QuickSort Simulation Design Discussion

1. **State the function that you think your plot represents and why.**

The plot shows a curve function representing the relationship between the size of each array (n, on the x-axis) and the average number of exchanges QuickSort makes for each array size, divided by the array size as a measure of efficiency (average / n, on the y-axis). A horizontal asymptote occurs in the function near where the y-value reaches 12, implying that increasing the array size no longer has a strong effect once the efficiency value is close to 12. Different functions that have similar behavior to the plot function include n1/2 and log(n). The plot increases quickly at first, but abruptly reaches its horizontal asymptote, so it matches log(n) best.

1. **Explain what this function has to do with the average case complexity of quicksort.**

QuickSort has an average case complexity of nlog(n), which has similar behavior to log(n). The difference between the two functions is that nlog(n) increases more quickly than log(n) at first. Both functions indicate efficient algorithms because both of them increase slowly near a horizontal asymptote. Eventually, as n approaches infinity, log(n) reaches its horizontal asymptote slightly sooner than nlog(n) does.

1. **Explain how you designed the simulation part of your program.**

Designing the QuickSort simulator required a few steps: filling arrays of increasing length with random numbers, running QuickSort on each length of array repeatedly (eventually 50 times), calculating the average number of exchanges made for each length of array with an efficiency value based on the array size ( average / n ), and creating a table for output. Other key programming concepts used include long type variables to handle the large numbers produced by the simulation, usage of C#’s System.Random class to create random arrays of numbers, and strings that made use of C#’s built-in string formatting syntax.

The random numbers that filled each array were determined by the   
Random.Next() method. Each randomized number ranged from 0 to a maximum value 1 less than the size of the array. Simulating QuickSort at various sizes of arrays was necessary to show how efficient the algorithm could be because the efficiency of algorithms is affected by the size of the input the algorithm is handling. For this simulation, the given array sizes ranged from 10 to 500000 values. As the array size increased, the number of exchanges made by QuickSort increased. The relationship between input size and the number of exchanges made the plot’s y-axis values (average / n) useful as a measure of efficiency. Recording the slope of the plot at any array size (n) also showed the trend of efficiency near the given array size.

After generating the random number array for an array size, continuing the simulation required running QuickSort several times by rewriting new sets of random numbers to the array repeatedly. This simulator ran QuickSort 50 times per array size. Running QuickSort multiple times allowed the random number generator to fill the arrays as randomly as possible and allowed for average run calculations. To test this part of the simulation, low array sizes were used first to check whether the algorithm was behaving as expected and to analyze the algorithm step by step. Tracking the values that fill each array and the pivot point each time QuickSort() calls Partition() was easier to accomplish with smaller arrays.

Finding the average number of exchanges for a given array length required adding counter variables to the QuickSort algorithm. The number of exchanges acted as a metric to track efficiency for QuickSort as the exchanges noticeably increased with the input size up to the horizontal asymptote y-value. The average number of exchanges also increased more quickly than the array size. To calculate an efficiency value for each array size, the average number of exchanges was divided by the size. Using a double type variable is important when calculating the efficiency value to show the differences between values. Near 12 on the plot’s y-axis, the points closed in on each other as the plot reached its horizontal asymptote.

The output table was made using C#’s string formatting syntax. The amount of space in each column of the table and the alignment of the values in each table were defined as -22, with “-“ representing a left-aligned column, and “22” being the number of characters allowed in each cell. The formatting syntax available in C# is indicated by the “$” and curly braces. This type of string formatting used variable values and the alignment options with the syntax “{<interploationExpression>,<alignment>}”.