Virtues of Correctness Proofs

Numerous computer scientists, cf. *Howell*, believe that

> *people do not fully understand an algorithm until they are able to prove its correctness.*

Why correctness proofs?

- easier to design algorithms if correctness proof in mind
- may uncover subtle errors that would be hard to find with testing alone
- allow us to understand specific algorithms on a much deeper level

Path ahead:

- this week: develop rigorous proofs
- rest of this course: keep correctness in mind as we develop advanced algorithms
- in your career: the more critical the code, the more the need for verified correctness

While Loops with Invariants

Form of generic while loop:

Preamble
while Guard
    Body

A Loop Invariant must hold each time that the Guard is evaluated. In particular:

- the Preamble must establish the Invariant
- the Body must maintain the Invariant

What do we know at loop exit?

- in a structured language:
  - the Invariant holds
  - the negation of the Guard holds
- in a non-structured language where one may jump out of loops:
  - we do not know much (without further efforts)
  - *Goto Statement Considered Harmful* (Dijkstra)

Checklist for While Loops

Consider a loop

Preamble
while Guard
    Body

with loop invariant Φ. Then we must verify

1. Establish: that Φ holds after Preamble
2. Maintain: that if Φ holds before Body then Φ also holds after Body
3. Correctness: the desired postcondition follows from Φ, and the negation of Guard.

Those 3 items suffice for partial correctness; for total correctness we also need

4. Terminate if Φ holds then Guard will eventually become false.

Iterative Fibonacci

Recall the recursive fibonacci algorithm:

$$\begin{aligned} \text{fib}(0) = \text{fib}(1) &= 1 \\ \text{fib}(n+2) &= \text{fib}(n) + \text{fib}(n+1) \text{ for } n \geq 0 \end{aligned}$$

and that we proposed an iterative algorithm

$i, j \leftarrow 1, 1$
for $k \leftarrow 1$ **to** $n - 1$
  $i, j \leftarrow j, i + j$
return j

which may be translated into a While loop:

$i, j, k \leftarrow 1, 1, 1$
while $k < n$
  $i, j \leftarrow j, i + j$
  $k \leftarrow k + 1$
return j

Verifying Iterative Fibonacci

We shall now prove that the iterative algorithm

$i, j, k \leftarrow 1, 1, 1$
while $k < n$
    $i, j, k \leftarrow j, i + j, k + 1$
return j

correctly computes the fibonacci function. The key step is to find a loop invariant; we shall choose

$$\Phi : 1 \leq k \leq n \text{ and } j = \text{fib}(k) \text{ and } i = \text{fib}(k-1)$$

- the preamble establishes Φ since if $n \geq 1$

$$1 \leq 1 \leq n \text{ and } 1 = \text{fib}(1) \text{ and } 1 = \text{fib}(1-1)$$

- the loop terminates since eventually $k \geq n$
- correctness holds since at loop exit we have $k \geq n$ (from negation of loop guard) and $k \leq n$ (from loop invariant) and thus $k = n$ and therefore (by loop invariant) $j = \text{fib}(n)$

Verifying Iterative Fibonacci (II)

To prove that the invariant

$$\Phi : 1 \leq k \leq n \text{ and } j = \mathrm{fib}(k) \text{ and } i = \mathrm{fib}(k-1)$$

is maintained by the loop body

$$\begin{array}{l} \textbf{while } k < n \\ \quad i, j, k \leftarrow j, i+j, k+1 \end{array}$$

we need to distinguish between old and new values; we shall prime the latter, to get the equations

$$i' = j \text{ and } j' = i + j \text{ and } k' = k + 1$$

We must prove that Φ will hold also for the new values:

$$\text{Goal}: \; 1 \leq k' \leq n \text{ and } j' = \mathrm{fib}(k') \text{ and } i' = \mathrm{fib}(k'-1)$$

But this follows (since the Loop Guard says $k \leq n - 1$)

$$\begin{aligned} i' &= j = \mathrm{fib}(k) = \mathrm{fib}(k'-1) \\ j' &= i + j = \mathrm{fib}(k-1) + \mathrm{fib}(k) = \mathrm{fib}(k+1) = \mathrm{fib}(k') \\ k' &= k + 1 \leq (n-1) + 1 = n \end{aligned}$$

Developing Provably Correct Program

Recall the Square Root specification:

Precondition $x \geq 0$

Postcondition $y^2 \leq x \ \wedge \ (y+1)^2 > x$

Let us guess loop invariant as

$$\Phi : y^2 \leq x$$

- Correctness will hold for

$$\text{Loop Guard: } (y+1)^2 \leq x$$

- Φ is Established by

$$\text{Loop Preamble}: \ y \leftarrow 0$$

- Φ is Maintained by SKIP but for progress we need

$$\text{Loop Body}: \ y \leftarrow y + 1$$

We have developed, while proving correct, the algorithm

$y \leftarrow 0$
while $(y+1)^2 \leq x$
    $y \leftarrow y + 1$

Verifying Algorithm Reading From Array

Goal: find last occurrence of x in $A[1..n]$:

Precondition $x \in A[1..n]$ (thus $n \geq 1$)

Postcondition $1 \leq r \leq n$, $A[r] = x$, $x \notin A[r+1..n]$

Loop Invariant:

$$1 \leq r \leq n,\ x \in A[1..r],\ x \notin A[r+1..n]$$

- to establish, we can use the Preamble $r \leftarrow n$
- for correctness, let Loop Guard be $A[r] \neq x$
- to maintain, observe that if Loop Guard is true then

$$x \in A[1..r-1] \text{ and } x \notin A[r..n]$$

  and thus a suitable Loop Body is $r \leftarrow r - 1$.

We have developed, while proving correct, the algorithm

$r \leftarrow n$
while $A[r] \neq x$
  $r \leftarrow r - 1$
return r

Dutch National Flag Problem

Input An array of items, each having property either red, white, or blue (in addition to other properties)

Output A permutation of the items such that all red items precede all white items, which precede all blue items. Also: the number of red items, the number of white items, and the number of blue items.

Dutch National Flag, Application to Selection

To find the kth smallest element in A, let

- p be some element in A;
- those element smaller than p be considered red;
- those elements equal to p be considered white;
- those element larger than p be considered blue.

If Dutch National Flag finds r red elements, w white elements, b blue elements:

- if $k \leq r$, recursively return k'th smallest element in red partition
- if $k > r + w$, recursively return $k - r - w$'th smallest element in blue partition
- otherwise, return p

Dutch National Flag in Linear Space

1. compute the number r of red items, w of white items, b of blue items
2. create a new array, with the first r slots reserved for red items, the next w reserved for white items, and the last b reserved for blue items
3. traverse the original array, moving each item into the first available slot in the area reserved for its color.

Resource use:

- Time is linear
- Space is also linear

We would rather have an algorithm that is in-place, with items rearranged only by swapping.

Dutch National Flag, In-Place

Loop Invariant:

- $r + w + b \leq n$
- for all i with $1 \leq i \leq r$ we know that $A[i]$ is red
- for all i with $n - b < i \leq n$ we know that $A[i]$ is blue
- for all i with $n - b - w < i \leq n - b$ we know that $A[i]$ is white

To establish, we use the Preamble

$$r, b, w \leftarrow 0$$

For correctness, we use the Loop Guard

$$r + b + w < n$$

To maintain, we examine $A[n - b - w]$:

- if Red: swap it with $A[r + 1]$, and add 1 to r
- if Blue: swap it with $A[n - b]$, and add 1 to b
- if White: just add 1 to w.

Dutch National Flag Algorithm

We developed:

$\textsc{DutchFlagIter}(A[1\ldots n])$
  $r \leftarrow 0;\ w \leftarrow 0;\ b \leftarrow 0$
  while $r + w + b < n$
    $k \leftarrow n - b - w$
    if $A[k]$ is red
      $A[k] \leftrightarrow A[r+1];\ r \leftarrow r + 1$
    else if $A[k]$ is blue
      $A[k] \leftrightarrow A[n-b];\ b \leftarrow b + 1$
    else
      $w \leftarrow w + 1$
  return r, w, b

This algorithm

- runs in linear time
- is in-place
- but is not *stable*

Iterative Insertion Sort, Outer Loop

Postcondition: $A[1..n]$ is non-decreasing.

for $i \leftarrow 2$ **to** n
  $j \leftarrow i$
  while $j > 1$ and $A[j] < A[j-1]$
    $A[j] \leftrightarrow A[j-1]$; $j \leftarrow j - 1$

Invariant for outer loop:

$$1 \leq i \leq n + 1 \text{ and } A[1..i-1] \text{ non-decreasing}$$

- is established by $i \leftarrow 2$ (or $i \leftarrow 1$)
- gives correctness since at loop exit we have $i = n + 1$
- to maintain it is the task of the inner loop

Iterative Insertion Sort, Inner Loop

$j \leftarrow i$
while $j > 1$ and $A[j] < A[j-1]$
  $A[j] \leftrightarrow A[j-1]$; $j \leftarrow j-1$

must implement the local specification:

Precondition $A[1..i-1]$ is non-decreasing

Postcondition $A[1..i]$ is non-decreasing.

What is a suitable loop invariant? We could try

$$A[1..j-1] \text{ and } A[j..i] \text{ are both non-decreasing}$$

- this is established by $j \leftarrow i$
- and gives correctness since at loop exit, either
  - $j = 1$, or
  - $A[j-1] \leq A[j]$
- but though it is maintained we cannot prove it (since it allows for some infeasible situations)

Iterative Insertion Sort, Inner Loop (II)

$j \leftarrow i$
while $j > 1$ and $A[j] < A[j-1]$
  $A[j] \leftrightarrow A[j-1]$; $j \leftarrow j - 1$

must implement the local specification:

Precondition $A[1..i-1]$ is non-decreasing

Postcondition $A[1..i]$ is non-decreasing.

What is a suitable loop invariant? Let us try

$$\forall k_1, k_2 \text{ with } 1 \leq k_1 < k_2 \leq i : \text{ if } k_2 \neq j \text{ then } A[k_1] \leq A[k_2]$$

- this is established by $j \leftarrow i$
- and gives correctness since at loop exit, either
  - $j = 1$, or
  - $A[j-1] \leq A[j]$
- and one can also prove (though tricky) that it is maintained

Reasoning About Recursive Calls

Form of generic recursive algorithm:

$f(x)$
  if G
    ...
  else
    $\ldots f(y_1) \ldots f(y_n) \ldots$

- the arguments to recursive calls, $y_1 \ldots y_n$, must be in some sense smaller than x.
- we do not want to unfold the recursive calls (when to stop?)

In general, when we see a function call we

- can use its specification
- but should not inspect its implementation

Verifying Recursive Algorithm

Recall problem: find last occurrence of x in $A[1..n]$:

Precondition $x \in A[1..n]$ (thus $n \geq 1$)

Postcondition $1 \leq r \leq n$, $A[r] = x$, $x \notin A[r+1..n]$

We want to prove that the recursive implementation

$\textsc{FindLast}(A, n, x)$
    if $A[n] = x$
        return n
    else
        return $\textsc{FindLast}(A, n-1, x)$

fulfills the specification for all $n \geq 1$, and do induction in n. For the recursive call, where $x \neq A[n]$, we observe

(1): $x \in A[1..n-1]$ *(2):* $n-1 \geq 1$ *(3):* $n-1 < n$

and now inductively *(2 & 3)* infer that the recursive call fulfills its specification and since its precondition *(1)* holds also its postcondition holds: $1 \leq r \leq n-1$, $A[r] = x$, $x \notin A[r+1..n-1]$ which implies the desired postcondition.

Verifying Recursive Insertion Sort

To make $A[1..n]$ non-decreasing, we wrote

$\textsc{InsertionSort}(A[1..n])$
  if $n > 1$
    $\textsc{InsertionSort}(A[1..n-1])$
    $\textsc{InsertLast}(A[1..n])$

which we shall prove correct in two independent steps:

- prove that InsertionSort is correct assuming InsertLast meets its specification
- implement InsertLast to meet its specification.

We shall focus on the former: if $n > 1$ we can

1. inductively assume that the call $\textsc{InsertionSort}(A[1..n-1])$ produces an array A'' with $A''[1..n-1]$ non-decreasing
2. which is the precondition for the call to InsertLast and hence we can assume that it produces an array A' such that $A'[1..n]$ is non-decreasing