Structural induction and Sets

1 Induction, Strong Induction, and Well-Ordering

1.1 Induction vs. strong induction

Strong induction looks like a more powerful proof method than ordinary induction, because it allows more information to be assumed in the inductive step. But it's not really stronger—anything that can be proved with strong induction can also be proved with ordinary induction. Here's an example.

Recall the proof that every natural number ≥ 2 can be written as product of primes (factored into primes). We used strong induction, where $P(n)$ was the predicate "n can be factored into primes". The inductive step involved writing $n+1 = ab$, and using $P(a)$ and $P(b)$ to get factorizations for a and b, then multiplying them.

Now we'll change this to an ordinary inductive proof. Define a new predicate $Q(n)$ to be $\forall i, 2 \leq i \leq n$ (i can be factored). We'll prove $Q(n)$ by ordinary induction:

$Q(2)$: Same as $P(2)$, use same proof.

$Q(n) \Rightarrow Q(n+1)$:
Our goal is now to show: $\forall i, 2 \leq i \leq n+1$ (i can be factored).
What we can assume is: $\forall i, 2 \leq i \leq n$ (i can be factored).

Notice that all the needs to be shown is that $n+1$ can be factored (the others already assumed). And we can assume factorizations of all smaller numbers. So the old strong induction step can be used to show this.
This idea is quite general. We can always convert a strong inductive proof of $\forall n \geq k(P(n))$ to an ordinary inductive proof. Just define $Q(n)$ to be $\forall i, k \leq i \leq n(P(i))$, that is, $P(k) \wedge P(k+1) \wedge \cdots \wedge P(n)$.

Base case: Show $Q(k)$, same as $P(k)$, which has already been shown as part of the strong induction proof.

Inductive step: Show $Q(n) \Rightarrow Q(n+1)$, in other words, $P(k) \wedge P(k+1) \wedge \cdots \wedge P(n) \Rightarrow P(k) \wedge P(k+1) \wedge \cdots \wedge P(n) \wedge P(n+1)$.

This really amounts to showing $P(k) \wedge P(k+1) \wedge \cdots \wedge P(n) \Rightarrow P(n+1)$. But this has already been shown in the strong induction proof.

Of course, any ordinary inductive proof **is** a strong induction proof has well. We are allowed to assume $P(1), \ldots, P(n)$ to prove $P(n+1)$, but all we actually use is $P(n)$.

Induction and strong induction are said to be "equivalent" proof rules, in that each can prove the same things (in the context of some standard number axioms and logical deduction rules).

1.2 Well-ordering

There is another variation of induction that is sometimes useful: well-ordering. The well-ordering axiom looks nothing like the induction axiom:

Axiom Every nonempty subset $S \subseteq \mathbb{N}$ has a smallest element.

The well-ordering axiom is often called the well-ordering principle. It may seem obvious and useless!

As for obvious, note that this axiom would be false if the set of natural number, $\mathcal{N}$, were replaced by, say, the set of rational numbers. The subset S consisting of positive rational numbers has no smallest element.

As for useless, it turns out the well-ordering is just as powerful as our very best weapon, strong induction!

For example: we can reprove the theorem about 3-cycles using well-ordering instead of induction.

Theorem 1.1 Any tournament that contains a cycle contains a 3-cycle.

Proof. Assume tournament T contains a cycle. Let n be the smallest cycle length (there is some smallest, by well-ordering). n must be at least 3. If it's 3, then we're done. So suppose $n \geq 4$. Consider a cycle with this length. Now consider p_1 and p_3 as before. Whichever way the arrow goes, we can use it in constructing a shorter cycle in T (as we did before). This is a contradiction, since we assumed that n is the smallest cycle length. Since we got a contradiction to our assumption that $n \geq 4$, we must have $n = 3$ as claimed. ■

This proof is really very similar to the other one. In general, well-ordering is equivalent to induction (and strong induction), in that they can be used to prove the same theorems. Again the transformation is standard. Given an induction proof for proposition $P(n)$, we can build a well ordering proof as follows. Suppose for the purpose of contradiction that $P(n)$ is *not* true for every n. That means it is false for some n. So consider the set of elements n for which $P(n)$ is false. We've just assumed this set is nonempty. So by well ordering it has a smallest element, say k. This means $P(k-1)$ is true (else $k-1$ would be in our set). But in our original inductive proof for $P(n)$, we showed that $P(k-1) \implies P(k)$ (this is the inductive step). So $P(k)$ must be true. This is a contradiction to our choice of k, which shows that our original assumption (that $P(n)$ is false for some n) is false.

2 Structural induction

Structural induction is a variant of induction for proving things about other data types besides N, in particular, about data types that are defined recursively. The induction is based explicitly on the recursive structure of the definition.

2.1 Some recursive definitions for data types

Fully parenthesized Boolean algebra expressions (with ands, ors, nots only).

Definition 2.1 The set F of fully parenthesized Boolean algebra expressions satisfies:

1. The symbols 0 and 1 are in F.
2. If e and e' are in F then the expression $(e \wedge e')$ is in F.
3. If e and e' are in F then the expression $(e \vee e')$ is in F.
4. If e is in F then the expression $(\neg e)$ is in F.
5. F contains nothing else.

And here is one for binary trees.

Definition 2.2 The set T of binary trees satisfies:

1. A single node (just a dot) is in T.
2. If t is in T, then the structure formed from t, a single new node, and an arrow labeled "left" from the new node to t, is also in T.
3. If t is in T, then the structure formed from t, a single new node, and an arrow labeled "right" from the new node to t, is also in T.
4. If t and t' are in T, then the structure formed from t, t', and a single new node, with an arrow labeled "left" from the new node to t and another arrow labeled "right" from the new node to t', is also in T.

Since the descriptions above are wordy, it's nice to introduce some notation, like makeleft(t) to denote the thing built up from t in the first recursive step. Likewise, makeright(t), makeboth(t,t').

And here is one for strings over an alphabet A.

Definition 2.3 The set S of (finite length) strings over alphabet A satisfies:

1. λ, the empty string, is in S.
2. If s is in the set S and $a \in A$, then sa is in S.
3. S contains nothing else.

This is a recursive description (definition) of the set of strings. Notice that it includes every (finite length) string. Moreover, every string has a unique derivation using these rules.

2.2 Inductive proofs based on recursive data type definitions

Structural induction is used to prove a property P of all the elements of some recursively-defined data type. The proof consists of two steps:

- Prove for the "base cases" of the definition.
- Prove for the result of any combination rule, assuming that it is true for all the parts.

For Boolean expressions:

Prove: $\forall e \in F(P(e))$
1. (Base) $P(0)$
2. (Base) $P(1)$
3. (Inductive step) $\forall e, e' \in F(P(e) \wedge P(e') \Rightarrow P((e \wedge e')))$
4. (Inductive step) $\forall e, e' \in F(P(e) \wedge P(e') \Rightarrow P((e \vee e')))$
5. (Inductive step) $\forall e \in F(P(e) \Rightarrow P((\neg e)))$
6. QED — Structural induction on Boolean expressions

For binary trees:

Prove: $\forall t \in T(P(t))$
1. (Base) $P(n)$, where n is the single-node tree.
2. (Inductive step) $\forall t \in T(P(t) \Rightarrow P(makeleft(t)))$
3. (Inductive step) $\forall t \in T(P(t) \Rightarrow P(makeright(t)))$
4. (Inductive step) $\forall t, t' \in T(P(t) \wedge P(t') \Rightarrow P(makeboth(t, t')))$
5. QED — Structural induction on binary trees

For strings over alphabet $A = \{0, 1\}$:

Prove: $\forall s \in S(P(s))$
1. (Base) $P(\lambda)$
2. (Inductive step) $\forall s \in S(P(s) \Rightarrow P(s0))$
3. (Inductive step) $\forall s \in S(P(s) \Rightarrow P(s1))$
4. QED — Structural induction on strings.

You can see how this would extend to different alphabets.

An example for each of the data types:

Theorem 2.4 Every Boolean expression has the same number of left and right parentheses.

Proof. Define $P(e)$: Expression e has the same number of left and right parentheses.

Prove: $\forall e \in F(P(e))$
1. (Base) $P(0)$ — The expression 0 has no parens.
2. (Base) $P(1)$ — No parens
3. (Inductive step) $\forall e, e' \in F(P(e) \wedge P(e') \Rightarrow P((e \wedge e'))$
   1. Fix $e, e' \in F$.
   2. Assume $P(e) \wedge P(e')$, that is, each of e and e' has the same number of left and right parens.
   3. $P((e \wedge e'))$, that is, the expression $e \wedge e'$ has the same number of left and right parens. — By inductive hypothesis (3.2), and the fact that we're adding one left and one right paren.
   4. QED — Implication, UG

4. (Inductive step) $\forall e, e' \in F(P(e) \wedge P(e') \Rightarrow P((e \vee e'))$ — Similar to the previous case.
5. (Inductive step) $\forall e \in F(P(e) \Rightarrow P((\neg e))$ — Similar to the previous case.
6. QED — Structural induction on Boolean expressions

Be careful of confusion of parens above. Some are used for P, some are parts of the actual Boolean expressions...this really needs better notation. ■

Theorem 2.5 The number of edges in any binary tree is exactly one fewer than the number of nodes.

Proof. Define $P(t)$: Binary tree t has exactly 1 fewer edge than nodes. We are using the notation $nodes(t)$, $edges(t)$. The key is that the inductive steps all preserve the "deficit" of 1. For instance, combining two trees involves adding two edges and one node; since each subtree had a "deficit" of 1, this maintains the deficit of 1:

Prove: $\forall t \in T(P(t))$
1. (Base) $P(n)$, where n is the single-node tree. — 0 edges and 1 node
2. (Inductive step) $\forall t \in T(P(t) \Rightarrow P(makeleft(t))$
    1. Fix t.
    2. Assume $P(t)$, that is, $|edges(t)| + 1 = |nodes(t)|$.
    3. $P(makeleft(t))$, that is, $|edges(makeleft(t))| + 1 = |nodes(makeleft(t))|$.
        1. $|edges(makeleft(t))| = |edges(t)| + 1$
        2. $|nodes(makeleft(t))| = |nodes(t)| + 1$.
        3. QED — Algebra, using 2.2, 2.3.1, 2.3.2
    4. QED — Implication, UG
3. (Inductive step) $\forall t \in T(P(t) \Rightarrow P(makeright(t))$ — Similar to the previous case.
4. (Inductive step) $\forall t, t' \in T(P(t) \wedge P(t') \Rightarrow P(makeboth(t, t'))$
    1. Fix t.
    2. Assume $P(t) \wedge P(t')$, that is, $|edges(t)| + 1 = |nodes(t)|$ and $|edges(t')| + 1 = |nodes(t')|$.
    3. $P(makeboth(t, t'))$, that is, $|edges(makeboth(t, t'))| + 1 = |nodes(makeboth(t))|$.
        1. $|edges(makeboth(t, t'))| = |edges(t)| + |edges(t')| + 2$
        2. $|nodes(makeboth(t, t'))| = |nodes(t)| + |nodes(t')| + 1$.
        3. QED — Algebra, using 4.2, 4.3.1, and 4.3.2
    4. QED — Implication, UG
5. QED — Structural induction on binary trees

■

Theorem 2.6 In a string of 0s and 1s, the number of occurrences of the pattern 01 is less than or equal to the number of occurrences of 10, plus one.

Let's try to prove this by structural induction. First we must define $P(s)$. Let's write $num(pat, s)$ as the number of occurrences of the pattern string pat in s. Now our inductive hypothesis is

$$P(s) : num(01, s) \leq num(10, s) + 1.$$

If you try to prove this by structural induction, you will get stuck. Why? Consider what happens when you add 1 at the end. This could increase the number of 01s without increasing the number of 10s.

So, to prove by structural induction on strings, let's strengthen the hypothesis by adding another clause. If a string ends in 0 then the number of 01s is less than or equal to the number of 10s. That solves the problem by weakening what we have to show when the string ends in 1. But maybe it causes another problem somewhere else. Let's give it a try:

Redefine $P(s)$:
$num(01, s) \leq num(10, s) + 1$,
and
If s ends in 0 then $num(01, s) \leq num(10, s)$.

This means that, for each inductive step, we have two things to show:

Prove: $\forall s \in S(P(s))$
1. (Base) $P(\lambda)$ — No patterns of either kind.
2. (Inductive step) $\forall s \in S(P(s) \Rightarrow P(s0))$
   1. Fix s.
   2. Assume $P(s)$.
   3. $P(s0)$
      1. $num(01, s0) \leq num(10, s0) + 1$. — ???
      2. If $s0$ ends in 0 then $num(01, s0) \leq num(10, s0)$. — ???
      3. QED — Conjunction
   4. QED — Implication, UG
3. (Inductive step) $\forall s \in S(P(s) \Rightarrow P(s1))$
   1. Fix s.
   2. Assume $P(s)$.
   3. $P(s1)$
      1. $num(01, s1) \leq num(10, s1) + 1$. — ???
      2. If $s1$ ends in 0 then $num(01, s1) \leq num(10, s1)$. — ???
      3. QED — Conjunction
   4. QED — Implication, UG
4. QED — Structural induction on strings.

First let's consider $s1$. This is the case that looks dangerous, because it might increase the number of 01s. We have to prove two statements. The second is easy, because the new string doesn't end in 0. We say it's "vacuously true".

The first statement now takes some work. We might be adding to the number of 01s. However, if we do, the previous string must have ended with 0. Then the inductive hypothesis says that the previous string had to satisfy the stronger inequality in the second statement. Adding one to the LHS of the stronger inequality yields the weaker inequality we want.

The following proof fragment considers cases based on whether s ends in 0 or not. If not, it might end in 1, or might be empty (don't forget this possibility).

3. $P(s1)$
   1. $num(01, s1) \leq num(10, s1) + 1$.
      1. If s ends in 0 then $num(01, s1) \leq num(10, s1) + 1$.
         1. Assume s ends in 0.
         2. $num(01, s) \leq num(10, s)$ — Inductive hypothesis (3.2), part 2.
         3. $num(01, s1) = num(01, s) + 1$ — Adding one more 01.
         4. $num(10, s1) = num(10, s)$
         5. $num(01, s1) \leq num(10, s1) + 1$. — Algebra (combining 3.3.1.1.2, ...3, and ...4)

         QED — Implication
      2. If s ends in 1 then $num(01, s1) \leq num(10, s1) + 1$.
         Inductive hypothesis, part 1; no new 01s.
      3. If $s = \lambda$ then $num(01, s1) \leq num(10, s1) + 1$.
         $s1$ is just 1, which has no 01s.
      4. QED — Cases
   2. If $s1$ ends in 0 then $num(01, s1) \leq num(10, s1)$.
      Vacuously true, because $s1$ doesn't end in 0.
   3. QED — Conjunction

Of course, you could also expand the step for s ending in 1 into a careful series of inequalities.

Now consider $s0$. We hope that what we did to make the $s1$ case work doesn't mess up the $s0$ case. But We have to check.

The first statement is easy. It follows from the first statement of the inductive hypothesis for s, because we are not increasing the number of 01s.
But now the second statement takes more work. The difficulty is that the new string ends in 0, which means that we have to show the stronger inequality in the second statement. But to do this, we might only have the weaker inequality for the previous string. The argument again depends on what the previous string s ended with. So again, we consider cases, based on whether s ends in 0 or 1, or is empty.
If s ends in 0 we rely on the second statement of the inductive hypothesis for s (with the stronger inequality), whereas if s ends in 1 we rely on the first statement (with the weaker inequality). In this case, we have to "turn the weaker inequality into the stronger inequality".

3. $P(s0)$
   1. $num(01, s0) \leq num(10, s0) + 1$. — Inductive hypothesis, part 1; no new 01s.
   2. If $s0$ ends in 0 then $num(01, s0) \leq num(10, s0)$.
      1. $num(01, s0) \leq num(10, s0)$.
         1. If s ends in 0 then $num(01, s0) \leq num(10, s0)$.
            Inductive hypothesis, part 2; no new 01s
         2. If s ends in 1 then $num(01, s0) \leq num(10, s0)$.
            1. Assume s ends in 1.
            2. $num(01, s) \leq num(10, s) + 1$. — Inductive hypothesis, part 1
            3. $num(01, s0) = num(01, s)$
            4. $num(10, s0) = num(10, s) + 1$ — Exactly one new occurrence of 10.
            5. $num(01, s0) \leq num(10, s0)$ — Algebra
            6. QED — Implication
         3. If $s = \lambda$ then $num(01, s0) \leq num(10, s0)$.

$s0$ is just 1, which has no 01s or 10s.
4. QED Cases
2. QED Propositional reasoning (truth table)
3. QED Conjunction

If you actually write out all these cases in the proof, you will notice that some facts are stated repeatedly, e.g., that when you add a 0 to the end of a string you are not increasing the number of 01s.
To avoid having to state these facts several times, you can move them earlier in the proof.

Note that these proofs aren't very different from ordinary inductive (or strong inductive) proofs. In fact, you can prove all the same theorems by strong induction on the number of "formation steps"—that is, the number of times you have to apply the recursive construction rules to get the object you are proving about. The base cases correspond to 0 applications of combination operations.

2.3 Recursively-defined functions on recursively-defined data types

Recursive definitions provide a natural way to define functions whose domains are recursively-defined data types.

Example 2.7 (Boolean Expressions) Define *eval*, a function from Boolean expressions to $\{0,1\}$, by:
$eval(0) = 0$
$eval(1) = 1$
$eval((e \wedge e')) = 1$ if $eval(e) = eval(e') = 1$, 0 otherwise.
$eval((e \vee e')) = 0$ if $eval(e) = eval(e') = 0$, 1 otherwise.
$eval((\neg e)) = 0$ if $eval(e) = 1$, 1 otherwise.

Example 2.8 (Binary Trees) Define the function $numnodes(t)$ (a formal definition of what we understand intuitively to be the number of nodes in the tree t), by:

$numnodes(\text{single node}) = 1$
$numnodes(makeleft(t)) = numnodes(t) + 1$
$numnodes(makeright(t)) = numnodes(t) + 1$
$numnodes(makeboth(t, t')) = numnodes(t) + numnodes(t') + 1$

Define $numedges(t)$:
$numedges(\text{single node}) = 0$
$numedges(makeleft(t)) = numedges(t) + 1$
$numedges(makeright(t)) = numedges(t) + 1$
$numedges(makeboth(t)) = numedges(t) + numedges(t') + 2$

We can use structural induction to prove properties of functions defined in this way. For example, we can reprove the relationship between numbers of nodes and edges in terms of the formal functions *numnodes* and *numedges*. (The argument is pretty much the same as before, just using the formal function definitions instead of relying on our understanding of the sizes of sets.)

Define $P(t)$: $\mathit{numnodes}(t) = \mathit{numedges}(t) + 1$.

Prove: $\forall t \in T(P(t))$

1. (Base) $P(t_0)$, where t_0 is the single-node tree. — By definition, $\mathit{numnodes}(t_0) = 1$ and $\mathit{numedges}(t_0) = 0$
2. (Inductive step) $\forall t \in T(P(t) \Rightarrow P(\mathit{makeleft}(t))$
   1. Fix t.
   2. Assume $P(t)$, that is, $\mathit{numedges}(t) + 1 = \mathit{numnodes}(t)$.
   3. $P(\mathit{makeleft}(t))$, that is, $\mathit{numedges}(\mathit{makeleft}(t)) + 1 = \mathit{numnodes}(\mathit{makeleft}(t))$.
      1. $\mathit{numedges}(\mathit{makeleft}(t)) = \mathit{numedges}(t) + 1$ — Definition of *numedges*
      2. $\mathit{numnodes}(\mathit{makeleft}(t)) = \mathit{numnodes}(t) + 1$ — Definition of *numnodes*
      3. QED — Algebra
   4. QED
3. (Inductive step) $\forall t \in T(P(t) \Rightarrow P(\mathit{makeright}(t))$ — Similar to the previous case.
4. (Inductive step) $\forall t, t' \in T(P(t) \wedge P(t') \Rightarrow P(\mathit{makeboth}(t, t'))$
   1. Fix t.
   2. Assume $P(t) \wedge P(t')$, that is, $\mathit{numedges}(t) + 1 = \mathit{numnodes}(t)$ and $\mathit{numedges}(t') + 1 = \mathit{numnodes}(t')$.
   3. $P(\mathit{makeboth}(t, t'))$, that is, $\mathit{numedges}(\mathit{makeboth}(t, t')) + 1 = \mathit{numnodes}(\mathit{makeboth}(t))$.
      1. $\mathit{numedges}(\mathit{makeboth}(t, t')) = \mathit{numedges}(t) + \mathit{numedges}(t') + 2$ — Definition of *numedges*
      2. $\mathit{numnodes}(\mathit{makeboth}(t, t')) = \mathit{numnodes}(t) + \mathit{numnodes}(t') + 1$ — Definition of *numnodes*
      3. QED — Algebra
   4. QED
5. QED — Structural induction on binary trees

3 Induction and Recursive Algorithms

We've spent a lot of time studying induction.
This is because induction comes up all the time in analyzing computation.
Why? Well, induction is a "one step at a time" proof method. Computations also evolve "one step at a time."

In fact, many of our inductive proofs can be seen as *algorithms* for solving a problem.

We proved "any 2^n by 2^n board can be tiled with Ls."
If you look at the inductive proof, we actually gave a *procedure* for carrying out that tiling.
Break the board in four, tile each piece, and merge the tilings.
What kind of algorithm is this? *recursive:* it calls itself on smaller subproblems.

Similarly, the definition of the Fibonacci numbers is also expressing a recursive algorithm for computing them.
And the inductive proof for Nim gave a game-winning algorithm.

Just about any inductive prove that a problem has a solution is in fact providing a recursive algorithm for solving the problem.

Similarly, when you have a recursive algorithm that you think solves some problem, and induction is the usual way to prove that it works.

3.1 RSA

RSA is at the core of many of the secure communication protocols we use today (SSH, SSL, etc). RSA works by treating any given message (sequence of bits) as a big number x and applying a mathematical function that "hides" the number x unless you know a secret key.

More precisely, it computes the remainder r of x^a on division by b, where a and b are two specially constructed *large* numbers. It turns out that if someone is given r and knows the right secret about the origin of a and b, they can "invert" the exponentiation and get x.

So, how can we compute the exponentiation?
The most obvious way is to work from the *recursive definition*:

- $x^0 = 1$
- $x^n = x \cdot x^{n-1}$

This is a definition of x^n, but also gives a recursive algorithm for computing it:

Procedure $\mathit{Exp}(x, n)$

if $n = 0$
    return 1
else
    return $x * \mathit{Exp}(x, n-1)$

Now it's pretty "obvious", but how would you *prove* that algorithm *Exp* returns the right answer?
Well, it doesn't always: what happens if you call $\mathit{Exp}(5, -2)$? $\mathit{Exp}(8, 2.5)$?
(Trying to prove correctness can reveal implicit assumptions. We assume n is a natural number).

Theorem 3.1 For all $x, n \in \mathbb{N}$, a call to $\mathit{Exp}(x, n)$ returns x^n

Note: the procedure *Exp* above can be thought of as (i) a computational process or (ii) a recursive function definition (defined as the output of the computational process). We need to prove that it defines the *same* function as x^n
We are actually claiming two distinct things in the theorem:

- The algorithm returns something (ie, doesn't run forever)
- The algorithm returns the *right* something.

Proof. By induction on n. The base case, that $Exp(x,0)$ returns $x^0 = 1$, follows by inspection: on that call, the "if" condition is satisfied, so we return 1. For the inductive step, assume $Exp(x,n)$ returns x^n and prove for $n+1$. Consider the call to $Exp(x,n+1)$. $n+1>0$, so the "else" clause is invoked. We call $Exp(x,n)$ which, by assumption, returns x^n. We multiply that quantity by x, yielding x^{n+1}, and return it. This proves that $Exp(x,n+1)$ returns the right value. ■

OK, we've shown the procedure returns the right answer.
Is it a good implementation?
How many multiplications are done to compute x^n? By induction, there are n multiplications.
Remember those "128 bits keys" we aren't allowed to export? That means the number n has about 128 bits.
So we will be doing about 2^{128} multiplications.
Still be working on it when 6.042 finishes...

3.2 Fast Exponentiation

Let's cut down on the number of multiplies.
Suppose we want to compute x^a and a is even, say $2c$.
So we want to compute $x^a = x^{2c}$
Any obvious way to beat a multiplies?
Note $x^{2c} = (x^c)^2$
So compute x^c (c multiplies) and then square it (1 multiply).
Total: $c+1 = 1 + a/2$ multiplies.
What if a is odd?

Generalize this:

Procedure $FastExp(x,n)$

if $n = 0$
  return 1
else
  if n is even
    let $y = FastExp(x, n/2)$
    return y^2
  else
    let $y = FastExp(x, (n-1)/2)$
    return $x * y^2$

Theorem 3.2 For all $x, n \in \mathbb{N}$, a call to $FastExp(x,n)$ returns x^n

Proof. Induction again. Base case ($n = 0$) follows by inspection as before. So let's prove the inductive step.
What kinds of subproblems are we going to use to prove it works for n? **not** just $n-1$, so let's use strong induction.
Suppose the theorem is true for every $k < n$, let's prove it for n

Suppose we call $FastExp(x, n)$. We can assume $n \neq 0$ (we already did a proof for 0) so the **else** clause is selected in the algorithm.
Two cases.
If n is even (say $n = 2m$) then we call $FastExp(x, m)$. By strong induction this returns x^m, which we assign to y.
Then we return $y = (x^m)^2 = x^n$ as required, so the proof works for this case.
If n is odd (say $n = 2m + 1$) then we call $FastExp(x, m)$ and set y to the result. Then we return $x * y^2 = x * (x^m)^2 = x^{2m+1} = x^n$
So the proof works in this case too.
These are all the cases. QED. ■

How many multiplications does **this** algorithm do?

Theorem 3.3 If n has b bits, then $FastExp(x, n)$ performs at most $2b$ multiplications.

Proof. Induction on b.
Base case: If $b = 0$, then $n = 0$ and we return without doing any multiplications. So the base case is true. Inductive step: assuming it true for b bits, consider a $b + 1$-bit number n.
We execute a recursive call to *FastExp* on either $n/2$ or $(n - 1)/2$, depending whether n is odd or even.
Both of these numbers have at most b bits.
So by induction, the recursion does at most $2b$ multiplications.
Then we do either one more (even case) or two more (odd case). So at most two more.
Overall, at most $2b + 2 = 2(b + 1)$ multiplications. QED. ■

Using this algorithm, a 128-bit key means only 256 multiplications: quite an improvement!

3.3 Greatest Common Divisor

Given 2 positive natural numbers, $a, b \in N^+$, their *greatest common divisor* is the largest number that divides both without remainder.

(Recall notation: $a \mid b$ means a divides b.)

Finding GCD is a problem that must be solved to generate the "secret" used to decrypt RSA messages.

One of the first recorded algorithms—Euclid's algorithm—finds greatest common divisors. Key lemma for "simplifying" problem:

Lemma 3.4 Given $x > y$, let $r = x - y$. Then any common divisor of x and y is also a common divisor of y and r, and vice versa.

Corollary 3.5 GCD(x, y) =GCD(y, r)

Proof. (of the corollary). (x, y) and (y, r) have exactly the same set of divisors (by lemma). Thus, the biggest number in each set, namely the greatest common divisor, is the same. ■

Proof. (of the lemma). Suppose $d \mid x$ and $d \mid y$. So $x = zd$ and $y = wd$. Thus $r = x - y = d(z - w)$ so $d \mid r$, meaning d is a common divisor of y (by assumption) and r (by deduction). The other direction of proof is similar. ■

From this lemma we get Euclid's algorithm:

Procedure *Euclid*(x, y)

if $x = 0$ return y
else if $y = 0$ return x
else if $x > y$ return *Euclid*$(y, x - y)$
else return *Euclid*$(x, y - x))$

Example:
Find $GCD(112, 84) = ?$
Step 1: $112 - 84 = 28$, reduce to $(28, 84)$.
Step 2: $84 - 28 = 56$, reduce to $(28, 56)$.
Step 3: $56 - 28 = 28$, reduce to $(28, 28)$.
Step 4: $28 - 28 = 0$, reduce to $28, 0$
Now done, answer is 28.

Let's prove this algorithm does the right thing.
First faint worry: maybe we've got an infinite recursion?
First thing to prove for any algorithm: it *terminates*

Lemma 3.6 For any natural number inputs x and y, *Euclid*(x, y) returns a value.

Why do we think this is true intuitively? Because the arguments "shrink" on each recursive call and eventually reach 0. Induction lets us formalize this.

Proof. Induction on $x + y$.
More formally, we prove by strong induction on n that for all inputs x and y such that $x + y = n$, the algorithm terminates.
Base case: if $x + y = 0$, then both x and y are 0 so we return immediately.
Inductive step. Assume the claim for $x + y < n$ and prove it for $x + y = n$.
Now consider two cases. Either $x \geq y$ or $y \geq x$. Suppose first that $x \geq y$.
So consider two subcases. If $y = 0$, then we return immediately and the inductive step is proved.
If $y > 0$, then we make a recursive call with arguments y and $x - y < x - 0 = x$.
So the sum of the two new arguments is less than $x + y = n$
So the inductive hypothesis holds: the recursive call returns a value.
So our original call returns (the same) value.

What about the second case ($y \geq x$)? Well, I would have to write the same proof all over again, swapping x and y everywhere. To avoid such tedium, we say "without loss of generality, assume $x \geq y$. This is code for "the other case works exactly the same way" and completes the proof. ■

Lemma 3.7 For any natural number inputs x and y, *Euclid*(x, y) returns $\mathrm{GCD}(x, y)$

Proof. Induction on $x + y$ as before. The base case is trivial (assuming we define $\mathrm{GCD}(0, 0)$ to be 0, which seems fair).

For the inductive step where $x + y = n > 0$, assume without loss of generality that $x \geq y$. Then we return the result of a recursive call on *Euclid*$(y, x - y)$. This means (by strong induction) that we return GCD$(y, x-y)$. But we proved a lemma above that GCD(x, y) =GCD$(y, x-y)$ so we are returning the right value. ∎

Wait a minute, what's with the GCD$(0, 0)$ definition? Well, technically every number is a common divisor of 0 and 0, so it should be infinite?

Why doesn't this matter? Because we never actually make a recursive call to $(0, 0)$—one number is 0, we terminate.

So even though we still need a base case for the induction pattern, we never actually apply that base case. Instead our base case is implicit in the "subcase" of the inductive step where one argument is 0.

4 Sets

Overview of set theory, including relations on sets.
Probably much of the first part of this will be review from high school math.

We will define an interesting kind of relation called an *equivalence relation*.

An important example of equivalence relation is number equivalence mod m, where m is a positive integer. We will define this relation and give some of its properties.
We will conclude with an application of modular equivalence ideas to get fast RSA encryption and decryption.

4.1 Definitions

A *set* is a collection of objects.
(Collection, object are undefined terms.)

The order of elements is not important. I.e., the sets $\{1, 2\}$ and $\{2, 1\}$ are considered the same set. Each element is included only once. So, if we write $\{a, b\}$ or $\{a, a, b\}$ they both denote the same set and the element a is considered to be present in the set only once. (repeating an element in the description of a set may be useful sometimes).
Examples of sets:

- $\{1, 2, 3\}$.

  A simple finite set, obtained by enumerating all the elements.
- $\emptyset$, the *empty set*.

  A special case of a finite set, with *no* elements.
- $\{\emptyset\}$.

  This funny set is different from $\emptyset$.
  It has one element, which is a set.
- $\{1, \{1, 2\}\}$.

  This one contains two elements, one of which is a set itself.
  The set happens to contain the other element 1, but that does not contradict the condition that an element is included only once.

- $\{n \in \mathbb{N} \mid n \text{ is even }\}$.

  An infinite set, specified by giving a universal set (domain of discourse) and a property. In general, given a domain of discourse D and a predicate P over D, $\{x \in D \mid P(x)\}$ denotes the set of all x in D that satisfy predicate P.

We write that $a \in A$ ("a is in A" or "a is an element of A") if a is an object in the set A.

The empty set satisfies the proposition $(\forall x) x \notin \emptyset$.

Sets can be defined by axioms and deduction rules, giving a very formal *axiomatic set theory*.
But usually, people reason about sets less formally, using *naive set theory*.
Either way, set theory can be used as the basis for defining almost everything else in mathematics and is therefore regarded as the most fundamental part of math.

E.g., natural numbers can be defined as certain kinds of sets (!). For example we can represent the number 0 by the empty set $\emptyset$, number 1 by the set $\{\emptyset\}$, number 2 by $\{\emptyset, \{\emptyset\}\}$, and so on. In general, if S_n is the set representing the number n, the successor of n can be represented by the set $S_{n+1} = \{\emptyset\} \cup \{\{x\} \mid x \in S_n\}$. Then the properties natural numbers can be *proved as theorems* of set theory, rather than being assumed.

Some typical axioms of axiomatic set theory:
(Seem ridiculously obvious but aren't)

- The *axiom of extensionality* says that two sets are equal iff they have the same elements.

  E.g., the two sets $\{n \in \mathbb{N} \mid 2 \text{ divides } n\}$ and $\{2n \mid n \in \mathbb{N}\}$
  are defined in different ways.
  But they are "equal" as sets because they have exactly the same elements.

- The *axiom of comprehension* says that we can form a new set by including all objects of a given set that share a common property.

  For example, $\{x \in \mathbb{R} \mid P(x)\}$ where $P(x) =$ "x is rational" denotes the set of all rational numbers.
  The axioms says that this is a set.

The set building notation that we used above has given the universe of discourse explicitly.
Sometimes, when this seems obvious, we will leave this out.
Just like what we did for predicate quantifiers.
E.g., we might write: $\{n \mid n \text{ is even }\}$ if the universe $\mathbb{N}$ seems clear.
(Could be confusing – the universe could alternatively be the integers, positive and negative.)

Definition 4.1 We say that $A \subseteq B$ ("A is a subset of B" or "A is contained in B") if every element of A is also an element of B (that is, $(\forall x)(x \in A \implies x \in B)$).

Definition 4.2 We say that A is a *proper* subset of B ($A \subset B$) if $A \subseteq B$ and $A \neq B$.

Note $(\forall X)\emptyset \subseteq X$, i.e. the empty set is contained in any other set.

Q: Can we have both $A \in B$ and $A \subseteq B$?

A: Yes, consider $\{1, \{1\}\}$.

Theorem 4.3 $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

Proof by axiom of extensionality. Ignore here.

This observation is very important as it provides a simple way to prove that 2 sets are equal.

Example 4.4 Prove that the two sets $\{n \in \mathbb{N} \mid 2 \text{ divides } n\}$ and $\{2n \mid n \in \mathbb{N}\}$ are equal.

Must show containment in both ways.
$\subseteq$:
We assume $x \in \mathbb{N}$ and 2 divides x, and show that there is some $y \in \mathbb{N}$ such that $x = 2y$.
This is true by definition of divisibility.
$\supseteq$:
Assume $x = 2y$. Then x is divisible by 2, again by definition of divisibility.

Example 4.5 Slightly more interestingly:

The set of evens ≥ 2, i.e., $\{2n \mid n \in \mathbb{N}, n \geq 1\}$,
is equal to $\{x \in \mathbb{N} \mid \exists y, z \text{ odd }, x = y + z\}$.
$\subseteq$:
If $x = 2n$, then write $x = 1 + (x - 1)$, sum of two odds.
$\supseteq$:
If $x = y + z$, where both y and z are odd, then write $y = 2a + 1$, $z = 2b + 1$.
Then $x = 2(a + b + 1)$, which is even.

The size of a set A (the number of elements it contains) is written $|A|$.
For example, the set $\{a, b, c\}$ has size $|\{a, b, c\}| = 3$. The empty set has size $|\emptyset| = 0$.
The size of a finite set is a natural number.
If A is an infinite set, we write $|A| = \infty$.

Notice that for any two finite sets A, B, we have $|A \cup B| \leq |A| + |B|$ and equality holds if and only if $A \cap B = \emptyset$. If we allow A and B to be infinite sets, we still have $|A \cup B| \leq |A| + |B|$ if we define $x + \infty = \infty + x = \infty$ and $x \leq \infty$ for all $x \in \mathbb{N}$ and $x = \infty$.

4.2 Russell's paradox

We might think that set theory is obvious, but this is not the case.
Consider the set $S = \{x \mid x \text{ is a set}\}$.
This seems like a perfectly reasonable set (well, maybe a bit strange—note $S \in S$).
If our set theory allows for such an entity, we get into trouble as follows.

Consider the set $Q = \{x \in S \mid x \text{ is a set and } x \notin x\}$.

If S is a set, then so is Q by standard axioms like comprehension.
Note $Q \subseteq S$.

Then some sets are in Q and some aren't.

Example of something in Q:
The set $A = \{1, 2, 3\}$ is not an element of itself, so $A \in Q$.

Example of something that is not in Q:
The set $P = \{$ all infinite sets$\}$ is a set (by comprehension from S).
Therefore, $P \in P$ as the set P itself is infinite.
So $P \notin Q$.

The troublesome question is: is $Q \in Q$?

1. If $Q \in Q$ then $Q \notin Q$, a contradiction.
2. If $Q \notin Q$ then $Q \in Q$ also a contradiction.

This is clearly a paradox.
In fact, this paradox, at the time it was discovered by Russell ($\sim$ 1900), was a blow to the then current attempts to formalize all of mathematics axiomatically.

Mathematicians had to work very hard to find a consistent set of axioms (note that a consistent set of axioms does not allow this paradox to arise).

The problem with naive set theory is that you can form a set by describing its members, as we did for Q above.
Axiomatic set theory has a specific set of rules for how you can form sets—rules that don't allow the set Q above to be specified, and therefore avoid the paradox.

Fortunately, we'll be working naively...

4.3 Basic set operations

There are several standard operations on sets.
In axiomatic set theory, the claims that these operations yield new sets are axioms.

1. Union

   $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.

2. Intersection

   $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$.

3. Difference

   $A - B = \{x \mid x \in A \text{ and } x \notin B\}$.

   Also written as $A \setminus B$.

4. Complement

   $A^c = \{x \mid x \notin A\}$.

   Also written as $\bar{A}$.

5. Symmetric difference

   $A \oplus B = A \cup B - A \cap B$.

Sets and operations on sets can be represented using Venn diagrams (see textbook, sections 1.4, 1.5).

Union and intersection can be extended to apply to more than two sets.
Given a finite collection of sets A_i, $i = 1, \ldots, n$,

$$\bigcup_{i=1}^{n} A_i$$

is the set of elements in at least one of the A_i, and

$$\bigcap_{i=1}^{n} A_i$$

is the set of elements in all of the A_i. Both of these make sense if $i = 1$ – reduce to just A_1.
In fact, for $i = 0$, default says union of no sets is $\emptyset$ and

intersection of no sets is universal set.
This definition of union and intersection of no sets might seem arbitrary, but in fact it can be motivated in several ways. For example, if we want to define the union of a finite family of sets $A_1, A_2, \ldots, A_n$, we can define $\bigcup_{i=1}^{1} A_1 = A_1$ and $\bigcup_{i=1}^{n+1} A_i = (\bigcup_{i=1}^{n} A_i) \cup A_{n+1}$. If we want this recursive definition to work also for base case $n = 0$ we must set $\bigcup \emptyset = \bigcup_{i=1}^{0} A_i = \emptyset$. Analogously, for intersection. The recursive definition of $\bigcap_{i=1}^{n+1} A_i = (\bigcap_{i=1}^{n} A_i) \cap A_{n+1}$ must have base case $\bigcap \emptyset = \bigcap_{i=1}^{0} A_i = U$ if we want $\bigcap\{A_1\} = (\bigcap \emptyset) \cap A_1 = A_1$.

Instead of using explicit index, can just consider a set of sets: If $\mathcal{S}$ is a *set of sets*, then

$$\bigcup \mathcal{S}$$

denotes the union of all the sets in $\mathcal{S}$, in any order.
(Can prove that the order doesn't matter, will do below.)
Likewise,

$$\bigcap \mathcal{S}$$

denotes the intersection of all the sets in $\mathcal{S}$.

Complement is different from all the other operations, because it assumes some universe U from which we extract the things not in A.
All sets in question are supposed to be subsets of U.
Given a universal set U, the complement operation can be expressed as $A^c = U - A$.
The universe U is often implicit, where there seems to be little chance of confusion.

There is a close correspondence between set operations and logical operations: consider sets A, B and C defined by $A = \{x \mid P(x)\}$, $B = \{x \mid Q(x)\}$, $C = \{x \mid R(x)\}$, where P, Q, R are predicates.

- set union vs. predicate or: $A \cup B = C$ if and only if $P(x) \vee Q(x) \equiv R(x)$
- set intersection vs. predicate and: $A \cap B = C$ if and only if $P(x) \wedge Q(x) \equiv R(x)$
- set difference vs. predicate "non implication": $A - B = C$ if and only if $\neg(P(x) \rightarrow Q(x)) \equiv R(x)$
- set complement vs. predicate not: $A^c = B$ if and only if $\neg P(x) \equiv Q(x)$
- set symmetric difference vs. predicate exclusive or: $A \oplus B = C$ if and only if $P(x) \oplus Q(x) \equiv R(x)$.

Predicates and sets often form alternative ways of looking at the same notions:
Given predicates p and q, define $P = \{x \mid p(x)\}$,
$Q = \{x \mid q(x)\}$.
Then $P \cup Q = \{x \mid p(x) \vee q(x)\}$.
Etc.

4.4 Power set (set of all subsets)

Another very important operation on sets is that of the *power set* of a given set A.
It is denoted by $\mathcal{P}(A)$ and consists of the set of all subsets of A.
In other words, $\mathcal{P}(A) = \{x \mid x \subseteq A\}$.

In particular, for any set A, $\emptyset \in \mathcal{P}(A)$ and $A \in \mathcal{P}(A)$.
Ex:
If $A = \{1, 2\}$ then $\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

Theorem 4.6 If A has n elements, then $\mathcal{P}(A)$ has 2^n elements.

Proof. Ordinary Induction on n. BASE: $n = 0$. Then $\mathcal{P}(A)$ has one element, $\emptyset$.
INDUCTIVE: Assume for n, prove for $n + 1$.
Pick one element, a.
The subsets can be divided into two groups – those containing a and those not containing a.

Moreover, the set of subsets of A not containing a is $\mathcal{P}(A - \{a\})$, and the set of subsets of A containing a is the set $\{\{a\} \cup B \mid B \in \mathcal{P}(A - \{a\})$ which has the same number of elements as $\mathcal{P}(A - \{a\})$. By induction hypothesis $\mathcal{P}(A - \{a\})$ has 2^{n-1} elements, so the total is $2^{n-1} + 2^{n-1} = 2^n$.

■

Q: I did a base case of 0 here. Does the inductive step from 0 to 1 work correctly?

A: Yes, it corresponds to the sets $\{\emptyset\}$ and $\{a\}$.

Because of this cardinality relationship, $\mathcal{P}(A)$ is sometimes denoted by 2^A.

4.5 Set identities

There are a large number of set identities involving relationships among sets built using various operations.
See p. 50 of book.

Here we just list some sample properties, and show how they can be proved.

Theorem 4.7 (Distributivity: Union distributes over intersection)
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

Proof. We show the two sets are equal by proving that each is contained in the other.

1. $A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C)$.

   Suppose $x \in A \cup (B \cap C)$.
   Then by definition, $x \in A$ or $x \in (B \cap C)$.
   We argue by cases.

   If $x \in A$, then by definition $x \in A \cup B$ and similarly $x \in A \cup C$.
   It follows by definition of intersection that x is in the intersection of these two sets.

   On the other hand, if $x \in (B \cap C)$, then $x \in B$ and $x \in C$.
   Then by definition of union, $x \in A \cup B$ and $x \in A \cup C$.
   It again follows that x is in the intersection.

2. $A \cup (B \cap C) \supseteq (A \cup B) \cap (A \cup C)$.

   Suppose $x \in (A \cup B) \cap (A \cup C)$.
   It follows that $x \in A \cup B$ and $x \in A \cup C$.
   We argue by cases.

   If $x \in A$, then certainly $x \in A \cup (B \cap C)$.

   If $x \notin A$, then since $x \in A \cup B$, we know $x \in B$.
   Similarly, $x \in C$.

Therefore, $x \in B \cap C$.
It follows that $x \in A \cup (B \cap C)$.

■

Theorem 4.8 De Morgan's rule: $(A \cup B)^c = A^c \cap B^c$.

Proof. We have to show that $(A \cup B)^c \subseteq A^c \cap B^c$ and that $A^c \cap B^c \subseteq (A \cup B)^c$.

1. $(A \cup B)^c \subseteq A^c \cap B^c$.

   Let $x \in (A \cup B)^c$. Then by definition of complement, $x \notin A \cup B$. It follows (proof by contradiction) that $x \notin A$ and $x \notin B$. Therefore, again by definition of complement, $x \in A^c$ and $x \in B^c$.
   Then by definition of intersection, $x \in A^c \cap B^c$.
2. $A^c \cap B^c \subseteq (A \cup B)^c$.

   Let $x \in A^c \cap B^c$.
   Then x is in A^c and in B^c.
   Therefore, x is in neither A nor B and therefore not in $A \cup B$.
   We conclude that $x \in (A \cup B)^c$.

■

The above two theorems have *dual theorems* produced by swapping $\cap$ and $\cup$.
This is a general phenomenon – look at p. 50, arranged in pairs.
Duality can be applied to any theorem that is true for *all* sets. Each theorem in a pair can be obtained from the other by swapping the operations $\cap$ and $\cup$, and the sets $\emptyset$ and U. Notice that complement operations are not changed.

Theorem 4.9 $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Theorem 4.10 $(A \cap B)^c = A^c \cup B^c$.

4.6 Cartesian products

Another important set operation is the *Cartesian product*.
For sets A and B define $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$.
Here, (a, b) is considered an *ordered* pair so that $(a, b) \neq (b, a)$.

Example 4.11 If $A = \{0, 1\}$ and $B = \{x, y\}$ then $A \times B = \{(0, x), (0, y), (1, x), (1, y)\}$.

Order is important!
In the above example, $(a, 0) \notin A \times B$, though $(a, 0) \in B \times A$.

We can generalize Cartesian products to apply to more than 2 sets and define $A_1 \times \cdots \times A_n = \prod_{i=1}^{n} A_i = \{(a_1, \cdots, a_n) \mid a_i \in A_i \text{ for } 1 \leq i \leq n\}$. Also, we write A^n for $\prod_{i=1}^{n} A$.
Examples:

- $\{0, 1\}^n$ is the set of all n-tuples of 0's and 1's. Or, length n strings of 0's and 1's.
- $\Re^2 = \Re \times \Re$ is the real plane.