Lab 7 Runtime Complexity and Exceptions

Goals-
Compare runtime performance to the Big-O complexity
Use exceptions with file I/O

1. Write a utility program that will create 4 data files. For this lab you will need files with a thousand, or more, values. Your program should accept an integer from the user and then generate an output file with that many integers. You should prompt for the output file names so you do not need to regenerate a data file more than once. It is a simple program but design it carefully. The way you write the numbers to the output file must be compatible with how you are going to read them in your other program(s).

You will be working with searching and sorting algorithms. To simplify things generate random integers between 1 and 9. You will need a 0 near the beginning of a file, in the middle of another file, and near the end of a third file. You will also need a file without the target value. You must include appropriate exceptions to test the file operations succeed.

HINT: For the first file, start with the 0 and then add random numbers. For the second file, generate the random numbers and insert the 0 halfway through. For the third file generate the random numbers and put the 0 at the end. The last file is just random numbers.

2. You may use your Lab 6 program(s) for this lab. You may need to modify them to calculate the time of execution of the different algorithms. You can use this reference to measure execution time: http://www.cplusplus.com/reference/ctime/clock/. You can use any other timing mechanism you chose (other than clock time). Or you can borrow code, for only the searching and sorting algorithms you are investigating. Include a citation if you do. Your program will need to include some method of timing the execution. You will need linear and binary searches as well as one of the less efficient sorting algorithms. Bubble sort is notoriously inefficient and may give the best results.

Do some research on the Big-O complexity of the searching or sorting algorithms you use. Also in last week's lecture. Look at best case, average case, and worst case complexities (the reason for putting target values in different locations).

Then run your algorithms on the data files you generated. Yes, just like a science lab. ☺ Make a table showing the algorithm and size and type (where the target value is) of the data file. Your algorithms should be linear search, some sort, and binary search.

NOTE: If you do not see significant differences in the results then create new input files with more values. If you are using O(log n), O(n), and O(n^2) algorithms it should not require really big numbers to see differences.

Compare the results with your expectations. Do you see anything surprising in the results? Do the results make sense if you consider how many comparisons are made inside each algorithm (i.e. how many times do you need to compare the target to values)? If you are doing a one-time project does the complexity of sorting first change consideration of using a logN search algorithm? Does the time of sorting first offset and performance gain in using the efficient logN search?

What to submit-

Your helper program to generate the data files.
Your source file(s) that contains the algorithms you chose to use.
A pdf file with your data table, answers to the questions, and discussion of the results.

Modular Grading

We are using modular grading. Each lab will be divided into specific modules. Each module will be graded pass/fail. It either works properly or it does not. 10% of every lab or assignment grade is style/comments or other elements of self-documenting code and clarity. Remember the labs are worth 10 points total.

Programming style- 1 point

Program to create the 4 necessary input files- 1 point

Proper use of exceptions to test file operations- 1 point

Implement and test the linear search algorithm- 1 point

Implement and test the sorting algorithm - 2 points

Implement and test the binary search algorithm- 2 points

Analysis of results- 2 points