# How to use induction and loop invariants to prove correctness

## 1 Format of an induction proof

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 sorted order."

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.

Thus, to prove some property by induction, it suffices to prove $p(a)$ for some value of $a$ and then to prove the general rule $\forall k[p(k) \rightarrow p(k+1)]$.

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 $\forall n[p(k) \rightarrow p(k+1)]$.

  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 $p(k)$, and our sub-goal is now $p(k+1)$. The assumption $p(k)$ is called the *induction hypothesis*. While you're getting used to doing proofs by induction, it's a good habit to explicitly state and label both the induction hypothesis $p(k)$ and the intended goal, $p(k+1)$. Once we get used to induction, we merge steps : "Let $k$ be any integer so that $p(k)$....blah, blah,..Therefore, $p(k+1)$. Thus we have proved the induction step."
- 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 $k$ 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 Loop invariants

Induction is the method of choice for analyzing properties of algorithms with loops. A typical such algorithm:

**Initialization** The algorithm initializes some variables based on the inputs.

**Loop** Then the loop starts. Each *iteration* of the loop changes variables according to some loop instructions until a *guard condition* fails.

**Produce Output** When the guard condition fails, the algorithm then produces some output based on the values of the variables.

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 loop always terminate? How many iterations might the loop make, and how much total time does it take?

All of these are about *global behavior* of the algorithm, properties of the entire run. But what the algorithm description gives us is *local behavior*, what happens in each step. Loop invariants allow us to bridge this gap.

To understand the global behavior, when all we are given explicitly is the local, we usually need to prove something stronger than what happens at the end. We need to understand what happens in the middle, something about the values of the variables after $t$ steps. Sometimes there's an explicit formula telling us this. But more typically, the art of finding the right loop invariant is to get a property that is simple enough to understand but captures the reason why the algorithm is making progress towards terminating with the correct answer.

So far we've been very abstract. Let's translate this to a specific example. Here's a very simple algorithm that computes the ceiling of the log of $x$.

LogRounded (x: integer): integer

1. $i \leftarrow 0$
2. $y \leftarrow 1$
3. IF $x \leq 0$then Return "error"
4. While $y < x$ do:
5. $\quad i \leftarrow i + 1$
6. $\quad y \leftarrow 2 * y$
7. Return $i$

We want to show that the program outputs $log_2 x$ rounded up to the nearest integer. But to do this we need to prove a stronger claim about the values $y$ and $i$ have throughout the loop. In this case, there is an explicit formula. Each time we go through the loop, $i$ is incremented, and $y$ doubles. So $y$ will always be a power of 2. We will show $y = 2^i$ at all times. Note that this *loop invariant* isn't just a translation of the definition of correctness. But it really is why the algorithm was designed the way it was, the explanation for *why* we are following these steps.

The proof breaks down into three basic parts, based on the three parts of a loop: Initialization, Looping, Returning Output.

**Stating the invariant** In this example, we are claiming that, if the loop executes at least $t$ times, after the $t$'th iteration, $y = 2^i$.

**The base case** Before the loop starts, i.e., after $t = 0$ iterations, $y = 1$ and $i = 0$. Since $2^0 = 1$, the invariant is true at the start.

**Induction step** Let $y_B$ and $i_B$ be the values of $y$ and $i$ at the end of the $t$'th interation (i.e, at the beginning of the $t + 1$'st iteration). B stands for "before" the $t + 1$'st iteration. Let $y_A$ and $i_A$ be the values of $y$ and $i$ at the end of the $t+1$'st interation. A stands for "after" the $t+1$'st iteration.

Assume the loop invariant holds at the end of the $t$'th iteration, that is, that $y_B = 2^{i_B}$ . This is the induction hypothesis.

In that iteration, $y$ is doubled and $i$ is incremented, so the new value of $y$ is $y_A = 2y_B$ and the new value of $i$ is $i_A = i_B + 1$.

Then, from the algorithm, we have

$$\begin{aligned} y_A &= 2y_B \\ &= 2 * 2^{i_B} \\ &= 2^{i_B+1} \\ &= 2^{i_A} \end{aligned}$$

Thus, at the end of the $t + 1$'st iteration, $y = 2^i$ as desired.

**Summary of induction argument** Since the invariant is true after $t = 0$ iterations, and if it is true after $t$ iterations it is also true after $t + 1$ iterations, by induction, it will remain true after any number of iterations until the loop ends.

**Using the invariant to prove correctness** The guard condition is that $y < x$. Consider the last iteration of this loop, and as before let $y_B, i_B$ be the values of $y$ and $i$ before this iteration, and $y_A = 2y_B$ and $i_A = i_B + 1$ be the values after this iteration. Before the last iteration , $y_B < x$ because the guard condition is still true. Afterwards, the new value of $y$, $y_A$ has $y_A \geq x$ because this is the last time through the loop, which means the guard condition is false when it ends. Since $y_A = 2y_B$, $y_B < x \leq 2y_B$. Using the loop invariant, $y_B = 2^{i_B}$. So $2^{i_B} < x \leq 2^{i_B+1}$. Since $i_B+1 = i_A$, this means $2^{i_A-1} < x \leq 2^{i_A}$ . Taking logs, $i_A - 1 < log_2 x < i_A$. Thus, $i_A$ is the closest integer greater than $\log_2 x$. The algorithm then returns $i_A = \lceil \log_2 x \rceil$.

# 3 General method

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

We will use $t$ to represent the number of iterations we have gone through the loop, which is the variable we are doing the induction on. The variable

name doesn't matter, as long as it's not being used for something else in the algorithm.

**Stating the invariant** It is important to state the invariant carefully. This is in some sense the most important part of the induction argument, and the art of algorithm correctness lies in picking the *right* loop invariant. The loop invariant needs to have two properties: it needs to be self-justifying (it's not enough that it's true at all iterations; it's truth at one iteration must logically follow from its truth at the previous iteration), and it must help prove correctness of the algorithm at the end.

The loop invariant is a property $p(t)$ of the form : If we make at least $t$ iterations, after the $t$'th iteration, the values of the variables have some relationship. (Note that this means we don't have to worry about $t$'s larger than the number of iterations; since a false hypothesis implies any conclusion, what we are proving is vacuously true once $t$ goes beyond the actual number of iterations.)

In this example, we are claiming that, if the loop executes at least $t$ times, then after the $t$'th iteration, $y = 2^i$.

Comments on notation:
That last equation is the relationship between variables, but the whole sentence is what we are proving by induction. Even though $t$ is not an explicit part of the relationship, it's the induction variable, and since $y$ and $i$ are dynamic variables changing values through the algorithm, $y$ and $i$ are implicitly functions of $t$. (Actually, we could make them explicit here: $i = t$ and $y = 2^t$, and we could have used those equations as the loop invariant instead.) Since variables used in algorithms are dynamic, changing values, we will use the notation $y_B, y_A, i_B, i_A$ to mean the values of the variables B=Before and A=After an iteration of the loop.

**The base case** The base case of the loop invariant is usually $t = 0$, after 0 times through the loop. That means the relationship needs to hold for the values the variables are *initialized* to. We need two sub-steps here:

1. Use the algorithm description to say what the variables are intialized to.

   In our example:
   "Before the loop starts, i.e., after $t = 0$ iterations, $y = 1$ and $i = 0$. "
2. Show that these values satisfy the relationship.

   In our example:
   "Since $2^0 = 1$, the invariant is true at the start."

**Induction step** In the induction step, we know the invariant holds after $t$ iterations, and want to show it still holds after the next iteration.

We start by stating all the things we know:

1. The invariant holds for the values of the variables at the start of the next iteration. This is the induction hypothesis.

   In our example:
   "Assume the loop invariant holds at the end of the $t$'th iteration, that is, that $y_B = 2^{i_B}$ ."

2. The guard condition is true at the start of the next iteration (or there won't be a next iteration).

   In our example:
   In this case, we didn't need to use it, but we could have said: "At the start, $y_B < x$."

3. The body of the loop tells us how the values of variables get updated.

   In our example:
   "In that iteration, $y$ is doubled and $i$ is incremented, so the new value of $y$ is $y_A = 2y_B$ and the new value of $i$ is $i_A = i_B + 1$."

We need to show that the updated values of the variables have the loop invariant relationship.

We have all of the above facts to work with, now. We can use algebra, logic, or any other method to work towards this conclusion. Be careful that your argument should use the induction hypothesis and the new values of the variables. Otherwise, it is a warning sign that you've forgotten something crucial.

In our example:
"Then, from the algorithm, we have

$$\begin{aligned} y_A &= 2y_B \\ &= 2 * 2^{i_B} \\ &= 2^{i_B+1} \\ &= 2^{i_A} \end{aligned}$$

Thus, at the end of the $t+1$'st iteration, $y = 2^i$ as desired."

**Summary of induction argument** Since the invariant is true after $t = 0$ iterations, and if it is true $t$ iterations it is also true after $t+1$ iterations, by induction, it will remain true after any number of iterations.

This step is just a reminder to ourselves and the reader of what we were doing and why we are done.

**Using the invariant to prove correctness** Now we've proved the loop invariant. But we still need to show that the invariant implies the correctness of the algorithm.

Again, start by stating what you know; let $t$ be the last iteration, where the guard fails.

1. Before the $t$'th iteration, the guard condition was true.

   In our example:
   "The guard condition is that $y < x$. Consider the last iteration of this loop, and as before let $y_B, i_B$ be the values of $y$ and $i$ before this iteration, and $y_A = 2y_B$ and $i_A = i_B + 1$ be the values after this iteration. Before the last iteration , $y_B < x$ because the guard condition is still true."

2. After the $t$'th iteration, the guard condition is false.

   In our example:
   "Afterwards, the new value of $y$, $y_A$ has $y_A \geq x$ because this is the last time through the loop, which means the guard condition is false when it ends."

3. The body of the loop defines the after values in terms of the before values.

   In our example:
   "Since $y_A = 2y_B$, $y_B < x \leq 2y_B$."

4. Both before and after, the loop invariants are true.

   In our example:
   "Using the loop invariant, $y_B = 2^{i_B}$."

   In this case, we return the after value of $i$, i.e, $i_A$.

Now we put the pieces together to say what actually gets returned:

In our example:
"So $2^{i_B} < x \leq 2^{i_B+1}$. Since $i_B + 1 = i_A$, this means $2^{i_A-1} < x \leq 2^{i_A}$ . Taking logs, $i_A - 1 < log_2 x < i_A$. Thus, $i_A$ is the closest integer greater than $\log_2 x$. The algorithm then returns $i_A = \lceil \log_2 x \rceil$.

The part of the algorithm after the loop says what gets returned."

In this case, this exactly matches the problem specification, but in more complex algorithms, you might then have to argue why it meets the correctness condition.