PROBLEM DEFINITION

The program is designed to gather information regarding three separate sorting algorithms. The sorting algorithms being tested are Bubble Sort, Selection Sort, and Quick Sort. The user will be able to enter the number of elements that will be sorted by the algorithm. The program will then sort identical arrays of integers and provide a table of the number of loop iterations and swaps that took place inside each sorting algorithm.

ANALYSIS

I have found that I will require five libraries in the program to create the desired result. The first library I will need is the **cstdlib**. I will need this library to use the **rand()** function which I will need to generate the random integers for elements of the array. Next, I will use the **ctime** library. This library will allow me to use the **time()** function inside of **srand()** to create a seed for the **rand()** function. After that, I will use **iostream**. This allows me access to the input/output stream of the console to get input from the user and output the results to the console. The output to the console will use **iomanip** in order to have better formatting for the user. Lastly I will use the **string** library to create a string that I will use to get the input from the user.

The main function of my program will use two separate arrays that hold 20001 integers. These arrays will be called *nums* and *numsCopy*. *Nums* will be used to hold the array of randomly generated integers and *numsCopy* the array that the sort functions do the sorting on. I will use six integer variables to hold the counters for the functions. These will be called *bubbleLoops*, *selectionLoops*, *quickLoops*, *bubbleSwaps*, *selectionSwaps*, and *quickSwaps*. As the names imply they will hold the count of loop iterations and sways performed during each sort algorithm. I will also use an integer named *UserNumOfNumbers* to hold the amount of elements that is entered by the user. Finally, I will use a string named *user_input* to allow the user to enter an integer or use the command ‘choose a number for me’ to set the *UserNumOfNumbers* to the default value.

Each of the sort algorithms will have it’s only function to sort a fresh copy of the nums array that will be in numsCopy. I also have two helper function named *swap* and *partition* that are used by the sorting functions.

The only important formula that I will be using involves create the random numbers in the array. This formula will look as follows:

nums[i] = rand()%1000000 + 1

IMPLEMENTATION

The program was completed using Visual Studio Code on a system running the KDE Neon Linux Distribution. The source files were located on the grace.bluegrass.kctcs.edu server. I used two extensions called Remote FS and SSHExtension that allowed me to open the remote files in my local text editor. I used SSH to connect to the grace.bluegrass.kctcs.edu server in a separate terminal to compile and run the program. The compiler used was g++ version 5.4.0.

Run Time Analysis

I ran the program with seven different values for the number of elements to be sorted in the array. The values were 1000, 2000, 4000, 6000, 8000, 10000, and 12000. I thought this would give me a decent picture of the pattern for loop counts and swap counts for each algorithm. The following tables show the results for the inputs to the program.

Bubble Sort Loop and Swap Count

Elements	Loop Count	Swap Count
1000	499500	255235
2000	1999000	1021887
4000	7998000	4015780
6000	17997000	8957043
8000	31996000	16083276
10000	49995000	25288596
12000	71994000	35899056

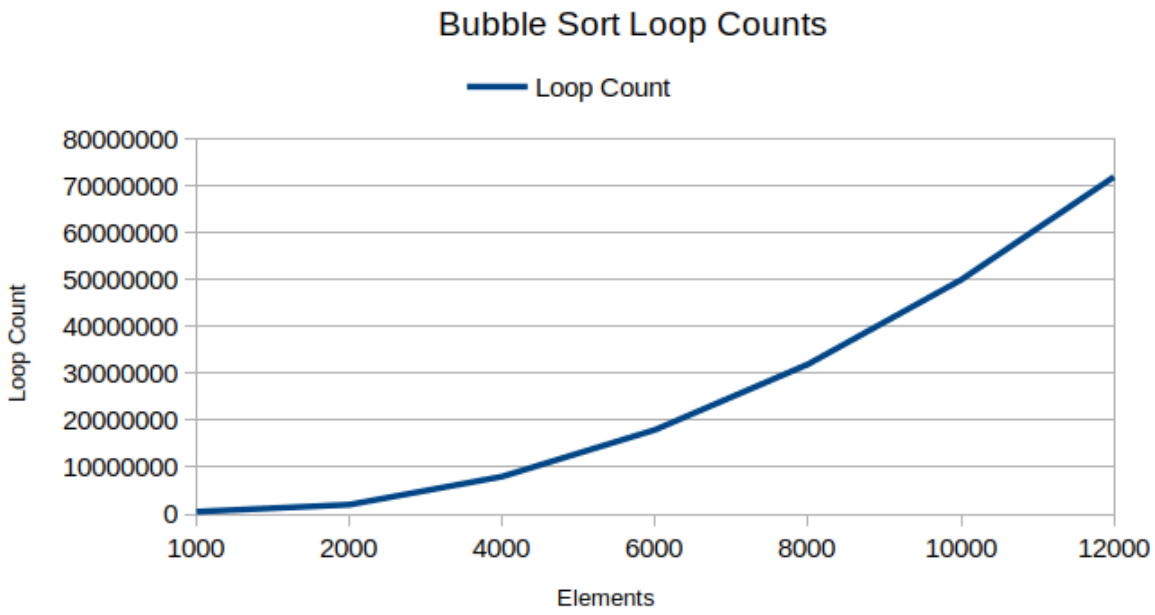
Selection Sort Loop and Swap Count

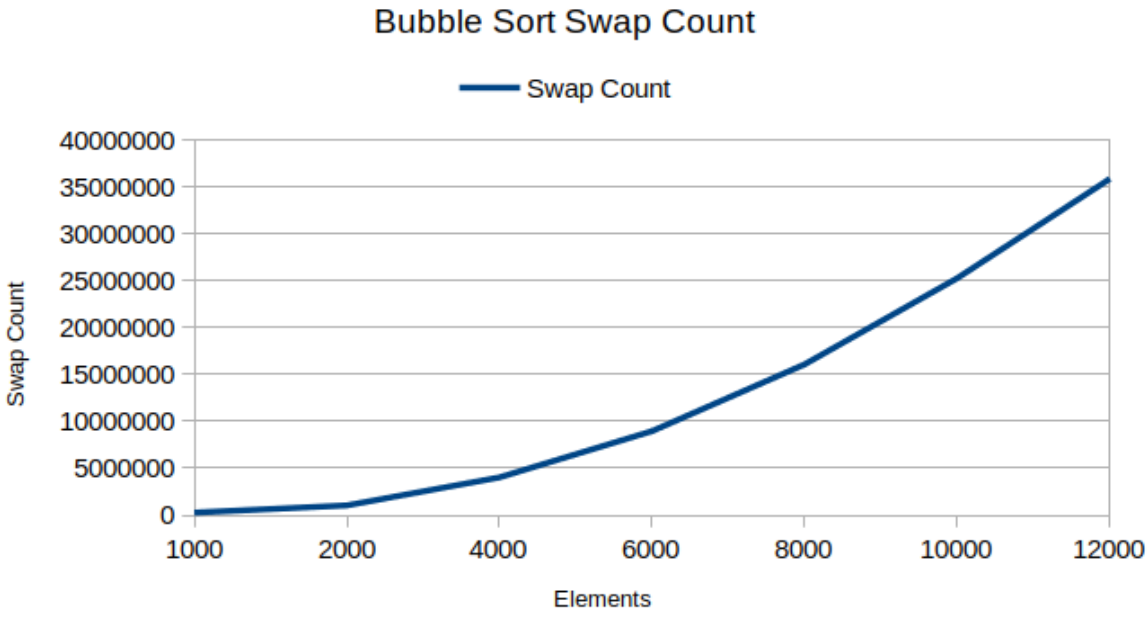
Elements	Loop Count	Swap Count
1000	499500	999
2000	1999000	1999
4000	7998000	3999
6000	17997000	5999
8000	31996000	7999
10000	49995000	9999
12000	71994000	11999

Quick Sort Loop and Swap Count

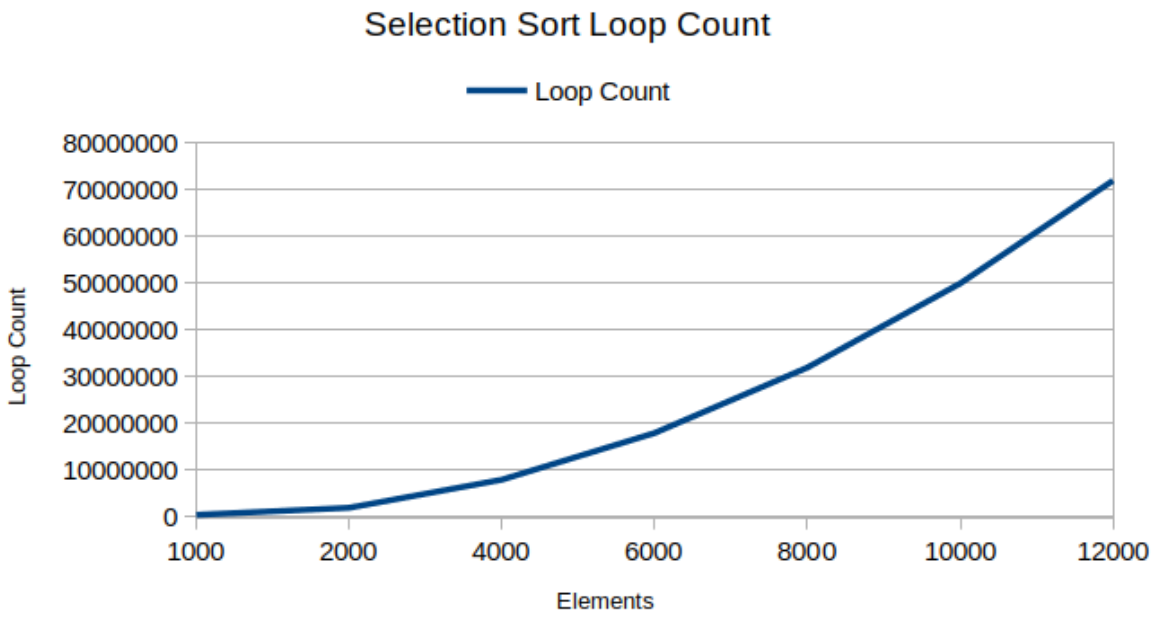
Elements	Loop Count	Swap Count
1000	7015	2377
2000	14442	5330
4000	30888	11631
6000	53969	18048
8000	81774	24611
10000	99768	31507
12000	156039	39622

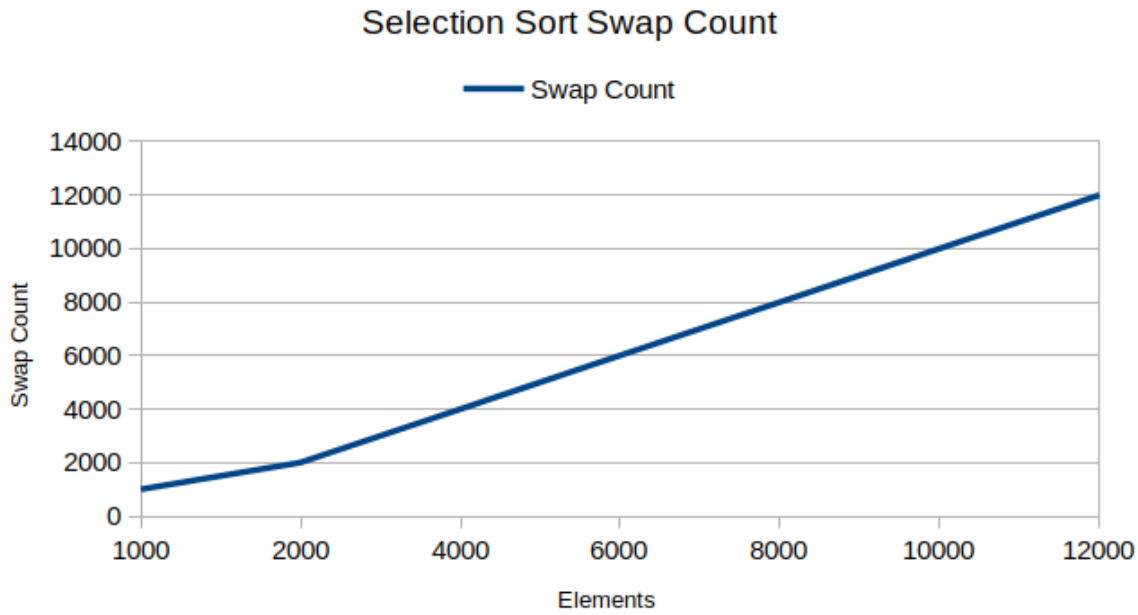
Firstly, I would like to say something about the bubble sort algorithm. All of the input tests on the bubble sort algorithm produced results of very high loop counts and very high swap counts. As the number of inputs increased the number of loop and swaps increased exponentially. The loop counts were right around ½ * N^2. The swap counts for bubble sort were approximately ¼ * N^2. The theoretical run time for the bubble sort algorithm is $O(N^2)$ which is exactly what the results of my tests have shown. The following graphs illustrate the loop counts and swap counts as the number of elements increased.





I ran the same inputs for the selection sort algorithm. The results show how the selection sort algorithm could perform better than the bubble sort but still have the same Big-O performance. The selection sort algorithm performed the same number of loop iterations as the bubble sort algorithm. This was due to both algorithms using nested for loops to traverse the array. The selection sort algorithm loop counts grew at the rate of ½ * N^2. The interesting difference with the selection sort algorithm is the number of swaps that are performed. This implementation of the selection sort performs exactly N-1 swaps. The $O(N^2)$ performance of the selection sort could perform better than bubble sort due to much fewer memory swaps. Memory swaps often require more time than reads so selection sort could perform better on some input sets. The theoretical performance of selection sort is $O(N^2)$ which is what I found to be the case with my tests. The following graphs give a picture of the rate of increase for the loops and swaps as the elements increased.





The quick sort algorithm performed much better when I applied the same amount of elements. My results as shown in the table above have shown that the loop iterations were increasing in a near linear fashion. The loop iterations were 7N, 7.5N, 7.72N, 10.2N, 10N, 13N. One difference with the quick sort is that the amount of swaps was high compared to the selection sort but the loop iterations was much lower. My results show that the run time for the quick sort is somewhere in between the quadratic and linear run times. The number of loop iteration grows in a logarithmic pattern in relation to N. The run time would be O(nlogn). The theoretical run time can be $O(N^2)$ at worst case but the average should be O(nlogn) which is the result got on the tests. The following graphs show the growth of the loops and swaps as the number of elements increases.