CMPU241 Analysis of Algorithms

Proofs of Algorithm Correctness
Using Loop Invariants

To prove an algorithm is correct, you need to know how the algorithm transforms input to output.

E.g., an algorithm to find the maximum value element in a set of totally ordered data is correct if its output is the largest number in the set.

What outcome is correct if you are running a sorting algorithm on a set of comparable data elements?

All the elements are in some specified ordering, commonly ascending order.

What outcome is correct if you are running a sorting algorithm on a set of comparable data elements?

All the elements are in some sorted order (increasing or decreasing).

A loop invariant generally refers to the actions inside a loop, starts by showing that the initial condition or basis fits some criteria, and argues that consecutive iterations of the loop uphold these criteria. We will generally use proof by induction on the number of loop iterations.

Unlike most proofs by induction, algorithms terminate, resulting in the entire data set upholding the loop invariant to produce the correct result.

FindMax(A[1...n])

INPUT: An array A of *n* comparable items

OUTPUT: The value of the maximum item in the array

1. max = A[1]
2. **for** (k = 2; k <=n; k++)
3. **if** (A[k] > max)
4. max = A[k]
5. **return** max

Loop invariant? **Let k be the position of the current max in the array A. At the start of iteration k of the for loop, max contains the largest value in A[1...k-1].**

Base case: **k = 2. Since max is set to equal A[1] before the first iteration, max holds the largest value in A[1...k-1] = A[1...1] = A[1].**

Inductive hypothesis: **Assume the invariant holds through the beginning of the iteration where 1<= k < n, when max is the largest value in A[1...k-1].**

Inductive Step (Maintenance): **Show the invariant holds at the end of iteration k, the beginning of iteration k+1. Show that max is the largest value in A[1...k].**

By the inductive hypothesis, we know that max is the largest value in A[1...k-1] at the start of iteration k. In iteration k, the maximum element in A[1...k] is found by comparing max to the value in A[k]. Due to the total ordering on comparable items, max is either unchanged in this iteration or it is set to the value in A[k]. In either case, at the beginning of iteration k+1, max is the largest value in A[1...k].

Termination: **The for loop ends when k = n+1. At that point, max is the largest value in A[1...n]. Therefore at the end of the algorithm, the value of the maximum item in A[1...n] is returned and the algorithm is correct.**

QED

Proving Correctness—Insertion Sort

<u>Loop invariant</u>:
Let j be the position of the current key in the array A.
At the start of each iteration of the for loop, the subarray A[1...j-1] consists of the elements originally in A[1...j-1], but in sorted order.

```
Insertion-Sort(A)
for j = 2 to A.length
  key = A[j]
  i = j - 1
  while i>0 and A[i]>key
      A[i+1] = A[i]
      i = i - 1
  A[i+1] = key
```

We need to show...

1. ...that the loop invariant is true at the start of the first iteration (base case or *initialization*),
2. ... the invariant remains true for the next k < n iterations (inductive hypothesis (IH) or *maintenance*), and
3. ...the algorithm has the correct result when the loop terminates.

Proving Correctness—Base Case

<u>Loop invariant</u>:
Let j be the position of the current key in the array A.
At the start of each iteration of the for loop, the subarray A[1...j-1] consists of the elements originally in A[1...j-1], but in sorted order.

```
Insertion-Sort(A)
for j = 2 to A.length
  key = A[j]
  i = j - 1
  while i>0 and A[i]>key
      A[i+1] = A[i]
      i = i - 1
  A[i+1] = key
```

<u>Base case (initialization)</u>: When j = 2, A[1...j-1] has a single element and is therefore trivially sorted.

Proving Correctness—Inductive Hypothesis

<u>Loop invariant</u>:
Let j be the position of the current key in the array A.
At the start of each iteration of the for loop, the subarray A[1...j-1] consists of the elements originally in A[1...j-1], but in sorted order.

```
Insertion-Sort(A)
for j = 2 to A.length
  key = A[j]
  i = j - 1
  while i>0 and A[i]>key
      A[i+1] = A[i]
      i = i - 1
  A[i+1] = key
```

<u>Inductive Hypothesis (IH)</u>: Assume the invariant holds through the beginning of the iteration where j = k < n.

Proving Correctness—Inductive Step

<u>Loop invariant</u>:
Let j be the position of the current key in the array A.
At the start of each iteration of the for loop, the subarray A[1...j-1] consists of the elements originally in A[1...j-1], but in sorted order.

```
Insertion-Sort(A)
for j = 2 to A.length
  key = A[j]
  i = j - 1
  while i>0 and A[i]>key
      A[i+1] = A[i]
      i = i - 1
  A[i+1] = key
```

<u>Inductive Step (Maintenance)</u>: Show the invariant holds at the end of the iteration when j = k.

Proving Correctness—Inductive Step

Loop invariant:
Let j be the position of the current key in the array A.
At the start of each iteration of the for loop, the subarray A[1…j-1] consists of the elements originally in A[1…j-1], but in sorted order.

```
Insertion-Sort(A)
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i>0 and A[i]>key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

When j = k, *key* = A[k]. By the IHOP, we know that the subarray A[1…k-1] is in sorted order. In iteration k, A[k-1], A[k-2], A[k-3] and so on are each moved one position to the right until either a value less than *key* is found or until k-1 values have been shifted right, when the value of *key* is inserted. In this iteration, *key* will be inserted in the right position in the values A[1…k], so at the beginning of iteration k+1, the subarray A[1…k] will contain only the elements that were originally in A[1…k], but in sorted order. Therefore, the loop invariant holds at the start of iteration k+1.

Proving Correctness—Termination

Loop invariant:
Let j be the position of the current key in the array A.
At the start of each iteration of the for loop, the subarray A[1…j-1] consists of the elements originally in A[1…j-1], but in sorted order.

```
Insertion-Sort(A)
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i>0 and A[i]>key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

Termination: The for loop ends when j = n+1. By the IHOP, we have that the subarray A[1…n] is in sorted order. Therefore, the entire array is sorted and the algorithm is correct. QED

Proving Correctness—BubbleSort

Loop invariant:
At the start of iteration i of the outer for loop, the subarray A[1…i – 1] is in sorted order with the (i–1) smallest elements of A in positions 1…i–1.

```
BubbleSort(A)      // A.length = n
(assume problem statement = that of InsertionSort)
for i = 1 to n – 1
    for j = n downto i+1
        if A[j] < A[j-1]
            swap A[j] with A[j-1]
```

We need to show…

1. … the loop invariant is true at the start of the first iteration (base case or *initialization*),
2. … the invariant remains true for the next k <= n iterations (inductive hypothesis (IHOP) or *maintenance*), and
3. …the algorithm has the correct result when the loop terminates.

Proving Correctness—BubbleSort Basis

Loop invariant:
At the start of iteration i, the subarray A[1 . . . i – 1] is in sorted order with the (i–1) smallest elements of A in positions 1…i–1.

```
BubbleSort(A)
for i = 1 to n – 1
    for j = n downto i+1
        if A[j] < A[j-1]
            swap A[j] with A[j-1]
```

- **Basis (initialization): In the first iteration, i = 1 and the subarray A[1 . . . 0] is trivially sorted with the 0 smallest elements.**

Proving Correctness—BubbleSort IHOP

Loop invariant:
At the start of iteration i, the subarray A[1 . . . i – 1] is in sorted order with the (i–1) smallest elements of A in positions 1…i–1.

```
BubbleSort(A)
for i = 1 to n – 1
    for j = n downto i+1
        if A[j] < A[j-1]
            swap A[j] with A[j-1]
```

Inductive Hypothesis: Assume the invariant holds through the beginning of the iteration where i = k < n, so subarray A[1 . . . k-1] is sorted and contains the k-1 smallest elements of A.

Now, we have to argue why the k^{th} pass through the inner for loop upholds the invariant for the beginning of iteration k+1.

Proving Correctness—BubbleSort inductive step

Loop invariant:
At the start of iteration i, the subarray A[1 . . . i – 1] is in sorted order with the (i–1) smallest elements of A in positions 1…i–1.

```
BubbleSort(A)
for i = 1 to n – 1
    for j = n downto i+1
        if A[j] < A[j-1]
            swap A[j] with A[j-1]
```

- **Inductive step: In iteration i=k, the inner for loop starts at the n^{th} element of A and compares it with the $(n\text{-}1)^{st}$ element; if the n^{th} element is smaller than the $(n\text{-}1)^{st}$ element, they are swapped, otherwise nothing changes.**

  This compare-and-swap or no swap continues, with smaller elements "bubbling down" as they are swapped with larger elements, until the inner for loop decrements to index k+1, when it compares the elements in positions k + 1 and k.

Proving Correctness—BubbleSort Maintenance

Loop invariant:
At the start of iteration i, the subarray A[1 . . . i − 1] is in sorted order with the (i−1) smallest elements of A in positions 1…i−1.

```
BubbleSort(A)
1.   for i  = 1 to n – 1
2.        for j = n downto i+1
3.             if A[j] < A[j-1]
4.                 swap A[j] with A[j-1]
```

- **In this iteration, the smallest element of subarray A[k + 1…n] is found and swapped all the way to position k. At this point, the smallest element of A[k…n] is in position k, and so the elements in subarray A[1…k] are in sorted order, with the k smallest elements in positions 1…k at the start of iteration k + 1, as required.**

Proving Correctness—BubbleSort Termination

Loop invariant:
At the start of iteration i, the subarray A[1 . . . i − 1] is in sorted order with the (i−1) smallest elements of A in positions 1…i−1.

```
BubbleSort(A)
1.   for i  = 1 to n – 1
2.        for j = n downto i+1
3.             if A[j] < A[j-1]
4.                 swap A[j] with A[j-1]
```

- **When the loop terminates, i = n. The invariant says that subarray A[1…n-1] is in sorted order and the n-1 smallest elements of A are in positions 1…n-1. So the element in position n must be the largest in A. Therefore, the entire array is sorted in ascending order and the algorithm is correct.   QED**