0. Introduction:

As you already know, sorting is one of the most important problems of computer science. Entire books have been devoted to the sorting problem alone. In this discussion, we shall consider one of the better sorting algorithms, Quicksort.

Quicksort is another exchange sort, meaning that, like other algorithms for sorting you know, it merely compares and exchanges values to achieve its goal, that of putting numbers or words into ascending (or dictionary) order. Quicksort has also been described as a partition-exchange sort, or a divide and conquer algorithm due to the nature of the process involved.

In order to discuss our sorting algorithms, we shall assume that we wish to sort a list of n integers into ascending numerical order. If we can do this problem, then other problems are similar and can be easily handled.

1. Quicksort:

The quicksort algorithm proceeds as follows. The first step is to select an element of the list which will serve as the pivot. The algorithm proceeds by then partitioning the list into sublists, the left sublist and the right sublist. We move all the elements of the list which are less than the pivot to the left sublist, and all the elements which are greater than the pivot to the right sublist. The partition element is placed in the middle. We then have two sublists to consider, the left sublist and the right sublist. These lists are recursively sorted in the same way as the original. The quicksort is best implemented as a recursive algorithm, although some authors claim that a non-recursive version might be faster. However, most computing languages handle recursion very efficiently, so the difference may not warrant the effort at coming up with a non-recursive version of the algorithm. (Although, we shall see, and have seen before, that Python does not do as well on some recursive algorithms as it should).

Your textbook has a very nice discussion of the quicksort (partition-exchange version) which you should read and implement carefully. Listing 4.23 on page 176 works very well as a Python implementation of this algorithm. You are strongly encouraged to read pages 174-178 of your text for a mostly complete discussion of the quicksort algorithm.

However, this, unfortunately, is not the end of the story.

Quicksort works very well on lists that are in random order. If we simply select the pivot element to be the first element of the list, the algorithm works very well. The textbook version described above selects the pivot as the first element of the list.

The performance of the quicksort algorithm is also of interest. In its average case behavior, the quicksort is O(n*ln(n)). However, if the list is in sorted order, reverse order, or nearly in one of these states, the performance of the quicksort algorithm can degenerate to O(n**2). What happens is that when the partition step above takes place, one of the lists, left sublist or right sublist, is empty or nearly empty. If that happens, then quicksort degenerates into a simple selection sort, which is O(n**2). We must take measures to attempt to insure that this behavior does not occur.

Moreover, python has some nice tools that can assist in the implementation of the quicksort. Python has

“list comprehensions” which we may exploit to write a very short and effective quicksort. Study the following interaction:

```
>>> import random
>>> lst = range(20)
>>> random.shuffle(lst)
>>> lst
[9, 6, 16, 17, 14, 10, 0, 19, 18, 15, 5, 3, 8, 12, 11, 2, 1, 13, 7,
4]
>>> pivot = lst.pop(0)
>>> pivot
9
>>> lefthalf = [x for x in lst if x <= pivot]
>>> righthalf = [x for x in lst if x > pivot]
>>> lefthalf
[6, 0, 5, 3, 8, 2, 1, 7, 4]
>>> righthalf
[16, 17, 14, 10, 19, 18, 15, 12, 11, 13]
>>>
```

It should be relatively clear that this can be exploited to produce a straightforward quicksort implementation. But questions arise. Is this algorithm as good as, better than, or the same as the algorithm described in the text? Does it solve the problem of having a list which is already sorted or nearly in sorted order? The answer to this last question is clearly no!

There are two possible solutions to the problem of quicksort with a nearly sorted list. Both involve making better choices of the pivot element instead of simply choosing the first one.

If we have no information in advance about the list, a better choice for pivot element might be to simply choose a random element from the list.

```
pivotindex = random.randint(0,len(lst)-1)
pivot = lst[pivotindex]
```

As suggested in your text, another approach would be to use the “median of three” approach. To choose the pivot element, we look at the first element, the middle element, and the last element of the list. Now pick the median of these three values and use it as the pivot. (Remember, the median of a set of values is simply the middle value). One would think that choosing such a median as the pivot would provide a better chance of a more nearly even division of the list into left half and right half, which of course is what we want.

2. Student Experiments:

Task Zero: Recall reading and writing files, and random number generation. Write a program which will generate a file inorder numbers, reverse order numbers, and random numbers in files of varying sizes. Create files of size 1000, 2000, 4000, 8000, 16000, 32000, 64000 and 128000. Use these as test

data for your algorithms. (When you test your algorithms, do not print the entire content of these files, just print the first few so we are convinced the algorithms work.)

Task One: Implement the quicksort algorithm in the textbook. Include a timer so that the time for the execution of the algorithm can be recorded. Time only the algorithm itself, not the file reading or writing or any of the other overhead associated with the implementation. Include a counter to count the number of comparisons required to complete the algorithm. You may use a global variable for the comparison counter. Test your algorithm on each of the files generated in task zero and record the results.

Task Two: Implement a version of the quicksort algorithm in which list comprehensions are used. Again time the algorithm, and (while it may not be possible) attempt to include a counter to count the number of comparisons required for the algorithm to complete. Test your algorithm on each of the files of task zero and record the results.

Task Three: Implement a version of the quicksort algorithm in which a random pivot element is chosen from the list. Use list comprehensions. Test your algorithm on each of the files of task zero and record the results.

Task Four: Implement a version of the quicksort algorithm in which a “median of three” strategy is used to select the pivot element. Use list comprehensions. Test your algorithm on each of the files of task zero and record the results.

Task Five: Write a summary paragraph in which you describe the results of your experiment. What do you observe? Compare and contrast. Include a spreadsheet which summarizes the results.

Put everything you wish to hand in into a folder named

youruserid.quicksort

and copy that folder into my inbox on or before midnight of the due date.

cp -r youruserid.quicksort /users/meggen/Public

Files are all time stamped so I will know whether or not you submitted your work on time.