Activity No 7.2

Hands-on Activity 7.2 Sorting Algorithms

Course Code: CPE021	**Program:** Computer Engineering
Course Title: Computer Architecture and Organization	**Date Performed: 10/21/2024**
Section: CPE21S1	**Date Submitted: 10/23/2024**
Name(s): Jhon Hendricks T. Bautista	**Instructor:** Engr. Roman M. Richard

A. Procedure: Output(s) and Observation(s)

Code + Console Screenshot	
Observation	The code shown is for creating random values. The numbers generated are put into an array with the size of 100. The generated numbers are limited to 100. I used a for loop to print all the numbers in the array.

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

Code + Console Screenshot	main.cpp
	Share
	Run
	1 #include <iostream>
	2 #include <cstdlib>
	3 #include <ctime>
	4 using namespace std;
	5
	6 const int max_size = 100;
	7
	8 void shellSort(int arr[], int size) {
	9 for (int gap = size / 2; gap > 0; gap /= 2) {
	10 for (int i = gap; i < size; i++) {
	11 int temp = arr[i];
	12 int j;
	13 for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
	14 arr[j] = arr[j - gap];
	15 }
	16 arr[j] = temp;
	17 }
	18 }
	19 }
	20
	21 int main() {
	22 int dataset[max_size];
	23 srand(time(0));
	24
	25 for (int i = 0; i < max_size; i++) {
	26 dataset[i] = rand() % 101;
	27 }
	28
	29
	30 cout<<"UNSORTED VALUES"<<endl;
	31 for (int counter = 0; counter < max_size; counter++) {
	32 cout<<dataset[counter] << " ";
	33 }
	34 cout<<endl;
	35
	36 shellSort(dataset, max_size);
	37
	38 cout<<"SORTED VALUES"<<endl;
	39 for (int counter = 0; counter<max_size; counter++) {
	40 cout<<dataset[counter] << " ";
	41 }
	42 cout<<endl;
	43
	44 return 0;
	45 }
	46
	
	/tmp/yu2S82T3cB.o
	UNSORTED VALUES
	41 69 87 89 94 51 58 12 16 70 81 84 25 80 7 78 42 65 27 17 33 33 17 81 51 22 61 73 4 27 45 12 62 31 0 55 82 25 67 64 95 47
	48 19 93 55 63 1 19 56 85 18 55 68 65 72 90 26 44 61 19 89 73 81 86 39 35 34 30 1 98 91 14 45 9 6 66 38 74 51 60 59 36
	81 26 0 52 83 26 62 43 11 16 82 92 2 20 93 36 17
	SORTED VALUES (SHELL SORT)
	0 0 1 1 2 4 6 7 9 11 12 12 14 16 16 17 17 17 18 19 19 19 20 22 25 25 26 26 26 27 27 30 31 33 33 34 35 36 36 38 39 41 42 43
	44 45 45 47 48 51 51 51 52 55 55 55 56 58 59 60 61 61 62 62 63 64 65 65 66 67 68 69 70 72 73 73 74 78 80 81 81 81 81 82
	82 83 84 85 86 87 89 89 90 91 92 93 93 94 95 98
	
	=== Code Execution Successful ===

Observation	In this code I used shell sort to sort the given random numbers of arrays. In the implementation of the shell sort I used a double for loop in order to traverse the array. The first for loop is used to create a division of the array in the middle to serve as the gap of our shell sort. Then the second for loop uses the gap to compare the current number whether it is greater or less than the gap. After

	that another for loop is used for properly placing the number in the correct position. This continues until all the numbers are sorted.

Table 8-2. Shell Sort Technique

Code + Console Screenshot	

main.cpp

Share Run

```
11
12      int* L = new int[n1];
13      int* R = new int[n2];
14
15      for (int i = 0; i < n1; i++)
16          L[i] = arr[left + i];
17      for (int j = 0; j < n2; j++)
18          R[j] = arr[mid + 1 + j];
19
20      int i = 0, j = 0, k = left;
21      while (i < n1 && j < n2) {
22          if (L[i] <= R[j]) {
23              arr[k] = L[i];
24              i++;
25          } else {
26              arr[k] = R[j];
27              j++;
28          }
29          k++;
30      }
31
32      while (i < n1) {
33          arr[k] = L[i];
34          i++;
35          k++;
36      }
37
38      while (j < n2) {
39          arr[k] = R[j];
40          j++;
41          k++;
42      }
43
44      delete[] L;
45      delete[] R;
46  }
47
48  void mergeSort(int arr[], int left, int right) {
49      if (left < right) {
50          int mid = left + (right - left) / 2;
51
52          mergeSort(arr, left, mid);
53          mergeSort(arr, mid + 1, right);
54          merge(arr, left, mid, right);
55      }
56  }
57
```

Output

Clear

```
/tmp/6z7a0QlGjr.o
UNSORTED VALUES
22 0 1 2 32 9 46 93 85 74 73 37 35 39 20 5 44 50 16 77 73 63 56 86 92 4 64 75 75 39 53 97 39 21 99 72 30 10 30 14 84 70 18
    19 8 4 91 18 55 6 95 94 35 16 45 26 88 9 1 62 48 20 59 53 41 23 24 37 34 21 52 17 91 36 2 65 40 59 83 61 32 43 21 33 59
    66 60 46 41 27 8 55 13 33 8 21 56 99 58 56
SORTED VALUES (MERGE SORT)
0 1 1 2 2 4 4 5 6 8 8 8 9 9 10 13 14 16 16 17 18 18 19 20 20 21 21 21 21 22 23 24 26 27 30 30 32 32 33 33 34 35 35 36 37 37
    39 39 39 40 41 41 43 44 45 46 46 48 50 52 53 53 55 55 56 56 56 58 59 59 59 60 61 62 63 64 65 66 70 72 73 73 74 75 75 77
    83 84 85 86 88 91 91 92 93 94 95 97 99 99
```

Observation	For the merge sort algorithm I used two functions. The first function is used to divide the array into two for our merging process. The function gets all the numbers below the mid value and after the middle value. After getting the two halves of the array. Each sub array is then called to be sorted. After sorting the two halves, they are merged and then checked if they are now sorted.

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot

main.cpp

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

using namespace std;

const int max_size = 100;


void quickSort(int arr[], int low, int high) {
    int i = low, j = high;
    int pivot = arr[(low + high) / 2];

    while (i <= j) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;
        if (i <= j) {
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }

    if (low < j)
        quickSort(arr, low, j);
    if (i < high)
        quickSort(arr, i, high);
}
```

Output

Clear

```
/tmp/xcJBbDf3it.o
UNSORTED VALUES
13 4 55 38 6 97 54 60 69 38 96 84 45 34 57 30 69 23 60 29 22 62 24 63 96 64 5 61 100 46 7 13 16 28 17 22 91 38 48 25 42 43 9
   87 78 32 16 12 55 42 41 77 4 31 5 100 96 10 26 61 22 100 74 38 27 58 26 85 62 75 9 3 17 18 56 61 16 38 74 71 81 81 14 85
   79 86 50 40 63 77 0 85 76 41 90 70 99 15 54 26
SORTED VALUES (Quick Sort)
0 3 4 4 5 5 6 7 9 9 10 12 13 13 14 15 16 16 16 17 17 18 22 22 22 23 24 25 26 26 26 27 28 29 30 31 32 34 38 38 38 38 38 40 41
   41 42 42 43 45 46 48 50 54 54 55 55 56 57 58 60 60 61 61 61 62 62 63 63 64 69 69 70 71 74 74 75 76 77 77 78 79 81 81 84
   85 85 85 86 87 90 91 96 96 96 97 99 100 100 100
```

Observation	In the code for quick sort, it uses a variable called the pivot which we can declare in the given array. This pivot is a vital element for the quick sort to work because the function bases where to put the current value in the array with the condition of the pivot.

Table 8-4. Quick Sort Algorithm

B. Supplementary Activity: Output(s) and Observation(s)

Problem 1

Code + Console Screenshot

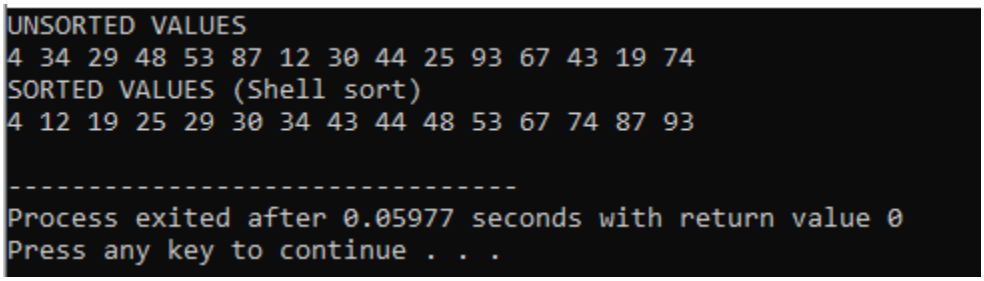
```
void bubbleSort(int arr[], int start, int end) {
    for (int i = start; i <= end; i++) {
        for (int j = start; j <= end - (i - start) - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high], 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 QuickSortwithOthers(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        bubbleSort(arr, low, pi - 1);
        bubbleSort(arr, pi + 1, high);
    }
}
```

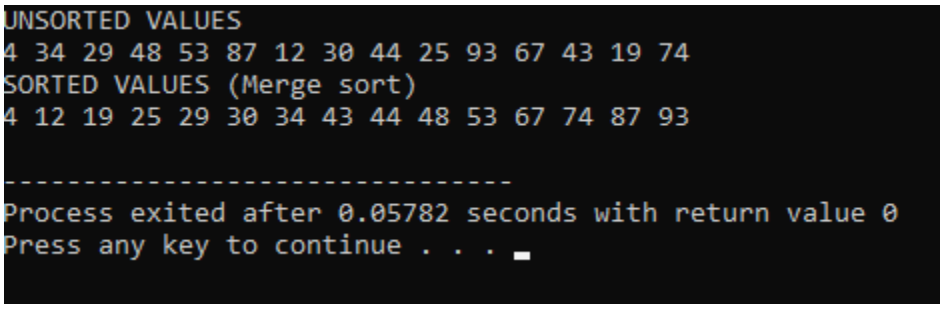
```
int main() {
    int dataset[max_size];
    srand(time(0));
    for (int i = 0; i < max_size; i++){
      dataset[i] = rand() % 50;
    };

    cout<< "UNSORTED VALUES"<<endl;
    for (int counter = 0; counter < max_size; counter ++ ){
      cout <<  dataset[counter]<<" ";
    };
    cout<<endl;
   QuickSortwithOthers(dataset, 0, max_size - 1);

    cout << "SORTED VALUES"<<endl;
    for (int counter = 0; counter < max_size; counter ++ ){
      cout <<  dataset[counter]<<" ";
    };

    return 0;
}
```

Observation	Yes it is possible to use different sorting techniques in the divided regions of the list in a partition method with a quick sort. In my example I was able to utilize the partition method working in tandem with a bubble sort in a quick sort function. The pivot allowed us to determine which side to put the value then those values will be sorted out in their own sides by the bubble sort.

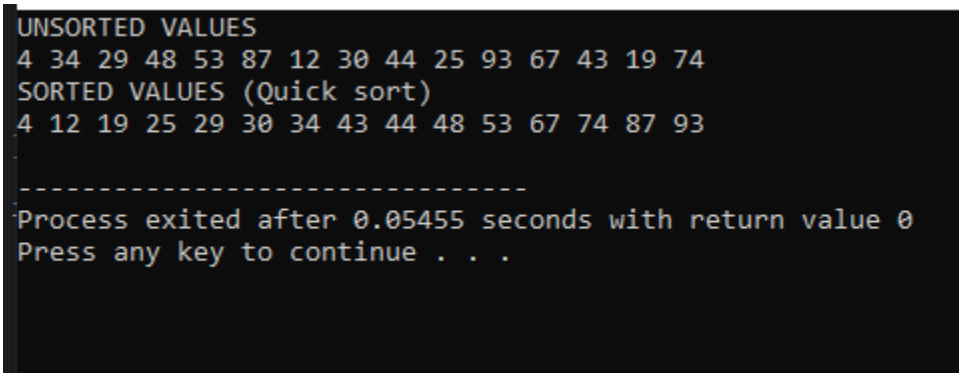
Problem 2

Code + Console Screenshot	

Shell Sort

Code + Console Screenshot	

Merge Sort

Code + Console Screenshot	

Quick Sort

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?

Based on my testing of the different algorithms, the quick sort was able to do it the fastest out of all the three. This was possible due to the way how quicksort works its divide and conquer principle and as well as the merge sort. They both have a time complexity of O(N • log N) because their initial steps always start with dividing the array or the list though they have differences. This is because merge sort uses the division by separating the list entirely which is log N and sorting them separately then merge them back to together to have N to have an overall of O(N • log N). On the other hand, quick sort partitions the array around a pivot element and then recursively sorts the left and right subarrays. The recursion is log N then the process give N factor to have a result of O(N • log N).

C. Conclusion & Lessons Learned

In this activity I was able to learn the different sorting techniques. These are the Shell sort, Merge sort, and Quick sort. I learned that quick sort is often times better than the two because it is O(n log n) in time complexity. I learned the basic structure of implementing these sorting techniques in sorting an array of numbers in C++. I understood that these sorting techniques have a common principle of divide and conquer where they divide the array and sort them based on their conditions. They also implement recursion because the process repeats itself until the array is sorted.