Lecture Notes

This is the first of two lectures about solving recurrences and recurrent problems. Needless to say, recurrent problems come up again and again. In particular, recurrences often arise in the analysis of recursive algorithms.

1 The Towers of Hanoi

In the Towers of Hanoi problem, there are three posts and seven disks of different sizes. Each disk has a hole through the center so that it fits on a post. At the start, all seven disks are on one post as shown in Figure 1. The disks are arranged by size so that the smallest is on top and the largest is on the bottom. The goal is to end up with all seven disks in the same order, but on a different post. This is not trivial because of two restrictions. First, the only permitted action is removing the top disk from a post and dropping it onto another post. Second, a larger disk can never lie above a smaller disk on any post. (These rules imply, for example, that it is no fair to pick up the whole stack of disks at once and then to drop them all on another post!)

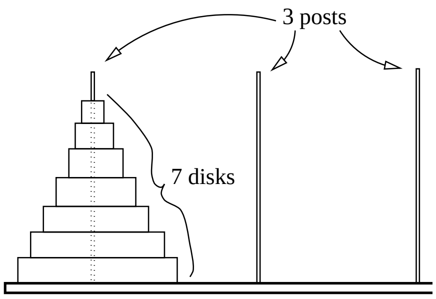
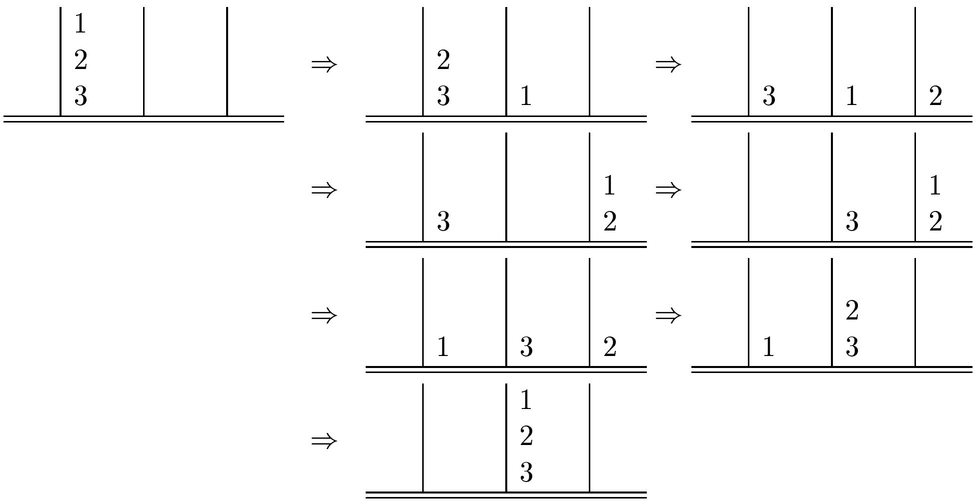


Figure 1: *This is the starting position in the Towers of Hanoi problem. All seven disks are stacked on the leftmost post. The goal is to move all the disks to a different post. At each step the top disk on a post can be removed and dropped onto another post. At no time, however, can a larger disk lie above a smaller disk.*

It is not immediately clear that a solution to this problem exists; maybe the rules are so stringent that the disks cannot all be moved to another post!

One approach to this problem is to consider a simpler variant with only three disks. We can quickly exhaust the possibilities of this simpler puzzle and find a 7-move solution like the one shown below. (The disks on each post are indicated by the numbers immediately to the right. Larger numbers correspond to larger disks.)



This problem was invented in 1883 by the French mathematician Edouard Lucas. In his original account, there were 64 disks of solid gold. At the beginning of time, all 64 were placed on a single post, and monks were assigned the task of moving them to another post according to the rules described above. According to legend, when the monks complete their task, the Tower will crumble and the world will end!

The questions we must answer are, "Given sufficient time, can the monks succeed?" and if so, "How long until the world ends?" and, most importantly, "Will this happen before the 6.042 final?"

1.1 Finding a Recurrence

The Towers of Hanoi problem can be solved in a recurrent fashion as follows. Let T_n be the minimum number of steps needed to move an n-disk tower. For example, it is not hard to see that $T_1 = 1$ and $T_2 = 3$. For 3 disks, the solution given above shows that $T_3 \leq 7$. We can generalize the approach used for 3 disks to the following recursive algorithm for n disks.

Step 1. Move the top $n-1$ disks from the first post to the third post. This can be done recursively in T_{n-1} steps.

Step 2. Move the largest disk from the first post to the to the second post. This requires just 1 step.

Step 3. Move the $n-1$ disks from the third post onto the second post. Again, T_{n-1} steps are required.

This algorithm shows that T_n, the number of steps required for n disks, is at most $T_n \leq 2T_{n-1} + 1$. We can now compute upper bounds on the number of steps required for various numbers of disks.

$$\begin{aligned} T_3 &\leq 2 \cdot T_2 + 1 \\ &= 7 \\ T_4 &\leq 2 \cdot T_3 + 1 \\ &\leq 15 \end{aligned}$$

The algorithm answers our first question; given sufficient time, the monks can finish their task and end the world.

1.2 A Lower Bound

We can not yet compute the exact number of steps that the monks need to move the 64 disks; we can only show an upper bound. Perhaps— having pondered the problem since the beginning of time— the monks have devised a better algorithm.

In fact, there is no better algorithm, and here is why. At some step, we must move the n-th disk from the first post to a different post. For this to happen, the $n-1$ smaller disks must all be stacked out of the way on the only remaining post. Arranging the $n-1$ smaller disks this way requires at least T_{n-1} moves. After the largest disk is moved, at least another T_{n-1} moves are required to pile the $n-1$ smaller disks on top.

This argument shows that the number of steps required is at least $2T_{n-1} + 1$. Since we gave an algorithm using exactly this number of steps, we now have a recurrence for T_n, the number of moves required to complete the Tower of Hanoi problem with n disks:

$$T_n = 2T_{n-1} + 1$$

Knowing the $T_1 = 1$, we can use this recurrence to conclude that $T_2 = 3, T_3 = 7, T_4 = 15, \ldots$.

1.3 Guess and Verify

Computing T_{64} from the recurrence would require a lot of work. It would be nice to have a closed form expression for T_n, so that we could quickly compute the number of steps required to solve the Towers of Hanoi problem for any given number of disks. (For example, we might want to know how much faster moving the disks would be if the monks donate one of the golden disks to their favorite political party as “soft money”.)

There are several different methods for solving recurrences. The simplest method is to guess the solution and then to verify that the guess is correct, usually with an induction proof. This method is called “Guess and Verify” or “substitution”. To help us make an intelligent guess, we can tabulate T_n for small values of n:

n	T_n
1	1
2	3
3	7
4	15
5	31

From the table, a natural guess is that $T_n = 2^n - 1$. Whenever you guess a solution to a recurrence, you should always verify it with a proof by induction or by some other technique; after all, your guess might be wrong. (But why bother to verify in this case? After all, if we're wrong, its not the end of the... no, let's check.)

Guess $T_n = 2^n - 1$

Verification. The proof is by induction on n. Let $P(n)$ be the predicate that $T_n = 2^n - 1$. In the base case, $P(1)$ is true because $T_1 = 1 = 2^1 - 1$. In the inductive step, assume $T_n = 2^n - 1$ to prove that $T_{n+1} = 2^{n+1} - 1$.

$$\begin{aligned} T_{n+1} &= 2T_n + 1 \\ &= 2(2^n - 1) + 1 \\ &= 2^{n+1} - 1 \end{aligned}$$

The first equality holds by assumption, and the second follows by application of the induction hypothesis. The final step is just a simplification. ■

Our guess is now verified. This shows, for example, that the 7-disk puzzle will require $2^7 - 1 = 127$ moves to complete. We can also now resolve our remaining questions about the 64-disk puzzle. Since $T_{64} = 2^{64} - 1$, the monks must complete more than 18 billion billion steps before the world ends. Better study for the final.

1.4 The Plug and Chug Method

In general, Guess and Verify is the best way to solve a recurrence. The only problem with the method is guessing the right solution. In the Towers of Hanoi example, guessing the solution was easy, but sometimes the solution has a very strange form that is quite hard to guess. Practice helps, of course, but so can some other methods.

Plug and Chug is one such alternative method for solving recurrences. Plug and Chug is also sometimes called "expansion", "iteration", or "brute force". The method consists of four calculation-intensive steps. These steps are listed for reference, but will only make real sense after we work through the Towers of Hanoi example below.

1. Expand the recurrence equation by alternately "plugging" (applying the recurrence equation) and "chugging" (simplifying the resulting expression).
2. Identify a pattern for the recurrence equation after i rounds of plug and chug. Verify that this pattern is correct by carrying out one additional round of plug and chug.
3. Substitute a value of i into the pattern so that T_n is expressed as a function of just the first few terms (e.g. $T_1, T_2, \ldots$). Substitute computed values for these terms to obtain an ordinary, non-recurrent expression for T_n.
4. Find a closed form for T_n from the ordinary expression.

The Plug and Chug calculations for the Towers of Hanoi recurrence are worked out below.

Step 1: Plug and Chug

$$\begin{array}{rcl} T_n & = & 1 + 2T_{n-1} \\ (plug) & = & 1 + 2(1 + 2T_{n-2}) \\ (chug) & = & 1 + 2 + 4T_{n-2}) \\ (plug) & = & 1 + 2 + 4(1 + 2T_{n-3}) \\ (chug) & = & 1 + 2 + 4 + 8T_{n-3} \\ (plug) & = & 1 + 2 + 4 + 8(1 + 2T_{n-4}) \\ (chug) & = & 1 + 2 + 4 + 8 + 16T_{n-4} \end{array}$$

The first equation is the recurrence that we want to solve. In the second step, we transform the original recurrence by *plugging*; that is, we apply the recurrence equation to make T_n depend on earlier terms. In this case, we eliminate T_{n-1} and make T_n depend on T_{n-2} instead. Next, we *chug*; that is, we simplify the new expression for T_n. Then we plug again so that T_n depends on a still earlier term, T_{n-3}. Then we chug and plug and chug some more.

Step 2: Identify and Verify a Pattern

$$\begin{array}{rcl} T_n & = & 1 + 2 + 4 + \ldots + 2^{i-1} + 2^i T(n-i) \\ (plug) & = & 1 + 2 + 4 + \ldots + 2^{i-1} + 2^i(1 + 2T_{n-(i+1)}) \\ (chug) & = & 1 + 2 + 4 + \ldots + 2^{i-1} + 2^i + 2^{i+1}T_{n-(i+1)}) \end{array}$$

After enough work, a pattern becomes clear: T_n is always a sum of consecutive powers of two together with an earlier T term.

The next step is to describe this pattern by writing down the expression obtained after i rounds of plug and chug. Then we do one last round to confirm that the pattern is correct. This check is very similar to the inductive step in a proof by induction.

Be careful in the "chug" stage; too much simplification can obscure an emerging pattern. For example, summing $1 + 2 + 4 + \ldots$ at every stage would have concealed the geometric series. The rule to remember— indeed, a rule applicable to the whole of college life— is *chug in moderation.*

Step 3: Express n-th Term Using Early Terms

$$\begin{aligned} T_n &= 1+2+4+\ldots+2^{i-1}+2^iT(n-i) \\ T_n &= 1+2+4+\ldots+2^{n-2}+2^{n-1}T(1) \\ &= 1+2+4+\ldots+2^{n-2}+2^{n-1} \end{aligned}$$

Now we can use the pattern to express T_n as a function of small, computable terms. In particular, if we set $i = n-1$ in the above pattern, then T_n becomes a function only of T_1, which we know is equal to 1. This leaves an ordinary, non-recurrent expression for T_n.

Step 4: Find a Closed Form for the n-th Term

All that remains is to reduce the ordinary expression for T_n to a closed form. We are fortunate in this case, because T_n is the sum of a geometric series. We learned how to tackle these last week!

$$\begin{aligned} T_n &= 1+2+4+8+\ldots+2^{n-2}+2^{n-1} \\ &= \sum_{i=0}^{n-1} 2^i \\ &= 2^n - 1 \end{aligned}$$

We are done! When using the Plug and Chug method, be sure to go back and verify that the answer is correct. It is very easy to make a mistake when observing the general pattern.

2 Merge Sort

There are many algorithms for sorting a list of n items; in fact, you will see a dozen of them in 6.046. One of the most popular sorting algorithms is Merge Sort.

2.1 The Algorithm

Here is how Merge Sort works. The input is a list of $n \geq 1$ items $x_1, x_2, \ldots, x_n$. If $n = 1$, then the algorithm returns the single item x_1. If $n > 1$, then the original list is broken into two lists, $x_1, \ldots, x_{n/2}$ and $x_{n/2+1}, \ldots, x_n$. Both of these lists are sorted recursively, and then they are merged to form a complete, sorted list of the original n items. All work is in the merging operation; the details are explained in the following example.

Suppose we want to sort the list 10, 7, 23, 5, 2, 4, 3, 9. Since the number of items is greater than 1, we divide into two lists; one is 10, 7, 23, 5, and the other is 2, 4, 3, 9. Each list is sorted recursively. The results are 5, 7, 10, 23 and 2, 3, 4, 9.

Now we must merge these two small sorted lists into one big sorted list. (In this case, the "small" lists have 4 items apiece, and the "big" list will eventaully have 8 items.) We start with the big list empty and add one item at a time. At each step, we compare the first items in the small lists. We remove the smaller item and append it to the end of the big list. This process repeats until one of the small lists becomes empty. At that point, the remaining small list is appended to the big list and we are done.

For the example, the contents of the three lists after each step are shown in the table below.

small list 1	*small list* 2	*big list*
$5, 7, 10, 23$	$2, 3, 4, 9$	
$5, 7, 10, 23$	$3, 4, 9$	2
$5, 7, 10, 23$	$4, 9$	$2, 3$
$5, 7, 10, 23$	9	$2, 3, 4$
$7, 10, 23$	9	$2, 3, 4, 5$
$10, 23$	9	$2, 3, 4, 5, 7$
$10, 23$		$2, 3, 4, 5, 7, 9$
		$2, 3, 4, 5, 7, 9, 10, 23$

2.2 Finding a Recurrence

In the analysis of a sorting algorithm, a traditional question is, "What is the maximum number comparisons used in sorting n items?" The number of comparisons is taken as an estimate of the run time. In the case of Merge Sort, we can find a recurrence for this quantity. Solving the recurrence will allow us to study the asymptotic behavior of the algorithm.

To make the analysis easier, assume for now that the number of items we are sorting is a power of 2. This ensures that we can divide the list of items exactly in half at every stage of the recursion.

Let $T(n)$ be the maximum number of comparisons used by Merge Sort in sorting a list of n items. If $n > 1$, then this is equal to the number of comparisons used in sorting both halves of the list plus the number of comparisons used in merging the two halves.

The number of comparisons used in sorting each half of the list is $T(n/2)$. Sorting both halves requires $2T(n/2)$ comparisons. The number of comparisons used in merging the two halves is at most $n - 1$. The reason is that one item is appended to the big list after each comparison, and at least one additional item is appended to the big list in the final step when one small list becomes empty. Since the big list eventually contains n items, there can be at most $n - 1$ comparisons. (Of course, there might be fewer comparisons, but we are analyzing the worst case.)

Adding all comparisons gives the following recurrence:

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

No comparisons are required to sort a list with a single item, so $T(1) = 0$. From this fact and the recurrence, we can compute $T(2) = 2T(1) + 2 - 1 = 1$, $T(4) = 2T(2) + 4 - 1 = 5$, $T(8) = 2T(3) + 8 - 1 = 17$, $T(16) = 2T(4) + 16 - 1 = 49$, etc.

2.3 Solving the Recurrence

We would like to find a closed form for the number of comparisons used by Merge Sort in sorting a list of n items. This requires solving the recurrence $T(n) = 2T(n/2) + n - 1$. The first method to try is Guess and Verify. The problem is that there is no obvious pattern to the values from small cases, 0, 1, 5, 17, 49. Our only other method is Plug and Chug. Here we go:

Step 1: Plug and Chug

$$\begin{aligned}
T(n) &= n - 1 + 2T(n/2) \\
(plug) &= n - 1 + 2(n/2 - 1 + 2T(n/4)) \\
(chug) &= n - 1 + n - 2 + 4T(n/4) \\
(plug) &= n - 1 + n - 2 + 4(n/4 - 1 + 2T(n/8)) \\
(chug) &= n - 1 + n - 2 + n - 4 + 8T(n/8) \\
(plug) &= n - 1 + n - 2 + n - 4 + 8(n/8 - 1 + 2T(n/16)) \\
(chug) &= n - 1 + n - 2 + n - 4 + n - 8 + 16T(n/16)
\end{aligned}$$

A pattern has emerged. Note that too much simplification would have obscured the pattern.

Step 2: Identify and Verify a Pattern

$$\begin{aligned}
T(n) &= n - 1 + n - 2 + \ldots + n - 2^{i-1} + 2^i T(n/2^i) \\
(plug) &= n - 1 + n - 2 + \ldots + n - 2^{i-1} + 2^i(n/2^i - 1 + 2T(n/2^{i+1})) \\
(chug) &= n - 1 + n - 2 + \ldots + n - 2^{i-1} + n - 2^i + 2^{i+1} T(n/2^{i+1})
\end{aligned}$$

The additional round of plug and chug confirms that the pattern is correct.

Step 3: Express n-th Term Using Early Terms

We must substitute a value for i into the pattern so that $T(n)$ depends on only small terms. A natural choice is $i = \log_2 n$, since then $T(n/2^i) = T(1)$. This substitution makes $T(n)$ dependent only on $T(1)$, which we know is 0.

$$\begin{aligned}
T(n) &= n - 1 + n - 2 + \ldots + n - 2^{i-1} + 2^i T(n/2^i) \\
&= n - 1 + n - 2 + \ldots + n - 2^{\log_2(n)-1} + 2^{\log_2 n} T(n/2^{\log_2 n}) \\
&= n - 1 + n - 2 + \ldots + n - 2^{\log_2(n)-1} + 2^{\log_2 n} T(1) \\
&= n - 1 + n - 2 + \ldots + n - 2^{\log_2(n)-1}
\end{aligned}$$

In the final equation, we used the fact that $T(1) = 0$; this completely eliminates the last term in the summation.

Step 4: Find a Closed-Form for the n-th Term

Now we have an ordinary, non-recurrent expression for $T(n)$. We can reduce this to a closed form by summing a series.

$$\begin{aligned}
T(n) &= n-1+n-2+\ldots+n-2^{i-1}+n-2^{\log_2(n)-1} \\
&= \sum_{i=0}^{\log_2(n)-1} n-2^i \\
&= \sum_{i=0}^{\log_2(n)-1} n - \sum_{i=0}^{\log_2(n)-1} 2^i \\
&= n\log_2 n - (2^{\log_2 n}-1) \\
&= n\log_2 n - n + 1 \\
&\sim n\log_2 n
\end{aligned}$$

What a weird answer— we never would have guessed this! To make sure that there is no error, we can check that the formula gives the same values for $T(n)$ that we computed earlier:

n	$n\log_2 n - n + 1$
1	$1\log_2 1 - 1 + 1 = 0$
2	$2\log_2 2 - 2 + 1 = 1$
4	$4\log_2 4 - 4 + 1 = 5$
8	$8\log_2 8 - 8 + 1 = 17$
16	$16\log_2 16 - 16 + 1 = 49$

Sure enough! Note that the $n\log_2 n$ comparisons used by Merge Sort are far fewer than the total of $n(n-1)/2$ possible comparisons.

3 More Recurrences

Let's compare the Tower of Hanoi and Merge Sort recurrences.

$$\begin{aligned}
T(n) &= 2T(n-1)+1 & \qquad T(n) &\sim 2^n \\
T(n) &= 2T(n/2)+n-1 & \qquad T(n) &\sim n\log_2 n
\end{aligned}$$

In the Towers of Hanoi, we divided a problem of size n into two subproblems of size $n-1$ and did only 1 additional step (moving the largest disk). In Merge Sort, we divided a problem of size n into two subproblems of size just $n/2$, but we did as many as $n-1$ additional steps (merging two sublists). Which algorithm runs faster? Merge Sort by a mile! The key point is that generating smaller subproblems is far more important than reducing the additional steps per recursive call.

Let's vary the above recurrences slightly and see how the solutions change.

3.1 A Speedy Algorithm

Suppose we have a speedy algorithm with the best properties of both earlier algorithms; that is, at each stage the problem is divided in half *and* we do only one additional step. The run time is described by the following recurrence:

$$\begin{aligned} S(1) &= 0 \\ S(n) &= 2S(n/2) + 1 \end{aligned}$$

(Another interpretation of this recurrence is that $S(n)$ is the number recursive calls made when Merge Sort is run on a list of n items.)

Guess and Verify is the first method to try. (Plug and Chug also works.) To motivate a guess, let's tabulate a few values of $S(n)$. As before, assume that n is a power of two.

n	$S(n)$
1	0
2	$2S(1) + 1 = 1$
4	$2S(2) + 1 = 3$
8	$2S(4) + 1 = 7$

Guess $S(n) = n - 1$

Verification. The proof is by strong induction on n. Let $P(n)$ be the predicate that if n is a power of two, then $S(n) = n - 1$. In the base case, $P(1)$ is true because $S(1) = 0$ by assumption. In the inductive step, for $n \geq 2$ assume $P(1)$, $P(2)$, …, and $P(n-1)$ to prove $P(n)$. If n is not a power of two, then $P(n)$ is vacuously true. Otherwise, we have:

$$\begin{aligned} S(n) &= 2S(n/2) + 1 \\ &= 2(n/2 - 1) + 1 \\ &= n - 1 \end{aligned}$$

The first equation is the recurrence. The second equality follows from the induction hypothesis, and the last step is simplification only. ■

The run time of this speedy algorithm is $\sim n$. This is better than the $\sim n \log n$ run time of Merge Sort, but only slightly so. This is consistent with the idea that decreasing the number of additional steps per recursive call is much less important than reducing the size of subproblems.

3.2 A Verification Problem

Sometimes verifying the solution to a recurrence using induction can be tricky. For example, suppose that we have the same recurrence as before, but we only try to prove that $S(n) \leq n$. This is true, but the proof goes awry!

Guess 1 $S(n) \leq n$

Verification (attempt). The proof is by strong induction. Let $P(n)$ be the predicate that if n is a power of two, then $S(n) \leq n$. In the base case, $P(1)$ is true because $S(1) = 0 \leq 1$. In the inductive step, for $n \geq 2$, assume $P(1)$, $P(2)$, $\ldots$, $P(n-1)$ to prove $P(n)$. If n is not a power of two, then $P(n)$ is vacuously true. Otherwise, we have:

$$\begin{aligned} S(n) &= 2S(n/2) + 1 \\ &\leq 2(n/2) + 1 \\ &= n + 1 \\ &\not\leq n \end{aligned}$$

The first equation is the recurrence. The second equality follows by induction. The third step is a simplification, and in the fourth step we crash and burn spectacularly.

We know that the result is true, but the proof did not work! The natural temptation is to ease up and try to prove something weaker, say $S(n) \leq n + 1$ or even $S(n) \leq 2n$. Bad plan!

As was often the case with other induction proofs, the trick is to choose a *stronger* induction hypothesis! In particular, if we use the stronger hypothesis that $S(n) \leq n - 1$, then the proof goes through. The crux of the argument in the inductive step is shown below.

$$\begin{aligned} S(n) &= 2S(n/2) + 1 \\ &\leq 2(n/2 - 1) + 1 \\ &= n - 1 \\ &\leq n - 1 \end{aligned}$$

As before, the first equation is the recurrence, the second follows by induction, and the third is a simplification. The final inequality shows that the induction hypothesis holds for $P(n)$ as desired.

3.3 A False Proof

What happens if we try an even stronger induction hypothesis? Shouldn't the proof work out even more easily? For example, suppose our hypothesis is that $S(n) \leq n - 2$. This hypothesis is false, since we proved that $S(n) = n - 1$. But let's see where the proof breaks. Here again is the crux of the argument:

$$\begin{aligned} S(n) &= 2S(n/2) + 1 \\ &\leq 2(n/2 - 2) + 1 \\ &= n - 3 \\ &\leq n - 2 \end{aligned}$$

Something is wrong; we proved a false statement! The problem is that we were lazy and did not write out the full proof; in particular, we ignored the base case. Since $S(1) = 0 \not\leq -1$, the induction hypothesis is actually false in the base case. This is why we cannot construct a proof with a "too strong" induction hypothesis.

3.4 Altering the Number of Subproblems

Some variants of the Merge Sort recurrence have truly peculiar solutions! The main difference in these variants is that we replace the constant 2 (arising because we create 2 subproblems) by a parameter a.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= aT(n/2) + n \end{aligned}$$

Intuitively, a is the number of subproblems of size $n/2$ generated at each stage of the algorithm; however, a is actually not required to be an integer. This recurrence can be solved by Plug and Chug, but we will give only the solutions here:

$$\begin{array}{rl} 0 \leq a < 2 & T(n) \sim \dfrac{2}{2-a} n \\ a = 2 & T(n) \sim n \log_2 n \\ a > 2 & T(n) \sim \dfrac{a}{a-2} n^{\log_2 a} \end{array}$$

The examples below show that the Merge Sort recurrence is extremely sensitive to the multiplicative term, especially when it is near 2.

$$\begin{array}{rl} a = 1.99 & T(n) = \Theta(n) \\ a = 2 & T(n) = \Theta(n \log n) \\ a = 2.01 & T(n) = \Theta(n^{1.007\ldots}) \end{array}$$

The constant $1.007\ldots$ is equal to $\log_2 2.01$.

4 The Akra-Bazzi Method

The Merge Sort recurrence and the variants we considered are called Divide and Conquer recurrences because they arise all the time in the analysis of Divide and Conquer algorithms.

Definition A *general Divide and Conquer recurrence* has the form

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

Here x is any non-negative real number; it need not be a power of two or even an integer. In addition, a_i and x_0 are positives constants, all b_i are constants between 1 and 0, and the number of terms in the summation is a constant k.

This general form describes all recurrences in this lecture, except for the Towers of Hanoi recurrence. (We will learn a standard approach to that type of problem in the next lecture.) Some hideous recurrences are also in the Divide and Conquer class:

$$T(x) = 2T(x/2) + 8/9T(3x/4) + x^2$$

In this example, $a_1 = 2$, $a_2 = 8/9$, $b_1 = 1/2$, $b_2 = 3/4$, $g(x) = x^2$, and x_0 can be any positive value.

4.1 Solving Divide and Conquer Recurrences

Last fall, two guys from Beirut named Akra and Bazzi discovered a very nice solution to all Divide and Conquer recurrences. This is a beautiful result that has not yet even been published!

Theorem 4.1 (Akra-Bazzi '96) Set p so that $\sum_{i=1}^{k} a_i b_i^p = 1$. If $|g'(x)| \leq x^{O(1)}$, then

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

The proof is given in a separate handout. Here is how the theorem can be used to solve the nightmarish recurrence given above:

$$T(n) = 2T(x/2) + 8/9T(3x/4) + x^2$$

First, we must find p:

$$\begin{aligned} a_1 b_1^p + a_2 b_2^p &= 1 \\ 2(1/2)^p + (8/9)(3/4)^p &= 1 \\ \Rightarrow p &= 2 \end{aligned}$$

Next, we have to check that $g'(x)$ has polynomial growth as required by the theorem:

$$|g'(x)| = |2x| < x^2 = x^{O(1)} \qquad \text{(for } x > 2\text{)}$$

Finally, we can compute the solution to the recurrence:

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

Similarly, the Akra-Bazzi method makes short work of the recurrence $T(n) = aT(n/2) + n$ discussed earlier.

4.2 The Irrelevance of Small Changes

We solved the Merge Sort recurrence only under the assumption that the number of items to be sorted was a power of two. The Akra-Bazzi solution has no such limitation. However, we must proceed carefully; if n is not a power of 2, then not only is the solution in doubt, but the recurrence itself does not describe the algorithm. If Merge Sort is called with an odd-length list, then we can not generate two subproblems of exactly equal size. To account for this, we must correct the Merge Sort recurrence as follows:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1$$

This recurrence does not fit the form of the theorem! Though the introduction of floor and ceiling operations alters the size of subproblems by at most 1, we have seen that small changes in a recurrence can radically alter the solution. Fortunately, this particular change does not affect the Akra-Bazzi result at all. This is made precise the following theorem, which is proven in the handout. The key feature of this theorem is the new term $h_i(x)$.

Theorem 4.2 The solution to:

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

is given by:

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

provided that for all i, $|h_i(x)| = O(\frac{x}{\log^2 x})$.

Now we can solve the revised Merge Sort recurrence. We can rewrite the recurrence as:

$$T(n) = T(n/2 + (\lceil n/2 \rceil - n/2)) + T(n/2 + (\lfloor n/2 \rfloor - n/2)) + n - 1$$

This is just the original Merge Sort recurrence with small correction factors. We can set:

$$\begin{aligned} h_1(n) &= \lceil n/2 \rceil - n/2 \\ h_2(n) &= \lfloor n/2 \rfloor - n/2 \end{aligned}$$

Since $|h_1(n)| < 1 = O(\frac{n}{\log^2 n})$ and the same holds for $h_2(n)$, we can apply the preceding theorem. In this case, $a_1 = a_2 = 1$ and $b_1 = b_2 = 1/2$, and so $p = 1$. We have to check that g does not change too rapidly: $|g'(n)| = 1 = n^{O(1)}$.

A short computation gives $T(n) = \Theta(n(1 + \log n)) = \Theta(n \log n)$. This is the same solution as before; the floor and ceiling made no difference.

In fact, even a recurrence such as $T(n) = 2T(n/2 + \sqrt{n}) + n$ has the same solution, since $\sqrt{n} = O(\frac{n}{\log^2 n})$. The rule of thumb is that *small* changes inside $T(\ )$ do not alter the solution to Divide and Conquer recurrences.

Small changes to the additive term $g(x)$ also do not matter. (Recall that in the Merge Sort recurrence, $g(x)$ corresponds to the number of additional steps per recursive call.) In particular, suppose that $g(x)$ were multiplied by a constant factor c. Looking at the Akra-Bazzi solution, we see that the constant c passes out of the integral and is absorbed into the Θ.

$$\begin{aligned} T(x) &= \Theta(x^p(1 + \int_1^x \frac{cg(u)}{u^{p+1}}\, du)) \\ &= \Theta(x^p(1 + c\int_1^x \frac{g(u)}{u^{p+1}}\, du)) \\ &= \Theta(x^p(1 + \int_1^x \frac{g(u)}{u^{p+1}}\, du)) \end{aligned}$$

Therefore, if we do not care about constant factors, then we can write a handy rule as follows:

$$\left.\begin{array}{l} T(n) = 2T(n/2) + \Theta(n) \\ T(1) = \Theta(1) \end{array}\right\} \Rightarrow T(n) = \Theta(n \log n)$$

4.3 A False Theorem

People often write recurrences with asymptotic notation as in the previous section, but be careful! As we have seen before, asymptotic notation mixed with induction leads to faulty proofs.

False Theorem 4.3 If $T(1) = 1$ and $T(n) = 2T(n/2) + n$, then $T(n) = O(n)$.

This theorem is false; we know that the correct solution to the recurrence is $T(n) = \Theta(n \log n)$.

Proof. The proof is by strong induction. Let $P(n)$ be the predicate that $T(n) = O(n)$. In the base case, $P(1)$ is true because $T(1) = 1 = O(1)$. In the inductive step, for $n \geq 2$ assume $P(1), P(2), \ldots P(n-1)$ to prove $P(n)$. We have:

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2O(n/2) + n \\
&= O(n)
\end{aligned}
$$

The first equation is the recurrence, the second follows by induction, and the third is a simplification. ■

The general problem is that Big Oh should not be used in an induction hypothesis. More specifically, the proof shows that for all $n \geq 1$, $T(n)$ is $O(n)$; that is, $T(1)$ is $O(1)$, $T(2)$ is $O(2)$, $T(47)$ is $O(47)$, etc. This is perfectly true, but wholly unrelated to the theorem statement; it says, in effect, that all constants are in fact constants.

Recalling the definition of Big Oh notation, the theorem implies that there exists a constant $c > 0$ and an n_0 such that for all $n \geq n_0$, $|T_n| \leq cn$. If we pick a c and x_0 and then attempt a proof by induction, then we fail as shown below.

Proof (failed attempt). The proof is by strong induction. Let $P(n)$ be the predicate that $|T_n| \leq cn$. In the base case, $P(n_0)$ is true because $T(n_0) \leq cn_0$. *(Suppose that we chose c large enough to make this true.)* In the inductive step, for $n \geq n_0 + 1$ assume $P(n_0), \ldots, P(n-1)$ to prove $P(n)$. We have:

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&\leq 2c(n/2) + n \\
&= cn + n \\
&= (c+1)n \\
&\not\leq cn
\end{aligned}
$$

The first equation is the recurrence. The next equality follows by applying the induction hypothesis. Thereafter we simplify until the proof collapses.