**Project-11**

**Name: Pawan Dhungana**

**Project no: 11**

**Due Date: December 13, 2018**

**Design Document**

**Introduction**

A sorting algorithm is an algorithm that puts elements of a list in a certain order.

We have recently described two sorting algorithms---Merge Sort and Quick Sort---whose times are, in general, O(n log n). Heap Sort is another sorting algorithm whose time is O(n log n). This project implements and exercises these algorithms and an earlier algorithm whose time is O(n2). This investigation should confirm our theoretical observations about their behaviors.

**Data Structures**

The program uses an array and as we need to bring the sorted array in the previous array to know the sort of other arrays, we have used another array in the class. There are different modification member functions in the class’s public section.

**Functions**

The default constructor is used to initialize the counts for different sorts. Different functions are used for different sorts. There are 6 functions for all the sorts and one function to print the result in the screen. They are:  
Insertion sort functions:  
- void insertionSort(): This will sort the array using Insertion Algorithm and return the merge count.

Merge sort functions:

-void mergeSort():This will sort the array using merge sorting Algorithm with the help of another array and return the merge count.   
-void Merge(): Help the function Merge\_sort to sort the array into another array.

Quick sort functions:

-void quickSortinsert():Inserts the sorted array into the quick\_sort() function

-void quicksort(): Returns the quick\_count and sort the arrays using quick sort algorithm.

-void partition(): Help Quick\_sort function to partition the array into corresponding length.

Heap sort functions:

-void HeapSort(): sorts the array

-void maxHeapify(): A function to heapify the array.

- void Build\_maxHeap(): Builds maximum heap.

Function to print the result  
-void writeCount(): write out the sorting algorithm counts.

**Main Program**

In the main program, different functions are being called. Other than that, there is a loop running the user interrupts the program. This helped in data collection for the graph.

**User Document**

A sorting algorithm is an algorithm that puts elements of a list in a certain order.

We have recently described two sorting algorithms---Merge Sort and Quick Sort---whose times are, in general, O(n log n). This project implements and exercises these algorithms and an earlier algorithm whose time is O(n2). This investigation should confirm our theoretical observations about their behaviors.

The program's name is Project11.cpp, to compile and run it, simply enter:

g++ Project11.cpp

a.out

A run of the program might look like this:

Enter the number of values to generate and sort, between 1 and 5000: 750

Enter an integer seed value: 42

Print the values? n

Insertion Sort count = 145364

Merge Sort count = 14452

Quick Sort count = 7785

Heap Sort count = 6780

**Code Listing:**

#include<iostream>

#include<cstdlib>

using namespace std;

class bigO

{

public:

int mergeCount;

int insertionCount;

int quicksortCount;

int heapsortCount;

//Constructor

bigO(){

mergeCount = 0;

insertionCount=0;

quicksortCount=0;

}

//Modification Member Functions

//Insertion Sort Function

void insertionSort(int Array[],int size);

//Merge Sort Functions

void mergeSort(int Array[], int low, int high);

void Merge(int a[], int low, int mid, int high);

//Quick Sort functions

void quickSortinsert(int a[], int size);

void quickSort(int a[], int low, int high);

void partition(int a[],int low, int high, int pivot, int &i, int &j);

//Heap Sort functions

void MaxHeapify(int a[], int i, int n);

void HeapSort(int a[], int n);

void Build\_MaxHeap(int a[], int n);

//Prints the content to main

void writeCount(int size);

private:

static const int MAX = 5000;

int sortedArray[MAX];

};

int main()

{

bigO Sort;

int seed;

char ch;

int size;

cout<<"Press any key to continue...|Q to quit|"<<endl;

cin>>ch;

do

{

cout<<"Enter number of values to generate and sort, between 1 and 5000: "<<endl;

cin>>size;

cout<<"Enter an integer seed value: ";

cin>>seed;

srand(seed);

int list[size];

//initializing random numbers into the array

for(int index = 0; index<size; ++index)

{

list[index]=rand()%(size-0)+size;

}

Sort.insertionSort(list, size);

Sort.quickSortinsert(list,size);

Sort.mergeSort(list, 0, size);

Sort.HeapSort(list,size);

Sort.Build\_MaxHeap(list,size);

Sort.writeCount(size);

}while (ch != 'q');

return 0;

}

void bigO::insertionSort(int Array[],int size)

{

int index2;

for(int i=0;i<size;++i)

{

sortedArray[i]=Array[i];

}

for(int index=1;index<size;++index)

{

index2=index;

while(index2>0 && sortedArray[index2]<sortedArray[index2-1])

{

swap(sortedArray[index2],sortedArray[index2-1]);

++insertionCount;

--index2;

}

}

}

void bigO::mergeSort(int Array[], int low, int high)

{

int mid;

++mergeCount;

if(low<high)

{

mid = (low+high)/2;

mergeSort(Array, low, mid);

Merge(Array,low,mid,high);

}

}

void bigO::Merge(int a[], int low, int mid, int high)

{

int b[MAX];

int i1, i2, index;

for(index=low;index<high;++index)

{

++mergeCount;

//copying the elements

b[index]=a[index];

}

i1=low;

i2=mid+1;

index=low;

while(i1<=mid && i2<=high)

{

++mergeCount;

if(a[i1]<a[i2])

a[index++]=b[i1++];

else

a[index++]=b[i2++];

}

while(i1<=mid)

{

a[index++] = b[i1++];

}

while(i2<=high)

{

a[index++] = b[i2++];

}

}

void bigO::quickSortinsert(int a[], int size)

{

for(int i=0;i<size;++i)

sortedArray[i]=a[i];

quickSort(sortedArray, 0, size);

}

void bigO::quickSort(int a[], int low, int high)

{

int i,j;

int pivot;

++quicksortCount;

if(low<high)

{

pivot = a[low];

partition(a,low,high,pivot,i,j);

quickSort(a,low,i);

quickSort(a,j,high);

}

}

void bigO::partition(int a[], int low, int high, int pivot, int&i, int&j)

{

int lastS1 = low-1;

int firstU = low;

int firstS3 = high+1;

while(firstU<firstS3)

{

if(a[firstU]<pivot)

{

++lastS1;

swap(a[firstU],a[lastS1]);

++firstU;

}

else if(a[firstU]==pivot)

++firstU;

else

{

--firstS3;

swap(a[firstU], a[firstS3]);

}

++quicksortCount;

}

i=lastS1;

j=firstS3;

}

// A function to heapify the array.

void bigO::MaxHeapify(int a[], int i, int n)

{

int j, temp;

temp = a[i];

j = 2\*i;

while (j <= n)

{

if (j < n && a[j+1] > a[j])

j = j+1;

// Break if parent value is already greater than child value.

if (temp > a[j])

break;

// Switching value with the parent node if temp < a[j].

else if (temp <= a[j])

{

a[j/2] = a[j];

j = 2\*j;

}

++heapsortCount;

}

a[j/2] = temp;

return;

}

void bigO::HeapSort(int a[], int n)

{

int i, temp;

for (i = n; i >= 2; i--)

{

// Storing maximum value at the end.

temp = a[i];

a[i] = a[1];

a[1] = temp;

// Building max heap of remaining element.

MaxHeapify(a, 1, i - 1);

}

}

void bigO::Build\_MaxHeap(int a[], int n)

{

int i;

for(i = n/2; i >= 1; i--)

{

MaxHeapify(a, i, n);

}

}

void bigO::writeCount(int size)

{

cout<<endl;

char ch;

cout<<"Print the values? "<<endl;

cin>>ch;

if(ch=='y')

{

for(int index = 0; index<size; ++index)

{

cout<<sortedArray[index]<<" \t ";

}

}

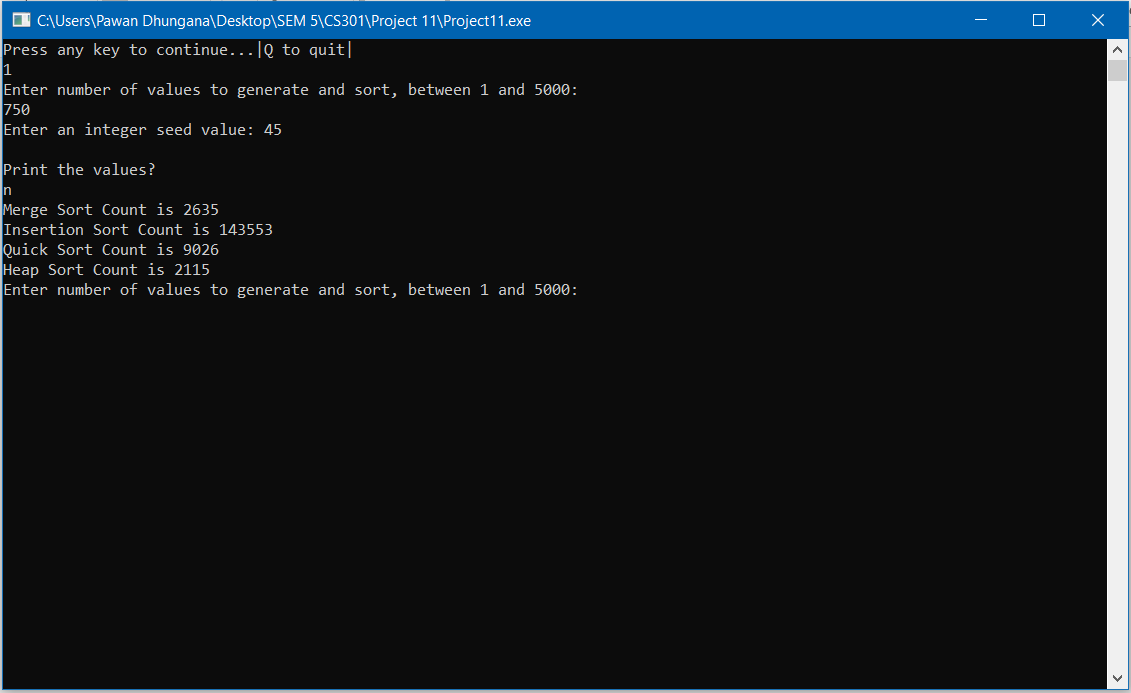
cout<<"Merge Sort Count is "<<mergeCount<<endl;

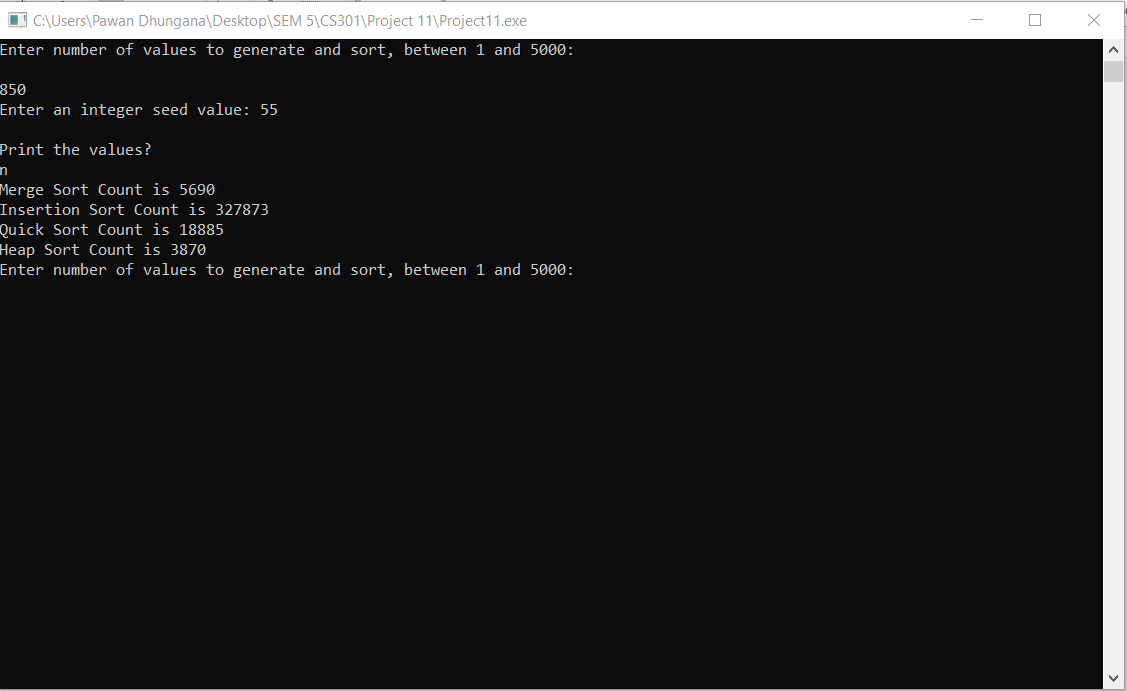
cout<<"Insertion Sort Count is "<<insertionCount<<endl;

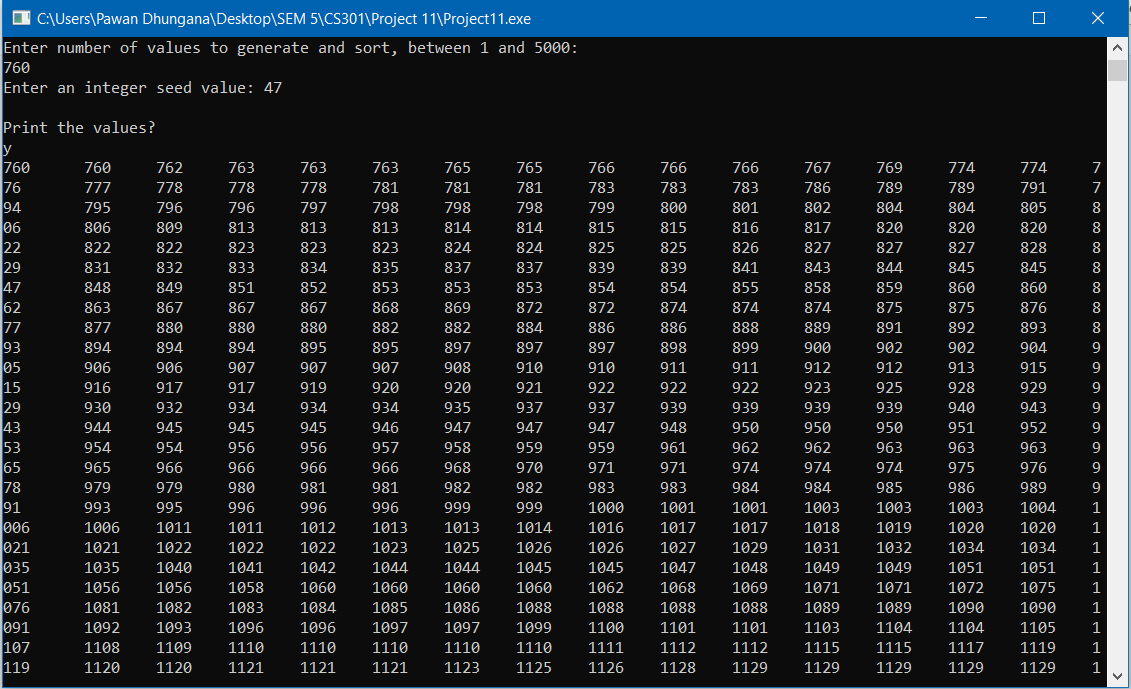
cout<<"Quick Sort Count is "<<quicksortCount<<endl;

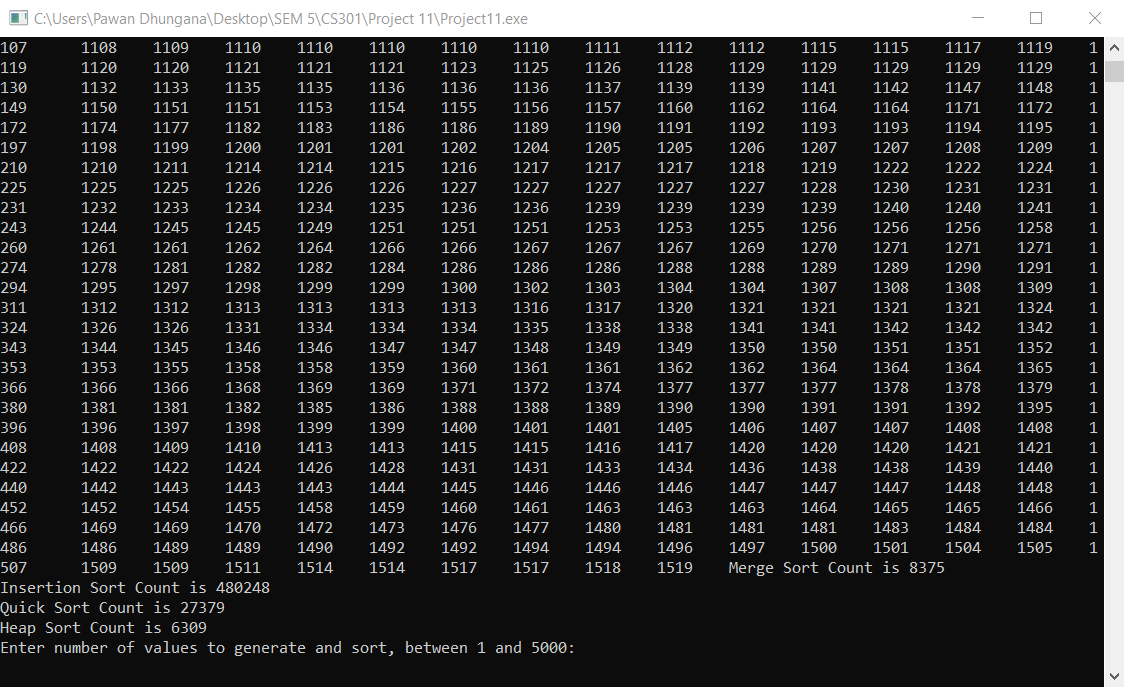
cout<<"Heap Sort Count is "<<heapsortCount<<endl;

}

**Test Document **

****

****

****

**Summary**

In this project, we implemented and exercised different sorting algorithms and confirmed our theoretical observations about their behaviours.

Other than that, I used different values and check the time count for each algorithm and I got the following result in the graphs I have attached. From the graphs, we can confirm that the Big O of algorithms that was discussed in class. Yes, the operations vary with the initial arrangement of the value because already sorted array need not be sorted again.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Insertion Sort | Merge Sort | Quick Sort | Heap Sort |
| 20 | 85 | 179 | 94 | 77 |
| 40 | 462 | 549 | 309 | 250 |
| 60 | 1450 | 1271 | 721 | 699 |
| 80 | 2980 | 2225 | 1309 | 1156 |
| 100 | 5505 | 3474 | 2609 | 2100 |
| 120 | 9500 | 5041 | 2982 | 2236 |
| 140 | 14210 | 6942 | 4295 | 3796 |