CS 6363.005.19S Lecture 3—January 22, 2019

Main topics are #recursion, #divide-and-conquer, and #recurrances

Prelude

- Please fill out the prerequisite form if you haven't already. Thank you!
- I'll release Homework 1 today or tomorrow to be due two weeks from today or tomorrow.

Reduction

- *Reduction* is probably the most important technique you should learn for designing algorithms.
- Reducing problem X to another problem Y means writing an algorithm for X that uses an algorithm for Y as a "black-box" or "subroutine".
- Correctness of the algorithm *cannot* depend on how the algorithm for Y 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, and *stay out of their business*.
- For example, we saw peasant multiplication in the first class. That algorithm used reductions to algorithms for addition, halving, and parity-checking. We never discussed how to add, half, or parity-check. We just assumed that we could do it.
- In a sense, all algorithms do reductions, its just that some of them may be to things you learned to do in grade school.
- That said, the *running time* of peasant multiplication might depend upon how we implement those black boxes, but *correctness* does not.
- Again, you don't need to know how the black boxes work, and even if you do, it's extremely useful to pretend you don't. That way, you can concentrate on what needs done for the present problem, not all the details at once.

Recursion

- *Recursion* is a special type of reduction where you reduce a problem to a simpler instance of itself.
  - If the problem instance can be solved directly, solve it directly.
  - Otherwise, reduce the problem instance to one or more **simpler instances of the same problem**.
- It's helpful to imagine that somebody else took your simpler instance and solved it for you. Following Erickson's lead, we'll call this person the *Recursion Fairy*.
- All the Recursion Fairy asks is that you give them a simpler instance of the problem. Then,

they'll take care of things using methods THAT ARE NONE OF YOUR BUSINESS. Is it magic? Who knows? (other than the Recursion Fairy)

- Mathematically, it's the same as applying the induction hypothesis in an inductive proof. The simpler statement is just true, OK?
- There's just one other catch. We need to make sure the recursion will eventually reach one or more *base cases*.
- So we almost always reduce to one or more *smaller* instance of the problem. If there are no smaller instances to reduce to, we're in a base case, and we solve it directly.
- As a simple example, let's rewrite the peasant multiplication algorithm using recursion. It relied on the following observation.

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y+y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y+y) + y & \text{if } x \text{ is odd} \end{cases}$$

- As an algorithm, it looks like

```
MULTIPLY(x, y):
  if x = 0
      return 0
  else
      x' ← ⌊x/2⌋
      y' ← y + y
      prod ← MULTIPLY(x', y')    ⟨⟨Recurse!⟩⟩
      if x is odd
          prod ← prod + y
      return prod
```

- So, if x is 0, there is nothing to do.
- Otherwise, we compute x′ and y′, and then *ask somebody else* (the Recursion Fairy) to multiply x′ and y′. Their instance is simpler, because x′ < x. How they do their multiplication is STILL NONE OF YOUR BUSINESS.

Merge Sort

- Most of what we do for the next month or so will be designing algorithms using recursion. We'll begin with a common algorithm design paradigm called divide-and-conquer.
- Before discussing *what* divide-and-conquer means, I'll show an example. Then we'll back up and discuss the higher level pattern I'm eluding to.
- We'll look at an algorithm called MergeSort proposed by John von Neumann around 1945. You may have seen it before. However, I like to use this algorithm as an example, because it's almost as if John designed it just to teach divide-and-conquer.
- So let's say we're given an array A[1.. n] of things we want to sort (numbers, letters, shoes, whatever).
  1. Divide the input array into two subarrays of roughly equal size.

2. Recursively MergeSort the two subarrays.
3. Merge the newly-sorted subarrays into a single sorted array.

- Let's look at an example and then write the details as pseudocode. **[the S and R and still transposed in this example]**

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse Left:	I	N	O	S	R	T	G	E	X	A	M	P	L
Recurse Right:	I	N	O	S	R	T	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X

- The first step is easy: just find the median array index. The second step is done by the recursion fairy. No, I won't demonstrate the recursive calls. The third step requires a good merging algorithm: here's one that runs in linear time along with the pseudocode for the whole sorting procedure.

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  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
  for k ← 1 to n
      A[k] ← B[k]
```

- Now, it's not exactly clear the algorithm does the correct thing, so let's prove it correct. This will also serve as good practice for induction.
- We'll actually need two induction proofs; one for merging and one for sorting. The first provides a nice example of performing induction over an interesting variable. The second provide a typical example of a correctness proof for a recursive algorithm.
- Lemma: Merge correctly merges the subarrays A[1..m] and A[m+1..n], assuming those subarrays are sorted in the input.
- The intuition for this proof is that the first iteration of the loop grabs the smallest element in A since its at the beginning of A[1..m] or A[m+1..n]. Then it *recursively* fills in the rest of the sorted array by going over the remaining members. But let's write out the details this one time to be sure.
- Proof: Fix A[1..n] and sorted subarrays A[1..m] and A[m+1..n].
  - We'll prove that for all k from 1 to n+1, the last n - k + 1 iterations of the main loop correctly merge A[i..m] and A[j..n] into B[k..n].
  - We'll use induction on **n - k + 1**. No, not induction on n. That happens sometimes.

Here, n - k + 1 will get smaller when we apply the induction hypothesis.
  - Now, consider any k from 1 to n + 1 (i.e., consider some n - k + 1).
  - Assume for any k′ where $0 \le n - k' + 1 < n - k + 1$ where i′ and j′ are the other indices when we're in iteration k′, the last n - k′ + 1 iterations of the main loop correctly merge A[i′..m] and A[j′..n] into B[k′..n].
  - If k > n (meaning n - k + 1 = 0), the algorithm correctly does nothing to merge two empty subarrays.
  - Otherwise:
    - If j > n, A[j..n] is empty, so min(A[i..m] cup A[j..n]) = A[i].
    - o.w., if i > m, A[i..m] is empty, so min(A[i..m] cup A[j..n]) = A[j]
    - o.w., if A[i] < A[j], then min(A[i..m] cup A[j..n]) = A[i]
    - o.w., A[i] ≥ A[j] and min(A[i..m] cup A[j..n]) = A[j].
  - In all four cases, B[k] is correctly assigned the smallest element of A[i..m] and A[j..n].
  - In the two cases with B[k] ← A[i], the induction hypothesis implies the last n - k iterations correctly merge A[i+1..m] and A[j..n] into B[k+1..n].
  - In the other cases, the induction hypothesis implies the last n - k iterations correctly merge A[i..m] and A[j+1..n] into B[k+1..n].
- Yes, that was slightly more tedious than I would require for homework, but only slightly.
- Theorem: MergeSort correctly sorts A[1..n].
- This one's easier and you should follow this pattern when arguing correctness for pretty much any recursive algorithm.
- Proof:
  - Assume the algorithm sorts arrays of length k < n.
  - If n ≤ 1, the algorithm correctly does nothing.
  - Otherwise, it correctly sorts A[1..m] and A[m+1..n]. Merge correctly merges them by the previous lemma.
- This is the usual style of proof for recursive algorithms. The algorithm works correctly on the smaller instances by the induction hypothesis. Then we make some argument that we combine the recursive results correctly.
- We'll come back to run time in a minute, but first, I want to describe the pattern MergeSort is based upon.

Divide-and-Conquer

- MergeSort is an example of a *divide-and-conquer* algorithm. They all share this form:
  1. **Divide** the given instance into several *independent smaller* instances.
  2. **Delegate** each smaller instance to the Recursion Fairy.
  3. **Combine** the solutions for the smaller instances into the final solution for the given instance.

- When the instance size falls below some threshold, the recursion *bottoms out* and you switch to some different (usually trivial or brute force) algorithm instead.
- Proving divide-and-conquer algorithms correct always requires induction.
- But how do we analyze the running time of these things?
- The trick is to write the running time as a mathematical *recurrence*, a function that is partially defined using itself.
- Let's define T(n) as the *worst-case* running time of MergeSort on an array of size n.
- For sufficiently small n, $n < n_0$ for some constant n_0, we can say $T(n) \leq C$ for some constant C (so $T(n) = O(1)$ when $n < n_0$). Sorting an array of constant length takes constant time.
- Merge takes O(n) time, because it's a simple for loop going from 1 to n. For $n \geq n_0$, we'll say merging the dividing the array takes at most cn time for some constant c.
- MergeSort calls itself twice on inputs of size ceiling(n/2) and floor(n/2) so $T(n) \leq T(\text{ceiling}(n/2)) + T(\text{floor}(n/2)) + cn$.
- Our goal is to find as tight an asymptotic (big-Oh) bound as possible to express T(n).
- But honesty, that recurrence is kind of ugly. There's floors and ceilings and that c there. I want to show you that it suffices to find an asymptotic bound for the recurrence $S(n) = 2S(n / 2) + n$.
- We can do this using a technique called a *domain transformation*.
- First, the actual constants don't matter when doing asymptotic bounds, pretty much by design. So solving $R(n) \leq R(\text{ceiling}(n/2)) + R(\text{floor}(n/2)) + n$ would also give us the asymptotic running time bound for MergeSort.
- The worst-case running time of MergeSort is increasing in n, so we can only make things worse by assuming the recurrence uses two ceilings instead of a ceiling and a floor. Let's solve $R(n) \leq 2R(\text{ceiling}(n/2)) + n \leq 2R(n/2 + 1) + n$.
- Still weird looking, but now we can play one more trick to get ride of that +1.
- Let S(n) = R(n + alpha). Hopefully we can get a clean S(n/2) in the recurrence if we just look at slightly higher values of R.
- S(n)
  - = R(n + alpha)
  - $\leq 2R(n/2 + \text{alpha}/2 + 1) + n + \text{alpha}$
  - $= 2S(n/2 - \text{alpha}/2 + 1) + n + \text{alpha}$
- If we set alpha = 2, we get $S(n) = 2S(n/2) + n + 2$. One important fact for big-Oh notation is that when adding functions together, only the biggest part of the function matters. So we may as well solve $2S(n/2) + n$ like we wanted to.
- Finally, R(n) = S(n - 2). Plugging that n - 2 into any reasonable running time function g(n) won't change the big-Oh bound, so it suffices to solve $S(n) = 2S(n/2) + n$.
- Yeah, that was a lot of junk, and it's totally not worth doing for every divide-and-conquer algorithm.

- From now on, I'm going to ask you to trust me that we can just ignore floors, ceilings, lower order terms, and the specific constant on the non-recursive part.
- So in the case of MergeSort, we'll just say the running time is bounded by $T(n) \leq 2T(n/2) + n$.
- On Thursday, I'll show you how to actually solve this and similar recurrences for divide-and-conquer algorithms so we can determine their running times. Then we'll look at another example and (much more quickly) analyze its running time.