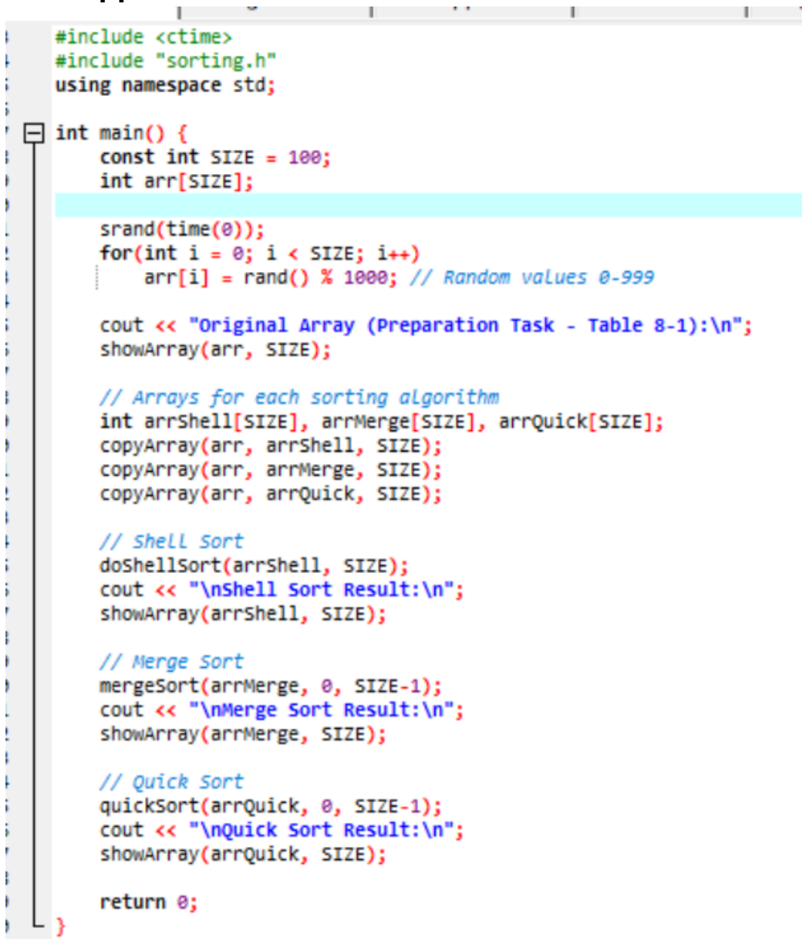
Hands-on Activity 8.1	
Sorting Algorithms Pt2	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 9/26/25
Section: CPE21S4	**Date Submitted:** 9/27/25
Name(s): Avila, Vince Gabriel V.	**Instructor:** Engr. Jimlord Quejado

6. Output

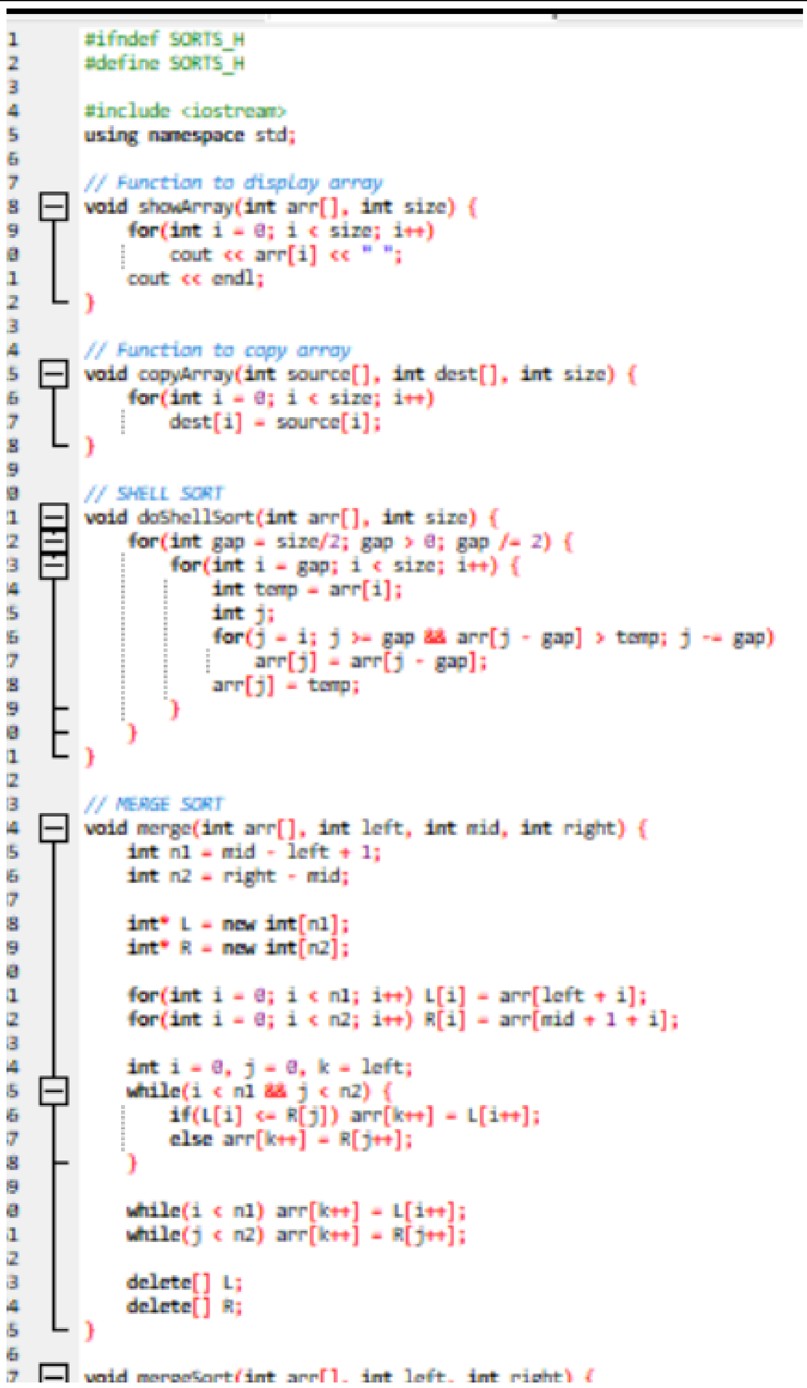
Table 8-1. Array of Values for Sort Algorithm Testing

main.cpp:



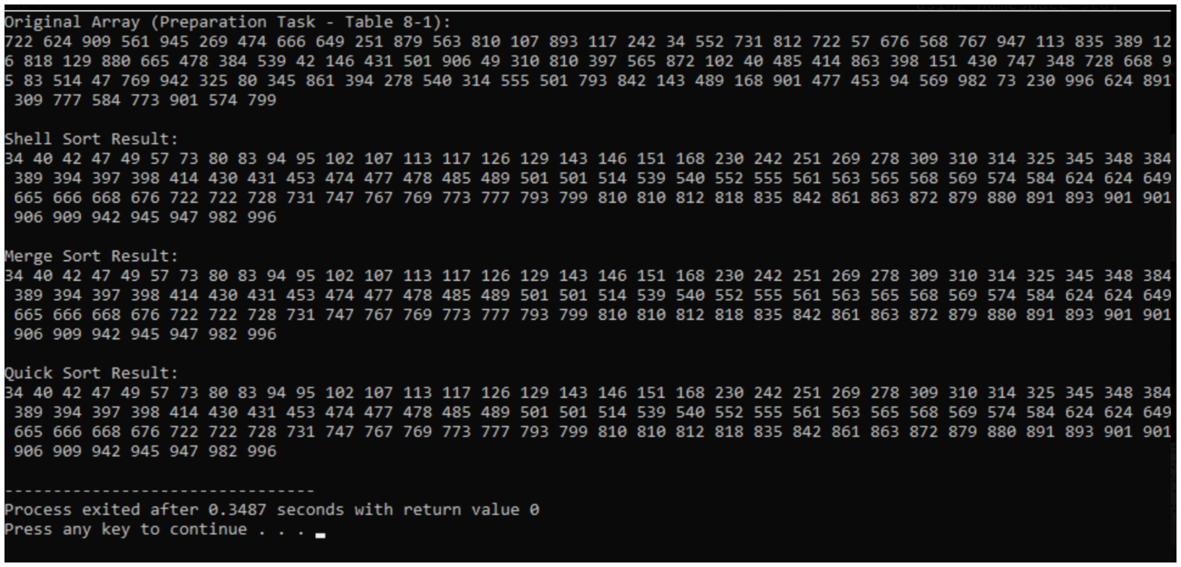
```cpp
#include <ctime>
#include "sorting.h"
using namespace std;

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

    srand(time(0));
    for(int i = 0; i < SIZE; i++)
        arr[i] = rand() % 1000; // Random values 0-999

    cout << "Original Array (Preparation Task - Table 8-1):\n";
    showArray(arr, SIZE);

    // Arrays for each sorting algorithm
    int arrShell[SIZE], arrMerge[SIZE], arrQuick[SIZE];
    copyArray(arr, arrShell, SIZE);
    copyArray(arr, arrMerge, SIZE);
    copyArray(arr, arrQuick, SIZE);

    // Shell Sort
    doShellSort(arrShell, SIZE);
    cout << "\nShell Sort Result:\n";
    showArray(arrShell, SIZE);

    // Merge Sort
    mergeSort(arrMerge, 0, SIZE-1);
    cout << "\nMerge Sort Result:\n";
    showArray(arrMerge, SIZE);

    // Quick Sort
    quickSort(arrQuick, 0, SIZE-1);
    cout << "\nQuick Sort Result:\n";
    showArray(arrQuick, SIZE);

    return 0;
}
```

Header File:



```cpp
#ifndef SORTS_H
#define SORTS_H

#include <iostream>
using namespace std;

// Function to display array
void showArray(int arr[], int size) {
    for(int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Function to copy array
void copyArray(int source[], int dest[], int size) {
    for(int i = 0; i < size; i++)
        dest[i] = source[i];
}

// SHELL SORT
void doShellSort(int arr[], int size) {
    for(int gap = size/2; gap > 0; gap /= 2) {
        for(int i = gap; i < size; 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 i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

    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;
}

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

Output:



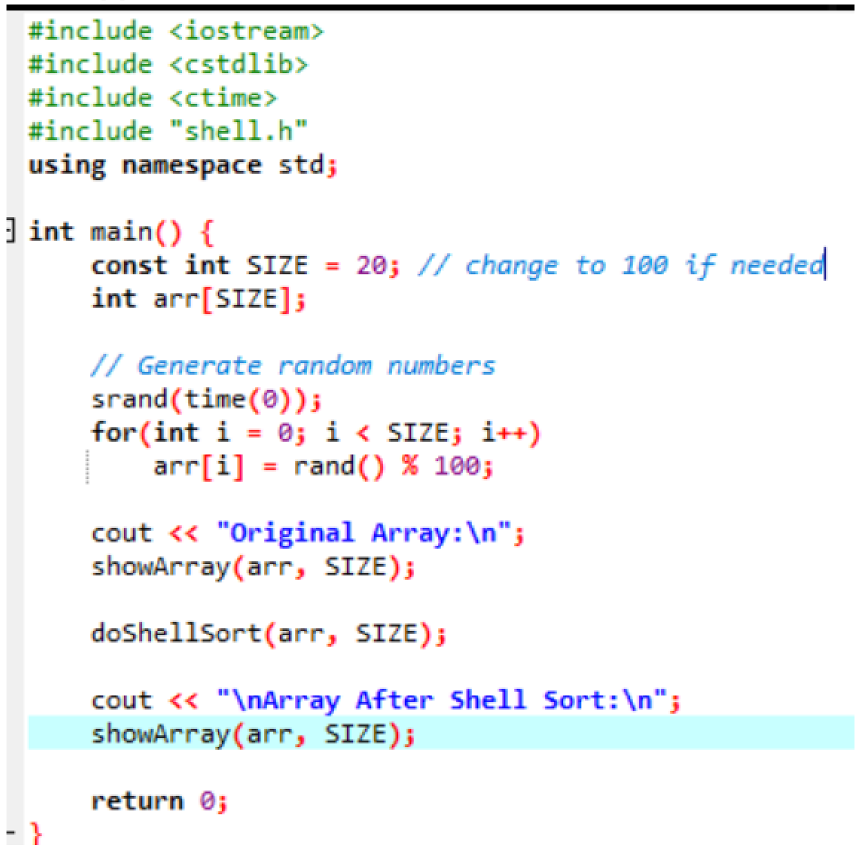
```
Original Array (Preparation Task - Table 8-1):
722 624 909 561 945 269 474 666 649 251 879 563 810 107 893 117 242 34 552 731 812 722 57 676 568 767 947 113 835 389 12
6 818 129 880 665 478 384 539 42 146 431 501 906 49 310 810 397 565 872 102 40 485 414 863 398 151 430 747 348 728 668 9
5 83 514 47 769 942 325 80 345 861 394 278 540 314 555 501 793 842 143 489 168 901 477 453 94 569 982 73 230 996 624 891
 309 777 584 773 901 574 799

Shell Sort Result:
34 40 42 47 49 57 73 80 83 94 95 102 107 113 117 126 129 143 146 151 168 230 242 251 269 278 309 310 314 325 345 348 384
 389 394 397 398 414 430 431 453 474 477 478 485 489 501 501 514 539 540 552 555 561 563 565 568 569 574 584 624 624 649
 665 666 668 676 722 722 728 731 747 767 769 773 777 793 799 810 810 812 818 835 842 861 863 872 879 880 891 893 901 901
 906 909 942 945 947 982 996

Merge Sort Result:
34 40 42 47 49 57 73 80 83 94 95 102 107 113 117 126 129 143 146 151 168 230 242 251 269 278 309 310 314 325 345 348 384
 389 394 397 398 414 430 431 453 474 477 478 485 489 501 501 514 539 540 552 555 561 563 565 568 569 574 584 624 624 649
 665 666 668 676 722 722 728 731 747 767 769 773 777 793 799 810 810 812 818 835 842 861 863 872 879 880 891 893 901 901
 906 909 942 945 947 982 996

Quick Sort Result:
34 40 42 47 49 57 73 80 83 94 95 102 107 113 117 126 129 143 146 151 168 230 242 251 269 278 309 310 314 325 345 348 384
 389 394 397 398 414 430 431 453 474 477 478 485 489 501 501 514 539 540 552 555 561 563 565 568 569 574 584 624 624 649
 665 666 668 676 722 722 728 731 747 767 769 773 777 793 799 810 810 812 818 835 842 861 863 872 879 880 891 893 901 901
 906 909 942 945 947 982 996

--------------------------------
Process exited after 0.3487 seconds with return value 0
Press any key to continue . . .
```

Analysis:

The code generates a list of random numbers and sorts it using three different sorting algorithms: Shell Sort, Merge Sort, and Quick Sort. You will see the original unsorted list, plus the sorted results for each sorting algorithm to show how they work. There are also simple functions to copy and print the original number list. Each sorting method offers a different approach to sorting the list, and to provide an easy comparison of the results. In addition, the sorting method you choose will help you learn how sorting algorithms work.

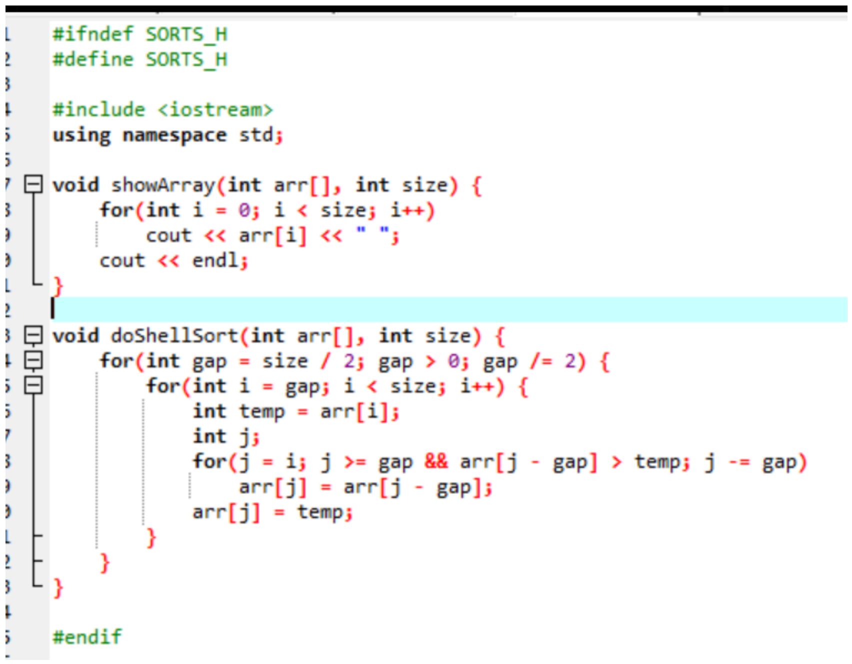
Table 8-2. Shell Sort Technique

main.cpp:



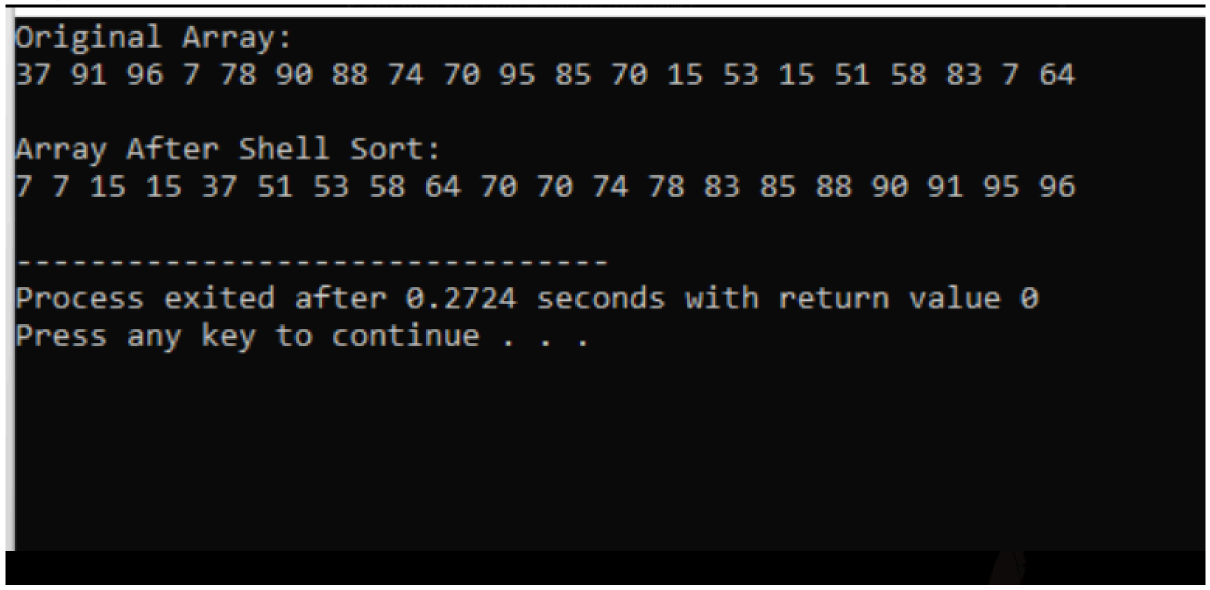
```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "shell.h"
using namespace std;

int main() {
    const int SIZE = 20; // change to 100 if needed
    int arr[SIZE];

    // Generate random numbers
    srand(time(0));
    for(int i = 0; i < SIZE; i++)
        arr[i] = rand() % 100;

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

    doShellSort(arr, SIZE);

    cout << "\nArray After Shell Sort:\n";
    showArray(arr, SIZE);

    return 0;
}
```

Header File:



```cpp
#ifndef SORTS_H
#define SORTS_H

#include <iostream>
using namespace std;

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

void doShellSort(int arr[], int size) {
    for(int gap = size / 2; gap > 0; gap /= 2) {
        for(int i = gap; i < size; 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;
        }
    }
}

#endif
```

Output:



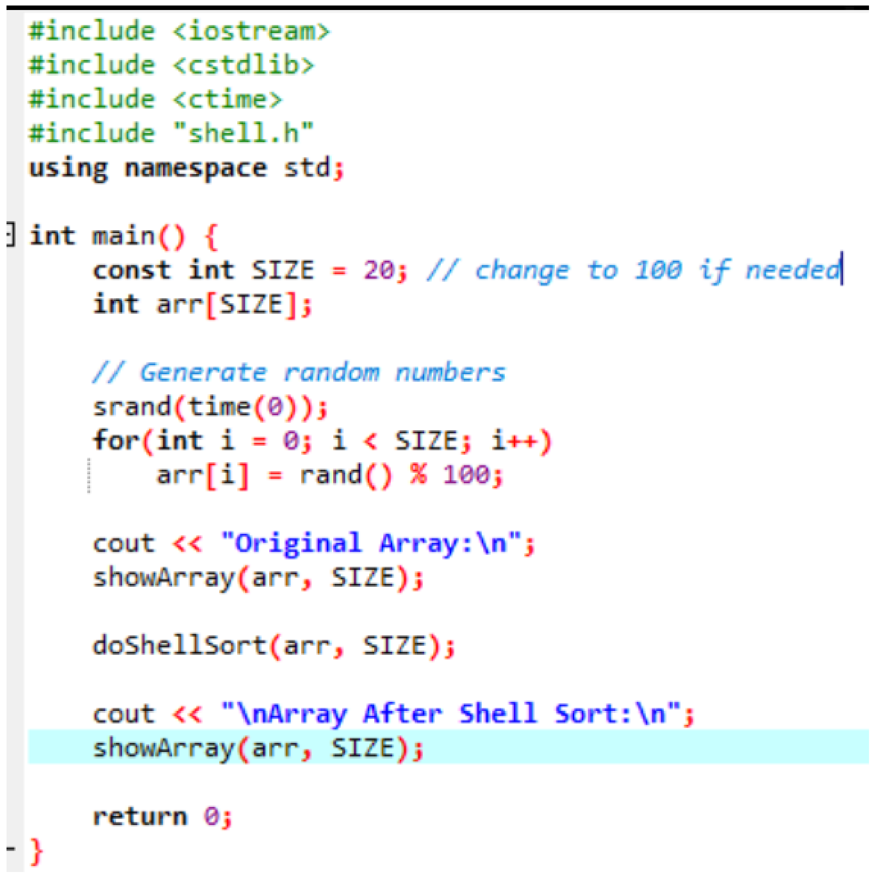
```
Original Array:
37 91 96 7 78 90 88 74 70 95 85 70 15 53 15 51 58 83 7 64

Array After Shell Sort:
7 7 15 15 37 51 53 58 64 70 70 74 78 83 85 88 90 91 95 96

--------------------------------
Process exited after 0.2724 seconds with return value 0
Press any key to continue . . .
```

Analysis:

This program generates a list of random numbers and sorts it with the Shell Sort algorithm. It displays the list to sort along with the finished sorted list so you can see the difference. Shell Sort compares numbers that are far apart before working its way closer together which means it sorts faster than a more simple approach. This program is a simple implementation to see how Shell Sort sorts a list step by step.

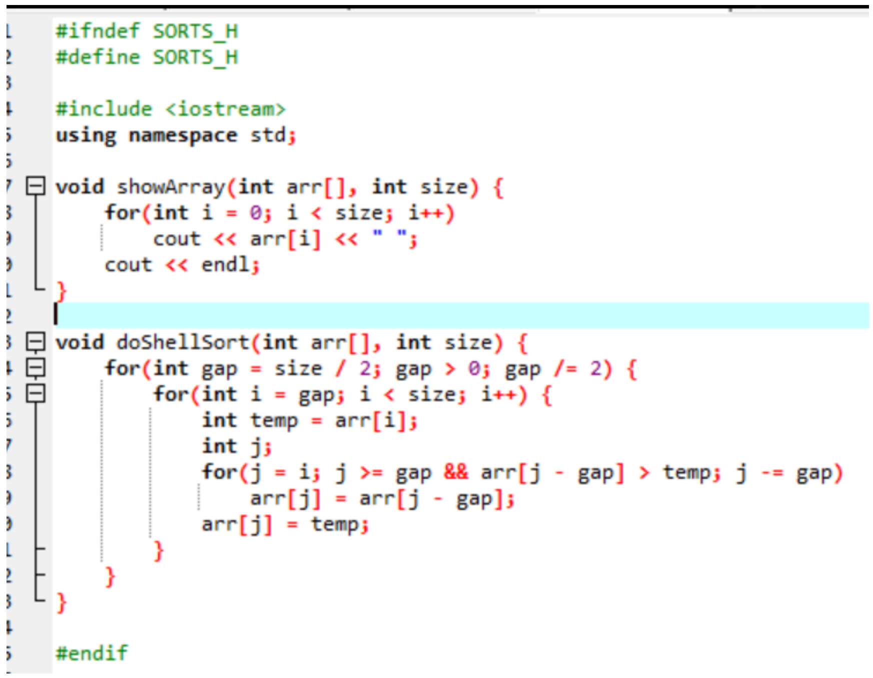
Table 8-3. Shell Sort Technique

main.cpp:



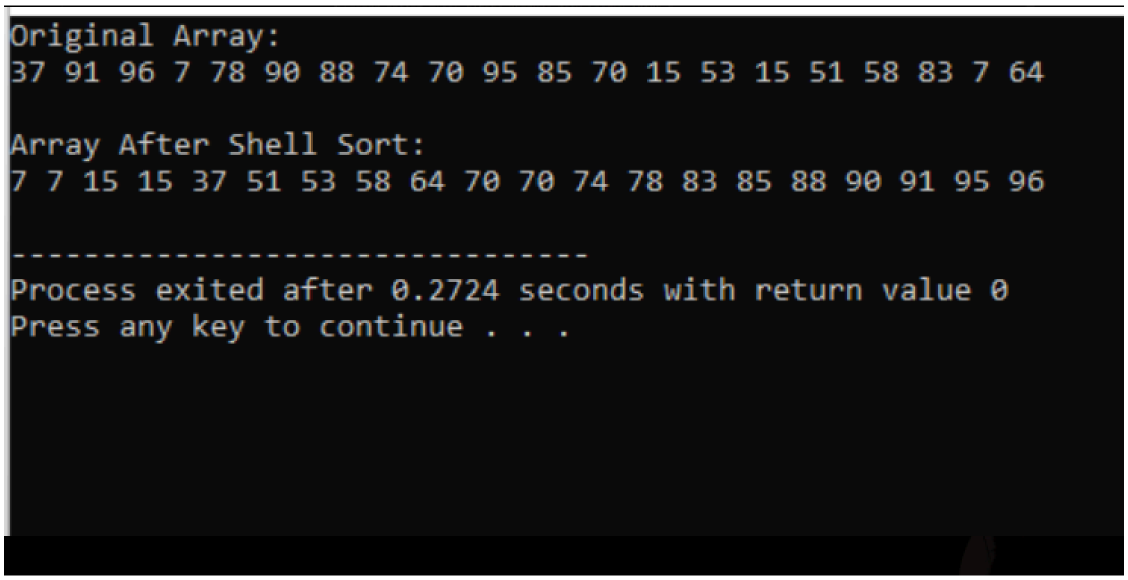
```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "shell.h"
using namespace std;

int main() {
    const int SIZE = 20; // change to 100 if needed
    int arr[SIZE];

    // Generate random numbers
    srand(time(0));
    for(int i = 0; i < SIZE; i++)
        arr[i] = rand() % 100;

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

    doShellSort(arr, SIZE);

    cout << "\nArray After Shell Sort:\n";
    showArray(arr, SIZE);

    return 0;
}
```

Header File:



```cpp
#ifndef SORTS_H
#define SORTS_H

#include <iostream>
using namespace std;

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

void doShellSort(int arr[], int size) {
    for(int gap = size / 2; gap > 0; gap /= 2) {
        for(int i = gap; i < size; 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;
        }
    }
}

#endif
```

Output:



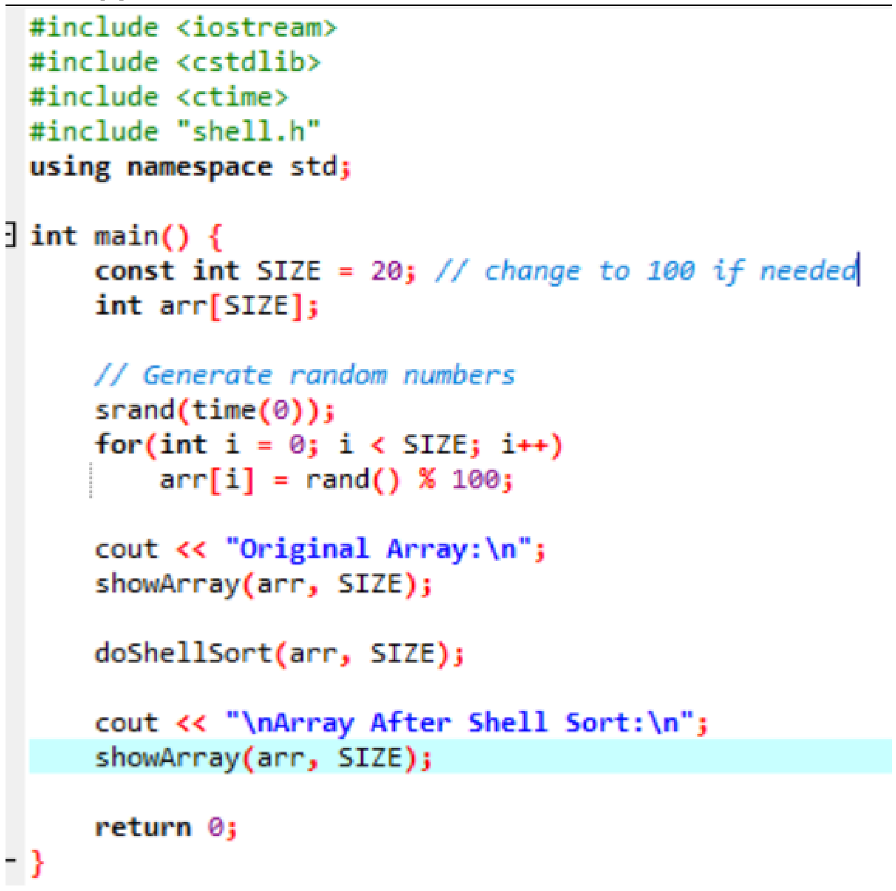
```
Original Array:
37 91 96 7 78 90 88 74 70 95 85 70 15 53 15 51 58 83 7 64

Array After Shell Sort:
7 7 15 15 37 51 53 58 64 70 70 74 78 83 85 88 90 91 95 96

--------------------------------
Process exited after 0.2724 seconds with return value 0
Press any key to continue . . .
```

Analysis:

This code creates a list of random numbers and sorts it using Merge Sort. It displays the list before sorting and after sorting so you can see the difference. Merge Sort works by breaking the list into smaller pieces, sorting those pieces, and then putting them back together in the correct order. Extra space is required to put the pieces together. This code is a very simple demonstration of how breaking a large problem into smaller pieces makes sorting simpler and faster.

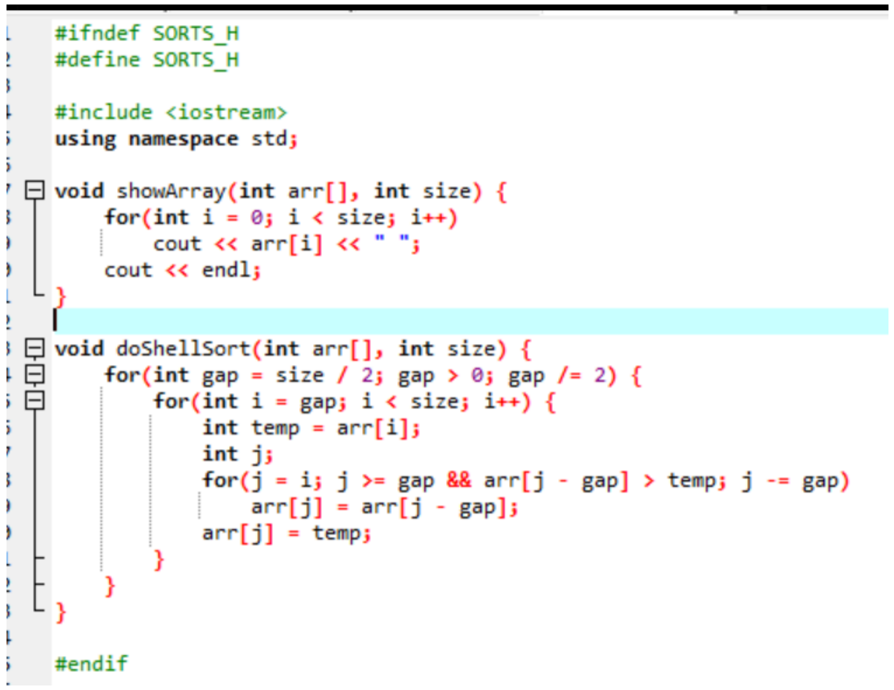
Table 8-4. Quick Sort Algorithm

main.cpp:



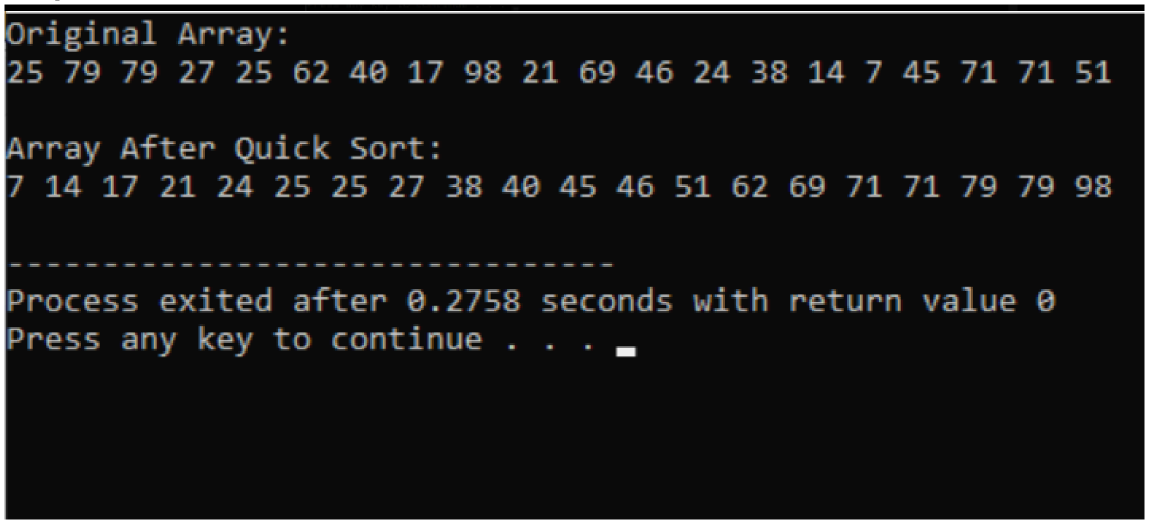
```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "shell.h"
using namespace std;

int main() {
    const int SIZE = 20; // change to 100 if needed
    int arr[SIZE];

    // Generate random numbers
    srand(time(0));
    for(int i = 0; i < SIZE; i++)
        arr[i] = rand() % 100;

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

    doShellSort(arr, SIZE);

    cout << "\nArray After Shell Sort:\n";
    showArray(arr, SIZE);

    return 0;
}
```

Header File:



```
#ifndef SORTS_H
#define SORTS_H

#include <iostream>
using namespace std;

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

void doShellSort(int arr[], int size) {
    for(int gap = size / 2; gap > 0; gap /= 2) {
        for(int i = gap; i < size; 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;
        }
    }
}

#endif
```

Output:



```
Original Array:
25 79 79 27 25 62 40 17 98 21 69 46 24 38 14 7 45 71 71 51

Array After Quick Sort:
7 14 17 21 24 25 25 27 38 40 45 46 51 62 69 71 71 79 79 98

--------------------------------
Process exited after 0.2758 seconds with return value 0
Press any key to continue . . .
```

Analysis:

This code generates a set of random numbers, and it sorts the list using the Quick Sort algorithm. It displays the list of numbers before and after sorting it so you can see how the order of the numbers has made a difference. With Quick Sort, you select one number from the list, called the pivot, and move all the values that are smaller than the pivot to one side of the pivot and all the numbers that are larger than the pivot to the other side of the pivot. Then you repeat this process on each side of the pivot until the entire list is sorted. So the idea of Quick Sort is that it can sort a list quickly enough by reducing the list down into smaller lists.

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.

Quick Sort is a method to sort numbers by taking a single number as the pivot, then putting the smaller numbers in one group, and the larger numbers in another. If we have 12, 7, 15, 20, 5, 10 and take 12 as the pivot, we would have 7, 5, and 10 in the left group and 15, 20 on the right side. We would be able to sort the left side using Insertion Sort and the right side with Bubble Sort, after which we would put everything back together. The sorted list would be 5, 7, 10, 12, 15, 20.

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?

Quick Sort or Merge Sort can sort an array significantly faster. Both are advantageous over the Bubble Sort, Selection Sort, and Insertion Sort because they cut the array into small pieces and sort them incrementally instead. In this manner, they reduce the time complexity from N squared to about N log N steps. This explains why Quick Sort and Merge Sort are appropriate for sorting larger lists.

8. Conclusion

In conclusion, I looked at how Shell Sort, Merge Sort, and Quick Sort were able to sort the data in ascending order. Each algorithm approaches the sorting differently, but they end up sorting the data the same way. Shell Sort was a quick and easy process, Merge Sort broke it down, and Quick Sort presented the idea that sorting can be done in many ways and still arrive at the same conclusion. This leads me to believe that sorting can be done many ways depending on your approach, especially with larger data sets.

9. Assessment Rubric