6.1200J/18.062J *Mathematics for Computer Science* Thursday 28th September, 2023
Massachusetts Institute of Technology, Spring 2024
Z. Abel, B. Chapman, E. Demaine
revised Wednesday 24th January, 2024

Lecture 07: Recurrences

A *recurrence* describes a sequence of numbers. Early terms are described explicitly, while later terms are expressed as a function of their predecessors. A very simple example is $a_0 = 1$, and $a_n = a_{n-1} + 1$. This has closed form $a_n = n + 1$. Another familiar example is the Fibonacci Numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. It takes some work, but we can even find a closed form for these:

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

Today we will see how to model computational problems, such as computing running time, by using recurrences, and we will see how to find closed-form solutions for many recurrences.

1 Towers of Hanoi

This is a puzzle originally proposed by the French mathematician Edouard Lucas in 1883, and it has an accompanying legend. According to legend, there is a tower in Hanoi which houses three tall posts, and there were 64 golden discs of different sizes stacked up on one of the posts, with the largest on the bottom, and decreasing in size up to the top. This tower was home to an order of monks who were tasked with moving these discs according to two rules:

1. Only one disc could be moved at a time.
2. A larger disc can never be on top of a smaller disc.

If/when the monks could move all of the discs to a different post, the tower would crumble and the world would end. We would like to compute how long the world will last.

1.1 Abstraction

In general, there are n disks, with sizes 1 (smallest) through n (largest). There are 3 posts, "left", "middle", and "right" (L, M, R). Must move a disk from the top of one stack to the top of another stack, but we can't put a larger disk on a smaller disk.

Let's come up with an algorithm to move all disks from L to R.

1.2 State Machine

Can model this as a state machine: each state is an n-element list $(x_1, \ldots, x_n) \in \{L, M, R\}^n$; in other words, x_i says whether the i'th disk is on the L, M, or R post. For example, the start state is $(L, L, \ldots, L)$, and the target state is $(R, R, \ldots, R)$.

Transitions are given by $(x_1, x_2, \ldots, x_i, \ldots, x_n) \to (x_1, x_2, \ldots, x_i', \ldots, x_n)$ if for every $j < i$, $x_j \notin\in x_i, x_i'$. In other words, all disks $1, \ldots, i-1$ must be out of the way on the third, untouched disk.

1.3 Playing with Boxes

Let's start small and work up to larger and larger solutions.

When $n = 1$, how to solve? $(1)()() \to ()()(1)$.

When $n = 2$? $(12)()() \to (2)(1)() \to ()(1)(2) \to ()()(12)$.

When $n = 3$? Plan ahead a bit. Need to "uncover" 3 before we can move it. Will look something like this: $(123)()() \stackrel{?}{\Rightarrow} (3)(12)() \to ()(12)(3) \stackrel{?}{\Rightarrow} ()()(123)$. So how can we move 12 from L to M? Wait, already solved this! The steps for $n = 2$ work just fine, because the stationary 3 disk doesn't affect the movements of 1 and 2. So we can use our $n = 2$ solution (swapping the roles of M and R) to move 12 from L to M, then move 3 from L to R, then use our $n = 2$ solution again.

Keep this firmly in your mind: We have a 7-move sequence to move 123 from one post to another. (Doesn't have to be L to R, can adjust for any two posts!) Don't have to remember exactly *how* that sequence works right now (can piece it back together later); for now, just trust that we *can* move 123 from any post to any other post with 7 moves.

When $n = 4$? Like before, it'll look something like this: $(1234)()() \Rightarrow (4)(123)() \to ()(123)(4) \Rightarrow ()()(1234)$. Can use our 7-step sequence for $n = 3$ twice. This gives us a $7 + 1 + 7 = 15$ step sequence to move 4 disks from L to R. It's often easier to temporarily try *not* to remember exactly how those 7-step sequences work; that way, we can understand our $n = 4$ solution at a high level with just 3 steps: move 123 from L to M, then 4 from L to R, then 123 from M to R.

What about $n = 5$? You guessed it! "Use the $n = 4$ recipe [whatever that is!] to get to $(5)(1234)()$, then move to $()(1234)(5)$, then use the $n = 4$ recipe again to get to $()()(12345)$."

We're just defining an algorith by induction! In general, to solve n disks, we can use the same idea, assuming we've already solved the $n - 1$ case:

$$\begin{aligned}
(1, 2, \cdots, n)()() &\stackrel{n-1}{\Rightarrow} (n)(1, 2, \ldots, n-1)() && n-1 \text{ solution from L to M} \\
&\to ()(1, 2, \ldots, n-1)(n) && \text{single move} \\
&\stackrel{n-1}{\Rightarrow} ()()(1, 2, \cdots, n) && n-1 \text{ solution from M to R}
\end{aligned}$$

In "code", it might look like this:

- $H(n, A, B)$: (move disks 1 through n from post A to post B)
  - If $n = 1$, move disk 1 from $A \to B$, done! Otherwise:
  - $H(n-1, A, C)$ (move the top $n-1$ disks onto the third post C)
  - move disk n from $A \to B$
  - $H(n-1, C, B)$ (move the rest of the disks onto B)

1.4 Guess and Check

Let $T(n)$ be the number of steps performed by our recursive algorithm above. Then $T(1) = 1$ and $T(n) = 2T(n-1) + 1$ for $n \geq 2$. In particular,

$$T(1) = 1 \qquad T(2) = 3 \qquad T(3) = 7 \qquad T(4) = 15 \qquad T(5) = 31 \qquad \cdots .$$

This is a great example of a *recurrence*: a sequence where terms are defined inductively, i.e., as a function of the previous terms.

A recurrence is an implicit representation, needing to iterate one at a time to discover more of the sequence. Just like with summations, we often want to know a closed form formula for the nth term without having to compute all the intermediate terms.

In this example, you might guess that $T(n) = 2^n - 1$, and you would be correct! How to prove? (Say it with me! Induction!) Just like with sums, if you know (or can guess) a closed form for the nth term of a recurrence, proving it by induction is straightforward.

Not hard to prove that our $T(n)$ is in fact the *optimal* algorithm for n disks (see book). This means $T(64) = 2^{64} - 1$, so at one move per second, the world will end in $(2^{64} - 1)$ sec, or around half a trillion years.

2 Selection Sort

Here's a simple sorting algorithm, called Selection Sort, arguably even simpler than Simple Sort. Read all n numbers to find the largest one ($n - 1$ comparisons), remove it from the list and put it at the end of the answer, then repeat with the remaining $n - 1$ numbers.

Easy recurrence for the number of comparisons: $T(1) = 0$ (already sorted), and $T(n) = n - 1 + T(n-1)$. Solution: $(n-1) + (n-2) + \cdots + 1 = (n-1)(n)/2$, our favorite! Are we doomed to quadratic runtimes for sorting algorithms!

3 Merge Sort

Idea: given two *sorted* lists, it's easy to find the smallest item, with just 1 comparison! Can merge entire list with at most $n - 1$ comparisons.

Do example: scrambled: 2,20,15,7,9,3,10,14 scrambled: ptown, ptonne, butternut, bubs, totoro, wire, ralph, coop

sorted: 2, 3, 7, 9, 10, 14, 15, 20 sorted: ptown, wire, bubs, totoro, ralph, coop, butternut, ptonne

Mergesort uses this merge routine together with recursion:

Mergesort (on $n = 2^k$ elements):

- Input: a list $X = (x_1, \ldots, x_n)$.[1]
- If $n = 1$, Done! Return X.
- Split list into two halves: $L = (x_1, \ldots, x_{n/2})$ and $R = (x_{n/2+1}, \ldots, x_n)$
- Use Mergesort recursively to sort L and R, resulting in lists L' and R'.
- Merge L' with R', and return the result.

A note about recursion: often seen as an evil scary impenetrable bogeyman topic, the bane of all CS students, woe unto us all, the end is nigh, and so on and so fifth.

What does it mean that we "sort L and R recursively" with an algorithm we haven't finished describing? How do we know the recursive calls worked correctly, if we haven't finished explaining why Mergesort works correctly?

Key takeaway: it's just *induction*. Mergesort works. Proof: MS(1) works. MS(2) works. MS(4) works because MS(2) works, and merge step is correct. MS(8) because MS(4) and merge step are both correct. In general, $MS(2^n)$ works because we're allowed to assume $MS(2^{n-1})$ works, because we're inducting.

How many comparisons do we do? Let's let $T(n)$ be (an upper bound on) the number of comparisons needed. Then we can write

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

How does this compare to selection sort?

n	1	2	4	8	16	32
Mergesort	0	1	5	17	49	129
Selection Sort	0	1	6	28	120	496

[1] Sometimes indices run 1-to-n, sometimes 0-to-$n-1$, it's anyone's guess!

3.1 Closed Form for $T(n)$: Plug and Chug

Guess and Check is only useful when we know the closed form already, or we have a reliable way to guess it. What if we don't? Here's a different method that can sometimes help us discover a pattern that we don't already know.

Plug and Chug: Substitute recurrence into itself, hope to find a pattern *in the expanded recurrence*. Let's try that!

$$\begin{aligned}
T(n) &= (n-1) + 2T(n/2) \\
&= (n-1) + 2(n/2 - 1 + 2T(n/4)) \\
&= (n-1) + (n-2) + 4T(n/4) \\
&= (n-1) + (n-2) + (n-4) + 8T(n/8) \\
&= \vdots \\
&= (n-1) + (n-2) + (n-4) + \cdots + (n-2^{k-1}) + 2^k T(n/2^k).
\end{aligned}$$

Recall we're assuming n is power of 2, so choose $k = \log_2 n$, so $2^k = n$. Then $T(n) = kn - (1 + 2 + 4 + \cdots + 2^{k-1}) + nT(1) = kn - (n-1) = n\log_2 n - n + 1$. Should verify with guess&check, but we'll skip that now.

4 Master Theorem

What about mergesort when n isn't a power of 2? Conservatively, can bound it with $T(n) = 2T(\lceil n/2 \rceil) + n - 1$. Is it still $T(n) \approx n\log_2 n - n + 1$? How can we tell? How precise is this approximation? Could probably still muddle through with an induction...

There are many *divide and conquer* algorithms like mergesort, with very similar recurrences. E.g.,

- Binary Search, from hw: $T(n) = T(\lceil n/2 \rceil) + 1$.
- Karatsuba integer multiplication: $T(n) = 3T(\lceil n/2 \rceil) + \Theta(n)$ (details omitted)

Let's learn how to solve lots of recurrences like this, in 3 easily-identified cases:

Theorem 1 (Master Theorem)**.** *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) = a \cdot T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n).$$

Then $T(n)$ has the following asymptotic bounds:

1. *If $f(n) = O(n^{(\log_b a)-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*

2. *If* $f(n) = \Theta(n^{\log_b a})$*, then* $T(n) = \Theta(n^{\log_b a} \log n)$*.*
3. *If* $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ *for some constant* $\epsilon > 0$*, and if* $af(\lfloor n/b \rfloor) \leq cf(n)$ *for some constant* $c < 1$ *and all sufficiently large* n*, then* $T(n) = \Theta(f(n))$*.*

This theorem is also true if we replace $\lfloor n/b \rfloor$ *with* $\lceil n/b \rceil$ *everywhere.*

Proof Sketch. Consider the recursive call tree: There's 1 node at n, a nodes at n/b, a^2 nodes at n/b^2, and in general, a^k nodes n/b^k. How big is this tree? It goes to depth $k = \log_b n$, and the total number of nodes is $\Theta(a^k) = \Theta(n^{\log_b a})$. How much work is done at every level? $f(n)$, $af(n/b)$, $a^2 f(n/b^2)$, ..., $a^k f(n/b^k)$.

If f grows really slowly as in case 1 (in the extreme case, imagine a constant $f(n) = 1$), then the *number* of nodes in the tree is all that matters, and the algorithm runs in $\Theta(\text{number of nodes}) = \Theta(n^{\log_b a})$. If f grows very quickly (case 3), then the top node dominates the recursion work, and the runtime is just $\Theta(\text{top node}) = \Theta(f(n))$. If these two forces are well balanced (in the sense that $f(n) \in \Theta(n^{\log_b a})$; this is case 2), then it turns out that each level contributes roughly the same amount of work, so the runtime looks like $\Theta(f(n) \cdot \text{depth}) = \Theta(n^{\log_b a} \cdot \log_b n) = \Theta(n^{\log_b a} \log n)$.

This can be made rigorous, and it can be shown that floors or ceilings don't affect the analysis. □

Example: Mergesort: $T(n) = 2T(\lceil n/2 \rceil) + n - 1$ has $a = b = 2$ and $f(n) = n - 1$. This is case 2, because $n^{\log_2 2} = n^1$. So $T(n) \in \Theta(n^1 \log n)$, as we hoped.

Example: Binary Search (saw in homework): $T(n) = T(\lceil n/2 \rceil) + 1$. This has $a = 1$, $b = 2$, $f(n) = 1$, $n^{\log_b a} = n^0 = 1$, so this is *also* case 2: $T(n) \in \Theta(n^0 \log n) = \Theta(\log n)$.

Example: Karatsuba, $T(n) = 3T(\lceil n/2 \rceil) + \Theta(n)$. This has $a = 3$, $b = 2$, $n^{\log_b a} = n^{\log_2 3}$. This is case 1 because $f(n) \in O(n^{\log_2 3 - \epsilon})$ because $\log_2 3 > 1$. So $T(n) \in \Theta(n^{\log_2 3})$.

Example: Hanoi: $T(n) = 2T(n-1) + 1$. Can't apply Master Theorem, because doesn't have correct form. Doesn't follow the "divide and conquer" paradigm. And to see a difference, consider the recursion tree: 1 node at n, 2 and $n-1$, 4 at $n-2$, etc. Since we subtract 1 instead of dividing by b, there are *linearly* many levels and *exponentially* many nodes in the tree, which is why the solution is exponential. By contrast, divide-and-conquer-style recurrences have log-depth and polynomial-size trees, so those solutions are typically much smaller.

Example: Master Theorem does have gaps! For example, $T(n) = 2T(n/2) + n \log n$ *cannot* be solved by master theorem. It is not case 2 because $f(n) = n \log n$ is strictly larger than $n^{\log_b a} = n$ asymptotically. But it is also not case 3, because it is only larger by a log factor, not by a polynomial factor n^ϵ. So we would need to resort to more careful recursion tree analysis, or some form of induction instead.

6.1200J Mathematics for Computer Science

Spring 2024