Jason James Pollman
ITCS2215-001
9/14/13

PROGRAMMING PROJECT #1 – README

FILES CONTAINED WITHIN THIS PACKAGE

1. *Readme.pdf*
2. *CountingDriver.java*
    - The driver class containing the "main" method. **This is the file that should be run from the command line.** It requires that you pass the program two *String* arguments—the first should be the file location of the input file and the second should be the file location of where to write the output file.
    - **Valid program calls (Windows):**
        java CountingDriver INPUT.txt OUTPUT.txt
        java CountingDriver \path\to\INPUT.txt \path\to\OUTPUT.txt
3. *Counting.java*
    - This is the class containing the methods that perform the program logic and algorithms.
4. *CountingException.java*
    - This is a simple, barebones exception to handle Counting class errors, which merely calls the parent *Exception* class constructor and prints the error *message* to the screen.

To compile these files, navigate to the directory in which they reside, and then run javac CountingDriver.java

These files were written in Java 7 and compiled using the JDK 1.7.0 (Major Version 51). Run the command *javap –verbose CountingDriver* for more information.

PROGRAM DESIGN

I designed the program to perform each task in separate methods, which leads to the most flexibility when working with the program later.

Counting.java constructs its objects taking in two *String* representations of the file locations of the input file and output file. Properties *inputFile* and *outputFile* are then instantiated as *File* objects with the arguments passed to the constructor. If the input file doesn't exist, a *CountingException* error is thrown and printed to the screen.

PROPERTIES:

1. *int[] inputArr*
    The array that holds the value of the integers from the input file, from lines 2 to *n*.
2. *int queryInt*
    The "x integer"; the integer extracted from the first line of the input file and used to compare the rest of the integers from the input file to.
3. *int queryCount*
    The integer that holds the number of integers less than or equal to *queryInt*. This value is assigned after calling the method *count()*.
4. *File inputFile*
    The file object for the input file.

5. *File outputFile*
   The file object for the output file.

METHODS:

1. *preprocess()*
   This method counts the number of lines in the input file, extracts the query comparison integer (*queryInt*), allocates the number of lines – 1 to *inputArr,* and assigns the integer values of the input file to *inputArr*, from the input file's lines 2 to *n*.
2. *insertionSort()*
   This method performs an insertion sort on the class property *inputArr*
3. *count()*
   This method compares the query integer (*queryInt*) to the integers stored in *inputArr*. If the value of an element in inputArr is *less than or equal to queryInt*, the integer property *queryCount* is increased by 1.
4. *writeOutput()*
   This method writes the value of query*C*ount to the first line of the output file and then writes each value of *inputArr* to the output file, in sorted order and one element per line.

ALGORITHM ANALYSIS – RUNNING TIME FOR THE count() METHOD

The asymptotic analysis of the algorithm used to compare the integers in *inputArr* to *queryInt*, can be expressed as $T(n) = c + \frac{n}{2}$ or $T(n) = c$ depending on the values of the elements in the *inputArr,* where $T(n)$ is the running time of the algorithm and c is the sum of the constant time assignment and comparison operations within the method.

The *count()* method essentially uses a very simple "divide and conquer" approach. It 'splits' the *inputArr* in half and compares either the first or second half of the array to the *queryInt* based on whether or not the middle element in the array is *greater than* or *less than or equal* to *queryInt*.

In the worst case, the *count()* method will have a running time of $T(n) = c + \frac{n}{2}$ because the loop used to compare the *inputArr* integers to *queryInt* must iterate from either 0 to $\frac{\text{inputArr.length}}{2}$, or $\frac{\text{inputArr.length}}{2}$ to $inputArr.length$ depending on whether *queryInt* is *greater* or *less than or equal* to *queryInt.* The algorithm runs at a linear rate, as there is only one loop in the method. As a summation, the worst case can be expressed as: $\sum_{i=0}^{k} n$ or $\sum_{i=\frac{k}{2}}^{k} n$ where k is equal to $inputArr.length$ and n is the number of elements in *inputArr*.

In the best case, the method runs in constant time. That is, considering that the array is sorted, we can check to first and last elements in the array to see if they are *greater* than *queryInt* or *less than or equal to queryInt*, respectively. These comparisons are made at the very beginning of the method. However, it is unlikely that, in a real world situation, all of the input integers are either *greater than* or *less than or equal to* the comparison integer—although possible. So, the minimal time to make these checks is well justified by the time saved if such a case were presented.

If the first element in the array is greater than *queryInt*, then we can return 0. Likewise, if the last element in the array is *less than or equal to queryInt*, then all the elements in the array are *less than or equal to queryInt* and we can return the size of the array (*inputArr.length).*

Thus, in the best case, $T(n) = c$ and $T(n) \in \Omega(1)$, where c is the number of constant time operations performed in these "initial" checks.

The order of growth for the *count()* method is:

$$T(n) \in O(\frac{n}{2}) \qquad T(n) \in \Omega(1) \qquad T(n) \in \Theta(\frac{n}{4})$$

WHAT WORKS, WHAT FAILS

I think *Counting.java* does a fairly good job of minimizing the time needed to perform the assignment of this project. However, along the lines of "divide and conquer," the problem could be broken down further and the *inputArr* could have been 'divided' into 4, 8, or even more sections and compared to *queryInt*. This would have lowered the time complexity even further. Note, that when I say divided, I don't actually mean I divided the input array into sub-arrays, I mean checking the value of the $\frac{\text{inputArr.length}}{n}$ *th* element, where n is the number of divisions. In the case of this algorithm, $n = 2$.

However, given the sample input of this assignment, one must consider the balance between the number of constant time operations needed to compare sections of the array with the query integer versus the maximum run time of the loop. In the sample input case, and given this algorithm, it takes a maximum of 5 constant time comparisons to set the lower and upper bounds, and then the loop will never run more than 10 times considering that there are 19 integers to compare to *queryInt*. Checking each quarter of the array would take a maximum of 12 comparisons, and then after finding the loops lower and upper bounds, it would take at most 5 iterations to determine the correct number of integers less than or equal to *queryInt*. So, in the worst case, dividing the array into quarters would take 1 more constant time comparison compared to dividing the array into halves (16 vs. 15).

Yet, if the sample size were much larger, comparing each 4th, 8th, nth element of the array and comparing it to the query integer to determine upper and lower bounds before comparing the query integer with the "divided" sections would be more efficient.