Kristina Frye
Laura DeWitt
CS 350
Class Project

Project topic: Sorting algorithms
Implementation language: Java with additional tools written in Python and Bash script. Java was chosen because we are both familiar with it, it does not require working with pointers (the cause of many programming bugs), and it has some nice functionality that should make the implementation of the sorting algorithms fairly straightforward. It also has an interface available (ThreadMXBean) that should allow us to isolate the runtime of the thread running the program, thus excluding other threads that the jvm might be running such as the garbage collector thread. According to http://blog.it-weise.de/p/277, we may need to run our programs many times before taking measurements in order to avoid errors associated with the JIT characteristics of the Java compiler.

Features:
The purpose of our project is to implement several different “good” sorting algorithms, perform measurements of our implementations, and determine if those measurements agree with the theoretical performance predicted by mathematical analysis of the algorithm. At a minimum, we will compare Merge Sort to Quick Sort and Quick Sort 3 (that uses three partitions). If time permits, we will also add comparisons of Heap Sort and Distribution Counting Sort. The Distribution Counting Sort belongs to a different class of sorting algorithms than the other algorithms because of the input requirements, but, if it works as predicted, it would be an interesting demonstration of how sorting can be improved in certain circumstances.

We plan to create a number of tools that will assist in our measurements.

- Number generator (Python): A tool that will generate a file containing a list of unsorted numbers. We will use this tool to generate a number of different types of test cases including:
  - a list of unique numbers
  - a list of *k* * *n* numbers in which n numbers are unique. That is, there will be *k* repeats of each number in the list.
  - a list of numbers containing only a few different numbers, each repeated many times
  - a list of unique numbers plus a single number repeated many times

  All of the test cases will be “shuffled” so that the numbers appear in pseudo-random order.
- Sort checker (Python): A tool that checks whether an input file containing a list of numbers is sorted correctly.
- Testing script (Bash): A tool that will run each sort program repeatedly and record measurement results. The first *n* runs will not be recorded in order to avoid

measurement errors associated with JIT. We will need to do some experimentation to discover what number *n* should be.
- Data collection (Python): A tool that will take the measurement results created as an output of the testing script, throw away any outlying measurements, and average the remaining results into a form that will be easy to graph.
- I/O library (Java): A common class used to get the unsorted data from a file specified in standard input and put it into a Java array that can be efficiently used by our sorting algorithms. Similarly, this library will contain a method for outputting the sorted array into a file. Using a common library for I/O functionality should eliminate one possible sort of measurement error when comparing the algorithms performance to each other.

The paper itself will consist of:
- Pseudocode of algorithms
- Short examples of how each algorithm works
- Code snippets of each implementation showing the core of each algorithm
- Description of the testing procedures including a description of the test cases, the size of each test case, and the number of times each test case was run for each sorting algorithm.
- Results of performance measurements including graphs. A description of how the measurement results were obtained will be included with details regarding how the averaging was performed and whether or not any results were thrown out instead of being included with the averages.
- Comparison of actual results to theory

Timeline:
We plan on creating most of the test tools at the beginning of the project, creating the actual sorting programs and performing the measurements in the middle of the project, and putting together the project report in the final week of the project.

Week of Feb 15th:
Research and plan project
Create test tools and common I/O library

Week of Feb 22nd:
Create MergeSort code and use it to determine the appropriate size of our input arrays
Generate test data with test tools
Create QuickSort code
Create QuickSort 3 partition code

Week of March 1st:
Add pseudo code and examples for the completed sorting algorithms to the report
Create bash script for running repeated measurements
Create Python script for averaging results

Create HeapSort code (time permitting)
Create Distribution Counting code (time permitting)

Week of March 8th:
Troubleshoot any problems
Perform measurements
Create report

Collaboration Plan:
The project proposal and initial draft of the project report will be created using a shared Google doc. Kristina will be primarily responsible for assembling the pieces of the report and performing the final edit, although both of us will make contributions to the body of the text, based upon the algorithms that each of us implements. The final document will probably be copied into LaTeX format for better readability, time permitting. The goal is to have the report largely completed by Tuesday, March 10th in order to allow sufficient time for final edits.

All code used for the project will be in a private GitHub repository. This repo can be made available upon request. Laura will be creating the initial drafts of QuickSort and QuickSort3. Kristina will be creating the initial draft of MergeSort. HeapSort and DistributionCounting will be assigned and created as time permits.

After creating the initial drafts of each sorting algorithm, we will collaborate by meeting in person and walking each other through each line of the code. This will allow us to explain how the algorithm works to each other, obtain a code review from the other teammate, and hopefully catch any errors in the implementation before a lot of testing is performed.

We will perform the initial measurements together in order to finalize our choice of test cases and the size of our data sets. At this time, we will also determine the number of measurements that will go into our test script and any other testing configurations.