Analysing algorithms

We can analyse an algorithm without needing to run it, and in so doing gain some understanding of its likely runtime behaviour.

This analysis can be done at the DESIGN STAGE, before the program is built. Even if the analysis is approximate, potential problems may detected.

The NOTATION that results from the analysis is helpful in documenting software libraries. The notation allows programs that use such libraries to be analysed without requiring analysis of the library code (which is often not available).

Runtime equations

Consider the following function to sum the elements of a list:

```
fun sumList [] = 0
  | sumList (x::xs) = x + sumList xs
```

We are interested in an expression for the runtime of this function, $T_{\texttt{sum}List}$. The runtime depends on the argument list, but, assuming the `+` operation takes the same time regardless of the numbers being added, we can see only the length of the list, n, matters to $T_{\texttt{sum}List}$.

$T_{\texttt{sum}List}(n)$ is defined by the following expression:

$$T_{\texttt{sum}List}(n) = \begin{cases} c_0 & n = 0 \\ c_1 + T_{\texttt{sum}List}(n-1) & n > 0 \end{cases}$$

for some constants c_0 and c_1.

Solving recurrence relations (CLR, ch4)

The expression for $T_{\mathtt{s}umList}(n)$ is a RECURRENCE RELATION. We can use it for computing runtimes (given suitable values c_0 and c_1) but it is a difficult form to work with.

What we require is a CLOSED FORM equation, if possible.

It turns out that the following is an equivalent definition of $T_{\mathtt{s}umList}(n)$:

$$T_{\mathtt{s}umList}(n) = c_0 + nc_1$$

Much simpler! But (a) how do you get there, and (b) can you prove it?

(a) Deriving closed forms

There is no general way of solving recurrences. The approved, recommended method is: GUESS THE ANSWER, then prove it using induction.

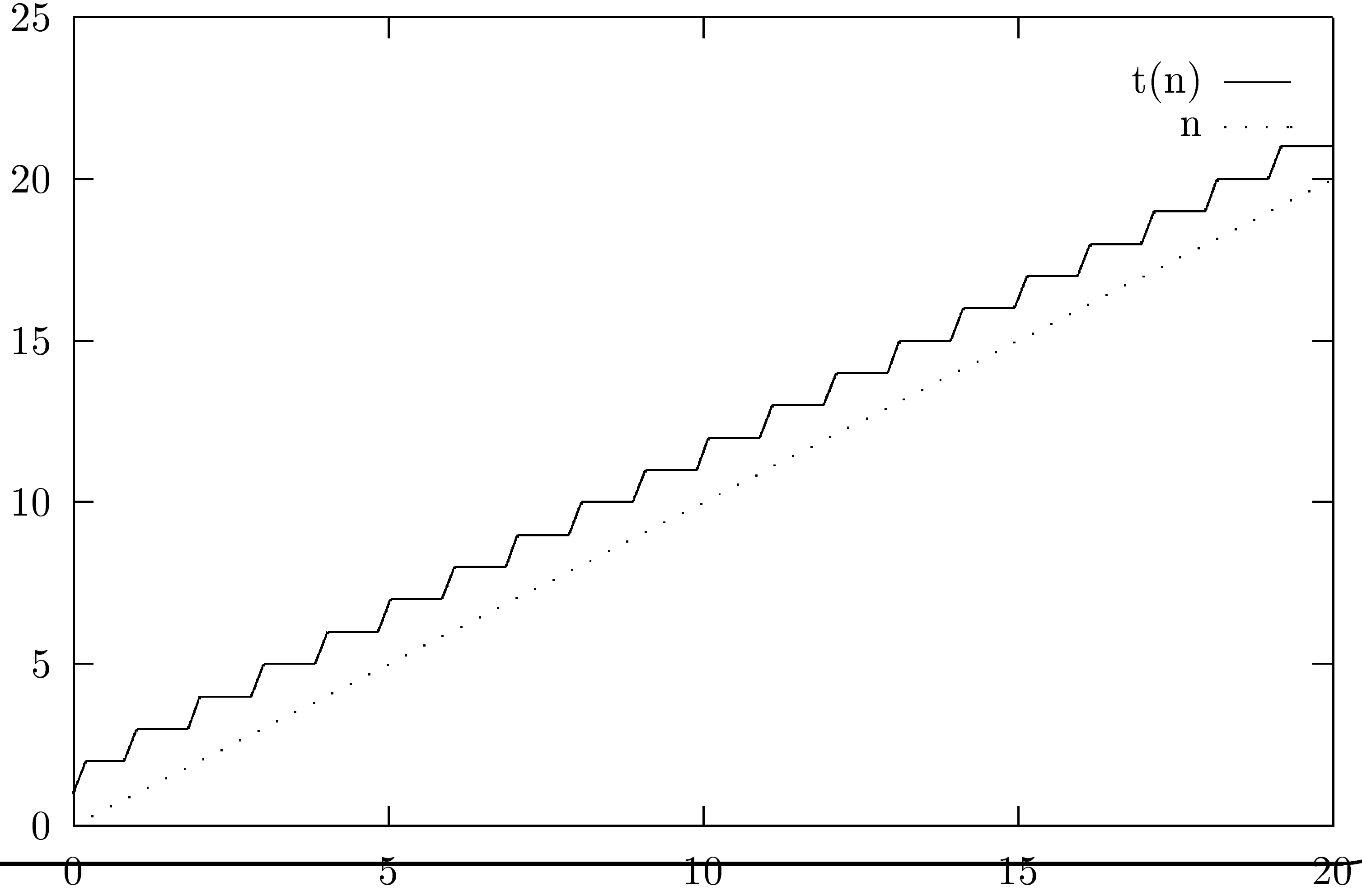
Suggestions for making a good guess:

- If the recurrence is similar to one you have seen before, guess a similar solution. Sometimes, substituting variables makes an unknown recurrence similar to one you know.
- Try expanding the recurrence for several terms, to see if you can detect a pattern. E.g.,

$$\begin{aligned} T(0) &= c_0 \\ T(1) &= c_0 + c_1 \\ T(2) &= c_0 + 2c_1 \\ T(3) &= c_0 + 3c_1 \end{aligned}$$

Plot the recurrence using gnuplot

```
gnuplot> t(n) = (n<=0) ? 1 : 1 + t(n-1)
gnuplot> plot [n=0:20] t(n), n
```



(b) Proof by induction

We seek to prove that

$$T(n) = \begin{cases} c_0 & n = 0 \\ c_1 + T(n-1) & n > 0 \end{cases}$$

is equal to $c_0 + nc_1$ for all $n \geq 0$.

BASIS: When $n = 0$, $T(n) = c_0 = c_0 + 0c_1$.

INDUCTIVE STEP: We assume that $T(n) = c_0 + nc_1$. Then

$$\begin{aligned} T(n+1) &= c_1 + T(n) \\ &= c_1 + c_0 + nc_1 \\ &= c_0 + (n+1)c_1 \end{aligned}$$

Back to algorithm analysis

The equation $T_{\texttt{sumList}}(n) = c_0 + nc_1$ is a useful, but *approximate* predictor of the actual runtime of `sumList`. Even if c_0 and c_1 were measured accurately, they actual runtime would vary with every change in the environment.

Therefore, no actual values of c_0 or c_1 are of any interest!

What's left?

The only interesting part of the equation is the bit that involves n. That's to say, the runtime of `sumList` is (within a constant factor) proportional to the length of the list. Calling `sumList` with a list twice as long will *approximately* double the runtime.

We say, $T_{\texttt{sumList}}(n) = \Theta(n)$ (pronounced "theta").

Θ-notation (CLR, ch2)

The Θ-notation is used to denote a set of functions that increase at the same rate (at least, within some constant bound).

Formally, $\Theta(g(n))$ is the set of all functions $f(n)$ that bounded by $c_1 g(n)$ and $c_2 g(n)$ (for some positive constants c_1 and c_2) when n gets sufficiently large (i.e., greater than some constant n_0).

We write $f(n) = \Theta(g(n))$ when we mean $f(n) \in \Theta(g(n))$. We also write $\Theta(1)$ to denote any constant function.

Example

$10 + 5n + n^2$ is $\Theta(n^2)$.

Proof: We need to choose positive constants c_1, c_2 and n_0 such that

$$c_1 n^2 \leq 10 + 5n + n^2 \leq c_2 n^2$$

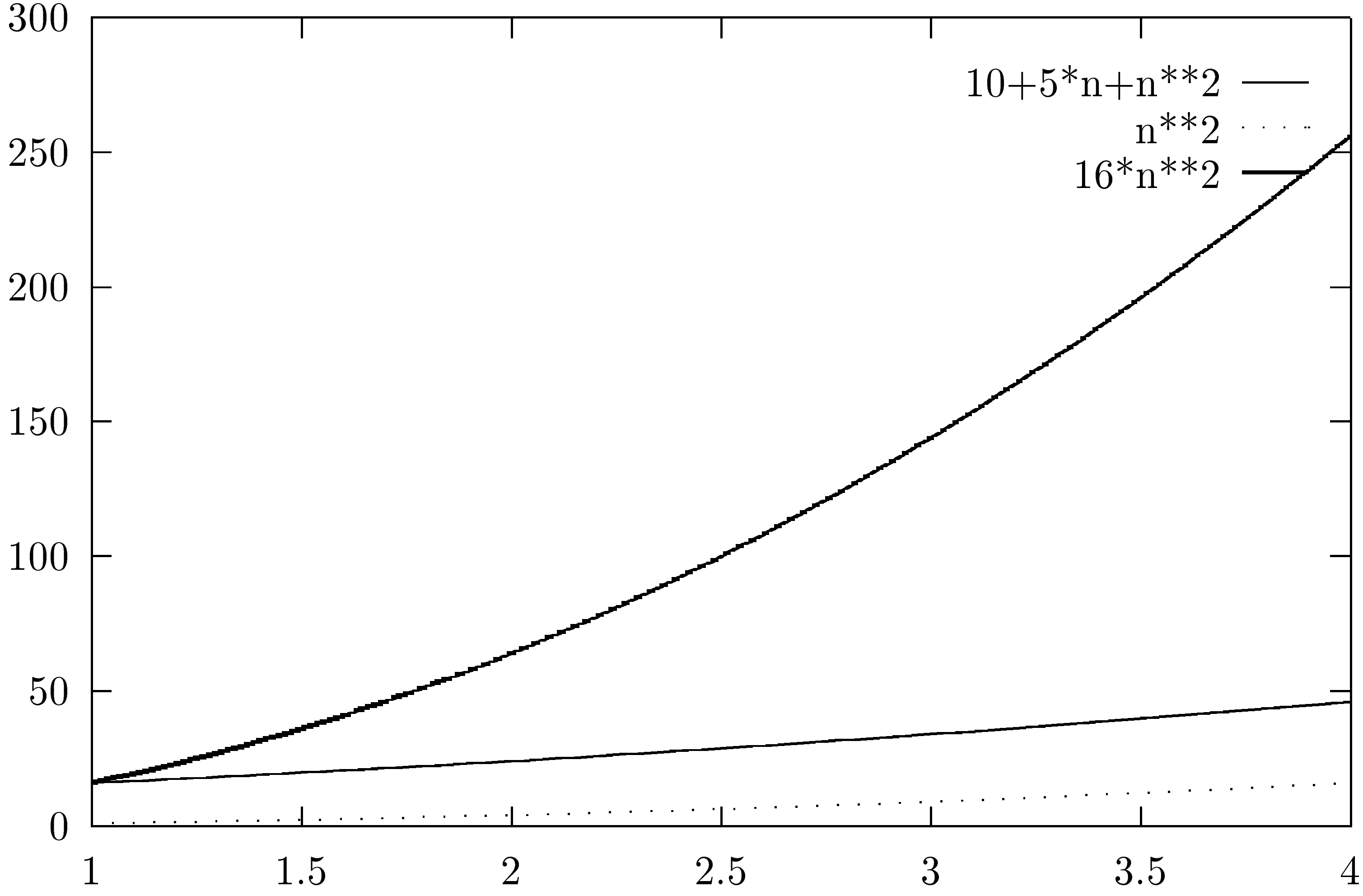
for all $n \geq n_0$

Dividing by n^2 gives

$$c_1 \leq \frac{10}{n^2} + \frac{5}{n} + 1 \leq c_2$$

The "sandwiched" term, $\frac{10}{n^2} + \frac{5}{n} + 1$, clearly gets smaller as n gets larger. It peaks at 16 when $n = n_0 = 1$, drops to 6 when $n = 2$, and becomes very close to 1 when $n = 1000$. It never get less than 1, so we can choose $c_1 = 1$. And it never exceeds $c_2 = 16$.

```
gnuplot> plot [n=1:4] 10+5*n+n**2, n**2, 16*n**2
```



Keep Θ-functions simple

While it is formally (and trivially) correct to say that $10 + 5n + n^2$ is $\Theta(10 + 5n + n^2)$, the whole purpose of the Θ-notation is to work with simple expressions. Thus, we never expect any arbitrary constants or lower-order terms inside the Θ.

Simplify Θ functions by

- setting all constants to 1
- dropping all lower-order terms

You will need to recognise lower order terms. If your mathemetics isn't up to the curlier examples, use `gnuplot` to observe the behaviour.

Variations on Θ (CLR, p25)

The Θ functions are "tight" bounds on runtime, bounding the function from above and below.

Variants include O ("big-oh"), which drops the lower-bound requirement, and Ω ("omega"), which drops the upper-bound.

Thus, any linear polynomial $a + bn$ is in $O(n^2)$ (and also in $O(n^3)$, etc.), but not in $\Theta(n^2)$, or $\Theta(n^3)$, etc.

Similarly, a quadratic function $a + bn + cn^2$ is $\Omega(n)$.

O and Ω arises whenever the actual runtime varies, depending on the input data. For example, insertion sort can run in either quadratic or linear time, depending on how close to sorted the input data happens to be.

O gives a worst-case running time of an algorithm; Ω the best case. Θ applies when both of these bounds are the same.