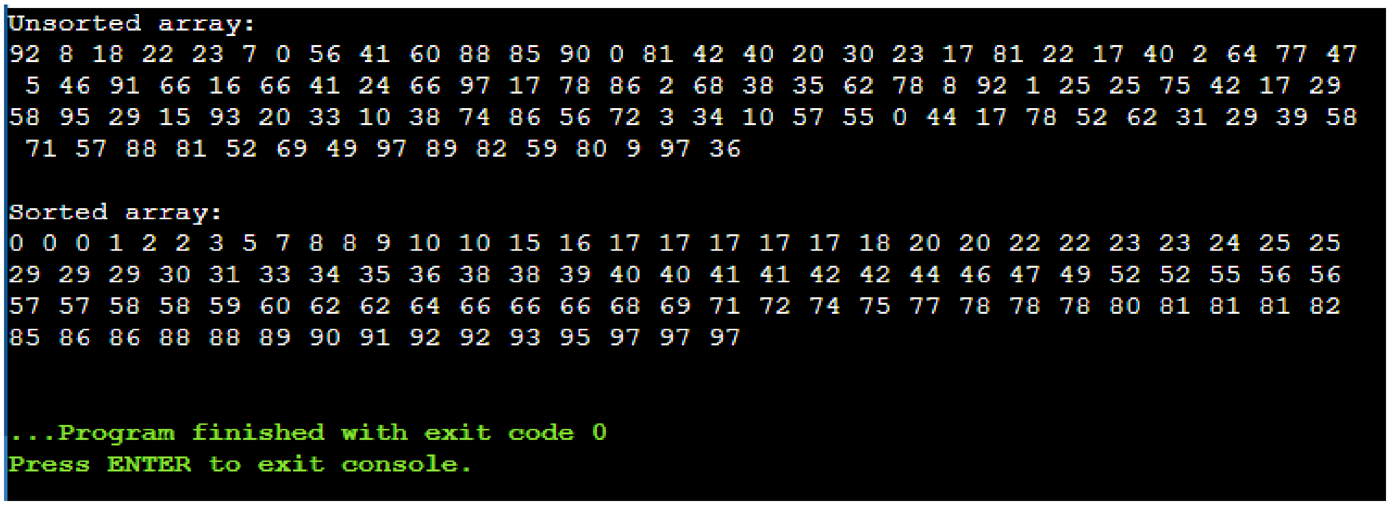
```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "sort.h"
using namespace std;

int main()
{
    const int max_size = 5;

    // generate random values
    int arr[max_size];
    srand(time(0));
    for (int i = 0; i < max_size; i++)
            {
        arr[i] = rand() % 100;
    }

    // show your array content
    cout << "Unsorted array: " <<endl;
    for (int i = 0; i < max_size; i++)
            {
        cout << arr[i] << " ";
    }
```

sort.h

```
void merge(int arr[], int left, int middle, int right)
{
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int* L = new int[n1];
    int* R = new int[n2];


    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];


    int i = 0;
    int j = 0;
    int k = left;

    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
```

```
    size_t arrSize = sizeof(arr) / sizeof(arr[0]);

    cout <<endl;
    cout <<endl;


    mergeSort(arr, 0, max_size - 1);

    cout << "Sorted array: " << endl;
    for (size_t i = 0; i < arrSize; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

```
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }


    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    delete[] L;
    delete[] R;
}


void mergeSort(int arr[], int left, int right)
{
    if (left < right) {

        int middle = left + (right - left) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);

        merge(arr, left, middle, right);
    }
}
```



```
Unsorted array:
92 8 18 22 23 7 0 56 41 60 88 85 90 0 81 42 40 20 30 23 17 81 22 17 40 2 64 77 47
 5 46 91 66 16 66 41 24 66 97 17 78 86 2 68 38 35 62 78 8 92 1 25 25 75 42 17 29
58 95 29 15 93 20 33 10 38 74 86 56 72 3 34 10 57 55 0 44 17 78 52 62 31 29 39 58
 71 57 88 81 52 69 49 97 89 82 59 80 9 97 36

Sorted array:
0 0 0 1 2 2 3 5 7 8 8 9 10 10 15 16 17 17 17 17 17 18 20 20 22 22 23 23 24 25 25
29 29 29 30 31 33 34 35 36 38 38 39 40 40 41 41 42 42 44 46 47 49 52 52 55 56 56
57 57 58 58 59 60 62 62 64 66 66 66 68 69 71 72 74 75 77 78 78 78 80 81 81 81 82
85 86 86 88 88 89 90 91 92 92 93 95 97 97 97


...Program finished with exit code 0
Press ENTER to exit console.
```

Observations	The Merge Sort works by breaking the array into smaller pieces and then putting them back together in the right order. The mergeSort function splits the array down to single elements, and the merge function combines them by comparing the pieces and sorting them as they go back into the main array.

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot

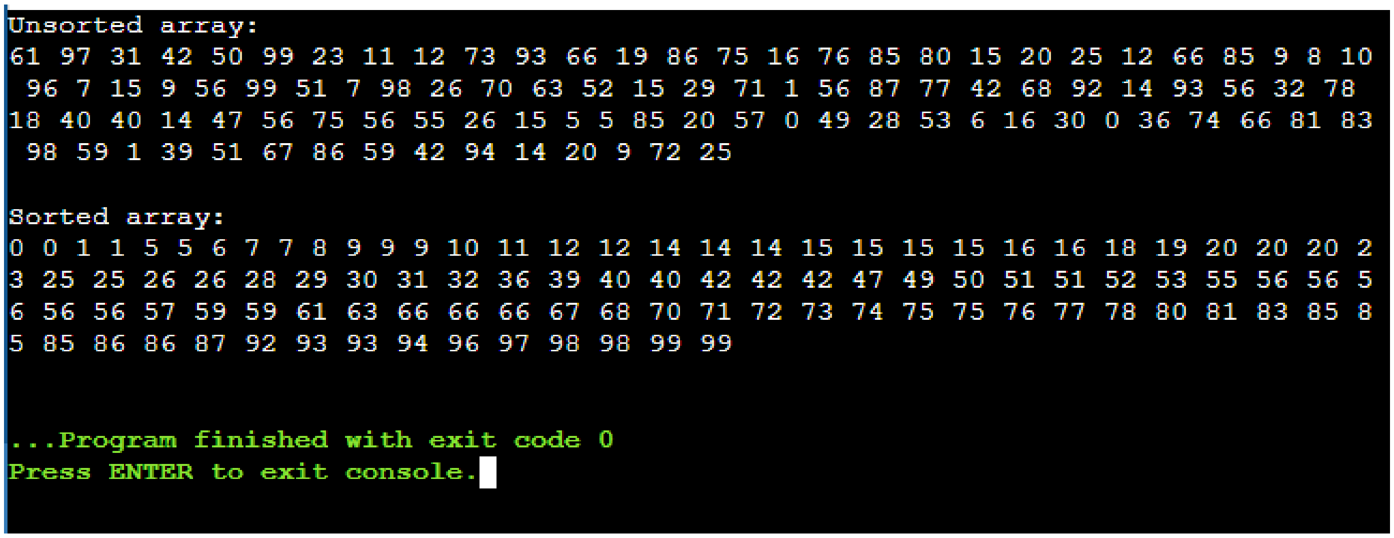
main.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "sort.h"
using namespace std;

int main()
{
    const int max_size = 5;

    // generate random values
    int arr[max_size];
    srand(time(0));
    for (int i = 0; i < max_size; i++)
            {
        arr[i] = rand() % 100;
    }

    // show your array content
    cout << "Unsorted array: " <<endl;
    for (int i = 0; i < max_size; i++)
            {
        cout << arr[i] << " ";
    }

    size_t arrSize = sizeof(arr) / sizeof(arr[0]);
```

sort.h

```
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}
```

```
    cout <<endl;
    cout <<endl;


    quickSort(arr, 0, max_size - 1);

    cout << "Sorted array: " << endl;
    for (size_t i = 0; i < arrSize; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```



```
Unsorted array:
61 97 31 42 50 99 23 11 12 73 93 66 19 86 75 16 76 85 80 15 20 25 12 66 85 9 8 10
 96 7 15 9 56 99 51 7 98 26 70 63 52 15 29 71 1 56 87 77 42 68 92 14 93 56 32 78
18 40 40 14 47 56 75 56 55 26 15 5 5 85 20 57 0 49 28 53 6 16 30 0 36 74 66 81 83
 98 59 1 39 51 67 86 59 42 94 14 20 9 72 25

Sorted array:
0 0 1 1 5 5 6 7 7 8 9 9 9 10 11 12 12 14 14 14 15 15 15 15 16 16 18 19 20 20 20 2
3 25 25 26 26 28 29 30 31 32 36 39 40 40 42 42 42 47 49 50 51 51 52 53 55 56 56 5
6 56 56 57 59 59 61 63 66 66 66 67 68 70 71 72 73 74 75 75 76 77 78 80 81 83 85 8
5 85 86 86 87 92 93 93 94 96 97 98 98 99 99


...Program finished with exit code 0
Press ENTER to exit console.
```

Observations	The Quick Sort algorithm works by selecting a pivot element and rearranging the array so that elements smaller than the pivot are on one side, and larger elements are on the other. The partition function does this by comparing elements to the pivot and swapping them as needed. Once partitioned, the quickSort function recursively sorts the subarrays on either side of the pivot.

Table 8-4. Quick Sort Algorithm

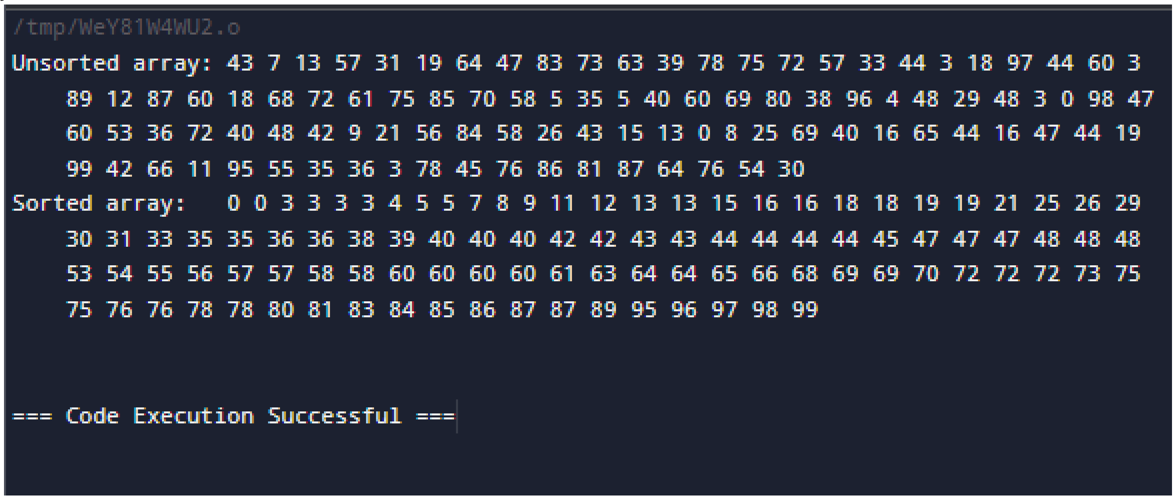
7. Supplementary Activity

Problem 1:

Yes, we can sort the left and right sublists from the partition method in Quick Sort using other sorting algorithms. Here's an example using the Bubble Sort algorithm on the left and right sublists.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;
```

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void bubbleSort(int arr[], int low, int high) {
    for (int i = low; i < high; i++) {
        for (int j = low; j < high - (i - low); j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

void hybridQuickBubbleSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        bubbleSort(arr, low, pivotIndex - 1);
        bubbleSort(arr, pivotIndex + 1, high);
    }
}

int main() {
    const int max_size = 100;
    int arr[max_size];

    srand(time(0));
    for (int i = 0; i < max_size; i++) {
        arr[i] = rand() % 100;
    }

    cout << "Unsorted array: ";
    for (int i = 0; i < max_size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    hybridQuickBubbleSort(arr, 0, max_size - 1);

    cout << "Sorted array:   ";
    for (int i = 0; i < max_size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```



```
/tmp/WeY81W4WU2.o
Unsorted array: 43 7 13 57 31 19 64 47 83 73 63 39 78 75 72 57 33 44 3 18 97 44 60 3
    89 12 87 60 18 68 72 61 75 85 70 58 5 35 5 40 60 69 80 38 96 4 48 29 48 3 0 98 47
    60 53 36 72 40 48 42 9 21 56 84 58 26 43 15 13 0 8 25 69 40 16 65 44 16 47 44 19
    99 42 66 11 95 55 35 36 3 78 45 76 86 81 87 64 76 54 30
Sorted array:   0 0 3 3 3 3 4 5 5 7 8 9 11 12 13 13 15 16 16 18 18 19 19 21 25 26 29
    30 31 33 35 35 36 36 38 39 40 40 40 42 42 43 43 44 44 44 44 45 47 47 47 48 48 48
    53 54 55 56 57 57 58 58 60 60 60 60 61 63 64 64 65 66 68 69 69 70 72 72 72 73 75
    75 76 76 78 78 80 81 83 84 85 86 87 87 89 95 96 97 98 99


=== Code Execution Successful ===
```

Problem 2:

Both algorithms use a divide-and-conquer approach. Merge Sort repeatedly splits the array until each part has one element, requiring logN splits while merging takes O(N) time. Quick Sort partitions the array in O(N) time and recursively reduces the problem size. Quick Sort is often faster because it performs better in practice, while Merge Sort is more predictable and stable. Either way, both algorithms would efficiently sort this array in O(NlogN) time.

8. Conclusion

In this laboratory, I learned other sorting algorithms, these are Merge Sort, Quick Sort, and Shell Sort. I discovered their strengths and weaknesses, particularly how both Merge Sort and Quick Sort have a time complexity due to their divide-and-conquer strategies. Implementing these algorithms helped me understand how theory translates into practice, especially when creating a hybrid sorting algorithm that combines Quick Sort with Bubble Sort.

The process of coding allowed me to evaluate which sorting method to use based on specific scenarios, emphasizing critical thinking in algorithm selection. Overall, I think I did well in this activity, effectively applying the algorithms and understanding their complexities. However, I see areas for improvement, such as increasing my efficiency in implementation and testing my code. This activity has been valuable in strengthening my coding skills and knowledge of algorithms.

9. Assessment Rubric