1. More complete induction

Definition. A prime factorization of a natural number \( n \) is a sequence of primes whose product is \( n \) (repeats are allowed).

For example, \( (2,2,59) \) is a prime factorization of 236 and \( (2,2,3,3,7) \) is a prime factorization of 252.

Theorem. Any natural number \( n \geq 2 \) has a prime factorization.

Define the predicate:

\[ P(n) : n \text{ has a prime factorization} \]

We will be using complete induction, starting at \( n = 2 \). Formally, here is our induction principle:

\[
\forall n \in \mathbb{N}, \left[ n \geq 2 \land \forall k \in \mathbb{N}, 2 \leq k < n \rightarrow P(k) \right] \rightarrow \forall n \in \mathbb{N}, n \geq 2 \rightarrow P(n)
\]

Proof. By induction. Let \( n \in \mathbb{N} \) be such that \( n \geq 2 \). Assume that \( P(k) \) holds for all \( 2 \leq k < n \).

Case 1: \( n \) is prime. Then \( \langle n \rangle \) is a prime factorization of \( n \).

Case 2: \( n \) isn’t prime. Then \( n = a \cdot b \) where \( a, b \in \mathbb{N} \) and \( 1 < a < n \) and \( 1 < b < n \). By the induction hypothesis, \( P(a) \) and \( P(b) \) hold.

So let \( \langle p_1, p_2, \ldots, p_l \rangle \) be a prime factorization of \( a \) and let \( \langle q_1, q_2, \ldots, q_m \rangle \) be a prime factorization of \( b \).

Since \( n = ab \) and \( a = p_1p_2\cdots p_l \) and \( b = q_1q_2\cdots q_m \), \( n = p_1p_2\cdots p_lq_1q_2\cdots q_m \). Furthermore, all of \( p_1, p_2, \ldots, p_l, q_1, q_2, \ldots, q_m \) are prime. Thus, \( \langle p_1, p_2, \ldots, p_l, q_1, q_2, \ldots, q_m \rangle \) is a prime factorization of \( n \). \( \square \)

2. Program correctness

What does it mean for a program to be correct?

It produces the correct output for all acceptable inputs.

How did we determine program correctness in 108/148/150?

Test cases!

Why isn’t this enough?

Only tests a finite number of inputs. If you are smart about your test cases, you can reasonably assure yourself that your program is correct, but test cases can never guarantee correctness.

So how can we guarantee program correctness?

We need a mathematical proof.

Iterative (with loops, i.e. for/while) and recursive programs can be especially tricky for proving program correctness. This is because the number of iterations (or recursive calls) depends on the input.

These proofs require induction!

In order to formally define program correctness, we must define precondition and postcondition.

Definition. The precondition must be true before the program is executed, and the postcondition must be true after the program ends.

Definition. A program is correct if whenever the precondition holds before execution, then

(a) the program terminates (termination), and

(b) the postcondition holds after the program ends (partial correctness).

Example. We want to prove the correctness of a sorting algorithm. What are the precondition and postcondition?

precondition: \( A \) is an array.

postcondition: \( A \) contains the same elements, and is in non-decreasing order.

Now let’s look at the Binary Search algorithm. Binary Search is a way of efficiently finding an element in a sorted array. It takes a sorted array \( A \) and an element \( x \) and returns if \( x \) is in \( A \) or not.

Binary Search works as follows:

- Divide array in half.
- Check if \( x \) is bigger or smaller than the middle element.
- If \( x \) is smaller than the middle, look for \( x \) in the first half of the array. If \( x \) is bigger than the middle, look for \( x \) in the second half of the array.
This is a very intuitive idea. If you can remember back to the days when dictionaries were real books (or when we used phonebooks), you would find what you were looking for by opening up the book, and based on where you were and what you were looking for, determine whether you should look earlier or later in the book.

Note that it is very important that the array is sorted. You cannot be this efficient in your search if the elements are in an arbitrary order.

Here’s some pseudocode (which is really almost Python) for Binary Search:

```python
def BINSEARCH(A, x):
    #Precondition: A is a sorted array of length at least 1, indexed from 0 to length(A)-1
    #Postcondition: Return an integer t such that 0 <= t <= length(A)-1, and A[t]=x if such a t exists, and -1 otherwise.
    f = 0 #f is the first index of the subarray that you are looking for x in
    l = length(A) - 1 #l is the last index of the subarray
    while f != l:
        m = (f+l)/2 #middle element (this is integer division, i.e. rounding down)
        if A[m] >= x: #x is in the first half
            l = m
        else:
            f = m+1
        if A[f] = x:
            return f
        else:
            return -1
```

**Theorem.** Suppose A is a sorted array of length at least 1. Then BINSEARCH(A, x) terminates and returns t such that 0 <= t <= length(A) – 1 and A[t] = x if such a t exists, and -1 otherwise.

There are two parts to prove:

(a) **Termination:** Suppose A is a sorted array of length at least 1. Then BINSEARCH(A, x) terminates.

(b) **Partial correctness:** Suppose A is a sorted array of length at least 1 and BINSEARCH(A, x) terminates. Then it returns t such that 0 <= t <= length(A) – 1 and A[t] = x if such a t exists, or -1 if no such t exists.

Partial correctness is generally the trickier part to prove, so we’ll look at it first. (But make sure not to forget to prove termination! It’s quite important to show that your program doesn’t get into an infinite loop.)

In proving the partial correctness of an iterative program, it’s generally helpful to use a loop invariant.

**Definition.** A loop invariant is a condition that is true before each iteration.

Generally the loop invariant is trivially true at the beginning, and will imply the postcondition if it is true at the end.

So, now, what’s the loop invariant for the Binary Search algorithm? What is true at every iteration? Think about how we defined the algorithm. In each iteration, we are making the subarray smaller, but also making sure the subarray is where x would be if it were in the array. So we ensure that if x is in the array, it is also in the subarray between f and l:

**loop invariant:** 0 <= f <= l <= length(A) – 1, and if x is in A, then x is in A[f...l].

In order to complete the proof of correctness of the Binary Search algorithm, we need to:

- Prove that if the precondition holds, the loop invariant holds for all iterations (this is where we’ll use Induction).
- Prove that if the precondition holds and the program terminates, the loop invariant is true after the last iteration, then the postcondition holds (i.e., the program outputs the correct output). (This will complete the proof of partial correctness.)
- Prove that if the precondition holds, the subarray A[f...l] decreases in size at each iteration (this will prove termination).