Algorithm Correctness

Rangaballav Pradhan

Asst. Professor, Dept. of CSE, ITER,

SOA Deemed to be University

Algorithm Correctness

- Correctness of an algorithm corresponds to the fact that the algorithm is correct with respect to a given specification.
- Functional correctness ensures that, for each input the algorithm produces the expected output.
- An algorithm is called **Totally correct** for the given specification if and only if for any correct input data it halts and returns the correct output.
- An algorithm is **Partially correct** if satisfies the following condition:
    - If the algorithm receiving correct input data stops then its result is correct
- Correctness of an algorithm can be verified using proofs
- Total correctness is not decidable

Total Correctness Proof

- A proof of total correctness of an algorithm usually follows 2 separate steps:

  1. To prove that the algorithm always stops for correct input data (stop property)

  2. To prove that the algorithm is partially correct

- Stop property is usually easier to prove

Algorithm Correctness

Example: computing the sum of array of numbers

```
sum(array, len){
        sum = 0
        i = 0
        do
                sum += array[i]
                i++
while(i)
        return sum
}
```

Is this algorithm partially correct?

Is it also totally correct?

Algorithm Correctness Proof

- How to prove that an algorithm is correct?

For any algorithm, we must prove that it always returns the desired output for all legal instances of the problem.

- Proof by:

    1. Counterexample (indirect proof)
    2. Induction (direct proof)
    3. Loop Invariant

Other approaches:

- proof by cases/enumeration
- proof by contradiction
- proof by contrapositive

Proof by Counterexample

- The best way to prove that an algorithm is incorrect is to produce an instance in which it yields an incorrect answer. Such instances are called counter-examples.
- Searching for counterexamples is the best way to disprove the correctness of some propositions especially, if the proof seems hard or tricky.
- Identify a case for which the algorithm is NOT true
- Sometimes a counterexample is just easy to see, and can shortcut a proof
- If a counterexample is hard to find, a proof might be easier

Proof by Counterexample

- Good counter-examples have two important properties:
- Verifiability – To demonstrate that a particular instance is a counter-example to a particular algorithm, you must be able to

  (1) calculate what answer your algorithm will give in this instance, and

  (2) display a better answer so as to prove the algorithm didn’t find it.
- Simplicity – Good counter-examples have all unnecessary details boiled away. They make clear exactly why the proposed algorithm fails. Once a counterexample has been found, it is worth simplifying it down to its essence.

Proof by Counterexample

- **Prove or disprove:** $\lceil x + y \rceil = \lceil x \rceil + \lceil y \rceil$.
  - Proof by counterexample: $x = \frac{1}{2}$ and $y = \frac{1}{2}$
- **Prove or disprove:** "Every positive integer is the sum of two squares of integers"
  - Proof by counterexample: 3
- **Prove or disprove:** $\forall x \forall y (xy \geq x)$ (over all integers)
  - Proof by counterexample: $x = -1, y = 3; xy = -3; -3 \ngeq -1$

Proof by Counterexample

- Searching for counterexamples is the best way to disprove the correctness of some things. While searching for a good counterexample,
- Think about small examples
- Think exhaustively
- Think about examples on trivial cases
- Think about extreme examples (big or small)

Proof by Induction

- Failure to find a counterexample to a given algorithm does not mean “it is obvious" that the algorithm is correct.
- Mathematical induction is a very useful method for proving the correctness of recursive algorithms.
- Proof by Induction involves the following steps:

  1. Prove the formula for the smallest number that can be used in the given statement. (base case)

  2. Assume it's true for an arbitrary number n.

  3. Use the previous steps to prove that it's true for the next number n + 1.

Proof by Induction

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof:

Does it hold true for $n = 1$?

$1 = \frac{1(1+1)}{2}$ ✓

Assume it works for n ✓

Prove that it's true when n is replaced by $n + 1$

Proof by Induction

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$1 + 2 + 3 + \ldots (n-1) + n = \frac{n(n+1)}{2}$$

$$1 + 2 + 3 + \ldots + ((n+1) - 1) + (n+1) = \frac{(n+1)[(n+1)+1]}{2}$$

$$1 + 2 + 3 + \ldots + n + (n+1) = \frac{(n+1)(n+2)}{2}$$

$$(1 + 2 + 3 + \ldots + n) + (n+1) = \frac{(n+1)(n+2)}{2}$$

$$\frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$$

Proof by Induction

$$\frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

$$\frac{n(n+1)+2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

$$\frac{(n+1)(n+2)}{2} = \frac{(n+1)(n+2)}{2} \qquad \checkmark$$

We've proved that the formula holds for $n+1$.

Proof by Induction

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof:

- Does it hold true for $n = 1$?
  $1 = \frac{1(1+1)}{2}$ ✓
- Assume it works for n ✓
- Prove that it's true when n is replaced by $n + 1$ ✓

Example: Factorial Function: FAC(n)

Prove that algorithm fac(n) returns n! for all nonnegative integers $n \geq 0$.

procedure fac(n: nonnegative integer)

  if (n = 0) then **return** 1

  else return $n * fac(n - 1)$

Basis step: for n = 0

The proof is trivial. Factorial (0) = 1

Inductive Hypothesis: Fac (n) produces k! for any $0 \leq k < n$

Inductive steps: For n=k+1,

Fac (n) returns $(k+1) * fac(k) = (k+1) * k! = (k+1)!$

Example: SQUARE Function: SQ(n)

```
S ← 0
i ← 0
while i < n
    S ← S + n
    i ← i + 1
return S
```

To prove: S_i = n $*$ n for any i = n

Basis Step: For n = 1, S = 0 +1 = 1

Induction Hypothesis: For an arbitrary value k, S_k = k $*$ n and i = k hold after going through the loop k times.

Inductive Step: When the loop is entered (k + 1)th time, S = k $*$ n and i = k at the beginning of the loop.

Inside the loop, S_{k+1} ← k $*$ n + n and i ← k + 1 producing S_{k+1} = (k + 1) $*$ n and i = k+1.

Thus S = k $*$ n and i = k hold for any natural number k.

Loop Invariant

- A loop invariant is a condition that is necessarily true immediately before and immediately after each iteration of a loop.
- A loop invariant is some predicate (condition) that holds for every iteration of the loop.
- The loop invariant must be true:
  - before the loop starts
  - before each iteration of the loop
  - after the loop terminates
- although it can temporarily be false during the body of the loop

Proof by Loop Invariant

- Built off of proof by induction.
- Useful for algorithms that loop.
- Formally: Define a Loop Invariant and then prove:
    1. Initialization
    2. Maintenance
    3. Termination
- Informally:
    1. Find p, a loop invariant
    2. Show the base case for p
    3. Use induction to show the rest.

Proof by Loop Invariant

- After finding the loop invariant:

❑ Initialization

  ❑ Prior to the loop initiating, does the property hold?

❑ Maintenance

  ❑ After each loop iteration, does the property still hold, given the initialization properties?

❑ Termination

  ❑ After the loop terminates, does the property still hold? And for what data?

Example: Linear Search

LinearSearch(A, v)

for $j = 1$ to $A.length$:

  if $A[j] == v$:

    return j

return NIL

Example: Linear Search

```
LinearSearch(A, v)
1  for j = 1 to A.length:
2      if A[j] == v:
3          return j
4      return NIL
```

Loop Invariant . At the start of each iteration of the for loop on line 1, the subarray $A[1 : j-1]$ does not contain the value v

Example: Linear Search

```
LinearSearch(A, v)
for j = 1 to A.length:
        if A[j] == v:
                return j
        return NIL
```

Initialization Prior to the first iteration, the array $A[1 : j - 1]$ is empty ($j == 1$). That (empty) subarray does not contain the value v.

Example: Linear Search

LinearSearch(A, v)

1 **for** $j = 1$ to $A.length$:
2   **if** $A[j] == v$:
3   **return** j
4   **return** NIL

Maintenance Line 2 checks whether $A[j]$ is the desired value (v). If it is, the algorithm will return j, thereby terminating and producing the correct behavior (the index of value v is returned, if v is present). If $A[j] \neq v$, then the loop invariant holds at the end of the loop (the subarray $A[1 : j]$ does not contain the value v).

Example: Linear Search

```
LinearSearch(A, v)
1   for j = 1 to A.length:
2       if A[j] == v:
3           return j
4       return NIL
```

Termination The for loop on line 1 terminates when $j > A.length$ (that is, n). Because each iteration of a for loop increments j by 1, then $j = n + 1$. The loop invariant states that the value is not present in the subarray of $A[1 : j - 1]$. Substituting $n + 1$ for j, we have $A[1 : n]$. Therefore, the value is not present in the original array A and the algorithm returns NIL.

Example: Sum of n numbers

```
Algorithm sum(n)
    Input: a non-negative integer n
    Output: the sum 1 + 2 + … + n
    sum ← 0
    i ← 1
    while i ≤ n
        // Invariant: sum = 1 + 2 + … + (i - 1)
        sum ← sum + i
        i ← i + 1
    return sum
```

Example: Sum of n numbers

1. Initialization: The loop invariant holds initially since $sum = 0$ and $i = 1$ at this point. (The empty sum is zero.)

2. Maintenance: Assuming the invariant holds before the ith iteration, it will be true also after this iteration since the loop adds i to the sum, and increments i by one.

3. Termination: When the loop is just about to terminate, the invariant states that $sum = 1 + 2 + \ldots + n$, just what's needed for the algorithm to be correct.

Example: Maximum in an array

Algorithm max(A)

Input: an array A storing n integers

Output: the largest element in A

$max \leftarrow A[\ 1\ ]$

for $i = 2$ to n

if $A[\ i\ ] > max$ then

$max \leftarrow A[\ i\ ]$

return max

LI: Before each iteration of the for loop, the variable *max* contains the maximum element from the subarray $A[1 \ldots i\text{-}1]$

Example: Maximum in an array

1. Initialization: The loop invariant holds initially since (i – 1) = 1 and there is only one element in the array (*A*[1]) and that is the maximum.

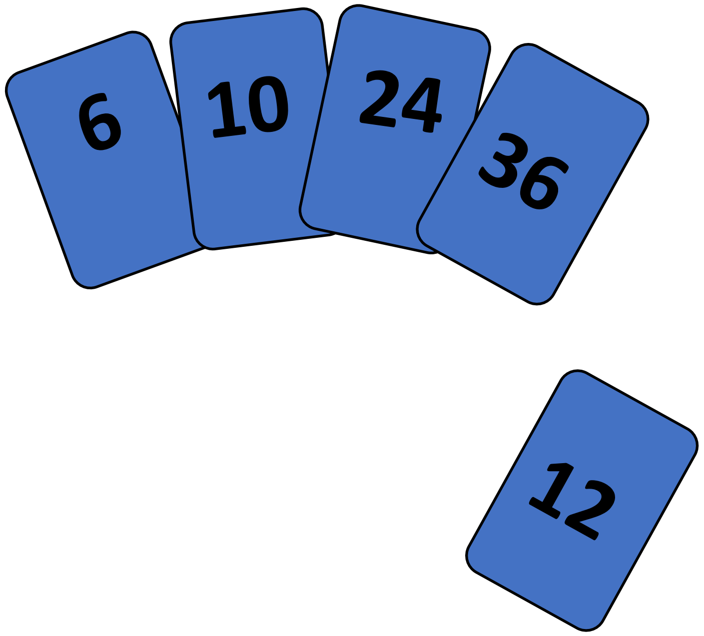
2. Maintenance: Lets say the invariant holds for any iteration *k*. i.e. *max* = max(A[1], A[2],…,A[*k*-1]). Now the *if* condition checks the k_th element in the next iteration and if it is greater than the current maximum, if changes the maximum to *max* = A[*k*]. So, before the (*k*+1)th iteration, *max* = max(A[1], A[2],…,A[*k*]).

3. Termination: When the loop terminates (at *i* = *n* + 1), the invariant states that *max* = max(A[1], A[2],…,A[*n*]), just what's needed for the algorithm to be correct.

Example: Insertion Sort

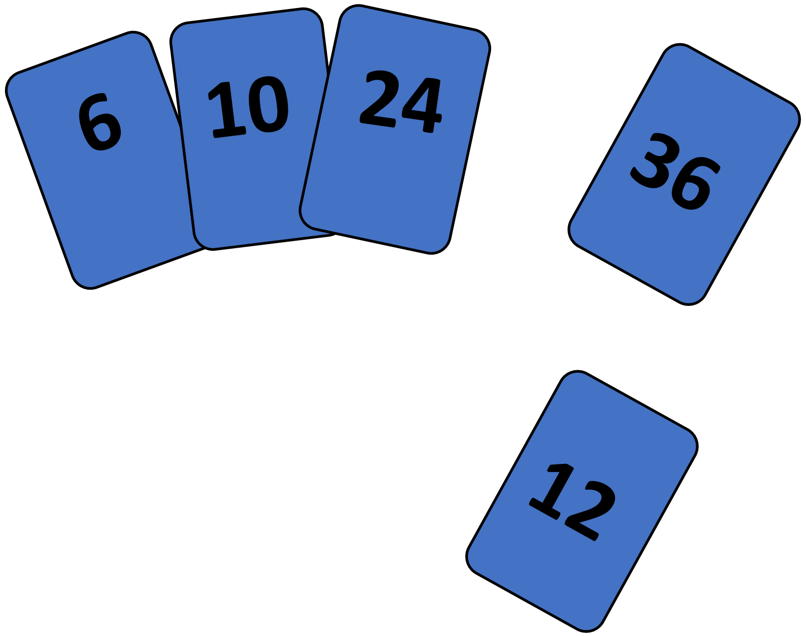
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Example: Insertion Sort

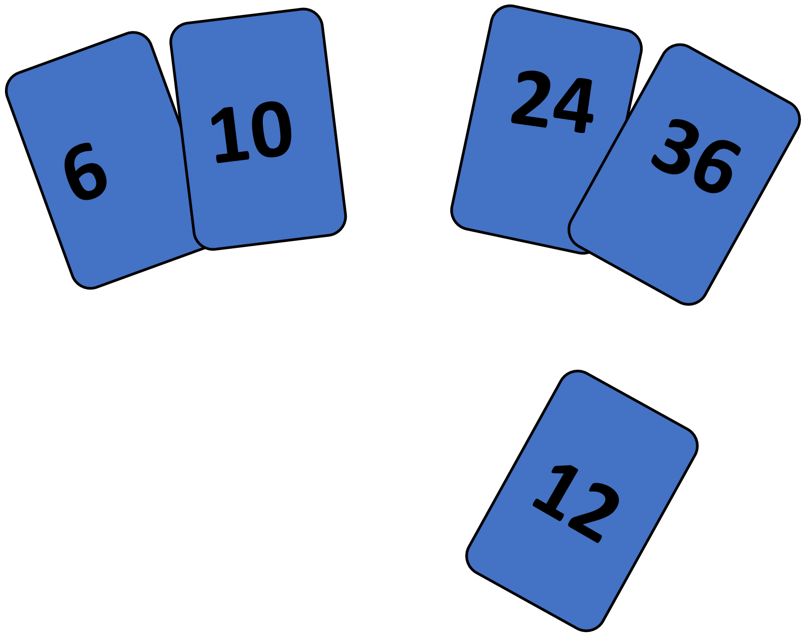


To insert 12, we need to make room for it by moving first 36 and then 24.

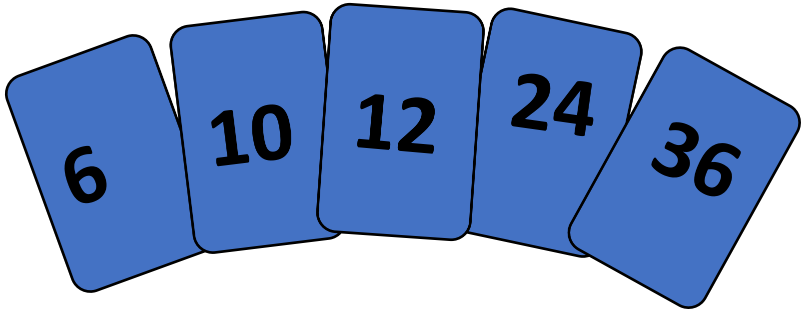
Example: Insertion Sort



Example: Insertion Sort



Example: Insertion Sort



Example: Insertion Sort

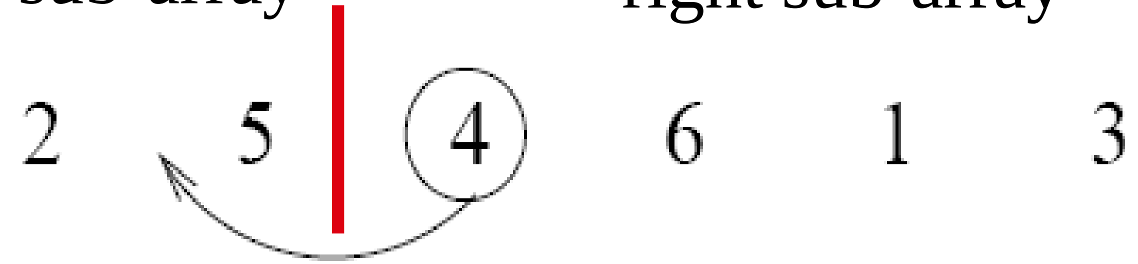
input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

left sub-array

right sub-array



sorted

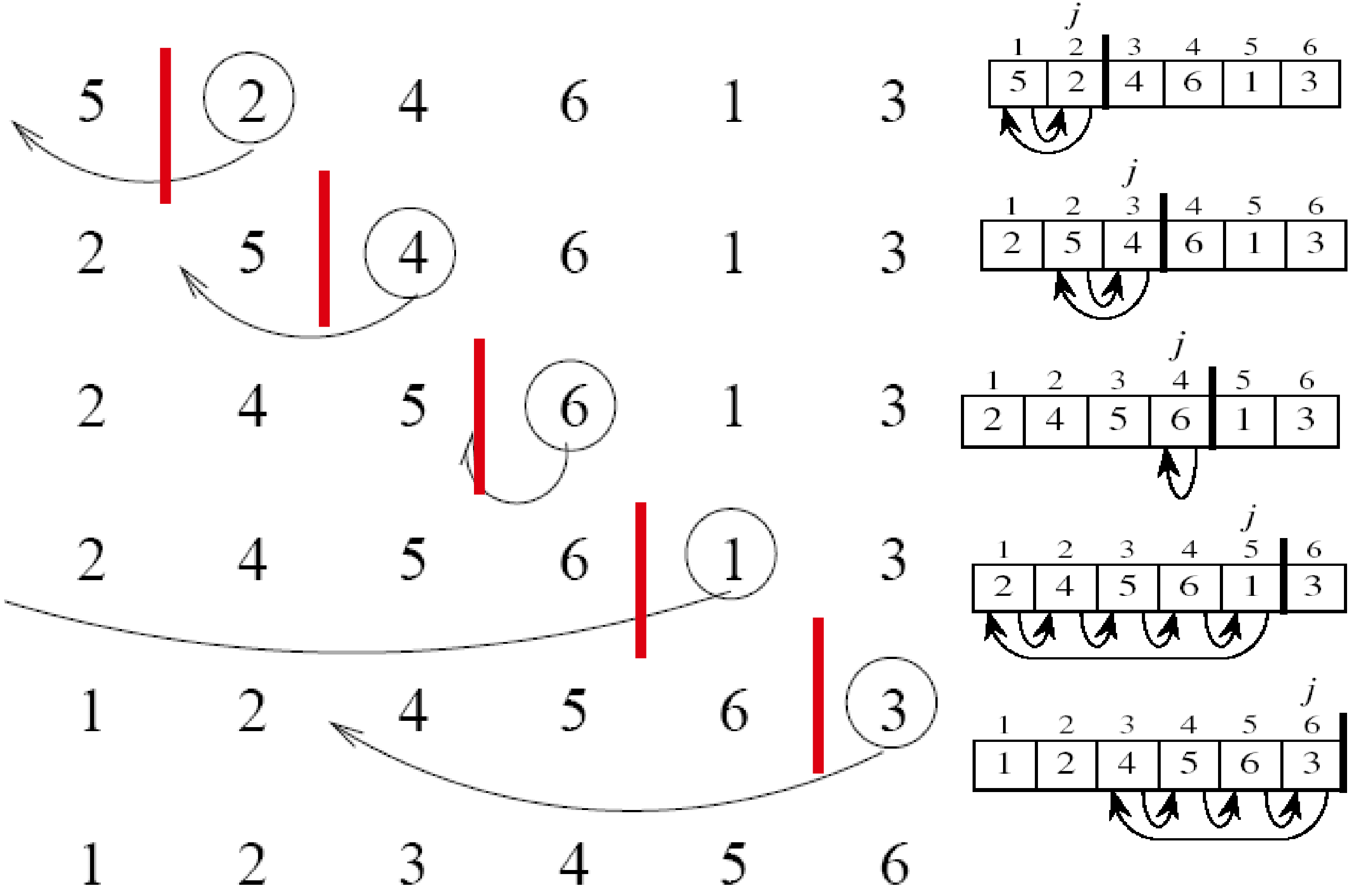
unsorted

Example: Insertion Sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    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
```

Example: Insertion Sort

```
INSERTION-SORT(A)
  for j = 2 to A.length
      key = A[j]
      // Insert A[j] into the sorte
      i = j - 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key
```



Example: 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
```

At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 \mathinner{..} j-1]$ consists of the elements originally in $A[1 \mathinner{..} j-1]$, but in sorted order.

Example: Insertion Sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    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
```

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$. The subarray $A[1..j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Example: 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
```

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1..j]$ then consists of the elements originally in $A[1..j]$, but in sorted order. Incrementing j for the next iteration of the **for** loop then preserves the loop invariant.

Example: Insertion Sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    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
```

Termination: Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

Example: Bubble Sort

BubbleSort(A)

```
for i = 1 to A.length − 1
    for j = A.length to i + 1
        if A[j] < A[j − 1]
            Swap(A[j], A[j − 1]
```

Example: Bubble Sort

BubbleSort(A)

1 **for** $i = 1$ to $A.length - 1$
2   **for** $j = A.length$ **to** $i + 1$
3   **if** $A[j] < A[j-1]$
4   SWAP($A[j], A[j-1]$)

Invariant At the start of each iteration of the **for** loop on line 1, the subarray $A[1 : i-1]$ is sorted

Example: Bubble Sort

BubbleSort(A)

for $i = 1$ to $A.length - 1$
  for $j = A.length$ **to** $i + 1$
    if $A[j] < A[j-1]$
      SWAP($A[j]$, $A[j-1]$)

Initialization Prior to the first iteration, the array $A[1:i-1]$ is empty ($i = 1$). That (empty) subarray is sorted by definition.

Example: Bubble Sort

BubbleSort(A)

1 **for** $i = 1$ to $A.length - 1$
2   **for** $j = A.length$ **to** $i + 1$
3   **if** $A[j] < A[j-1]$
4   SWAP($A[j], A[j-1]$)

Maintenance Given the guarantees of the inner loop, at the end of each iteration of the **for** loop at line 1, the value at $A[i]$ is the smallest value in the range $A[i : A.range]$. Since the values in $A[1 : i-1]$ were sorted and were less than the value in $A[i]$, the values in the range $A[1 : i]$ are sorted.

Example: Bubble Sort

BubbleSort(A)

1 **for** $i = 1$ to $A.length - 1$
2   **for** $j = A.length$ **to** $i + 1$
3   **if** $A[j] < A[j - 1]$
4   SWAP($A[j], A[j - 1]$)

Termination The **for** loop at line 1 ends when i equals $A.length$. Based on the maintenance proof, this means that all values in $A[1 : A.length - 1]$ are sorted and less than the value at $A[length]$. So, by definition, the values in $A[1 : A.length]$ are sorted.

Example: Bubble Sort (Inner loop)

```
BubbleSort(A)
for i = 1 to A.length - 1
    for j = A.length to i + 1
        if A[j] < A[j - 1]
            SWAP(A[j], A[j - 1])
```

Example: Bubble Sort (Inner loop)

BubbleSort(A)

1 **for** $i = 1$ to $A.length - 1$
2   **for** $j = A.length$ **to** $i + 1$
3   **if** $A[j] < A[j-1]$
4   SWAP($A[j], A[j-1]$)

Invariant At the start of each iteration of the **for** loop on line 2, the value at location $A[j]$ is the smallest value in the subrange from $A[j : A.length]$

Example: Bubble Sort (Inner loop)

```
BubbleSort(A)
for i = 1 to A.length − 1
    for j = A.length to i + 1
        if A[j] < A[j − 1]
            SWAP(A[j], A[j − 1])
```

Initialization Prior to the first iteration, $j = A.length$. The subarray $A[j : A.length]$ contains a single value ($A[j]$) and the value at $A[j]$ is (trivially) the smallest value in the range from $A[j : A.length]$)

Example: Bubble Sort (Inner loop)

```
BubbleSort(A)
1 for i = 1 to A.length − 1
2     for j = A.length to i + 1
3         if A[j] < A[j − 1]
4             SWAP(A[j], A[j − 1]
```

Maintenance The **if** statement on line 3 compares the elements at $A[j]$ and $A[j-1]$, swapping $A[j]$ into $A[j-1]$ if it is the lower value and leaving them in place, if not. Given the initial condition that the value in $A[j]$ was the smallest value in the range $A[j : A.length]$, this means the value in $A[j-1]$ is now the smallest value in the range $A[j-1 : A.length]$. This also means that every value in the subarray $A[j : A.length]$ is greater than the value at $A[j-1]$.

Example: Bubble Sort (Inner loop)

BubbleSort(A)

1 **for** $i = 1$ to $A.length - 1$
2   **for** $j = A.length$ **to** $i + 1$
3   **if** $A[j] < A[j-1]$
4   SWAP($A[j], A[j-1]$)

Termination 2 The **for** loop on line 2 terminates when $j = i$ and given the Maintenance property, this means that the value at $A[i]$ (which is $A[j]$) will be the smallest value in the range $A[i : A.range]$ ($A[j : A.range]$)