COSC242 Administrivia

Your teaching team:
Brendan McCane (lectures), Iain Hewson (labs).

Recommended Textbook:
Cormen, Leiserson, Rivest, and Stein: *Introduction to Algorithms*, any edition, MIT Press.

Lecture notes, Lab book, and Assessment:
Available on the COSC242 coursework webpage:
http://www.cs.otago.ac.nz/cosc242/

Honour System:
Our students don't do silly things such as copying assignments or removing reference books from the Labs. Nor do we mess up the place.

Optional tut: Mondays 4pm in G34 - no new work, just exercises. I am willing to have a similar tut on Tuesdays if there are students who can't make it on Mondays.

Overview

COSC242 has several goals:

- to foster a mindset that focuses on the efficiency of your programs;
- to introduce more advanced algorithms and data structures than those you met in COSC241;
- to give you experience in building both data structures and algorithms from scratch;
- to help you learn a new programming language (C).

The learning of C will happen in the labs. Sometimes we'll do efficiency-related stuff in the lectures and practical stuff in the labs, so the labs and the lectures won't seem to match up — but this is all part of the plan. Attending both lectures and labs is very important. The programming assignment will be easy if you have completed the labs leading up to it, but very difficult otherwise. And the exam is based on the lectures, particularly on worked examples from the lectures. The lectures are not recorded.

Algorithm Analysis

COSC241 introduced you to the analysis of algorithms, and made the following points:

1. To calculate the time efficiency of a program, ignore implementation details (e.g. clock speed). We even ignore the programming language used. So we'll talk of *algorithms* instead of programs.
2. How long an algorithm takes depends on how much work the algorithm has to do.
3. The amount of work is the number of basic steps the algorithm takes.
4. We want to relate the amount of work to the number n of data items that need to be processed (the *problem size*).
5. We want to compare the work this algorithm does with that of other algorithms for producing the same result.

NB: As professional programmers, we will always be part of a team, so we need to know not only how to calculate the amount of work but how to share our calculation with co-workers in a clearly understandable way.

Algorithms

Programs are written in a programming language such as C or Java; all details must be spelled out for the compiler.

Algorithms are written in **pseudocode**: English enriched by symbols so we can clearly understand the essence of what must be done without drowning in detail.

Example: an algorithm for Insertion Sort

```
for i $\leftarrow$ 1 to n - 1 do
  key $\leftarrow$ A[i]
  j $\leftarrow$ i - 1
  while j $\geq$ 0 and A[j] $>$ key do
    A[j + 1] $\leftarrow$ A[j]
    j $\leftarrow$ j - 1
  A[j + 1] $\leftarrow$ key
```

(Note: This differs from the textbook, whose array indices go from 1 to n while ours go from 0 to $n - 1$. And I've swapped i with j.)

How much work insertion sort does

So how much work does insertion sort do? And how does this compare with other sorting algorithms?

Exercise: Trace the operation of Insertion Sort on each of the following input arrays:

1. $[1\ 2\ 3\ 4\ 5]$
2. $[5\ 4\ 3\ 2\ 1]$

Which exercise represents a best, and which a worst case?

In each case, how many comparisons have to be made between the value *key* and array entries when i = 1? And when i = 2? And when i = $n-1$?

So what is the total number of comparisons in the best case? In the worst case?

Important questions:

- Is the amount of work done by insertion sort a lot or a little? What benchmark can we use?
- Is it the amount of work for a specific size of input that we care about, or whether the algorithm scales up?

Landmarks

We saw that the amount of work done by Insertion Sort, in the worst case, is roughly indicated by
$f(n) = 1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2 = (n^2 - n)/2$

We'd like to tie this in to some special **landmark functions,** which are given by assigning to input n the outputs

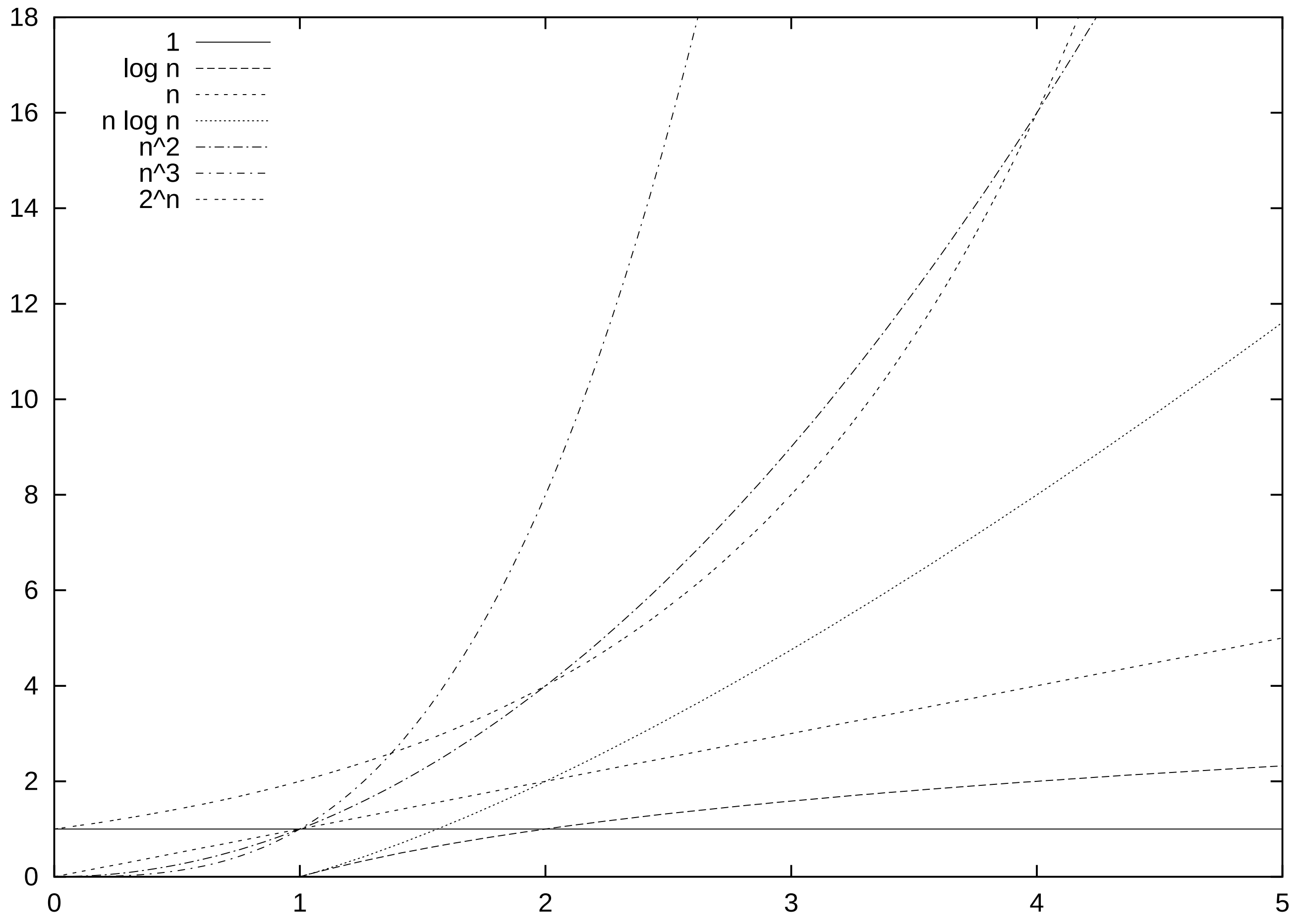
$$
\begin{aligned}
&f(n) = 1\\
&f(n) = \log n\\
&f(n) = n\\
&f(n) = n \log n\\
&f(n) = n^2\\
&f(n) = n^3\\
&\ldots\\
&f(n) = 2^n\\
&f(n) = n!
\end{aligned}
$$

These simple functions are landmarks in the sense that we use their rates of growth to group other functions into classes. Functions that most closely resemble, say, n^2 form one class, those resembling n^3 another, etc.

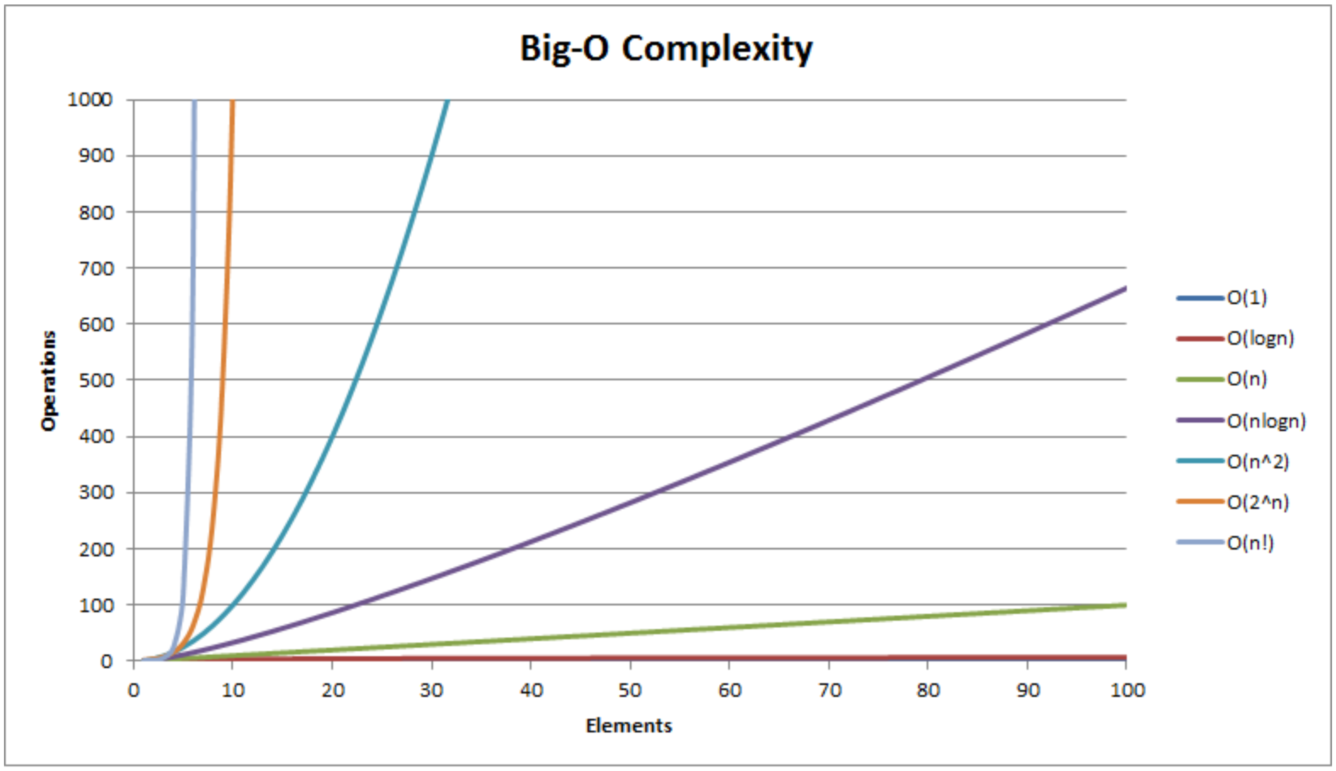
By a function's *rate of growth* we mean how fast its output $f(n)$ increases in size as the input n gets bigger. Intuitively, a function with a slow rate of growth scales up better than a function having a high rate of growth.

Rates of Growth

As n gets bigger, $f(n)$ may increase slowly or rapidly.



Rates of Growth (Zooming Out)


Big-O Complexity
Operations
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)