**Project-9**

**COMPARING THREE SORTING ALGORITHMS**

**Name: Pawan Dhungana**

**Project no: 09**

**Due Date: November 19, 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). 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 data structure to store data as we need to generate a specified number of random integers, records them in three arrays, then sorts the arrays with different sorting algorithms. An array is declared in the private section of the class to store the pseudo-random numbers generated.

**Functions**

The default constructor is used to initialize the counts for different sorts. Other than that, other 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()

Merge sort functions:

-void merge\_sort()  
-void Merge()

Quick sort functions:

-void quick\_sort()

-void quicksort()

-void partition()

Function to print the result  
-void writeCount()

**Main Program**

In the main program, different functions are being called. Also, from the main function, an array of pseudo-random numbers are created and passed as parameters in different sorting functions.

**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 Project9.cpp, to compile and run it, simply enter:

g++ Project9.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

**Code Listing:**

#include<iostream>

#include<cstdlib>

using namespace std;

class bigO

{

public:

int insertionCount;

int mergeCount;

int quicksortCount;

//Constructor

bigO()

{

mergeCount = 0;

insertionCount=0;

quicksortCount=0;

}

//Modification Member Functions

//Insertion Sort Function

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

//Merge Sort Functions

void merge\_sort(int a[], int low, int high);

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

//Quick Sort functions

void quick\_sort(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);

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

cin>>size;

cout<<"\nEnter an integer seed value: ";

cin>>seed;

srand(seed);

int a[size];

//initializing random numbers into the array

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

{

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

}

Sort.insertionSort(a, size);

Sort.quick\_sort(a,size);

Sort.merge\_sort(a, 0, size);

Sort.writeCount(size);

return 0;

}

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

{

int j;

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

{

sortedArray[i]=a[i];

}

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

{

j=index;

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

{

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

++insertionCount;

--j;

}

}

}

void bigO::merge\_sort(int a[], int low, int high)

{

int mid;

if(low<high)

{

mid = (low+high)/2;

merge\_sort(a, low, mid);

merge\_sort(a, mid+1, high);

Merge(a,low,mid,high);

}

++mergeCount;

}

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

{

int b[MAX];

int i, i1, i2;

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

{

++mergeCount;

//copying the elements of the array

b[i]=a[i];

}

i1=low;

i2=mid+1;

i=low;

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

{

++mergeCount;

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

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

else

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

}

while(i1<=mid)

{

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

}

while(i2<=high)

{

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

}

}

void bigO::quick\_sort(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) //recursive function

{

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) //patition step in quick sort

{

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;

}

void bigO::writeCount(int size) //write out the contents in the array

{

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 ";

}

}

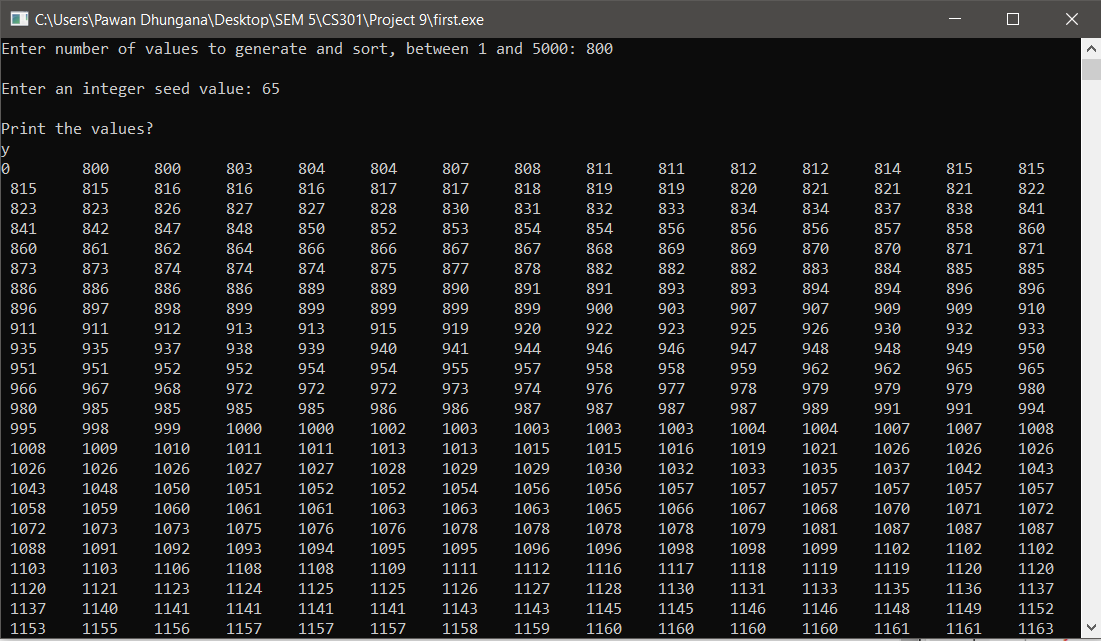
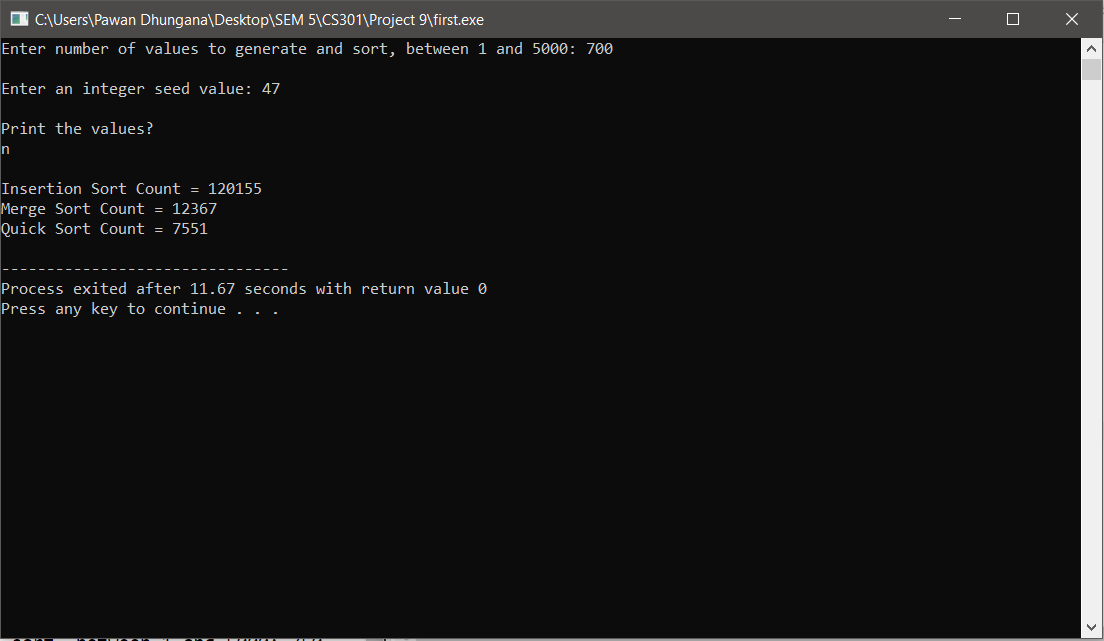
cout<<"\nInsertion Sort Count = "<<insertionCount<<endl;

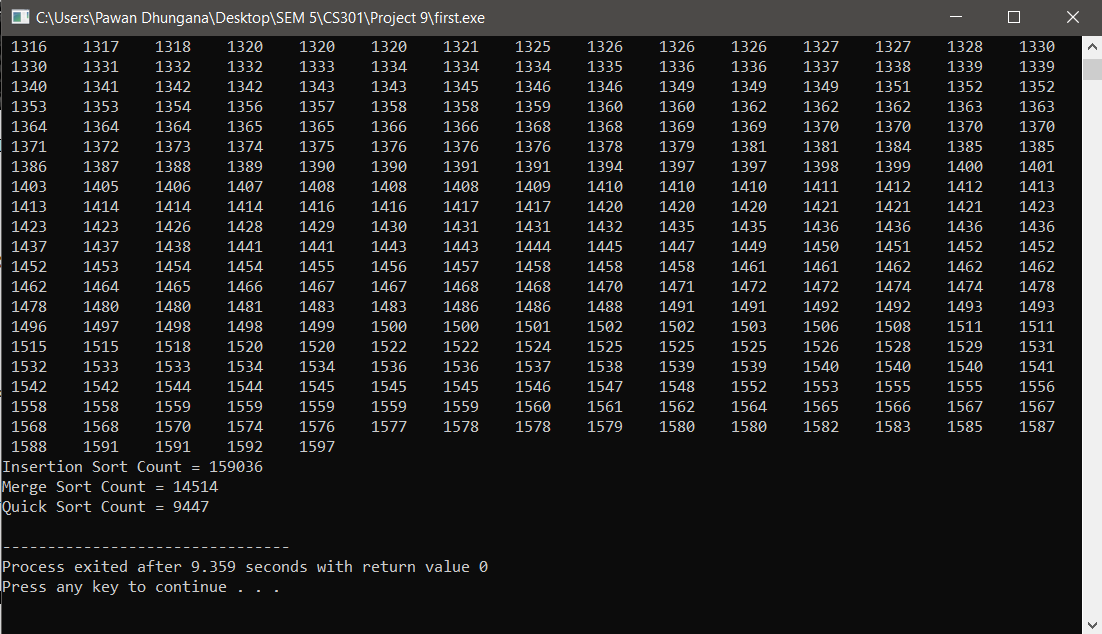
cout<<"Merge Sort Count = "<<mergeCount<<endl;

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

}

**Test Document**

****

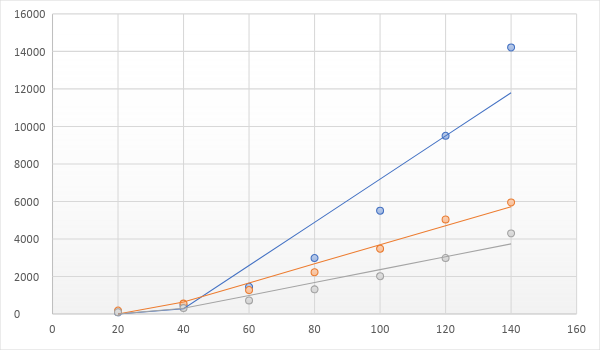
****

**Summary**

In this project, we implemented and exercised different sorting algorithms , compared and confirmed our theoretical observations about their big O behaviors.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. of values to generate & sort | Seed | Insertion Sort | Merge Sort | Quick Sort |
| 20 | 42 | 72 | 183 | 111 |
| 40 | 45 | 368 | 427 | 220 |
| 60 | 45 | 884 | 689 | 373 |
| 80 | 5 | 1579 | 972 | 551 |
| 100 | 25 | 2468 | 1264 | 688 |





 Yes, the numbers of operations vary with the initial arrangement of the values