Project 10

Design Document

Introduction

Three sorting methods that we have discussed are insertion sort, merge sort, and quick sort. We have shown in class that the time of insertion sort is O(n2), and that merge sort and insertion sort are both O(n log n). After discussing this theoretically we want to show that this is indeed the case. This program creates and initializes three arrays with the same random values. It then sorts the three arrays using the three methods we have discussed. The functions that sort the arrays count the number of characteristic operations that occur while sorting, and the program displays this information to the console.

Data Structures

There are three data structures in the program – three arrays. The maximum size of the arrays is set by a global constant. The user choses the size of the arrays. The arrays are filled with integers using a pseudo random number generator, and sorted by functions in the program.

Functions

T

Main Program

The main program initializes three arrays, and all of the necessary variables. It prompts for array size, seed value and whether or not to display the values. It calls the three sorting functions, counts the number of characteristic operations and displays them to the console.

Project Code

*project10.cpp* file:

// Created By: John Blee 3 April 2018

// CSCI 301, Section 1, Dr. Julstrom, Spring 2018

// Last Modified: 10 April 2018, 08:31

// Due April 10 2018

// Project 10: Comparing Three Sorting Algorithms

/\* This program sorts three identical arrays that are the size of the users choosing and

are filled with a pseudo random number generator. The three arrays are sorted using insertion sort,

quick sort, and merge sort. The number of characteristic equations is counted during the sorting,

and the functions display the number of characteristic equations for each sort method at the end. \*/

#include <iostream>

using namespace std;

// function prototypes

void printArray(int [], int);

void exchange(int &, int &);

void initialize\_Array(int, int, int[]);

int insertion\_sort(int [], int, int &);

void new\_merge\_step(int [], int, int, int, int &);

int new\_merge\_sort(int [], int, int, int &);

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

int quick\_sort(int[], int, int, int &);

// global constant that represent the maximum size an array can be

const int ARRAYMAX = 5000;

int main()

{

// variable declarations

int numOfValues, seedNum, mergeOperations, insertionOperations, quickOperations;

int mergeArray[ARRAYMAX], quickArray[ARRAYMAX], insertionArray[ARRAYMAX];

char exitChoice, print;

do{

int numOfQuickOperations = 0, numOfInsertionOperations = 0, numOfMergeOperations = 0;

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

cin >> numOfValues;

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

cin >> seedNum;

cout << "Print the values? Y or N:\n";

cin >> print;

// fill each array with the entered parameters

initialize\_Array(numOfValues, seedNum, mergeArray);

initialize\_Array(numOfValues, seedNum, insertionArray);

initialize\_Array(numOfValues, seedNum, quickArray);

// display the unsorted contents of the array if the user chose Y or y

if(print == 'Y' || print == 'y')

{

cout << "Contents of mergeArray: ";

printArray(mergeArray, numOfValues);

cout << "\n\nContents of insertionArray: ";

printArray(insertionArray, numOfValues);

cout << "\n\nContents of quickArray: ";

printArray(quickArray, numOfValues);

}

// call quick sort

quickOperations = quick\_sort(quickArray, 0, numOfValues - 1, numOfQuickOperations);

// show contents of of the sorted quickArray

if(print == 'Y' || print == 'y')

{

cout << "\nContents of quickArray after quick\_sort:\n";

printArray(quickArray, numOfValues);

}

// call insertion sort

insertionOperations = insertion\_sort(insertionArray, numOfValues, numOfInsertionOperations);

// show contents of the sorted insertionArray

if(print == 'Y' || print == 'y')

{

cout << "\nContents of insertionArray after insertion\_sort:\n";

printArray(insertionArray, numOfValues);

}

// call merge sort

cout << "\nCalling merge\_sort\n";

mergeOperations = new\_merge\_sort(mergeArray, 0, numOfValues - 1, numOfMergeOperations);

// show contents of sorted mergeArray

if(print == 'Y' || print == 'y')

{

cout << "\nContents of mergeArray after merge\_sort:\n";

printArray(mergeArray, numOfValues);

}

// display characteristic operation results

cout << "\n\nQuick Sort count = " << quickOperations;

cout << "\nInsertion Sort Count = " << insertionOperations;

cout << "\nMerge Sort Count = " << mergeOperations;

// user enters choice to continue or exit

cout << "\n\nEnter E to exit or any other key to run again: ";

cin >> exitChoice;

// repeat menu unless E or e is entered

} while (exitChoice != 69 && exitChoice != 101);

cout << "END\n";

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: function is passed a non negative integer that represents the size of the array being passed, a

// non negative integer that represents the seed value for the random number generator, and an array

// post-condition: The function fills the given array with random numbers. It returns no value.

void initialize\_Array(int size, int seed, int array[])

{

srand(seed);

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

{

int n = rand() % 1000; //% 4999 + 1;

//cout << n << " ";

array[index] = n;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, a non negative integer that is the size of the array, and

// an integer 'count' which is currently at zero. This counts the number of characteristic operations

// post-condition: The function sorts the given array, counts the number of characteristic operations that occur, and

// returns the number of characteristic operations

int insertion\_sort(int a[], int n, int &count)

{

for(int i = 1; i < n; i++)

{

int j = i - 1;

int temp = a[i];

count++;

while(j >= 0 && temp < a[j])

{

count++;

a[j + 1] = a[j];

j--;;

}

a[j + 1] = temp;

}

return count;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, an integer that represents the lowest index of the array, an

// integer that represents the largest index of the array, and an integer currently set to 0 that

// counts the number of characteristic operations

// post-condition: The function returns the number of characteristic operations that were performed

int new\_merge\_sort(int arr[], int i, int j, int &count)

{

int mid = 0;

if(i < j)

{

mid = (i + j) / 2;

new\_merge\_sort(arr, i, mid, count);

new\_merge\_sort(arr, mid + 1, j, count);

new\_merge\_step(arr, i, mid, j, count);

}

return count;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: This is the merge step of merge sort. The function is passed an array, three integers that

// represent low, mid and high of the array, and a counter

// post-condition: The function sorts the passed array and counts all of the characteristic operations

void new\_merge\_step(int arr[], int i, int mid, int j, int &count)

{

int temp[j];

int l = i;

int r = j;

int m = mid + 1;

int k = l;

while(l <= mid && m <= r)

{

count++;

if(arr[l] <= arr[m])

temp[k++] = arr[l++];

else

temp[k++] = arr[m++];

}

while(l <= mid)

{

count++;

temp[k++] = arr[l++];

}

while(m <= r)

{

count++;

temp[k++] = arr[m++];

}

for(int i1 = i; i1 <= j; i1++)

{

arr[i1] = temp[i1];

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, integers that represent the low and high indices of the array,

// a integer that is the chosen pivot point for the partition step, and a counter that counts the

// number of characteristic operations

// post-condition: The function performs the partition step of quick sort. It moves all values less than 'pivot'

// to the left of pivot and all values greater than 'pivot' to the right of pivot. It returns nothing.

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

{

int lastS1 = low - 1;

int firstU = low;

int firstS3 = high + 1;

while(firstU < firstS3)

{

count++;

if(a[firstU] < pivot)

{

lastS1++;

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

firstU++;

}

else if(a[firstU] == pivot)

firstU++;

else

{

firstS3--;

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

}

}

i = lastS1;

j = firstS3;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, the low and high indices of the array, and a counter

// post-condition: The function sorts the given array using the quick sort method, counts the number of characteristic

// equations, and returns that number as an integer

int quick\_sort(int a[], int low, int high, int &count)

{

int pivot;

int lastS1, firstS3;

if(low < high)

{

pivot = a[(low + high) / 2];

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

quick\_sort(a, low, lastS1, count);

quick\_sort(a, firstS3, high, count);

}

return count;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed two values by reference. They are integers.

// post-condition: The function swaps the two values and returns nothing.

void exchange(int &i, int &j)

{

int temp = i;

i = j;

j = temp;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, and the size of the array.

// post-condition: The function prints the array to the screen, and returns no value.

void printArray(int a[], int n)

{

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

cout << a[i] << " ";

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

User Document

The user will first be prompted for how large they want the arrays to be. They will enter a positive integer. Then the user will be prompted for a seed value. Again they enter a positive integer. Last, the user will be asked if they want the contents of the arrays to be displayed or not. If they enter Y, the contents are displayed, and if they enter N, they will not be displayed. Here is an example:

Enter the number of values to generate and sort, between 1 and 5000: 10

Enter an integer seed value: 482

Print the values? Y

Contents of mergeArray: 974 257 678 169 628 169 974 627 317 588

Contents of insertionArray: 974 257 678 169 628 169 974 627 317 588

Contents of quickArray: 974 257 678 169 628 169 974 627 317 588

Contents of quickArray after quick\_sort:

169 169 257 317 588 627 628 678 974 974

Contents of insertionArray after insertion\_sort:

169 169 257 317 588 627 628 678 974 974

Contents of mergeArray after merge\_sort:

169 169 257 317 588 627 628 678 974 974

Quick Sort count = 28

Insertion Sort Count = 34

Merge Sort Count = 34

Enter E to exit or any other key to run again:

If the user entered ‘N’, then the array contents are not shown:

Enter the number of values to generate and sort, between 1 and 5000: 10

Enter an integer seed value: 482

Print the values? N

Quick Sort count = 28

Insertion Sort Count = 34

Merge Sort Count = 34

Tests

Script started on Tue Apr 10 09:01:22 2018

[?1034hbash-3.2$ ls[K[Kpwd

/Users/johnblee/Desktop/CSCI/CSCI 301/Project\_10

bash-3.2$ ls

Project 10 Report.docx project10\_4\_april\_18\_1624.cpp typescript

a.out project10bookquicksort.cpp ~$oject 10 Report.docx

insertion\_sort.cpp project10generatevalues.cpp ~$proj10data.xlsx

proj10data.xlsx project10testing.cpp

project10.cpp project10x.cpp

bash-3.2$ cat project10.cpp

// Created By: John Blee 3 April 2018

// CSCI 301, Section 1, Dr. Julstrom, Spring 2018

// Last Modified: 10 April 2018, 08:31

// Due April 10 2018

// Project 10: Comparing Three Sorting Algorithms

/\* This program sorts three identical arrays that are the size of the users choosing and

are filled with a pseudo random number generator. The three arrays are sorted using insertion sort,

quick sort, and merge sort. The number of characteristic equations is counted during the sorting,

and the functions display the number of characteristic equations for each sort method at the end. \*/

#include <iostream>

using namespace std;

// function prototypes

void printArray(int [], int);

void exchange(int &, int &);

void initialize\_Array(int, int, int[]);

int insertion\_sort(int [], int, int &);

void new\_merge\_step(int [], int, int, int, int &);

int new\_merge\_sort(int [], int, int, int &);

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

int quick\_sort(int[], int, int, int &);

// global constant that represent the maximum size an array can be

const int ARRAYMAX = 5000;

int main()

{

// variable declarations

int numOfValues, seedNum, mergeOperations, insertionOperations, quickOperations;

int mergeArray[ARRAYMAX], quickArray[ARRAYMAX], insertionArray[ARRAYMAX];

char exitChoice, print;

do{

int numOfQuickOperations = 0, numOfInsertionOperations = 0, numOfMergeOperations = 0;

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

cin >> numOfValues;

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

cin >> seedNum;

cout << "Print the values? Y or N:\n";

cin >> print;

// fill each array with the entered parameters

initialize\_Array(numOfValues, seedNum, mergeArray);

initialize\_Array(numOfValues, seedNum, insertionArray);

initialize\_Array(numOfValues, seedNum, quickArray);

// display the unsorted contents of the array if the user chose Y or y

if(print == 'Y' || print == 'y')

{

cout << "Contents of mergeArray: ";

printArray(mergeArray, numOfValues);

cout << "\n\nContents of insertionArray: ";

printArray(insertionArray, numOfValues);

cout << "\n\nContents of quickArray: ";

printArray(quickArray, numOfValues);

}

// call quick sort

quickOperations = quick\_sort(quickArray, 0, numOfValues - 1, numOfQuickOperations);

// show contents of of the sorted quickArray

if(print == 'Y' || print == 'y')

{

cout << "\nContents of quickArray after quick\_sort:\n";

printArray(quickArray, numOfValues);

}

// call insertion sort

insertionOperations = insertion\_sort(insertionArray, numOfValues, numOfInsertionOperations);

// show contents of the sorted insertionArray

if(print == 'Y' || print == 'y')

{

cout << "\nContents of insertionArray after insertion\_sort:\n";

printArray(insertionArray, numOfValues);

}

// call merge sort

cout << "\nCalling merge\_sort\n";

mergeOperations = new\_merge\_sort(mergeArray, 0, numOfValues - 1, numOfMergeOperations);

// show contents of sorted mergeArray

if(print == 'Y' || print == 'y')

{

cout << "\nContents of mergeArray after merge\_sort:\n";

printArray(mergeArray, numOfValues);

}

// display characteristic operation results

cout << "\n\nQuick Sort count = " << quickOperations;

cout << "\nInsertion Sort Count = " << insertionOperations;

cout << "\nMerge Sort Count = " << mergeOperations;

// user enters choice to continue or exit

cout << "\n\nEnter E to exit or any other key to run again: ";

cin >> exitChoice;

// repeat menu unless E or e is entered

} while (exitChoice != 69 && exitChoice != 101);

cout << "END\n";

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: function is passed a non negative integer that represents the size of the array being passed, a

// non negative integer that represents the seed value for the random number generator, and an array

// post-condition: The function fills the given array with random numbers. It returns no value.

void initialize\_Array(int size, int seed, int array[])

{

srand(seed);

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

{

int n = rand() % 1000; //% 4999 + 1;

//cout << n << " ";

array[index] = n;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, a non negative integer that is the size of the array, and

// an integer 'count' which is currently at zero. This counts the number of characteristic operations

// post-condition: The function sorts the given array, counts the number of characteristic operations that occur, and

// returns the number of characteristic operations

int insertion\_sort(int a[], int n, int &count)

{

for(int i = 1; i < n; i++)

{

int j = i - 1;

int temp = a[i];

count++;

while(j >= 0 && temp < a[j])

{

count++;

a[j + 1] = a[j];

j--;;

}

a[j + 1] = temp;

}

return count;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, an integer that represents the lowest index of the array, an

// integer that represents the largest index of the array, and an integer currently set to 0 that

// counts the number of characteristic operations

// post-condition: The function returns the number of characteristic operations that were performed

int new\_merge\_sort(int arr[], int i, int j, int &count)

{

int mid = 0;

if(i < j)

{

mid = (i + j) / 2;

new\_merge\_sort(arr, i, mid, count);

new\_merge\_sort(arr, mid + 1, j, count);

new\_merge\_step(arr, i, mid, j, count);

}

return count;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: This is the merge step of merge sort. The function is passed an array, three integers that

// represent low, mid and high of the array, and a counter

// post-condition: The function sorts the passed array and counts all of the characteristic operations

void new\_merge\_step(int arr[], int i, int mid, int j, int &count)

{

int temp[j];

int l = i;

int r = j;

int m = mid + 1;

int k = l;

while(l <= mid && m <= r)

{

count++;

if(arr[l] <= arr[m])

temp[k++] = arr[l++];

else

temp[k++] = arr[m++];

}

while(l <= mid)

{

count++;

temp[k++] = arr[l++];

}

while(m <= r)

{

count++;

temp[k++] = arr[m++];

}

for(int i1 = i; i1 <= j; i1++)

{

arr[i1] = temp[i1];

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, integers that represent the low and high indices of the array,

// a integer that is the chosen pivot point for the partition step, and a counter that counts the

// number of characteristic operations

// post-condition: The function performs the partition step of quick sort. It moves all values less than 'pivot'

// to the left of pivot and all values greater than 'pivot' to the right of pivot. It returns nothing.

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

{

int lastS1 = low - 1;

int firstU = low;

int firstS3 = high + 1;

while(firstU < firstS3)

{

count++;

if(a[firstU] < pivot)

{

lastS1++;

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

firstU++;

}

else if(a[firstU] == pivot)

firstU++;

else

{

firstS3--;

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

}

}

i = lastS1;

j = firstS3;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, the low and high indices of the array, and a counter

// post-condition: The function sorts the given array using the quick sort method, counts the number of characteristic

// equations, and returns that number as an integer

int quick\_sort(int a[], int low, int high, int &count)

{

int pivot;

int lastS1, firstS3;

if(low < high)

{

pivot = a[(low + high) / 2];

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

quick\_sort(a, low, lastS1, count);

quick\_sort(a, firstS3, high, count);

}

return count;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed two values by reference. They are integers.

// post-condition: The function swaps the two values and returns nothing.

void exchange(int &i, int &j)

{

int temp = i;

i = j;

j = temp;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// pre-condition: The function is passed an array, and the size of the array.

// post-condition: The function prints the array to the screen, and returns no value.

void printArray(int a[], int n)

{

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

cout << a[i] << " ";

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bash-3.2$

bash-3.2$ ls

Project 10 Report.docx project10\_4\_april\_18\_1624.cpp typescript

a.out project10bookquicksort.cpp ~$oject 10 Report.docx

insertion\_sort.cpp project10generatevalues.cpp ~$proj10data.xlsx

proj10data.xlsx project10testing.cpp

project10.cpp project10x.cpp

bash-3.2$ g++ project10.cpp

bash-3.2$ ./a.out

Enter the number of values to generate and sort, between 1 and 5000: 10

Enter an integer seed value: 321

Print the values? Y or N:

n N

Calling merge\_sort

Quick Sort count = 28

Insertion Sort Count = 29

Merge Sort Count = 34

Enter E to exit or any other key to run again: a

Enter the number of values to generate and sort, between 1 and 5000: 100

Enter an integer seed value: 652

Print the values? Y or N:

N

Calling merge\_sort

Quick Sort count = 740

Insertion Sort Count = 2471

Merge Sort Count = 672

Enter E to exit or any other key to run again: a

Enter the number of values to generate and sort, between 1 and 5000: 1000

Enter an integer seed value: 653443

Print the values? Y or N:

N

Calling merge\_sort

Quick Sort count = 10002

Insertion Sort Count = 249031

Merge Sort Count = 9976

Enter E to exit or any other key to run again: 2000

Enter the number of values to generate and sort, between 1 and 5000: Enter an integer seed value: 1

Print the values? Y or N:

n

Calling merge\_sort

Quick Sort count = 0

Insertion Sort Count = 0

Merge Sort Count = 0

Enter E to exit or any other key to run again: a

Enter the number of values to generate and sort, between 1 and 5000: 3000

Enter an integer seed value: 654

Print the values? Y or N:

n

Calling merge\_sort

Quick Sort count = 37681

Insertion Sort Count = 2226064

Merge Sort Count = 34904

Enter E to exit or any other key to run again: 4000

Enter the number of values to generate and sort, between 1 and 5000: Enter an integer seed value: 1

Print the values? Y or N:

n

Calling merge\_sort

Quick Sort count = 0

Insertion Sort Count = 0

Merge Sort Count = 0

Enter E to exit or any other key to run again: a

Enter the number of values to generate and sort, between 1 and 5000: 4000

Enter an integer seed value: 34213

Print the values? Y or N:

n

Calling merge\_sort

Quick Sort count = 44802

Insertion Sort Count = 3929520

Merge Sort Count = 47904

Enter E to exit or any other key to run again: a

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

Enter an integer seed value: 41324

Print the values? Y or N:

n

Calling merge\_sort

Quick Sort count = 55304

Insertion Sort Count = 6322800

Merge Sort Count = 61808

Enter E to exit or any other key to run again: a

Enter the number of values to generate and sort, between 1 and 5000: 10

Enter an integer seed value: 90987

Print the values? Y or N:

y

Contents of mergeArray: 509 467 801 489 232 930 768 337 872 315

Contents of insertionArray: 509 467 801 489 232 930 768 337 872 315

Contents of quickArray: 509 467 801 489 232 930 768 337 872 315

Contents of quickArray after quick\_sort:

232 315 337 467 489 509 768 801 872 930

Contents of insertionArray after insertion\_sort:

232 315 337 467 489 509 768 801 872 930

Calling merge\_sort

Contents of mergeArray after merge\_sort:

232 315 337 467 489 509 768 801 872 930

Quick Sort count = 33

Insertion Sort Count = 33

Merge Sort Count = 34

Enter E to exit or any other key to run again: a

Enter the number of values to generate and sort, between 1 and 5000: 100

Enter an integer seed value: 5362522

Print the values? Y or N:

Y

Contents of mergeArray: 727 0 456 389 988 53 749 694 379 881 594 623 181 653 431 950 403 731 602 490 447 267 298 512 884 477 217 978 117 467 729 131 641 111 835 965 952 29 265 570 877 75 692 503 216 877 731 635 57 899 307 857 286 897 494 774 608 765 404 647 250 447 909 341 292 146 359 597 334 819 912 492 991 984 576 708 711 120 358 232 394 790 108 690 756 610 870 949 164 320 872 127 688 77 771 134 328 421 943 554

Contents of insertionArray: 727 0 456 389 988 53 749 694 379 881 594 623 181 653 431 950 403 731 602 490 447 267 298 512 884 477 217 978 117 467 729 131 641 111 835 965 952 29 265 570 877 75 692 503 216 877 731 635 57 899 307 857 286 897 494 774 608 765 404 647 250 447 909 341 292 146 359 597 334 819 912 492 991 984 576 708 711 120 358 232 394 790 108 690 756 610 870 949 164 320 872 127 688 77 771 134 328 421 943 554

Contents of quickArray: 727 0 456 389 988 53 749 694 379 881 594 623 181 653 431 950 403 731 602 490 447 267 298 512 884 477 217 978 117 467 729 131 641 111 835 965 952 29 265 570 877 75 692 503 216 877 731 635 57 899 307 857 286 897 494 774 608 765 404 647 250 447 909 341 292 146 359 597 334 819 912 492 991 984 576 708 711 120 358 232 394 790 108 690 756 610 870 949 164 320 872 127 688 77 771 134 328 421 943 554

Contents of quickArray after quick\_sort:

0 29 53 57 75 77 108 111 117 120 127 131 134 146 164 181 216 217 232 250 265 267 286 292 298 307 320 328 334 341 358 359 379 389 394 403 404 421 431 447 447 456 467 477 490 492 494 503 512 554 570 576 594 597 602 608 610 623 635 641 647 653 688 690 692 694 708 711 727 729 731 731 749 756 765 771 774 790 819 835 857 870 872 877 877 881 884 897 899 909 912 943 949 950 952 965 978 984 988 991

Contents of insertionArray after insertion\_sort:

0 29 53 57 75 77 108 111 117 120 127 131 134 146 164 181 216 217 232 250 265 267 286 292 298 307 320 328 334 341 358 359 379 389 394 403 404 421 431 447 447 456 467 477 490 492 494 503 512 554 570 576 594 597 602 608 610 623 635 641 647 653 688 690 692 694 708 711 727 729 731 731 749 756 765 771 774 790 819 835 857 870 872 877 877 881 884 897 899 909 912 943 949 950 952 965 978 984 988 991

Calling merge\_sort

Contents of mergeArray after merge\_sort:

0 29 53 57 75 77 108 111 117 120 127 131 134 146 164 181 216 217 232 250 265 267 286 292 298 307 320 328 334 341 358 359 379 389 394 403 404 421 431 447 447 456 467 477 490 492 494 503 512 554 570 576 594 597 602 608 610 623 635 641 647 653 688 690 692 694 708 711 727 729 731 731 749 756 765 771 774 790 819 835 857 870 872 877 877 881 884 897 899 909 912 943 949 950 952 965 978 984 988 991

Quick Sort count = 667

Insertion Sort Count = 2560

Merge Sort Count = 672

Enter E to exit or any other key to run again: a

Enter the number of values to generate and sort, between 1 and 5000: 3

Enter an integer seed value: 45241

Print the values? Y or N:

Y

Contents of mergeArray: 487 359 545

Contents of insertionArray: 487 359 545

Contents of quickArray: 487 359 545

Contents of quickArray after quick\_sort:

359 487 545

Contents of insertionArray after insertion\_sort:

359 487 545

Calling merge\_sort

Contents of mergeArray after merge\_sort:

359 487 545

Quick Sort count = 5

Insertion Sort Count = 3

Merge Sort Count = 5

Enter E to exit or any other key to run again: e

END

bash-3.2$ exit

exit

Script done on Tue Apr 10 09:03:49 2018

Summary

This program was meant to support what we have discussed in class about times of sorting algorithms. The results show that yes, insertion sort is O(n2) and merge sort and quick sort are O(n log n). Yes, the number of operations varied with the initial arrangement of values.