/\* Assignment: Sorts and Search

Author: Ryan Wood

Created On: March 18, 2018

Purpose: Demonstrate an understanding of common sorting algorithms

and their level of efficiency as well as an understanding

of searching algorithms and their level of efficentcy

through a series of sorting and searching exercises

\*/

#include <iostream>

#include <fstream>

#include <cmath>

#include <time.h>

#include <iomanip>

using namespace std;

ofstream outFile("SearchAndSort.out");

const size\_t SIZE = 55000;

const int space = 8;//space between output chunks

const int MAX\_VAL = 200000;

const int MIN\_VAL = 1;

/\* Function: getRandomInt()

\* Purpose: returns a random integer between MIN and MAX

\* Parameters: min, max

\* Return: a random number

\*/

int getRandomInt(int MIN, int MAX)

{

int random = 0;

random = (rand() % (MAX - MIN + 1)) + MIN;

return random;

}

/\* Function: swap(type&, type&)

Purpose: this function swaps the values of

the two variables of any type

Parameters: the first value, the second value

\*/

template <typename dataType>

void swap(dataType &item1, dataType &item2)

{

dataType tmp;

tmp = item1;

item1 = item2;

item2 = tmp;

}

/\* Function: bubbleSort(type\*, size\_t)

Purpose: using the bubble sort algorithm, this function

sorts the array of items of any datatype, of

the given number of items.

Parameters: a pointer to the first element of the array,

the size of the array

Return: the number of swaps made to sort the array

\*/

template <typename dataType>

int bubbleSort(dataType \*arry, size\_t size)

{

int numSwaps = 0;

bool swapped = false;

dataType \*val = NULL;

dataType tmp;

cout << "Bubble sorting array of size " << size << endl;

outFile << "Bubble sorting array of size " << size << endl;

do

{

swapped = false;

for(val = arry; val <= &arry[SIZE-2]; val++)

{

if(\*val > \*(val+1))

{

tmp = \*val;

\*val = \*(val+1);

\*(val+1) = tmp;

swapped = true;

numSwaps++;

}

}

}while(swapped);

return numSwaps;

}

/\* Function: selectionSort(type\*, size\_t)

Purpose: using the selection sort algorithm, this

function sorts the data in the given array

of the given number of elements

Parameters: a pointer to the first element of the array,

the size of the array

Return: the number of swaps made to sort the array

\*/

template <typename dataType>

int selectionSort(dataType \*arry, size\_t size)

{

int numSwaps = 0;

dataType \*start = NULL;

dataType \*val = NULL;

dataType \*minVal = NULL;

dataType tmp;

cout << "Selection sorting array of size " << size << endl;

outFile << "Selection sorting array of size " << size << endl;

//from the first element to the penultimate

for(start = arry; start <= &arry[SIZE-2]; start++)

{

//arbitrarily set minVal to start

minVal = start;

//now loop through all other elements

for(val = start+1; val <= &arry[SIZE-1]; val++)

{

//if we find a smaller element, store it in minVal

if(\*val < \*minVal)

minVal = val;

}

//swap our start with the smallest value found from start to end

tmp = \*start;

\*start = \*minVal;

\*minVal = tmp;

numSwaps++;

}

return numSwaps;

}

/\* Function: shellSort(type\*, size\_t)

Purpose: using the shell sort algorithm, this function

sorts the data in the given array of any datatype,

having the given number of elements.

Parameters: a pointer to the first element of the array,

the size of the array

Return: the number of swaps made so sort the array

\*/

template <typename dataType>

int shellSort(dataType \*arry, size\_t size)

{

int numSwaps = 0;

int gap = 0;

int j = 0;

int i = 0;

dataType temp = 0;

//Start with a big gap, then reduce the gap

for(gap = size/2; gap > 0; gap /= 2)

{

/\*Do a gapped insertion sort for this gap size.

The first gap elements a[0..gap-1] are already in gapped order

keep adding one more element until the entire array is

gap sorted\*/

for(i = gap; i < size; i++)

{

/\*add a[i] to the elements that have been gap sorted

save a[i] in temp and make a hole at position i\*/

temp = arry[i];

/\*shift earlier gap-sorted elements up until the correct

location for a[i] is found\*/

for(j = i; j >= gap && arry[j - gap] > temp; j -= gap)

{

arry[j] = arry[j - gap];

numSwaps++;

}

//put temp (the original a[i]) in its correct location

arry[j] = temp;

numSwaps++;

}

}

return numSwaps;

}

/\* Function: partition(type[], int, int, int&)

Purpose: this function partitions the data of the given

array to the left or the right of the given split

point depending on whether the data is less than

the data at the split point or greater. When the

function completes execution, all data in the array

will be sorted into one group or the other, greater

or less than the data at the split point

Parameters: the array, the first index, the last index,

the index of the array's split point

Return: the number of swaps made in the data

\*/

template <typename dataType>

int partition(dataType arry[], int first, int last, int &splitPoint)

{

int numSwaps = 0;

int saveFirst = 0;

bool onCorrectSide = false;

dataType splitVal;

dataType tmp;

splitVal = arry[first];

saveFirst = first;

first++;

do

{

onCorrectSide = true;

while(onCorrectSide)

{

if(arry[first] > splitVal)

onCorrectSide = false;

else

{

first++;

onCorrectSide = (first <= last);

}

}

onCorrectSide = (first <= last);

while(onCorrectSide)

{

if(arry[last] <= splitVal)

onCorrectSide = false;

else

{

last--;

onCorrectSide = (first <= last);

}

}

if(first < last)

{

tmp = arry[first];

arry[first] = arry[last];

arry[last] = tmp;

first++;

last--;

}

}while(first <= last);

splitPoint = last;

tmp = arry[saveFirst];

arry[saveFirst] = arry[splitPoint];

arry[splitPoint] = tmp;

numSwaps++;

return numSwaps;

}

/\* Function: quickSort(type[], int, int)

Purpose: using the quick sort algorithm, this function

sorts the given array using the given first and

last indexes. The quick sort is recursive

Parameters: the array of any datatype,

the first index to sort, the last index to sort

Return: the number of swaps made to sort the array

\*/

template<typename dataType>

int quickSort(dataType arry[], int first, int last)

{

int splitPoint = 0;

int numSwaps = 0;

if(first < last)

{

numSwaps += partition(arry, first, last, splitPoint);

numSwaps += quickSort(arry, first, splitPoint-1);

numSwaps += quickSort(arry, splitPoint+1, last);

}

return numSwaps;

}

/\* Function: linearSearch(type\*, size\_t, type, int&)

Purpose: using the linear search algorithm, this function

searches the given array of any data type

of the given size for the given data item. The

array does not need to be in sorted order for the

search to work. The index is populated with the

index of the array at which the data was found if

it is, indeed located. Otherwise, the index is

populated with a -1.

Parameters: a pointer to the first element of the array,

the array size, the data for which to search,

a reference to an index to populate if found

Return: the number of probes to the array,

by reference the index where the item was found or -1

\*/

template<typename dataType>

int linearSearch(dataType \*arry, size\_t size, dataType searchVal, int &index)

{

dataType \*iter = NULL;

bool found = false;

int numProbes = 0;

index = 0;

iter = arry;

while(iter <= &arry[size-1] && !found)

{

numProbes++;

if(\*iter == searchVal)

found = true;

else

{

iter++;

index++;

}

}

if(!found)

index = -1;

return numProbes;

}

/\* Function: binarySearch(type\*, size\_t, type, int&)

Purpose: this function uses the binary search algorithm to find the

given data item in the given array of the given size having

any data type. The array must be in sorted order before calling

this function.

Parameters: a pointer to the first element of the array, the array size,

the data for which to search,

a reference to an index which will be populated with the index

of the item if it is found or -1 if it is not found

Return: the number of probes made to the array to find the data,

by reference the index where the data was found or -1

\*/

template<typename dataType>

int binarySearch(dataType \*arry, size\_t size, dataType searchVal, int &index)

{

int numProbes = 0;

int first = 0;

int last = SIZE-1;

int mid = 0;

bool found = false;

index = 0;

while(!found && first <= last)

{

mid = (first + last) / 2;

if(arry[mid] == searchVal)

found = true;

else if(arry[mid] > searchVal)

last = mid - 1;

else

first = mid + 1;

numProbes++;

}

if(!found)

index = -1;

index = mid;

return numProbes;

}

/\* Function: interpolationSearch(int\*, int, int, int&)

Purpose: this function uses the interpolation search to find the

data passed as the third argument to the array given as

the first argument, which must be of the size given as the

second argument of the array.

Parameters: a pointer to the first element of the array, the size of the array,

the search value, a reference to the index where the search value

Return: the number of probes made to the array to find the data,

by reference the index where the data was found or -1

\*/

int interpolationSearch(int \*arry, int size, int searchVal, int &index)

{

int numProbes = 0;

int lo = 0;

int hi = 0;

int val = 0;

// Find indexes of two corners

lo = 0;

hi = (size - 1);

index = 0;

while(lo <= hi && searchVal >= arry[lo] && searchVal <= arry[hi])

{

//x = l + (((v - a[l]) \* (r - l))/ (a[r] - a[l]))

index = lo + (((double)(searchVal - arry[lo]) \* (hi - lo)) / (arry[hi] - arry[lo]));

numProbes++;

if(arry[index] == searchVal)

return numProbes;

if(arry[index] < searchVal)

lo = index + 1;

else

hi = index - 1;

}

return numProbes;

}

/\* Function: copyArray(type\*, size\_t)

Purpose: this function makes a copy of the data in the given array

of the given size and returns the new array

Parameters: a pointer to the first element of the array,

the size of the array

Return: a pointer to the first element of the copy of the array

\*/

template<typename dataType>

dataType \*copyArray(dataType \*arry, size\_t size)

{

dataType \*newArry = NULL;

dataType \*iterA = NULL;

dataType \*iterB = NULL;

int index = 0;

if(NULL == arry)

return newArry;

//create a new array of the same size as the copy array

if(size > 0)

newArry = new dataType[size];

else

return newArry;

//set up a couple iterators

iterA = arry;

iterB = newArry;

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

{

/\*set the data at new array for the current index to the

data at the copy array for the current index and increment

both of the iterators\*/

\*iterB = \*iterA;

iterB++;

iterA++;

}

return newArry;

}

/\* Function: \*newRandomArray(size\_t, int, int)

Purpose: this function creates a new array having the given

size and having integer values between the given

min and max. The values are randomly generated

Parameters: the array size, the minimum element value,

the maximum element value

Return: the array

\*/

int \*newRandomArray(size\_t size, int MIN, int MAX)

{

int index = 0;

int \*arry = NULL;

int \*iter = NULL;

if(size > 0)

arry = new int[size];

else

return arry;

iter = arry;

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

{

\*iter = getRandomInt(MIN, MAX);

iter++;

}

return arry;

}

/\* Function: print(type\*, size\_t)

Purpose: this function prints the fifty values that are in the

middle portion of the given array of the given size

to the console and the output file

Parmaeters: a pointer to the first element of the array,

the size of the array

\*/

template<typename dataType>

void print(dataType \*arry, size\_t size)

{

int index = 0;

int start = 0;

int end = 0;

dataType \*iter = NULL;

//just print out the middle fifty values

start = size/2;

end = start + 51;

iter = &arry[start];

for(index = start; index < end; index++)

{

if(index %20 == 0)

{

cout << endl;

outFile << endl;

}

cout << left << setw(space) << \*iter;

outFile << left << setw(space) << \*iter;

iter++;

}

cout << endl;

outFile << endl;

}

int main()

{

unsigned t = 0;

int numSwaps = 0;

int \*originalArry = NULL;

int \*sortedArry = NULL;

int randomIndex = 0;

int foundIndex = 0;

int numProbes = 0;

int val = 0;

t = time(NULL);

srand(t);

//generate the original array and make a copy to be sorted

originalArry = newRandomArray(SIZE, MIN\_VAL, MAX\_VAL);

sortedArry = copyArray(originalArry, SIZE);

/\*\*\*\*\* sort the array using the bubble sort algorithm \*\*\*\*\*\*\*/

numSwaps = bubbleSort(sortedArry, SIZE);

print(originalArry, SIZE);

print(sortedArry, SIZE);

cout << "Bubble sort numSwaps = " << numSwaps << endl;

/\*\*\* delete the sorted array, copy the original array,

and sort it using the seleciton sort algorithm \*\*\*\*\*\*\*/

delete sortedArry;

sortedArry = copyArray(originalArry, SIZE);

numSwaps = selectionSort(sortedArry, SIZE);

print(sortedArry, SIZE);

cout << "Selection sort numSwaps = " << numSwaps << endl;

/\*\*\*\* delete the sorted array, copy the orignal and sort

the array using the shell sort algorithm \*\*\*\*\*\*/

delete sortedArry;

sortedArry = copyArray(originalArry, SIZE);

numSwaps = shellSort(sortedArry, SIZE);

print(sortedArry, SIZE);

cout << "Shell sort numSwaps = " << numSwaps << endl;

/\*\*\*\* delete the sorted array, copy the original array,

and sort it using the quick sort algorithm\*\*\*\*/

delete sortedArry;

sortedArry = copyArray(originalArry, SIZE);

numSwaps = quickSort(sortedArry, 0, SIZE-1);

print(sortedArry, SIZE);

cout << "Quick sort numSwaps = " << numSwaps << endl;

//generate a random index between 0 and arry size and retrieve the value

randomIndex = getRandomInt(0, SIZE);

val = originalArry[randomIndex];

/\*\*\*Search for the value using linear search\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

numProbes = linearSearch(originalArry, SIZE, val, foundIndex);

cout << "Searched unsorted arry for " << val << endl;

cout << "The value should have been found at index " << randomIndex

<< " and was found at index " << foundIndex << " after "

<< numProbes << " probes to the array." << endl << endl;

outFile << "Linear Searched unsorted arry for " << val << endl;

outFile << "The value should have been found at index " << randomIndex

<< " and was found at index " << foundIndex << " after "

<< numProbes << " probes to the array." << endl << endl;

/\*\*\*\*\*\*\*Search for the value in sorted array using binary search\*\*\*\*\*\*\*\*/

numProbes = binarySearch(sortedArry, SIZE, val, foundIndex);

cout << "Binary searched sorted arry for " << val << endl

<< "and it was found at index " << foundIndex << " after "

<< numProbes << " probes to the array." << endl << endl;

outFile << "Binary searched sorted arry for " << val << endl

<< "and it was found at index " << foundIndex << " after "

<< numProbes << " probes to the array." << endl << endl;

/\*\*\*\*\*\*Search for the value in sorted array using interpolation search\*\*\*\*\*\*/

numProbes = interpolationSearch(sortedArry, SIZE, val, foundIndex);

cout << "Interpolation searched sorted arry for " << val << endl

<< "and it was found at index " << foundIndex << " after "

<< numProbes << " probes to the array." << endl << endl;

outFile << "Interpolation searched sorted arry for " << val << endl

<< "and it was found at index " << foundIndex << " after "

<< numProbes << " probes to the array." << endl << endl;

delete originalArry;

delete sortedArry;

outFile.close();

return 0;

}

Bubble sorting array of size 55000

3957 40615 134579 199589 75532 78652 152331 75784 111545 69468 149358 26150 30415 2198 35013 138668 157688 88184 178069 45760

175505 85753 59551 167977 42309 76060 50643 45764 35913 123914 3375 117179 172714 50679 194510 178630 94839 16084 131775 171087

13907 110730 133746 47728 55106 165107 165608 136344 170044 102564 127587

98866 98873 98876 98892 98893 98895 98903 98909 98911 98913 98915 98921 98932 98937 98940 98940 98942 98942 98945 98945

98960 98961 98966 98967 98967 98967 98967 98969 98971 98975 98976 98981 98981 98983 98985 98987 98988 98990 98991 98994

98999 99001 99004 99004 99010 99010 99012 99016 99019 99019 99023

Selection sorting array of size 55000

98866 98873 98876 98892 98893 98895 98903 98909 98911 98913 98915 98921 98932 98937 98940 98940 98942 98942 98945 98945

98960 98961 98966 98967 98967 98967 98967 98969 98971 98975 98976 98981 98981 98983 98985 98987 98988 98990 98991 98994

98999 99001 99004 99004 99010 99010 99012 99016 99019 99019 99023

98866 98873 98876 98892 98893 98895 98903 98909 98911 98913 98915 98921 98932 98937 98940 98940 98942 98942 98945 98945

98960 98961 98966 98967 98967 98967 98967 98969 98971 98975 98976 98981 98981 98983 98985 98987 98988 98990 98991 98994

98999 99001 99004 99004 99010 99010 99012 99016 99019 99019 99023

98866 98873 98876 98892 98893 98895 98903 98909 98911 98913 98915 98921 98932 98937 98940 98940 98942 98942 98945 98945

98960 98961 98966 98967 98967 98967 98967 98969 98971 98975 98976 98981 98981 98983 98985 98987 98988 98990 98991 98994

98999 99001 99004 99004 99010 99010 99012 99016 99019 99019 99023

Linear Searched unsorted arry for 29294

The value should have been found at index 47962 and was found at index 47962 after 47963 probes to the array.

Binary searched sorted arry for 29294

and it was found at index 8113 after 16 probes to the array.

Interpolation searched sorted arry for 29294

and it was found at index 8113 after 3 probes to the array.