# DM207 I/O-Efficient Algorithms and Data Structures

Fall 2015

Project 1

Department of Mathematics and Computer Science
University of Southern Denmark

October 7, 2015

## Introduction

The goal of this project is to get a feeling for the possible impact of differences in memory access patterns on the running time of programs.

The project is to be done in groups of size two (although single person projects are allowed).

## General remarks

The programming language should be Java (unless something else is agreed upon with the lecturer). The tasks below should each be solved by a simple programs. Each program should time its own *core activity*, excluding creating and filling arrays, writing results, etc. The timing should be done using two calls to `http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#currentTimeMillis()`, one just before the core activity and one just after.

Each program is to be run for many values of the input parameter (usually input size $n$), and plots of this are to be made, with the parameter along the $x$-axis and time divided by expected running time (often $n$ or $n \log n$) along the $y$-axis. For better stability of plots, each run should be repeated 5 times, smallest and largest value discarded and the average of the remaining three should be used.

It will be beneficial to create scripts (possibly in another language such as Python) executing all runs of each Java program for all parameter values,

finding the averages of the measurements as described above, and making the plots.

Make sure that you during development test that the output of each program is correct. Measuring algorithms not working correctly will teach you nothing.

It is legal to base your programs on existing code from the web. In particular, explicit references are given below to Java code by Sedgewick and Wayne. For each reference, grab the essential part of the code, and if necessary rewrite it slightly to suit your needs, and to avoid reliance on helper methods and libraries by the two authors.

In order to let the Java runtime utilize more of the RAM on your machine, it may be necessary to increase the "heap size" at invocation using the `-Xmx` option (see e.g. `http://stackoverflow.com/questions/1565388/increase-heap-size-in-java`).

In order to better discuss your plots, it may be beneficial to investigate the amount of cache (L1, L2, possibly L3) and RAM, as well as cache line sizes, on your computer. Use the Web to find info on how to do this on your platform.

## Tasks

1. Implement *Quicksort* and *Heapsort* on `ints`. For the former, you can base your code on the core parts of `http://algs4.cs.princeton.edu/23quicksort/Quick.java.html`, and for the latter on `http://algs4.cs.princeton.edu/24pq/Heap.java.html`. In both cases, you will need to change the type `Comparable` to `int`. It is also important to remove the line with the `shuffle` operation (which is essentially random memory accesses, and which will perturb the measurements).

   Each program should start by creating an array of size $n$, filled with random `int`'s (use for instance `http://docs.oracle.com/javase/7/docs/api/java/util/Random.html`). The core part (to be timed) of the program is only the sorting itself.

   Run each program for values of $n$ between approximately 1.000 (4 Kb) and 500.000.000 (2 Gb), increasing by a factor of 1.7 each time. Plot for each program the running time divided by $n \log n$ as a function of $n$

   Discuss your plots.

2. Make a copy of your *Quicksort* program which sorts an array of `Integer` objects (again initialised to random integer values) instead of an array of `int`'s (a Java primitive type, i.e., not an object). The core part is again only the sorting itself.

   Run the `int` program for values of $n$ between approximately 1.000 (4 Kb) and 500.000.000 (2 Gb) and the `Integer` program for values of $n$ between approximately 1.000 and 30.000.000 (`Integer`'s use much more space than `int`'s), in both cases increasing by a factor of 1.7 each time. Plot for each program the running time divided by $n \log n$ as a function of $n$.

   Discuss your plots (hint: how are objects in Java stored in memory, what is an array of objects?). For a slightly more pronounced effect, shuffle each array before sorting (outside of the timing), for instance using Sattolo's algorithm (see next task). Why will this make the effect more pronounced?

3. An array $A$ of length $n$ containing the integers $0, 1, \ldots, n-1$ can be viewed as a set of cyclic linked lists of array cells: if $A[i]$ is equal to $j$, then array cell $i$ points to (has as successor in list) array cell $j$. In particular, if there is only one linked list in total, the array is called an $n$-cycle.

   Create a program which allocates an array of `ints` of length $n$, and makes it into the specific $n$-cycle given by $A[i] = i+1$ (and $A[n-1] = 0$).

   Create another program which allocates an array of `ints` of length $n$, and makes it into a random $n$-cycle by using *Sattolo's algorithm*, which can be found at `http://algs4.cs.princeton.edu/11model/Sattolo.java`.

   Each program should then start at $A[0]$ and follow the cycle (reading the contents of an array cell to find the next array cell) for $m$ steps. Only this is the core part of the program to be timed.

   For values of $n$ between approximately 1.000 and 500.000.000, run each program for $m = 1.000.000.000$ steps of the cycle. Plot the running time divided by $m$ as a function of $n$.

   Discuss your plots.

4. Create a program which allocates an array of `ints` of length $n$, and fills it with random `int`'s.

The program should then touch array cells with indices $0, d, 2d, 3d, \ldots$ for a parameter $d$, in a circular fashion, returning to index 0 when the end of the array is reached. This should done for $m$ steps in total (suggestion: create a double for-loop, with the inner loop being a single pass of the array, and the outer loop repeating this $m/(n/d) = dm/n$ times). The touching can be adding the contents of the array cell to a global variable.

Only the circular touching of the array is the core part of the program to be timed.

Run the program for $n = 2^{29}$ (1 Gb), $m = 2^{30}$ and $d = 2^i$ for $i = 0, 1, 2, 3, \ldots, 28$. Plot running time divided by $m$ as a function of $i$.

Discuss your plots (hint: what is the L1, L2 (and L3, if available) cache line size and cache size of your machine)?

5. Boot the machine (or another machine, such as one in the IMADA Computer Lab) with 1 Gb of RAM [see below how] and run experiments again with values of $n$ from 10.000 to 400.000.000 (except for the `Integer` sorting, where you should stop around 20-30.000.000). You should also set the java "heap size" to 4 Gb by using the `-Xmx` option. Note: some experiments (*Heapsort* and the random cycle) may not terminate in reasonable time (such as one hour) for large $n$. If so, then stop the program[1] and just report this observation. For this task, it is for time reasons OK just do each run once (not averaging over three out of five). Also for time reasons, in task 4 set $m = \min(d, 2^{16})$, as larger $d$ will mean a significantly larger time per touch (don't forget that you should plot running time divided by $m$, where $m$ is now varying).

## Formalities

Make a report of 8–10 pages describing your experiments and the characteristics (such as disk, RAM, and cache sizes) of the machine used for these. Any further details from the implementation besides those from the task description above should be mentioned. Use plots of your experimental data (not tables), and make sure it is explained what they show. Draw conclusions based on the observed data. Source code should not be included in the report, but the source files should be part of the hand-in.

[1] Command `kill -9 ProcessID` is useful here, where ProcessID can be found using the command `top`.

You should hand in your report (in pdf) and your source files as one zip-file using *SDU Assignment* at the Blackboard page of the course.

The project will be evaluated by pass/fail grading. The grading will be based on:

- The clarity of the writing and of the structure of the report.
- The thoroughness of the experiments—execution as well as discussion.

Deadline:

**Monday, October 19, 2015, at 23:59.**

## Booting with less RAM

The following can be done on machines in the terminal room (and probably in a similar way on any Ubuntu Linux installation). It will make the operating system only see the specified amount (1 Gb in the example) of RAM. In the terminal room, don't use any of the machines marked "don't shut down", and don't use any machine with a running batch job from another user (check first few lines of output of the `top` command for other (non-system) user names).

1. Push briefly the physical power button on the machine.
2. Choose "Shut down" from the appearing menu on the screen, and wait for the shut-down to complete.
3. Then press press the power button again to start up the machine. Just after the HP splash screen goes away in the beginning of the boot process, press down the Esc key. The ASCII-based GRUB menu appears after some time.
4. Press `e` for edit, move (with arrow keys) to the long line starting with `linux`, and move to the end of this line with Ctrl-e (or arrow keys).
5. Write (append) the string " `mem=1024M`" to end of line. (The character '`=`' will be on the key just left of the backspace key.)
6. Press Ctrl-x, and wait for the boot to complete (note: less RAM means slower Ubuntu interface).

7. Perform the experiments.

8. Shut down and start the machine again as usual (with no editing of the GRUB menu). This will make the machine return to its default settings.[2]

[2] I experienced that keyboard input was based on US keyboard layout until yet another reboot—if the same happens to you, reboot twice.