Lecture 6: Induction

This lecture is about a proof technique called induction, which is a way to prove that a statement holds for infinitely many cases, especially if those cases are 'for every natural number'.

If you can do a proof *without* induction, then that is usually simpler: if you want to show that for every natural number n, the value $n^2 + n$ is even then you can do that directly, and you should. But if you want to prove that for any natural number n, the sum $1 + \ldots + n$ is $(n^2 + n)/2$ then induction is the right tool to use.

Induction comes in at least three forms:

1. Induction over the natural numbers, the basic version.
2. Structural induction, a more general version of the same principle that is particularly important for computer science.
3. Invariant induction, a variation for proving properties of code that uses loops.

All of these are based on the same principle, which we will explain later in these notes.

I hope it is obvious that we cannot claim a statement is true for all natural numbers just because we have tried it out for quite a few. For a silly example, the statement '$P(n)$: n is less than 9000' would be true for the first few thousand numbers you tried.

Induction over the Natural Numbers

Legend has it that the mathematician Gauss, as a schoolboy, was asked to add up the numbers from 1 to 100. The teacher thought this would keep the class busy for a while, but Gauss noticed that $1 + 100 = 101$, $2 + 99 = 101$, $3 + 98 = 101$ and so on; the numbers from 1 to 100 fit nicely into 50 such pairs so the sum is $50 \times 101 = 5050$. More generally, the following is known as Gauss' formula:

$$1 + 2 + \ldots + n \quad = \quad n(n+1)/2$$

which we can of course write without dots[1] as

$$\sum_{i=1}^{n} i \quad = \quad n(n+1)/2$$

The idea to prove a statement like this for every natural number is to split the proof into two steps:

Definition 1 (induction over $\mathbb{N}$)**.** To prove that a statement $S(n)$ holds for all natural numbers $n \in \mathbb{N}$ by induction,

1. Prove the base case $S(0)$.
2. Prove the induction step $S(n) \to S(n+1)$, leaving n as a free variable.

If you only want to prove that your statement holds for all natural numbers $n \geq k$ for some k (for example all numbers $n \geq 1$), then you change the base case to showing $S(k)$; if you need to, you can then use $n \geq k$ as an extra assumption in the induction step.

Informally, the argument is then:

[1] The three-dots notation is a shorthand that is tolerated in mathematics as long as it is obvious how you would write the same statement without the dots.

- By 1., $S(0)$ holds.
- By 2., since we know $S(0)$ holds, then $S(1)$ holds.
- By 2., since we know $S(1)$ holds, then $S(2)$ holds.
- ... (this is not mathematically formal, we will come back to this later)
- Therefore, $S(n)$ holds for all $n \in \mathbb{N}$.

If we are proving a statement of the form $L(n) = R(n)$, then we can use a template for the proof.

1. Define $L(n)$ and $R(n)$ and say: 'We prove $L(n) = R(n)$ by induction'.
2. Say 'Base case: $n = 0$' and calculate out $L(0)$ and $R(0)$. Check they are the same.
   (If your statement only holds for all $n \geq k$ for some k (such as $n \geq 1$) then start with k instead.)
3. Say 'Induction step. Assume $L(n) = R(n)$. We want to show $L(n+1) = R(n+1)$.'
   Unlike in the last step, here you leave n as a variable.
4. Write out $L(n+1)$ and start calculating.
5. At some point, when you have a subterm $L(n)$, say 'Using the induction assumption' and replace it with $R(n)$.
6. Continue calculating until you have, hopefully, $R(n+1)$.
7. Say 'This proves the statement by induction.' or words to that effect.

The key to the induction step is that if you are trying to prove an implication of the form

$$a = b \quad \rightarrow \quad c = d$$

then you can start by assuming $a = b$, write down c, calcluate, and whenever you find a term a you can replace it with b (or vice versa) as you are working inside the scope of that assumption. If this gets you to d, then you have proved $c = d$ under the assumption $a = b$, which is what $a = b \rightarrow c = d$ means. This is a general fact about proofs that works even if you are not doing induction.

Proof Examples

Here is a proof of Gauss' formula using the template approach.

Claim. For every natural number $n \geq 1$,we have $\sum_{i=1}^{n} i = n(n+1)/2$.
Proof. Following the template:

1. We prove the statement by induction. Let $L(n) = \sum_{i=1}^{n} i$ and let $R(n) = n(n+1)/2$.
   The claim is $L(n) = R(n)$ for all natural $n \geq 1$.
2. For the base case, $L(1) = \sum_{i=1}^{1} i = 1$ and $R(1) = 1 \times 2/2 = 1$.
   This proves the base case $L(1) = R(1)$.
3. For the induction step, assume that $L(n) = R(n)$. We want to show that $L(n+1) = \sum_{i=1}^{n+1} i$ and $R(n+1) = (n+1)(n+2)/2$ are equal.
4. Start with $L(n+1) = \sum_{i=1}^{n+1} i$ which is the sum of all numbers from 1 to $n+1$, and split off the last term to get $L(n+1) = (\sum_{i=1}^{n} i) + (n+1) = L(n) + (n+1)$.
5. Using the induction assumption, replace $L(n)$ by $R(n)$ to get $L(n+1) = R(n) + (n+1)$. Writing this out, $L(n+1) = n(n+1)/2 + (n+1) = (n^2 + 3n + 2)/2 = (n+1)(n+2)/2$.
6. But this is exactly $R(n+1)$, so we have shown $L(n+1) = R(n+1)$ under the assumption $L(n) = R(n)$.
7. This proves $L(n) = R(n)$ for all natural numbers $n \geq 1$ by induction.

We started the induction with $n = 1$ instead of $n = 0$, because that was part of the original claim. If you define $1 + \ldots + n$ for $n = 0$ to be 0, which would be normal to most mathematicians, then Gauss' formula is also true for $n = 0$, as $R(0) = 0$.

For a second example, the sum of the squares of natural numbers follows a similar formula:

Claim. For any natural number $n \geq 1$, we have

$$1^2 + 2^2 + \ldots n^2 = \frac{n(n+1)(2n+1)}{6}$$

Proof. By induction, using the template:

1. Let $L(n) = 1^2 + \ldots + n^2$ (or $\sum_{i=1}^{n} i^2$ if you prefer) and $R(n) = n(n+1)(2n+1)/6$.
   We prove $L(n) = R(n)$ for $n \geq 1$ by induction.
2. For the base case, $L(1) = 1^2 = 1$ and $R(1) = 1 \times 2 \times 3/6 = 1$. This proves $L(1) = R(1)$.
3. For the induction step, assume that $L(n) = R(n)$. We want to prove that $L(n+1) = 1^2 + \ldots + (n+1)^2$ and $R(n+1) = (n+1)(n+2)(2n+3)/6$ are equal.
4. Write out $L(n+1) = 1^2 + \ldots + n^2 + (n+1)^2 = L(n) + (n+1)^2$, again splitting off the last term.
5. Using the induction assumption, replace this with $L(n+1) = R(n) + (n+1)^2$ and calculate out to get $L(n+1) = n(n+1)(2n+1)/6 + (n+1)(n+1) = (2n^3 + 9n^2 + 13n + 6)/6$.
6. $R(n+1) = n(n+1)(2n+1)/6 = (2n^3 + 9n^2 + 13n + 6)/6$ which is the same term as above, so $L(n+1) = R(n+1)$. This proves the induction step $L(n) = R(n) \to L(n+1) = R(n+1)$.
7. This proves the claim by induction for all natural $n \geq 1$.

You can see that in both cases, the real work happens in the calculation in steps 4–6, and often it is helpful to expand $R(n+1)$ first to see what term you are aiming for.

For another example, in this case an inequality:

Claim. For all integers $n \geq 4$ we have $n! > 2^n$.
Proof. Using the template, suitably adapted because we have an inequality here:

1. We prove the statement by induction. Here $L(n) = n!$ and $R(n) = 2^n$.
2. For the base case, $L(4) = 4! = 24$ and $R(4) = 2^4 = 16$ so $L(4) > R(4)$.
   Note, $L(3) \not> R(3)$ but that is why we said the claim holds only for $n \geq 4$.
3. For the induction step, assume that $L(n) > R(n)$ and that $n \geq 4$.
   We want to prove that $L(n+1) > R(n+1)$, that is $(n+1)! > 2^{n+1}$.
4. Write out $L(n+1) = (n+1)! = 1 \times \ldots \times n \times (n+1) = (n!) \times (n+1)$.
5. For any two numbers x, y with $x > y$ and any positive number a, we have $ax > ay$ too (this is a law of arithmetic, which you probably know from working with inequalities — you can cancel the a as long as it is positive, without flipping the $>$ sign). So, using the induction assumption $n! \geq 2^n$, we get $(n!)(n+1) > 2^n(n+1)$ since $n+1$ is positive for any natural number n. Further, since $n + 1 > 2$ (from $n \geq 4$), we also have $2^n(n+1) > 2^n \times 2$.
6. Since $a > b$ and $b > c$ implies $a > c$ (another fact of arithmetic), and $2^n \times 2 = 2^{n+1}$, this gives $(n!)(n+1) > 2^{n+1}$.
7. This proves the claim $(n+1)! > 2^{n+1}$ by induction.

Be careful about a possible trap here. Suppose you wanted to prove the weaker statement $n! \geq 2^n$ for $n \geq 4$:

n	$n!$	2^n	$(n! \geq 2^n)?$
0	1	1	T
1	1	2	F
2	2	4	F
3	6	8	F
4	24	16	T
5	120	32	T
...	...	...	T

The statement is true from $n = 4$ onwards (as we have just proved), but the weaker statement with a $\geq$ is also true for $n = 0$, but not $n = 1$. If you try and prove the step $n! \geq 2^n \rightarrow (n + 1)! \geq 2^{n+1}$ with no further assumptions, this will not work, because it is false for $n = 0$. In our proof, we had to make the extra assumption in the induction step that $n+1 > 2$, and you would have to do the same if you were working with $\geq$ everywhere. This is not a problem for the induction: if you are proving something by induction for $n \geq 4$ for example, then you can always assume $n \geq 4$, but you do have to mention it in the proof.

Well-Ordering

Why does induction work — or, for an even better question, when does induction work?

Claim: suppose that $S(n)$ is a proposition with a variable n, and that $S(0)$ is true, and $S(n) \rightarrow S(n+1)$ is true. Then for every $x \in \mathbb{N}$, $S(x)$ is true.
Proof: the proof is by contradiction. Suppose this is not the case, that is there are some natural numbers x for which $S(x)$ is not true. Pick the smallest such number x.

1. If $x = 0$, then $S(0)$ is false, but this contradicts the assumption that $S(0)$ is true.
2. If $x > 0$, then $x - 1$ is also a natural number. Since x was the smallest natural number where S is not true, then $S(x - 1)$ must be true. But then by $S(n) \rightarrow S(n + 1)$, setting $n = x - 1$, we get that $S(x)$ is true as well, which is a contradiction.

This relies on the fact that if S is false for any $x \in \mathbb{N}$, then we can get the *smallest* such x. A set, such as the natural numbers, where every non-empty subset has a smallest element is called *well-ordered* in set theory. This property is true for the natural numbers, but not for the integers or the real numbers, or even the positive real numbers (whether or not you include 0). Indeed, induction as stated here does not work on many other sets of numbers.

The key step in the proof above, as far as set theory is concerned, is to define the subset of all natural numbers where the proposition S does not hold. If S is not a tautology, then this subset is non-empty, so we can apply the well-ordering principle to pick the smallest element of this subset, which gets us a contradiction.

This also explains why induction works for 'all natural numbers greater or equal to some k', for example $k = 1$. In other words, $S(1)$ and $S(n) \rightarrow S(n + 1)$ implies $S(n)$ for all natural $n \geq 1$. This is because the subset of all natural numbers greater or equal to k, and where S does not hold, is still a subset of the natural numbers. So, it must still have a smallest element if it is not empty.

Structural Induction

Whenever you have a recursively defined data type, then you can do proofs by induction over the structure of the data type. For example, consider the following rules for building term strings:

T1 An integer is a term, called a value term.
T2 A variable is a term (assume variable names are the letters a–z or something like that).
T3 If X, Y are terms then so are $X + Y$ and $X - Y$ and $X \times Y$. These are called operation terms.
T4 If X is a term then so is (X).

From this structure, we can prove that every term string has balanced brackets. Let T be any term. Then:

1. If T is a value term, then T contains no brackets at all (assume we write integers without brackets, even if negative). Therefore, T has balanced brackets, namely 0 opening and 0 closing brackets.
2. If T is a variable term, then T still contains no brackets (assume brackets are not allowed in variable names) so brackets are still balanced.
3. If T is an operation term, then our induction assumption is that the subterms X, Y both have balanced brackets. Suppose X has a opening and closing brackets, and Y has b opening and closing brackets. Then T has $a + b$ opening and closing brackets each, which is still balanced.
4. if T is a bracket term, then our induction assumption is that the subterm X has balanced brackets; let the number of opening (and closing) brackets in X be n. Then T has $n+1$ opening and closing brackets each, which is still balanced.

For another example, we prove that every term built according to these rules has one more leaf term (value and variable terms) than it has operation terms. For any term T,

1. If T is a value term, then it contains $m = 1$ value terms (itself) and $n = 0$ operation terms, and $m = n + 1$ as required.
2. If T is variable term, then $m = 1, n = 0, m = n + 1$ as above.
3. If T is an operation term with two children X and Y, let m_X, n_X and m_Y, n_Y be the numbers of leaf and operation terms in X, Y respectively. Our induction assumption is that $m_X = n_X + 1$ and $m_Y = n_Y + 1$. Then the number of operation terms in total in T is $n = n_X + n_Y + 1$, and the number of leaf terms in total is $m = m_X + m_Y$. It follows that $m = n + 1$ again.
4. If T is a bracket term with a child term X with m, n leaf and operation terms respectively, then our induction assumption is $m = n+1$. Since bracket terms are not operation terms, the same m, n apply to T itself.

For another example, consider the following rules for producing a certain strings (this kind of thing is called a context-free grammar, and is used to create parsers for programming languages):

S1 The string S is a valid string.
S2 If you have a valid string with the letter S in it, you can replace one instance of the letter S with the substring aSb and the result is a valid string.
S3 If you have a valid string with the letter S in it, you can delete one instance of the letter S and the result is still a valid string.

Let us call valid strings that do not contain the letter S 'final strings'.

The final strings you can build with this grammar are of the form $a^n b^n$ where a^n means the letter a repeated n times, for n a natural number (this includes the case $n = 0$ giving the empty string). For example, you can use the sequence of rules

$$\xrightarrow{\text{S1}} S \xrightarrow{\text{S2}} aSb \xrightarrow{\text{S2}} aaSbb \xrightarrow{\text{S3}} aabb$$

to produce the string `aabb`. We can prove by induction:

Claim. All strings produced by this grammar have equal numbers of `a` and `b` characters.
Proof. By structural induction,

1. If X is a string created by rule S1, then X has no `a` or `b` characters. If we set m, n to be the counts of these characters respectively, then $m = 0$ and $n = 0$, so $m = n$.
2. If X is a string created by rule S2 from another string Y, then our induction assumption is that Y has equal numbers of `a` and `b` characters; let their counts be m and n respectively. Then for X, we have $m + 1$ and $n + 1$ `a` and `b` characters respectively, and since we had $m = n$, we also have $m + 1 = n + 1$.
3. If X is a string created by rule S3 from a string Y, then the number of `a` and `b` characters does not change; by induction assumption, the counts were balanced in Y, so they are still balanced in X.

Induction over the natural numbers is actually a special case of structural induction. We can define the natural numbers as follows in an alternate notation:

1. 0 is a natural number.
2. If N is a natural number, then so is SN (called the successor of N).

This gives us $0, 1 = S0, 2 = SS0, 3 = SSS0, \ldots$ so the natural numbers themselves are a recursively defined structure, and a general version of the well-ordering principle applies for all such structures. An induction proof for a claim C then has to show that

1. $C(0)$ is true (the base case).
2. If $C(N)$ is true then $C(SN)$ is true (the induction step, normal people write SN as $N + 1$).

We summarise the rules for structural induction in this definition.

Definition 2 (recursively defined structures)**.** A recursively defined structure (RDS), also known as an algebraic data type (ADT), is a type of mathematical objects defined by a set of rules, where any object of this type can be built using the rules a finite number of times.

Some of the rules may build new objects of the type out of other objects of the same type. These rules are called recursive, whereas rules that do not do this are called basic.

A proof by structural induction for a property P shows that

- For every basic rule that constructs an object A, property P holds for object A.
- For every recursive rule that constructs an object Z out of objects $A, B, C, \ldots$ of the same type, if the objects $A, B, C, \ldots$ have property P, then so does the new object Z.

Then, all objects of the type defined by this RDS have property P.

Well-Ordering for Structural Induction

Here is a sketch of the well-ordering principle for structural induction. Consider simplified terms that can be build from only two rules:

T1 If n is an integer, then n is a term, called a value term.
T2 If A, B are terms, then $A + B$ is a term, called an operation term.

On any RDS, we can define a relation $\sqsubseteq$ by saying that $X \sqsubseteq Y$ if X is a subterm of Y. This is called a *partial order* as it has some of the properties of $\leq$, such that if $X \sqsubseteq Y$ and $Y \sqsubseteq Z$ then also $X \sqsubseteq Z$, but for any two non-equal terms X, Y, there are three possible cases: $X \sqsubseteq Y$, $Y \sqsubseteq X$, or neither (this last case cannot happen with $\leq$ for integers).

For example, $1 \sqsubseteq 1 + 2$ and $1 + 2 \sqsubseteq 1 + 3$ but for $X = 1 + 2$ and $Y = 1 + 3$, neither is a subterm of the other.

This relation still has the well-ordering property that any non-empty subset of terms has a minimal term M in the sense that no other term N in the subset has $N \sqsubseteq M$. So the claim is: if a property P (a) holds for all value terms, and (b) if $P(A) \wedge P(B) \rightarrow P(A + B)$ holds, then $P(X)$ holds for all terms X. The proof is by contradiction.

Suppose there is some term X where $P(X)$ does not hold. Then the set of terms where P does not hold is not empty, so we can pick a minimal such term, that is a term M where (1) $P(M)$ does not hold and (2) for all terms Y with $Y \sqsubseteq M$, we have that $P(Y)$ does hold.

If our minimal term M is a value term, this contradicts (a) where we stated that P holds for all value terms. If M is an operation term, then $M = A + B$ for some terms A, B with $A \sqsubseteq M$ and $B \sqsubseteq M$. Since M was a minimal term where P does not hold, then $P(A)$ and $P(B)$ do hold, as these are subterms. But this contradicts (b) where we said that $P(A) \wedge P(B) \rightarrow P(A + B)$.

This last section was very abstract, and it is fine if only the top students understand it in Year 1 — the idea will come around again in Year 2 or later. You should be able to do structural induction on RDSs, like arguing why all terms in some definition must have balanced brackets, even if you do not understand the full theory yet.

Invariant Induction

Invariant induction is a way to prove that a program containing a loop does the right thing. This can include two parts:

1. A loop *invariant* is a formula that is true before the loop starts, and after every pass through the loop, if the formula was true at the start then it is still true at the end of the pass. By induction, this means that the formula will hold if the loop ever finishes, even if we do not know how many times the loop will run.
2. A loop *variant* is a formula that produces a fixed positive value before the loop starts, decreases by at least a fixed positive value after every pass through the loop, and can never become negative[2]. By induction, this proves that the loop will finish at some point (as opposed to looping forever).

[2]Alternatively, one can show that there is some integer w, which could be positive or negative, such that the variant cannot drop below w. This works just as well, because if we have a variant v that cannot drop below (-5) for example, then $v' = v + 5$ would be a variant that cannot become negative.

Together, a loop invariant and variant can prove that a loop always computes the value that it is supposed to.

For example, here is a loop for multiplication by repeated addition:

```
// inputs: a, b natural numbers
// output: c = a * b
c = 0
while b > 0 do
    c = c + a
    b = b - 1
end while
return c
```

We claim that at the end of the loop, the value of the variable c is $a_0 b_0$ where a_0 and b_0 are the initial values of a, b (we need new names for these values since the variables a, b themselves might have changed). This assumes that a, b, c are all natural numbers, that is they do not 'wrap around' like fixed-width integers (though the argument would work the same if they did).

Loop variant: The loop variant is simply b.

- Before the loop starts, since b is a natural number, we have $b \geq 0$.
- In each pass through the loop, b decreases by 1.
- The condition at the start of the loop terminates the loop if b ever reaches 0. Therefore, decreasing b in the loop cannot make it negative.

It follows that this loop cannot go on looping forever. In fact, in this case we can say that it terminates after exactly b passes.

Loop invariant: for the original values a_0, b_0 of the inputs, and the current values of a, b, c, we have $c = a_0 b_0 - ab$.

- Before the first pass, $a = a_0$ and $b = b_0$. Then $a_0 b_0 - ab = 0$, and that is the initial value for c.
- Suppose that before a pass through the loop, we have a, b set to values A, B and c set to $C = a_0 b_0 - AB$. After the pass, we have $a = A$, $b = B - 1$ and $c = C + A$. (Here lowercase variables are the values at the end of the pass, and uppercase variables are the values at the start.) Then $c = C + A = a_0 b_0 - AB + A = a_0 b_0 - a(b + 1) + a = a_0 b_0 - ab$. So, if the invariant was true before the pass, it remains true after the pass through the loop.
- By induction, when the loop terminates (which we know from the variant that it does), then we have $c = a_0 b_0 - ab$, and $b = 0$ (since that is what terminates the loop). Therefore, at the end of the loop, $c = a_0 b_0$.

We summarise the principles of invariant induction:

Definition 3 (invariant induction)**.** A 'formula' here means a term involving, potentially, both the initial values of a program's variables at the start and the current values of any such variables.

If you have a formula v called a *loop variant* that is

1. Set to a non-negative value v_0 before the loop starts.
2. Decreases by at least some fixed positive value v_1 on each pass through the loop.

3. The loop terminates if v ever reaches 0 (or falls below 0).

then the loop is guaranteed to terminate, and will not run forever.
(Actually, it will terminate after at most v_0/v_1 (rounded up) passes.)

If you have a formula j called a *loop invariant* that

1. Holds when the loop starts for the first time.
2. If the formula holds at the start of a pass through the loop, then it still holds at the end of this pass.

then the formula j will hold if the loop ever terminates (which you can, for example, show with a loop variant).

The reasoning behind this principle is another proof by contradiction. Suppose, at some point when running the program, the invariant does not hold at the start of the pass through the loop. Pick the first time this happens. If this was before the first pass, then it contradicts the statement that the invariant holds before the loop starts. If this was before the n-th pass for $n > 1$, then since this was the first time the invariant does not hold, it must have held before the $n-1$-st pass (since $n > 1$, then $n - 1 > 0$ so there definitely was a $n-1$-st pass). But this contradiction the claim that if the formula held at the start of the $n-1$-st pass, then it also holds at the end of the $n-1$-st pass, which is the start of the n-th pass.

Euclid's Theorem: Existence

With loop invariants, we can prove the existence part of Euclid's theorem.

Claim. For integers a, b with $b > 0$, a pair of integers q, r exists with $a = qb + r$, $r \geq 0$ and $r < b$.

Proof. We prove this by giving an algorithm that finds such a pair. Before we do this, we split into three cases again:

- If $a = 0$, then $q = 0, r = 0$ is a valid pair. We do not need the algorithm in this case.
- For $a > 0$, we run the algorithm below.
- For $a < 0$, we run the algorithm on $(-a)$ and b, giving q', r' with $(-a) = q'b + r'$. If $r' = 0$ then we set $q = -q'$ and $r = 0$. If $r' > 0$, we set $q = -q' - 1$ and $r = b - r'$. This gives us $qb + r = (-q' - 1)b + (b - r') = -q'b - r' = -(-a) = a$, and because $0 \leq r' < b$ from the algorithm, and $r' > 0$, then $r = b - r'$ also has the property $0 \leq r < b$.

Here is the algorithm, that does division with remainder:

```
// input: integers a, b with a, b > 0
// output: integers q, r
q = 0
r = a
while r >= b do
    q = q + 1
    r = r - b
end while
return q, r
```

First, the loop variant is r.

1. At the start of the loop, $r = a$ and we know $a > 0$
   (otherwise we would have run the algorithm on $(-a)$, to make it positive).
2. In each pass through the loop, r decreases by b which we know is a positive integer.
3. Whenever $r < b$ at the end of a pass, the loop terminates, so r cannot become negative.

Therefore, this algorithm terminates.

To prove that it works correctly, we use the loop invariant $a = qb + r$.

1. Before the loop starts, we have $q = 0, r = a$ so $a = qb + r$ holds.
2. Suppose that, at the start of a loop iteration, the variables are set to values Q, R and $a = Qb+R$ holds. Then for the values q, r at the and of the loop, we have $q = Q + 1$ and $r = R - b$. Then $qb + r = (Q + 1)b + (R - b) = Qb + R = a$, so $a = qb + r$ still holds at the end of the loop.

When the loop terminates, which the loop variant proof shows that it always will, then we have $a = qb + r$ from the loop invariant, and $r < b$ (since that is what terminates the loop) and $r \geq 0$ (as we showed in the loop variant proof).

In conclusion, at the end of the loop, we have values q, r with $a = qb + r$, $r \geq 0$ and $r < b$. This meets the conditions of the theorem.

We note that this algorithm is not how you would do division in practice, it is much less efficient than how you would really do division with remainder. But the algorithm is simple enough to reason about in a proof.

Another example of loop variants and invariants is a much more efficient multiplication algorithm. Suppose we want to multiply numbers a, b on a computer, and we have a data type that is big enough to not cause an overflow. If b is written in binary, which is normally the case on a computer, then we can label the individual bits of b as $b_{(n)} \dots b_{(2)} b_{(1)} b_{(0)}$ where $b_{(0)}$ is the least significant bit, so mathematically, $b = \sum_{i=0}^{n} 2^i b_{(i)}$.

Here is the multiplication algorithm for natural numbers.

```
// input: a, b >= 0
// output: c = a * b
c = 0
x = a
y = b
while y > 0
    if (y & 1) == 1 then
        c = c + x
    end if
    x = x << 1
    y = y >> 1
end while
return c
```

The operation `x << 1` shifts x to the left by one bit adding a 0 bit on the end, which has the effect of doubling it (assuming unsigned integers and no overflow, which we are assuming here as we are

working on natural numbers), and shifting to the right halves a number, rounding down (so both $2k$ and $2k+1$ become k).

The operation `y & 1` is the bitwise logical and of y and the number 1, which in binary is $0\ldots0001$ so this operation[3] selects the least significant bit of y, that is $y_{(0)}$.

The loop variant here is y (for any loop where the condition is 'while the loop variable is greater than zero', the loop variable itself is an obvious candidate for the loop variant).

1. At the start, $y = b$ and we know $b \geq 0$ as this algorithm is only specified for natural numbers.
2. In each pass through the loop, we shift y one bit to the right, which decreases y by at least one as long as $y > 0$ (bit shifting does not change 0, but when $y = 0$ the loop stops, so we will never do a pass through the loop that does not decrease y).
3. Whenever y hits 0, we terminate the loop, and bit shifting a positive number y to the right can never make it negative, so $y \geq 0$ throughout the entire program.

The loop invariant is $c = ab - xy$. Here a, b are fixed initial values that do not change, and x, y are variables that do change so we use our convention of capital letters for previous values.

1. Before the loop starts, $xy = ab$ and $c = 0$ so $c = ab - xy$ holds.
2. Suppose that at the end of the loop, our variables have values x, y, c, and that X, Y, C were their values at the start of this pass of the loop, and by the induction assumption, $C = ab - XY$.
   We make a case distinction on whether Y is odd or even. If Y is even, then $Y_{(0)}$ is 0, so $c = C$, $Y = 2y$ and $x = 2X$. Therefore, $x(2y) = (2X)Y$ so $xy = XY$. By the induction assumption, $C = ab = XY$ so $c = ab - xy$ by substitution.
   If Y is odd, then $c = C + X$, $x = 2X$ and $2y = (Y - 1)$. The last equation holds because to get from y back to Y, we could double to shift left by one bit, and then add 1 to restore the missing 1 bit since Y was odd. So, $x(2y) = (2X)(Y - 1)$ giving $xy = XY - X$. By the induction assumption, $C = ab - XY$ so we substitute in to get $c = ab - XY + X = ab - (XY - X) = ab - xy$. Therefore, if $C = ab - XY$ we always end up with $c = ab - xy$, which is our induction step.
3. When y becomes 0, which must happen eventually due to the loop variant, then $c = ab - 0 = ab$, which completes the proof that the algorithm works correctly.

We can also ask how long this algorithm takes. Multiplying ab by repeated addition of b as we showed earlier would take b passes through the loop, but here, we only make one pass per bit in b, and the number of bits in a number b is $\log_2(b)$ (rounded up). So instead of b passes, this algorithm only takes roughly $\log(b)$ passes, which is much faster. You will see in your algorithms units that we can more or less ignore both the rounding up and the base of the logarithm when we calculate the running time of an algorithm.

[3] In C, we could just write `if (y & 1) { ... }` since integers have truth values: 0 is false, anything else is true.