EECS 3101 M: Design and Analysis of Algorithms

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/3101A`
Also on Moodle

Reasoning (formally) about algorithms

- I/O specs: Needed for correctness proofs, performance analysis.
  E.g. for **sorting** in non-decreasing order:
  INPUT: $A[1 \ldots n]$ - an array of integers
  OUTPUT: a *permutation* B of A such that
  $B[1] \leq B[2] \leq \ldots \leq B[n]$

- CORRECTNESS: The algorithm satisfies the output specs for EVERY valid input.

- ANALYSIS: Compute the performance of the algorithm, e.g., in terms of running time

Correctness

- How can we show that the algorithm works correctly for all possible inputs of all possible sizes?
- Exhaustive testing not feasible.
- Analytical techniques are ~~useful~~ essential here.

Assertions

- An assertion is a statement about the state of the program at a specified point in its execution
- May be implemented in code, as an error-check
- Types:
  - Preconditions: Any assumptions that must be true about the code that follows
  - Postconditions: The statement of what must be true about the preceding code
  - Exit condition: The statement of what must be true to exit a loop or a method or program
  - Loop invariants: Some property that holds in each iteration of the loop, and is useful for proving correctness of the loop

Correctness Definition: Code Segment

- $\langle pre - condition \rangle \wedge \langle code \rangle \Rightarrow \langle post - condition \rangle$
- If the input meets the preconditions, then the output must meet the post-conditions.
- If the input does not meet the preconditions, then nothing is required.

Uses

- If the assertions can be checked automatically, correctness checking can be automated
- Caveat: undecidability issues
- EECS 3311 will teach you to do this in practice

Proving correctness - A Simple Example

Problem: find the maximum element of an array of integers

FIND-MAX(A)

```
1   // INPUT: A[1..n] - an array of integers
2   // OUTPUT: an element m of A such that m ≥ A[j],
    for all 1 ≤ j ≤ A.length
3   max = A[1]
4   for j = 2 to A.length
5         if max < A[j]
6         max = A[j]
7   return max
```

Can you think of another algorithm?

Proof by Contradiction

Proof: Suppose the algorithm is incorrect. Then for some input A, either

1. max is not an element of A or
2. A has an element $A[j]$ such that $max < A[j]$

max is initialized to and assigned to elements of A – so (1) is impossible
After the j-th iteration of the for-loop (lines 4 - 6), $max \geq A[j]$. From lines 5,6, max only increases. Therefore, upon termination, $max \geq A[j]$, which contradicts (2).

Proof by Contradiction - Remarks

- The preceding proof reasons about the whole algorithm
- It is possible to prove correctness by induction as well: this is left as an exercise for you
- What if the algorithm was very big and had many function calls, nested loops, if-then's and other standard commands?
- For example....

Proof by Contradiction - Remarks

- The preceding proof reasons about the whole algorithm
- It is possible to prove correctness by induction as well: this is left as an exercise for you
- What if the algorithm was very big and had many function calls, nested loops, if-then's and other standard commands?
- Even proving that the algorithm terminates may be non-trivial!

Need a simpler, more "modular" strategy.

Correctness Proofs for Loops

Decompose the job into checking:

- Pre-condition for the loop is true

Correctness Proofs for Loops

Decompose the job into checking:

- Pre-condition for the loop is true
- Loop Invariant holds for each iteration

Correctness Proofs for Loops

Decompose the job into checking:

- Pre-condition for the loop is true
- Loop Invariant holds for each iteration
- Termination condition is met

Correctness Proofs for Loops

Decompose the job into checking:

- Pre-condition for the loop is true
- Loop Invariant holds for each iteration
- Termination condition is met
- Upon termination the post-condition holds

Correctness Proofs for Loops

Decompose the job into checking:

- Pre-condition for the loop is true
- Loop Invariant holds for each iteration
- Termination condition is met
- Upon termination the post-condition holds
- Note the similarities with induction.

Proving correctness of FindMax with LI

FIND-MAX(A)

```
1  // INPUT: A[1..n] - an array of integers
2  // OUTPUT: an element m of A such that m ≥ A[j],
   for all 1 ≤ j ≤ A.length
3  max = A[1]
4  for j = 2 to A.length
5        if max < A[j]
6        max = A[j]
7  return max
```

What is the precondition of the loop?

Correctness of FindMax: Steps

Show that:

- Pre-condition for the loop: *max* contains $A[1]$
- Loop Invariant for each iteration:
  At the beginning of iteration j of the for loop, $j \geq 2$, *max* contains the maximum of $A[1..j-1]$
- Termination condition: $j = A.length + 1$
- Partial correctness and Termination implies post-condition: *max* is the correct maximum, i.e., of $A[1..A.length]$.

Proof of the Loop Invariant - Partial Correctness

LI: At the beginning of iteration j of the for loop, max contains the maximum of $A[1..j-1]$

- Initialization: max contains $A[1]$, so $LI(1)$ is true
- Maintenance: For $j \geq 2$, assume $LI(j)$; so before iteration j, max = maximum of $A[1..j-1]$
  - Case 1: $A[j]$ = maximum of $A[1..j]$. In lines 5-6, max is set to $A[j]$
  - Case 2: $A[j]$ is not the maximum of $A[1..j]$, so the maximum of $A[1..j]$ is in $A[1..j-1]$. By our assumption, max already has this value, and max is unchanged in this iteration.

Proof of the Loop Invariant - Termination

- Termination: When the loop terminates, $j = A.length + 1$ (WHY?)

- Partial correctness and Termination imply the post-condition:
  LI: At the beginning of iteration j of the for loop, max contains the maximum of $A[1..j-1]$
  At termination: $j = A.length + 1$
  Therefore, max contains the maximum of $A[1..A.length]$
  Therefore, it is the correct maximum

Loop Invariants - Summary

We must show three things about loop invariants:

- Initialization – it is true prior to the first iteration
- Maintenance – if it is true before an iteration, it remains true before the next iteration
- Termination – when loop terminates the invariant gives a useful property to show the correctness of the algorithm

$$\boxed{\text{Partial Correctness} \wedge \text{Termination} \Rightarrow \text{Correctness}}$$

What about more complex algorithms?

INSERTION-SORT(A)

```
1   for j = 2 to A.length
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[1..j-1].
4       i = j - 1
5       while i > 0 and A[i] > key
6           A[i+1] = A[i]
7           i = i - 1
8       A[i+1] = key
```

- How to formulate loop invariants (2 loops, so 2 LI needed)
- How to prove correctness?

Correctness of Insertion Sort

INSERTION-SORT(A)

```
1   for j = 2 to A.length
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[1..j-1].
4       i = j - 1
5       while i > 0 and A[i] > key
6           A[i+1] = A[i]
7           i = i - 1
8       A[i+1] = key
```

- What is a good loop invariant?
- It is easy to write a loop invariant if you understand what the algorithm does.

Forming Loop Invariants

- LI for outer loop:

Forming Loop Invariants

- LI for outer loop: LI1: at the start of for loop iteration j, $A[1\,..\,j-1]$ consists of elements originally in $A[1\,..\,j-1]$ but in sorted order

Forming Loop Invariants

- LI for outer loop: LI1: at the start of for loop iteration j, $A[1\,..\,j-1]$ consists of elements originally in $A[1\,..\,j-1]$ but in sorted order
- The inner while loop moves elements finds the highest position k so that $A[k] \leq key$ and moves $A[k+1\,..\,j-1]$ one position right without changing their order.
  Then, in the outer loop, the key element is inserted into $A[k+1]$ so that $A[k] \leq A[k+1] \leq A[k+2]$.

Forming Loop Invariants

- LI for outer loop: LI1: at the start of for loop iteration j, $A[1\,..\,j-1]$ consists of elements originally in $A[1\,..\,j-1]$ but in sorted order
- The inner while loop moves elements finds the highest position k so that $A[k] \leq key$ and moves $A[k+1\,..\,j-1]$ one position right without changing their order.
  Then, in the outer loop, the key element is inserted into $A[k+1]$ so that $A[k] \leq A[k+1] \leq A[k+2]$.
  LI2: at the start of for loop iteration i, key contains $A[j]$ and $A[i+2\,..\,j]$ consists of elements originally in $A[i+1\,..\,j-1]$ but moved one spot to the right and the original $A[i+1] \geq key$

Correctness of LI2: Initialization

Putting $i = j - 1$ in LI2, we get the statement
key contains $A[j]$ and $A[j+1\,..\,j]$ consists of elements originally in $A[j\,..\,j-1]$ but moved one spot to the right and the original $A[j] \geq key$
This is true, because of line 2.

Correctness of LI2: Maintenance

LI2: at the start of for loop iteration i, *key* contains $A[j]$ and $A[i+2\,..\,j]$ consists of elements originally in $A[i+1\,..\,j-1]$ but moved one spot to the right and the original $A[i+1] \geq key$

We assume that LI2 holds before iteration i, the loop body executes and will show that LI2 holds before iteration $i-1$ (the loop index is decreasing).
Since the loop body executes we know that $i > 0$ and $A[i] > key$. Also, *key* contains $A[j]$. The loop body moves (copies) $A[i]$ to $A[i+1]$, and decrements i. So LI2 holds before iteration $i-1$.

Correctness of LI2: Termination and Correctness

The loop terminates when $i = 0$ or $A[i] \leq key$

Case 1: $i = 0$: Plugging $i = 0$ into LI2 we get:
at the start of for loop iteration i, key contains $A[j]$ and $A[2\,..\,j]$ consists of elements originally in $A[1\,..\,j-1]$ but moved one spot to the right and the original $A[1] \geq key$
Thus in this case the loop found the correct place $k = 0$.

Case 2: $A[i] \leq key$. Plugging in this value of i, we get
$A[i+2\,..\,j]$ consists of elements originally in $A[i+1\,..\,j-1]$ but moved one spot to the right and the original $A[i+1] \geq key$
So in both cases, the loop does what it was meant to do and it is therefore correct.

Proving Correctness of LI1: Initialization

LI1: at the start of for loop iteration j, $A[1\,..\,j-1]$ consists of elements originally in $A[1\,..\,j-1]$ but in sorted order

Before the first iteration, $j = 2$, LI1 trivially holds because $A[1\,..\,1]$ is a sorted array

Proving Correctness of LI1: Maintenance

Since the inner loop is correct, we know it moves elements finds the highest position k so that $A[k] \leq key$ and then moves $A[k \,..\, j-1]$ one position right without changing their order.

Then, in line 8 in the outer loop, the *key* element is inserted into $A[k+1]$ so that $A[k] \leq A[k+1] \leq A[k+2]$.

Since LI1 held before the current iteration, $A[1 \,..\, j-1]$ was sorted and $A[j]$ was inserted in the correct place, so $A[1 \,..\, j]$ consists of elements originally in $A[1 \,..\, j]$ but in sorted order.

Proving Correctness of LI1: Termination and Correctness

The loop terminates with $j = A.length + 1$

Plugging this value of j into LI1 we get
$A[1\,..\,A.length]$ consists of elements originally in
$A[1\,..\,A.length]$ but in sorted order

This is what the loop (program) was meant to do and it is therefore proven correct.

More on Correctness of Algorithms: Exponentiation

Let us prove the correctness of the following algorithm for computing the n-th power of a given real number.

```
POWER(y, z)
1   // return y^z where y ∈ R, z ∈ ℕ
2   x = 1
3   while z > 0
4         if ODD(z)
5               x = x * y
6         z = ⌊z/2⌋
7         y = y^2
8   return x
```

Formulating the Loop Invariant

```
POWER(y, z)
1   x = 1
2   while z > 0
3         if ODD(z)
4               x = x * y
5         z = ⌊z/2⌋
6         y = y^2
7   return x
```

How does the algorithm proceed?

The loop invariant for this code segment needs to capture the entire state of the program, viz., variables x, y, z. Otherwise proving the invariant may be difficult.

Formulating the Loop Invariant

```
POWER(y, z)
1  x = 1
2  while z > 0
3        if ODD(z)
4              x = x * y
5        z = ⌊z/2⌋
6        y = y^2
7  return x
```

Since y, z are changed in the program, let y_0, z_0 denote the initial values of y, z respectively. We want to express the fact that in each iteration the loop stores in x, y_0 raised to the power "the last $i-1$ digits of z_0". The part in quotes can be compactly expressed as $z_0 \mod 2^{i-1}$.

Formulating the Loop Invariant

- Suppose the number of bits in z is n. Suppose also that the initial values of x, y, z are x_0, y_0, z_0 respectively. We see that the while loop goes from $i = 1$ to n. After studying the program the following loop invariant seems reasonable:
  LI: Before iteration i, $z = \lfloor \frac{z_0}{2^{i-1}} \rfloor$, $x = y_0^{z_0 \mod 2^{i-1}}$ and $y = y_0^{2^{i-1}}$.

- We prove Initialization, correctness and termination

Proving Correctness: Initialization

Initialization: Before the first iteration, the invariant yields $z = \lfloor \frac{z_0}{2^0} \rfloor = z_0$, $x = y_0^{z_0 \bmod 2^0} = y_0^0 = 1$ and $y = y_0^{2^0} = y_0$. All these values match the code – x is initialized to 1 in line 1 and y, z are unchanged.

Proving Correctness: Maintenance

Assume that the loop invariant holds at the beginning of iteration i. We want to show that it holds at the beginning of iteration $i+1$. So before the current iteration, we have $z = \lfloor \frac{z_0}{2^{i-1}} \rfloor$, $x = y_0^{z_0 \mod 2^{i-1}}$ and $y = y_0^{2^{i-1}}$.
Lines 4 and 5 (potentially) change x. Notice that $z_0 \mod 2^i = 2^i + z_0 \mod 2^{i-1}$ if the i^{th} bit of z_0 is a 1; otherwise $z_0 \mod 2^i = z_0 \mod 2^{i-1}$. Notice also that the i^{th} bit of z_0 is a 1 iff $z = \lfloor \frac{z_0}{2^{i-1}} \rfloor$ is odd. Therefore if the i^{th} bit of z_0 is a 0 x is unchanged. This is what lines 4,5 do.
Otherwise,
$x = x * y = y_0^{z_0 \mod 2^{i-1}} * y_0^{2^{i-1}} = y_0^{2^i + z_0 \mod 2^{i-1}} = y_0^{z_0 \mod 2^i}$,
which is exactly the loop invariant for y at the beginning of the next iteration.

Proving Correctness: Maintenance

Line 6 changes z to $\lfloor z/2 \rfloor = \lfloor \lfloor \frac{z_0}{2^{i-1}} \rfloor /2 \rfloor = \lfloor \frac{z_0}{2^i} \rfloor$, and line 7 changes y to $(y_0^{2^{i-1}})^2 = y_0^{2^i}$. Thus the maintenance proof is complete.

Proving Correctness: Termination and Correctness

The loop terminates with $i = n + 1$.

Plugging this value of i into the invariant we get $x = y_0^{z_0} \mod 2^{n+1-1} = y_0^{z_0}$.

This is what the program was meant to do and it is therefore proven correct.

Note that the final values of y, z are not really important, since x is what the program returns.

More on Correctness of Algorithms: Selection Sort

I/O specs: same as insertion sort

Algorithm: Given an array A of n integers, sort them by repetitively selecting the smallest among the yet unselected integers.

Q: Is this precise enough?

More on Correctness of Algorithms: Selection Sort

I/O specs: same as insertion sort

Algorithm: Given an array A of n integers, sort them by repetitively selecting the smallest among the yet unselected integers.

Q: Is this precise enough?

Swap the smallest integer with the integer currently in the place where the smallest integer should go.

Loop invariant

Algorithm: Given an array A of n integers, sort them by repetitively selecting the smallest among the yet unselected integers.

LI: at the beginning of the j^{th} iteration,
the smallest $j-1$ values are sorted in ascending order in locations $A[1\,..\,j-1]$

Q: Is this precise enough?

Loop invariant

Algorithm: Given an array A of n integers, sort them by repetitively selecting the smallest among the yet unselected integers.

LI: at the beginning of the j^{th} iteration,
the smallest $j-1$ values are sorted in ascending order in locations $A[1\,..\,j-1]$

Q: Is this precise enough?

and the rest are in locations $A[n-j\,..\,n]$.
Exercise: prove the correctness of selection sort

More on Correctness of Algorithms: Binary Search

Precondition: A an array of sorted integers, *key* an integer
Postcondition: Index in which *key* is found, if it exists in the array

Algorithmic idea:
1. Cut sublist in half
2. Determine which half the key would be in
3. Keep that half.

Note: LI must not assume that the element is present in the list. So it should say something like
If the key is contained in the original list, then the key is contained in the sublist

Pinning down the algorithm: Binary Search

- If $key \leq A[mid]$ then key is in the left half
  Else key is in the right half
- Maintain sublist from i to j
- Which element is mid? Must be consistent
- Subtle issue: Suppose we use
  $mid = \lfloor \frac{i+j}{2} \rfloor$
  If $key \geq A[mid]$ then $i = mid$
  Else $j = mid$
  A = $\boxed{10}\boxed{20}\boxed{30}$, $key = 20$