CS30 (Discrete Math in CS), Summer 2021 : Lecture 8

Topic: Induction

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please discuss in Piazza/email errors to deeparnab@dartmouth.edu

---

Proving Correctness of Programs.

All of you have experience with programming (indeed, it is one of the pre-reqs for this course). However, I am not sure how many of you have formally reasoned that the code you wrote is "correct". That is, it does what it is supposed to do. Note that running on some test cases does not comprise a reasoning : it shows no more and no less than the fact that your code works on the examples you tried. Indeed, proving code works, or writing code in such a way that one can prove code works is a huge area of CS called *Formal Verification*, and there are "verifiers" out there which can check if certain kinds of code are correct. This is especially true when the company is in the business of writing HUGE chunks of code. Today, we are going to see a sliver of how "inductive reasoning" plays a big role in proving certain kinds of programs are indeed correct.

- **Proving Recursive Programs Correct.** Induction *is* the way to prove that a recursive program is correct. In this lecture we consider a couple of examples, and in the UGP there is another example.
- **Factorial.**

```
1: procedure FACT(n)  ▷ Assume n ∈ ℕ.
2:     if n = 1 then:
3:         return 1
4:     else:
5:         return n·FACT(n − 1)
```

We now prove the following

Theorem 1. For all positive integers n, FACT(n) returns $n!$

Proof. We prove by induction the statement $\forall n \in \mathbb{N} : P(n)$, where $P(n)$ is the predicate "FACT(n) returns $n!$".

Base Case: Let us verify $P(1)$. By definition of the factorial function, $1! = 1$. Now, if $n = 1$, then Line (3) returns 1. Thus, the base case is verified; $P(1)$ is indeed true.

Inductive Case: Let us now assume for a *fixed* $k \in \mathbb{N}$ that $P(k)$ is true. That is FACT(k) indeed returns $k!$. We need to prove $P(k+1)$, that is, we need to prove FACT($k+1$) returns $(k+1)!$.

Inspecting Line (5), we see that FACT($k+1$) returns $(k+1)$ times the number returned by FACT(k). By the induction hypothesis, the latter number is $k!$. Therefore, FACT($k+1$) returns $(k+1)\cdot k! = (k+1)!$. Therefore, the inductive case is true, and so by induction, the theorem is proved. □

- **Binary Search**

```
1:  procedure BinSearch(A, x)  ▷ Assume A is sorted strictly increasing.
2:      ▷ Returns true if x ∈ A, otherwise returns false.
3:      n ← A.length.
4:      if n = 1 then:
5:          if x = A[1] then:
6:              return true.
7:          else:
8:              return false.
9:      else:
10:         m = ⌊n/2⌋.
11:         if x = A[m] then:
12:             return true.
13:         else if x < A[m] then:
14:             return BinSearch(A[1 : m], x).
15:         else:  ▷ That is, x > A[m]
16:             return BinSearch(A[m + 1 : n], x).
```

We say that BinSearch works properly on the input (A, x) if it return true when $x \in A$, and returns false otherwise. The correctness of BinSearch amounts to proving the following theorem.

Theorem 2. For any sorted array of numbers A and any number x, BinSearch works properly on (A, x).

Proof. Let $P(n)$ be the predicate which is true if for *any* sorted array A of *length* n and any x, BinSearch works properly on the input (A, x). We wish to prove $\forall n \in \mathbb{N} : P(n)$. We proceed by induction.

Base Case. Is $P(1)$ true? That is, given any sorted array A of length 1 (that is, containing exactly one element), and any x, does BinSearch work properly on (A, x)? To answer this, let us fix a sorted array A and a number x. If x was indeed in the array A, then it must be that $x = A[1]$. Line 6 then tells us that in this case BinSearch does return true. Similarly, if x was not in the array A, then $x \neq A[1]$. Line 8 then tells us that in this case BinSearch does return false. Thus, in both cases the algorithm behaves properly. $P(1)$ is thus established to be true.

Inductive Case. Fix a natural $k \in \mathbb{N}$. The (strong) induction hypothesis is that $P(1), P(2), \ldots, P(k)$ are all true. We now need to prove $P(k + 1)$. For brevity's sake, let us call $N := k + 1$; we wish to prove $P(N)$, and $P(a)$ is true for all $a < N$. And we have $N > 1$.

That is, we need to show given given any sorted array A of length N, and any x, BinSearch works properly on (A, x). To this end, let us fix a sorted array A of length N and an x.

Case 1: $x \notin A$. In this case, the algorithm should return false. Since $x \notin A$, $x \neq A[m]$. Thus, Line 11 does not run. Furthermore, $x \notin A$, implies $x \notin A[1 : m]$ *and* $x \notin A[m + 1 : N]$. Since both the arrays $A[1 : m]$ and $A[m + 1 : N]$ have lengths $\lfloor N/2 \rfloor$ and $\lceil N/2 \rceil$ which are $< N$ for all $N > 1$, by the (strong) induction hypothesis we have that both BinSearch$(A[1 : m], x)$ and BinSearch$(A[m + 1 : N], x)$ return false. Thus, no matter which of Line 14 or Line 16 runs,

the algorithm BINSEARCH$(A[1:N], x)$ will return false. Thus, in this case, the algorithm works properly.

Case 2: $x \in A$. In this case, there is some $1 \le j \le N$ such that $x = A[j]$. If $j = m$, then Line 11 will return true. If $j < m$, then *since the array is sorted* $x = A[j] < A[m]$. Thus, Line 14 will run. Since $x \in A[1:m]$ and $m = \lfloor N/2 \rfloor < N$, by the (strong) inductive hypothesis, we know that BINSEARCH$(A[1:m], x)$ will return true. If $j > m$, then *since the array is sorted* $x = A[j] > A[m]$. Thus, Line 16 will run. Since $x \in A[m+1:N]$ whose length is $\lceil N/2 \rceil < N$, by the (strong) inductive hypothesis, we know that BINSEARCH$(A[m+1:N], x)$ will return true.

□

- **Proving Iterative Programs Correct.** But not all programs are recursive programs, or at least, not written that way. Often for practical reasons of how the programming language handles recursion. Most of you have probably experienced this first hand : an "iterative" (non-recursive) version of a program is often faster. Below we see a method which is broadly used to prove the correctness of programs involving "loops".

- **A Simple While-Loop.** Let us begin with the following simple piece of code. All of you must have implemented such a thing in CS 1.

```
1:  procedure MAX(A[1 : n]):
2:      ▷ Assume n ≥ 1, that is the array is non-empty.
3:      rmax ← 1; j ← 2.
4:      ▷ Pre: j = 2; rmax = 1; n ≥ 1.
5:      while j ≤ n do:
6:          ▷ Loop Invariant: rmax is the index of a maximum element of A[1 : j − 1].
7:          if A[j] > A[rmax] then:
8:              rmax ← j
9:          j ← j + 1.
10:     ▷ Post: rmax is the index of a maximum element in A[1 : n]
11:     return rmax.
```

To argue that this while-loop is correct, we need to first ***specify*** what the while-loop is supposed to do. This condition is specified in what is called the *Post* condition. It also is useful to specify what we know *before* the while-loop begins. This is called the **Pre** condition. To prove the post condition at the *end* of the loop, one proceeds in the following fashion using **loop invariants**.

- (Creative Step.) Find an assertion/predicate which is supposed to be true at the beginning and end of *every* loop. This is called the loop Invariant (LI). Coming up with the correct loop invariant is the creative step.
- (Usefulness.) When the program terminates, (LI) ⇒ (Post).
- (Base Case.) Prove that (LI) is true at the beginning of the **first loop**. This will involve (Pre) ⇒ (LI)
- (Inductive Case.) Fix any loop number $k \ge 1$. Assume (LI) is true at the beginning of the loop k. Prove that is true at the end of loop k (which is the beginning of loop $k+1$).

– (Termination.) Argue that the while loop terminates. This is done by defining a function D which maps the "state" of the code (all variables) to a natural number while the loop is running, that is, $\{1, 2, \ldots\}$. If we can show that in every loop this **strictly decreases** then we would prove that the code terminates.

For the snippet above, the loop invariant is written in blue. We will mention this here as well

(**Loop Invariant**) rmax is the index of a maximum element in the sub-array $A[1 : j-1]$

– (Usefulness.) At the end of the while-loop, the value of $j = n+1$. And thus (LI) $\Rightarrow$ (Post).
– (Base Case.) At the beginning of the first loop, $j = 2$ and rmax $= 1$. And 1 is indeed the index of the maximum element of the array $A[1:1]$ which contains only one element.
– (Inductive Case.) Fix any loop. At the beginning of the loop, let the value of rmax be r. We know $A[r] = \max(A[1], \ldots, A[j-1])$ by (LI). In Line 7 if $A[r] \geq A[j]$ then rmax is unchanged, and we get $A[\text{rmax}] = \max(A[1], \ldots, A[j])$. Otherwise, rmax $\leftarrow j$, and since $A[j] > A[r] \geq \max(A[1], \ldots, A[j-1])$, we see $A[j] = \max(A[1], \ldots, A[j-1])$. Thus, in this case as well $A[\text{rmax}] \max(A[1], \ldots, A[j])$. Then in Line 9 $j \leftarrow j+1$, and thus $A[\text{rmax}] = \max(A[1], \ldots, A[j-1])$, that is, (LI) holds.
– (Termination.) This is especially easy here since j increments. Consider the function $D(j) = (n+1) - j$. Does this map to natural numbers? Yes, because while the loop's condition is satisfied, $j \leq n$ and thus $(n+1) - j \geq 1$. Does this decrement in each loop? Sure, since j increments by 1 in each loop. Therefore the while loop terminates.

To summarize succinctly, to prove a while loop of the form

while (cond) : code

is correct, the ingredients are

– Establish (Pre) and (Post).
– Figure out a loop invariant (LI) and a decrementing function D.
– (Usefulness): Establish (LI) and $\neg$code $\Rightarrow$ (Post).
– (Base Case): (Pre) $\Rightarrow$ (LI).
– (Inductive case): (LI) + cond $\Rightarrow$ (LI), at the end of any run of code.
– (Termination): Establish D maps to natural numbers **and** strictly decreases in every loop.

- **What about For-loops?** For-loops are often much easier to argue about. But one can "reduce" for-loops to while loops by noticing that every for-loop can be written as a while loop as follows.

```
1: procedure FOR(input):
2:     for a ≤ j ≤ b do:
3:         stuff.
```

≡

```
1: procedure WHILE(input):
2:     j ← a.
3:     while j ≤ b do:
4:         stuff.
5:         j ← j + 1
```