Ch 5.5: Program Correctness

ICS 141: Discrete Mathematics for Computer Science I

KYLE BERNEY
DEPARTMENT OF ICS, UNIVERSITY OF HAWAII AT MANOA

Program Verification

- Experimental approach:
  - Test the algorithm with sample input
  - Check whether it produces the correct output
- For many problems, it is not feasible nor realistic to test all possible inputs to the algorithm
- Ex: Sorting n elements
  - $n!$ total inputs
  - $2^{10} = 1024$ elements
  - $2^{10}! \approx 5.42 \times 10^{2639}$ total input sequences
  - For comparison, the estimated number of atoms in the observable universe is 10^{82}

Proof of Correctness

- Theoretical approach:
  - Provide a proof of correctness

1. Iterative algorithms:
   - Loop Invariants
2. Recursive algorithms:
   - Mathematical Induction

- A recursive algorithm may also include loops
  - Correctness of loops are proved using loop invariants
  - Overall recursive algorithm are proved using mathematical induction

Loop Invariants

- Definition: A loop invariant is a formal property that is (claimed to be) true at the start of each iteration of a loop.

- Must show three things about a loop invariant:

1. Initialization: It is true prior to the first iteration
2. Maintenance: If it is true before a given iteration, then it remains true before the next iteration
3. Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Loop Invariants

- Must show three things about a loop invariant:
  1. Initialization: It is true prior to the first iteration
  2. Maintenance: If it is true before a given iteration, then it remains true before the next iteration
  3. Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

- *Remark:* Notice the similarity to mathematical induction
  - Initialization $\approx$ Base case
  - Maintenance $\approx$ Inductive case
  - Unlike induction, loop invariants have a termination condition

Correctness of Insertion Sort

```
InsertionSort(A[1 . . . n])
    for j = 2 to n
        key = A[j]
        // Insert A[j] into the sorted sequence A[1 . . . j − 1]
        i = j − 1
        while i > 0 and A[i] > key
            A[i + 1] = A[i]
            i = i − 1
        A[i + 1] = key
```

- Loop Invariant:
  - At the start of each iteration of the **for** loop, the subarray $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$, but in sorted order.

Correctness of Insertion Sort

- Loop Invariant:
  - At the start of each iteration of the **for** loop, the subarray $A[1 \dots j-1]$ consists of the elements originally in $A[1 \dots j-1]$, but in sorted order.
- Initialization:
  - Prior to the first iteration, $j = 2$.
  - The subarray $A[1 \dots j-1] = A[1 \dots 1] = A[1]$ is a single element.
  - Trivially, $A[1]$ is sorted.

Correctness of Insertion Sort

- Loop Invariant:
  - At the start of each iteration of the **for** loop, the subarray $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$, but in sorted order.
- Maintenance:
  - Prior to the j-th iteration, we know that our loop invariant is true, i.e., the subarray $A[1 \ldots j-1]$ is sorted
  - In the body of the **for** loop, the elements $A[j-1], A[j-2], A[j-3]$, etc. are shifted by one position to the right, until it finds the correct position for $A[j]$.
  - Then, it inserts $A[j]$ into this position.
  - Therefore, at the start of the $(j+1)$-th iteration, the subarray $A[1 \ldots j]$ is sorted.

Correctness of Insertion Sort

- Loop Invariant:
  - At the start of each iteration of the **for** loop, the subarray $A[1 \dots j-1]$ consists of the elements originally in $A[1 \dots j-1]$, but in sorted order.
- *Remark:* Formally, another loop invariant is needed for the **while** loop, inside of the body of the **for** loop
  - For simplicity of exposition, we presented an informal argument in the maintenance step

Correctness of Insertion Sort

- Loop Invariant:
  - At the start of each iteration of the **for** loop, the subarray $A[1 \dots j-1]$ consists of the elements originally in $A[1 \dots j-1]$, but in sorted order.
- Termination:
  - The **for** loop terminates when $j = n + 1$.
  - Therefore, $A[1 \dots j-1] = A[1 \dots n]$ consists of the elements originally in $A[1 \dots n]$, but in sorted order.

Correctness of Recursive Factorial

FACTORIAL(n)
  if $n == 0$
    return 1
  return $n \cdot$ FACTORIAL($n - 1$)

- Proposition: For all non-negative integers n, FACTORIAL(n) correctly returns the value of $n!$.

Correctness of Recursive Factorial

- Proposition: For all non-negative integers n, FACTORIAL(n) correctly returns the value of $n!$.

- Proof: Let n be an arbitrary non-negative integer.
  Inductive Hypothesis: Assume inductively that for all integers k, such that $0 \leq k < n$, $P(k)$ is true. In other words, FACTORIAL(k) correctly returns the value of $k!$.
  Base Case: Assume $n = 0$.
  We know that $0! = 1$, hence, FACTORIAL(0) correctly returns 1.

Correctness of Recursive Factorial

- Proposition: For all non-negative integers n, $\text{F}\textsc{actorial}(n)$ correctly returns the value of $n!$.

- Proof:
  Inductive Case: Assume $n > 0$.
  From our inductive hypothesis, we know that for $0 \leq n - 1 < n$, $\text{F}\textsc{actorial}(n-1)$ correctly returns the value of $(n-1)!$. Therefore, $\text{F}\textsc{actorial}(n)$ correctly returns

$$
\begin{aligned}
n \cdot \text{F}\textsc{actorial}(n-1) &= n \cdot (n-1)! \\
&= n!\,. \qquad \blacksquare
\end{aligned}
$$