How to Write Math

Ryan Culpepper

CS 624, 2024 Fall

1 General Comments

1.1 Your Goal

Pragmatically, you have two goals in your course work for this class, including both the homeworks and the exams:

- to gain a better understanding of algorithms, their design, and the analysis of their correctness and running times
- to convince me, your professor, that you have gained that understanding

That is the standard I use for grading: Did you convince me that you understand the question and its correct answer? Let me warn you: It is possible to mention many relevant words and to state many true facts and still get 0% credit on a homework or exam question.

1.2 Proving a Universal

An isolated statement like "If a number n is odd, its successor $n + 1$ is even" is *implicitly* universally quantified. It is a claim about all (natural) numbers.

To prove a universal statement like the one above, you must not assume anything about n other than what is given—that n is odd. In particular, it is not sufficient to simply *check* the claim on one value of n, or even multiple values of n.

For example, "Suppose $n = 5$. Then n is odd, and $5 + 1 = 6$, which is even." *This is not a proof, it is a test case.* The fact that one test case passes does not mean that the statement is universally true.

If you don't know whether a statement is true or false, it can be useful to check it against a couple concrete examples to help you decide whether you want to try to prove it or disprove it. But checking the examples never proves the universal statement. (A single counter-example, though, does *disprove* a universal statement.)

Here is a proper proof:

Theorem 1. *Let $n \in \mathbb{N}$. If n is odd, then $n + 1$ is even.*

Proof. Since n is odd, there exists $k \in \mathbb{N}$ such that $n = 2k + 1$. (This is the definition of "odd".) Then $n + 1 = 2k + 1 + 1 = 2k + 2 = 2(k + 1)$. That is, $n + 1$ is exactly twice some natural number, so by definition $n + 1$ is even. □

2 Proof by Induction

Induction is a technique for establishing a property holds for *all natural numbers*, or in some cases *all natural numbers starting at some point.*

Examples of universal properties that could potentially be proven by induction:

- $\forall n \in \mathbb{N} : \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- Every natural number greater than or equal to 2 can be written as the product of one or more primes. (The same prime can occur multiple times in the product.)

A proof by induction always contains two cases:

- the base case, where you must prove that the property holds for 0 (or for the alternative starting point, if it isn't 0)
- the inductive case, where you must prove that if the property holds for some number n, it also holds for the successor $n + 1$ (this is slightly different for strong induction)

Requirements: You must clearly label the base case and state the base case's goal. You must clearly label the inductive case, state the inductive case's goal, label the inductive hypothesis, and clearly state the inductive hypothesis.

2.1 Standard (Weak) Induction

Theorem 2. $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$, *for all* $n \in \mathbb{N}$.

Proof. By induction on n.

Base case:

Goal: $\sum_{k=1}^{0} k = \frac{0 \cdot (0+1)}{2}$

This simplies to $0 = 0$, true.

Inductive case:

Assume the inductive hypothesis: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
Goal: $\sum_{k=1}^{n+1} k = \frac{(n+1)(n+2)}{2}$

Calculate:

$$\begin{aligned}
\sum_{k=1}^{n+1} k &= (n+1) + \sum_{k=1}^{n} k && \text{split off last iteration} \\
&= (n+1) + \frac{n(n+1)}{2} && \text{by IH} \\
&= \frac{(n+2)(n+1)}{2} && \text{simplify}
\end{aligned}$$

□

2.2 Strong Induction

In strong induction (aka "course of values" induction), the inductive hypothesis states that the property holds not only for the *previous* value, but for *all strictly smaller* values.

Theorem 3. *Every natural number $n > 2$ can be written as the product of one or more primes. (The same prime can occur multiple times in the product.)*

Proof. By strong induction on n.

Base case ($n = 2$): Goal: 2 can be written as the product of one or more primes. In fact, 2 is prime, so it can be written as the product of a single prime, itself. Done.

Inductive case:

Inductive hypothesis: k can be written as the product of one or more primes, for all $2 \leq k < n$

Goal: n can be written as the product of one or more primes

There are two cases:

1. n is prime: Then n can be written as the product of one prime, itself.
2. n is composite. Then there must be natural numbers $a, b < n$ such that $n = ab$. Since $a < n$, we can apply the IH, and so a must be expressible as the product of one or more primes; likewise $b < n$, so b is expressible as the product of one or more primes. And since $n = ab$, n is expressible as the product of all of a's primes and all of b's primes.

□

3 Proof by Loop Invariant

A *loop invariant* is a proposition, generally involving variables that are updated by the loop body, that holds at key points during the program. Specifically, it must hold

- immediately before the execution of the loop
- at the beginning of each iteration

- at the end of each iteration
- immediately after the loop exits

The loop invariant typically does not hold at all points during the execution of the loop body. The loop body usually temporarily *disrupts* the loop invariant and then *repairs* it before the end of the loop iteration. This definition assumes that the loop test does not cause side-effects (it does not update variables or state, for example).

For a **for** loop, "immediately before the execution of the loop" is considered to be *after* the initialization of the loop counter, and "at the end of each iteration" is considered to be *after* the increment or decrement of the loop counter. The next section illustrates this point with an example of a **for** loop and its equivalent **while** loop.

A proof by loop invariant has three parts: two obligations and a payoff.

- Initialization (obligation): Show the loop invariant holds immediately before the start of the loop.
- Maintenance (obligation): Assuming that the loop invariant and the loop test are true at the start of the loop, show that the loop invariant is true at the end of the loop.
- Termination (payoff): Immediately after the loop exits, you may rely on both the loop invariant and the negation of the loop test.

Requirements: In a proof by loop invariant, your proof must contain all three parts, clearly labeled. If the loop invariant is not given to in the problem text, you must clearly state and label the loop invariant itself before you begin the three parts of the proof.

3.1 Loop Invariant for a *for* Loop

Consider the following algorithm which computes the sum of an array:

Algorithm 1 Sum(A)

```
1: r ← 0
2: for i ← 1 to Length(A) do
3:     r ← r + A[i]                    ← LI holds
4:     // increment loop counter i     ← LI holds
5: end for                             ← LI holds
6: return r
```

Algorithm 2 Sum′(A)

```
1: r ← 0, i ← 1                        ← LI holds
2: while i ≤ Length(A) do              ← LI holds
3:     r ← r + A[i]
4:     i ← i + 1                       ← LI holds
5: end while                           ← LI holds
6: return r
```

The two ways of expressing the algorithm (**for** vs **while**) are equivalent. The **while** formulation makes the counter initialization and update explicit, and it helps clarify where the loop invariant holds. The loop invariant temporarily does *not* hold between lines 3 and 4.

Theorem 4. *Let A be an array of numbers. Then*

$$\text{Sum}(A) = \sum_{k=1}^{\text{Length}(A)} A[k]$$

Proof. By loop invariant. The loop invariant (LI) is

$$r = \sum_{k=1}^{i-1} A[k]$$

Initialization: At the start of the loop, $r = 0$ and $i = 1$, so the range of summation in the LI is empty, and the LI is just $0 = 0$, true.

Maintenance: We assume the loop invariant holds at the start of some iteration, and we show that the loop invariant holds after the execution of the loop body (including the increment of the loop counter).

We assume the LI: $r = \sum_{k=1}^{i-1} A[k]$.

The loop body updates r by adding $A[i]$, so afterwards

$$\begin{aligned} r &= \left(\sum_{k=1}^{i-1} A[k] \right) + A[i] && \text{(by line 3)} \\ &= \sum_{k=1}^{i} A[k] && \text{(absorb term into summation)} \end{aligned}$$

which is $r = \sum_{k=1}^{(i+1)-1} A[k]$, which is the LI for the next value of the counter, $i + 1$.

Termination: After the loop exits, $i = \text{Length}(A) + 1$. (Not $\text{Length}(A)$!)

So the LI is $r = \sum_{k=1}^{(\text{Length}(A)+1)-1} A[k] = \sum_{k=1}^{\text{Length}(A)} A[k]$. Since r is the procedure's return value, this is the value of $\text{Sum}(A)$, and so the procedure's correctness is proved. □

Here is another way to argue maintenance, which is more precise and flexible:

Maintenance:

Let r_0 and i_0 refer to the values of r and i at the beginning of the iteration, and let r and i refer to the values of the variables at the end of the iteration, *including* the increment of the loop counter i.

Assume the LI holds at the beginning. That is, $r_0 = \sum_{k=1}^{i_0-1} A[k]$.

After the execution of the loop body (and counter increment), $r = r_0 + A[i_0]$ and $i = i_0 + 1$.

So:

$$\begin{aligned} r &= r_0 + A[i_0] && \text{(by line 3)} \\ &= \left(\sum_{k=1}^{i_0-1} A[k]\right) + A[i_0] && \text{(by LI)} \\ &= \sum_{k=1}^{i_0} A[k] && \text{(absorb term into summation)} \\ &= \sum_{k=1}^{i-1} A[k] && \text{(because } i = i_0 + 1\text{)} \end{aligned}$$

So at the end of the iteration, the LI holds for the new values of the variables.

3.2 Loop Invariant for a *while* Loop

Consider the following algorithm:

Algorithm 3 Insert(A, p, r)

Require: $A[p \,..\, r-1]$ is sorted
Ensure: $A[p \,..\, r]$ is sorted

```
1: q ← r
2: while p < q and A[q − 1] > A[q] do
3:     exchange A[q − 1] ↔ A[q]
4:     q ← q − 1
5: end while
```

Theorem 5. Insert *is correct. That is, if* Insert *is called with arguments satisfying its precondition, its postcondition is satisfied upon its return.*

Proof. By loop invariant. The loop invariant is

(a) $A[p \,..\, q-1]$ is sorted

(b) $A[q \,..\, r]$ is sorted

(c) $A[p \,..\, q] \leq A[q+1 \,..\, r]$

Initialization:

Immediately before the loop, $q = r$. LI(a) is $A[p \,..\, r-1]$ is sorted, which holds by the precondition. LI(b) is $A[r \,..\, r]$ is sorted, which holds because the range contains a single element. LI(c) is $A[p \,..\, r] \leq A[r+1 \,..\, r]$ which holds vacuously because the second range is empty.

Maintenance:

Note: I'm using a different convention for old/new values this time.

Let q refer to the value of q at the beginning of the iteration, and q' refer to the value of the variable at the end. Similarly, let A refer to the contents of the array at the beginning, and A' refer to the contents at the end of the iteration.

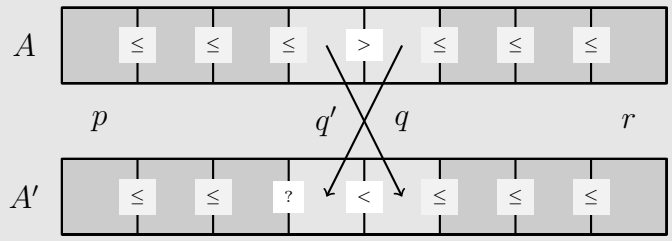
From LI: $A[p \,..\, q-1]$ is sorted, $A[q \,..\, r]$ is sorted, and $A[p \,..\, q-1] \le A[q+1 \,..\, r]$.

From the loop test we know $p < q$ and $A[q-1] > A[q]$.

From the exchange on line 3 we have $A'[q-1] = A[q]$ and $A'[q] = A[q-1]$, and from line 4 we have $q' = q-1$. Combining them, we get $A'[q'] = A[q]$ and $A'[q'+1] = A[q-1]$.

Except for the exchanged indexes ($q-1, q$ or $q', q'+1$) the array is unchanged, so $A'[p \,..\, q'-1] = A[p \,..\, q'-1]$ and $A'[q'+2 \,..\, r] = A[q'+2 \,..\, r]$.

In summary:



We need to establish three properties:

(a) Goal: $A'[p \,..\, q'-1]$ is sorted.

This part of the array is unchanged, and it is sorted by LI(a).

(b) Goal: $A'[q' \,..\, r]$ is sorted

The array elements at $q'+2 \,..\, r$ are unchanged and are sorted by LI(b).

From the loop test, $A[q-1] > A[q]$, so after the exchange, $A'[q-1] < A'[q]$; that is, $A'[q'] < A'[q'+1]$.

By LI(c), $A[q-1] \le A[q+1]$. That rewrites to $A'[q'+1] \le A'[q'+2]$.

Combining them, we get $A'[q'] < A'[q'+1] \le A'[q'+2]$, and we already knew $A'[q'+2 \,..\, r]$ is sorted, so $A'[q' \,..\, r]$ is sorted.

(c) Goal: $A'[p \,..\, q'] \le A'[q'+1 \,..\, r]$

Again, except for the exchanged indexes ($q-1, q$ or $q', q'+1$) the array is unchanged, so $A'[p \,..\, q'-1] = A[p \,..\, q'-1]$ and $A'[q'+2 \,..\, r] = A[q'+2 \,..\, r]$.

Combining that with LI(c), we have $A'[p \,..\, q'-1] \le A'[q'+2 \,..\, r]$. To complete the goal, we need to show two additional facts:

- Goal: $A'[q'] \le A'[q'+1 \,..\, r]$.

  This is implied by condition (b), already proved.

- Goal: $A'[p \,..\, q'] \leq A'[q'+1]$.

  From LI(a) we have $A[p \,..\, q-1]$ is sorted, so $A[p \,..\, q-2] \leq A[q-1]$. Rewriting with $A[p \,..\, q-2] = A'[p \,..\, q'-1]$ and $A[q-1] = A'[q'+1]$ gives us that $A'[p \,..\, q'-1] \leq A'[q'+1]$.

  From the loop test, we have $A'[q'] < A'[q'+1]$ (see part (b)).

  Combining those we get $A'[p \,..\, q'] \leq A'[q'+1]$.

Termination: After the loop exits, there are two possibilities arising from the negation of the loop test:

- Case $p = q$:

  Then by LI(b), $A[p \,..\, r]$ is sorted, as the postcondition requires.

  (Technically, it is possible that $q < p$, but only if $r < p$. In that case the array range in question is empty, the loop never executes, and the postcondition is trivially satisfied.)

- Case $A[q-1] \leq A[q]$:

  From LI(a), we have $A[p \,..\, q-1]$ is sorted; from LI(b), we have $A[q \,..\, r]$ is sorted. We glue them together with $A[q-1] \leq A[q]$ to get $A[p \,..\, r]$ is sorted, as required.

□