How to use strong induction to prove correctness of recursive algorithms

April 12, 2015

1 Format of an induction proof

Remember that the principle of induction says that if $p(a) \wedge \forall k[p(k) \rightarrow p(k+1)]$, then $\forall k \in \mathbf{Z}, n \geq a \rightarrow p(k)$. Here, $p(k)$ can be any statement about the natural number k that could be either true or false. It could be a numerical formula, such as "The sum of the first k odd numbers is k^2" or a statement about a process: "After the first k passes of BubbleSort, the last k positions contain the k largest elements".

In other words, if a process evolves in discrete steps, it starts out having a property, and in no one step does the property change from true to false, then it will always have the property after any number of steps.

Actually, though, if $k+1$ were the *first* integer where $p(k+1)$ failed to be true, we would know something a lot stronger than just $p(k)$ but not $p(k+1)$. It would also be true that $p(0), p(1), ..p(k-1), p(k)$ all were true, but still $p(k+1)$ would be false. So it suffices to show that this cannot occur to show that $p(k)$ always holds.

In other words, if $p(a)$, and if for all $k \geq a$, $p(a) \wedge p(a+1) ... \wedge p(k) \rightarrow p(k+1)$. then $p(k)$ holds for every integer $k \geq 0$. Strong induction (as this is called) is more complicated, but actually easier to use than plain induction, because the induction hypothesis we're allowed is much stronger, which makes it easier to prove the implication.

Thus the format of an induction proof:

- Part 1: We prove a *base case*, $p(a)$. This is usually easy, but it is essential for a correct argument.
- Part 2: We prove the *induction step*. In the induction step, we prove $\forall k[(\forall a \leq k' \leq kp(k')) \rightarrow p(k+1)]$.

  Since we need to prove this universal statement, we are proving it for an abstract variable k, not for a particular value of k. Thus, we let k be an arbitrary non-negative integer, and our sub-goal becomes: $p(k) \rightarrow p(k+1)$.

To prove such an implication, we assume $\forall a \leq k' \leq k, p(k')$, and and our sub-goal is now $p(k+1)$. The assumption $\forall a \leq k' \leq k, p(k')$ is called the *strong induction hypothesis*. Note that it includes $k' = k$, so $p(k)$ is a special case. That means that any proof by induction is also a proof by strong induction (although not vice versa). While you're getting used to doing proofs by induction, it's a good habit to explicitly state and label both the induction hypothesis and the intended goal, $p(k+1)$. Once we get used to induction, we merge steps.

Sometimes it is easier to do a change of variables and call $k+1$ "k" which makes k "$k-1$". Then the strong induction hypothesis becomes : $\forall a \leq k' \leq k-1, p(k')$ and in the induction step we are proving $p(k)$. These are logically equivalent, and it is a matter of preference which you like better for a particular problem. As long as you are consistent and clear in which version you are using, either is correct.

- Part 3: State what induction then allows us to conclude: "Since we have shown that the property (equation , inequality, relationship, predicate as appropriate) is true for $k = a$ in the base case, and since we have shown in the induction step that if the property is true for all $a \leq k' \leq k$ then it is also true for $k+1$, by the principle of induction we have shown that the property is true for all integers $k \geq a$."

2 Recursive algorithms

Strong induction is the method of choice for analyzing properties of recursive algorithms. This is because the strong induction hypothesis will essentially tell us that all recursive calls are correct. Don't try to mentally unravel the recursive algorithm beyond one level of recursive calls. Strong induction allows us just to think about one level of recursion at a time. The reason we use strong induction is that there might be many sizes of recursive calls on an input of size k. But if all recursive calls shrink the size or value of the input by exactly one, you can use plain induction instead (although strong induction is still okay, too.)

Here's the generic form of a recursive algorithm:

Base case of recursion For small inputs, or some other special cases, the algorithm will give an answer directly, without making recursive calls.

Defining sub-problems If a base case doesn't apply, the algorithm defines one or more "sub-problems", smaller inputs computed from the main input.

Solving sub-problems The recursive algorithm calls itself on the sub-problems and saves the recursively computed answer.

Produce Output After solving all the sub-problems, the algorithm then produces some output based on the input and answers to the sub-problems.

There are many things we want to know about such algorithms. How does the output depend on the input, and why does it meet the correctness property in the problem specification? Does the recursion always terminate? How much total time does the algorithm take?

All of these are about *global behavior* of the algorithm, properties of the entire sequence of recursive calls (since each recursive call might itself make recursive calls, this can be quite complex). But what the recursive structure gives us is *local behavior*, what happens at the main recursive call. By using a strong induction argument, we only need to worry about the level of recursion that is explicitly given by the algorithm. We let the other levels worry about themselves, because we've handled them "inductively" using the same method. In particular, we use strong induction so that we *do not need to unravel* the recursion. Note that unlike for loop invariants, the induction variable is the *input size*, not the time step or level of the recursion.

3 An example algorithm

So far we've been very abstract. Let's translate this to a specific example. Here's a very simple algorithm that computes the floor of the log of x. Here, the command $x \div y$ returns the floor of x/y.

RLogRounded (x: positive integer): non-negative integer;

1. IF $x = 1$ return 0
2. $L \leftarrow RLogRounded(x \div 2)$.
3. Return $L + 1$.

We want to show that for $x \geq 1$, the program outputs $log_2 x$ round down to the nearest integer, $RLogRounded(x) = \lfloor \log_2 x \rfloor$. We do this by *strong induction*.

Base case: When $x = 1$, $RLogRounded(1) = 0 = \lfloor 0 \rfloor = \lfloor \log 1 \rfloor = \lfloor \log x \rfloor$.

Strong induction step: Assume $RLogRounded(x') = \lfloor \log_2 x' \rfloor$ for all $1 \leq x' \leq x - 1$, for some $x \geq 2$.

We will show $RLogRounded(x) = \lfloor \log_2 x \rfloor$.

Since $x > 1$, $RLogRounded(x) = RLogRounded(x \div 2) + 1$ (from lines 2 and 3).

If x is even, this is $RLogRounded(x/2) + 1$.

Since $1 \leq x/2 < x$, we can apply the strong induction hypothesis with $x' = x/2$ and see $RLogRounded(x) = RLogRounded(x/2) + 1 = \lfloor \log_2(x/2) \rfloor + 1 = \lfloor \log_2 x - 1 \rfloor + 1 = \lfloor \log_2 x \rfloor$.

If x is odd, this is $RLogRounded((x-1)/2)+1$. Note that since x is odd, $x \geq 3$ and x cannot be a power of 2. Since x is not a power of 2, $\lfloor \log_2 x \rfloor = \lfloor \log_2(x - 1) \rfloor$. Since $x \geq 3$, $1 \leq (x - 1)/2 < x$, so we can apply the strong induction

hypothesis with $x' = (x-1)/2$ and see $RLogRounded(x) = RLogRounded((x-1)/2)+1 = \lfloor \log_2(x-1/2) \rfloor + 1 = \lfloor \log_2(x-1) - 1 \rfloor + 1 = \lfloor \log_2(x-1) \rfloor = \lfloor \log_2 x \rfloor$.

Thus, by strong induction on x, $RLogRounded(x) = \lfloor \log_2 x \rfloor$ for all integers $x \geq 1$.

4 General method

Now let's abstract what we did above to see what steps we go through in general.

Stating correctness It is important to state what correctness means to the algorithm carefully. Unlike with loop invariants, this is just making the problem specification precise. It is not at all creative, just a matter of being careful. In the above example, correctness is an equation, but sometimes it is a more complex property, such as producing a sorted list with the same elements as the input, or deciding whether a string is a palindrome.

The base case Recursive algorithms need to eventually bottom out, otherwise, they are creating an endless list of recursive calls. The input sizes where the recursion always bottoms out are those where we prove the statement as a base case. There can be more than one. Whule we only need one base case in a strong induction proof, what this is really doing if we have multiple base cases is dividing up the induction step into cases, ones where the input is small and the rest when the input is large. If you prefer to break up the induction step into cases rather than call them seperate base cases, that is also correct.

Here, the algorithm does not call itself recursively when $x = 1$, just returns 0. So $x = 1$ is the base case of the induction argument.

We need to show that the program is correct on each base case. There are two parts to this, for each such case:

1. Use the algorithm description to say what gets returned in the the base case.
   " When $x = 1$, $RLogRounded(1) = 0$"
2. Show that this value satisfies the correctness property.
   "$0 = \lfloor 0 \rfloor = \lfloor \log 1 \rfloor = \lfloor \log x \rfloor$. "

Strong Induction step In the induction step, we can assume that the algorithm is correct on all smaller inputs. We use this to prove the same thing for the current input.

We do this in the following steps:

1. State the induction hypothesis: The algorithm is correct on all inputs between the base case and one less than the current case. We

also know that the current value is not a base case. The exact formalism that this corresponds to will vary according to the problem specification.

"Assume $RLogRounded(x') = \lfloor \log_2 x' \rfloor$ for all $1 \leq x' \leq x - 1$, for some $x \geq 2$. "

2. Represent what the algorithm returns on the current value in terms of recursive calls.

   "Since $x > 1$, $RLogRounded(x) = RLogRounded(x \div 2) + 1$ (from lines 2 and 3). If x is even, this is $RLogRounded(x/2) + 1$." (Note that we are breaking it up into cases at this point, because the div operation is different for the two cases odd and even. The cases come from cases you need to divide things up into to simulate the steps of the algorithm).

3. Apply the strong induction hypothesis to replace each recursive call with the correct answer.

   " Since $1 \leq x/2 < x$, we can apply the strong induction hypothesis with $x' = x/2$ and see $RLogRounded(x) = RLogRounded(x/2)+1 = \lfloor \log_2(x/2) \rfloor + 1$"

4. Then we need to use algebra and logic to show that this is the same as a correct answer for the current input.

   "$= \lfloor \log_2 x - 1 \rfloor + 1 = \lfloor \log_2 x \rfloor$." (the last expression being what we wanted the algorithm to return.)

5. If we broke up the algorithm into cases, we need to repeat the last three parts for all cases.

6. Summary of induction argument:

   Just a reminder that we are finished:

   "Thus, by strong induction on x, $RLogRounded(x) = \lfloor \log_2 x \rfloor$ for all integers $x \geq 1$."

Strong induction proofs of correctness for recursive algorithms are actually easier and more direct than loop invariants, because the recursive structure is telling us what correctness means at all levels. The statement we are proving is direct from the correctness condition, so doesn't need to be modified in a creative way.