Introduction to Algorithms, Data Structures, and Problem Solving

DA4002 (HT11) Halmstad University

Assignment for Project 1: Complexity Analysis

roland.philippsen@hh.se

September 21, 2011

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 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, September 30, 2011, at 18h00**. Teams who miss the deadline will receive a penalty by lowering their grade by one. 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 Assignment

Participants are provided with a collection of classes and two fully functional example applications [1]. The first is a benchmark for container operations, the second is a benchmark for sorting algorithms. All code can be compiled with the help of the command "`make`" which relies on the provided `Makefile`.

2.1 Overview

Each team first chooses one of the two provided examples as starting point. This means you will either investigate the time complexity of container operations (Alternative A), or the complexity of sorting algorithms (Alternative B). The first set of mandatory tasks for each alternative will be to extend the provided code with additional implementations. The remaining "bonus" tasks will be to make the benchmark more complete, and there are several possible ways of achieving this. For example, add different types of input data, or compare theoretical with practical complexity. More details about the mandatory task will be provided in sections 3.1 and 4.1. Possible avenues for the bonus tasks are discussed in section 5.

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. Well-performed mandatory tasks that are properly documented in the report will receive a good grade. In order to achieve an excellent grade, at least one bonus task needs to be done.

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 book, developing and debugging code, running the benchmarks, and documenting the work. How these aspects are shared between the team members is for each team to decide.

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`
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 which can be implemented as part of alternative B.

3 Alternative A: Container Operation Benchmark

The aim of this benchmark is to determine how long it takes to perform fundamental container operations, such as insertion and removal of elements. The provided application is in `ExampleA.java`. It is limited to two container implementations (doubly-linked lists and dynamically expanding arrays). It is also limited to just a single container operation (repeated insertion at the front or back of the container).

Example A is launched by the command "`java ExampleA`." The application relies on reusable code for logging execution times, saving these logs to a file, and producing figures from the data. Note that the name of the files containing the data log and the plot scripts changes each time that `ExampleA` is run (see section 6 for more details).

3.1 Mandatory Tasks for Alternative A

1. Implement a binary search tree which stores strings, and measure how long it takes to insert N random strings into it. You can use the code of `Dictionary.java` provided in exercise 3.3 as a starting point, but beware that for this project we only store a value, instead of a key-value pair. Verify that your tree functions correctly, and produce benchmark plots that clearly illustrate the different growth rates of item insertion for trees when compared to sequence containers.

2. Extend the benchmark to also measure the time it takes to remove N items from each container. Choose fast operations, such as `popBack` for the list and the vector, and `removeMin` for the binary search tree. First insert N items without measuring that time, and then measure the time required to remove all items one by one. Produce a benchmark plot which clearly illustrates the differences between the remove operation based on container type.

3. Reduce the effect of variation in the runtime measurements by performing each measurement several times and taking the average. The `LogSeries` class, which is used to log the actual runtimes, provides 4 methods to help with this task. Find those methods, read their documentation, and apply them to the benchmark. Create plots which compare the data with and without averaging, and discuss your findings.

4 Alternative B: Sorting Algorithm Benchmark

The aim is to determine how long it takes to sort sequences of varying lengths. The provided application is in `ExampleB.java`. It is limited to two search algorithms (insertion sort and merge sort). It is also limited to just a single class of input data (arrays of random strings).

Example B is launched by running "`java ExampleB`." It relies on the same logging and plotting functionality as example A.

4.1 Mandatory Tasks for Alternative B

Table 1 list some sorting algorithms which can added to the benchmark[1]. The (estimated) difficulty of implementing each algorithm is also given, along with a link to a Wikipedia page providing more details.

1. Read the Wikipedia pages listed in table 1. Then, choose either **two *easy*** algorithms or **one *moderate*** algorithm, and add them to the benchmark. Produce plots which clearly show the running times of the added algorithm(s) in relation to the ones that were already there.
2. Extend the benchmark to also measure the time it takes each algorithm to process data which is

   - already sorted
   - sorted in reverse

   Create clear plots which illustrate the differences between the sorting algorithms in all three cases of input data (random, sorted, and reversed).
3. Reduce the effect of variation in the runtime measurements by performing each measurement several times and taking the average. The `LogSeries` class, which is used to log the actual runtimes, provides 4 methods to help with this task. Find those methods, read their documentation, and apply them to the benchmark. Create plots which compare the data with and without averaging, and discuss your findings.

5 Bonus Tasks

Choose **one** of the following bonus tasks in order to achieve an excellent grade. You are of course free to perform more than one bonus task, but keep in mind that documenting your choice and its results will also take some time. The first list of bonus tasks is applicable to either the container or the sorting benchmark. The following two lists are specific to each alternative.

- Make the data containers or sorting algorithms generic. Then create a benchmark which determines whether there is a significant performance difference between containers which store `String` values and containers which store `Integer` values. You will need to extend the `Factory` with a method to create arrays of random integers, similar to the one which creates random strings. Create plots which clearly demonstrate your results.
- Determine how well the running times match the typical complexity classes, as shown during the lecture and practiced during exercise 4. This is done by computing the theoretical growth rate according to the $F(N)$ of the $O(F(N))$ expression, dividing the actual runtime by it, and detecting whether that ratio converges to a non-zero value. Create corresponding plots and discuss your findings in the report.

For Alternative A

- Investigate if and how the running times depend on the order of the input data. Create benchmarks for inserting already-sorted data, data which is sorted in reverse, and data which has many duplicate items. Produce benchmark plots that clearly demonstrate the different running times, and discuss the measured effects.
- Compare the running times for finding strings, using the `find` methods of `StringList` and your binary search tree implementation. The `Factory` provides a method which makes it easy to generate a list of strings to find: "`Factory.createSamples(data, 1000, 0.1);`" will create an array with 1000 strings, where 90% are taken from the provided `data`, and the remaining 10% are randomly generated. Verify that your `find` method works, and produce

[1]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 exercise.

benchmark plots that clearly demonstrate the different running times of finding items in lists and trees.

- Add a hash table and a balanced binary tree from the Java Collections library to the benchmark, similar to exercise 1.16. Produce benchmark plots that clearly demonstrate the different running times, and dicsuss the measured effects.

For Alternative B

- Add two *moderate* algorithms from table 1 to the benchmark.
- Add one *hard* algorithm from table 1 to the benchmark.

6 Additional Remarks

6.1 Managing the Automatically Generated Files

The provided code for logging and and producing figures has a tendency to clutter the working directory with many files. The idea is to never delete data or plots, unless explicitly desired by the user. In order to simplify the management of these files, they follow very strict naming conventions.

prefix: all automatically produced files begin with `log-`

timestamp: number which changes each time the application is run, for example `1316351173681`

optional annotations: short dash-separated tokens such as `all-scr` which are used to separate different types of plots from each other

extension: `.data` denotes a text file with logged data, `.plot` is a plot script which reads the corresponding data file to produce a figure (either on screen or as a PDF), and `.pdf` is a figure produced by the corresponding script file.

It is recommended that the “good” data, plot scripts, and the produced figures are manually copied to a separate directory in order to save them. Also, it is a good idea to rename them in the process, for example to indicate what is on the plot. Then, it will be much easier to include the data and plots in the report.

For example, suppose that `log-1316351173681.data` contains the data of a benchmark run which was particularly good at showing some effect. Then, the following sequence of UNIX commands will save the data and the associated plots in a separate directory with a more descriptive name:

```
gnuplot log-1316351173681-all-pdf.plot
gnuplot log-1316351173681-sep-pdf.plot
mkdir good-example-directory-name
mv log-1316351173681* good-example-directory-name/.
```

Of course, adapt the directory name to something more meaningful first. Then, it will be possible to remove most (or even all) of the `log-` files to make the project directory more tidy.

6.2 Customizing the Plot Scripts

The automatically generated scripts for plotting the data logs are text files with `gnuplot` commands. It is possible to edit these scripts in order to customize the plots. It is also possible to run `gnuplot` in interactive mode as in exercise 4.2, and copy-paste-adapt individual commands from the script files. To enter interactive mode, simply launch `gnuplot` without arguments.

Gnuplot comes with extensive online documentation [2], and it has a built-in `help` command, too.

Example: Merging Parts from Different Plots

Suppose there are two log files `log-123.data` and `log-456.data` and their corresponding plot scripts. You wish to create a figure with the first data set from `log-123.data` and the second data set from `log-456.data`.

1. choose one of the plot files as starting point and copy it to a new file, for example:
   `cp log-123-all-scr.plot log-fused.plot`
2. open `log-fused.plot` in a text editor and remove the parts you do not want to plot anymore
3. copy-paste the relevant plot commands from the other script (`log-456-all-scr.plot` in this example)
4. safe `log-fused.plot` and run it using "`gnuplot log-fused.plot`"

Note that the `plot` command creates a fresh plot. If you want to have several data sets on the same plot they must come as comma-separated entries on one line.

References

[1] Roland Philippsen. Itads project 1. `http://tinyurl.com/itads-ht11-p1`, September 2011.

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