CS 70 Discrete Mathematics for CS
Spring 2008 David Wagner
Note 6

Well Ordering Principle

How can the induction axiom fail to be true? Recall that the axiom says the following:

$$[P(0) \wedge (\forall n \,.\, P(n) \implies P(n+1))] \implies \forall n \,.\, P(n).$$

What would it take for $\forall n \in \mathbb{N} \,.P(n)$ to be false, i.e., for $\neg(\forall n \in \mathbb{N} \,.P(n))$ to be true? This would mean that $(\exists n)(P(n)$ is false$)$. In other words, at least one of the propositions $P(0)$, $P(1)$, $\ldots$, $P(n-1)$, $P(n)$ must be false. Let m be the smallest integer for which $P(m)$ is false. In other words, $P(m-1)$ is true and $P(m)$ is false. But this directly contradicts the fact that $P(m-1) \implies P(m)$.

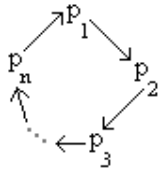
It may seem as though we just proved the induction axiom. Not quite. Instead, what we have actually done is to show that the induction axiom follows from another axiom, which was used implicitly in defining "the smallest m for which $P(m)$ is false."

Well ordering principle: If $S \subseteq \mathbb{N}$ and $S \neq \emptyset$, then S has a smallest element.

We assumed something when defining m that is usually taken for granted: that we can actually find a smallest element of the set. This cannot always be accomplished, and to see why consider the set $\{x \in \mathbb{R} : 0 < x < 1\}$. Whatever number is claimed to be the smallest of the set, we can always find a smaller one. This shows that the real real numbers are not well ordered: if we consider a set $S \subseteq \mathbb{R}$ and $S \neq \emptyset$, then S may or may not have a smallest element. In other words, every non-empty set of natural numbers has a smallest element, but not every non-empty set of real numbers has a smallest element.

Again, the well ordering principle may seem obvious but it should not be taken for granted. It is only because the natural numbers (and any subset of the natural numbers) are well ordered that we can find a smallest element. Not only does the principle underlie the induction axioms, but it also has direct uses in its own right. Here is a simple example.

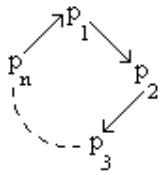
Round robin tournament: Suppose that, in a round robin tournament, we have a set of n players $\{p_1, p_2, \ldots, p_n\}$ such that p_1 beats p_2, p_2 beats p_3, $\ldots$, and p_n beats p_1. This is called a *cycle* in the tournament:



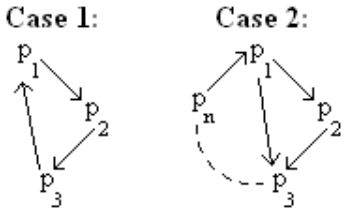
(A round robin tournament is a tournament where each contestant plays every other contestant exactly once. Thus, if there are n players, there will be exactly $n(n-1)/2$ matches. Also, we are assuming that every match ends in either a win or a loss; no ties.)

Claim: If there exists a cycle in a tournament, then there exists a cycle of length 3.

Proof: Assume for a contradiction that the smallest cycle is:



with $n > 3$. Let us look at the game between p_1 and p_3. We have two cases: either p_3 beats p_1, or p_1 beats p_3. In the first case (where p_3 beats p_1), then we are done because we have a 3-cycle. In the second case (where p_1 beats p_3), we have a shorter cycle, and thus a contradiction. Therefore, if there exists a cycle, then there must exist a 3-cycle as well. □



Here is another example of the well ordering principle in action:

Theorem: There are no solutions to $x^3 + 2y^3 = 4z^3$ in the natural numbers.

Proof: We first show that if there is any solution to this equation in the natural numbers, we can find another "smaller" solution in the natural numbers. Suppose (a,b,c) is a solution to this equation in the natural numbers, so that $a,b,c \in \mathbb{N}$ and $a^3 + 2b^3 = 4c^3$. Re-arranging terms, we have $a^3 = 2(2c^3 - b^3)$. Since $2c^3 - b^3$ is an integer, a^3 must be even. It follows that a must be even (if a were odd, then a^3 would have to be odd, too), so let $A = a/2$. Since a is even, A must be a natural number. Plugging $a = 2A$ into the original equation, we get

$$(2A)^3 + 2b^3 = 4c^3.$$

Each of these terms is divisible by two, so we divide both sides by two, which yields

$$4A^3 + b^3 = 2c^3.$$

The same reasoning mentioned earlier shows that b^3 must be even, so b must be even, too. Let $B = b/2$, noticing that we have $B \in \mathbb{N}$. Using the fact that $b = 2B$, we get

$$4A^3 + (2B)^3 = 2c^3.$$

Dividing both sides by two yields

$$2A^3 + 4B^3 = c^3.$$

By similar reasoning, c^3 must be even, so c must be even, too, and we can define $C = c/2$. Since c is even, we have $C \in \mathbb{N}$. Plugging in $c = 2C$, we get

$$2A^3 + 4B^3 = (2C)^3.$$

Dividing both sides by two yields

$$A^3 + 2B^3 = 4C^3.$$

In other words, (A,B,C) is also a solution to the original equation. In short, if (a,b,c) is a solution in the natural numbers, so too is (A,B,C), and moreover (since A was defined to be $a/2$) we have $A < a$.

Next let S be the set of values of a that can appear among the solutions to this equation in the natural numbers, i.e.,

$$S = \{a \in \mathbb{N} : \exists b,c \in \mathbb{N} \,.\, a^3 + 2b^3 = 4c^3\}.$$

We have proven that S has no smallest element: for any element $a \in S$, we can find another natural number A such that $A \in S$ and $A < a$. Also, by definition, $S \subseteq \mathbb{N}$. The well ordering principle says that any non-empty subset of $\mathbb{N}$ must have a smallest element. Since S is a subset of $\mathbb{N}$ with no smallest element, the only possible conclusion is that S must be empty. This means that the equation $x^3 + 2y^3 = 4z^3$ has no solutions in the natural numbers. □

Induction and Recursion

There is an intimate connection between induction and recursion in mathematics and computer science. A recursive definition of a function over the natural numbers specifies the value of the function at small values of n, and defines the value of $f(n)$ for a general n in terms of the value of $f(m)$ for $m < n$. Let us consider the example of the Fibonacci numbers, defined in a puzzle by Fibonacci (in the year 1202).

Fibonacci's puzzle: A certain man put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive?

Let $F(n)$ denote the number of pairs of rabbits in month n. According to the above specification, the initial conditions are $F(0) = 0$ and, when the pair of rabbits is introduced, $F(1) = 1$. Also $F(2) = 1$, since the pair is not yet productive. In month 3, according to the conditions, the pair of rabbits begets a new pair. So $F(3) = 2$. In the fourth month, this new pair is not yet productive, but the original pair is, so $F(4) = 3$. What about $F(n)$ for a general value of n? This is a little tricky to figure out unless you look at it the right way. The number of pairs in month $n-1$ is $F(n-1)$. Of these how many were productive? Only those that were alive in the previous month—i.e., $F(n-2)$ of them. Thus there are $F(n-2)$ new pairs in the n-th month, and so $F(n) = F(n-1) + F(n-2)$. This completes the recursive definition of $F(n)$:

- $F(0) = 0$, and $F(1) = 1$.
- For $n \geq 2$, $F(n) = F(n-1) + F(n-2)$.

This admittedly simple model of population growth nevertheless illustrates a fundamental principle. Left unchecked, populations grow exponentially over time. [Exercise: can you show, for example, that $F(n) \geq 2^{(n-1)/2}$ for all $n \geq 3$?] Understanding the significance of this unchecked exponential population growth was a key step that led Darwin to formulate his theory of evolution. To quote Darwin: "There is no exception to the rule that every organic being increases at so high a rate, that if not destroyed, the earth would soon be covered by the progeny of a single pair."

Be sure you understand that a recursive definition is not circular—even though in the above example $F(n)$ is defined in terms of the function F, there is a clear ordering which makes everything well-defined. Here is a recursive program to evaluate $F(n)$:

```
function Fib(n):
  if n=0 then return 0
```

```
  if n=1 then return 1
  else return Fib(n-1) + Fib(n-2)
```

Is execution of `Fib(n)` is guaranteed to terminate within some finite number of steps, for every natural number `n`? The answer is yes. If we let S denote the set of numbers passed to some recursive invocation of `Fib`, when we start by calling `Fib(n)`, then it is easy to see that S is non-empty (since it includes `n`) and that $S \subseteq \mathbb{N}$ (since every argument to `Fib` is a natural number). Consequently, by the well ordering principle S must have some smallest element—which means that the recursion must terminate at some point. Since this program executes only finitely many steps of computation during each recursive invocation of `Fib`, the total running time must be finite. In general, the well ordering principle can often help show that a recursive program is guaranteed to terminate.

Even though this program does terminate eventually, it is a very inefficient way to compute the n-th Fibonacci number. (Can you figure out how long this program takes to compute $F(n)$?) A much faster way is to turn this into an iterative algorithm (this should be a familiar example of turning a tail-recursion into an iterative algorithm):

```
function Fib2(n):
  if n=0 then return 0
  if n=1 then return 1
  a = 1
  b = 1
  for k = 2 to n-1 do
    t = a
    a = a + b
    b = t
  return a
```

How would we prove that this program correctly computes the Fibonacci numbers, i.e., that `Fib2(n)` $= F(n)$ for all $n \in \mathbb{N}$? It turns out there is an elegant way to do so. We point out that, just before any iteration of the loop, the variable `a` holds the value $F(k)$ and the variable `b` holds the value $F(k-1)$. This is, in fact, an invariant.

How would we prove that the invariant is valid? First, we notice that the invariant is true before the first iteration of the loop, since at that point we have `a=1`, `b=1`, and `k=2`. Next, we show that if the invariant is true just before some iteration of the loop, then it will be true just before the next iteration. In particular, suppose that `a` and `b` hold the values $F(k)$ and $F(k-1)$, respectively, before the k-th iteration of the loop. Then the loop body sets `a` to $F(k)+F(k-1) = F(k+1)$ and `b` to $F(k)$, which means that the invariant will be true before the $k+1$-th iteration of the loop.

This notion of a *loop invariant* is so useful in programming that good programmers will often document loop invariants in their code, like this:

```
function Fib2(n):
  if n=0 then return 0
  if n=1 then return 1
  a = 1
  b = 1
  for k = 2 to n-1 do
    /* At this point we have a=F(k) and b=F(k-1).  */
```

```
    t = a
    a = a + b
    b = t
  return a
```