Introduction to Algorithms, Data Structures, and Problem Solving

DA4002 (HT12) Halmstad University

Assignment for Project 1: Sorting Algorithms Benchmarks

Roland Philippsen
roland.philippsen@hh.se

September 28, 2012

1 Introduction

An important aspect of the ITADS course is to teach participants how to evaluate possible solutions for a given problem. Complexity analysis is a tool within this evaluation process. The lecture and the course book by Loudon [1] discuss several theoretical aspects of computational complexity, with a focus on how runtime grows with problem size. The first project complements the theoretical treatment with hands-on experience. It provides a practical and empirical understanding of the theory, and prepares the participants to make informed choices in real-life programming situations.

The deadline for handing in the source code and the report is **Friday, October 5, 2012, at 18h00**. Teams who miss the deadline will receive a penalty of 5 points (the maximum number of points is 25). In case of exonerating circumstances, such as sickness certified by a medical doctor, a deadline extension will be granted. Participants must notify the lecturer of such circumstances as soon as possible when they arise.

2 Starting Point

Participants are provided with a collection of header and source files that provide fully functional (but limited) benchmarking programs for sorting algorithms. The aim is to determine how long it takes to sort arrays of varying lengths.

Two benchmarking applications are provided in the files `main-insertion-sort.c` and `main--merge-sort.c`. Both use arrays of random integers as input data on a sorting algorithms and measure the time taken to sort the data. They print the measured times to `stdout` (the terminal) in a format appropriate for plotting with the `gnuplot` program. Plotting such data has already been practiced during exercise 7.

Several plotting examples are provided along with the benchmark sources. Look in the `examples/` directory for files ending in `.plot` (there are comments at the beginning of each plot file).

Also as practiced during exercise 7, all code can be compiled with the help of the `make` command which relies on the provided `Makefile`. The `Makefile` is generic enough that you can add new headers and sources and they will automatically be added to the make process, as long as you follow some simple rules:

- Filenames that match "test*.c" or "main*.c" are assumed to be sources for executables (i.e. they must contain a main function).
- All other .c files are assumed to be implementations of functionality required by the executables (or other .c files), in other words they must not contain a main function.

3 Assignment

There is a set of mandatory tasks that have to be performed in order to pass. And there is a list of suggested bonus tasks which can lead to a higher grade. A properly performed set of mandatory

algorithm	**difficulty**	**link**
cocktail sort	easy	`http://en.wikipedia.org/wiki/Cocktail_sort`
selection sort	easy	`http://en.wikipedia.org/wiki/Selection_sort`
gnome sort	easy	`http://en.wikipedia.org/wiki/Gnome_sort`
radix sort	easy	`http://en.wikipedia.org/wiki/Radix_sort`
shell sort	moderate	`http://en.wikipedia.org/wiki/Shell_sort`
comb sort	moderate	`http://en.wikipedia.org/wiki/Comb_sort`
cycle sort	moderate	`http://en.wikipedia.org/wiki/Cycle_sort`
quicksort	hard	`http://en.wikipedia.org/wiki/Quicksort`
heapsort	hard	`http://en.wikipedia.org/wiki/Heapsort`

Table 1: List of sorting algorithms and their difficulty level.

tasks that are well documented in the project report is worth 16 points (HH grade 4, ECTS grade D). Bonus tasks can give up to 9 extra points, such that the maximum achievable number of points is 25 (HH grade 5, ECTS grade A).

Keep in mind that writing the report is an integral part of the project, so do not spend too much time on extending the functionality. It is important to note that teams are expected to manage their resources by themselves. This includes apportioning the time available for finding information online and in the course books, developing and debugging code, running the benchmarks, documenting the work, and preparing and testing the project archive file that you submit for evaluation. How these aspects are shared between the team members is for each team to decide.

3.1 Mandatory Tasks

Table 1 lists sorting algorithms which can added to the benchmark. The (estimated) difficulty of implementing each algorithm is also given, along with a link to a Wikipedia page providing more details. Note that the list excludes bubble sort, insertion sort, and merge sort, because they are either provided as part of the project starting point or have been treated during an earlier exercise.

1. Browse the Wikipedia pages listed in table 1. Then, choose either **two *easy*** algorithms or **one *moderate*** algorithm, and add them to the benchmark. The easiest way to achieve this is to copy the existing `main-insertion-sort.c` file. Choose a file name which reflects the algorithm(s) that you have chosen, for example `main-radix-sort.c`, and also change the name of the sorting function, for example to `radix_sort`. It will be easiest to keep exactly the same function signature, to minimize the amount of changes that you need to do in the `main` function. Also, make sure to replace all other occurrences of "insertion sort," for example in the messages printed to stdout. Don't forget to verify that your implementation is correct, for example by writing a little test application. Finally, produce plots which clearly show the running times of the added algorithm(s) in relation to the ones that were already there. You will probably need to adjust the variables `nstart` and `nmax` in order to get good coverage of the relevant array lengths, without creating a benchmark that takes too long to run.

2. Reduce the effect of variation in the runtime measurements by performing each measurement several times and finding the minimum, maximum, and average runtime required to sort an array of length N. Note that each run for the averaging should use different input data. The easiest way to achieve this is probably to modify your benchmark code to have M input data arrays, and run the sorting algorithm M times for each N. Again, create plots which compare the data with and without averaging, and discuss your findings.

3. Determine how well the measured runtimes match the Big-Oh complexity classes for all of the sorting algorithms that you have (merge sort, insertion sort, and the one(s) that you implemented). This was discussed in lecture 4 and practiced during exercise 7. Create corresponding plots and discuss your findings in the report.

   Hints: if you get really noisy runtime measurements, it is probably best to use the minimum instead of the average value, as that tends to be more stable with respect to variations caused by other processes running on the machine where you execute the benchmark.

4 Bonus Tasks

1. Extend the benchmark to illustrate the dependency of the runtime on input data distribution. Do this by also measuring the time it takes each algorithm to process data which is

   - already sorted
   - sorted in reverse
   - partially sorted

   Create clear plots which illustrate the differences between the sorting algorithms for all the classes of input data. Note that the files `random.h` and `random.c` provide a utility function that can easily be used to generate the above types of data, and that the `test-random.c` program is a good illustration of how to use them.

2. Add two *moderate* or one *hard* algorithm from table 1 to the benchmark.

5 Further Information

How to Save the Runtime Measurements

As practiced during exercise 7, you can use redirection to capture the output of a program into a text file. In order to see the output on the terminal while also capturing it into a file, you have to "pipe" the output through the `tee` command, and tell the latter which filename you'd like to use to store the captured output.

For example, use "`./main-merge-sort | tee msort.data`" to run the merge sort benchmark and save the resulting timing measurements into the file `msort.data`. Notice that **this will overwrite the contents of `msort.data`**. This is a desired effect if you're just experimenting, for example you'll probably have an interactive gnuplot session open in another terminal and just use the cursor keys to plot the data again and again. But this overwriting is not nice if `msort.data` contained measurements that you wanted to include in the report.

Thus, once you are happy with a captured data file, you should copy it and keep it safe. You do this using the `cp` command (or you can use the graphical file manager). For example, the command "`cp msort.data good-msort.data`" will make a copy of `msort.data` in a file called `good-msort.data`.

Notice that it is a very good idea to submit copies of the data which you use to produce the figures in the report. The same goes for plot files (see next section).

How to Use Gnuplot Script Files

Gnuplot can be used interactively (like in exercise 7) by just giving its command in a terminal. But it can also run in batch mode by passing the name of a plot script file on its command line. A gnuplot script file is just a text file with a list of gnuplot commands. The `examples/` subdirectory of the project contains several such scripts. Notice that those examples which create a figure window must be launched with the additional option "`-persist`" (otherwise the figure just flashes briefly on the screen). This is explained in the comments of the relevant .plot example files. Notice that there are also example plot files which save the figure in a PNG image, intended for inclusion in your report.

If you want to use one of the example plot files, but modify it to suit your needs, it is easiest to copy the scripts and then modify the copy. For example, you can do "`cp examples/isort-example.plot radix-sort.plot`" and then edit the `radix-sort.plot` file in a text editor. Things to change are: the name of the data file, the title, possibly the data column numbers, and maybe the output file name when creating PNG images. Please read the gnuplot documentation [2] for more details.

Also notice that you are free to use some other software to plot your data. For example, if you are familiar with Matlab, you may find that much easier to work with. Excel may be another viable option. But you will probably need to adjust the output format of the benchmarking applications.

Preparing Archives for Submitting Your Project

You already encountered the `tar` program in exercise 7 to extract project archives. In order to create an archive, use the command "`tar xfvj archive-name.tar.bz2 project-directory`" where you have to replace `project-directory` with the name of your project directory, and the result will be in `archive-name.tar.bz2`.

However, please don't choose just any name. Rather, make a separate directory based on the family names of the team members, copy all the source and data files there, then create the archive with an archive name that has the directory name as its base.

For example, two students named "Alice Foo" and "Bob Bar" would proceed as follows:

1. Create a directory based on student and project names.
   For example: "`mkdir itads-proj1-foo-bar`"

2. Copy all the relevant files there.
   For example: "`cp *.h *.c Makefile itads-proj1-foo-bar/.`"

3. Create the archive: "`tar cfvj itads-proj1-foo-bar.tar.bz2 itads-proj1-foo-bar`"

4. **IMPORTANT!** Double-check that the files you placed in the archive are complete and free of errors. This is best done by going into the directory which you just used to create the archive, and trying to build and run your benchmarks. For example:

   (a) "`cd itads-proj1-foo-bar`"

   (b) "`make`"

   (c) "`./main-radix-sort`" (or whatever you named your additional bechmark applications). Be sure to check each of your benchmarks.

References

[1] Kyle Loudon. *Mastering Algorithms with C*. O'Reilly Media, 1999.

[2] Thomas Williams, Colin Kelley, and many others. Gnuplot. `http://www.gnuplot.info/`, September 2011.