GSB/RF
September 4, 2000

Project 1 — Sorting

Part A

The first project will be given as two smaller parts. At the end of the project each project group (2–3 persons) is expected to hand in a report documenting the work done, the behavior observed, and the possible explanations of this behavior.

The project is centered around *sorting*. The goal is *not* to do major implementation work, but rather to

- gain some experience in measuring the performance of actual programs.
- make observations about the influence of the memory hierarchy on the running time of implementations of simple algorithms—specifically to see whether individual components of the memory hierarchy show themselves when the input size grows.

This first part of the project consists of the tasks below. Monday, September 18, we will spend some time on discussing any problems that might occur while addressing these tasks. It is expected that by then *everybody have obtained some preliminary results.* In other words, start programming right away, even though the final report is only to be handed in later.

Tasks:

1. Write a simple C program that takes two arguments n and r and allocates an array of n integers and r times runs trough the array and increments each of the n entries by one. The program should not generate any output.
2. Make a similar program that $n \cdot r$ times increments a *random* entry of the array.
3. Run your programs for different values of n and measure the time used by the program. Choose r such that running time becomes acceptable.

   Identify the properties of the measured results and try to give an explanation of the measured times, especially how the memory hierarchy influences the growth in the running time when the problem size n increases. Repeat your experiments on a few different machine architectures (e.g. Sun, HP, SGI, PC).
4. Implement in C the standard Heapsort algorithm that sorts an array of n integers. A description of this algorithm can be found in almost any textbook on algorithms, including [1] and [2] (the latter without the linear time heap construction algorithm—but building by repeated insertions is probably fine for our purposes).
5. Run your program for different problem sizes and measure the running time of your program. Explain the bounds.

Hints:

- There are quite a number of UNIX commands to time the execution of a program, among these `time` and `timex`. Read the man page for usage of these. In some shells (e.g. csh), there is also a built-in version of `time`—read the man page for the shell for information on this. There are also corresponding C library routines called `times` and `getrusage`, (which also have man pages). The availability may differ from machine to machine, though. It is part of the project to consider how to best time the execution of your programs.

- If there are too many program instructions per memory access in your program, the cost of going from cache to RAM may be hard to see in the running times. Your programs should therefore be as optimized as possible, even at the cost of normal programming conventions. In-line functions, use global variables, minimize the number of `if/else`'s etc. Also, set the compilers options for maximal optimization.

- Make graphs of the results, e.g. using `gnuplot`. A sensible value to plot could be running time divided by asymptotic complexity (e.g. n or $n \log n$), with n on a logarithmic x-axis.

- Run the program a suitable number of time for each input size and use the average running time. This should level out fluctuations due to other processes.

- Your program should not produce any output (as output is extremely slow compared to access to e.g. RAM). However, during development you should make sure that your program is actually doing what it is supposed to do. One way of achieving this is to incorporate a *checker*—i.e. a routine *verifying* the desired property, e.g. that the array is actually in sorted order after performing Heapsort. An $O(n)$ time routine for checking sortedness is easy to make (how?). The same holds for checking heaporder in an array.

- *Do not run experiments on file servers!* Rather, try to find a machine with no other user processes executing.

- The file `/users/verksted/Maskiner/udstyrs_reg` contains information about the hardware configuration of the machines at DAIMI. Under Linux the files `/proc/cpuinfo` and `/proc/meminfo` contain information about cache and memory size.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 7. MIT Press, Cambridge, Mass., 1990.

[2] Erik M. Schmidt, and Michael Schwartzbach. *Programmeringsteori og Datastrukturer*, section 2.2. DAIMI FN - 56, January 1997.