CSCI 204 – Introduction to Computer Science II

Lab 11 – Quicksort and Efficiency

1 Objectives

The objectives of this lab are to:

- Examine program efficiency
- Improve a program's efficiency
- Become familiar with Quicksort

In this lab you will learn about program efficiency by analyzing the efficiency of the *quicksort* algorithm. You will also learn a technique that improves the efficiency of *quicksort* by providing better pivots in the case of certain inputs that cause *quicksort* to take a long time (i.e., $O(n^2)$ or quadratic time) to run.

Read this whole document first before you begin the assignment.

2 Introduction

In this lab you will analyze the efficiency of quicksort. To summarize, the algorithm works by selecting an element in the list, called a pivot. Next, the algorithm partitions the list into two separate lists - those elements that are less than the pivot, and those that are greater than or equal to the pivot. You recursively call the quicksort algorithm on these sublists. This process repeats until the sublists contain only one element (i.e., the base case in the recursion.) By definition, a list of one element is sorted, and the recursion returns the sorted list. Some implementations take these sublists and merge them back together. However, the best implementations of quicksort perform an in-place sort, alleviating the need for a merge at the end. (In-place means we only use one list instead of creating new smaller lists with each partition. We use the in-place version here). Though this lab makes light of the selection of the pivot, this is the most essential part of the algorithm. Selecting the pivot determines how well the list can be partitioned, and it can have a dramatic effect on the running time of the algorithm.

You will perform a very basic analysis on the observed running time of the algorithm on a couple of different test input lists. To gather data about the running time, you will observe the number of key comparisons that quicksort uses to sort a list of data. Key comparisons are comparisons between the values in the list. You will perform a test on a list of random data, and then on data that is already sorted. The first implementation of the partition algorithm, that is, picking the first element in the list as the pivot, will demonstrate the worst-case running time to be $O(n^2)$ (that's quadratic) on the sorted list! This is hardly a desirable outcome. You will modify quicksort algorithm so that its worst-case behavior on sorted lists is dramatically improved. We'll call the improved version of quicksort the modified version of quicksort. After you have completed test runs of the unmodified and modified versions of quicksort and collected data on their performance when sorting random data and data that is already sorted, you will graph your data and compare it with our predicted behavior of quicksort.

3 Getting Started

Make a directory for this lab. Copy the file(s) from the Linux directory ~csci204/2017-fall/student/labs/lab11/. You should see the following set of files.

MyGraph1.m MyGraph2.m quicksort.py

You will work from these files to complete your lab exercises. Please create a lab.txt file to record the results.

4 Running Quicksort from main()

You will be editing the quicksort algorithm and you need to make sure it is still working so you'll need to test it.

Task 1: Add code to the main() method in quicksort.py to test quicksort.

1. Create a Python list of 10 random integers (in the range 0…10). Use randint() to generate random integers. Print the list, sort it using the given quicksort() method, and print the list again. You will be able to quickly verify that the given quicksort() method is working correctly.
2. Do the same thing for a list of 10 sorted integers. Use the Python range() method to generate your sorted list.
3. Run your code to make sure that you don't have any unexpected runtime errors.

Next, you will run the quicksort algorithm on Python lists of increasing size, from 8, 16, 32, … , 16384 (i.e., 2^3, 2^4,· · · , 2^{14}). That is, the list size will double each time.

Task 2: Add more code to main to test the Big-O runtime of quicksort.

1. Create a list of all the required sizes [8, 16, 32,· · · , 16384] using a list comprehension.
2. Use a for loop to create a list of each size: Fill the list with random numbers in the range 0 to 20,000 and then sort it. Do not print these lists because it will be too long.
3. Do the same thing for sorted numbers but only create lists up to size 512 (2^9) since lists larger than that will exceed Python 3's maximum recursion depth (too many recursive calls).
4. Run your code to make sure that you don't have any unexpected runtime errors.

5 Counting Key Comparisons in Quicksort

A key comparison occurs whenever two values in the list are compared. Therefore, comparing the contents at two list indices is a key comparison while comparing two list indices is not.

Here are some examples.

```
if array[index] < array[otherIndex]: # a key comparison
if array[index] < smallest: # a
```

```
key comparison
if index < otherIndex: # NOT a key comparison
```

Task 3: Count the key comparisons.

Revise the partition()method so that it will count the number of key comparisons that were required for a sort. Be sure to count only key comparisons.

1. Define the counting variable as a globalvariable so it can be used in different methods.
2. Reset the count to zero before each call to quicksort()in main().
3. Increment the count each time you do a key comparison in partition().
4. In main(), create files called outru (where the letter 'r' means 'random', and the letter 'u' means 'unmodified') and outsu ('sorted', 'unmodified').
5. After quicksort returns, print a line containing the size of the list and the number of key comparisons. Print only these two numbers separated by a space character on each line. Don't label your output or print any other words or numbers. The tests on random lists should print to outru and the tests on sorted lists should print to outsu.
6. Remember to close both files at the end of main().

A sample of your output should look *something* like:

```
8 18
16 41
32 173
64 312
128 859
256 1950
512 4595
1024 10816
2048 24813
4096 67139
8192 118310
16384 269269
```

Glance at the files outru and outsu. With the pivot scheme as defined in the partition method as it stands right now, Quicksort's worst case behavior occurs when the list is already sorted so outsu should have much higher key comparison counts than outru.

6 Graphing the Results

Now you will use the MATLAB commands in MyGraph1.m to produce the plot.

Task 4: Graph your unmodified results.

Open a Linux shell window, change to the directory for today's lab. Start MATLAB at the prompt by typing matlab &. Run the command file MyGraph1.m inside of MATLAB by typing the command MyGraph1 at the MATLAB prompt and it will execute the script for you. Have the TA or instructor verify that this graph is correct.

7 Improving quicksort's Efficiency

First, make sure you understand why quicksort is performing so poorly on the sorted input. The reasoning has everything to do with how we choose the pivot. Currently, we always choose the first element in the list as the pivot. When the list is already sorted, the result is a worst-case split, meaning, no elements will be less than the pivot, and the rest of the specified list are all greater than the pivot! As a result, the list is split into two lists with one list of size 1 and the other list of size n-1. You cannot have a worse outcome with quicksort! In this case, our quick-sort case ends up performing no better than typical O(n^2) sorting algorithms.

There are numerous algorithms out there that improve the running time of quicksort by coming up with a better scheme for choosing the pivot. To get the theoretical best case scenario with quicksort, we want to choose a pivot that splits the list equally during every step of the recursion. (In other words, you want to choose the exact median of the sublist at every step of the recursion.) This would give us a worst case running time of O(nlogn). To do this precisely, however, would require a sorted list so the position that splits the list in exact half can be computed. Not exactly ideal. In situations like this, we use reasonable approximations that take very little extra time to compute. One simple solution is to use median-of-three partitioning.

The median-of-three partition takes the first, middle, and last numbers in the list and puts the median of those numbers into the first spot so it can be used as the pivot. This means the pivot is guaranteed not to be the absolute highest or lowest number in the unsorted portion of the list. You will implement this improved partition algorithm and graph the key comparison counts to see how much this change improves the quicksort algorithm.

8 Modifying the quicksort() Method

Task 5: Make a copy of quicksort() and partition().

Copy the quicksort() and partition() methods and name the copies quicksortm() and partitionm() (*m* for modified). Doing so allows one to be able to run to the unmodified code again when need-ed. Edit quicksortm() so it calls quicksortm() (recursively) and partitionm(), not to the original methods of patition() and quicksort().

Task 6: Add more code to main to test the Big-O runtime of quicksortm().

1. Test your modified quicksortm() on size 10 lists for both random and sorted data.
2. Run your modified quicksortm() on Python lists of increasing size, from 8, 16, 32, … , 16384 (i.e., 2^3, 2^4,· · · , 2^{14}). (This time, you will be able to do 2^{14} even for the sorted list). You will be producing files named outrm and outsm (random modified and sorted modified).

9 Modifying the partition() Method to Use a Better Pivot

You need to make a change to how the pivot is selected in the partition() method. (Remember the change(s) are to be made in partitionm()!) To implement median-of-three partitioning, examining the values in the first, middle and last positions of the list. (You already have first and last positions. How do you calculate the middle position?) The current method uses the first element as the pivot. Therefore, you need to examine the elements at these three positions, swap the median of these values into the first (i.e., the pivot) position, and then proceed as usual.

You can determine which of these three values has the median value by using if statements and calls to the swap() method. Be sure to add any new key comparisons into your key comparison count.

Task 7: Implement the median-of-three strategy inpartition().

Run your program to generate all four output files and to verify that all four short lists (0 through 9, inclusive) were successfully sorted.

10 Graphing your results again

Now it's time to graph all four output files and see how the efficiency of quicksort has changed with the median-of-three partition algorithm. This time, you will also edit the MATLAB program so it puts your name on the graph.

Now you will use the MATLAB commands in MyGraph2.mto produce the plot.

Task 8: Graph all of your results.

Open a Linux shell window, change to the directory for today's lab. Start MATLAB at the prompt by typing matlab &.

Open the command file MyGraph2.m inside of MATLAB by typing the command edit MyGraph2.m at the MATLAB prompt. Edit the title command so that it inserts your name where it says *YOUR NAME HERE*. Save your work and close the editor. Run the command file MyGraph2.m inside of MATLAB by typing the command MyGraph2 at the MATLAB prompt and it will execute the script for you. Have the TA or instructor verify that this graph is correct.

When the graph is nicely formatted on screen within MATLAB's plotting window, select File | Export Setup … from the plot window. In the dialog window that comes after the exporting command, click on the Export button. In the next dialog select Portable Document Format as the file format and name the file graph.pdf. Finally, click on the Savebutton.

To view the resulting PDF file, open a shell window and type atril graph.pdf &.

11 Submitting the lab work

Upload your code and the pdf to Moodle. Double check to make sure your files all got uploaded successfully.

12 Double checking your results

The following graphs show some of the common errors in programming when generating data. The last one shows the correct graph. Check against these graphs to see if your results are correct.

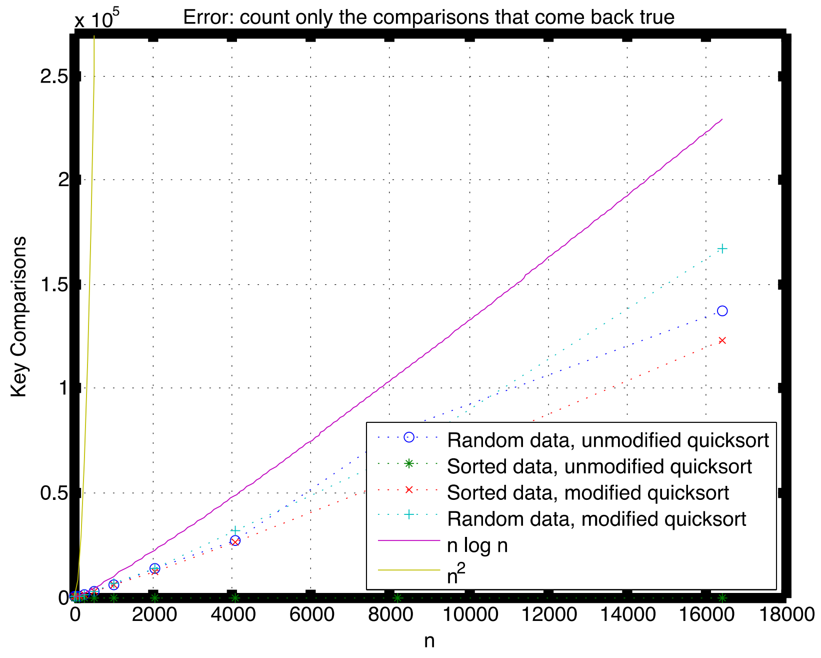

Figure 1. This shows a graph achieved when comparisons are counted incorrectly

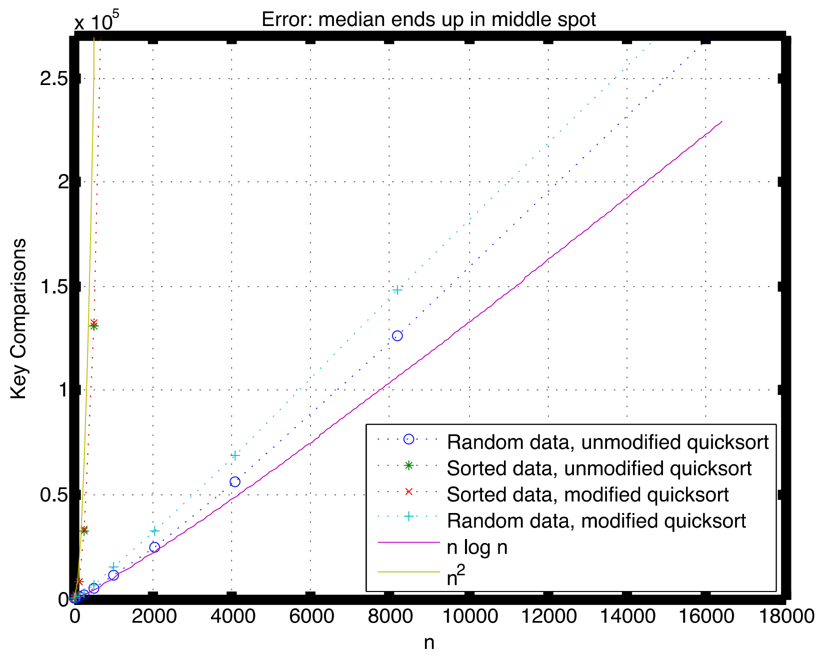

Figure 2. This shows a graph that reflects a bad pivot value

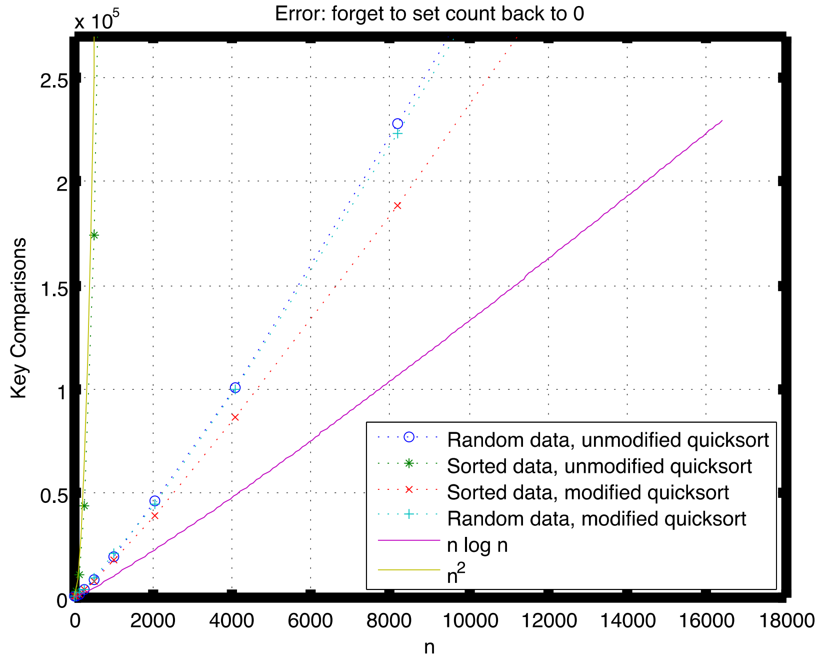

Figure 3. This graph is fairly common when count is not correct reset

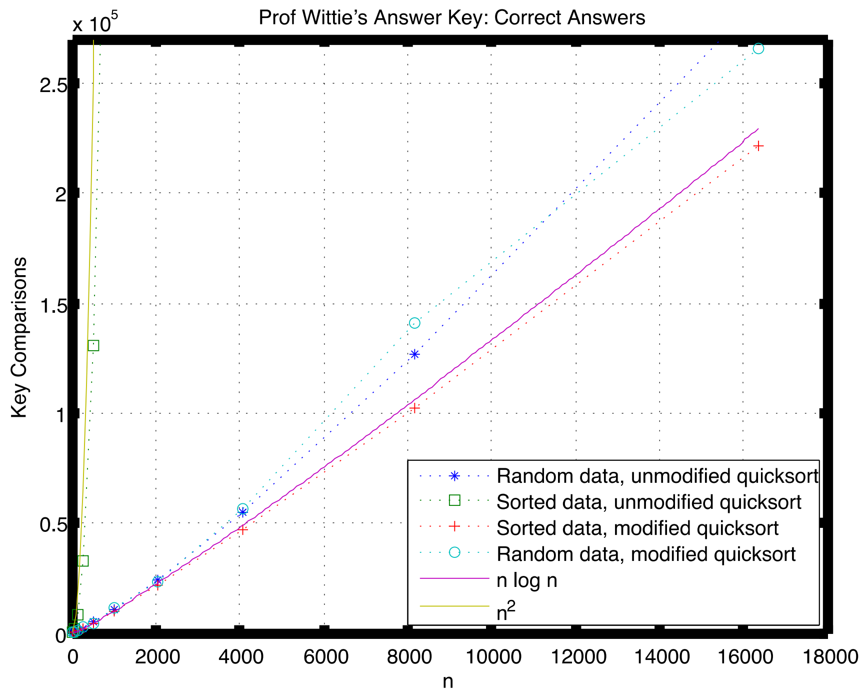

Figure 4. That's a nice graph. (It's Correct!)