Notes 13

This week we are going to talk about methods for solving reccurrences and reccurrent problems. As we will see, this stuff is very useful for analyzing the performance of recursive algorithms.

1 Hanoi Towers

We'll start with a simple and famous example of a recurrent problem known as the towers of Hanoi, invented by French mathematician Edouard Lucas in 1883.

In this problem, we are given a collection of disks of different sizes and three pegs. At the beginning, all disks are stocked in decreasing order of size on one peg. Our objective is to move all disks from this peg to another one moving just one disk at a time and never moving a larger disk onto a shorter one.

At first, it is not even clear that it is possible to do. We are going to give a general method which works for n disks whatever n is (the original problem was about 64 disks) and we will estimate the number of moves needed.

Definition 1.1 Let $T(n)$ be the number of steps needed to move the n-disk tower of the first peg to another peg.

It is easy to check that $T(1) = 1$, $T(2) = 3$ and $T(3) \leq 7$. Now let us consider the following recursive solution to our problem:

- first, move the upper n-1 disks to the third peg,
- then move the largest disk to the second peg,
- finally move the (n-1)-disk-tower from the third peg to the second peg.

As the first and the third step take $T(n-1)$ moves each and the second step takes one move, this shows that $T(n) \leq 2T(n-1) + 1$. Therefore, there exist a solution for all n. However, it could possibly be the case that our recursive algorithm is not optimal. We are going to show that in fact, it is optimal, there is no better way to move the disks.

To see why, notice that at some point we have to move the largest disk, say, from peg 1 to peg 2. The only moment when it is possible is when all the other disks are on peg 3 and then they necessarily are in decreasing order of size. But we need at least $T(n-1)$ moves to reach that position and then we need one move to move the largest disk and then $T(n-1)$ more moves to put the (n-1)-disk-tower onto it. Therefore, $T(n) \geq 2T(n-1)+1$, thus $T(n) = 2T(n-1)+1$.

It is not hard to guess a closed formula for $T(n)$. If we compute some values using the just obtained recursion, we will conjecture that $T(n) = 2^n - 1$. A very easy inductive proof shows that this is indeed correct. (For the base case, we check that $T_1 = 1 = 2^1 - 1$ and that the inductive step is $T_n = 2T_{n-1} + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$, where we used the inductive hypothesis to get from the second step to the third.)

In general this "guess and verify" method is best to solve recurrences. If it is not so easy to guess what the correct formula is, one can try the so-called *Plug and Chug* or *Brute Force* method we illustrate here:

$$T(n) = 1+2T(n-1) = 1+2+4T(n-2) = 1+2+4+T(n-3) = \cdots = 1+2+4+\cdots+2^{n-1}$$

However, it is very easy to make a mistake when observing the pattern, so it is important to verify our answer is correct.

2 Merge Sort

One of the most popular sorting algorithms is as follows:

To sort n items $x_1, x_2, \cdots, x_n$

- sort $x_1, x_2, \cdots, x_{n/2}$ and $x_{n/2+1}, x_{n/2+2}, \cdots, x_n$
- merge the two lists together.

To merge two lists, simply compare the smallest items of the two lists and put the smaller at the start of the merged list and keep the other one at the top of its short list. Then do it again with the remains of the two lists. Do it as long as there are elements which haven't been put on the new merged list yet.

For example to sort the list $\{10, 7, 23, 5, 2, 4, 3, 9, 5\}$, we do the following:

- Sort $\{10, 7, 23, 53\}$ to get $5, 7, 10, 23$ and sort $\{2, 4, 3, 9\}$ to get 2,3,4,9.
- Merge. Compare 2 and 5, the smallest items from each list. Put 2 into our output list. Now compare the smallest items (4 and 5) from the two remaining lists, and put the smaller on our output list and so on.

Definition 2.1 Let $T(n)$ be the most number of comparisons needed to sort n numbers.

To estimate $T(n)$, let us note that it takes $2T(n/2)$ steps to recursively sort the two lists of length $n/2$ and $n-1$ steps (at worst; merging might take fewer comparisons if, say, one of the lists empties completely before the other) to merge the two lists together. Therefore, $T(n) = 2T(n/2) + n - 1$.

For now, we will only consider n a power of 2 to keep things simple. It is easy to see that $T(1) = 0$, $T(2) = 1$, $T(4) = 5$, $T(8) = 17$ and so on. But it is not easy to guess the correct answer.

Applying the plug and chug method we get that

$$T(n) = (n-1) + 2T(n/2) = n - 1 + n - 2 + 4T(n/4) + \cdots + n - 2^i + 2^i T(n/2^i)$$

, where $T(n/2^i) = 0$ for $i = \log_2(n)$. Thus

$$T(n) = \sum_{i=0}^{\log(n)-1} (n - 2^i) = \sum_{i=0}^{\log(n)-1} n - sum_{i=0}^{\log(n)-1} 2^i = n\log(n) - (2^{\log(n)} - 1) = n\log(n) - n + 1$$

The reader should verify by induction that the estimate proven for $T(n)$ is indeed correct

Notice that the recurrence for the Tower of Hanoi problem, $T(n) = 2T(n-1)+1$ looks similar to this one, $T(n) = 2T(n/2) + n - 1$, but the two have very different solutions, $T(n)\ 2^n$ and $T(n)\ n\log n$ respectively. It is much better to break a problem into two problems of half size than into two problems of size one less even if it takes more work to do it.

Now consider a similar recurrence where we reduce the additive $n-1$ term. In particular define $S(1) = 0$ and $S(n) = 2S(n/2) + 1$ for $n \geq 2$ and n a power of 2. (Note that this is the number of recursive calls to srot in the merge sort algorithm just discussed.)

This recurrence can be solved with guess and verify or plug and chug. In either case, it is easy to prove inductively that $S(n) = n - 1$.

Sometimes things do not work out so nicely. We can make a good guess and induction fails us. This does not mean that our guesswas wrong, just that we have not been able to verify the guess. For example, take a closer look at the recurrence we just solved $S(n) = 1 + 2S(n/2)$, and try to prove by induction that $S(n) < n$.

- $S(1) = 0 < 1$
- $S(n) = 1 + 2S(n/2) = 2(n/2) + 1 \leq n + 1$

which is not enough, as we need $S(n) < n$.

This doesn't mean that the result is not true, it is straightforward to prove by induction that $S(n) = n - 1$, it just means that we couldn't establish it this way, as the choice of our claim was wrong, *it was not strong enough.* This shows that sometimes it is easier to prove a stronger statement by induction than a weaker one. It is important in cases like these to strengthen the hypothesis.

But if it is easier to prove things with stronger hypotheses, what keeps them from begin to strong?

Consider the flase claim that $S(n) \leq n-2$. We know this is not true. But we have that $S(n) \leq 2S(n/2)+1$, so that by the inductive hypothesis $S(n) \leq 2(n/2-2)+1 = n-3 \leq n-2$. This is almost a correct proof, but we still do not have the base case, which can be easitly seen to be false, since $S(1) = 0$ which is not less thatn -1.

We can now examine the list of recurrences that we have solved and their respective solutions.

$$T(n) = 2T(n-1) + 1 \qquad T(n)\ 2^n$$
$$T(n) = 2T(n/2) + n - 1 \qquad T(n)\ n\log n$$
$$S(n) = 2S(n/2) + 1 \qquad S(n)\ n$$

So decreasing $n-1$ to 1 in the additive term decreased solution by $\log n$ factor, not too huge.

Let us now consider what happens if we change the multiplicative factor of 2.

Let $T(n) = aT(n/2) + bn$ and $T(1) = 1$, where a and b are constant. We'll examine the effect of having a instead of 2 here. When you solve this recurrence by expansion, you get that

- for $a < 2$, you get $T(n) = \frac{2bn}{2-a} = \Theta(n)$,
- for $a = 2$, you get $T(n) = bnlog(n) = \Theta(nlogn)$, and
- for $a > 2$, you get $T(n) = \frac{ab}{a-2}n^{log(a)} = \Theta(n^{log(a)})$.

Thus we can conclude that the reccurence is very sensitive to any changes in a, but it varies only linearly if we change b. Hence, if we don't care about constant factors, we can write $T(n) = aT(n/2) + \Theta(n)$, with $\Theta(1) = 1$, but we cannot write $T(n) = \Theta(T(n/2))$ without losing a lot of sharpness in oour estimate.

3 General Divide and Conquer Recurrences

Consider the following general form of Divide and Conquer Recurrences.

$$T(x) = \begin{cases} \Theta(1) & \text{for } x \leq x_0 \\ \sum_{i=1}^{k} a_i T(b_i x) + g(x) & \text{for } x > x_0 \end{cases}$$

where $a_i > 0$, $b_i \in (0,1)$, k and x_0 are all constants and x_0 needs to be big enough to cover the base case (details in handout) and x is real (not just a power of 2).

All the recurrences that we have seen so far (except for the Tower of Hanoi) fits into this form.

A recent theorem (Akra-Bazzi '96) tells us the following:

Set p so that $\sum_{i=1}^{k} a_i b_i^p = 1$. If $|g'(x)| \leq x^{O(1)}$ (polynomial growth condition), then

$$T(x) = \Theta(x^p(1 + \int_1^x \frac{g(u)}{u^{p+1}} du)).$$

The proof uses the guess and verify technique (see handout).

Consider the example $T(x) = 2T(x/2) + \frac{8}{9}T(3x/4) + x^2$. Here $k = 2$, $a_1 = 2$, $b_1 = 1/2$, $a_2 = 8/9$, $b_2 = 3/4$. We want to find p such that $a_1 b_1^p + a_2 b_2^p = 1$. It turns out that $p = 2$ will work. Also, $g(x) = x^2$, so $g'(x) = 2x$, which is certainly polynomial. So the theorem tells us that

$$T(x) = \Theta(x^2(1 + \int_1^x \frac{u^2}{u^3} du)) = \Theta(x^2 \log x)$$

.

The expressions we had for the recurrence $T(x) = aT(x/2) + bx$ can be solved using the above.

Notice that the Akra-Bazzi result works for all reals, not just powers of 2. But we need to be a little careful here. For example, take a closer look a the merge sort recurrence. For general n (not just powers of 2), it looke like

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + n - 1$$

which does not fit the form of our theorem. The variation is small, but we have seen that small variations in the $b_i x$ part of the recurrence can lead to big changes in the solution. Fortunately, these kinds of variations do not affect the Akra-Bazzi result. In fact, we can sate a more general version of the theorem:

Theorem 3.1 The solution to

$$T(x) = \begin{cases} \Theta(1) & \text{for } x \leq x_0 \\ \sum_{i=1}^{k} a_i T(b_i x + h_i(x)) + g(x) & \text{for } x > x_0 \end{cases}$$

where $a_i > 0$, $b_i \in (0,1)$ and $|h_i(x)| = O(x/\log^2 x)$ for every i, is

$$T(x) = \Theta(x^p(1 + \int_1^x \frac{g(u)}{u^{p+1}} du)).$$

The proof (see handout) uses induction on a strengthened hypothesis.

For the recurrence we had for merge sort, $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + n - 1$, we have $a_1 = a_2 = 1, b_1 = b_2 = 1/2, h_1(n) = -n/2 + \lceil n/2 \rceil, h_2(n) = -n/2 + \lfloor n/2 \rfloor$. It is easy to see that p should be 1 and that $|h_i(n)|$ is $O(n/\log^2 n)$. So the solution is

$$T(n) = \Theta(n(1 + \log n)) = \Theta(n \log n),$$

which is the same as before (just powers of 2).

What about a recurrence like $T(n) = 2T(n/2 + \sqrt{n}) + n$? The solution is still $T(n) = \Theta(n \log n)$ since $h(n) = \sqrt{(}n) = O(n/\log^2 n)$. The bottom line is that small changes in the $b_i x$ term do not affect the solutiosn.

Another neat feature of the solution is that you can also change constant factors on the additive term without changing the solution by more than a constant factor. For example if we multiply the $g(x)$ term by a constant factor, the solution changes by a constant factor, which disappears in the Θ asymptotic notiation. We could try this with $T(n) = 2T(n/2) + 10n$ if we liked.

Hence, we may as well write the recurrence $T(n) = 2T(n/2) + \Theta(n)$, which happens fairly often. But we do not write $\Theta(T(n/2))$ or $T(\Theta(n))$, since the solution is very sensitive to the the values of a_i and b_i.

As mentioned, people often use asymptotic notation when they write recurrences. While doing this, it is easy to make false proofs by mixing big O and induction.

For example, in the recuurence $T(1) = 1, T(n) = 2T(n/2) + n$, we can show that $T(n) = O(n)$ by the following faulty inductive argument. The base case is $T(1) = 1 = O(1)$ and the inductive step looks like $T(n) = 2T(n/2) + n = 2O(n/2) + n = 2(n/2) + n = O(n)$. However, we know that the correct answer is $\Theta(n \log n)$.

The key to avoid these mistakes is to make sure that your induction hypothesis makes sense, that is, that it states something meaningful on a number n. For example, it is meaningless to say that "for all n, T(n) = O(n), since T(n) is just a number, not a function, whereas O(n) denotes a class of functions. The correct way to state the above assertion is: "for all n, we have $T(n) \leq cn$. Then our inductive step will fail, since we will have $T(n) = 2(T(n/2)) + n \leq 2cn/2 + n = cn + n = (c+1)n$ which is not less than or equal to cn As a general advice, keep big O out of inductive proofs!

4 Recurrences in a product form, Changing the variable

Consider the recurrence

$$T(n) = 2^n (T(\frac{n}{2}))^2$$

where n is a power of 2 and $T(1) = 1$. This particular recurrence arises in certain counting problems on trees. It does not look like anything we have seen but we can actually convert it into something familiar.

The trick is to convert products into sums using logarithms. This gives us

$$\log T(n) = 2 \log T(\frac{n}{2}) + n$$

. We can now set $S(n) = \log T(n)$ to get

$$S(n) = 2S(n/2) + n$$

and $S(1) = 0$. We know that $S(n) = n \log n$ (this is easy to check, we can also use the Akra Baksi result) so the solution is

$$T(n) = 2^{n \log n} = n^n$$

.

Another good method to make a nasty looking recurrence easier is by a change of variable. This is similar to the change of variable technique in calculus. For example, we cansider the recurrence

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$$

, with base condition $T(1) = 1$.

If we replace n by 2^m, we get

$$T(2^m) = 2T(\lfloor 2^{m/2} \rfloor) + m \leq 2T(2^{m/2}) + m.$$

Now set $S(m) = T(2^m)$ and we see that $S(m) \leq 2S(m/2) + m$, so that $S(m) = O(m \log m)$.

So the answer is

$$T(n) = O(\log n \log \log n)$$

.

The hard part is knowing what substitution to make, but this is a skill that comes out of practice, just like in calculus.

Sometimes it is necessary to employ a change of variable and a manipulation of the recurrence. Consider the recurrence $T(n) \leq nT(\sqrt{n})^2$. As we deal with a product, it is convenient to take logarithms to get

$$logT(n) \leq 2logT(\sqrt{n}) + logn.$$

Next, set $S(n) = logT(n)$ to get

$$S(n) \leq 2S(\sqrt{n}) + logn.$$

Expanding this we have $S(n) = O(log(n) \cdot log(log(n))$, which yields

$$T(n) = 2^{S(n)} = 2^{O(log(n) \cdot log(log(n))} = n^{O(log(log(n))}.$$