Ben Betts
ECE 218

Lab 2 Lab Report

Compiling Instructions:
This code will compile very easily. Make sure you have the following files together in the same directory:

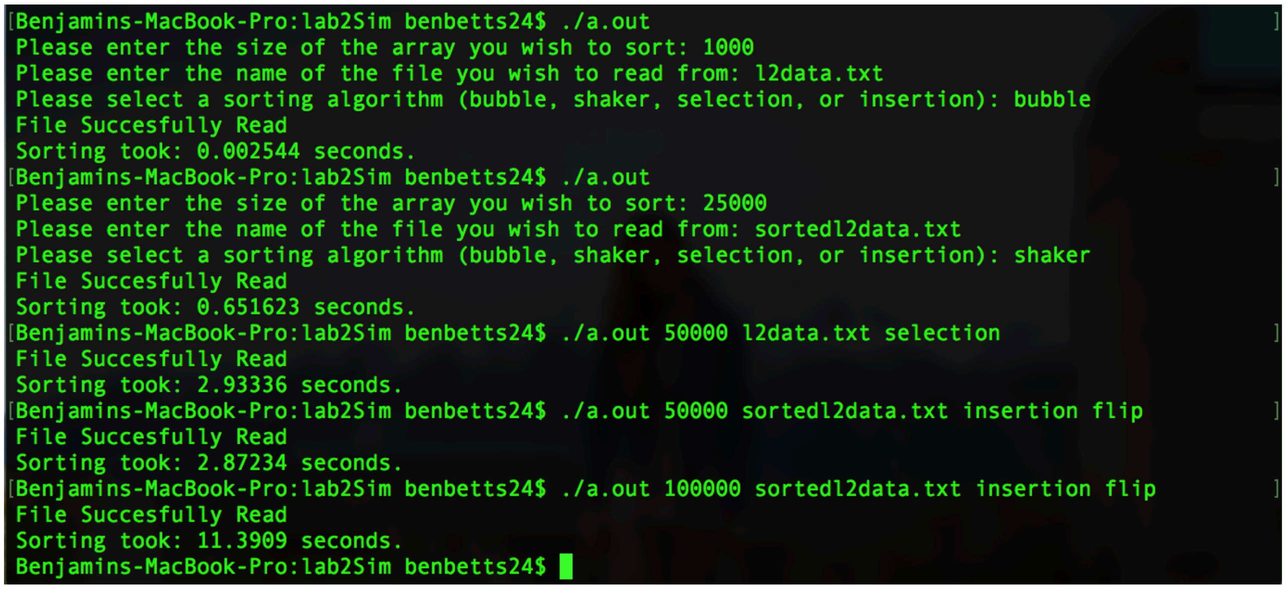
- main.cpp
- support.h
- support.cpp
- a list of integers to sort (not required to compile)

With the files listed together, type “g++ main.cpp support.cpp”. This will compile into a.out, the new executable file.

In this program I implemented command line input to save myself time while testing the algorithms. Using it is completely optional. Running without cause the program to ask for your array size, the name of the filename to be sorted, and the algorithm to use. If you wish to use command line input, simply type those values in in the same order. For example: “a.out 10000 l2data.txt insertion” would sort an array of 10,000 integers from the file l2data.txt with the insertion algorithm. An additional feature of the command line input is telling the algorithms to sort biggest to smallest instead of smallest to biggest. To do this, simply type anything at all after the algorithm. I usually type flip.

Ben Betts
ECE 218

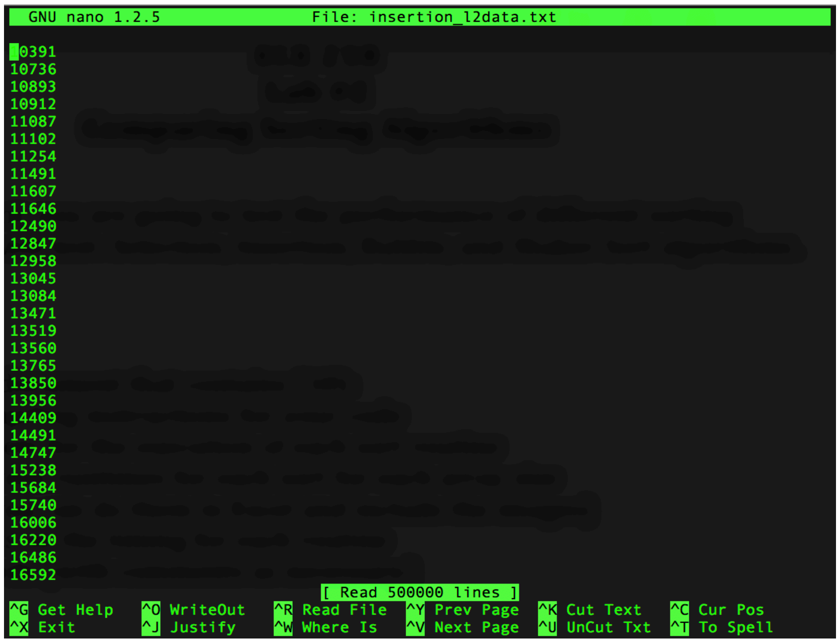
Screenshot of Implementations:



```
Benjamins-MacBook-Pro:lab2Sim benbetts24$ ./a.out
Please enter the size of the array you wish to sort: 1000
Please enter the name of the file you wish to read from: l2data.txt
Please select a sorting algorithm (bubble, shaker, selection, or insertion): bubble
File Succesfully Read
Sorting took: 0.002544 seconds.
Benjamins-MacBook-Pro:lab2Sim benbetts24$ ./a.out
Please enter the size of the array you wish to sort: 25000
Please enter the name of the file you wish to read from: sortedl2data.txt
Please select a sorting algorithm (bubble, shaker, selection, or insertion): shaker
File Succesfully Read
Sorting took: 0.651623 seconds.
Benjamins-MacBook-Pro:lab2Sim benbetts24$ ./a.out 50000 l2data.txt selection
File Succesfully Read
Sorting took: 2.93336 seconds.
Benjamins-MacBook-Pro:lab2Sim benbetts24$ ./a.out 50000 sortedl2data.txt insertion flip
File Succesfully Read
Sorting took: 2.87234 seconds.
Benjamins-MacBook-Pro:lab2Sim benbetts24$ ./a.out 100000 sortedl2data.txt insertion flip
File Succesfully Read
Sorting took: 11.3909 seconds.
Benjamins-MacBook-Pro:lab2Sim benbetts24$
```

This screenshot demonstrates every feature of the code (except for file writing):

- The first command is a test of the bubble algorithm sorting 1000 integers from l2data.txt
- Next is the shaker command sorting 25,000 integers from sortedl2data.txt
- Next is a demonstration of the command line input, using the selection sort algorithm to sort 50,000 integers from the file l2data.txt
- Next is another use of command line input, using the insertion algorithm to reverse the order of the sortedl2data.txt file for the first 50,000 integers
- Last is a repeat of the previous, except for 100,000 integers



```
  GNU nano 1.2.5                 File: insertion_l2data.txt

10391
10736
10893
10912
11087
11102
11254
11491
11607
11646
12490
12847
12958
13045
13084
13471
13519
13560
13765
13850
13956
14409
14491
14747
15238
15684
15740
16006
16220
16486
16592
                                [ Read 500000 lines ]
^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Txt  ^T To Spell
```

This screenshot shows one of the files written by running the algorithms. This is the list of 500,000 integers sorted from l2data.txt by the insertion algorithm, as you can see in the filename listed at the top.

Ben Betts
ECE 218

Tables of Runtimes:

Sorting Unsorted Data							
	1000	**10000**	**25000**	**50000**	**100000**	**250000**	**500000**
1	0.001819	0.249800	1.600810	6.603420	26.148500	164.749000	649.682000
2	0.001777	0.246188	1.602730	6.603950	25.914900	162.484000	648.612000
3	0.001716	0.244667	1.611030	6.617140	26.060100	162.445000	648.314000
Bubble	**0.001771**	**0.246885**	**1.604857**	**6.608170**	**26.041167**	**163.226000**	**648.869333**
1	0.001796	0.188705	1.229120	5.040880	20.423000	127.898000	511.965000
2	0.001640	0.189350	1.264830	5.054080	20.446200	128.008000	511.928000
3	0.001551	0.189447	1.238520	5.054060	20.446400	128.128000	511.933000
Shaker	**0.001662**	**0.189167**	**1.244157**	**5.049673**	**20.438533**	**128.011333**	**511.942000**
1	0.001447	0.116984	0.729601	2.933680	11.642900	72.788600	291.160000
2	0.001220	0.117913	0.728284	2.917850	11.637900	72.774400	291.572000
3	0.001220	0.117072	0.728307	2.915620	11.650200	72.780700	291.124000
Selection	**0.001296**	**0.117323**	**0.728731**	**2.922383**	**11.643667**	**72.781233**	**291.285333**
1	0.000641	0.056206	0.353532	1.427830	5.647500	35.361400	141.775000
2	0.000568	0.056135	0.353057	1.431310	5.657500	35.303000	141.619000
3	0.000569	0.055952	0.352348	1.417730	5.655790	35.412700	141.468000
Insertion	**0.000593**	**0.056098**	**0.352979**	**1.425623**	**5.653597**	**35.359033**	**141.620667**

This table lists the data from every test that sorted unsorted data. Each algorithm is listed on the left, and the size of the array sorted is on the top. Each algorithm was tested three times for each size (shown as rows 1, 2, and 3), and then the average was taken (shown in bold).

It is important to note that all of these tests were run on my personal MacBook pro. I started on rabbit but discovered that my computer completed these sorts in often less than half of the time rabbit took.

Sorting Sorted Data							
	1000	**10000**	**25000**	**50000**	**100000**	**250000**	**500000**
1	0.001341	0.103927	0.657139	2.648920	10.527600	65.875200	263.421000
2	0.001058	0.103580	0.660816	2.643120	10.569600	65.831600	263.292000
3	0.001064	0.103636	0.657057	2.640930	10.543400	65.793600	263.506000
Bubble	**0.001154**	**0.103714**	**0.658337**	**2.644323**	**10.546867**	**65.833467**	**263.406333**
1	0.001064	0.102861	0.646804	2.588710	10.353700	64.734600	259.042000
2	0.001036	0.103242	0.647550	2.586370	10.353700	64.724100	259.036000
3	0.001061	0.103176	0.646257	2.602300	10.343100	64.738400	259.149000
Shaker	**0.001054**	**0.103093**	**0.646870**	**2.592460**	**10.350167**	**64.732367**	**259.075667**
1	0.001360	0.116561	0.735564	2.911450	11.650900	72.763800	291.090000
2	0.001171	0.116308	0.726542	2.922950	11.641100	72.711900	291.170000
3	0.001171	0.116453	0.727135	2.912270	11.636100	72.798300	291.252000
Selection	**0.001234**	**0.116441**	**0.729747**	**2.915557**	**11.642700**	**72.758000**	**291.170667**
1	0.000006	0.000049	0.000097	0.000191	0.000381	0.000948	0.001923
2	0.000006	0.000040	0.000096	0.000191	0.000380	0.000947	0.001892
3	0.000005	0.000039	0.000096	0.000191	0.000381	0.000947	0.001896
Insertion	**0.000006**	**0.000043**	**0.000096**	**0.000191**	**0.000381**	**0.000947**	**0.001904**

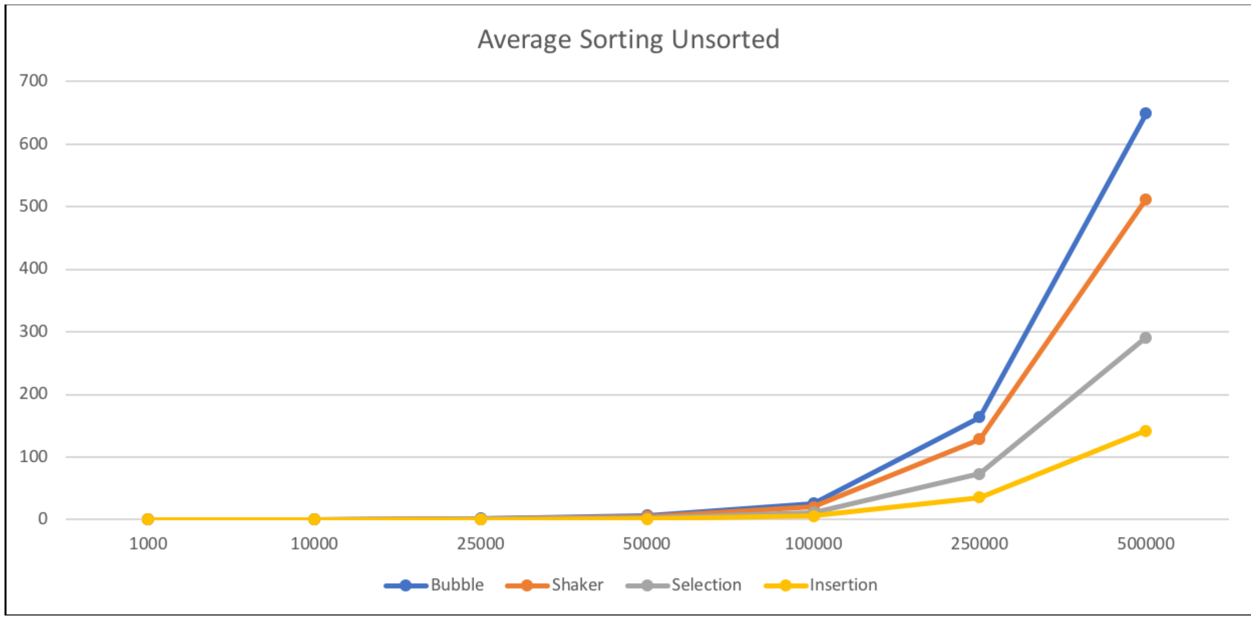
This table lists the data from every test that sorted already sorted data. The layout is exactly the same as above.

Ben Betts
ECE 218

Reversing Sorted Data							
	1000	**10000**	**25000**	**50000**	**100000**	**250000**	**500000**
1	0.001992	0.172567	1.083520	4.372930	17.466600	109.227000	436.718000
2	0.001715	0.172374	1.088040	4.343190	17.521600	109.134000	436.846000
3	0.001716	0.171988	1.084650	4.353070	17.411300	109.089000	436.888000
Bubble	**0.001808**	**0.172310**	**1.085403**	**4.356397**	**17.466500**	**109.150000**	**436.817333**
1	0.001876	0.160437	1.007990	4.039040	16.247100	101.669000	407.346000
2	0.001616	0.160504	1.010730	4.047980	16.250900	101.661000	407.086000
3	0.001616	0.161383	1.010020	4.058890	16.233900	101.632000	407.612000
Shaker	**0.001703**	**0.160775**	**1.009580**	**4.048637**	**16.243967**	**101.654000**	**407.348000**
1	0.001327	0.110228	0.690865	2.778920	11.077900	69.281400	277.098000
2	0.001113	0.110219	0.690354	2.767130	11.077300	69.319900	277.143000
3	0.001117	0.110154	0.698908	2.770780	11.076500	69.278900	277.084000
Selection	**0.001186**	**0.110200**	**0.693376**	**2.772277**	**11.077233**	**69.293400**	**277.108333**
1	0.001245	0.112067	0.702969	2.822510	11.299800	70.822100	283.323000
2	0.001126	0.112117	0.703218	2.844480	11.305100	70.722000	283.007000
3	0.001184	0.112520	0.707052	2.827350	11.295100	70.662200	283.377000
Insertion	**0.001185**	**0.112235**	**0.704413**	**2.831447**	**11.300000**	**70.735433**	**283.235667**

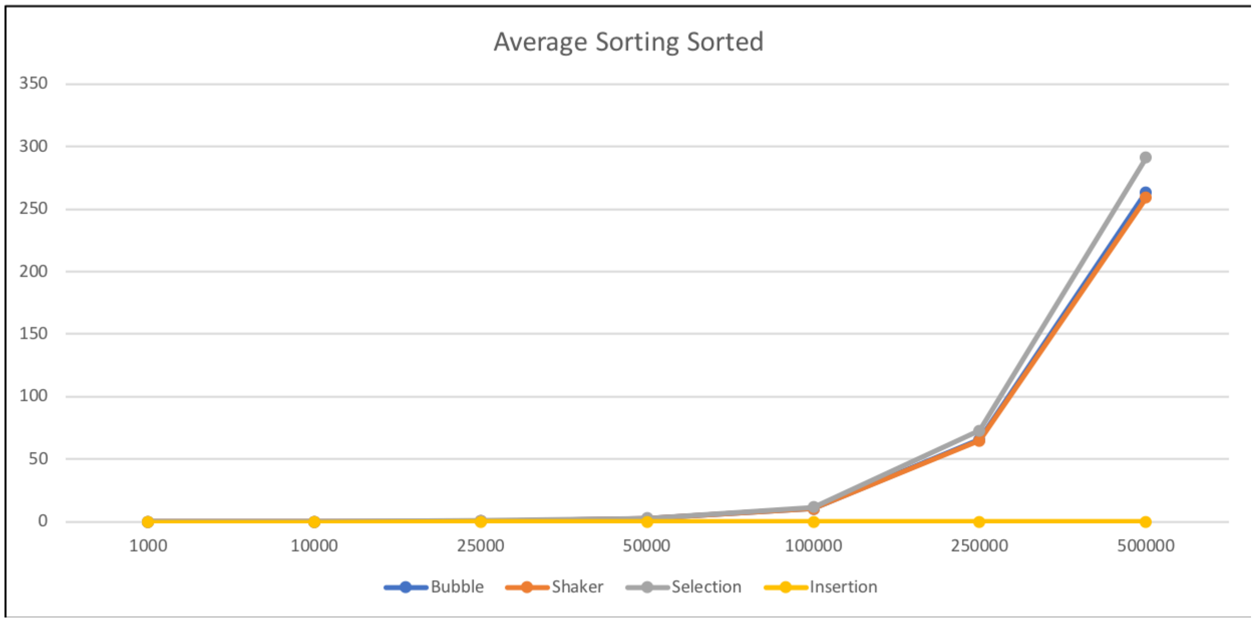
This table lists the data from every test that reversed already sorted data. The layout is once again the same as the previous two tables.

Graphs:

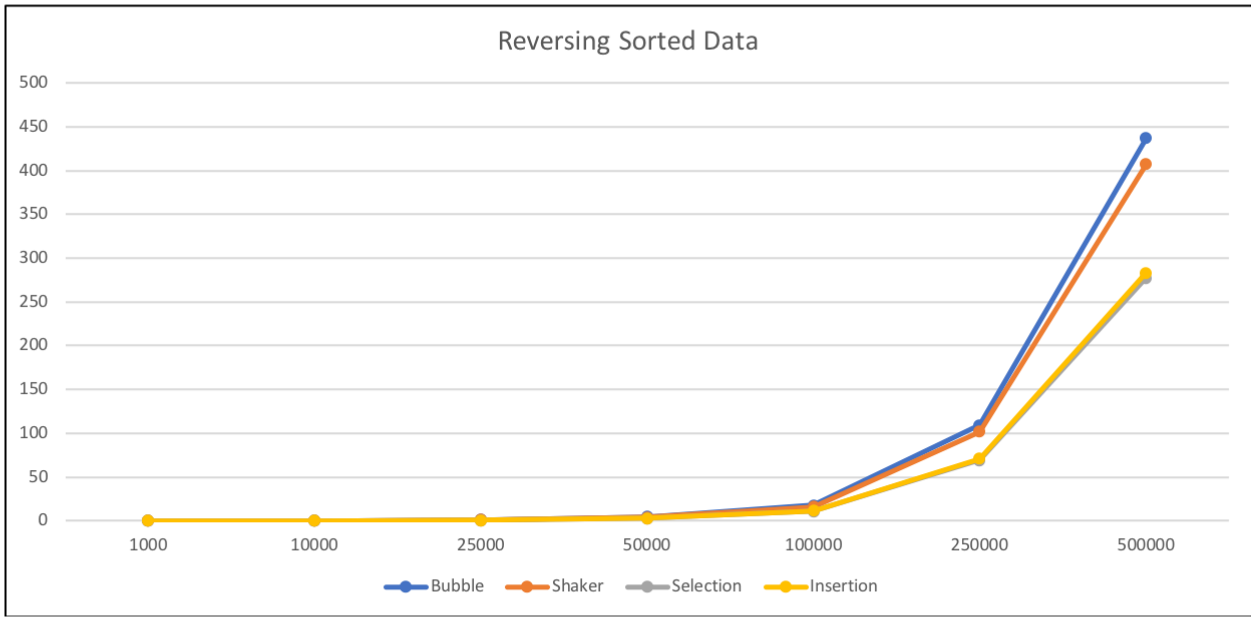


This graph shows the results of sorting unsorted data with each algorithm. The layout is fairly self explanatory. The y axis is the number of seconds taken to sort, and the x axis is the size of the array sorted. Each line represents a different algorithm as labeled at the bottom. Note that the values used for each array size was the average of the three sorts recorded above.

Ben Betts
ECE 218



This graph shows the results of sorting already sorted data with each algorithm, with the same layout as above.



This graph shows the results of reversing already sorted data with each algorithm, with the same layout as above.

Ben Betts
ECE 218

Comments:

There were several surprises once I had finished tabulating the results of the sorting. In fact, there were several things that I double and triple checked just to make sure they were correct.

Let's start with the obvious:

The bubble sort is almost always the slowest, followed by the shaker sort. These two hang near each other most of the time with marginal differences in their sort times. Next is the selection sort, which is significantly faster than both bubble and shaker sort. Lastly is the insertion sort (meaning it is the fastest). This sort shone above the others as incredibly quick, consistently the fastest in almost every scenario.

But things started to get a little weird when you mess with already sorted data though.

Although I expected the bubble sort to be faster while sorting sorted data, I did not expect the difference to be so significant. Even weirder though is the fact that it reverses sorted data faster than it sorts unsorted data. Something about the data being sorted to begin with speeds things up, even if it has to completely reverse the data. At first, I thought that this was an error with my testing script, but after manual testing several times I found this to stay true. Sorting an unsorted file in either direction was consistently slower than sorting a sorted file in either direction.

The Shaker sort acted pretty much the same as the bubble sort. Same surprises. The only new thing that I noticed is that the difference in sorting time is greatest when the data starts unsorted. The times are much closer when the data is already sorted.

The Selection sort gave me a lot of trouble. The results confused me for a long time. It took me a while to realize that the way the selection sort works means it's generally going to always take roughly the same amount of time. The results were almost exactly the same regardless of whether the data was sorted or not. The only offset of this observation is when reversing sorted data. In this case, the selection sort actually *speeds up* slightly, catching me completely by surprise. Lots of checking and testing made sure that this is accurate, and this leads me to believe that potentially having the data already sorted in reverse makes it easier to find the number it is selecting quickly. Lastly, it's worth mentioning that because of the consistent speed of the selection sort, when sorting sorted data, it is actually *slower* than the bubble and shaker sorts.

Lastly, the insertion sort. This sort was obviously fast in the beginning, but the difference between the best case and the worst case was by far larger than any other algorithm. When sorting the sorted data, it was blisteringly fast, never taking over 2 one thousandths of a second to sort even 500,000 integers. This speed was difficult to believe, but more testing showed that it was accurate. When sorting the reversed data, it was actually slower than the selection sort and took *twice* the time it took to sort unsorted data.

Ben Betts
ECE 218

Source Code:

To get all of the data from the algorithms, I originally was hand typing array sizes and algorithm names for each individual test. This quickly ramped up in time taken and was incredibly frustrating when running on rabbit. At 100,000 and above the bubble sorts were taking several minutes, and at 500,00 the bubble sorts took roughly 30 minutes to run. Even the insertions took 5-10 minutes. This was very difficult to test, as if I left my computer for even a few minutes and it went to sleep, the connection to rabbit was lost, and I had to start the test over again.

To remedy this, I wrote a script that would automatically test each algorithm 3 times in each situation for each array size, find the average of the three times, then put all three times and the average into a nice and organized file. I'll include the script here just to help show my work process. It should work on rabbit, but I haven't tested it there.

Note that the properly view the output of the script, you should use the command "less -r datafile.txt". This lets less show color and makes it easy to scroll through the data. If you want to test the script with different combos of algorithms and sizes, simply modify the arrays established at the start of the file.

Files Included:

- main.cpp
- GetData.sh

main.cpp

```
#include <iostream>
#include <string>
#include <fstream>
#include "support.h"
using namespace std;

void bubble(int arr[], const int size, int argc) {

	for (int i = 0; i < size - 1; i++) {
		for (int j = 0; j < size - i - 1; j++) {
			if (argc != 5) { //if argc == 5, you told it to reverse, Only usable with
command line input
				if (arr[j] > arr[j+1]) {
					int temp = arr[j+1];
					arr[j+1] = arr[j];
					arr[j] = temp;
				}
			} else {
		    if (arr[j] < arr[j+1]) {
            int temp = arr[j+1];
            arr[j+1] = arr[j];
            arr[j] = temp;
          }
            }
        }
    }
}

void shaker(int arr[], const int size, int argc) {

    int start = 0;
    int end = size - 1;

    if (argc != 5) {
        while (start < end) {

            for (int i = start; i < end; i++) {
                if (arr[i] > arr[i+1]) {
                    int temp = arr[i+1];
                    arr[i+1] = arr[i];
                    arr[i] = temp;
                }
            }

            end--;

            for (int i = end; i > start; i--) {
                if (arr[i] < arr[i-1]) {
                    int temp = arr[i-1];
                    arr[i-1] = arr[i];
                    arr[i] = temp;
                }
            }

            start++;
        }
    } else {
        while (start < end) {

      for (int i = start; i < end; i++) {
          if (arr[i] < arr[i+1]) {
          int temp = arr[i+1];
            arr[i+1] = arr[i];
```

Ben Betts
ECE 218

```
                    arr[i] = temp;
                }
                }

                end--;

                for (int i = end; i > start; i--) {
                if (arr[i] > arr[i-1]) {
                    int temp = arr[i-1];
                    arr[i-1] = arr[i];
                    arr[i] = temp;
                }
                }

                start++;
        }
        }
}

void selection(int arr[], const int size, int argc) {
        int min;
        for (int i = 0; i < size - 1; i++) {
                min = i;
                for (int j = i + 1; j < size; j++) {
                        if (argc != 5) {
                                if (arr[min] > arr[j]) {
                                        min = j;
                                }
                        } else {
                    if (arr[min] < arr[j]) {
                        min = j;
                    }
                        }
                }

        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;

        }
}

void insertion(int arr[], const int size, int argc) {
```

Ben Betts
ECE 218

```
	for (int i = 1; i < size; i++) {

		int temp = arr[i];
		int j = i - 1;

		if (argc != 5) {
			while (j >= 0 && arr[j] > temp) {

				arr[j+1] = arr[j];
				j--;

			}
		} else {
	       while (j >= 0 && arr[j] < temp) {

		   arr[j+1] = arr[j];
		   j--;

	       }
		}

		arr[j+1] = temp;

	}
}

int getNumber(string fileName, int arr[], const int size) {

	ifstream source(fileName.c_str());
	int index = 0;

	while (!source.eof() && (index < size)) {
		source >> arr[index];
		index++;
	}

	source.close();

	return (index);
}

void menu(int &size, string &fileName, int &type, int argc, char** argv) {

	string typeString;
	if (argc <= 1) { //checks if the number of arguments passed in included this information
		cout << "Please enter the size of the array you wish to sort: ";
		cin >> size;
		cout << "Please enter the name of the file you wish to read from: ";
		cin >> fileName;
		cout << "Please select a sorting algorithm (bubble, shaker, selection, or
insertion): ";
		cin >> typeString;
		type = 5;
	} else {
		size = atoi(argv[1]);
		fileName = argv[2];
		typeString = argv[3];
		type = 5;
	}

	do {

	if (type == 0) {
		cout << "Please enter a listed algorithm: ";
		cin >> typeString;
	}

	if (typeString == "Bubble" || typeString == "bubble" || typeString == "b")
		type = 1;
	else if (typeString == "Shaker" || typeString == "shaker" || typeString == "c")
		type = 2;
	else if (typeString == "Selection" || typeString == "selection" || typeString == "s")
		type = 3;
	else if (typeString == "Insertion" || typeString == "insertion" || typeString == "i")
		type = 4;
	else
		type = 0;

	} while (type == 0);


}

void writeFile(string fileName, int arr[], int size, int type) {

	ofstream output;
	string  outName;
```

```
		if (type == 1) {
			outName = "bubble_" + fileName;
			output.open(outName.c_str());
		} else if (type == 2) {
		  outName = "shaker_" + fileName;
		  output.open(outName.c_str());
	    } else if (type == 3) {
		  outName = "selection_" + fileName;
		  output.open(outName.c_str());
	    } else if (type == 4) {
		  outName = "insertion_" + fileName;
		  output.open(outName.c_str());
	    }

		for (int i = 0; i < size; i++) {
			output << arr[i] << endl;
		}

		output.close();
}

void print(int arr[], const int size) {
		for (int i = 0; i < size; i++)
			cout << arr[i] << " ";
		cout << endl;
}

int main(int argc, char** argv) {

		int size = 5;
		string fileName;
		int type;

		menu(size, fileName, type, argc, argv);

		int * arr = new int[size];

		int actual = getNumber(fileName, arr, size);
		cout << "File Succesfully Read" << endl;
		if (actual != size)
			cout << "The requested size was " << size << " but the file only contained " <<
actual << " integers!" << endl;
```

```
	double startTime = getCPUTime();

	switch (type) {
		case 1: bubble(arr, actual, argc); break;
		case 2: shaker(arr, actual, argc); break;
		case 3: selection(arr, actual, argc); break;
		case 4: insertion(arr, actual, argc); break;
	}

	double endTime = getCPUTime();

	double timeTaken = endTime - startTime;
	cout << "Sorting took: " << timeTaken << " seconds." << endl;;

	//print(arr, size);
	writeFile(fileName, arr, size, type);

	return 0;
}
```

GetData.sh

```
#!/bin/bash
# saves me a ton of time

#TEXT VARIABLES:
purpletext="\033[35m"
bluetext="\033[34m"
bold="\033[1m"
normal="\033[0m"

#HELPER VARIABLES:
sortTypes=(Bubble Shaker Selection Insertion)
arraySizes=(1000 10000 25000 50000 100000 250000 500000)
logfile=dataFile.txt

#STORAGE VARIABLES:
storage=0
data1=0
data2=0
data3=0
average=0

rm datafile.txt
```

Ben Betts
ECE 218

```
printf "Script Started at " >> $logfile
date "+%H:%M:%S">> $logfile
echo >> $logfile

echo "-----------------------" >> $logfile
echo -e $bluetext"Sorting Unsorted Data"$normal >> $logfile
echo "-----------------------" >> $logfile
echo >> $logfile

for i in ${sortTypes[*]}; do
   echo -e $purpletext$i" Sort:"$normal >> $logfile #if you want to see color, you need to view
the file with "less -r"
   echo >> $logfile
   for j in ${arraySizes[*]}; do
          echo "Array Size:" $j >> $logfile
          echo "-----------------------" >> $logfile
      for k in `seq 1 3`; do
             storage=$(./a.out $j l2data.txt $i | grep Sorting | cut -f 3 -d " " | sed 's/e/*10^/g')
             if [ $k == 1 ]; then
                    data1=$storage
            echo "First Sort:" $data1 >> $logfile
             elif [ $k == 2 ]; then
                    data2=$storage
            echo "Second Sort:" $data2 >> $logfile
             elif [ $k == 3 ]; then
                    data3=$storage
            echo "Third Sort:" $data3 >> $logfile
             fi
      done
      echo "-----------------------" >> $logfile
          average=$(bc -l <<< "scale=6; ($data1+$data2+$data3)/3")
          echo "Average Sort Time:" $average >> $logfile
          echo >> $logfile
   done
done

echo "-----------------------" >> $logfile
echo -e $bluetext"Sorting Sorted Data"$normal >> $logfile
echo "-----------------------" >> $logfile
echo >> $logfile

for i in ${sortTypes[*]}; do
```

Ben Betts
ECE 218

```
   echo -e $purpletext$i" Sort:"$normal >> $logfile #if you want to see color, you need to view
the file with "less -r"
   echo >> $logfile
   for j in ${arraySizes[*]}; do
      echo "Array Size:" $j >> $logfile
      echo "-----------------------" >> $logfile
      for k in `seq 1 3`; do
         storage=$(./a.out $j sortedl2data.txt $i | grep Sorting | cut -f 3 -d " " | sed 's/e/*10^/g')
         if [ $k == 1 ]; then
            data1=$storage
            echo "First Sort:" $data1 >> $logfile
         elif [ $k == 2 ]; then
            data2=$storage
            echo "Second Sort:" $data2 >> $logfile
         elif [ $k == 3 ]; then
            data3=$storage
            echo "Third Sort:" $data3 >> $logfile
         fi
      done
      echo "-----------------------" >> $logfile
      average=$(bc -l <<< "scale=6; ($data1+$data2+$data3)/3")
      echo "Average Sort Time:" $average >> $logfile
      echo >> $logfile
   done
done

echo "-----------------------" >> $logfile
echo -e $bluetext"Reversing Sorted Data"$normal >> $logfile
echo "-----------------------" >> $logfile
echo >> $logfile

for i in ${sortTypes[*]}; do
   echo -e $purpletext$i" Sort:"$normal >> $logfile #if you want to see color, you need to view
the file with "less -r"
   echo >> $logfile
   for j in ${arraySizes[*]}; do
      echo "Array Size:" $j >> $logfile
      echo "-----------------------" >> $logfile
      for k in `seq 1 3`; do
         storage=$(./a.out $j sortedl2data.txt $i flip | grep Sorting | cut -f 3 -d " " | sed
's/e/*10^/g')
         if [ $k == 1 ]; then
            data1=$storage
            echo "First Sort:" $data1 >> $logfile
      elif [ $k == 2 ]; then
        data2=$storage
        echo "Second Sort:" $data2 >> $logfile
      elif [ $k == 3 ]; then
        data3=$storage
        echo "Third Sort:" $data3 >> $logfile
      fi
    done
    echo "-----------------------" >> $logfile
    average=$(bc -l <<< "scale=6; ($data1+$data2+$data3)/3")
    echo "Average Sort Time:" $average >> $logfile
    echo >> $logfile
  done
done

printf "Script Finished at " >> $logfile
date "+%H:%M:%S">> $logfile
echo >> $logfile

echo All Algorithms Complete!
```