Tyler Bartlett
01/27/2019
Dr. Breeann Flesch
CS361 – Algorithms

Lab 1 Report

This was the first assignment I have ever done in C++ and I wanted to push myself to learn the language while doing this first lab. I have written ANSI C before though, so it is a little familiar already, and is the reason why many of my functions return an int, namely zero, to indicate function success.

I started by wring the code to read in a text file. I made a simple test file to use in place of the data file provided for this lab while writing and testing all my functions. It is eleven lines long and included the values: 12, 15, 123, 20, 100, 200, 50, and 10. I had a couple empty lines in between a few data points so I could prepare my file reader function to skip them. I originally had written the function to read in the entire file and later modified it to read in a variable amount of lines by adding an additional parameter to the function and adding an additional check in the while loop to read n lines.

This is my function to read in data from a text file.

```
//read the input fine into a vector<int>
int readFileToArray(vector<int>& arr, int numLinesToRead)
{
    ifstream infile; // create a file object
    infile.open (DATA_FILE);

    string line; // for reading lines in from file
    int intFromLine; // int to hold converted line string
    int numLinesRead = 1; //start reading at line 1 of file
    double sum = 0;

    //if the file is open, read it into the array
    if (infile.is_open())
    {
        while(getline(infile, line) && numLinesRead <= numLinesToRead)
        {
            //skip blank lines
            if (line.size() != 0)
            {
                intFromLine = stoi (line); // stoi converts string to int
                arr.push_back(intFromLine); // push_back adds element to next available spot in vector
                sum += intFromLine;
            }
            numLinesRead++;
        }
    }
    infile.close();
    cout << fixed << "sum from file is: " << sum;

    return 0;
}
```

I checked the sum of all the data in the file provided, and it was 49,999,995,000,000, which matched the provided sum in the lab instructions.

This is where I call the readFile function in main, and how I timed the output. numLinesToRead is declared in main prior to this function call and was changed from 1,000 by 10x until it was equal to 10 million lines to read in.

```
/* read file into vector and time function duration */
start = clock();
readFileToArray(dataFileArr, numLinesToRead);
end = clock();
algoRunTime = (float)(end - start) / CLOCKS_PER_SEC;
cout << "time to complete File read: " << algoRunTime << '\n';
cout << "data read in from file:\n";
vectorPrinter(dataFileArr);
```

I verified that the function worked by printing the contents of the read-in array, and as we see below, the printed values matched the values of my test file, in order they were read, without including the empty lines.

```
data read in from file:
12
15
123
20
100
200
50
10
```

After my readFile function was done, I started working on the merge sort. I used pseudo code from the text book to write the merge sort function and the merge sort helper function, which has code snippets on the following page.

```
//recursive function to merge sort an array.
int auxMergeSort(vector<int>& arr, int startIndex, int endIndex)
{
    if (startIndex < endIndex)
    {
        int mid = (startIndex + endIndex)/2; //get the middleIndex

        auxMergeSort(arr, startIndex, mid); //break array in half from startIndex to midIndex
        auxMergeSort(arr, (mid+1), endIndex); //break array in half from midIndex to endIndex
        mergeHelper(arr, startIndex, mid, endIndex); //sort the array
    }

    return 0;
}
```

The two following code snippets are from my merge sort helper function, which receives an address to a vector, a startIndex, a midIndex and an endIndex values. I copy the contents of the passed in vector into two sub-vectors and set the sub-vectors next available space to a very large number, which is necessary in the logic of reassembling the pieces into a sorted vector.

I was having some trouble with this function initially because I did not include the step of adding a large number to next available spots in the two sub-vectors, which Breeann helped my fix after class.

```
//copy subarray arr[startIndex..midIndex] into 'left side' arr
for (int i = 0; i <= num1; i++)
{
    left.push_back(arr[startIndex + i]);
}
//copy subarray arr[midIndex+1..endIndex] into 'right side' arr
for (int j = 1; j <= num2; j++)
{
    right.push_back(arr[midIndex + j]);
}

//setting left and right [numX + 1] to a large number to correct for comparison logic in reassemly
left.push_back(9999999);
right.push_back(9999999);
```

```
// begin reassembling the array in sorted order
int i = 0;
int j = 0;
for (int k = startIndex; k <= endIndex; k++)
{
    if (left[i] <= right[j])
    {
        arr[k] = left[i];
        i++;
    }
    else
    {
        arr[k] = right[j];
        j++;
    }
}
```

This is where I call the merge sort function in main, and how I timed the output.

```
/* recursive merge sort vector and time function duration */
start = clock();
auxMergeSort(dataFileArr, 0, dataFileArr.size()-1);
end = clock();
algoRunTime = (float)(end - start) / CLOCKS_PER_SEC;
cout << "time to complete merge sort: " << algoRunTime << '\n';
cout << "Array after merge sort:\n";
vectorPrinter(dataFileArr);
```

I verified that the function worked by printing the contents of the sorted array, and as we see in the photo below, the array is sorted properly.

```
Array after merge sort:
10
12
15
20
50
100
123
200
```

The quick sort algorithm was quick to setup and get working. I based my function off the pseudo code in the book.

```
//equivalent to quick sort psuedo code from textbook
int auxQuickSort (vector<int>& arr, int startIndex, int endIndex)
{
    if (startIndex < endIndex)
    {
        int q = qsHelper(arr, startIndex, endIndex);
        auxQuickSort (arr, startIndex, q-1); //sort from startIndex to q
        auxQuickSort (arr, q+1, endIndex); //sort from q to endIndex
    }

    return 0;
}
```

```
//this is equivalent to the partition function from the textbook psuedo code
int qsHelper(vector<int>& arr, int startIndex, int endIndex)
{
    int x = arr[endIndex]; //the pivot for quick sort
    int i = startIndex-1;

    for (int j = startIndex; j < endIndex; j++)
    {
        //if value is less than the pivot
        if (arr[j] <= x)
        {
            i++;
            swap(arr, i, j);
        }
    }

    swap(arr, i+1, endIndex);//swap the pivot

    return i+1;
}
```

The following snippet is where I call the quick sort function in main.

```
/* recursively quick sort vector and time function duration */
start = clock();
auxQuickSort(dataFileArrCopy, 0, dataFileArr.size()-1);
end = clock();
algoRunTime = (float)(end - start) / CLOCKS_PER_SEC;
cout << "time to complete quick sort: " << algoRunTime << '\n';
cout << "Array after quick sort:\n";
vectorPrinter(dataFileArrCopy);
```

I verified that the function worked by printing the contents of the sorted array, and as we see in the photo below, the array is sorted properly.

```
Array after quick sort:
10
12
15
20
50
100
123
200
```

I was having quite a bit of trouble with the flgIsSorted function. I set it up to take only one parameter, the vector, and run through it to confirm data is sorted. I set it up to copy the contents of the passed in vector into a first half, second half vectors, ran quick sort on those two halves, then compared the vector passed into the function against the two halves. I then checked if the last element of the first half was larger than the first element of the 2nd half to ensure the array was sorted in ascending order. I had some difficulty with a bug not making this work with my test data, so I had talked to Mike online and got some help from him debugging. This function was not recursive as I did not see that it needed to be in the instructions, and when testing the function, it caused my program to crash when trying to verify the 100,000 data values were sorted properly. Bellow are some snippets of the broken function.

```
//copy array data over into both half arrays
for (int i = 0; i < arr.size(); i++)
{
    if (i < arr.size()/2)
    {
        sortedArrHalf1.push_back(arr[i]);
        lastEleHalf1 = arr[i];
    }
    else
    {
        sortedArrHalf2.push_back(arr[i]);

        if (setFirstEleHalf2 == true)
        {
            firstEleHalf2 = arr[i];
            setFirstEleHalf2 = false;
        }
    }
}
```

```
//check that arr passed into this function is sorted by comparing against the first half
for (int i = 0; i < sortedArrHalf1.size(); i++)
{
    if (sortedArrHalf1[i] != arr[i])
    {
        cout << "arr half1 not sorted \n";
        return false;
    }
}

//to fix logic in comparisons of 2nd half against second half of arr
int evenOddAdjuster = 0;
if (arr.size()%2 != 0)
{
    evenOddAdjuster++;
}

//check that arr passed into this function is sorted by comparing against the second half
for (int i = 0; i < sortedArrHalf2.size(); i++)
{
    if (sortedArrHalf2[i] != arr[i+ sortedArrHalf2.size()-evenOddAdjuster])
    {
        cout << "arr half2 not sorted \n";
        return false;
    }
}
```

```
//arr passed in not sorted if first ele of 2nd half < last ele of 1st half
if (firstEleHalf2 < lastEleHalf1)
{
   cout << "arr is not sorted! \n";
    return false;
}
else
{
   cout << "arr is sorted! \n";
    return true;
}
```

After I had found that this function breaks at 100,000 data values, I quickly rewrote another function to verify if the data was sorted in ascending order. I was struggling to figure out how to do this recursively with only one parameter on the flgIsSorted function and while working with vectors, so for the sake of time, it too is an iterative function. Below is a snippet of the new flgIsSorted function.

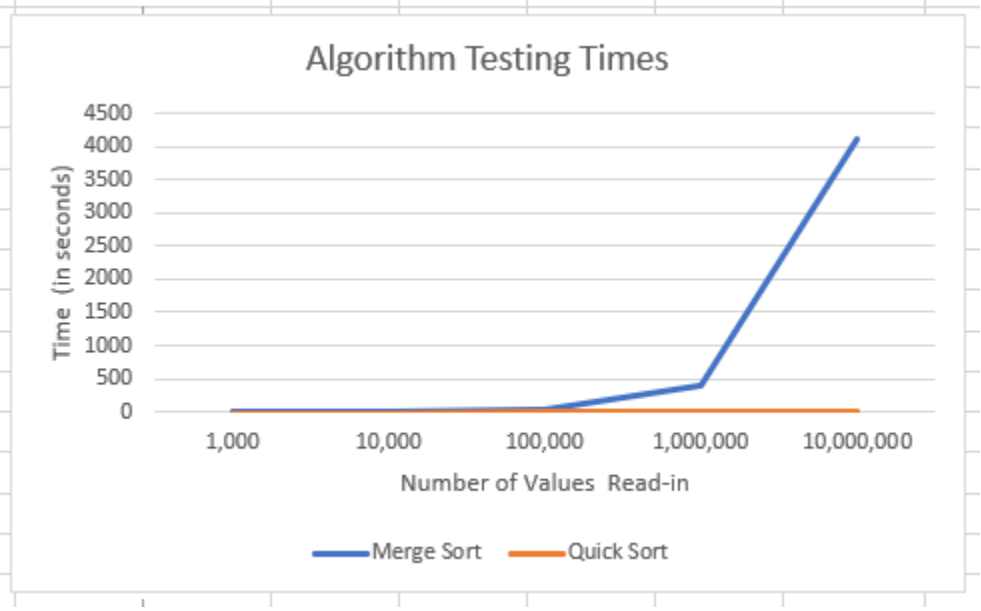
```cpp
//iteratively check if array is sorted in ascending order
bool flgIsSorted2(vector<int>& arr)
{
    //array size one is sorted
    if (arr.size() == 1)
    {
        cout << "arr is sorted! \n";
        return true;
    }
    //array size greater than 1
    else
    {
        for (int i = 0; i < arr.size()-1; i++)
        {
            //checks that arr[i] is less than the next value
            if (arr[i] > arr[i+1])
            {
                cout << "arr is not sorted! \n";
                return false;
            }
        }
    }

    cout << "arr is sorted! \n";
    return true;
}
```

Now onto the data recording:

The time taken for the quick sort algorithm is drastically less at every interval of amount of lines read in. even at 10 million records, my average quick sort run time is 2.6 seconds in length, which is drastically better than the aver of 4,100 seconds to complete a merge sort. This is why the line for quick sort on the graph appears flat.

	A	B	C	D	E	F	G
1	Lines Read:	Run Number:	time (seconds) taken to read file:	time (seconds) taken on merge sort:	time (seconds) taken to verify sort after merge sort:	time (seconds) taken on quick sort:	time (seconds) taken to verify sort after quick sort:
2	1000	Run 1:	0.001	0.424	0	0	0
3		Run 2:	0.001	0.414	0	0	0
4		Run 3:	0.001	0.448	0	0	0
5		Average:	0.001	0.428666667	0	0	0
6	10000	Run 1:	0.003	4.241	0	0.002	0
7		Run 2:	0.001	4.241	0	0.001	0
8		Run 3:	0.002	4.162	0	0.001	0
9		Average:	0.002	4.214666667	0	0.001333333	0
10	100000	Run 1:	0.015	40.568	0	0.02	0
11		Run 2:	0.014	40.606	0	0.02	0
12		Run 3:	0.015	40.343	0.001	0.019	0.001
13		Average:	0.014666667	40.50566667	0.000333333	0.019666667	0.000333333
14	1000000	Run 1:	0.147	405.512	0.004	0.221	0.005
15		Run 2:	0.145	405.526	0.005	0.222	0.004
16		Run 3:	0.145	405.006	0.004	0.222	0.004
17		Average:	0.145666667	405.348	0.004333333	0.221666667	0.004333333
18	10000000	Run 1:	1.497	4072.98	0.045	2.669	0.045
19		Run 2:	1.47	4132.35	0.045	2.669	0.044
20		Run 3:	1.462	4096.66	0.045	2.664	0.045
21		Average:	1.476333333	4100.663333	0.045	2.667333333	0.044666667



The following series of screen dups shows a printout that the array was verified to be in ascending order.

1,000:

```
run 1:
time to complete File read: 0
time to complete merge sort: 0.425
arr is sorted!
time to complete sort check: 0
time to complete quick sort: 0
arr is sorted!
time to complete sort check: 0

run 2:
time to complete File read: 0.001
time to complete merge sort: 0.437
arr is sorted!
time to complete sort check: 0
time to complete quick sort: 0
arr is sorted!
time to complete sort check: 0

run 3:
time to complete File read: 0
time to complete merge sort: 0.409
arr is sorted!
time to complete sort check: 0
time to complete quick sort: 0.001
arr is sorted!
time to complete sort check: 0

done!

[Done] exited with code=0 in 1.71 seconds
```

10,000:

```
run 1:
time to complete File read: 0.002
time to complete merge sort: 4.113
arr is sorted!
time to complete sort check: 0
time to complete quick sort: 0.002
arr is sorted!
time to complete sort check: 0

run 2:
time to complete File read: 0.002
time to complete merge sort: 4.077
arr is sorted!
time to complete sort check: 0
time to complete quick sort: 0.002
arr is sorted!
time to complete sort check: 0

run 3:
time to complete File read: 0.001
time to complete merge sort: 4.078
arr is sorted!
time to complete sort check: 0
time to complete quick sort: 0.001
arr is sorted!
time to complete sort check: 0

done!

[Done] exited with code=0 in 12.696 seconds
```

100,000:

```
run 1:
time to complete File read: 0.015
time to complete merge sort: 40.568
arr is sorted!
time to complete sort check: 0
time to complete quick sort: 0.02
arr is sorted!
time to complete sort check: 0

run 2:
time to complete File read: 0.014
time to complete merge sort: 40.606
arr is sorted!
time to complete sort check: 0
time to complete quick sort: 0.02
arr is sorted!
time to complete sort check: 0

run 3:
time to complete File read: 0.015
time to complete merge sort: 40.343
arr is sorted!
time to complete sort check: 0.001
time to complete quick sort: 0.019
arr is sorted!
time to complete sort check: 0.001

done!

[Done] exited with code=0 in 122.063 seconds
```

1,000,000:

```
run 1:
time to complete File read: 0.147
time to complete merge sort: 405.512
arr is sorted!
time to complete sort check: 0.004
time to complete quick sort: 0.221
arr is sorted!
time to complete sort check: 0.005

run 2:
time to complete File read: 0.145
time to complete merge sort: 405.526
arr is sorted!
time to complete sort check: 0.005
time to complete quick sort: 0.222
arr is sorted!
time to complete sort check: 0.004

run 3:
time to complete File read: 0.145
time to complete merge sort: 405.006
arr is sorted!
time to complete sort check: 0.004
time to complete quick sort: 0.222
arr is sorted!
time to complete sort check: 0.004

done!

[Done] exited with code=0 in 1217.613 seconds
```

10,000,000:

```
run 1:
time to complete File read: 1.497
time to complete merge sort: 4072.98
arr is sorted!
time to complete sort check: 0.045
time to complete quick sort: 2.669
arr is sorted!
time to complete sort check: 0.045

run 2:
time to complete File read: 1.47
time to complete merge sort: 4132.35
arr is sorted!
time to complete sort check: 0.045
time to complete quick sort: 2.669
arr is sorted!
time to complete sort check: 0.044

run 3:
time to complete File read: 1.462
time to complete merge sort: 4096.66
arr is sorted!
time to complete sort check: 0.045
time to complete quick sort: 2.664
arr is sorted!
time to complete sort check: 0.045

done!

[Done] exited with code=0 in 12315.173 seconds
```