Activity No. 8

Sorting Algorithms

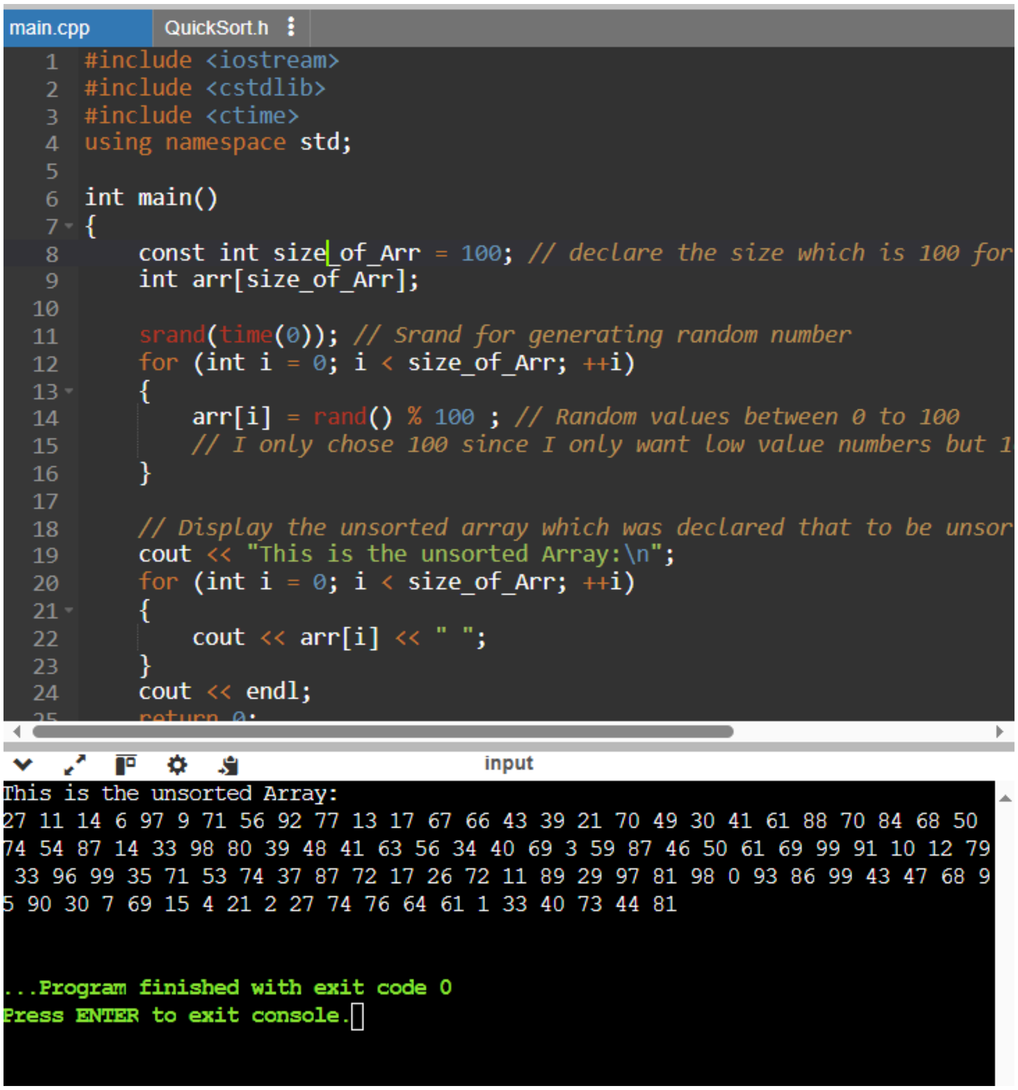
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/21/24**
Section: CPE21S4	**Date Submitted: 10/23/24**
Name(s): SANTOS, ANDREI R.	**Instructor: Professor Maria Rizette Sayo**

6. Output

Code + Console Screenshot

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    const int size_of_Arr = 100; // declare the size which is 100 for the array
    int arr[size_of_Arr];

    srand(time(0)); // Srand for generating random number
    for (int i = 0; i < size_of_Arr; ++i)
    {
        arr[i] = rand() % 100 ; // Random values between 0 to 100
        // I only chose 100 since I only want low value numbers but 100 elements
    }

    // Display the unsorted array which was declared that to be unsorted
    cout << "This is the unsorted Array:\n";
    for (int i = 0; i < size_of_Arr; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

<table>
<tr>
<td></td>
<td>



```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    const int size_of_Arr = 100; // declare the size which is 100 for
    int arr[size_of_Arr];

    srand(time(0)); // Srand for generating random number
    for (int i = 0; i < size_of_Arr; ++i)
    {
        arr[i] = rand() % 100 ; // Random values between 0 to 100
        // I only chose 100 since I only want low value numbers but 1
    }

    // Display the unsorted array which was declared that to be unsor
    cout << "This is the unsorted Array:\n";
    for (int i = 0; i < size_of_Arr; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
```

```
This is the unsorted Array:
27 11 14 6 97 9 71 56 92 77 13 17 67 66 43 39 21 70 49 30 41 61 88 70 84 68 50
74 54 87 14 33 98 80 39 48 41 63 56 34 40 69 3 59 87 46 50 61 69 99 91 10 12 79
 33 96 99 35 71 53 74 37 87 72 17 26 72 11 89 29 97 81 98 0 93 86 99 43 47 68 9
5 90 30 7 69 15 4 21 2 27 74 76 64 61 1 33 40 73 44 81


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

</td>
</tr>
<tr>
<td>Observations</td>
<td>The program generates an array of 100 random integers between 0 and 99 using the rand() function and seeds the random number generator with srand(time(0)) to show random values. It then displays the unsorted array or the raw data.</td>
</tr>
</table>

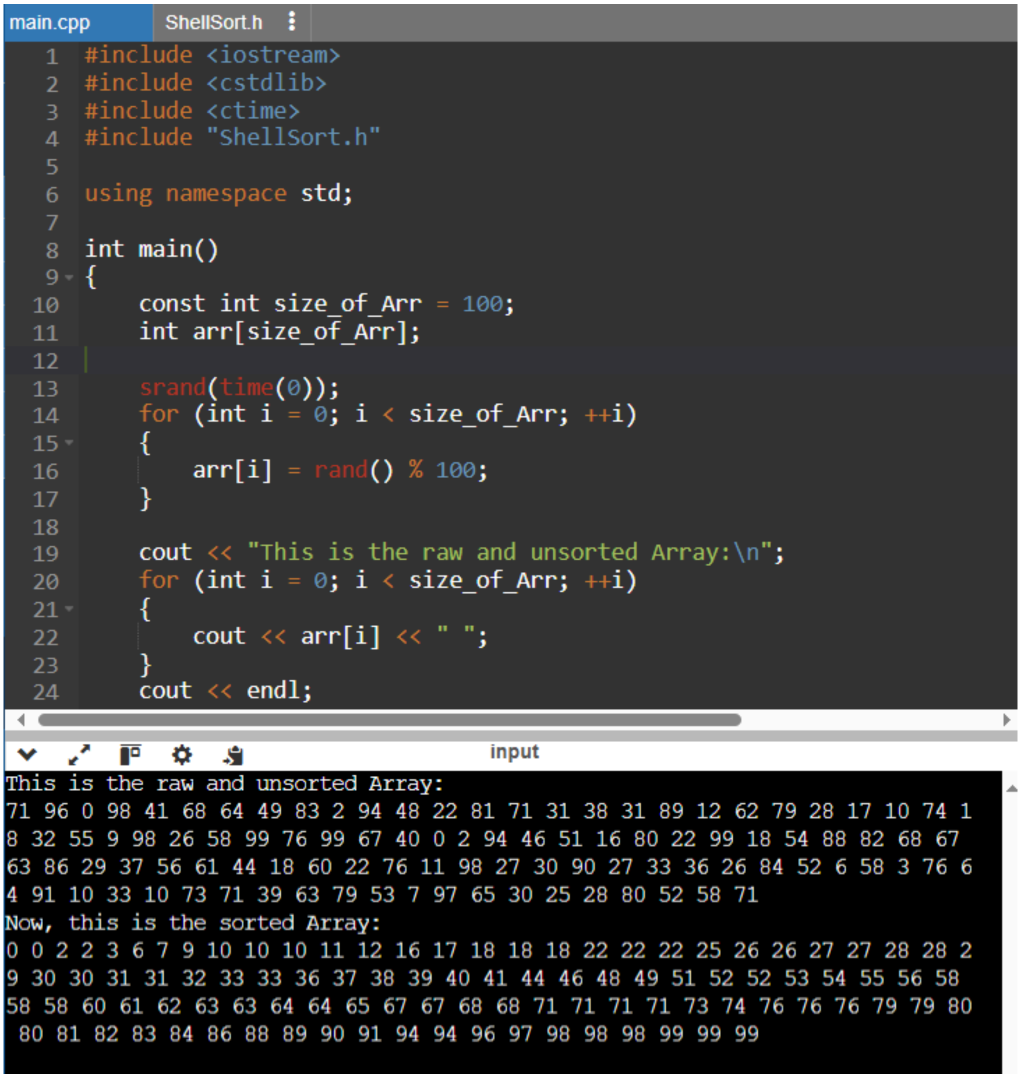
Table 8-1. Array of Values for Sort Algorithm Testing

<table>
<tr>
<td>Code + Console Screenshot</td>
<td>

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "ShellSort.h"

using namespace std;

int main()
```

</td>
</tr>
</table>

```
{
    const int size_of_Arr = 100;
    int arr[size_of_Arr];

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

    cout << "This is the raw and unsorted Array:\n";
    for (int i = 0; i < size_of_Arr; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    shellSort(arr, size_of_Arr);
    cout << "Now, this is the sorted Array:\n";
    for (int i = 0; i < size_of_Arr; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}

#ifndef SHELLSORT_H
#define SHELLSORT_H

void shellSort(int arr[], int size)
{
    for (int intvl = size / 2; intvl > 0; intvl /= 2)
    {
        for (int i = intvl; i < size; i++)
        {
            int temp = arr[i];
            int j;
            for (j = i; j >= intvl && arr[j - intvl] > temp; j -= intvl)
            {
                arr[j] = arr[j - intvl];
            }
            arr[j] = temp;
        }
    }
}

#endif
```



```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "ShellSort.h"

using namespace std;

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

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

    cout << "This is the raw and unsorted Array:\n";
    for (int i = 0; i < size_of_Arr; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
```

```
This is the raw and unsorted Array:
71 96 0 98 41 68 64 49 83 2 94 48 22 81 71 31 38 31 89 12 62 79 28 17 10 74 1
8 32 55 9 98 26 58 99 76 99 67 40 0 2 94 46 51 16 80 22 99 18 54 88 82 68 67
63 86 29 37 56 61 44 18 60 22 76 11 98 27 30 90 27 33 36 26 84 52 6 58 3 76 6
4 91 10 33 10 73 71 39 63 79 53 7 97 65 30 25 28 80 52 58 71
Now, this is the sorted Array:
0 0 2 2 3 6 7 9 10 10 10 11 12 16 17 18 18 18 22 22 22 25 26 26 27 27 28 28 2
9 30 30 31 31 32 33 33 36 37 38 39 40 41 44 46 48 49 51 52 52 53 54 55 56 58
58 58 60 61 62 63 63 64 64 65 67 67 68 68 71 71 71 71 73 74 76 76 76 79 79 80
 80 81 82 83 84 86 88 89 90 91 94 94 96 97 98 98 98 99 99 99
```

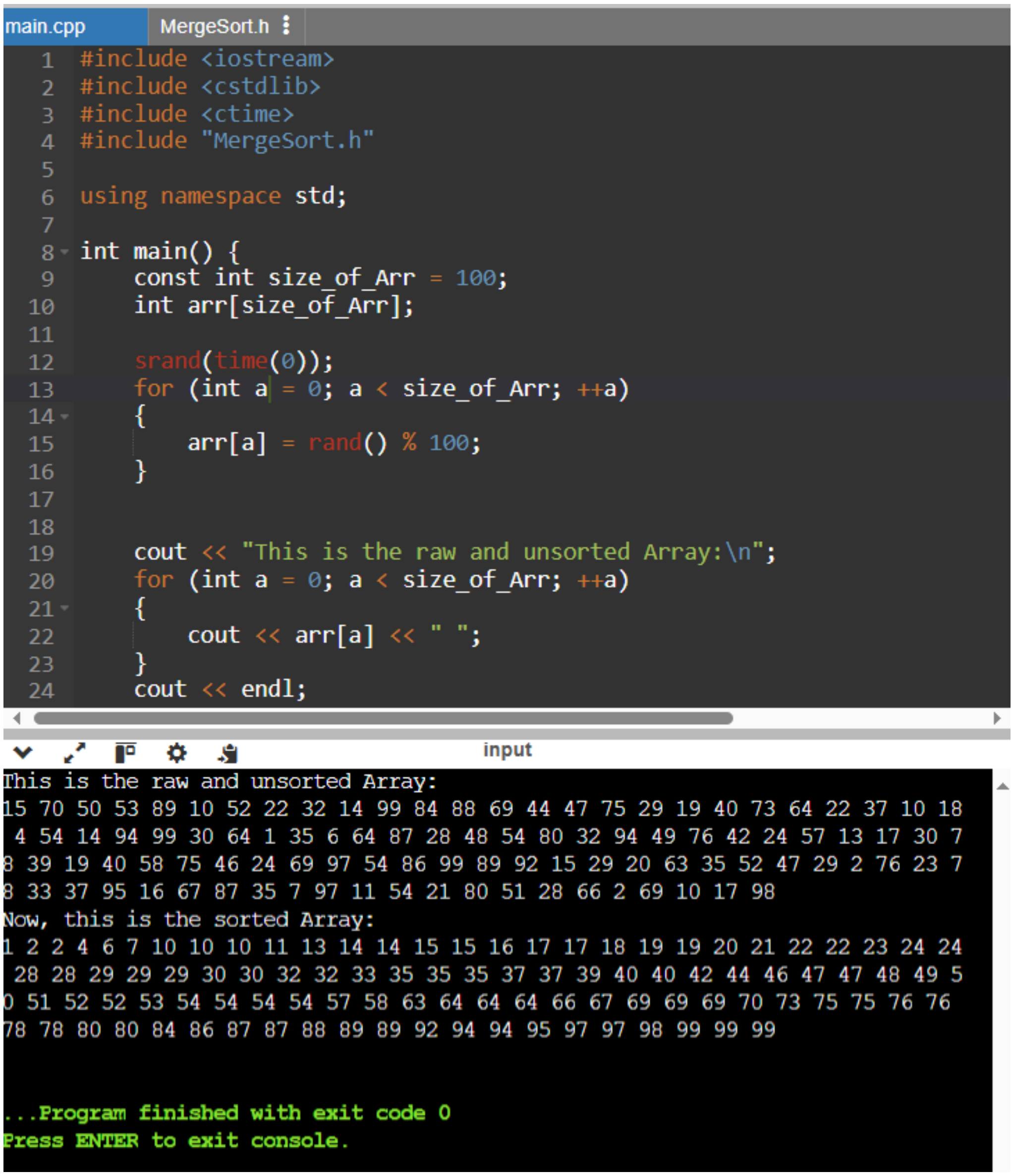
Observations	In Shell Sort, I first declare the unsorted or raw data of the array, which is why it displays the raw array before sorting it in ascending order. The Shell Sort function works by reducing the gap between elements to be compared, which improves the sorting efficiency by allowing elements to move closer to their positions and order more quickly.

Table 8-2. Shell Sort Technique

Code + Console Screenshot

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "MergeSort.h"

using namespace std;

int main()
{
   const int size_of_Arr = 100;
```

```
    int arr[size_of_Arr];

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


    cout << "This is the raw and unsorted Array:\n";
    for (int a = 0; a < size_of_Arr; ++a)
    {
        cout << arr[a] << " ";
    }
    cout << endl;

    merge_sort(arr, 0, size_of_Arr - 1);

    cout << "Now, this is the sorted Array:\n";
    for (int a = 0; a < size_of_Arr; ++a)
    {
        cout << arr[a] << " ";
    }
    cout << endl;

    return 0;
}


#ifndef MERGESORT_H
#define MERGESORT_H
#include <iostream>
using namespace std;

void merge(int arr[], int lft, int mid, int rght)
{
    int n1 = mid - lft + 1;
    int n2 = rght - mid;
    int* L = new int[n1];
    int* R = new int[n2];

    for (int a = 0; a < n1; a++)
        L[a] = arr[lft + a];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int a = 0;
    int j = 0;
    int k = lft;

    while (a < n1 && j < n2)
    {

        if (L[a] <= R[j])
        {
            arr[k] = L[a];
            a++;
        }

        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

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

    delete[] L;
    delete[] R;
}

void merge_sort(int arr[], int lft, int rght)
{
    if (lft < rght)
    {
        int mid = lft + (rght - lft) / 2;
        merge_sort(arr, lft, mid);
        merge_sort(arr, mid + 1, rght);
        merge(arr, lft, mid, rght);
    }
}

#endif
```

	
Observations	In Merge Sort, I first declare the unsorted or raw data of the array, which is why it displays the raw array before sorting it in ascending order. The Merge Sort function recursively divides the array into smaller subarrays or sub smaller parts, then sorts them, and then merges them back together, ensuring a time complexity of O(N log N) for efficient sorting and faster time performance.

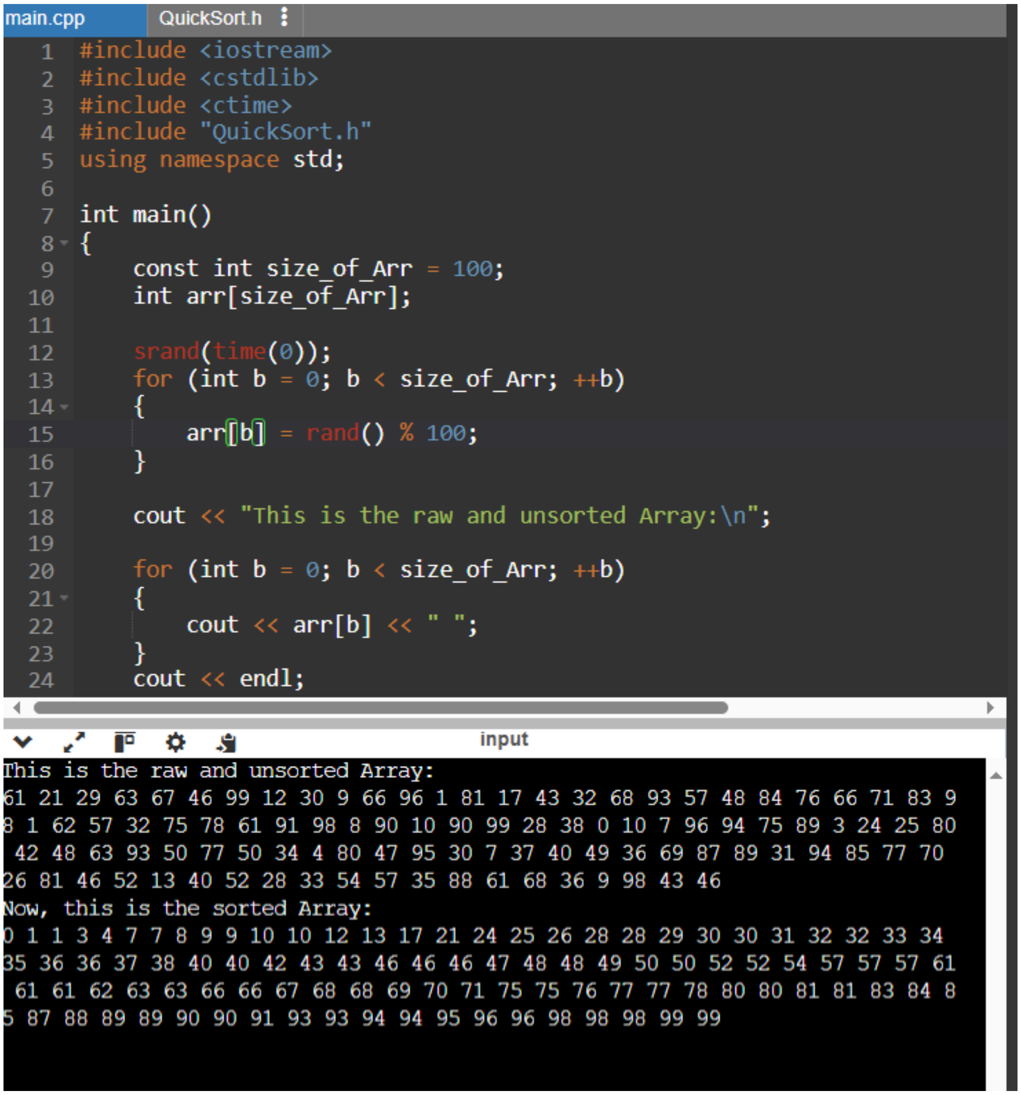
```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "MergeSort.h"

using namespace std;

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

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


    cout << "This is the raw and unsorted Array:\n";
    for (int a = 0; a < size_of_Arr; ++a)
    {
        cout << arr[a] << " ";
    }
    cout << endl;
```

```
This is the raw and unsorted Array:
15 70 50 53 89 10 52 22 32 14 99 84 88 69 44 47 75 29 19 40 73 64 22 37 10 18
 4 54 14 94 99 30 64 1 35 6 64 87 28 48 54 80 32 94 49 76 42 24 57 13 17 30 7
8 39 19 40 58 75 46 24 69 97 54 86 99 89 92 15 29 20 63 35 52 47 29 2 76 23 7
8 33 37 95 16 67 87 35 7 97 11 54 21 80 51 28 66 2 69 10 17 98
Now, this is the sorted Array:
1 2 2 4 6 7 10 10 10 11 13 14 14 15 15 16 17 17 18 19 19 20 21 22 22 23 24 24
 28 28 29 29 29 30 30 32 32 33 35 35 35 37 37 39 40 40 42 44 46 47 47 48 49 5
0 51 52 52 53 54 54 54 54 57 58 63 64 64 64 66 67 69 69 69 70 73 75 75 76 76
78 78 80 80 84 86 87 87 88 89 89 92 94 94 95 97 97 98 99 99 99


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

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot	#include <iostream>
	#include <cstdlib>
	#include <ctime>
	#include "QuickSort.h"
	using namespace std;

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

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

    cout << "This is the raw and unsorted Array:\n";

    for (int b = 0; b < size_of_Arr; ++b)
    {
        cout << arr[b] << " ";
    }
    cout << endl;

    quicksort(arr, 0, size_of_Arr - 1);

    cout << "Now, this is the sorted Array:\n";

    for (int b = 0; b < size_of_Arr; ++b)
    {
        cout << arr[b] << " ";
    }
    cout << endl;
    return 0;
}


#ifndef QUICKSORT_H
#define QUICKSORT_H
#include <iostream>
using namespace std;

int pattn(int arr[], int lw, int hgh)
{
    int pvt = arr[hgh];
    int i = (lw - 1);

    for (int c = lw; c < hgh; c++)
    {
        if (arr[c] <= pvt)
        {
            i++;
            swap(arr[i], arr[c]);
        }
    }
    swap(arr[i + 1], arr[hgh]);
    return (i + 1);
}

void quicksort(int arr[], int lw, int hgh)
{
    if (lw < hgh)
    {
        int pvt = pattn(arr, lw, hgh);
        quicksort(arr, lw, pvt - 1);
        quicksort(arr, pvt + 1, hgh);
    }
}

#endif
```



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

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

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

    cout << "This is the raw and unsorted Array:\n";

    for (int b = 0; b < size_of_Arr; ++b)
    {
        cout << arr[b] << " ";
    }
    cout << endl;
```

```
This is the raw and unsorted Array:
61 21 29 63 67 46 99 12 30 9 66 96 1 81 17 43 32 68 93 57 48 84 76 66 71 83 9
8 1 62 57 32 75 78 61 91 98 8 90 10 90 99 28 38 0 10 7 96 94 75 89 3 24 25 80
 42 48 63 93 50 77 50 34 4 80 47 95 30 7 37 40 49 36 69 87 89 31 94 85 77 70
26 81 46 52 13 40 52 28 33 54 57 35 88 61 68 36 9 98 43 46
Now, this is the sorted Array:
0 1 1 3 4 7 7 8 9 9 10 10 12 13 17 21 24 25 26 28 28 29 30 30 31 32 32 33 34
35 36 36 37 38 40 40 42 43 43 46 46 46 47 48 48 49 50 50 52 52 54 57 57 57 61
 61 61 62 63 63 66 66 67 68 68 69 70 71 75 75 76 77 77 78 80 80 81 81 83 84 8
5 87 88 89 89 90 90 91 93 93 94 94 95 96 96 98 98 98 99 99
```

Observations	In Quick Sort, I first declare the unsorted or raw data of the array, which is why it displays the raw array before sorting it in ascending order. The Quick Sort function uses a pivot to partition the array into subarrays, ensuring that elements less than the pivot come before it and elements greater than the pivot come after.

Table 8-4. Quick Sort Algorithm

7. Supplementary Activity

Problem 1: Can we sort the left sub list and right sub list from the partition method in quick sort using other sorting algorithms? Demonstrate an example.

// We will be using this example for the rest of the sorting algorithms so that we do not get confused. :
[7, 5, 9, 6, 3, 10]

Example 1: Quick Sort (Insertion Sort)

1. **Initial List**: [7, 5, 9, 6, 3, 10]
    a. Pivot : 7
    b. The pivot is being chosen on the **first, last or random number** in the list.
2. **Partition**: We divide the list since we choose the 7 as pivot, we create the left and right sublist.
    a. **Left sublist:** [5, 6, 3]
    b. **Right sublist:** [9, 10]
3. **Sorting:**
    a. In insertion method, we check with the **first number** then compare or look for the **next number**, if the number is **higher than** the first number, then put in a correct place, in **ascending order**, like this
    b. Compare 5 with 6 then place 5 before 6 → [5, 6].
    c. Compare 3 with 5 and 6 then move 3 to the front → [3, 5, 6].
4. **Final of Left Sublist**: [3, 5, 6].
5. **Final of Right Sublist**: [9, 10] *// (It is already sorted).*
6. **Final List: [3, 5, 6, 7, 9, 10].**

That is how Insertion Sort with Quick Sort runs in the program.

Example 2: Quick Sort (Merge Sort)

1. **Initial List**: [7, 5, 9, 6, 3, 10]
    a. Pivot : 7
    b. The pivot is being chosen on the **first, last or random number** in the list.
2. **Partition**: We divide the list since we choose the 7 as pivot, we create the left and right
    Left sublist: [7, 5, 9]
    Right sublist: [6, 3, 10]
3. **Merge Sort** works by **splitting the list** into **smaller sublists** until each sublist contains one element then we will merge them back, that is why it is merge sort in sorted order.
4. **Sorting:** We continue dividing:
    a. **Left sublist:** [7] and [5, 9] → further divide [5, 9] into [5] and [9].

        - Now we merge [5] and [9] into a sorted list → [5, 9]
        - Next, we merge [7] and [5, 9] into a sorted list → **[5, 7, 9]**
    b. **Right sublist:** Merge it to [6] and [3, 10] → further divide [3, 10] into [3] and [10].
        - Merge [3] and [10] → [3, 10].
        - Finally, merge [6] and [3, 10] → **[3, 6, 10].**
5. Now we merge the two sorted halves: **[5, 7, 9]** and **[3, 6, 10] → [3, 5, 6, 7, 9, 10].**
6. **Final List: [3, 5, 6, 7, 9, 10].**

That is how Merge Sort runs in the program.

Example 3: Quick Sort (Shell Sort)

1. **Initial List:** [7, 5, 9, 6, 3, 10]
    - Pivot: 7
    - the pivot is being chosen on the first, last, or random number in the list.
2. **Partition:** We divide the list since we chose 7 as the pivot, creating the left and right sublists.
    - **Left sublist:** [5, 6, 3]
    - **Right sublist:** [9, 10]
3. **Sorting:** Using Shell Sort, we **sort elements** that are a **specific gap apart.**
4. First gap of 3 (elements at positions 0 and 3): Compare 7 and 6.
    - **Swap them → [6, 5, 9, 7, 3, 10].**
    Next gap of 1 (standard insertion sort):
    - Compare 6 and 5. Swap → [5, 6, 9, 7, 3, 10].
    - Compare 9 and 7. Swap → [5, 6, 7, 9, 3, 10].
    - Compare 9 and 3. Swap → [5, 6, 7, 3, 9, 10].
    - Compare 7 and 3. Swap → [5, 6, 3, 7, 9, 10].
    - Compare 6 and 3. Swap → [5, 3, 6, 7, 9, 10].
    - Compare 5 and 3. Swap → [3, 5, 6, 7, 9, 10].
5. **Final Left Sublist: [3, 5, 6, 7].**
    Final Right Sublist: **[9, 10]** // (It is already sorted).
6. **Final List: [3, 5, 6, 7, 9, 10].**
    That is how Shell Sort with Quick Sort runs in the program.

Example 4: Quick Sort (Bubble Sort)

1. **Initial List:** [7, 5, 9, 6, 3, 10]
    - Pivot: 7
    - The pivot is being chosen on the first, last, or random number in the list.
2. **Partition:** We divide the list since we chose 7 as the pivot, creating the left and right sublists.
    - Left sublist: [5, 6, 3]
    - Right sublist: [9, 10]
3. **Sorting:** Using Bubble Sort, we **repeatedly** compare **adjacent numbers** and swap them if they're out of order.
4. Compare 5 and 6. They are already in order, so no swap is needed.
    Compare 6 and 3. Since 3 is smaller, we swap them → [5, 3, 6].
    Start over. Compare 5 and 3. Swap them → [3, 5, 6].
5. **Final Left Sublist:** [3, 5, 6].
    Final Right Sublist: [9, 10] // (It is already sorted).

6. **Final List:** [3, 5, 6, 7, 9, 10]
   That is how Bubble Sort with Quick Sort runs in the program.

Example 5: Quick Sort (Selection Sort)

1. **Initial List:** [7, 5, 9, 6, 3, 10]
   - Pivot: 7
   - The pivot is being chosen on the first, last, or random number in the list.
2. Partition: We divide the list since we chose 7 as the pivot, creating the left and right sublists.
   - **Left sublist:** [5, 6, 3]
   - **Right sublist:** [9, 10]
3. **Sorting:** Using Selection Sort, we repeatedly select the minimum element from the unsorted sublist and swap it with the first unsorted element.
4. For the left sublist [5, 6, 3]:
   - The minimum is 3. Swap it with 5 → [3, 6, 5].
   - Now the minimum is 5. No swap needed → [3, 5, 6].
5. **Final Left Sublist:** [3, 5, 6].
   Final Right Sublist: [9, 10] // (It is already sorted).
6. **Final List:** [3, 5, 6, 7, 9, 10].
   That is how Selection Sort with Quick Sort runs in the program.

Problem 2: Suppose we have an array which consists of {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19,74}. What sorting algorithm will give you the fastest time performance? Why can merge sort and quicksort have O(N • log N) for their time complexity?

The **fastest time performance** is the merge sort base from the example that I gave above we can that wherein the **merge sort** divides it into smaller parts and sorts them and merges them which has the consistency of the **O(N • log N) for their time complexity.**

Here is an example :

Merge Sort :

1. **Split the array:** {4, 34, 29, 48, 53} and {87, 12, 30, 44, 25, 93, 67, 43, 19, 74}
   - Left and right sublists.
2. Continue splitting until you have single elements. **// Just like the examples there above**
3. **Merge** it back while sorting which shows **faster time performance.**
4. **In left sublist:**
   - Merge {87} and {34} to get {4, 34}.
   - Merge {4, 34} and {29} to get {4, 29, 34}
5. **In right Sulist:**.
   - Merge {87} and {12} to get 12, 87}.
   - Merge {12, 87} and {30} to get {12, 30, 87} and so on.
   - We must continue to sort it until we finally need to fully merged the array:

Now, this is the output : {4, 12, 19, 25, 29, 30, 34, 43, 44, 48, 53, 67, 74, 87, 93}

8. Conclusion

In this activity, I learned how to use Shell, Merge, and Quick Sort algorithms and implement them in C++. I practiced the steps of each sorting method, which helped me understand how they work and their differences. Solving Supplementary activity shows the importance of choosing the right sorting algorithm based on the dataset characteristics and performance. I did good,, but I will improve my skills by learning and striving more on optimizing sorting algorithms. This laboratory activity has motivated me to continue exploring more different techniques.

9. Assessment Rubric