Sushil Khanal

Section 1

Project 10

**EVALUATING PREFIX EXPRESSIONS USING A POINTER-BASED STACK**

**Introduction**

Shorting help to arrange data in a particular fashion, either in ascending or descending order. We use short() to perform sorting algorithm. In this program we use merge short, quick sort and insertion sort. In the class lecture we proved this by looking at The Big O notation for each short and proved merge sort and quick sort are faster than insertion sort. The Big O notation for merge short and quick short is(n.logn) and insertion sort is n.

Insertion sort works by placing an unsorted element at its suitable place in each iteration. Quick sort works by choosing a pivot value which is usually the first value, the pivot value helps split the list. The exact position where the pivot value belongs in the final sorted list, which is called the split point, it will be used to divide the list for subsequent calls to the quick sort. Merge short works by breaking down a list into several sublists until each sublist consists of a single element and after merging them we get a sorted list.

**Data structure**

We use array in this program. This program sorts the given array in ascending order using merge sort, quick sort and insertion sort.

**Functions**

Int merge() : This is the main algorithm of merge sort, this takes four parameters. One of which is array and the rest are integers.

Int quick(): This algorithm takes three parameters. One of them is array and the other two are integers.

Void insertionSort(): Insertion sort places an unsorted element at its suitable place in each iteration.

Void merge(): Merge short works by breaking down a list into several sublists until each sublist consists of a single element and after merging them we get a sorted list.

Void mergeSort(): This provides the platform for merge sort recurcively.

Void quickSort():This provides the platform for quick sort recurcively.

Void partition(): Quick sort works by choosing a pivot value which is usually the first value, the pivot value helps split the list. The exact position where the pivot value belongs in the final sorted list, which is called the split point, it will be used to divide the list for subsequent calls to the quick sort.

Void exchange(): It helps swaps the two value between arrays for quick sort.

**The main program**

At first the variables are declared. After declaration of variables, the program asks for input from the user. And we add random variable in the array using rand functions. We call three different sorting algorithm: merge sort, quick sort, insertion sort. At the very end it prints out the output.

**Code**

Sort.cpp

#include <iostream>

#include "sort.h"

using namespace std;

void sort::insertionSort(int ary[], int high)

{

int val; //decleration of variable

int i,j; //decleration of variable

for (i = 1; i < high; i++) //starting loop

{

val = ary[i]; //assigning the value

j = i;

count++;

while (j >= 0 && ary[j] > ary[i-1]) //starting while loop

{

exchange(ary[j],ary[j-1]); //calling the function

j--;

count++; //increasing the value by 1

}

ary[j+1] = val; //assigning the value

}

}

void sort:: merge(int a[], int low, int mid, int high) //starting merge function

{

int N; //decleration of variable

int b[N]; //decleration of variable

int i1, i2, index; //decleration of variable

for (index = low; index <= high; index++) // starting the loop

{

count++; // increasing the value by 1

b[index] = a[index];

i1=low;

i2=mid+1;

index = low;

while(i1 <= mid && i2 <=high) //starting while loop

{

count++;

if(b[i1]<b[i2]) /// starting the loop

{

a[index]=b[i1];

i1++;

}

else

{

a[index]=b[i2]; //assigning the value

i2++; // increasing the value by 1

}

index++; // increasing the value by 1

}

while(i1<=mid) // starting while loop

{

count++; //increasing the value by 1

a[index++] =b[i1++]; // assigning the value

}

while(i2<=high) //starting while loop

{

count++; // increasing the value by 1

a[index++]=b[i2++]; // assigning the value

}

}

}

void sort:: mergeSort(int a[], int low, int high) //starting function

{

int mid; //declaration of variable

if (low<high)

{

mid =(low+high)/2;

mergeSort(a, low, mid);

mergeSort(a,mid+1,high);

merge(a,low,mid,high);

}

}

void sort:: partition(int a[],int low, int high, int pivot, int&i, int&j) //starting function

{

int lastS1 =low-1; //declaration of variable

int firstU = low; //declaration of variable

int firstS3 =high+1; //declaration of variable

while(firstU<firstS3) // starting while loop

{

count++; //increasing the value by 1

if(a[firstU]<pivot) //starting the loop

{

lastS1++; //increasing the value by 1

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

firstU++; //increasing the value by 1

}

else if(a[firstU]==pivot)

{

firstU++;

}

else if(a[firstU]> pivot)

{

firstS3--;

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

}

}

}

void sort::quickSort(int a[], int low, int high) // starting function

{

int pivot; //declaration of variable

int lastS1,firstS3; //declaration of variable

if (low< high) //starting the loop

{

pivot = a[low];

partition(a,low,high,pivot,lastS1, firstS3);

quickSort(a,low,lastS1);

quickSort(a,firstS3,high);

}

}

void sort::exchange(int &i, int &j) //starting the function

{

int temp = i;

i = j;

j = temp;

}

Main.cpp

#include<iostream>

#include "sort.h"

#include "sort.cpp"

#include <cstdlib>

#include<time.h>

using namespace std;

int main() { //main function

const int size = 5000;

int merge[size];

int quick[size];

int insert[size];

sort q,i,m;

int val; //declaring the variable

char ch;

int seed; //declaring the variable

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

cin >> val;

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

cin >> seed;

cout <<"Print the values: ";

cin >> ch;

srand(seed);

for (int j=0; j<val; j++)

{

merge[j]=(rand()%5000)+1;

quick[j]=merge[j];

insert[j]=merge[j];

}

m.mergeSort( merge, 0, val);

q.quickSort( quick, 0, val);

i.insertionSort( insert, val);

if(ch=='y')

{

cout<<"Insertion Sort- "<<endl;

for(int j=1; j<=val; j++)

{

cout<<insert[j]<<" ";

}

cout<<endl<<cout<<"Merge Sort-"<<endl;

for(int j=1; j<=val; j++)

{

cout<<merge[j]<<" ";

}

cout<<endl<<" Quick Sort-"<<endl;

for(int j=1; j<=val; j++)

{

cout<<quick[j]<<" ";

}

}

cout<<endl;

cout<<"Count Insertion Sort = "<<i.size()<<endl;

cout<<"Count Merge Sort = "<<m.size()<<endl;

cout<<"Count Quick Sort = "<<q.size()<<endl;

return 0;

}

Sort.h

#ifndef SORT\_H\_INCLUDED

#define SORT\_H\_INCLUDED

class sort

{

private: //private member function

int count;

public: //public member function

sort()

{

count =0;

}

int size()

{

return count;

}

void insertionSort(int[], int);

void merge(int[], int, int, int);

void mergeSort(int[], int, int);

void quickSort (int[], int, int );

void partition (int[], int, int, int, int&, int&);

void exchange(int&, int&);

};

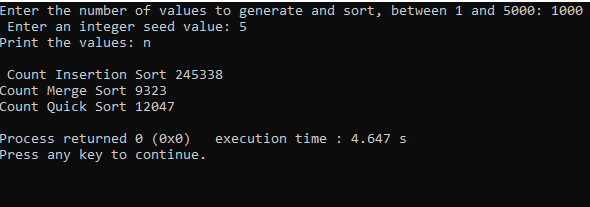
#endif // SORT\_H\_INCLUDED

**User Document**

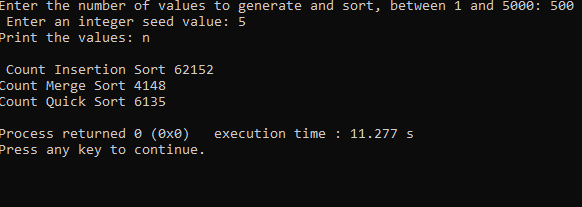
After the completion of code writing, we run the program by compiling it. In code blocks, there is an icon that says build and run. If the program has any errors, it will show up in the bottom of the console, just below the codes. If there are no errors, the program runs. And we get the desired output in the terminal window.

**Tests**

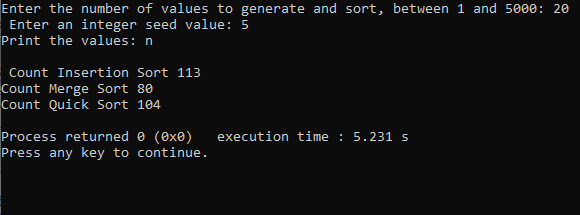
1.



2.



3.



**Summary**

We compared three different sorting. Among them merge sort is the most efficient while working with large data whereas insertion sort is most efficient while working with small data.

Yes, the results confirm our discussions of the algorithms big-o times in classs.