Maths for DAPA

This note describes some of the important mathematical notation and facts with which you will need to become familiar as you study DAPA. In an attempt to be pragmatic and useful, it will deliberately **not** provide full in-depth discussions, proofs from first principles, generalisations and so on. If you are interested in such material you could consult any decent standard text on algorithms (for example, Cormen et al, Introduction to Algorithms).

1 Interesting Functions

Most of the results in DAPA boil down to comparing the performance of algorithms on the basis of their run-times, expressed as a functions of problem size (usually denoted by n). It is therefore important to have a feel for the relative significance of the functions which typically occur. In decreasing order of "bigness" these include

- *exponential* functions, such as 2^n (the n is in the "exponent")
- *polynomial* functions, such as n^2 (a constant exponent ≥ 1)
- *polynomial* functions, such as $n^{\frac{1}{2}}$ (a constant exponent < 1). Note that $n^{\frac{1}{2}} = \sqrt{n}$.
- *polylogarithmic* functions, such as $\log^k\ n$ (constant exponent of a log, meaning $(\log n)^k$, not $\log(\log n)$, which is different and will not occur in DAPA)
- *constant* "functions", such as 6 (i.e. independent of n)

To give you a flavour of the relative speed with which these functions grow, consider figures 1 and 2. Notice that these have been drawn to different scales!

2 Facts about logs and exponentials

In DAPA we will only ever come across base 2 logs, so whenever I write log you can assume that's what is meant. Here are some useful facts

- $\log a + \log b \;=\; \log(ab)$
- $\log a - \log b \;=\; \log(a/b)$
- $2^{\log\ n} = n$
- $(2^a)^b \;=\; 2^{(ab)}$
- $(2^a)(2^b) \;=\; 2^{(a+b)}$

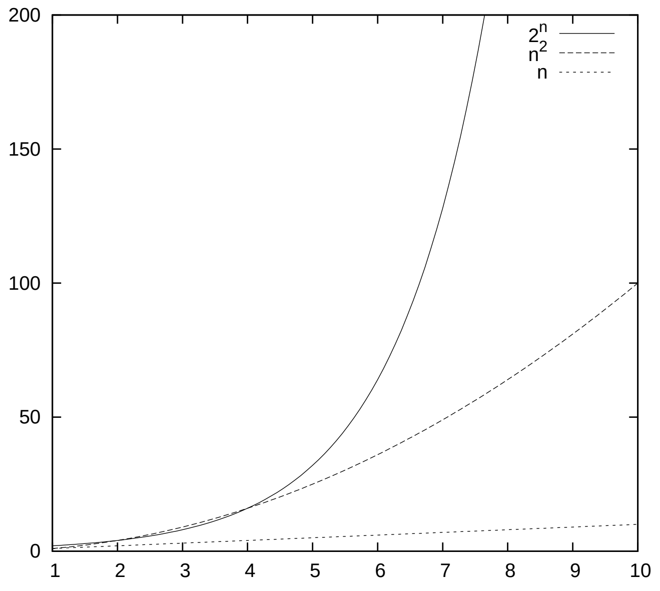

Figure 1: Some functions

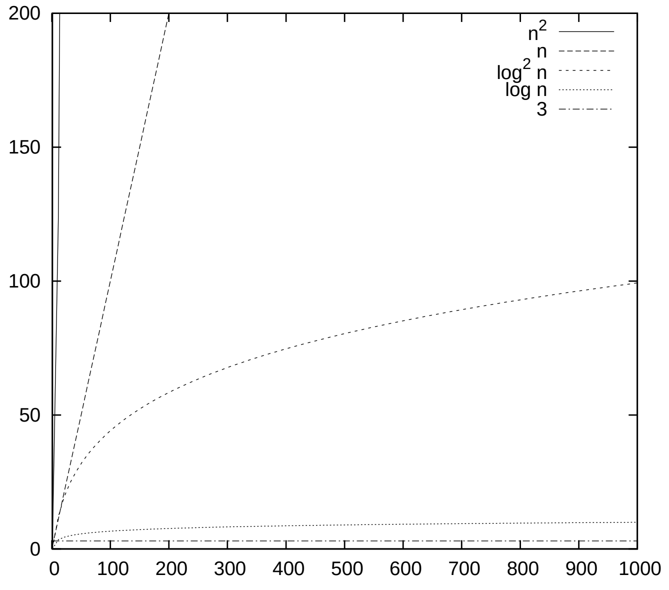

Figure 2: Some more functions

3 Summations

In DAPA we often find ourselves needing to sum up a sequence of terms, generated by the iterations (or recursions) of an algorithm. In this context, the notation

$$\Sigma_{i=a}^{b}(f(i))$$

(where a is usually 0 or 1 and b is a simple function of n) is just a shorthand for

$$f(a) + f(a+1) + f(a+2) + ... + f(b-1) + f(b)$$

For DAPA, the key examples, and their solutions are

- $\Sigma_{i=1}^{n}(i) \; = \; \frac{n(n+1)}{2}$
- $\Sigma_{i=0}^{n}(2^i) \; = \; 1 + 2 + 4 + ... + 2^n \; = \; 2^{n+1} - 1$
- $\Sigma_{i=0}^{\log n \, - \, 1}(2^i) \; = \; 1 + 2 + 4 + ... + \frac{n}{2} \; = \; n - 1$

4 Manipulating Big-O Notation

This section will be particularly lacking in mathematical detail! The full definition of the notation is in the course notes, but my intention here is to give you enough intuition to handle the situations which turn up within DAPA examples. This could all be justified formally from the real definitions (but I don't expect you to do that).

The key thing to understand is that when we say that some algorithm has, for example, a run-time of $\Theta(n^2)$, we are actually being quite imprecise. We are throwing away any knowledge of constant factors (i.e. the difference between $2n^2$ and $1000n^2$) and also any less significant additive terms. For example, the statements "$T(n) = \Theta(n^2)$" and "$T(n) = \Theta(6.5n^2 + 30n\log n - 23)$" are actually **identical**! Neither carries any more information. We might write the second to remind us how we derived the statement, but ultimately, both only tell us that the behaviour of T, as n grows large, is "similar" to that of the behaviour of n^2. This explains why, with care, we can appear to be quite rash in our treatment of quantities in the notation.

4.1 An Example

Suppose that we have analysed some algorithm and computed its run time in terms of three phases, which we believe take time roughly $6n^2$, $2n^2+25$ and $3n\log n$ respectively. We might then write

$$T(n) = \Theta\left(6n^2 + 2n^2 + 25 + 3n\log n\right) = \Theta\left(n^2 + n\log n\right) = \Theta\left(n^2\right)$$

Each apparent simplification, really changes nothing, it just throws away some more spurious detail. In a nutshell, the only thing which really matters in a $\Theta()$ expression is the term which involves the largest function of n (and even on that we can ignore any constant factor).

Things get trickier when we have quantities which are functions of both n and p. In such circumstances we have to be careful to only discard terms which are definitely insignificant, irrespective of the relationship between n and p, or alternatively, to qualify our statements with restrictions on that relationship.

4.2 Another Example

Suppose we have analysed some algorithm and decided that its run time satisfies

$$T(n,p) = \Theta\left(\frac{n}{p} + \log p\right)$$

For example, it may have a first phase in which each processor handles an equal share of data items independently, then a second phase in which some one-result-per-processor items from the first phase are combined. Can we simplify this to

$$T(n,p) = \Theta\left(\frac{n}{p}\right)$$

No! If $p = n$ then we have thrown away the most important term. Can we simplify it to

$$T(n,p) = \Theta\left(\log p\right)$$

Again, no. If p was a constant, then again we have thrown away the most important term. On the other hand, we could make this second simplification if we also required that $p = n$. In effect we would be saying that the run time is logarithmic **provided that** we have as many processors as input items.

4.3 Constant Functions

Sometimes we want to able to say that some operation or algorithm takes constant time (i.e. that the time is independent of the number of data items). For example, adding 1 to every element of an n element array within an n processor PRAM has this property. In asymptotic notation this is captured by saying $T(n) = \Theta(1)$. This doesn't mean that it takes exactly one step. We could just as well say $T(n) = \Theta(100000)$! Both of these statements simply say that the function is constant, independent of n. The first is usually preferred, for simplicity.