Recurrence Relations

In analyzing the Towers of Hanoi, we might want to know how many moves it will take. Let T(n) stand for the number of moves it takes to solve the Towers problem for n disks. Then, we have the following formula:

$$T(n) = T(n-1) + 1 + T(n-1)$$

This is because in order to move a tower of n disks, we first move a tower of n-1 disks, which takes T(n-1) moves. Then we move the bottom disk (this is the +1 above), and then we move a tower of n-1 disks again, which takes us T(n-1) moves again.

Simplifying, we get:

$$T(n) = 2T(n-1) + 1$$

Unfortunately, this isn't terribly helpful to us, because it's not a formula in terms of n.

To get a formula in terms of n, we will use the iteration technique, which simply utilizes the fact that the formula above is true for all positive integers n. We will also use the fact that T(1) = 1, since it takes one move to move a tower of one disk.

Iterating to Solve the Recurrence

$T(n) = \underline{2T(n-1) + 1}$

$= 2[2T(n-2) + 1] + 1$, because $T(n-1) = 2T(n-2) + 1$.

$= 4T(n-2) + 2 + 1$

$= \underline{4T(n-2) + 3}$

$= 4[2T(n-3) + 1] + 3$, because $T(n-2) = 2T(n-3) + 1$

$= 8T(n-3) + 4 + 3$

$= \underline{8T(n-3) + 7}$

The three underlined steps indicate the three iterations in our work. A pattern should emerge from these three steps. The numbers in front of T(…) are successive powers of two. The number inside the T is $n – k$, where k is which power of k. Finally the number at the end is one less than the same power of two. Thus, we can conjecture that

$= 2^k T(n – k) + 2^k – 1$.

Finally, we want to plug in a value of k into this expression so that we can evaluate $T(n – k)$. This we know $T(1)$, we want $n – k = 1$. Equivalently, $k = n – 1$.

Plug in $k = n – 1$ into our formula:

$= 2^{n-1}T(1) + 2^{n-1} – 1 = 2^{n-1} + 2^{n-1} – 1 = 2^n – 1$.

Another recurrence that arises from the analysis of a recursive program is the following recurrence from binary search:

$T(n) = T(n/2) + 1$, since a binary search over n elements uses a comparison, and then a recursive call to an array of size n/2.

We use iteration:

$$T(n) = \underline{T(n/2) + 1}$$
$$= (T(n/4) + 1) + 1$$
$$= \underline{T(n/4) + 2}$$
$$= (T(n/8) + 1) + 1$$
$$= \underline{T(n/8) + 3}$$

We should see the pattern here and conjecture:

$$= T(n/2^k) + k.$$

We want a value of k that makes $n/2^k = 1$. This means that $n = 2^k$. By the definition of the logarithm, we have $k = \log_2 n$. Plugging in, we get:

$$= T(1) + \log_2 n$$

$= 1 + \log_2 n$, since a binary search of 1 element takes 1 step.

This is essentially (within 1) number of comparisons in the recursive binary search algorithm.

Let's analyze one last recurrence using this technique:

$T(n) = 2T(n/2) + n$, $T(1) = 1$.

$$
\begin{aligned}
T(n) &= \underline{2T(n/2) + n} \\
&= 2[\ 2T(n/4) + n/2\ ] + n, \text{ since } T(n/2) = 2T(n/4) + n/2 \\
&= 4T(n/4) + n + n \\
&= \underline{4T(n/4) + 2n} \\
&= 4[\ 2T(n/8) + n/4\ ] + 2n, \text{ since } T(n/4) = 2T(n/8) + n/4 \\
&= 8T(n/8) + n + 2n \\
&= \underline{8T(n/8) + 3n} \\
&= 2^kT(n/2^k) + kn.
\end{aligned}
$$

Once again we want to set $k = \log_2 n$.

$$
\begin{aligned}
&= nT(1) + n(\log_2 n) \\
&= n\log_2 n + n
\end{aligned}
$$

Analysis of Exponentiation

First, let's analyze the iterative algorithm shown in the function slowModPow.

This is straight-forward and does not require a recurrence relation. Basically, the loop runs exactly exp times. Counting each multiplication as a constant time operation (which for modular exponentiation is appropriate), the running time is simply O(exp).

If we analyze the recursive version, powerA, in terms of the value of exp, we get the following recurrence relation:

$T(exp) = T(exp - 1) + 1, T(1) = 1.$

because we make a recursive call with the exponent exp-1 and in addition to that, do a multiplication.

Using the iteration technique, we have

$$
\begin{aligned}
T(exp) &= T(exp - 1) + 1 \\
&= T(exp - 2) + 1 + 1 \\
&= T(exp - 2) + 2 \\
&= T(exp - 3) + 1 + 2 \\
&= T(exp - 3) + 3
\end{aligned}
$$

= …

= T(exp – k) + k

= T(1) + exp – 1, plugging in k = exp – 1.

= 1 + exp – 1 = exp.

Analysis of Fast Exponentiation

Now, let's take a look at fast exponentiation. When exp is even, we have

$T(exp) = T(exp/2) + 1$

when exp is odd, we have

$T(exp) = T(exp - 1) + 1$

Note that when exp is odd, exp – 1 is even, so really, we have:

$T(exp) = T(exp - 1) + 1 = T((exp - 1)/2) + 2$

Thus, roughly speaking, we can establish that

$T(exp) <= T(exp/2) + 2$.

Let's just solve $T(exp) = T(exp/2) + 2$, $T(1) = 1$ using the iteration technique.

Hopefully you can note that this is virtually identical to the binary search recurrence relation:

T(n) = T(n/2) + 1. (Just change the 1 to a 2 and you get the recurrence above.)

Thus, it follows that T(exp) = O(lg exp).

Thus, if exp = 10^{20}, we would do on the order of lg 10^{20} operations, which is simply around 66, as opposed to 100 billion billion operations. Now that's a HUGE difference.