Dept. of Computer Science, University of California, Davis
ECS122b Instructor: Rob Gysel April 20, 2016

Program #1: Due 4/27/16 by 11:59pm.

1 Overview

For this program, you will write a pipeline to experiment with sorting algorithms. Programs are to be turned in to Gradescope, which has an auto-grader and immediately gives feedback on your grade. Programs are to be turned in on Gradescope by the due date. **No late homeworks will be accepted**. See the end of this document for a submission checklist.

2 Grading

This program is worth 100 points. 30 points come from your Gradescope submission and 70 points come from interactive grading.

3 Sorting Algorithms

Implement the two algorithms described below. You must have three programs per algorithm: one program that outputs the sorted array (`verbose`), one program for timing the algorithm (`timed`), and one program that runs the algorithm and collects relevant statistics described below (`stats`). Implement your program with either C++, Java, or Python. Each program will take as its first command line argument, the name of a file of integers to sort, and use file I/O to store and sort the file.

You are encouraged to start with the code for the algorithms as presented in the textbook or at `http://algs4.cs.princeton.edu/home/` and modify them as necessary. These files will not be run through MOSS, since they should all be quite similar. Prepare to answer questions about these files during interactive grading. When turning in your programs, include a file named `version.txt` that has exactly one line, describing the language and standard you have used. Use one of the following for this line:

C++ One of C++98, C++11, C++14

Java The output of `javac -version`

Python The output of `python -version`

3.1 Sourcefiles

Quicksort with Insertion Sort Basecase: Quicksort[1] with the following two modifications. First, randomly pick a pivot instead of choosing the last element. Second, use Insertion Sort if the sub-array has k elements or less, where k is specified to the program as a command-line argument. Filenames:

C++: `quicksort_verbose.cpp`, `quicksort_timed.cpp`, `quicksort_stats.cpp`

Java: `quicksort_verbose.java`, `quicksort_timed.java`, `quicksort_stats.java`

Python: `quicksort_verbose.py`, `quicksort_timed.py`, `quicksort_stats.py`

Mergesort with Tripartition: Mergesort with `INFTY` sentinel values[2] and three sub-arrays instead of two. You will submit three files per algorithm.

C++: `mergesort_verbose.cpp`, `mergesort_timed.cpp`, `mergesort_stats.cpp`

Java: `mergesort_verbose.java`, `mergesort_timed.java`, `mergesort_stats.java`

Python: `mergesort_verbose.py`, `mergesort_timed.py`, `mergesort_stats.py`

The verbose files (`quicksort_verbose.cpp`, `mergesort_verbose.py`, etc.) will output the sorted input, one integer per line. Each `*_timed.*` file will not output anything, and is used for timing your algorithm. Each `*_stats.*` will output a collection of statistics, one on each line.
The statistics to report are as follows.

3.2 Mergesort Statistics

See §1.2 Program 1.1 and examine the last for loop, the merge stage. Notice that the algorithm initially starts copying the values of `c` (or `b`) into `a`, and then *transitions* to copying the values of `b` (or `c`) into `a`. Consider the following examples.

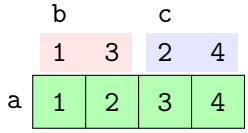

Figure 3.1: An example with three transitions: the merge stage transitions `b - c - b - c` because `a`[0] = `b`[0], `a`[1] = `c`[0], `a`[2] = `b`[1], `a`[3] = `c`[1].

Mergesort: Number of recursive calls, number of transitions between `b` and `c`, and number of compares. Format your output as follows:

```
Recursive Calls:  211425
Transitions:  2893451
Compares:  3583722
```

[1] See §1.5 Program 1.2.
[2] This is presented in the text in §1.2 as Program 1.1.

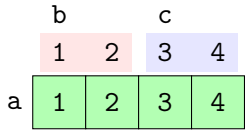

Figure 3.2: An example with one transitions: the merge stage transitions `b - c` because `a`[0] = `b`[0], `a`[1] = `b`[1], `a`[2] = `c`[0], `a`[3] = `c`[1].

3.3 Quicksort Statistics

Quicksort: Number of partitioning stages, number of exchanges, and number of compares. Format your output as follows:

```
Partitioning Stages:  133428
Exchanges:  1061992
Compares:  2199275
```

4 Sanity Checking

Write a bash shell script, `sanity_check.sh`, to sanity check your algorithms. It should do the following:

1. Take the the number of tests (number of files to sort) as its first command line argument, and the number of integers in each sample as its second command line argument. Your script should compile your source files and create executables if necessary.

2. For each i in $\{1, 2, \ldots, n\}$ where n is the number of samples:
   a) Generate a file with the relevant number of integers to sort.
   b) Run both the your `quicksort_verbose` and `mergesort_verbose` implementations and save the output.
   c) Compare the output of your algorithms from the previous step by creating files via output redirection and `diff`'ng them. If the files are the same, delete the output files and continue. If they files differ, save the output as `quicksort_failed_test_i.txt` and `mergesort_failed_test_i.txt`
   d) Clean-up by removing all files generated.

3. Report the results. If the test is successful, report:
   ```
   All tests passed.
   ```
   Otherwise, report the number of failed tests:
   ```
   233 tests failed.
   ```

5 Pipeline

Write a bash shell script to create a pipeline for your experiments. It should do the following.

1. Take the executable to run as the first command line argument, the number of samples (number of files to sort) as its second command line argument, and the number of integers in each sample as its third command line argument.
2. Print the header row to a `.csv` file. Your header row should look like the following for `quicksort` (this is a single line).

   ```
   "Sample Number","Language","Time","Number of Partitioning Stages","Number of Exchanges","Number of Compares"
   ```

   Your header row should look like the following for `mergesort` (this is a single line).

   ```
   "Sample Number","Language","Time","Number of Recursive Calls","Number of Transitions","Number of Compares"
   ```

3. For each i in $\{1, 2, \ldots, n\}$ where n is the number of samples:
   a) Generate a file with the relevant number of integers to sort.
   b) Run and time the `timed` program using the Unix command `time`.
   c) Print the relevant statistics for sample i to a `.csv` file.

Your `.csv` file should share the same base as its corresponding algorithm. For example, the statistics generated by `quicksort_stats.py` should be stored in the file `quicksort.csv`.

6 Plots

Create two scatter plots as pdf's from your data, one from each algorithm, which you will discuss during interactive grading.

1. Go to `https://plot.ly` and create an account.
2. Run your experiment and generate both a `quicksort.csv` and `mergesort.csv`.
3. For each `.csv` file:
   a) Upload the file to `plot.ly`.
   b) Select and open the file.
   c) Select "Scatter plot" on the left menu bar.
   d) Pick any two distinct columns from your csv file as your x and y-axis. Do not pick either `"Sample Number"` or `"Language"` for these columns.

e) Export the scatter plot as a `pdf` file, `quicksort.pdf` or `mergesort.pdf`

f) Turn in your `csv`s and corresponding `pdf`s on Gradescope.

7 Submission Checklist

1. Your source files for both algorithms.
2. Your shell script for sanity checking.
3. Your shell script for running experiments.
4. Your `csv` and `pdf` files. Make sure that the `pdf` files were generated from the `csv` files you turned in.