CS 483 - Data Structures and Algorithm Analysis

Some notes on correctness proofs

R. Paul Wiegand

George Mason University, Department of Computer Science

February 20, 2006

Outline

When is an Algorithm "*Correct*"?

- Recall the definition of a *correct* algorithm: One returns the correct solution for every valid instance of a problem
- There are a variety of ways to prove correctness
- Correctness proofs are easy for some algorithms, hard for others
- But there's a standard way to prove correctness for many common algorithms using loops or recursion: Identify and prove a *loop invariance property*
- There is a good discussion of this on pp. 17–19 of *Introduction to Algorithms (2nd edition)*, by Cormen *et al.* [These notes come from that text]

The Loop Invariance Property

- We need to define a key property of the data manipulated by the main loop of an algorithm
    - The property must help us understand why the algorithm is correct
    - We must show that the property holds in the initial case, is maintained each iteration, and that when the loop terminates the property yields correctness
- Determining this property in general can be difficult
- But for simple, common algorithms the property is often the key defining feature of the algorithm

From Loop Invariance to Correctness

Initialization	— The loop invariance must be true *prior* to the first iteration of the loop
Maintenance	— If the property holds prior to an iteration of the loop, it must *still* hold after the iteration is complete
Termination	— When the loop terminates, the invariant provides a useful property that helps demonstrate that the algorithm is correct

- To prove correctness, we must prove the above about the loop invariance property
- The first two pieces are similar to induction. When they hold, the loop invariant is true prior to every iteration of the loop
- The *termination* is where is differs from induction and is the most important piece. We are not showing that the loop invariant holds *ad infinitum*, but rather that it results in a correct answer after a *finite* number of steps

Reminder: SELECTIONSORT

SELECTIONSORT($A[0 \ldots n-1]$)

```
for $i \longleftarrow 0$ to $n-2$ do
  $min \longleftarrow i$
  for $j \longleftarrow i+1$ to $n-1$ do
    if $A[j] < A[min]$, $min \longleftarrow j$
  SWAP($A[i], A[min]$)
```

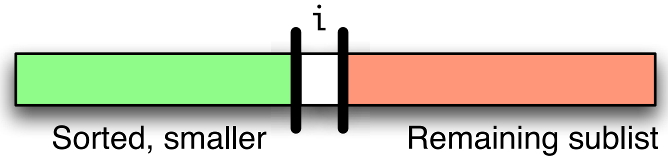
Loop Invariant:
At the start of each iteration of the outermost `for` loop, the sublist $A[0 \ldots i-1]$ consists of the $i-1$ smallest elements originally in A in sorted order. The sublist $A[i \ldots n-1]$ contain the remaining elements in A.

What Does the Loop Invariant Mean?

Essentially, the loop invariant says that at each step, the data set can be divided into two parts:

- The part to the left of i is a sorted sublist from elements in A
- The part from i to the right on which the algorithm is still working

Loop Invariant:
At the start of each iteration of the outermost `for` loop, the sublist $A[0 \ldots i-1]$ consists of the $i-1$ smallest elements originally in A in sorted order. The sublist $A[i \ldots n-1]$ contain the remaining elements in A.



Proving the PARTITION Loop Invariant

Initialization — At the first iteration, $i = 0$, the sublist to the left of i is empty. It is reasonable to say that an empty sublist is ordered and obeys the loop invariant.

Maintenance — At the start of any arbitrary iteration, no element from 0 to $i - 1$ can ever be disturbed again. During the step, the inner `for` loop will find the smallest element in $A[i+1 \ldots n-1]$ and will swap it for the $A[i]$ element. This element is swapped into the i^{th} position. The new $A[i]$ element *cannot* be smaller than any element in $A[0 \ldots i-1]$ because the loop invariant is true prior to the start of the iteration, so it will simultaneously be the smallest element from i to the right and no smaller than any element to its left. The loop invariant is preserved.

Termination — When the last iteration of the algorithm terminates, the loop counter will be on $n - 2$. If the i^{th} element is smaller, it will be exchanged with the $n - 1^{st}$ element. Given that the sublist to the left of i is already sorted and neither $A[n-2]$ nor $A[n-1]$ can be smaller than any item in $A[0 \ldots n-2]$ because of the loop invariant, the combined list will be in sorted order. If the i^{th} element is *not* smaller, the list is already sorted.

Wait ... What Did We Just Prove?

- Recall: The sorting problem is to take a list of unordered items and order them by value
- We showed that the SelectionSort algorithm will, at each step, preserve the property that items to the left of the main index are in sorted order and not greater than any item to the right.
- If this property holds at initialization, is maintained each step, and terminates properly, SelectionSort *must* be a *correct* implementation of an algorithm for solving the sorting problem

Reminder: QUICKSORT and PARTITION

QUICKSORT($A[l \ldots r]$)

```
if $l > r$
  $s \longleftarrow$ PARTITION($A[l \ldots r]$)
  QUICKSORT($A[l \ldots s-1]$)
  QUICKSORT($A[s+1 \ldots r]$)
```

PARTITION($A[l \ldots r]$)

```
$p \longleftarrow A[l]$
$i \longleftarrow l$
$j \longleftarrow r+1$
repeat
  repeat $i++$ until $p \geq A[i]$
  repeat $j--$ until $p \leq A[j]$
  SWAP($A[i], A[j]$)
until $i \geq j$
SWAP($A[i], A[j]$)
SWAP($A[l], A[j]$)
```

Loop Invariant (for Partition):
At the start of the first repeat loop in the PARTITION function, the following holds for any array index k:

- $k \in (l, i] \Rightarrow A[k] \leq p$
- $k \in [j, r] \Rightarrow A[k] \geq p$

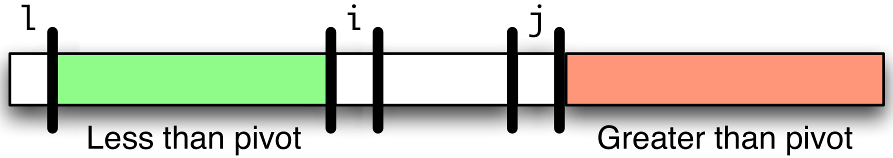
What Does the Loop Invariant Mean?

Essentially, the loop invariant says that at each step, the data set can be divided into three parts:

- The part to the left of i (except l) is never greater than the pivot value
- The part to the right of j is never less than the pivot value
- The part in the middle on which the algorithm is still working

Loop Invariant (for Partition):
At the start of the first `repeat` loop in the PARTITION function, the following holds for any array index k:

- $k \in (l, i] \Rightarrow A[k] \leq p$
- $k \in [j, r] \Rightarrow A[k] \geq p$



Proving the Partition Loop Invariant

Initialization — Prior to the first iteration of the the loop, $i = l$ and $j = r + 1$. There are no values at $r + 1$ and we excuse the case where $i = l$.

Maintenance — The invariant holds prior to the step so everything to the left of i (except l) obeys the first condition of the variant and everything to the right of j obeys the second condition. The first `repeat` loop inside the main loop will skip past all consecutive values from that point forward that are less than p, so we only need to consider the case when $A[i] \leq p$. Likewise with $A[j \ldots r]$, we only need to consider when $A[j] \geq p$. Under that condition, the two are swapped — the previous $A[i]$ value now resides in the right-hand portion of the data list, and the previous $A[j]$ value now resides in the left-hand portion. The invariant is preserved

Termination — The exception, on the final iteration, occurs when $i \geq j$. This can only occur once because of the `until` condition of the main loop and is corrected immediately after the loop terminates. The pivot point is moved to the partition point by swapping it with the $A[j]$ value. Since j is now less than or equal to i, it resides in the left-hand set ... so $A[j] \leq p$, obeying the first condition of the variant.

Wait ... What Did We Just Prove?

- Recall: We want the PARTITION function to select a pivot value and divide the sublist such that all values to the left of that pivot point are no greater than the pivot and all values to the right are no less than the pivot
- We just proved that the algorithm implemented in the book is *correct*: For all valid input lists, it performs the above function correctly
- We *did not* prove that QUICKSORT is correct ...
- But to prove that QUICKSORT is correct, we *must* show that PARTITION is correct