CS 6363.003.21S Lecture 2—January 21, 2021

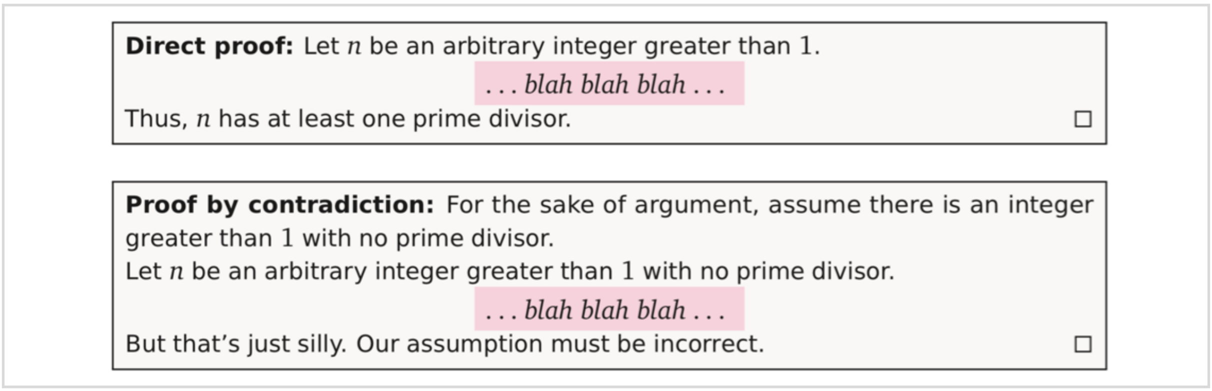
Main topics are #reductions, #induction, and #recursion#.

Reductions

- On Tuesday, we went over what an algorithm is, and how we're expected to describe them. Now we're ready to actually discuss designing them.
- I firmly believe that the single most important technique to get out of this class not just for designing algorithms but for almost anything to do with computers or mathematics (science?) is *reductions*.
- 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 for X *does not* depend on how the algorithm for Y works. The whole point is that somebody else solved Y for you and you don't need or even want to worry about how.
- In short, you still care about the What of Y and likely the How fast, but your algorithm should not depend at all upon the How or Why.
- For example, FibonancciMultiplication uses a reduction to addition. The particular details of how to add two numbers or why your favorite addition algorithm works don't matter. We just need to know that you can in fact add two numbers, and maybe how long that takes. That's it.
- If you've ever worked with the C++ or Java standard libraries you're probably familiar with what I'm talking about. The documentation tells you what the functions do and maybe how long they take, but the implementation details are a mystery and may even be different on different systems.
- And even if the details weren't a mystery, it's often extremely helpful to pretend they are. You'll have more room in your brain for designing your own algorithm if you're not busy thinking about how somebody else's works.
- The same idea applies to pure mathematics as well. Mathematicians will prove lemmas, simple theorems that don't necessarily have much use on their own, so they can refer to the *statement* of the lemma later without having to reprove it.
- And both pure mathematics and algorithms often rely on special types of reductions known as *induction* and *recursion*.
- These types of reduction form the basis of most of the algorithm design paradigms we're going to see in this class, but many students and even some professionals have some misconceptions about them. So today, we're going to review induction and recursion, focusing on many of the more useful points that don't always come across the first time you see them.

Prime Divisors and Induction

- We'll begin with induction. To help motivate what induction is, we're going to iterate over a couple proofs of a fundamental theorem in algebra (but not **the** fundamental theorem of algebra).
- Let n be a positive integer.
- A *divisor* of n is a positive integer p such that n / p is also an integer.
- Positive integer n is *prime* if it has exactly two divisors, n and 1.
- n is *composite* if it has more than two divisors.
- So, if n = 1, then it is neither prime nor composite.
- Theorem: Every integer n greater than 1 has a prime divisor.
- This is a universally quantified statement. We need to prove it about *all* the integers greater than one. And there are two ways to do that.



Direct proof: Let n be an arbitrary integer greater than 1.

. . . blah blah blah . . .

Thus, n has at least one prime divisor. □

Proof by contradiction: For the sake of argument, assume there is an integer greater than 1 with no prime divisor.
Let n be an arbitrary integer greater than 1 with no prime divisor.

. . . blah blah blah . . .

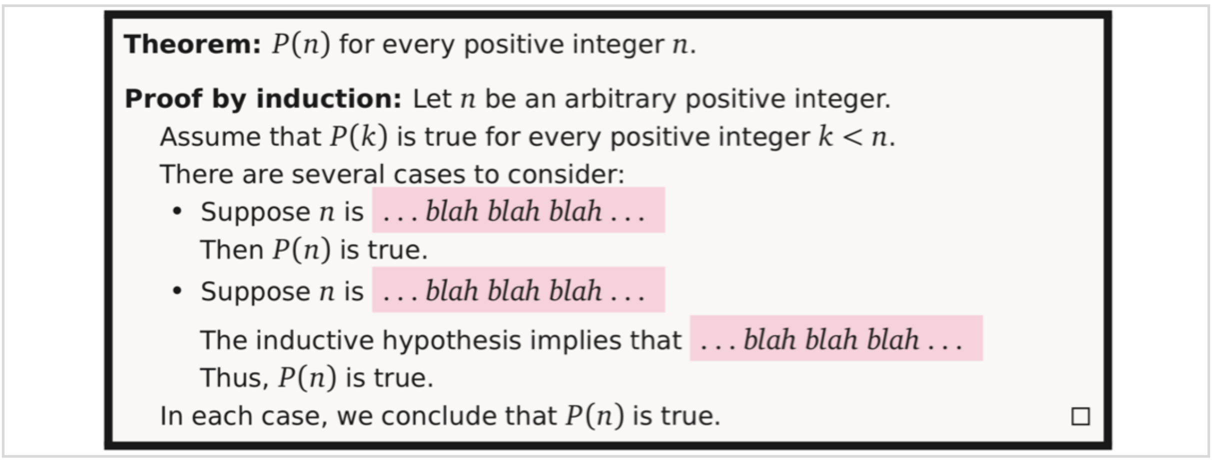
But that's just silly. Our assumption must be incorrect. □

- Proofs by contradiction are usually easier to discover, so let's start there.
- And to make things easier, we're going to pick a very specific counterexample. The *smallest* one.
- Proof by contradiction:
  - For the sake of argument, assume there is an integer greater than 1 with no prime divisor.
  - Let n be **the smallest** integer greater than 1 with no prime divisor.
    - Since n is a divisor of n and n has no prime divisors, n cannot be prime.
    - Thus, n must have at least one divisor d such that 1 < d < n.
  - Let d be an arbitrary divisor of n such that 1 < d < n.
    - **Because n is the smallest counterexample, d has a prime divisor.**
  - Let **p** be a prime divisor of d.
    - Because d / p is an integer, (n / d) * (d / p) = n / p is also an integer.
    - Thus, p is also a divisor of n.
    - But this contradicts our assumption that n has no prime divisors!
  - So our assumption must be incorrect.
- Great, we have a proof!

- But we can do better. Contradiction proofs are easy to write, but direct proofs are much easier to read.
- But something to notice about that previous proof: When we assumed n is the smallest counterexample, we, uh, assumed there are no smaller counterexamples.
- So let's just directly state our assumption that there are no smaller counterexamples and use it in a direct proof.
- Direct proof: Let n be an arbitrary integer greater than 1.
    - **Assume that for every integer k such that 1 < k < n, integer k has a prime divisor.**
    - There are two cases to consider: Either n is prime, or n is composite.
    - Suppose n is prime.
        - Then n is a prime divisor of n.
    - Suppose n is composite.
        - Then n has a divisor d such that 1 < d < n.
        - Let d be a divisor of n such that 1 < d < n.
        - By assumption (i.e. there are no smaller counterexamples), d has a prime divisor.
        - Let p be a prime divisor of d.
        - Because d / p is an integer, (n / d) * (d / p) = n / p is also an integer.
        - Thus, p is also a divisor of n.
    - In each case, we conclude that n has a prime divisor.
- This is called *proof by induction*.
- The assumption that there are no counterexamples smaller than n is called the *induction hypothesis*.
- The case we argued directly is called a *base case*. Yes, *every* prime number is a base case. While it is *usually* the true, nothing says the base cases have to be small. And there may even be an infinite number of them.
- The other case was the *inductive case*.
- Maybe you want to label the induction hypothesis, the base case(s), and the inductive case(s). That's up to you and how comfortable you feel doing these proofs.
- But until you're very comfortable doing these proofs, you do need to explicitly write out your induction hypothesis and make the case analysis obviously exhaustive.
- Again, I want to emphasize there is very little difference between this proof by induction and a proof by smallest counterexample except in readability.
- In a sense, our original argument relied on the observation that if n has no prime divisor, then there must be some smaller positive integer greater than 1 with no prime divisor.
- The direct proof relies on the contrapositive of this observation, "every 1 < k < n has a prime divisor => n has a prime divisor"

Proofs by Induction

- So what's the recipe here?
  1. **Write down the boilerplate (template).** Write down the universal invocation "Let n be an arbitrary…", the induction hypothesis, the conclusion, and leave blank space for the remaining details. **This is the easy part.** In fact, I'll just leave this here.



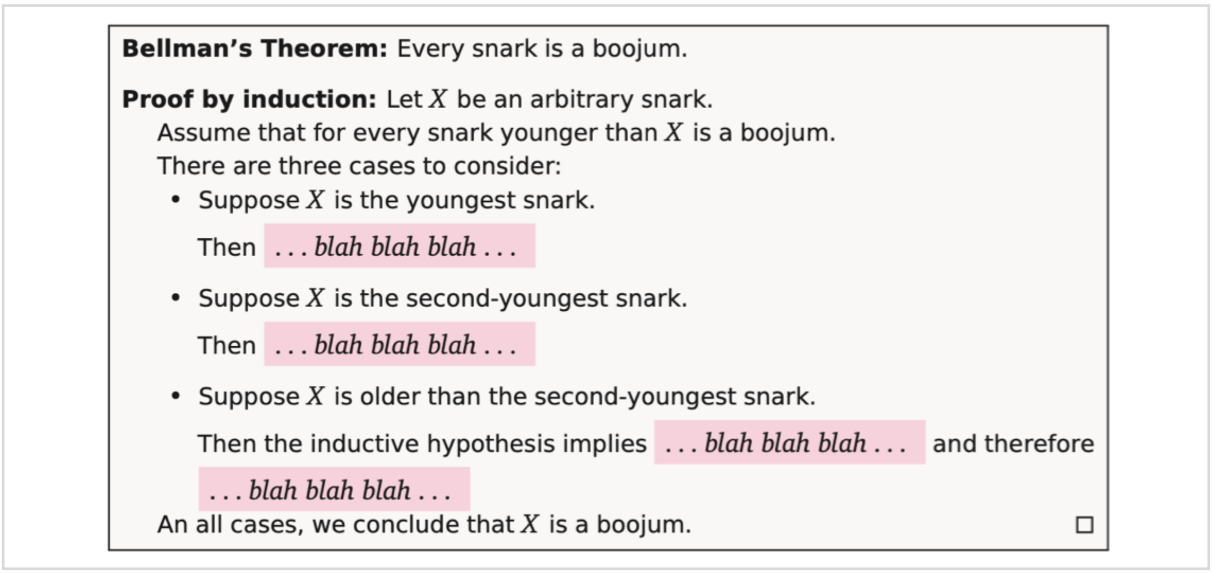
Theorem: $P(n)$ for every positive integer n.

Proof by induction: Let n be an arbitrary positive integer.
Assume that $P(k)$ is true for every positive integer $k < n$.
There are several cases to consider:

- Suppose n is *. . . blah blah blah . . .*
  Then $P(n)$ is true.
- Suppose n is *. . . blah blah blah . . .*
  The inductive hypothesis implies that *. . . blah blah blah . . .*
  Thus, $P(n)$ is true.

In each case, we conclude that $P(n)$ is true. □

or more abstractly:



Bellman's Theorem: Every snark is a boojum.

Proof by induction: Let X be an arbitrary snark.
Assume that for every snark younger than X is a boojum.
There are three cases to consider:

- Suppose X is the youngest snark.
  Then *. . . blah blah blah . . .*
- Suppose X is the second-youngest snark.
  Then *. . . blah blah blah . . .*
- Suppose X is older than the second-youngest snark.
  Then the inductive hypothesis implies *. . . blah blah blah . . .* and therefore *. . . blah blah blah . . .*

An all cases, we conclude that X is a boojum. □

  2. **Think big.** Don't think about small numbers like 1 or 5 or 10^100. And DO NOT think how to solve the problem all the way down to the ground. Think about how to reduce proofs about gigantic numbers (the inductive case) to claims about some other smaller number or numbers. Seriously, the key thing that makes this all possible for a human is that we can compartmentalize. In a sense, we reduced the proof for n to claims for k < n. We don't have to worry about why things are true for those k, so just trust they are.**This is the hard part.**
  3. **Look for holes.** Look for cases where your inductive argument breaks down (the base cases) and solve them directly. Don't be clever; be stupid but thorough. Doing base cases *after* the inductive cases makes it much easier to verify you really did handle the right set of base cases. **Seriously, do the base cases last!**
  4. **Rewrite everything.** You probably didn't leave the correct amount of space. Rewrite

the proof so the argument is easier for someone (the TA) to follow.

- All cases in induction proofs are either inductive cases or base cases. Base cases are *usually* a few small values of n, but they may be other things like all prime numbers. Induction proofs are usually easier to read if they describe the base cases first, but you should attempt to figure out the inductive cases first so you know which gaps need to be filled in with base cases.
- Until you are so comfortable with induction that you can just nod along thinking "ah yes, of course" whenever I make a claim based on an inductive assumption, I highly recommend you stick to full boilerplate for your proofs. **Write it down before you even start thinking.** I may even require the boilerplate for Homework 1.
- This boilerplate captures every kind of induction you might do. Claims about integers. Claims about graphs (where n is the number of vertices or edges). *Anything*.
- But here are two things you should **never, ever do** even if you're careful enough to make them technically correct.
- **Do not** make the induction hypothesis refer to k = n - 1. Use all k < n just in case you need them. Why would you tie n - 2 arms behind your back?
- **Do not** Assume the theorem for n and then attempt a proof for n + 1, especially when arguing about something "structural" like graphs. In doing so, you'll be tempted to build a convenient (n + 1)st object by modifying the nth object instead of arguing about an arbitrary object directly. Now you have to prove that you were able to create any arbitrary object using your construction which is more work, harder to follow, and much easier to mess up. Also, you're doing a direct proof involving an arbitrary number. Those always involve proving something about n, not n + 1. And yes, it's likely you were introduced to induction using n + 1. I consider that a bad thing.

Recursion

- The algorithm design counterpart to induction is called *recursion*.
- *Recursion* is a special type of reduction where you reduce a problem to a simpler instance of itself.
    - First, you try to reduce the problem instance to one or more **simpler instances of the same problem**.
    - And in the cases where you can't or it's inconvenient to do so (often when the input is sufficiently small), you just solve the problem directly.
- Just like how applying an induction hypothesis is like using a lemma, for recursion it's helpful to imagine somebody else taking the simpler instance of your problem and solving 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)

- Again, it's the same as applying the induction hypothesis in an inductive proof. The simpler statement is just true, OK?
- All that said, we do need to be a bit careful here. Induction requires some value (usually n) to decrease; the induction hypothesis talks about smaller values. For recursion, "simpler instance" means something needs to decrease so we're guaranteed to hit a *base case*.
- So we almost always reduce to one or more *smaller* instance of the problem. If we can't easily reduce to a smaller instance, then we're in a base case, and we solve our original instance directly.
- As a simple example, let's consider another multiplication algorithm for non-negative integers sometimes called *peasant multiplication*. It relies on the following observation:

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

and understanding how to add and half integers.

- As an algorithm, it looks like

```
PeasantMultiply($x, y$):
    if $x = 0$
        return 0
    else
        $x' \leftarrow \lfloor x/2 \rfloor$
        $y' \leftarrow y + y$
        $prod \leftarrow$ PeasantMultiply($x', y'$)    ⟨⟨Recurse!⟩⟩
        if $x$ is odd
            $prod \leftarrow prod + y$
        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.
- But seriously, how do we know this algorithm is correct? It turns out the connection between induction and recursion is more than conceptual. **Correctness of recursive algorithms always follows from induction.**
- If x = 0, then we correctly return x * y = 0. Otherwise, we assume PeasentMultiply is correct when the first parameter x' < x. So that recursive call works, and the product is computed correctly according to the formula.
- When I said earlier that recursion forms the basis of most of the algorithm design paradigms for this class, I was holding back a bit. It really forms the basis of nearly every algorithm we write.

- People that do a lot of functional programming are probably already comfortable with this concept, but even something as fundamental as a for loop can be thought of recursively, and its correctness argued inductively: The first iteration does something useful. You then recursively do something useful with the rest of whatever you're looping over. The second half of this process involves strictly fewer iterations, so you know it succeeds by induction. Or vice versa, you recursively do something in the first n - 1 iterations and then combine that work with the last iteration.
- Next week, we'll start discussing the divide-and-conquer paradigm, one of the most "pure" algorithm design paradigms based on recursion. Along the way, we'll do a refresher on how to describe algorithm running times, and discuss how to successfully analyze this class of recursive algorithms.