Misc topics in Recursion

O, Ω and Θ

Although most algorithms' time complexity will just be expressed in terms of time big -O, there are actually 3 related ways of measuring the time taken.

We have already seen that something like $O(n^2)$ just means that some constant times of n^2 effectively acts as an upper bound on the running time of the algorithm.

Big - omega (Ω) - A function $f(n)$ is said to be $\Omega(g(n))$ if and only if there exist n_0 and M such that $M|f(n)| \geq |g(n)|$ once $n \geq n_0$.

Big - theta (Θ) - The function $f(n)$ is $\Theta(g(n))$ if $f = O(g)$ and $f = \Omega(g)$.

Usually when programmers are expressing their time complexity they are talking about a big-Theta value. But they tend to say things like 'Order n squared' when it is $\Theta(n^2)$.

Fibonacci

The well known Fibonacci sequence can be defined by the following function

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{ow} \end{cases}$$

While the program becomes annoyingly slow if written in the most naive recursive manner, it is actually easier to analyze from a theoretical perspective when written as this recurrence.

A linear homogeneous recurrence relation of degree k has the following form

$f_n = c_1 f_{n-1} + c_2 f_{n-2} + \ldots + c_k f_{n-k}$

where the c_j's are constants that do not depend on n, and $c_k \neq 0$.

A standard way of trying to solve for these types of recurrences in closed form is to substitute x^n as f_n and see what happens.

For instance in the fibonacci case

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ x^n &= x^{n-1} + x^{n-2} \\ x^2 - x - 1 &= 0 \end{aligned}$$

The final equation that is obtained is called the characteristic equation. In this case it is a quadratic equation and it can be solved for roots which are

$$\frac{1+\sqrt{5}}{2} and \frac{1-\sqrt{5}}{2}$$

It is also easy to see (prove it by induction for a complete proof) that any linear combination of these two roots will satisfy the recurrence relation $f(n) = f(n-1) + f(n-2)$

So we know the closed form solution for fibonacci is

$$f(n) = c\left(\frac{1+\sqrt{5}}{2}\right)^n + s\left(\frac{1-\sqrt{5}}{2}\right)^n$$

How do we get the values of c and s? We use the base cases to say

$$c + s = 1$$
$$c\left(\frac{1+\sqrt{5}}{2}\right) + s\left(\frac{1-\sqrt{5}}{2}\right) = 1$$

Solving these two simultaneous equations we get the values of c and s and the closed form

$$f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$$

The amazing aspect of this is that fibonacci is a sequence of integers but the closed form contains terms that are filled with the irrational value $\sqrt{5}$.

Algorithm for solving recurrence relations in

1. convert the recurrence to a characteristic equation (can be done by claiming x^n to be the solution for the recurrence and working towards an equation)
2. find roots of the equation.
3. A theorem (proven in the zybook) shows how any linear combination of the roots will be a valid solution for the recurrence.
4. To get the values for the coefficients, use the base cases.

Recursion trees

When solving recurrences that involve dividing the initial input into halves, thirds etc, the best way to go about doing it is to to use the concept of the recursion tree.

See the handout on canvas for complete explanation. Look under the 'other resources' folder.

Master theorem

The recursion trees follow a general pattern and the more common ones can actually all be solved using a very powerful theorem called the master theorem that can be used quite easily for things like mergesort.

Theorem 1. *Let a and b be positive real numbers, with $a \geq 1$ and $b > 1$. Let $T(n)$ be defined for integers n that are powers of b by*

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases}$$

Then we have the following:

1. *If $f(n) = \Theta(n^c)$, where $\log_b a < c$, then $T(n) = \Theta(n^c) = \Theta(f(n))$.*
2. *If $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n^c log n) = \Theta(f(n) log n)$.*
3. *If $f(n) = \Theta(n^c)$, where $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$.*