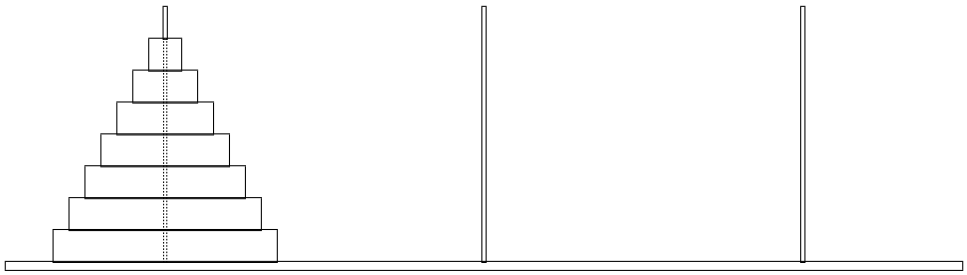
Chapter 12

Recurrences I

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.

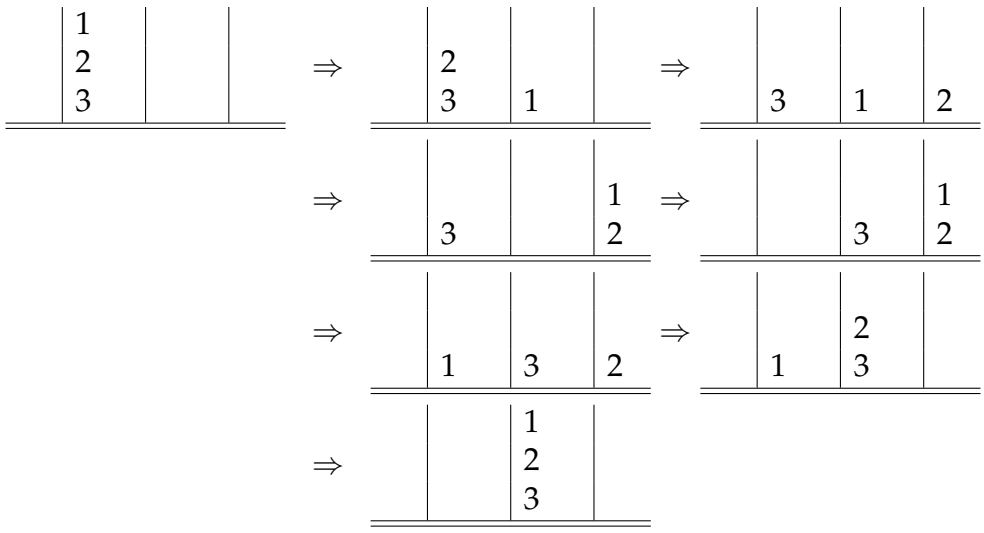
12.1 The Towers of Hanoi

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 post #1 as shown below. 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!)



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 such as the one shown below. (The disks on each post are indicated by the numbers immediately to the right. Larger numbers correspond to larger disks.)



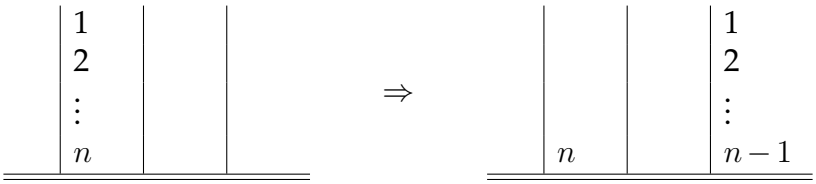
This problem was invented in 1883 by the French mathematician Edouard Lucas. In his original account, there were 64 disks of solid gold. At the beginning of time, all 64 were placed on a single post, and monks were assigned the task of moving them to another post according to the rules described above. 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?"

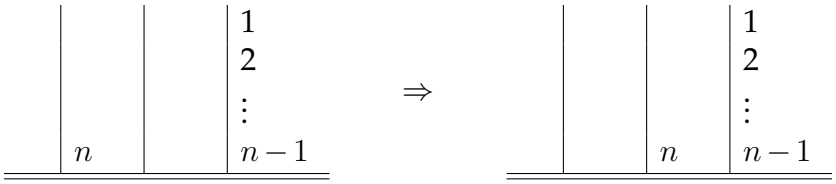
12.1.1 Finding a Recurrence

The Towers of Hanoi problem can be solved recursively as follows. Let T_n be the minimum number of steps needed to move an n-disk tower from one post to another. For example, a bit of experimentation shows that $T_1 = 1$ and $T_2 = 3$. For 3 disks, the solution given above proves that $T_3 \leq 7$. We can generalize the approach used for 3 disks to the following recursive algorithm for n disks.

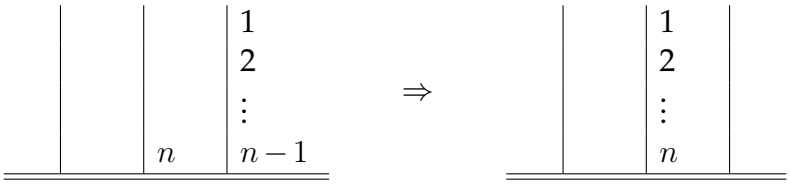
Step 1. Move the top $n-1$ disks from the first post to the third post. This can be done in T_{n-1} steps.



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 to move n disks to a different post, is at most $2T_{n-1}+1$. We can use this fact to 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 described above answers our first question: given sufficient time, the monks will finish their task and end the world. (Which is a shame. After all that effort they'd probably want to smack a few high-fives and go out for burgers and ice cream, but nope— world's over.)

12.1.2 A Lower Bound for Towers of Hanoi

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, the monks 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 that 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:

$$\begin{aligned} T_1 &= 1 \\ T_n &= 2T_{n-1} + 1 \qquad\qquad (\text{for } n \geq 2) \end{aligned}$$

We can use this recurrence to conclude that $T_2 = 3, T_3 = 7, T_4 = 15, \ldots$.

12.1.3 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 sooner the world would end if the monks melted down one disk to purchase burgers and ice cream *before* the end of the world.)

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". As a basis for a good guess, let's tabulate T_n for small values of n:

n	T_n
1	1
2	3
3	7
4	15
5	31
6	63

Based on this 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.)

Claim. *If:*

$$\begin{aligned} T_1 &= 1 \\ T_n &= 2T_{n-1} + 1 \qquad\qquad (\textit{for } n \geq 2) \end{aligned}$$

then:

$$T_n = 2^n - 1$$

Proof. The proof is by induction on n. Let $P(n)$ be the proposition that $T_n = 2^n - 1$.

Base case: $P(1)$ is true because $T_1 = 1 = 2^1 - 1$.

Inductive step: Now we assume $T_n = 2^n - 1$ to prove that $T_{n+1} = 2^{n+1} - 1$, where $n \geq 1$.

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

The first equality is the recurrence relation, and the second equation follows by the assumption $P(n)$. The last step is 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.

12.1.4 The Plug-and-Chug Method

In general, guess-and-verify is a great way to solve recurrences. The only problem with the method is guessing the right solution. This was easy in the Towers of Hanoi example, but sometimes the solution has a 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 are described below and illustrated with the Tower of Hanoi examle.

Step 1: Plug and Chug

Expand the recurrence equation by alternately "plugging" (applying the recurrence equation) and "chugging" (simplifying the resulting expression).

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

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 2: Identify and Verify a Pattern

Identify a pattern for the recurrence equation after i rounds of plugging and chugging. Verify that this pattern is correct by carrying out one additional round of plug and chug. In the Towers of Hanoi example, a strong pattern emerges: T_n is always a sum of consecutive powers of two together with an earlier T term:

$$T_n = 1 + 2 + 4 + \ldots + 2^{i-1} + 2^i T_{n-i}$$

We do one last round of plug-and-chug to confirm that the pattern is correct. This is amounts to the inductive step of a proof that we have the right general form.

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

Step 3: Express n-th Term Using Early Terms

Substitute a value of i into the pattern so that T_n is expressed as a function of just the base cases. Substitute values for these terms to obtain an ordinary, non-recurrent expression for T_n. For the Towers of Hanoi recurrence, substituting $i = n - 1$ into the general form determined in Step 2 gives:

$$\begin{aligned} T_n &= 1 + 2 + 4 + \ldots + 2^{n-2} + 2^{n-1}T_1 \\ &= 1 + 2 + 4 + \ldots + 2^{n-2} + 2^{n-1} \end{aligned}$$

The second step uses the base case $T_1 = 1$. Now we have 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're done! When using plug-and-chug method, you might want to verify your solution with induction. It is easy to make a mistake when observing the general pattern.

12.2 Merge Sort

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

12.2.1 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.

Let's work through an example. Suppose we want to sort this list:

$$10, 7, 23, 5, 2, 4, 3, 9$$

Since there is more than one item, 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:

$$5, 7, 10, 23$$
$$2, 3, 4, 9$$

Now we must merge these two small sorted lists into one big sorted list. We start with an empty big list and add one item at a time. At each step, we compare the first items in the small lists. We move the smaller of these two 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. The next items to move are underlined.

small list #1	small list #2	big list
$5, 7, 10, 23$	$\underline{2}, 3, 4, 9$	
$5, 7, 10, 23$	$\underline{3}, 4, 9$	2
$5, 7, 10, 23$	$\underline{4}, 9$	$2, 3$
$\underline{5}, 7, 10, 23$	9	$2, 3, 4$
$\underline{7}, 10, 23$	9	$2, 3, 4, 5$
$10, 23$	$\underline{9}$	$2, 3, 4, 5, 7$
$\underline{10, 23}$		$2, 3, 4, 5, 7, 9$
		$2, 3, 4, 5, 7, 9, 10, 23$

Because we keep dividing up the original list recursively until only 1 item remains, all the work is in the merging!

12.2.2 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 running time. In the case of Merge Sort, we can find a recurrence for this quantity. Solving this 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 original 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 there is only one item, then no comparisons are required, so $T(1) = 0$. If $n > 1$, then $T(n)$ is the sum of:

- The number of comparisions used in sorting both halves of the list, which is at most $2T(n/2)$.
- The number of comparisons used in merging two lists of length n. This is at most $n-1$ because 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. (There might be fewer comparisons if one small list empties out quickly, but we are analyzing the worst case.)

Therefore, the number of comparisons required to sort n items is at most:

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

12.2.3 Solving the Recurrence

Now we need a closed form for the number of comparisons used by Merge Sort in sorting a list of n items. This requires solving the recurrence:

$$\begin{aligned} &T(1) = 0 \\ &T(n) = 2T(n/2) + n - 1 \qquad\qquad (\text{for } n > 1) \end{aligned}$$

Let's first compute a few terms and try to apply guess-and-verify:

n	$T(n)$
1	0
2	1
4	5
8	17
16	49

There is no obvious pattern. We could compute more values and look harder, but let's try our other method, plug-and-chug.

Step 1: Plug and Chug

First, we alternately plug and chug until a pattern emerges:

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

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

Step 2: Identify and Verify a Pattern

Now we identify the general pattern and do one more round of plug-and-chug to verify that it is maintained:

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

Step 3: Express n-th Term Using Early Terms

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

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

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.

$$\begin{aligned}
T(n) &= (n-1) + (n-2) + \ldots + (n - n/2) \\
&= n \log n - (1 + 2 + 4 + \ldots + n/2) \\
&= n \log n - n + 1 \\
&\sim n \log n
\end{aligned}$$

What a weird answer— no one would ever guess that![1] As a check, we can verify 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	$16\log_2 16 - 16 + 1 = 49$

The values match! If we wanted certainty, we could verify this solution with an induction proof.

12.3 More Recurrences

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

$$\begin{array}{rll} \text{Hanoi} & T(n) = 2T(n-1) + 1 & \Rightarrow T(n) \sim 2^n \\ \text{Merge Sort} & T(n) = 2T(n/2) + (n-1) & \Rightarrow T(n) \sim n\log n \end{array}$$

Though the recurrence equations are quite similar, the solutions are radically different!

At first glance each recurrence has one strength and one weakness. In particular, in the Towers of Hanoi, we broke a problem of size n into two subproblem of size $n-1$ (which is large), but needed only 1 additional step (which is small). In Merge Sort, we divided the problem of size n into two subproblems of size $n/2$ (which is small), but needed $(n-1)$ additional steps (which is large). Yet, Merge Sort is faster by a mile! The take-away point is that generating smaller subproblems is far more important to algorithmic speed than reducing the additional steps per recursive call.

12.3.1 A Speedy Algorithm

Let's try one more recurrence. 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. Then the run time is described by the following recurrence:

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

[1] Except for the couple people in lecture who actually did. Oh well.

Let's first try guess-and-verify. As usual, we 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	$2S(4)+1=7$
16	$2S(8)+1=15$

The obvious guess is that $S(n) = n - 1$. Let's try to verify this.

Claim. *Suppose:*

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

If n is a power of 2, then:

$$S(n) = n - 1$$

Proof. The proof is by strong induction. Let $P(n)$ be the proposition that if n is a power of 2, then $S(n) = n - 1$.

Base case: $P(1)$ is true because $S(1) = 1 - 0 = 0$.

Inductive step: Now assume $P(1), \ldots, P(n-1)$ in order to prove that $P(n)$, where $n \geq 2$. If n is not a power of 2, then $P(n)$ is vacuously true. Otherwise, we can reason as follows:

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

The first equation is the recurrence. The second equality follows from assumption $P(n/2)$, and the last step is simplification only. □

Thus, the running time of this speedy algorithm is $S(n) \sim n$. This is better than the $T(n) \sim n \log n$ running 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.

12.3.2 A Verification Problem

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

Claim 75. *If n is a power of two, then $S(n) \leq n$.*

Proof. (failed attempt) The proof is by strong induction. Let $P(n)$ be the proposition that if n is a power of two, then $S(n) \leq n$.

Base case: $P(1)$ is true because $S(1) = 1 - 0 < 1$.

Inductive step: For $n \geq 2$, assume $P(1)$, $P(2)$, ..., $P(n-1)$ 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 the assumption $P(n/2)$. 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 $S(n) \leq 2n$. Bad plan! Here's what would happen in the inductive step:

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

We're still stuck! As with other induction proofs, the key is to use a *stronger* induction hypothesis such as $S(n) = n - 1$ (as above) or $S(n) \leq n - 1$.

12.3.3 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 valid proof with a "too strong" induction hypothesis.

12.3.4 Altering the Number of Subproblems

Some variations 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'll omit the details. The solution depends strongly on the value of a:

$$T(n) \sim \begin{cases} \dfrac{2n}{2-a} & \text{for } 0 \le a < 2, \\ n \log n & \text{for } a = 2, \\ \dfrac{an^{\log a}}{a-2} & \text{for } a > 2. \end{cases}$$

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

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

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

12.4 The Akra-Bazzi Method

The Merge Sort recurrence and all the variations we considered are called divide-and-conquer recurrences because they arise all the time in the analysis of divide-and-conquer algorithms. In general, a ***divide-and-conquer*** recurrence has the form:

$$T(x) = \begin{cases} \text{is defined} & \text{for } 0 \le x \le x_0 \\ \displaystyle\sum_{i=1}^{k} a_i T(b_i x) + g(x) & \text{for } x > x_0 \end{cases}$$

Here x is any nonnegative real number; it need not be a power of two or even an integer. In addition, $a_1, \ldots a_k$ are positive constants, $b_1, \ldots, b_k$ are constants between 0 and 1, and x_0 is "large enough" to ensure that $T(x)$ is well-defined. (This is a technical issue that we'll not go into in any greater depth.)

This general form describes all recurrences in this lecture, except for the Towers of Hanoi recurrence. (We'll learn a method for solving that type of problem in the next lecture.) Some hideous recurrences are also in the divide-and-conquer class. Here is an example:

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

In this case, $k = 2$, $a_1 = 2$, $a_2 = 8/9$, $b_1 = 1/2$, $b_2 = 3/4$, and $g(x) = x^2$.

12.4.1 Solving Divide and Conquer Recurrences

A few years ago, two guys in Beirut named Akra and Bazzi discovered an elegant way to solve *all* divide-and-conquer recurrences.

Theorem 76 (Akra-Bazzi, weak form). *Suppose that:*

$$T(x) = \begin{cases} \textit{is defined} & \textit{for } 0 \le x \le x_0 \\ \displaystyle\sum_{i=1}^{k} a_i T(b_i x) + g(x) & \textit{for } x > x_0 \end{cases}$$

where:

- $a_1, \ldots, a_k$ *are positive constants*
- $b_1, \ldots, b_k$ *are constants between 0 and 1*
- x_0 *is "large enough" in a technical sense we leave unspecified*
- $|g'(x)| = O(x^c)$ *for some* $c \in \mathbb{N}$

Then:

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

where p *satisfies the equation* $\sum_{i=1}^{k} a_i b_i^p = 1$.

We won't prove this here, but let's apply the theorem to solve the nasty recurrence from above:

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

The first step is to find p, which is defined by the equation:

$$2\left(\frac{1}{2}\right)^p + \frac{8}{9}\left(\frac{3}{4}\right)^p = 1$$

Equations of this form don't always have closed-form solutions. But, in this case, the solution is simple: $p = 2$. Next, we have to check that $g'(x)$ does not grow too fast:

$$|g'(x)| = |2x| = O(x)$$

Finally, we can compute the solution to the recurrence by integrating:

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

The Akra-Bazzi method an be frustrating, because you do a lot of inexplicable intermediate calculations and then the answer just pops out of an integral. However, it goes through divide-and-conquer recurrences like a Weed Wacker.

Let's try one more example. Suppose that the following recurrence holds for all sufficiently large x:

$$T(x) = T(x/3) + T(x/4) + x$$

Here $k = 2$, $a_1 = 1$, $a_2 = 1$, $b_1 = 1/3$, $b_2 = 1/4$, and $g(x) = x$. Note that $|g'(x)| = 1 = O(1)$, so the Akra-Bazzi theorem applies. The next job is to compute p, which is defined by the equation:

$$\left(\frac{1}{3}\right)^p + \left(\frac{1}{4}\right)^p = 1$$

We're in trouble: there is no closed-form expression for p. But at least we can say $p < 1$, and this turns out to be enough. Let's plow ahead and see why. The Akra-Bazzi theorem says:

$$\begin{aligned} T(x) &= \Theta\left(x^p\left(1+\int_1^x \frac{u}{u^{p+1}}\,du\right)\right) \\ &= \Theta\left(x^p\left(1+\int_1^x u^{-p}\,du\right)\right) \\ &= \Theta\left(x^p\left(1+\left(\frac{u^{1-p}}{1-p}\Big|_{u=1}^{x}\right)\right)\right) \\ &= \Theta\left(x^p\left(1+\frac{x^{1-p}-1}{1-p}\right)\right) \\ &= \Theta\left(x^p + \frac{x}{1-p} - \frac{x^p}{1-p}\right) \\ &= \Theta\left(x\right) \end{aligned}$$

In the last step, we use the fact that $x^p = o(x)$ since $p < 1$; in other words, the term involving x dominates the terms involving x^p, so we can ignore the latter. Overall, this calculation show that we don't need to know the exact value of p because it cancels out!

In recitation we'll go over a slightly more general version of the Akra-Bazzi theorem. This generalization says that *small* changes in the sizes of subproblems do not affect the solution. This means that some apparent complications are actually irrelevant, which is nice.