**Part 1:**

The Quicksort's performance depends on several factors, two of which are the choice of the pivot (partition element) and the way the file (array) is presented. In this assignment you will empirically examine the run time efficiency of the Quicksort as the above factors are allowed to change.

**Choice of Quicksort Pivot**

Quicksort requires the identification of a pivot, an element (usually from the array to be sorted) with the property that all items on one side of the pivot have a value greater than or equal to it, and all items on the other side of the pivot have a value less than or equal to it. Three methods for choosing a pivot:

1. Pick the first element of the array.

2. Pick a random element from the array.

3. Pick the middle element

Creation of the arrays

The timing of the Quicksort is also dependent on the size of the array and how the array is currently ordered. You should execute the Quicksort algorithm with the following array possibilities of varying sizes:

1. Already ordered

2. Reverse ordered

3. Randomly ordered

The Experiment

Using the Quicksort algorithm, count the number of statements needed to sort the arrays above and generate the following tables. (Note: count the number of statements within your sorting algorithm only. Do not count the statement that keeps tracks of the number of statements executed)

Already ordered - number of statements needed to sort

Size (N) First Pivot Random Pivot Middle Pivot

5000 ????? ????? ?????

10,000 ????? ????? ?????

50,000 ????? ????? ?????

Reverse ordered - number of statements needed to sort

Size (N) First Pivot Random Pivot Middle Pivot

5000 ????? ????? ?????

10,000 ????? ????? ?????

50,000 ????? ????? ?????

Randomly ordered - number of statements needed to sort

Size (N) First Pivot Random Pivot Middle Pivot

5000 ????? ????? ?????

10,000 ????? ????? ?????

50,000 ????? ????? ?????

Print out the first 5 elements of each type of the array (not every array).

Write a Big-oh analysis of your sort algorithm using the results above. A short paragraph in well-constructed English should suffice. Be rigorous!

**Part 2:**

Create three arrays as such:

1. Already ordered

2. Reverse ordered

3. Randomly ordered

Count the number of statements needed to sort the arrays above using the sort algorithms below and generate the following tables. (Note: count the number of statements within your sorting algorithm only. Do not count the statements that keep track of the number of statements executed)

* Merge sort
* Heap sort

For input sizes of 50,000, and 75,000.

Your output should look like

Already ordered - number of statements needed to sort

Algorithm 50,000 75,000

Merge sort ?????? ???????

Heap sort ?????? ???????

Reverse ordered - number of statements needed to sort

Algorithm 50,000 75,000

Merge sort ?????? ???????

Heap sort ?????? ???????

Random ordered - number of statements needed to sort

Algorithm 50,000 75,000

Merge sort ?????? ???????

Heap sort ?????? ???????

Print out the first 5 elements of each type of array.

Write a Big-oh analysis of your sort algorithm using the results above. A short paragraph in well-constructed English should suffice. Be rigorous!

**Part 3:**

Use the Radix sort algorithm to sort the following strings.

acdes asdsd ksjtr kerpt absqa zabaa rkdsb qqqqq kdfaa zedsd

Five passes will be required.

Print the order of the strings after each pass.