CS 4349 Lecture—August 28th, 2017

Main topics for #lecture include #recursion, #divide-and-conquer, and #recurrances featuring #example/tower_of_hanoi and #example/merge_sort.

Prelude

- Homework 1 is assigned today, you can find a link in the schedule section of the website. It is due in a week and a half, so Wednesday, September 6th.
- Next Monday is Labor Day, so there is no class. I will hold an extra office hour Friday, September 1st from 3:00pm to 4:00pm. You can always email me to schedule an appointment in case the normal times don't work.
- Finally, you need to fill out these prerequisite forms. The prerequisites are CS 3305 with a grade of C or better, and (CE 3345 or CS 3345 or SE 3345 or TE 3345). Please fill out the forms and hand them back before class ends. We can talk if you skipped something, but I can't guarantee I'll let you skip the prerequisite.

Recursion

- A primary goal in computing is to avoid manually repeating work. We write scripts to carry out mundane tasks where each individual task is not too hard, but we don't want to be bothered doing them all by hand.
- We also use other people's libraries so we don't have to solve the same problems again and again. Besides, we might trust those libraries more than we trust ourselves.
- The same idea holds in algorithm design. A *reduction* from a problem X to another problem Y is an algorithm for X that uses an algorithm for Y as a "black-box" or "subroutine".
- A correct reduction cannot assume anything about how the algorithm for Y works except that it solves Y correctly. In fact, you don't want to know how that algorithm works. The whole point is that somebody else solved Y for you. Maybe it was your neighbor down the hall. Maybe it was you five minutes ago. Whoever it was, just trust them, OK?
- *Recursion* is a special type of reduction where you reduce a problem to a simpler instance of itself.
    - If the problem instance is small enough, solve the problem using whatever algorithm seems easiest, like brute force searching for a solution or maybe doing nothing at all.
    - For all other instances, you reduce to a **simpler** instance of the same problem.
- The key here is that you are actually writing a reduction. **you do not need or want to worry about how the simpler instance is solved**.
- Imagine you're given a problem of size n, and you're just delegating to others to solve

problems of size less than n for you. Maybe they're your peers. Maybe they're elves; it doesn't matter. But whoever they are, they don't like being micromanaged, **so just trust them**.

- Mathematically, it's the same as applying the inductive hypothesis in an inductive proof. The simpler statement is just true, OK?

Tower of Hanoi

- So I borrowed this toy from my 18 month old son. I thought we could play with it too.
- It has three wooden pegs and a few disks. I think we're supposed to put the disks on a peg in order like this.
- So here's a puzzle for you. I want to move the disks to a different peg, but I can only move one disk at a time, and I don't want a larger disk to sit on a smaller one. This puzzle is called the tower of Hanoi puzzle. How do we solve it?
- It's tempting to say we move this top disk somewhere, then we move the second disk to the other peg, but now we have two choices essentially so what do we do now?
- But let's reset the puzzle and see if there's a bit more structure there.
- At some point, we have to move that bottom disk, but it's the largest, so we can't move it until all other disks have been moved off of it.
- But moving those other disks off the bottom disk is just the same problem again!
- And when we want to finish the puzzle, we need to put them back.
- So let's go ahead and move the smaller disks. Since it's a simpler instance of the same problem, we can just solve it recursively. **camera tricks**. That was easy. Move the big disk. And move the others back **more camera tricks**.
- Wait, shouldn't I show you how to move the smaller disks around? No, because we're already done. Part of the point of recursion is to remove the cognitive load of dealing with the simpler problems. We reduced this instance of [six] disks to two instances of the puzzle with [five] disks and just trust that the smaller instance has a solution.
- We don't need to understand what happens with the other five disks. We shouldn't even care. And you certainly don't want to watch me solve those recursive subproblems. Had I brought a puzzle with a dozen or more disks, I'd still be moving these things around by the time lecture ended. A few more and I wouldn't get to go home tonight.
- Oh, but we do need to figure out the base case to have a full solution! Let's choose a simple one, say no disks [remove disks]. So I want to move an empty stack to another empty stack. OK, that was easy.
- So what if we had an arbitrary number of disks, say n? Well, if n = 0, then we'd do nothing. Otherwise, we'd move the top n -1 disks, move the bottom disk, and then move the top n -1 disks back on top of the big disk. Or in pseudocode:
    - Hanoi(n, src, dst, tmp):

        - If $n > 0$:
            - Hanoi(n-1, src, tmp, dst)
            - move biggest disk from src to dst
            - Hanoi(n-1, tmp, dst, src)
- So we designed this algorithm. Correctness isn’t too hard to see. For $n = 0$, the algorithm correctly does nothing. For larger n, we inductively assume it works for smaller sets. i.e., the recursive calls are correct. We move the largest disk from a peg with nothing smaller to another peg with nothing smaller. The rest of the moves are correct by the inductive hypothesis.
- There’s a special case of this problem you may have heard of curtesy of Henri de Parville.
    - “In the great temple at Benares beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.”
- I would like to know how much time we have left, so how do we analyze this algorithm? Let $T(n)$ denote the exact number of moves we make with n disks. We move two sets of $n-1$ disks, and we do one additional move.
- So $T(n) = 2(n-1) + 1$. This is an example of a *recurrence* which is a function defined on smaller instances of itself.
- We would like to *solve* the recurrence into a simple closed form expression for $T(n)$.
- There are a few ways we could do this. The most failproof is to simply guess the solution and prove it is correct.
- So let’s try to guess by looking at a few examples.
    - $T(0) = 0$. $T(1) = 1$. $T(2) = 3$. $T(3) = 7$. $T(4) = 15$. Does anybody have a guess for what $T(n)$ equals?
- $T(n) = 2^n - 1$. Proof:
    - $2^0 - 1 = 0$.
    - IH: $T(k) = 2^k - 1$ for $0 \leq k < n$.
    - IS. $T(n) = 2T(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$.
- So it takes a while to solve this puzzle. Had I brought a puzzle with 15 disks, even moving one disk a second it would take me over nine hours to solve this thing.

- And it will take the Brahmins approximately 585 billion years to end the world. So I guess we're safe.
- We'll go over other ways to solve recurrences later. But if you can guess the answer, then a proof by induction should work to verify it.

Merge Sort

- So we kept looking at this insertion sort example during the first week of class. It was a sorting algorithm that ran in O(n^2) time.
- But can we do better?
- So let's say we're given A[1.. n].
- A subsequence or subarray is just a smaller sequence or array. Maybe we can find smaller arrays or break A apart so we can delegate using recursion, and once the subarrays are sorted, we can somehow use them to quickly sort A.
- This general idea is called the *divide-and-conquer* paradigm which involved three steps:
  - *Divide* the problem into simpler subproblems.
  - *Conquer or delegate*: solve the subproblems recursively.
  - *Combine* the solutions to the subproblems into a solution to our original problem.
- If the original problem is very small, say n ≤ c for some constant c, then just use whatever simple algorithm you'd like. Any reasonable function of c is a constant, so the base case runs in constant time.
- Merge Sort is one example of a divide-and-conquer algorithm. You
  - Divide the array A into two subarrays of nearly equal size.
  - Conquer by running merge sort on the two subarrays
  - Combine the sorted subarrays using a merging procedure that sorts A.
- The base case is n = 1. You don't have to do anything.
- So let's start with the divide and conquer steps:
- Merge-Sort(A[1.. n]):
  - if n > 1
    - m = \floor{n/2}
    - Merge-Sort(A[1.. m])
    - Merge-Sort(A[m+1..n])
    - Merge(A[1..n], m) merges sorted arrays A[1..m] and A[m+1 .. n] to sort A[1.. n].
- Again, we don't worry about what happens in those two Merge-Sort calls! They happen on smaller arrays. That's all you need to know.
- So what would this look like?
  - 24 1 63 97 88 7 . 84 64 75 49 82 65
  - 1 7 24 63 88 97 . 49 64 65 75 82 84
- Merging takes advantage of the subarrays being sorted. The smallest element of A must

be at the beginning of one of the two subarrays, so we can find it with a single comparison, pluck it off its subarray, and continue from there.

- Merge(A[1.. n], m):
  - assumes A[1.. m] and A[m+1 .. n] are sorted

  - build a sorted array B
  - i = 1; j = m + 1
  - for k = 1 to n
    - if j > n
      - B[k] = A[i]; i = i + 1
    - else if i > m
      - B[k] = A[j]; j = j + 1
    - else if A[i] < A[j]
      - B[k] = A[i]; i = i + 1
    - else
      - B[k] = A[j]; j = j + 1

  - copy result to A
  - for k = 1 to n
    - A[k] = B[k]
- So what would this look like?
  - 1 7 24 63 88 97 . 49 64 65 75 82 84 ➡ 1 7 24 49 63 64 65 75 82 84 88 97
- How do we prove correctness? Induction! But for both procedures!
- Merge, by induction on n - k + 1, the number of elements added to B[k.. n] from the arrays A[i .. m] and A[j .. n]. When n - k + 1 = 0, there's nothing to do and nothing done. Assume the algorithm correctly merges A[i' .. m] and A[j' .. n] into B[k' .. n] in iteration k' > k and beyond. (n - k' + 1 < n - k + 1). Some cases:
  - If i ≤ m and j > n, A[j.. n] is empty. A[i] is the smallest element of sorted array A[i.. m] so it goes first in B[k .. n]. A[i+1 .. m] and A[j.. n] are merged correctly into B[k+1 .. n] by the IH.
  - If i > m and j ≤ n, the assignment B[k] = A[j] is correct and subsequent assignments are correct by the same reasoning.
  - If i ≤ m, j ≤_ n, and A[i] < A[j}, then the smallest element in both arrays is A[i], so B[k] should equal A[i]. By IH, the rest of the merging is correct.
  - If i ≤ m, j ≤_ n, and A[i] ≥_ A[j], then A[j] is the smallest element. The assignment is right and the rest are right by the IH.
- Merge-Sort is a bit easier. Do induction on n.
  - If n ≤ 1, the array is sorted and we do nothing.
  - We use the strong inductive hypothesis that Merge-Sort is correct for arrays of length

n′ < n. So it correctly sorts the two subarrays A[1 .. m] and A[m + 1.. n] since both m and n - m are less than n. Merge correctly combines them by the previous proof.

- The second part is how proofs of correctness usually go for recursive algorithms. By the induction, the recursive calls do the right thing. Now you just need to argue why your combine step is correct.
- But what is the run time of Merge-Sort?
- Well, dividing the array takes Theta(1) time, and the merge operation takes Theta(n) time since you're just doing a single for loop with constant time innards.
- But when n = 1, the whole thing just takes Theta(1) time.
- Let T(n) be the *worst-case* running time on an array of length n.
- T(n) = T(\floor{n/2}) + T(\ceiling{n/2}) + \Theta(n)
- Let's fine a good upper bound on T(n).
- Floors and ceiling usually don't matter when solving recurrences for divide-and-conquer algorithms, so we'll assume n is a power of 2 to ignore them.
- We can pick a large constant c so that T(n) ≤
    - c if n ≤ 1.
    - 2T(n/2) + cn otherwise
- Let's guess and verify by induction. We'll learn better methods on Wednesday.
- Lemma: For n > 1, there exists a constant c′ > 0 such T(n) ≤_ c′ n lg n.
- Proof:
    - T(2) = 4c ≤ 2c′ for any c′ ≥_ 2c.
    - For n > 2,

      T(n) ≤ 2T(n/2) + cn

      ≤ 2 * c′ n/2 lg (n/2) + cn [IH]

      ≤ c′ n(lg n - 1) + cn

      ≤ c′ n lg n (for any c′ ≥_ c)
- So, T(n) = O(n lg n), which means Merge-Sort runs in O(n log n) time.
- With similar math, we can prove a lower bound of c″ n lg n for some small c″ > 0, meaning Merge-Sort actually takes Theta(n log n) time. The actual array involved has no great effect on the running time.