

ACTIVITY NO. 8

SORTING ALGORITHMS: SHELL, MERGE, AND QUICK SORT	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed: 09/27/25
Section: CPE21S4	Date Submitted: 09/27/25
Name: Cruz, Axl Waltz E.	Instructor: Engr. Jimlord Quejado
1. Objective(s)	
Create C++ code to sort through a dataset using additional sorting algorithms	
2. Intended Learning Outcomes (ILOs)	
After this activity, the student should be able to:	
<ul style="list-style-type: none">• Create C++ code for implementation of shell, merge and quick sort.• Solve given data sorting problems using appropriate sorting algorithms	
3. Discussion	

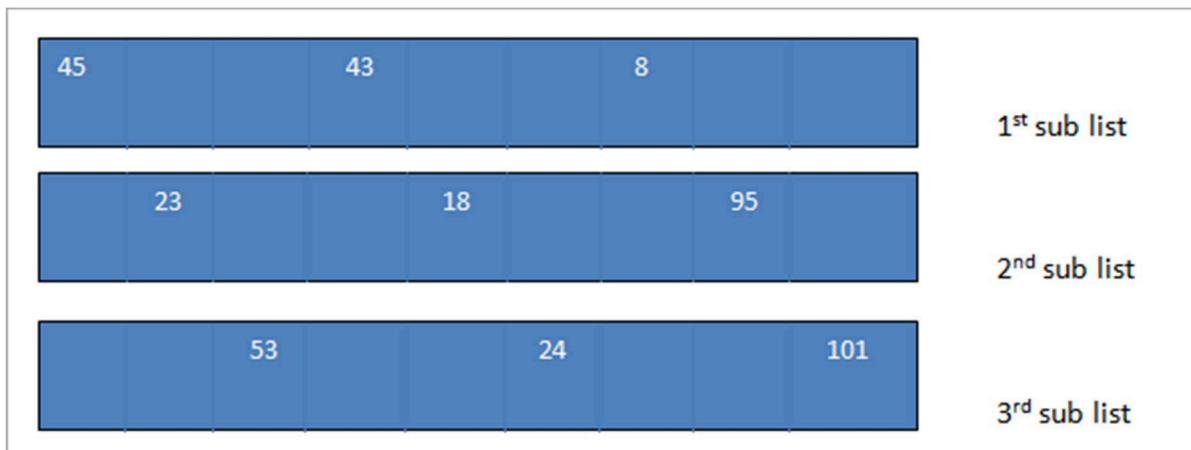
PART A: Shell Sort

Shell sort is often termed as an improvement over insertion sort. In insertion sort, we take increments by 1 to compare elements and put them in their proper position. In shell sort, the list is sorted by breaking it down into several smaller sub lists. It's not necessary that the lists need to be with contiguous elements. Instead, shell sort technique uses increment "i", which is also called "gap" and uses it to create a list of elements that are "i" elements apart.

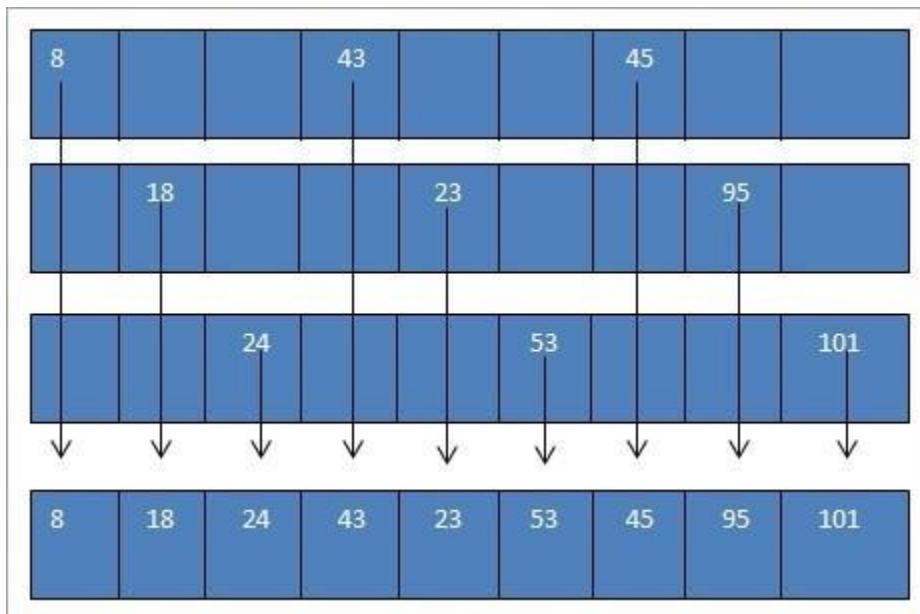
Consider the following array of 10 elements.

45	23	53	43	18	24	8	95	101
----	----	----	----	----	----	---	----	-----

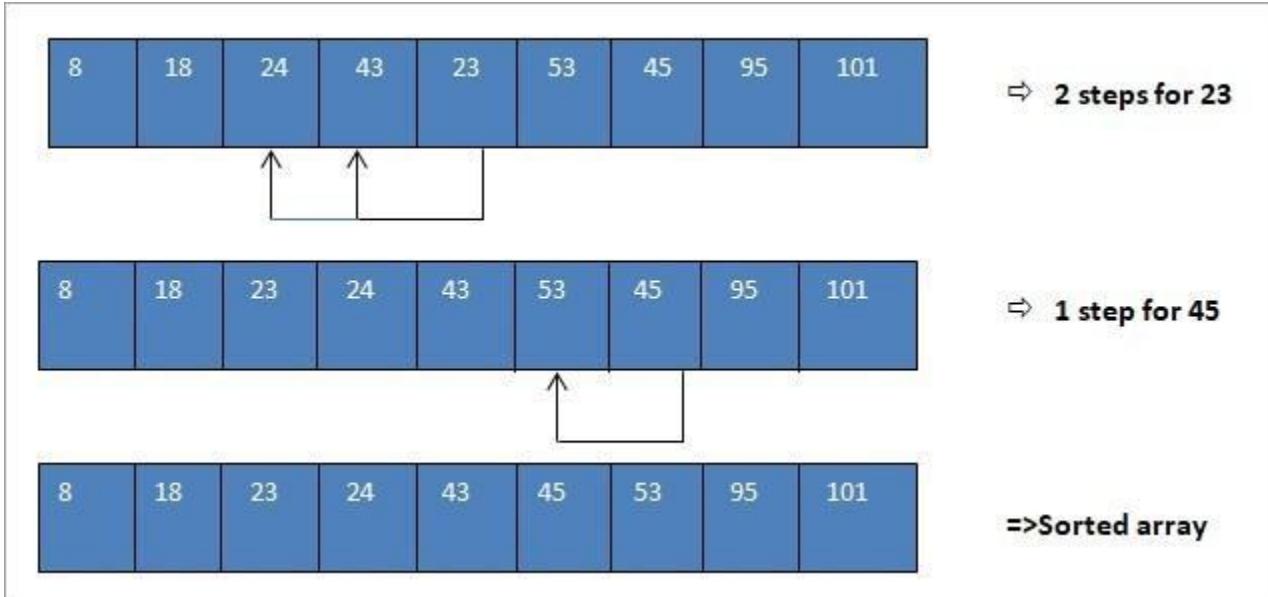
If we provide a gap of 3, then we will have the following sub-lists with each element that is 3 elements apart. We then sort these three sublists.



The sorted sub-lists and the resultant list that we obtain after combining the three sorted sublists are shown below.



The above array that we have obtained after merging the sorted subarrays is nearly sorted. Now we can perform insertion sort on this list and sort the entire array. This final step is shown below for your reference.



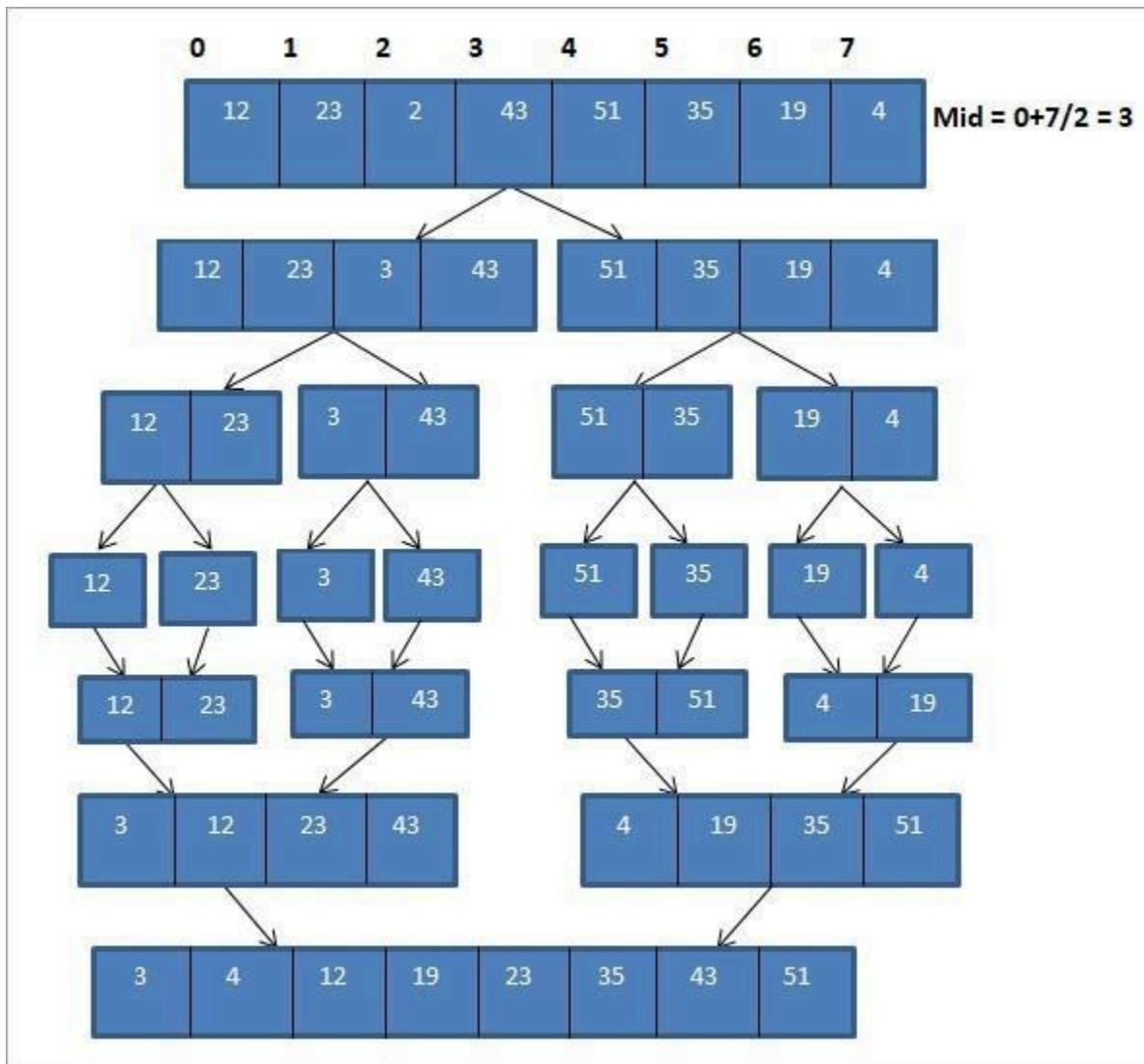
As seen above, after performing shell sort and merging the sorted sub lists, we only required three moves to completely sort the list. Thus, we can see that we can significantly reduce the number of steps required to sort the array. The choice of increment to create sub-lists is a unique feature of shell sort.

PART B: Merge Sort

Merge sort algorithm uses the “divide and conquer” strategy wherein we divide the problem into subproblems and solve those subproblems individually. These subproblems are then combined or merged to form a unified solution.

Merge sort is performed using the following steps:

1. The list to be sorted is divided into two arrays of equal length by dividing the list on the middle element. If the number of elements in the list is either 0 or 1, then the list is considered sorted.
2. Each sub list is sorted individually by using merge sort recursively.
3. The sorted sub lists are then combined or merged to form a complete sorted list.



As shown in the above representation, first the array is divided into two sub-arrays of length 4. Each sub-array is further divided into two more sub arrays of length 2. Each sub-array is then further divided into a sub-array of one element each. This entire process is the “Divide” process.

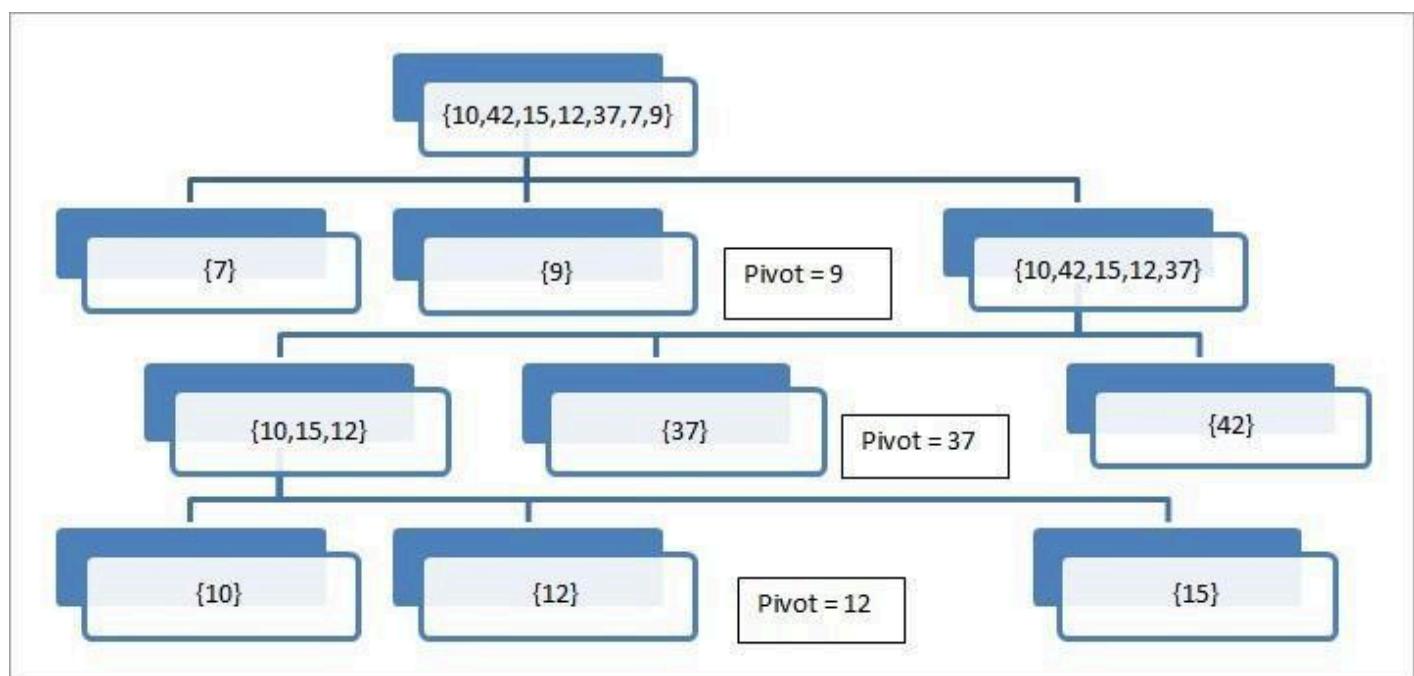
Once we have divided the array into sub-arrays of single element each, we now have to merge these arrays in sorted order.

As shown in the illustration above, we consider each subarray of a single element and first combine the elements to form sub-arrays of two elements in sorted order. Next, the sorted subarrays of length two are sorted and combined to form two sub-arrays of length four each. Then we combine these two sub-arrays to form a complete sorted array.

PART C: Quick Sort

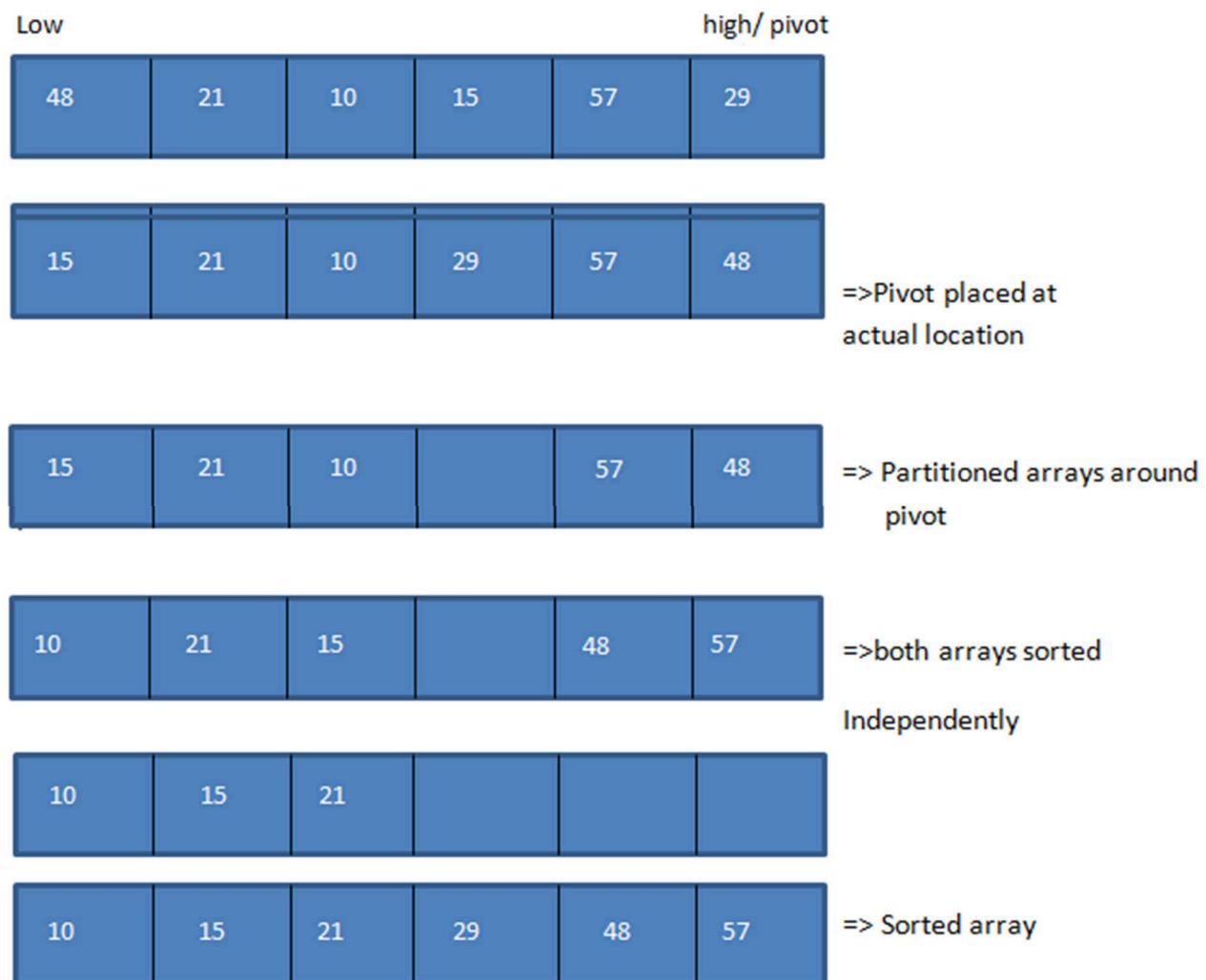
Quicksort is a widely used sorting algorithm which selects a specific element called “pivot” and partitions the array or list to be sorted into two parts based on this pivot s_0 that the elements lesser than the pivot are to the left of the list and the elements greater than the pivot are to the right of the list.

Thus, the list is partitioned into two sub lists. The sub lists may not be necessary for the same size. Then Quicksort calls itself recursively to sort these two sub lists.



In this illustration, we take the last element as pivot. We can see that the array is successively divided around the pivot element until we have a single element in the array.

Now we present an illustration of the Quicksort below to better understand the concept.



From the illustration, we can see that, we move the pointers high and low at both the ends of the array. Whenever low points to the element greater than the pivot and high points to the element lesser than the pivot, then we exchange the positions of these elements and advance the low and high pointers in their respective directions.

This is done until the low and high pointers cross each other. Once they cross each other the pivot element is placed at its proper position and the array is partitioned into two. Then both these sub-arrays are sorted independently using quicksort recursively.

4. Materials and Equipment

Personal Computer with C++ IDE

Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

5. Procedure

ILO A: Create C++ code for implementation of shell, merge and quick sort.

Preparation Task:

- Create an array of elements with random values. The array must contain 100 elements that are not sorted. Use the created array for each sorting algorithm below. Show the output of the preparation task on table 8-1 in section 6.
- Create a header file for the implementation of the different sorting algorithms.
- Import this header file into your main.cpp.

A.1. Shell Sort

General Algorithm

The general algorithm for shell sort is given below.

```
shellSort(array, size)
    for interval i <- size/2n down to 1
        for each interval "i" in array
            sort all the elements at interval "i"
    end shellSort
```

A.2. Merge Sort

General Algorithm

The general pseudo-code for the merge sort technique is given below.

```
Declare an array Arr of length N
If N=1, Arr is already sorted
If N>1,
Left = 0, right = N-1
Find middle = (left + right)/2
Call merge_sort(Arr, left, middle) =>sort first half recursively
Call merge_sort(Arr, middle+1, right) => sort second half recursively
Call merge(Arr, left, middle, right) to merge sorted arrays in above steps. Exit
```

A.3. Quick Sort

```
quicksort(A, low, high)
begin
    Declare array A[N] to be sorted
        low = 1st element; high = last element; pivot
    if(low < high)
        begin
            pivot = partition (A, low, high);
            quicksort (A, low, pivot-1)
            quicksort (A, pivot+1, high)

    End
```

6. Output

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

Code + Console Screenshot

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <ctime>
4 #include "sort.h"
5 using namespace std;
6
7 const int SIZE = 100;
8
9 void printArray(int arr[], int n) {
10    for (int i=0; i<n; i++) {
11        cout << arr[i] << " ";
12    }
13    cout << endl;
14 }
15
16 void copyArray(int src[], int dest[], int n) {
17    for (int i=0; i<n; i++) {
18        dest[i] = src[i];
19    }
20 }
21
22 int main() {
23    srand(time(0));
24
25    int arr[SIZE];
26    for (int i=0; i<SIZE; i++) {
27        arr[i] = rand() % 1000;
28    }
29
30    cout << "Original Array (unsorted):\n";
31    printArray(arr, SIZE);
32    cout << "-----\n";
33
34    // Shell Sort
35    int arrShell[SIZE];
36    copyArray(arr, arrShell, SIZE);
37    shellsort(arrShell, SIZE);
38    cout << "Array after Shell Sort:\n";
39    printArray(arrShell, SIZE);
40    cout << "-----\n";
41
42    // Merge Sort
43    int arrMerge[SIZE];
44    copyArray(arr, arrMerge, SIZE);
45    mergesort(arrMerge, 0, SIZE-1);
46    cout << "Array after Merge Sort:\n";
47
48    int main() {
49        srand(time(0));
50
51        int arr[SIZE];
52        for (int i=0; i<SIZE; i++) {
53            arr[i] = rand() % 1000;
54        }
55
56        cout << "Original Array (unsorted):\n";
57        printArray(arr, SIZE);
58        cout << "-----\n";
59
60        // Shell Sort
61        int arrShell[SIZE];
62        copyArray(arr, arrShell, SIZE);
63        shellsort(arrShell, SIZE);
64        cout << "Array after Shell Sort:\n";
65        printArray(arrShell, SIZE);
66        cout << "-----\n";
67
68        // Merge Sort
69        int arrMerge[SIZE];
70        copyArray(arr, arrMerge, SIZE);
71        mergesort(arrMerge, 0, SIZE-1);
72        cout << "Array after Merge Sort:\n";
73        printArray(arrMerge, SIZE);
74        cout << "-----\n";
75
76        // Quick Sort
77        int arrQuick[SIZE];
78        copyArray(arr, arrQuick, SIZE);
79        quicksort(arrQuick, 0, SIZE-1);
80        cout << "Array after Quick Sort:\n";
81        printArray(arrQuick, SIZE);
82        cout << "-----\n";
83
84    }
85
86    return 0;
87 }
```

```

1 #ifndef SORTS_H
2 #define SORTS_H
3
4 #include <iostream>
5 using namespace std;
6
7 // Shell Sort
8 void shellSort(int arr[], int n) {
9     for (int gap = n/2; gap > 0; gap /= 2) {
10        for (int i = gap; i < n; i++) {
11            int temp = arr[i];
12            int j = i;
13            while (j >= gap && arr[j-gap] > temp) {
14                arr[j] = arr[j-gap];
15                j -= gap;
16            }
17            arr[j] = temp;
18        }
19    }
20}
21
22 // Merge Sort
23 void merge(int arr[], int left, int mid, int right) {
24     int n1 = mid - left + 1;
25     int n2 = right - mid;
26
27     int *L = new int[n1];
28     int *R = new int[n2];
29
30     for (int i=0; i<n1; i++) L[i] = arr[left+i];
31     for (int j=0; j<n2; j++) R[j] = arr[mid+1+j];
32
33     int i=0, j=0, k=left;
34     while (i<n1 && j<n2) {
35         if (L[i] <= R[j]) arr[k++] = L[i++];
36         else arr[k++] = R[j++];
37     }
38     while (i<n1) arr[k++] = L[i++];
39     while (j<n2) arr[k++] = R[j++];
40
41     delete[] L;
42     delete[] R;
43 }
44
45 void mergeSort(int arr[], int left, int right) {
46     if (left < right) {
47         int mid = (left + right) / 2;
48         mergeSort(arr, left, mid);
49         mergeSort(arr, mid+1, right);
50         merge(arr, left, mid, right);
51     }
52 }
53 }

Original Array (unsorted):
128 411 939 954 467 718 722 64 398 985 145 573 327 655 401 135 639 64 333 400 393 60 590 626 581 773 344 982 481 643 160 731 255 743 247 992 888 558 612 839 988 984 971 405 738 281 803 649 45
131 518 333 542 546 556 573 585 593 612 649 667 703 706 722 731 738 741 743 751 771 882 826 828 836 839 853 863 879 888 890 895 982 994 985 921 924 938 939 941 967 971 974 982 988 992 994
Array after Shell Sort:
128 411 939 954 467 718 722 64 398 985 145 573 327 655 401 135 639 64 333 400 393 60 590 626 581 773 344 982 481 643 160 731 255 743 247 992 888 558 612 839 988 984 971 405 738 281 803 649
131 518 333 542 546 556 573 585 593 612 649 667 703 706 722 731 738 741 743 751 771 882 826 828 836 839 853 863 879 888 890 895 982 994 985 921 924 938 939 941 967 971 974 982 988 992 994
Array after Merge Sort:
128 411 939 954 467 718 722 64 398 985 145 573 327 655 401 135 639 64 333 400 393 60 590 626 581 773 344 982 481 643 160 731 255 743 247 992 888 558 612 839 988 984 971 405 738 281 803 649
131 518 333 542 546 556 573 585 593 612 649 667 703 706 722 731 738 741 743 751 771 882 826 828 836 839 853 863 879 888 890 895 982 994 985 921 924 938 939 941 967 971 974 982 988 992 994
Array after Quick Sort:
128 411 939 954 467 718 722 64 398 985 145 573 327 655 401 135 639 64 333 400 393 60 590 626 581 773 344 982 481 643 160 731 255 743 247 992 888 558 612 839 988 984 971 405 738 281 803 649
131 518 333 542 546 556 573 585 593 612 649 667 703 706 722 731 738 741 743 751 771 882 826 828 836 839 853 863 879 888 890 895 982 994 985 921 924 938 939 941 967 971 974 982 988 992 994

```

Original Array (unsorted):
128 411 939 954 467 718 722 64 398 985 145 573 327 655 401 135 639 64 333 400 393 60 590 626 581 773 344 982 481 643 160 731 255 743 247 992 888 558 612 839 988 984 971 405 738 281 803 649
131 518 333 542 546 556 573 585 593 612 649 667 703 706 722 731 738 741 743 751 771 882 826 828 836 839 853 863 879 888 890 895 982 994 985 921 924 938 939 941 967 971 974 982 988 992 994
Array after Shell Sort:
128 411 939 954 467 718 722 64 398 985 145 573 327 655 401 135 639 64 333 400 393 60 590 626 581 773 344 982 481 643 160 731 255 743 247 992 888 558 612 839 988 984 971 405 738 281 803 649
131 518 333 542 546 556 573 585 593 612 649 667 703 706 722 731 738 741 743 751 771 882 826 828 836 839 853 863 879 888 890 895 982 994 985 921 924 938 939 941 967 971 974 982 988 992 994
Array after Merge Sort:
128 411 939 954 467 718 722 64 398 985 145 573 327 655 401 135 639 64 333 400 393 60 590 626 581 773 344 982 481 643 160 731 255 743 247 992 888 558 612 839 988 984 971 405 738 281 803 649
131 518 333 542 546 556 573 585 593 612 649 667 703 706 722 731 738 741 743 751 771 882 826 828 836 839 853 863 879 888 890 895 982 994 985 921 924 938 939 941 967 971 974 982 988 992 994
Array after Quick Sort:
128 411 939 954 467 718 722 64 398 985 145 573 327 655 401 135 639 64 333 400 393 60 590 626 581 773 344 982 481 643 160 731 255 743 247 992 888 558 612 839 988 984 971 405 738 281 803 649
131 518 333 542 546 556 573 585 593 612 649 667 703 706 722 731 738 741 743 751 771 882 826 828 836 839 853 863 879 888 890 895 982 994 985 921 924 938 939 941 967 971 974 982 988 992 994

Process exited after 0.1626 seconds with return value 0
Press any key to continue . . .

Observations

- The three sorting algorithms—Shell Sort, Merge Sort, and Quick Sort—are illustrated in this code. 100 random numbers are generated by the main program, which then replicates the array so that each algorithm can sort the data evenly. By comparing the data items with a gap and reducing the gap until the list is sorted, the Shell Sort method operates. Using a divide and conquer strategy, Merge Sort divides the array into smaller components, sorts them, and then recombines them. Similar to divide and conquer, Quick Sort selects a pivot and rearranges elements around it before sorting the subarrays recursively.

Table 8-2. Shell Sort Technique

Code + Console Screenshot

```
1 #include "myshellsort.h"
2
3 int main() {
4     int data[] = {10, 55, 3, 98, 32, 17, 65};
5     int size = sizeof(data) / sizeof(data[0]);
6
7     cout << "Before Sorting:\n";
8     printArray(data, size);
9
10    shellSort(data, size);
11
12    cout << "After Shell Sort:\n";
13    printArray(data, size);
14
15    return 0;
16}
17
18 #ifndef MYSHELLSORT_H
19 #define MYSHELLSORT_H
20
21 #include <iostream>
22 #include <iomanip>
23 using namespace std;
24
25 void shellSort(int arr[], int n) {
26     int gap, i, j, temp;
27
28     for (gap = n / 2; gap > 0; gap /= 2) {
29         for (i = gap; i < n; i++) {
30             temp = arr[i];
31             j = i;
32
33             while (j >= gap && arr[j - gap] > temp) {
34                 arr[j] = arr[j - gap];
35                 j -= gap;
36             }
37             arr[j] = temp;
38         }
39     }
40 }
41
42 void printArray(int arr[], int n) {
43     cout << "\nPosition | Number\n";
44     cout << "-----\n";
45     for (int i = 0; i < n; i++) {
46         cout << setw(8) << i << " | " << setw(6) << arr[i] << endl;
47     }
48     cout << endl;
49 }
50
51 #endif
```

	<pre> Before Sorting: Position Number ----- 0 10 1 55 2 3 3 98 4 32 5 17 6 65 After Shell Sort: Position Number ----- 0 3 1 10 2 17 3 32 4 55 5 65 6 98 Process exited after 0.1017 seconds with return value 0 Press any key to continue . . </pre>
Observations	<ul style="list-style-type: none"> - The numbers are arranged in order by this program using shell sort. This begins by creating a gap in the list and gradually reduces it. Numbers get closer to their respective locations with each step. The numbers in the table are shown with the display array function. Finally, it prints the list both before and after sorting. -

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot	
---------------------------	--

```
1 #include "merge.h"
2
3 int main() {
4     int n;
5     cout << "Enter the number of elements: ";
6     cin >> n;
7
8     int arr[n];
9     cout << "Enter " << n << " elements: ";
10    for (int i = 0; i < n; i++)
11        cin >> arr[i];
12
13    cout << "Array before sorting: ";
14    display(arr, n);
15
16    mergeSort(arr, 0, n - 1);
17
18    cout << "Array after sorting: ";
19    display(arr, n);
20
21    return 0;
22}
23|
```

```

1 #ifndef MERGE_H
2 #define MERGE_H
3
4 #include <iostream>
5 using namespace std;
6
7 void combine(int a[], int low, int mid, int high) {
8     int n1 = mid - low + 1;
9     int n2 = high - mid;
10
11    int leftArr[n1], rightArr[n2];
12
13    for (int i = 0; i < n1; i++)
14        leftArr[i] = a[low + i];
15    for (int j = 0; j < n2; j++)
16        rightArr[j] = a[mid + 1 + j];
17
18    int i = 0, j = 0, k = low;
19    while (i < n1 && j < n2) {
20        if (leftArr[i] <= rightArr[j]) {
21            a[k] = leftArr[i];
22            i++;
23        } else {
24            a[k] = rightArr[j];
25            j++;
26        }
27        k++;
28    }
29
30    while (i < n1) {
31        a[k] = leftArr[i];
32        i++;
33        k++;
34    }
35    while (j < n2) {
36        a[k] = rightArr[j];
37        j++;
38        k++;
39    }
40 }
41
42 void mergeSort(int a[], int low, int high) {
43    if (low < high) {
44        int mid = (low + high) / 2;
45
46        mergeSort(a, low, mid);
47
48        mergeSort(a, mid + 1, high);
49
50        combine(a, low, mid, high);
51    }
52 }
53
54 void display(int a[], int n) {
55    for (int i = 0; i < n; i++)
56        cout << a[i] << " ";
57    cout << endl;
58 }
59
60 #endif
61

```

```

Enter the number of elements: 4
Enter 4 elements: 10 5 8 15
Array before sorting: 10 5 8 15
Array after sorting: 5 8 10 15
-----
```

```

Process exited after 15.73 seconds with return value 0
Press any key to continue . . .

```

Observations

- This program arranges numbers in ascending order using merge sort. First, the merge sort is used to split the array in half. The two sorted halves are then combined into a single sorted component by the combined function. The application displays the array before and after sorting.

once the user enters the size elements of the arrays. Because it adheres to the divide and conquer strategy, this approach is incredibly effective.

-

-

Table 8-4. Quick Sort Algorithm

Code + Console Screenshot

```
1 #include <iostream>
2 #include "qsort.h"
3 using namespace std;
4
5 int partition(int arr[], int left, int right) {
6     int pivot = arr[(left + right) / 2];
7     int i = left;
8     int j = right;
9
10    while (i <= j) {
11        while (arr[i] < pivot) i++;
12        while (arr[j] > pivot) j--;
13
14        if (i <= j) {
15            int temp = arr[i];
16            arr[i] = arr[j];
17            arr[j] = temp;
18            i++;
19            j--;
20        }
21    }
22    return i;
23}
24
25 void quickSort(int arr[], int left, int right) {
26     if (left < right) {
27         int index = partition(arr, left, right);
28
29         if (left < index - 1)
30             quickSort(arr, left, index - 1);
31
32         if (index < right)
33             quickSort(arr, index, right);
34     }
35}
36
37 int main() {
38     int n;
39     cout << "How many numbers? ";
40     cin >> n;
41
42     int arr[n];
43     cout << "Enter numbers: ";
44     for (int i = 0; i < n; i++) {
45         cin >> arr[i];
46     }
47
48     quickSort(arr, 0, n - 1);
49
50     cout << "Sorted: ";
51     for (int i = 0; i < n; i++) {
52         cout << arr[i] << " ";
53     }
54     cout << endl;
55
56     return 0;
57}
58
```

```

1  #ifndef QSORT_H
2  #define QSORT_H
3
4  void quickSort(int arr[], int left, int right);
5  int partition(int arr[], int left, int right);
6
7  #endif
8

```

```

How many numbers? 4
Enter numbers: 15 13 12 10
Sorted: 10 12 13 15
-----
Process exited after 14.1 seconds with return value 0
Press any key to continue . . .

```

Observations

- This application keeps the elements in an array after asking the user how many they wish to sort. In this scenario, the rapid sort operates by selecting a pivot for the final components and rearranging the array so that the larger numbers go to the right and the smaller ones go to the left. The partition function does this rearrangement by switching the pivot's elements.

7. Supplementary Activity

ILO B: Solve given data sorting problems using appropriate basic sorting algorithms

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.

- Using pivot elements, the arrays are separated into a left and a right sublist for the rapid sort. Other sorting techniques, such as bubble sort or insertion sort, can be applied to the left and right sections after division. For instance, if the pivot is 19, the array {25 15 18 19 20 11} yields {11 15 18} on the left and {20 25} on the right. Insertion can be used to sort the left side, while bubble sort can be used to sort the right. The final sorted array after combining them all is {11 15 18 19 10 15}.

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 quick sort have $O(N \cdot \log N)$ for their time complexity?

- Quick sort or merger sort are the fastest ways to sort the provided array. Compared to bubble sort, selection sort, or insertion sort, both of these algorithmic techniques are quicker. Their speed is derived from breaking the array up into smaller components and working through each one one at a time. Instead of N^2 , this procedure makes the number of steps approximately $N \log N$. For larger arrays, merge sort and rapid sort are therefore particularly helpful.

8. Conclusion

Provide the following:

- Summary of lessons learned
 - I learned that different sorting algorithms including Shell Sort, Merge Sort, and Quick Sort that can solve the same problem in different ways. Each has its strengths and weaknesses, but all aim to sort data correctly.
- Analysis of the procedure
 - Shell Sort was simple and efficient. Merge Sort used a divide-and-conquer approach that made sorting easier to manage. Quick Sort was flexible and could be written in many ways, always giving the right result.
- Analysis of the supplementary activity
 - Comparing the algorithms helped me see which one works best in different situations, especially with large arrays. It also showed that the choice of algorithm depends on the problem's size and complexity.
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?
 - I understood the concepts well and successfully applied each algorithm. However, I need to improve on writing the code more confidently and choosing the best algorithm for different data sizes.

9. Assessment Rubric