I2A: Notes to go with "slides 1" & "slides 2"

Richard Bornat, January 1998

Because there is no set book (at the time of writing I'm not sure when the book will be published in the UK), I'm taking the unusual step of writing some notes to go with the first batch of slides. With luck, the book will be available within a week.

Introduction

This course is about small programs, examined very closely, about beauty found in unexpected places, about general solutions, and about measuring the amount of work a computer can do in a particular amount of time or space. We shall start with very simple problems, and find much more than you might expect in them. We are often going to deal with very fine distinctions, which the non-specialist, the non-computer-scientist, might think irrelevant. We can't apologise for that: it's the essence of our subject.

The course is about 'analytical programming'; it's useful because it introduces you to a lot of ways of thinking about programs which you haven't written yet; it's interesting because it gets you quite quickly to areas where surprisingly little is known; it's hard, and it's interesting because it is hard.

For that reason we shall have time to look at only two problems:

- sorting;
- searching.

The slides will contain 'side notes', which are points intended to provoke reflection and argument. Reflective learning is, apparently, much more efficient than rote learning.

If you want to read around the material in this course, any book with "algorithms" in the title would be appropriate – especially those by Weiss and Sedgewick, recommended last year and mentioned on the course information sheet, because the C++/Java differences aren't too great to begin with. Some books on "specification" might help, but many will be a little too detailed and/or technical. Books about on the specification notation *Z* will help with predicate calculus, but only after you have done a little set theory in the IDS course.

Correctness matters more than efficiency

Obviously it does – who wants a fast program that doesn't work? But which should we concentrate on first: correctness or efficiency?

In the Landin diagram (see slides) route *cd* takes us first to a fast program which doesn't work – usually to one which works some of the time – and then tries to iron out the bugs or extend the range. Route *ab* takes us first to a slow program which works everywhere, and then tries to refine the bits which slow the program down.

Route *ab* is better because:

- speeding a program up makes it more complicated – that makes it harder to find bugs on leg *d* than on *a*;
- you never need to speed up every bit of the program – but until it's working you don't know which bits you must work on and which you can ignore, which makes leg *b* shorter than *c*.

So I shall continue to emphasise correctness whenever it is possible to do so. And in tests we shall ask as many questions about correctness as about efficiency, or perhaps even more.

Some properties of specifications

1. They are *conditional*, *hypothetical*, $A \rightarrow B$ remarks - *if* the input is like this *then* the output must be like that. A washing machine is specified to wash your clothes *if* it is supplied with electricity, water, soap (and clothes), and not otherwise. If you don't plug it in, it won't work – and *that's ok*.

2. Read strictly, a specification says *nothing* about what should happen if you give input that doesn't correspond to its pre-conditions. In those circumstances the machine/program can do anything at all *and still satisfy the specification.*

2a. In technical language, specifications are *satisfied* or *not satisfied.*

2b. A program might even blow up the computer if you give it the wrong input (we shall try not to write programs which are so silly, but things like that happen and are allowed according to the meaning of 'specification').

3. A specification gives a *minimum* requirement, which a particular program can *exceed* and yet still satisfy the specification. Our algorithms might do their work and at the same time record some statistics – unless the specification prohibits that, it's ok.

3a. If a washing machine magically worked even though it wasn't connected to the electricity, would you complain? I wouldn't. It would be *exceeding* its specification.

3b. I might complain if it telephoned MI5 and told them what I was washing, though.

4. A specification is independent of a program. We shall see several sorting algorithms, in various versions, all of which can be claimed to satisfy our specification.

5. A specification is always more or less mathematical, but it isn't always written in mathematical notation.

5a. Logical notation is a kind of mathematical notation. Learn to use the word 'notation'.

5b. English isn't a kind of mathematical notation, but sometimes it's the best we can do.

5c. Pictures are often the best specifications.

6. Miranda programs are more concise than (and perhaps more mysterious than) Java programs, so sometimes we can write a Miranda program to stand as the specification of a Java program.

6a. We can make chains of specifications in this way. A Java program might specify an assembly-language program, a Miranda program might specify the Java program, there might be a higher-level program which specified a Miranda program, and so on.

7. We might hope to *prove* that our programs satisfy their specifications (but we won't do that in this course).

7a. We can't hope to prove that our specifications describe the algorithm we are thinking about – and that's a major problem with the mathematical approach.

7b. Our specifications don't always say "and that's all there is to say", which can be a problem if our programs have extra, unspecified, undesirable, effects.

Calculating costs of assignment

We are going to use as a model of a computing machine the sort of machine described in the CS1 course. But to keep every action of the machine out in the open, I'm going to outlaw 'register offset' addressing - stuff like `r2(7)` or `sp(-6)`, because each of those means a hidden addition operation, and we can't have any hidden operations.

But actually this detail *doesn't matter* in the context of this course, because ***we make a major simplification when assessing costs***. All we need to know is that to execute `A[i]=A[i-1]` we will need to do a bit of address manipulation (involving addition and subtraction), and a bit of moving (involving store accesses). And then we know that modern computers are designed so that ***arithmetic and store access are constant-time zero-space operations***. For example, on a CS1-style machine the operation `ADD r1, r2` always takes the same amount of time no matter what numbers are in `r1` and `r2`. Ditto for `SUB`, `DIV` and `MULT` - arithmetic always takes the same amount of time no matter what the operands.

Speed-up of linear and quadratic programs

You should attempt, in the lab, to verify (or repudiate) claims made in the lectures about execution times, speedups and the like.