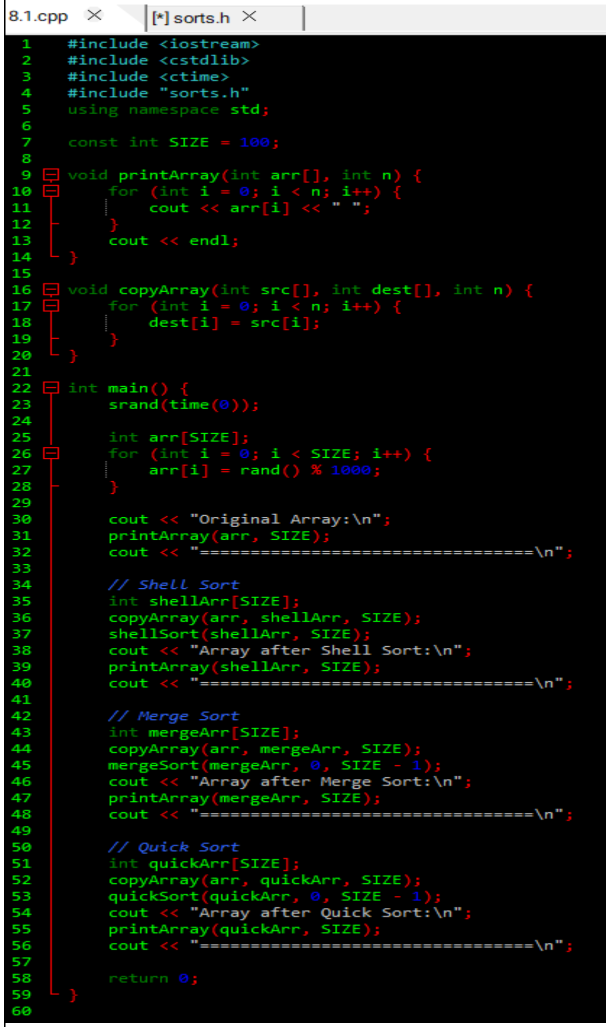
Activity No. <8>	
SORTING ALGORITHMS: SHELL, MERGE, AND QUICK SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 9/27/25**
Section: CPE21S4	**Date Submitted: 9/27/25**
Name(s): Alcantara, Jason P.	**Instructor: Engr. Jimlord Quejado**

6. Output:

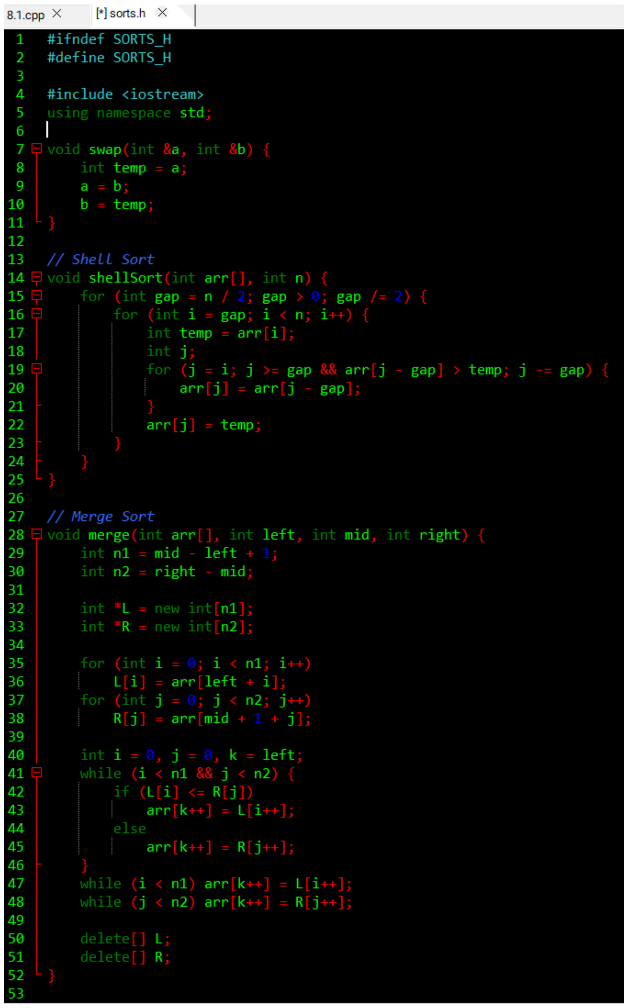
Table 8-1:
Main CPP:



8.1.cpp [*] sorts.h

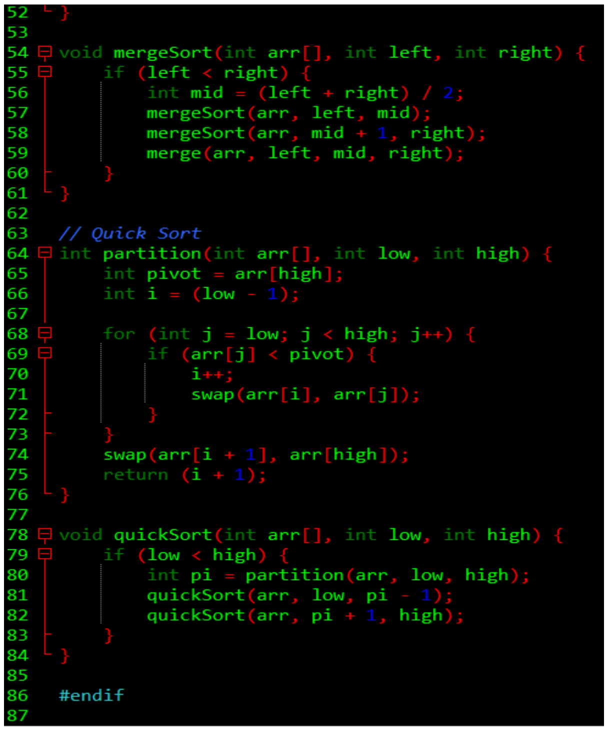
```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "sorts.h"
using namespace std;

const int SIZE = 100;

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

void copyArray(int src[], int dest[], int n) {
    for (int i = 0; i < n; i++) {
        dest[i] = src[i];
    }
}

int main() {
    srand(time(0));

    int arr[SIZE];
    for (int i = 0; i < SIZE; i++) {
        arr[i] = rand() % 1000;
    }

    cout << "Original Array:\n";
    printArray(arr, SIZE);
    cout << "=================================\n";

    // Shell Sort
    int shellArr[SIZE];
    copyArray(arr, shellArr, SIZE);
    shellSort(shellArr, SIZE);
    cout << "Array after Shell Sort:\n";
    printArray(shellArr, SIZE);
    cout << "=================================\n";

    // Merge Sort
    int mergeArr[SIZE];
    copyArray(arr, mergeArr, SIZE);
    mergeSort(mergeArr, 0, SIZE - 1);
    cout << "Array after Merge Sort:\n";
    printArray(mergeArr, SIZE);
    cout << "=================================\n";

    // Quick Sort
    int quickArr[SIZE];
    copyArray(arr, quickArr, SIZE);
    quickSort(quickArr, 0, SIZE - 1);
    cout << "Array after Quick Sort:\n";
    printArray(quickArr, SIZE);
    cout << "=================================\n";

    return 0;
}

```

Header File:



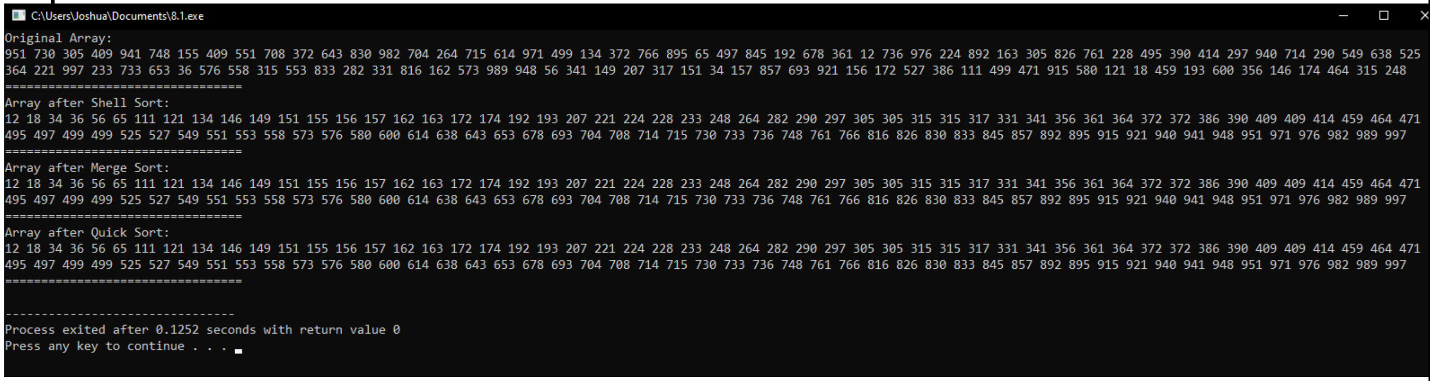
8.1.cpp × [*] sorts.h ×

```cpp
#ifndef SORTS_H
#define SORTS_H

#include <iostream>
using namespace std;

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

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

// Merge Sort
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    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[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    delete[] L;
    delete[] R;
}

```



```cpp
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// Quick Sort
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 pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

#endif

```

Output:



C:\Users\Joshua\Documents\8.1.exe

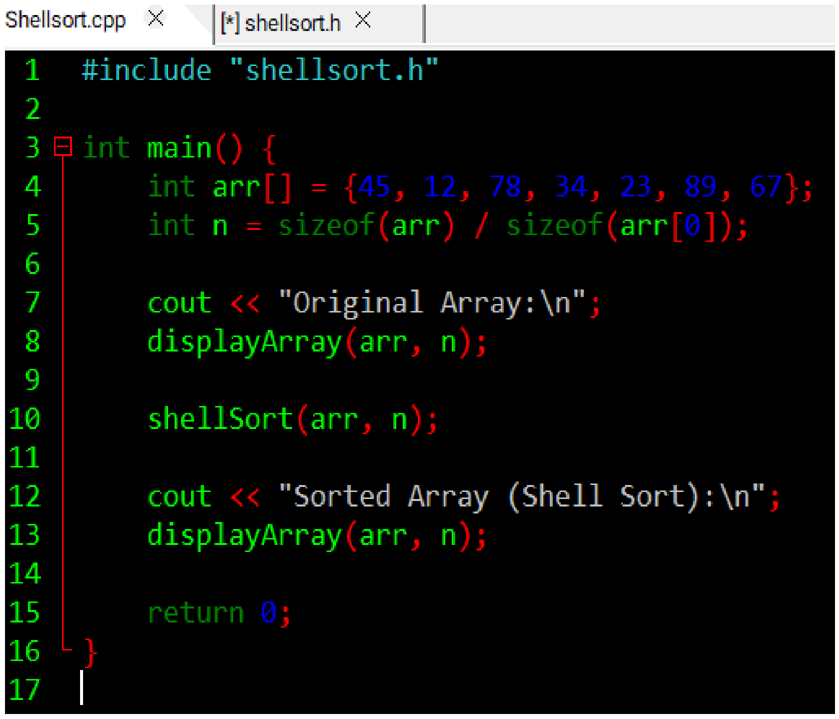
```
Original Array:
951 730 305 409 941 748 155 409 551 708 372 643 830 982 704 264 715 614 971 499 134 372 766 895 65 497 845 192 678 361 12 736 976 224 892 163 305 826 761 228 495 390 414 297 940 714 290 549 638 525
364 221 997 233 733 653 36 576 558 315 553 833 282 331 816 162 573 989 948 56 341 149 207 317 151 34 157 857 693 921 156 172 527 386 111 499 471 915 580 121 18 459 193 600 356 146 174 464 315 248
=================================
Array after Shell Sort:
12 18 34 36 56 65 111 121 134 146 149 151 155 156 157 162 163 172 174 192 193 207 221 224 228 233 248 264 282 290 297 305 305 315 315 317 331 341 356 361 364 372 372 386 390 409 409 414 459 464 471
495 497 499 499 525 527 549 551 553 558 573 576 580 600 614 638 643 653 678 693 704 708 714 715 730 733 736 748 761 766 816 826 830 833 845 857 892 895 915 921 940 941 948 951 971 976 982 989 997
=================================
Array after Merge Sort:
12 18 34 36 56 65 111 121 134 146 149 151 155 156 157 162 163 172 174 192 193 207 221 224 228 233 248 264 282 290 297 305 305 315 315 317 331 341 356 361 364 372 372 386 390 409 409 414 459 464 471
495 497 499 499 525 527 549 551 553 558 573 576 580 600 614 638 643 653 678 693 704 708 714 715 730 733 736 748 761 766 816 826 830 833 845 857 892 895 915 921 940 941 948 951 971 976 982 989 997
=================================
Array after Quick Sort:
12 18 34 36 56 65 111 121 134 146 149 151 155 156 157 162 163 172 174 192 193 207 221 224 228 233 248 264 282 290 297 305 305 315 315 317 331 341 356 361 364 372 372 386 390 409 409 414 459 464 471
495 497 499 499 525 527 549 551 553 558 573 576 580 600 614 638 643 653 678 693 704 708 714 715 730 733 736 748 761 766 816 826 830 833 845 857 892 895 915 921 940 941 948 951 971 976 982 989 997
=================================


--------------------------------
Process exited after 0.1252 seconds with return value 0
Press any key to continue . . .
```

Explanation:

In this code I demonstrate the Three sorting algorithms: Shell Sort, Merge Sort, and Quick Sort. The main program generates a 100 randoms numbers, then make a copies of the array so each algorithms can sort the data equally. The Shell Sort Works by comparing the data elements with a gap and decreasing the gap until the list is sorted. Merge Sort uses a divide and conquer approach by splitting the array into smaller parts, sorting them and merging back together. Quick Sort also uses a divide and conquer but chooses a pivot and rearranges elements around it before recursively sorting the subarrays.

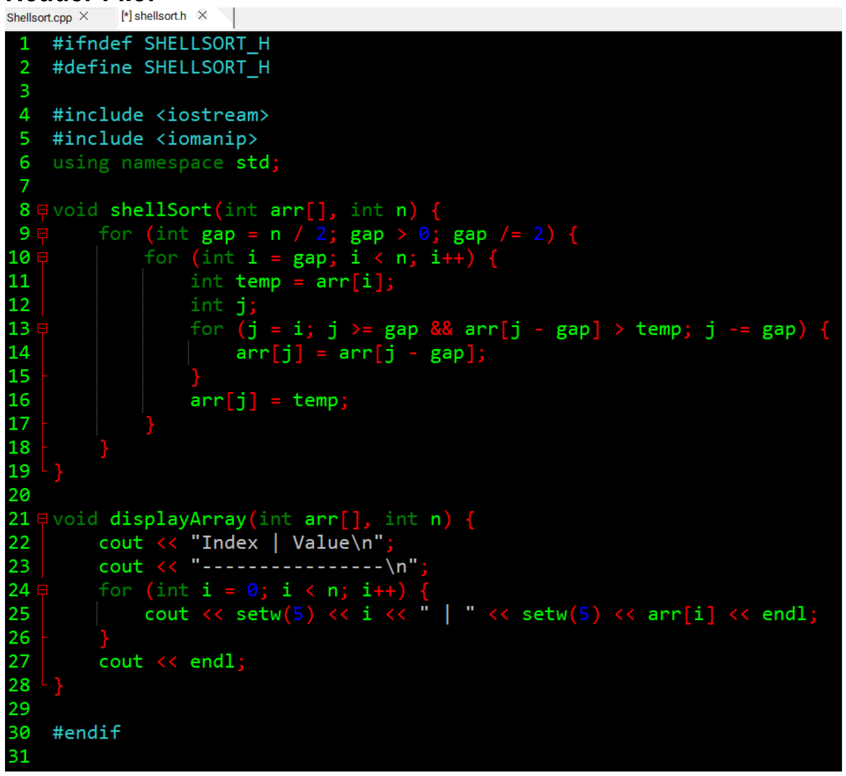
Table 8.2:

Main CPP:


Shellsort.cpp × | [*] shellsort.h ×

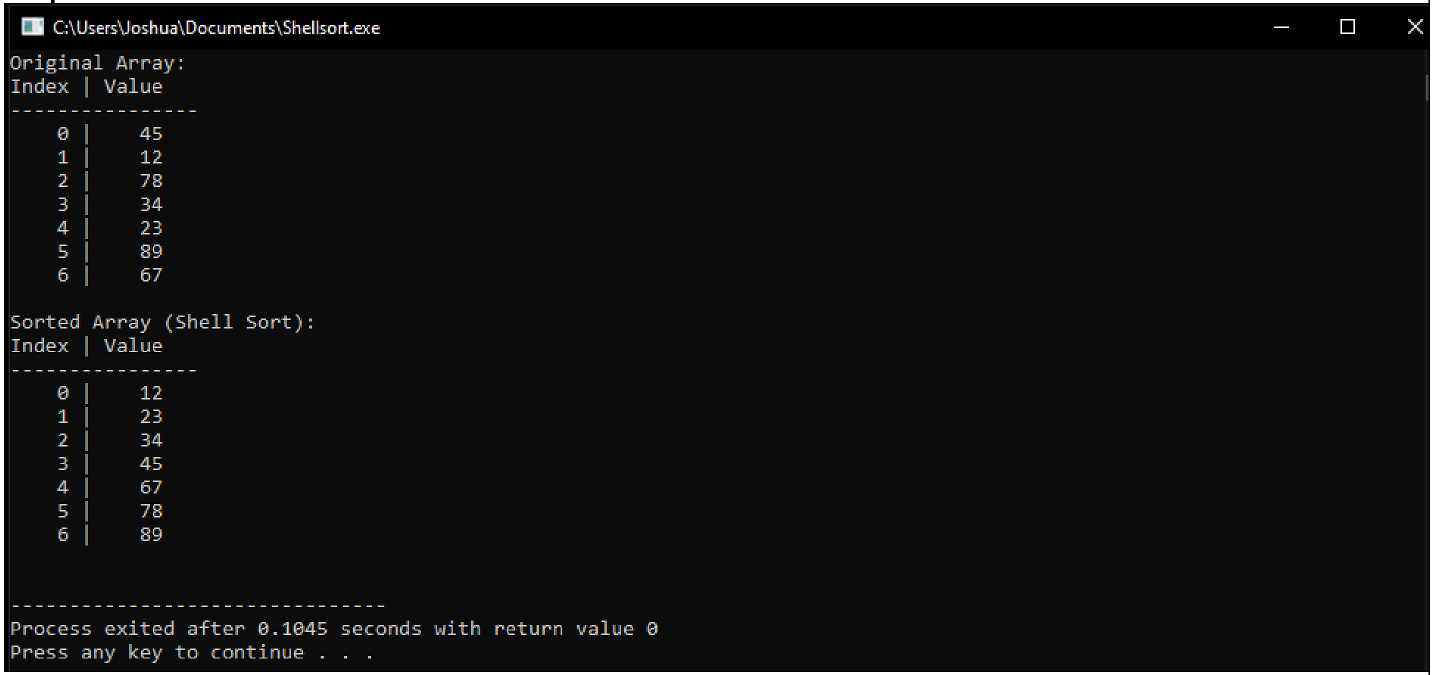
```cpp
#include "shellsort.h"

int main() {
    int arr[] = {45, 12, 78, 34, 23, 89, 67};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Original Array:\n";
    displayArray(arr, n);

    shellSort(arr, n);

    cout << "Sorted Array (Shell Sort):\n";
    displayArray(arr, n);

    return 0;
}

```

Header File:


Shellsort.cpp × | [*] shellsort.h ×

```cpp
#ifndef SHELLSORT_H
#define SHELLSORT_H

#include <iostream>
#include <iomanip>
using namespace std;

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

void displayArray(int arr[], int n) {
    cout << "Index | Value\n";
    cout << "----------------\n";
    for (int i = 0; i < n; i++) {
        cout << setw(5) << i << " | " << setw(5) << arr[i] << endl;
    }
    cout << endl;
}

#endif

```

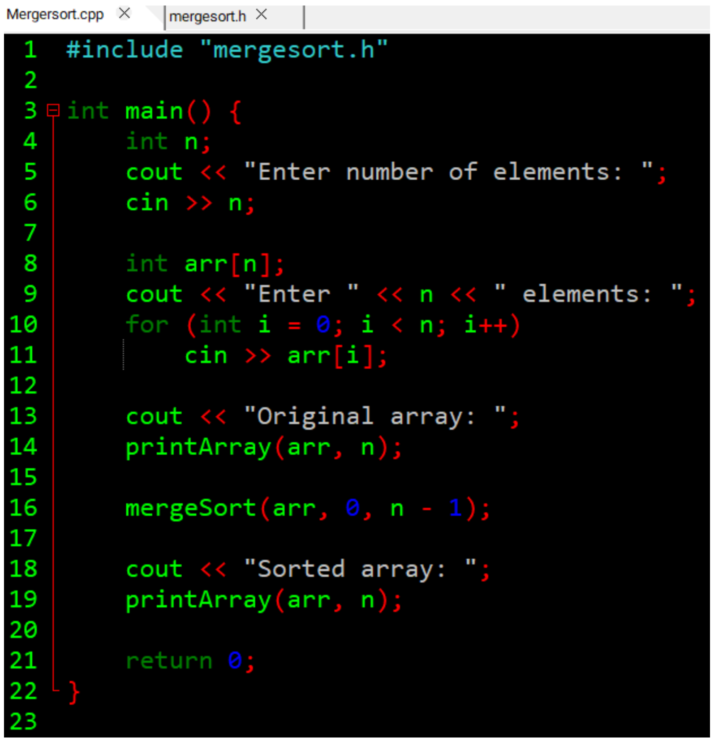
Output:



```
C:\Users\Joshua\Documents\Shellsort.exe
Original Array:
Index | Value
----------------
    0 |    45
    1 |    12
    2 |    78
    3 |    34
    4 |    23
    5 |    89
    6 |    67

Sorted Array (Shell Sort):
Index | Value
----------------
    0 |    12
    1 |    23
    2 |    34
    3 |    45
    4 |    67
    5 |    78
    6 |    89


--------------------------------
Process exited after 0.1045 seconds with return value 0
Press any key to continue . . .
```

Explanation:
This program uses shell sort to arrange the numbers in order. This starts by dividing the list with gap and slowly makes the gap smaller. Each step moves numbers closer to their corresponding spot. The display Array function appear the numbers in the table. And in the end, it prints the list before and after the of sorting.

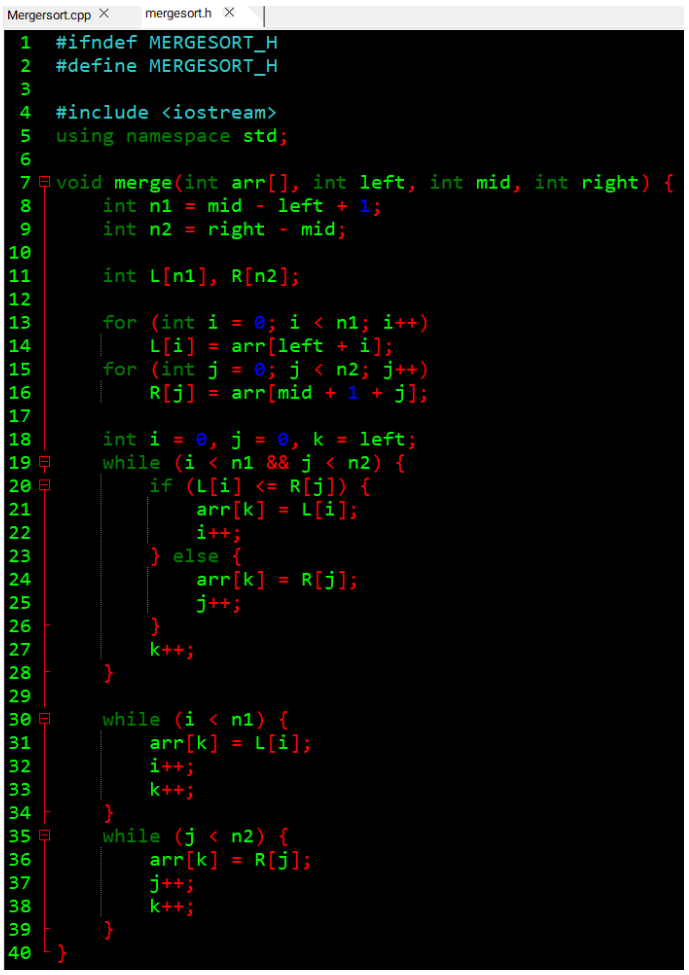
Table 8.3:
Main CPP:



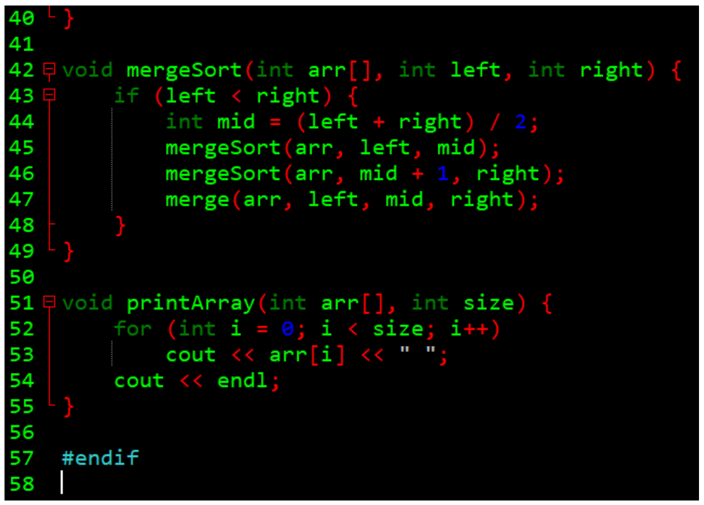
Mergersort.cpp | mergesort.h

```cpp
#include "mergesort.h"

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    int arr[n];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    cout << "Original array: ";
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}

```

Header File:

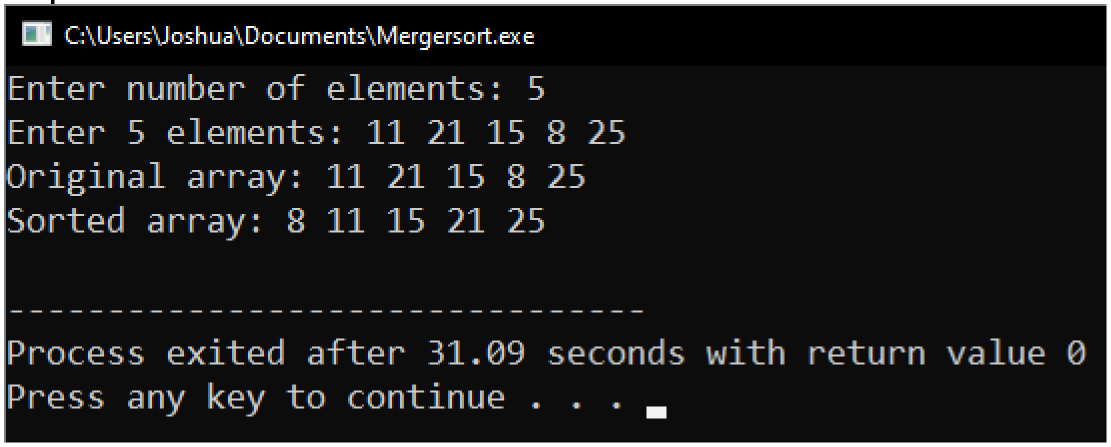


```cpp
#ifndef MERGESORT_H
#define MERGESORT_H

#include <iostream>
using namespace std;

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

    int L[n1], R[n2];

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

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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



```cpp
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

#endif

```

Output:

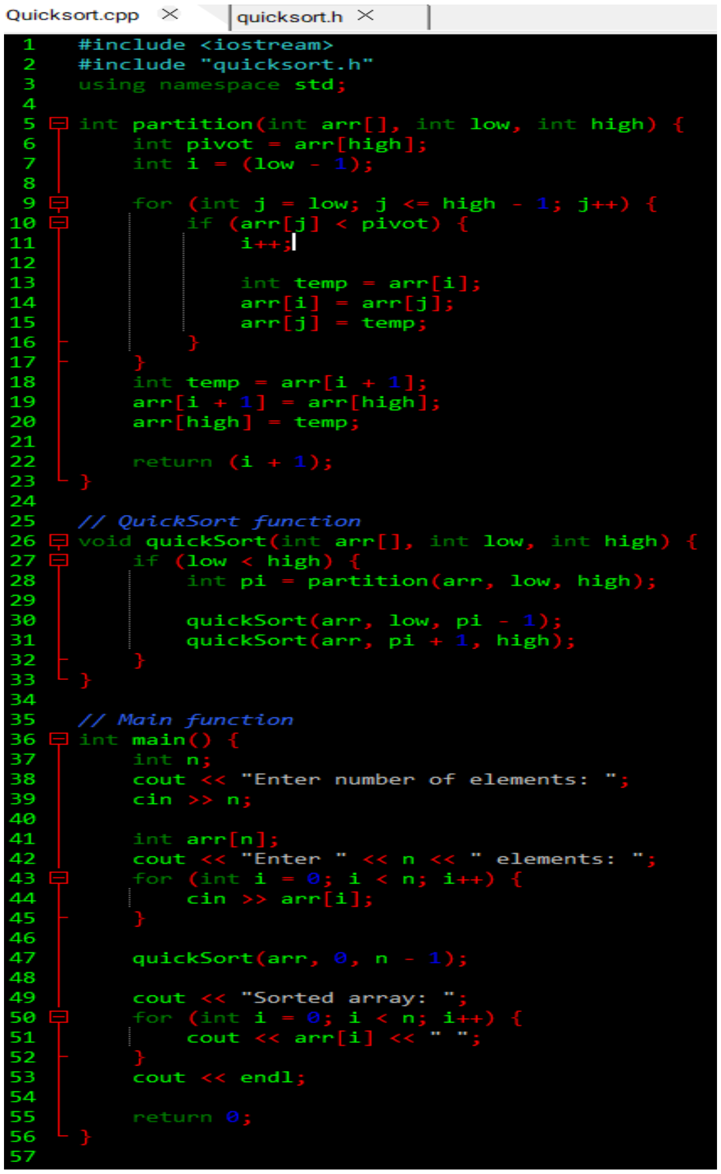


```
Enter number of elements: 5
Enter 5 elements: 11 21 15 8 25
Original array: 11 21 15 8 25
Sorted array: 8 11 15 21 25


--------------------------------
Process exited after 31.09 seconds with return value 0
Press any key to continue . . .
```

Explanation:
This program uses merge sort to arrange number in qascending orders. First, the array is divided into smaller halves using the merge sort. The combined function then merges the two sorted halves into one sorted part. The user enter the size elements of the arrats and the program shows the array before and the after of sorting. This method is very efficient because it follows the divide and conquer principles.

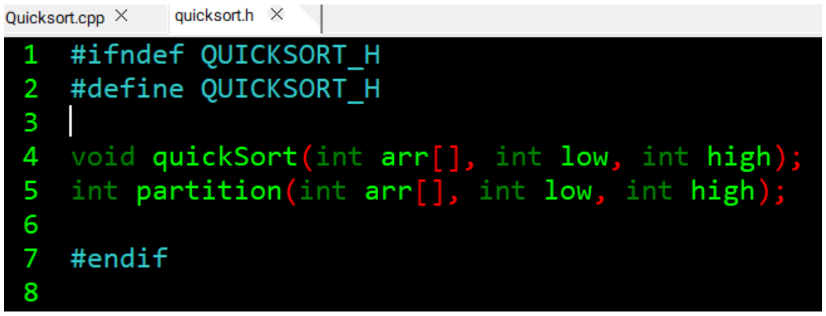
Table 8.4

Main CPP:



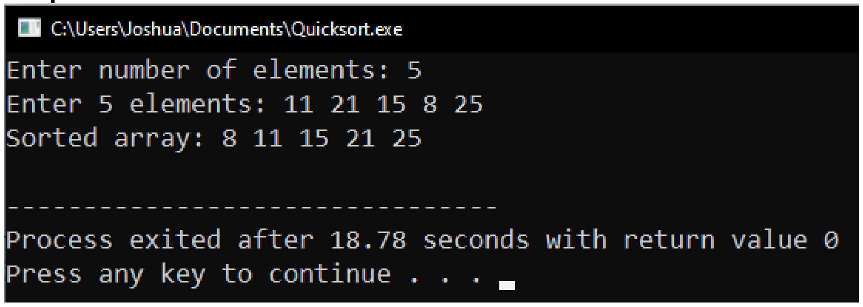
```cpp
#include <iostream>
#include "quicksort.h"
using namespace std;

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

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;

            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}

// QuickSort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Main function
int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    int arr[n];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```

Header File:



```cpp
#ifndef QUICKSORT_H
#define QUICKSORT_H

void quickSort(int arr[], int low, int high);
int partition(int arr[], int low, int high);

#endif
```

Output:



```
C:\Users\Joshua\Documents\Quicksort.exe
Enter number of elements: 5
Enter 5 elements: 11 21 15 8 25
Sorted array: 8 11 15 21 25

--------------------------------
Process exited after 18.78 seconds with return value 0
Press any key to continue . . .
```

Explanation:
This program ask the user to input how many elements they want to sort and stores them in an array. The quick sort works by choosing a pivot the last elements in this case and rearranging the array so the smaller number go to the left and the bigger number go to the right. This rearrangement is done inside the partition function, which swaps elements the pivot

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.

- **In the quick sort, the arrays is divided into a left sublist and the right sublist based on a pivot elements. After dividing we can use other sorting methods like insertion sort or bubble sort on the left and right parts. Example, with the array of {25 15 18 19 20 11} the pivot is 19 gives the left {11 15 18} and the right {20 25}. The left side can be sorted by insertion and the right side by bubble sort. After combining all of them the final sorted array 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 • log N) for their time complexity?

- **The array that been given can be sorted fastest using quick sort or the merger sort. Both of these algorithm technique are faster than bubble sort, selection sort or insertion sort. Their speed comes from dividing the array into smaller parts and solving them step by step. This process makes the number of steps around N log N instead of N^2. That's why merge sort and quick sort are very useful for bigger arrays.**

8. Conclusion:

- **These experiments showed that Shell Sort, Merge Sort, and Quick Sort all correctly ordered values in ascending order. Though the ways in which they approach the sorting problem differ, all have the same end of ordering data items. Shell Sort showed its simplicity and efficiency, while Merge Sort showed that dividing a problem into smaller problems makes its solutions easier to achieve. The flexibility of the Quick Sort algorithm writable as many ways it can be, producing at least one correct answer all the time is yet another demonstration of the same parameter. In instant, all of these show that the choice of sorting algorithm is highly subjective, and the possibility is an instant consideration, especially for the bigger array in concern.**

9. Assessment Rubric