

Correctness of Algorithm

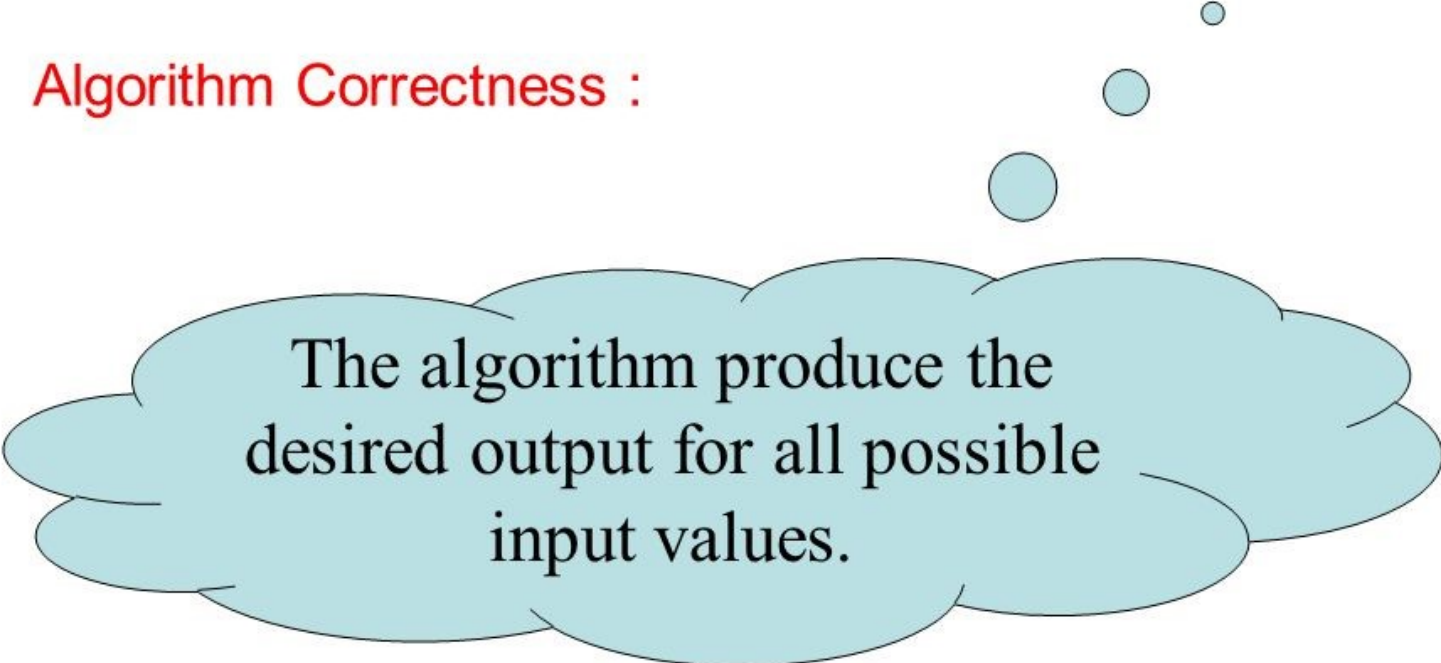
Dr. Bibhudatta Sahoo

Communication & Computing Group

CS215, Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Algorithm Correctness :



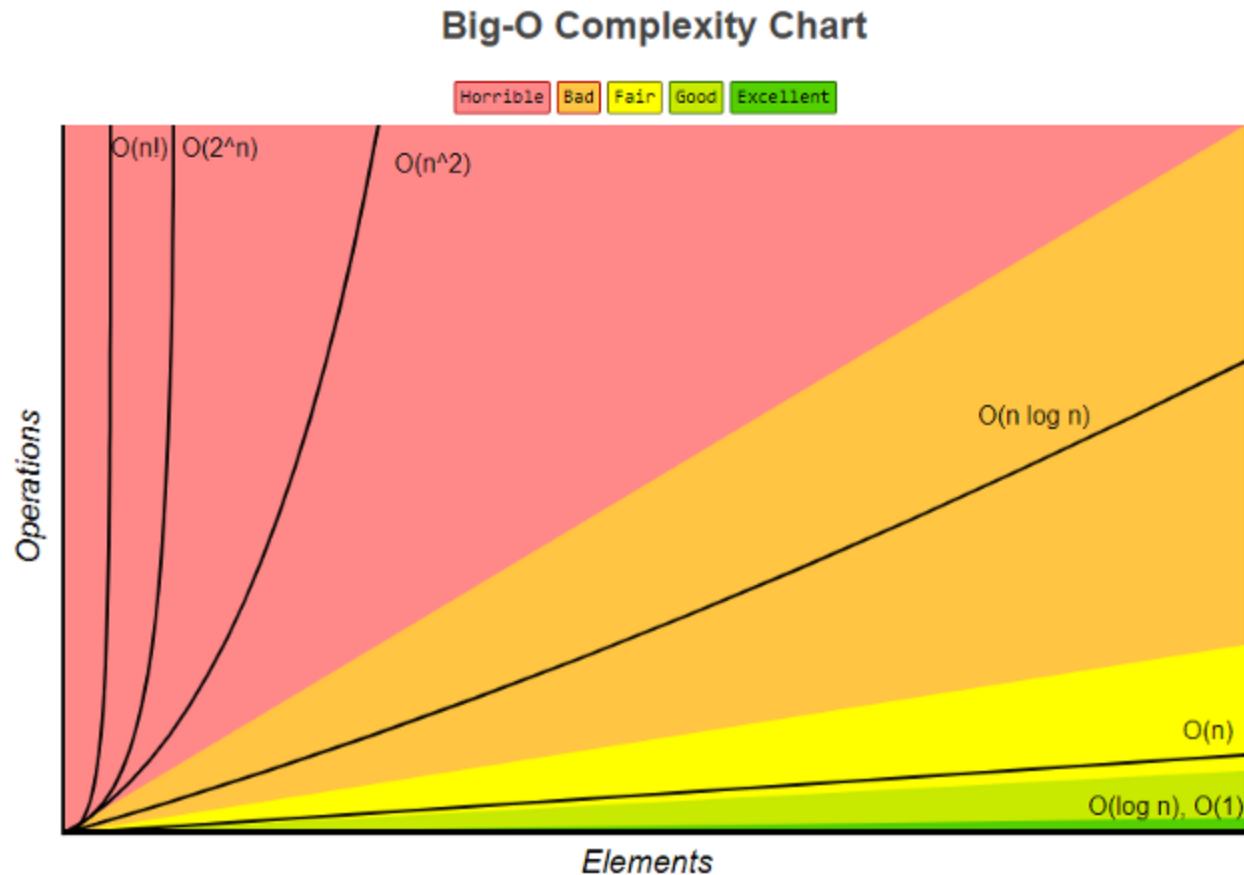
The algorithm produce the
desired output for all possible
input values.

Algorithm ?

- Algorithm is a method/idea/procedure to solving a problem.
- An algorithm is independent of the implementation environment and resources(programming language and hardware).
- An algorithm is efficient when it has an adequate execution performance. This means: **an efficient algorithm is when it solves a problem without taking long time or consuming a great amount of memory**(but remember these amounts for time and consuming memory depends on the type and greatness of the problem that we are trying to solve).

Time Complexity

- The more the number of instructions of an algorithm is used/executed then bigger its time complexity tends to be.*



Time Complexity

- *The more the number of instructions of an algorithm is used/executed then bigger its time complexity tends to be.*
- If a program runs in the order $O(1)$, it means that it has a constant time and never grow its time of execution or the number of used instructions.
- If a program runs in the $O(n)$ order, that means that given an N size input, the time complexity of that program will be N . It depends directly on the input size.
- If a program runs in $O(n^2)$, like our above example, means that the number of iterations tends to repeat times the square of the input size.

Correctness of an algorithm

- In theoretical computer science, **correctness** of an **algorithm** is asserted when it is said that the **algorithm** is correct with respect to a **specification**.
- A **fully correct algorithm** is when it's implementation can solve any instance of the problem that the solution is proposed, that include inconvenient(but possible) and big(arbitrary big) inputs(instances of the problem that our program takes as parameter/input).
- Functional **correctness** refers to the input-output behaviour of the **algorithm** (i.e., for each input it produces the expected output)

Total Correctness of Algorithm

- **Definition:** An algorithm which for any correct input data: (i) **stops** and (ii) **returns correct output** is called **totally correct** for the given specification.

These split into 2 sub-properties in the definition above.

- **correct input data** is the data which satisfies the initial condition of the specification
- **correct output data** is the data which satisfies the final condition of the specification

Partial Correctness of Algorithm

- Usually, it is convenient to separately check whether an algorithm stops and then the remaining part. This remaining part of correctness is called Partial Correctness of algorithm
- **Definition:** An algorithm is **partially correct** if satisfies the following condition:
 - ❑ If the algorithm receiving **correct** input data **stops** then its result is correct.
 - ❑ Partial correctness does not guarantee that the algorithm ever stops.

Verification of algorithms correctness

Objective is to verify if an algorithms really solves the problem for which it is designed?

Approaches for algorithms correctness verification

Experimental analysis (testing).

Formal analysis (proving).

Basic steps in algorithms correctness verification

Experimental analysis (testing).

- We test the algorithm for different instances of the problem (for different input data).
- The main advantage of this approach is its simplicity while the main disadvantage is the fact that testing cannot cover always all possible instances of input data (it is difficult to know how much testing is enough).
- However the experimental analysis allows sometimes to identify situations when the algorithm doesn't work.

Basic steps in algorithms correctness verification

Formal analysis (proving).

- The aim of the formal analysis is to prove that the algorithm works for any instance of data input.
- The main advantage of this approach is that if it is rigourously applied it guarantee the correctness of the algorithm.
- The main disadvantage is the difficulty of finding a proof, mainly for complex algorithms.
- In this case the algorithm is decomposed in subalgorithms and the analysis is focused on these (simpler) subalgorithms.
- On the other hand the formal approach could lead to a better understanding of the algorithms.

Formal Analysis

- Formal rules of logic to show that an algorithm meets its specification.

The main steps in the formal analysis of the correctness of an algorithm are:

- Identification of the properties of **input** data (the so-called problem's **preconditions**).
- Identification of the properties which must be satisfied by the **output** data (the so-called problem's **post-conditions**).
- Proving that starting from the preconditions and executing the actions specified in the algorithms one obtains the post-conditions.

Algorithm Correctness : (or how to prove programs are correct

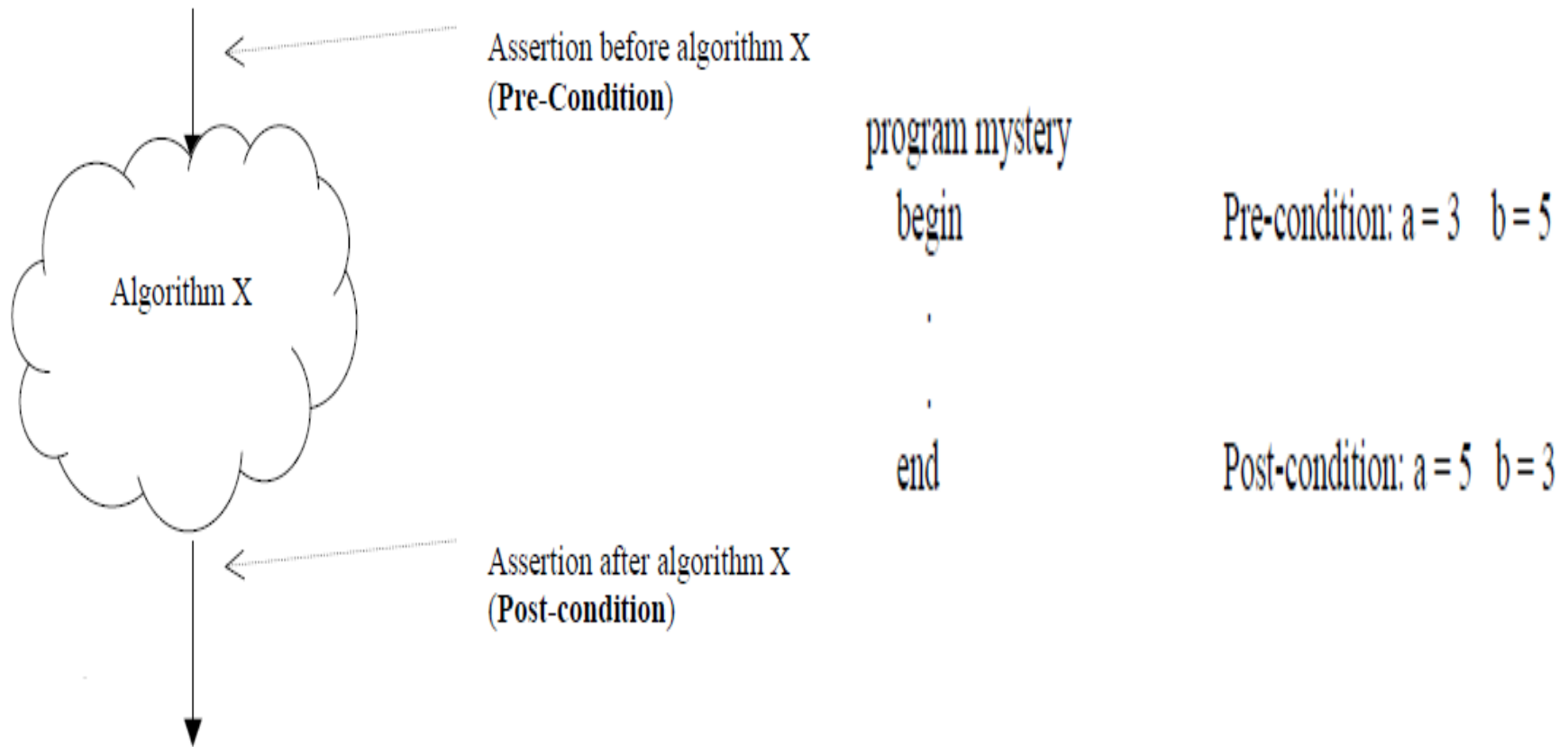
Pre-condition of the algorithm: a predicate that describes the initial state (before execution)

Post-condition of the algorithm: a predicate that describes the final state (after execution)

The algorithm is correct if it can be proved that if the pre-condition is true, the post-condition must be true.

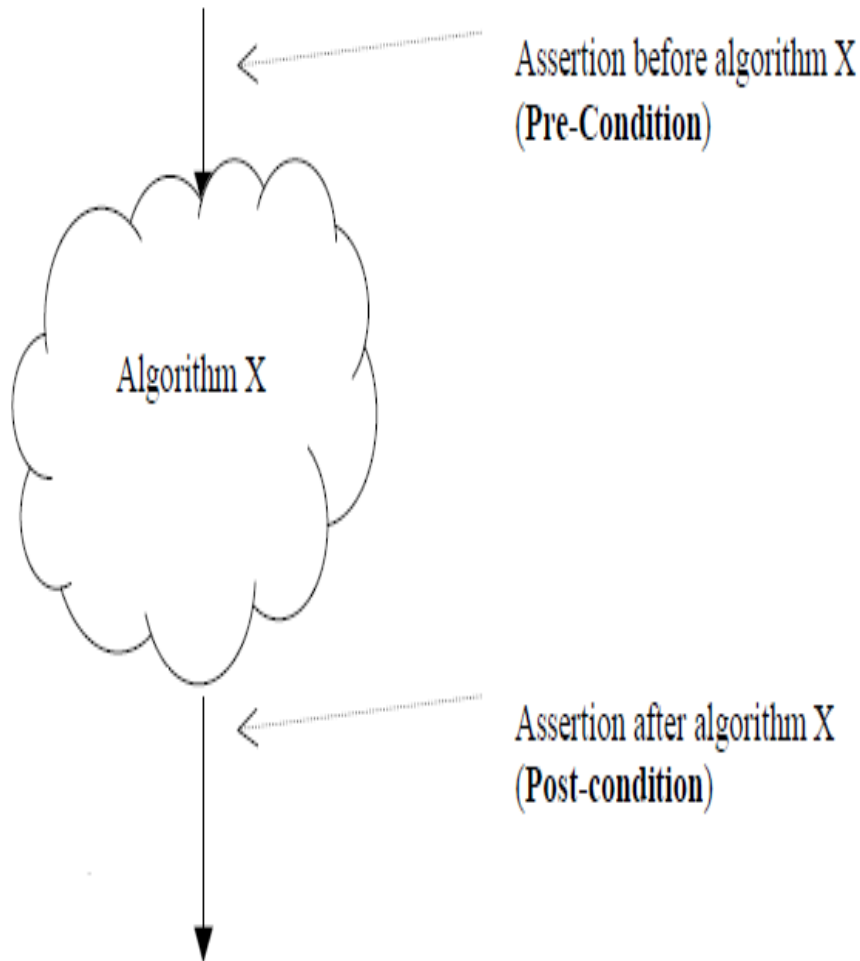
Hence, Pre-condition \Rightarrow Post-condition

Algorithm Correctness



- *The algorithm is correct if it can be proved that if the pre-condition is true, the post-condition must be true; **Pre-condition \Rightarrow Post-condition***

Algorithm Correctness



```
x = 2
z = x + y
if (y > 0)
    z = z + 1
else
    z = 0
```

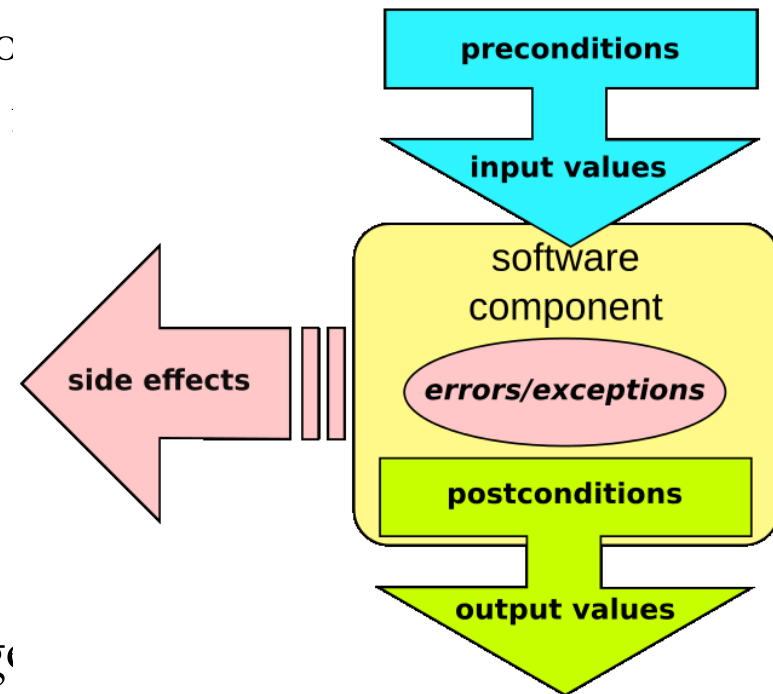
Pre-condition: $y = 6$

Post-condition: $z = 9$

What does an algorithm ?

An algorithm is described by:

- Input data
- Output data
- **Preconditions**: specifies restrictions on the input
- **Postconditions**: specifies what is the result



Example: Binary Search(a, n, x)

- Input data: a:array of integer; x:integer
- Output data: found: boolean;
- **Precondition**: a is sorted in ascending order
- **Postcondition**: found is true if x is in a, and found is false otherwise

Correct algorithms

- An algorithm is correct if:
 - for **any correct** input data:
 - it **stops** and
 - it produces **correct output**.
 - Correct input data: satisfies precondition
 - Correct output data: satisfies post condition

Proving correctness

- An algorithm = a list of actions
- *Proving that an algorithm is totally correct:*
 1. *Proving that it will terminate*
 2. *Proving that the list of actions applied to the precondition imply the postcondition*
- 1. This is easy to prove for simple sequential algorithms
- 2. This can be complicated to prove for repetitive algorithms (containing loops or recursivity)
 - use techniques based on loop invariants and *induction*

Example: a sequential algorithm

```
1. Swap1 (x, y) :  
2.     aux := x  
3.     x := y  
4.     y := aux
```

Precondition:

$x = a$ and $y = b$

Postcondition:

$x = b$ and $y = a$

Proof: *the list of actions applied to the precondition imply the postcondition*

1. Precondition:
 $x = a$ and $y = b$
2. $aux := x \Rightarrow aux = a$
3. $x := y \Rightarrow x = b$
4. $y := aux \Rightarrow y = a$
5. $x = b$ and $y = a$ is the Postcondition

Example : a repetitive algorithm

1. Algorithm Sum_of_N_numbers
2. Input: a , an array of N numbers
3. Output: s , the sum of the N numbers in a
4. $s := 0$;
5. $k := 0$;
6. While ($k < N$) do
7. $k := k + 1$;
8. $s := s + a[k]$;
9. end

Proof: the list of actions applied to the precondition imply the postcondition

but: we cannot enumerate all the actions in case of a repetitive algorithm !

We use techniques based on loop invariants and induction

Loop invariants

Loop invariants

- A loop invariant is a logical predicate such that: if it is satisfied before entering any single iteration of the loop then it is also satisfied after the iteration.
- The loop's *invariant* is a statement about what is true before you begin the body of the loop and after iteration of the loop-body.
- Put another way, the loop-invariant **must** be true each time you go to check the loop condition; it is true the first time you check to see if you go into the loop, and still true the last time you check and “skip” the loop.

Loop Invariants

- A loop invariant is associated with a given loop of an algorithm, and it is a formal statement about the relationship among variables of the algorithm such that:
 - ❑ [**Initialization**] It is true prior to the first iteration of the loop
 - ❑ [**Maintenance**] If it is true before an iteration of the loop, it remains true before the next iteration
 - ❑ [**Termination**] When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

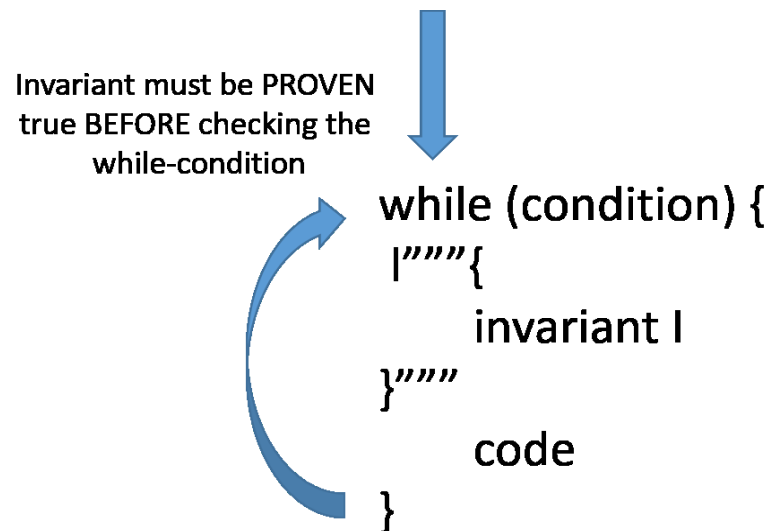
Example: Loop invariant for Sum of n numbers

```
1. Algorithm Sum_of_N_numbers
2. Input: a, an array of N numbers
3. Output: s, the sum of the N numbers in a
4. s:=0;
5. k:=0;
6. While (k<N) do
7.   k:=k+1;
8.   s:=s+a[k];
9. end
```

Loop invariant = induction hypothesis: at step k, s holds the sum of the first k numbers

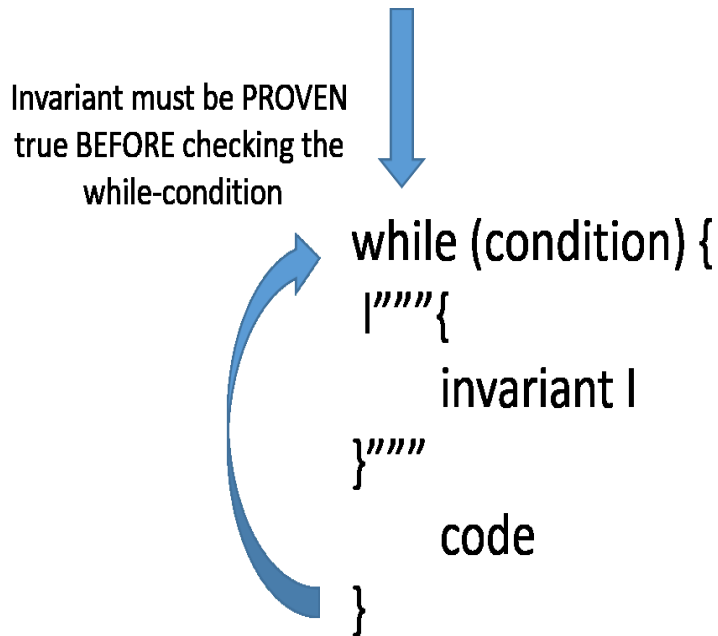
Using loop invariants in proofs

- We must show the following 3 things about a loop invariant:
 1. **Initialization:** It is true prior to the first iteration of the loop.
 2. **Maintenance:** If it is true before an iteration of the loop, 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.



Using loop invariants in proofs

- In order to analyze loop behavior we will need to know: what is true before the loop executes, what is done inside the loop and what true after the loop, regardless of how many times the loop executes.
- At the end of the loop, the **conjunction** of the negation of the loop-condition and property called the loop-invariant are true.



Example: Proving the correctness of the Sum algorithm (1)

- Induction hypothesis: $S = \text{sum of the first } k \text{ numbers}$

1. *Initialization: The hypothesis is true at the beginning of the loop:*

Before the first iteration: $k=0$, $S=0$. The first 0 numbers have sum zero (there are no numbers) \Rightarrow hypothesis true before entering the loop

```
1. Algorithm Sum_of_N_numbers
2. Input: a, an array of N numbers
3. Output: s, the sum of the N numbers in a
4. s:=0;
5. k:=0;
6. While (k<N) do
7.   k:=k+1;
8.   s:=s+a[k];
9. end
```

Example: Proving the correctness of the Sum algorithm (2)

- Induction hypothesis: $S =$ sum of the first k numbers
- 2. *Maintenance: If hypothesis is true before step k , then it will be true before step $k+1$ (immediately after step k is finished)*

We assume that it is true at beginning of step k : “ S is the sum of the first k numbers”

We have to prove that after executing step k , at the beginning of step $k+1$: “ S is the sum of the first $k+1$ numbers”

We calculate the value of S at the end of this step

$k := k+1, s := s + a[k+1] \Rightarrow s$ is the sum of the first $k+1$ numbers

Example: Proving the correctness of the Sum algorithm (3)

- Induction hypothesis: $S = \text{sum of the first } k \text{ numbers}$
- 3. *Termination: When the loop terminates, the hypothesis implies the correctness of the algorithm*

The loop terminates when $k=n \Rightarrow s = \text{sum of first } k=n \text{ numbers} \Rightarrow$
postcondition of algorithm, DONE

Problem: Linear Search

Correctness Proof of Linear Search

- Use **Loop Invariant** for the while loop:
 - At the start of each iteration of the while loop, the search *key* is not in the subarray $A[1..i-1]$.

LinearSearch(A, key)

```
1   $i \leftarrow 1$ 
2  while  $i \leq n$  and  $A[i] \neq key$ 
3      do  $i++$ 
4  if  $i \leq n$ 
5      then return true
6      else return false
```

♦ If the algorithm terminates, then it produces correct result.

♦ Initialization.

♦ Maintenance.

♦ Termination.

♦ Argue that it terminates.

Problem 2: Binary Search

Binary Search

```
def binary_search(A, target):  
    lo = 0  
    hi = len(A) - 1  
    while lo <= hi:  
        mid = (lo + hi) / 2  
        if A[mid] == target:  
            return mid  
        elif A[mid] < target:  
            lo = mid + 1  
        else:  
            hi = mid - 1
```

You've all seen this a billion times.

But how do we prove that it's correct?

Given that A is sorted and A contains target, prove that `binary_search(A, target)` always returns target's index within A

Use Loop Invariants!!

Step 1: Hypothesize a Loop Invariant

```
def binary_search(A, target):
```

```
    lo = 0
```

```
    hi = len(A) - 1
```

```
    while lo <= hi:
```

```
        mid = (lo + hi) / 2
```

```
        if A[mid] == target:
```

```
            return mid
```

```
        elif A[mid] < target:
```

```
            lo = mid + 1
```

```
        else:
```

```
            hi = mid - 1
```

Say we're searching for 14 in the following array A

0	1	2	3	4	5	6
-5	10	14	33	42	42	42

1st step: lo = 0, hi = 6, mid = 3

2nd step: lo = 0, hi = 2, mid = 1

3rd step: lo = 2, hi = 2, mid = 2

At each step of the while loop, **lo** and **hi** *surrounded* the actual location of where 14 is! This was always true!

THIS IS OUR LOOP INVARIANT.

Step 1: Construct Loop Invariant

0	1	2	3	4	5	6
-5	10	14	33	42	42	42

↑ ↑
lo hi

0	1	2	3	4	5	6
-5	10	14	33	42	42	42
↑ lo		↑ hi				

0	1	2	3	4	5	6
-5	10	14	33	42	42	42

↑
lo
hi

At each iteration of the while loop,
lo and **hi** are such that:

$$A[lo] \leq \text{target} \leq A[hi]$$

Step 2: Prove that loop invariant is inductive

- Base Case: when the algorithm begins, $lo = 0$ and $hi = \text{len}(A) - 1$. lo and hi enclose ALL values, so **target** must be between lo and hi .
- Inductive Hypothesis: suppose at any iteration of the loop, lo and hi still enclose the **target** value.
- Inductive Step:
 - Case 1: If $A[mid] > \text{target}$, then the target must be between lo and mid
 - We update $hi = mid - 1$
 - Case 2: If $A[mid] < \text{target}$, then the target must be between mid and hi
 - we update $lo = mid + 1$
 - In either cases, we preserve the inductive hypothesis for the next loop

Step 3: Prove correctness property using loop invariant

- Notice for each iteration, **lo** always increases and **hi** always decreases. These value will converge at a single location where $lo = hi$.

0	1	2	3	4	5	6
-5	10	14	33	42	42	42

↑
lo
hi

- By the induction hypothesis, $A[lo] \leq \text{target} \leq A[hi]$.

Food for thought: How will the proof change if **target** isn't in the array?

Problem 3: array reversal

Correctness Proof

In--place Array Reversal

```
//inputs: array A of size n
void reverse_array(int *A, int n):
    int i = (n - 1) / 2;
    int j = n / 2;
    int tmp;
    while (i >= 0 && j <= (n - 1))
        tmp = A[i];
        A[i] = A[j];
        A[j] = tmp;
        i--;
        j++;
```

Prove that array A of size n is reversed as a result of invoking reverse_array(A, n)

Step 1: Hypothesize a Loop Invariant

0	1	2	3	4	5	6
-5	10	14	33	42	42	42

 ↑ ↑
 i j

0	1	2	3	4	5	6
-5	10	14	33	42	42	42

 ↑ ↑
 i j

0	1	2	3	4	5	6
-5	10	42	33	14	42	42

 ↑ ↑
 i j

Before iteration of the while loop,
 i and j are such that:

$A[i+1 : j-1]$ is reversed

Or more formally,

$\text{new_A}[i+1 : j-1] = \text{reverse}(\text{old_A}[i+1 : j-1])$

where,

$\text{reverse}([]) = []$

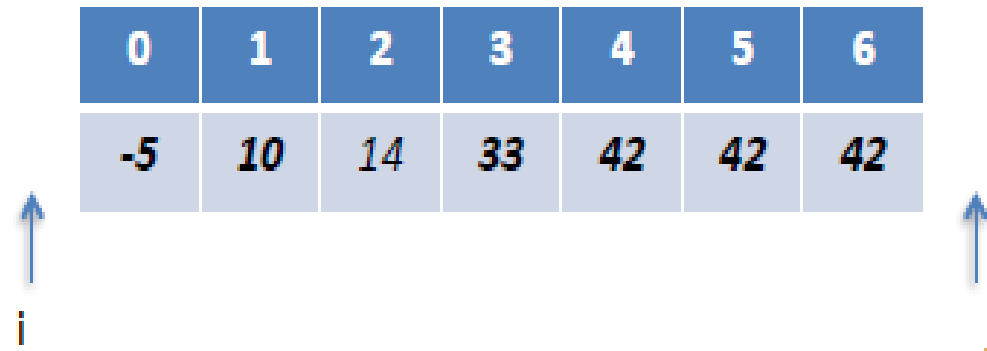
$\text{reverse}([a_0]) = [a_0]$

$\text{reverse}([a_0, a_1, \dots]) = [\text{reverse}([a_1, \dots]), a_0]$

Step 2: Prove that loop invariant is inductive

- Loop invariant: $A[i+1 : j-1]$ is reversed
- Base Case: Upon loop entry, $j - 1 < i + 1$. Invariant holds trivially.
- Inductive case:
At the start of k -th iteration, assume that $A[i+1 : j-1]$ is reversed.
The loop body swaps $A[i]$ and $A[j]$, decrements i and increments j .
Therefore, at the start of $(k+1)$ -th iteration, we can prove that $A[i+1 : j-1]$ is reversed.

Step 3: Prove correctness property using loop invariant



0	1	2	3	4	5	6
-5	10	14	33	42	42	42

i j

- After the loop terminates, $i = -1$ and $j = n$.
- Loop invariant tells us that $A[i+1 : j-1]$ is reversed.
- Therefore, $A[0:n-1]$ is reversed.

Invariants, preconditions, and post conditions

- A high enough level of abstraction, most correctness proofs look something like this: define a predicate P that is true for “correct” states of the algorithm. Then prove:
 1. P holds in the initial state.
 2. P holds after step k if it holds before step k .
 3. If P holds when the algorithm terminates, then the output of the algorithm is correct.
- Such a predicate P is called an invariant of the algorithm.
- It is called an invariant for the simple reason that it is always true; this follows by induction from the first two statements above.
- It is a useful invariant because of the third statement.

Loop Invariant

- **Logical expression with the following properties.**
 - Holds true before the first iteration of the loop — **Initialization**.
 - If it is true before an iteration of the loop, it remains true before the next iteration — **Maintenance**.
 - When the loop terminates, the **invariant** — **along with the fact that the loop terminated** — gives a useful property that helps show that the loop is correct — **Termination**.
- **Similar to mathematical induction.**
- **Proving loop invariants is similar to mathematical induction:**
 - showing that the invariant holds before the first iteration corresponds to the **base case**, and
 - showing that the invariant holds from iteration to iteration corresponds to the **inductive step**.

Mathematical foundation for more rigorous proofs of algorithm correctness.

A correctness proof is a formal mathematical argument that an algorithm meets its specification, which means that it always produces the correct output for any permitted input.

Algorithm correctness

- A **well-defined computational problem** is a pair $P = (I, O, R)$ such that I is a specification of the set of allowed inputs, O is a specification of the set of outputs and R is a specification of the desired relation between an input and an output.
- A **problem instance** is given by a specific input $i \in I$. Note also that R is a functional relation.
- An algorithm for solving a problem P is **totally correct** iff for all problem instances $i \in I$ it terminates and produces the correct output $o \in O$ (i.e. the pair $(i, o) \in R$).
- An algorithm for solving a problem P is **partially correct** iff for all problem instances $i \in I$ if it terminates then it produces the correct output $o \in O$ (i.e. the pair $(i, o) \in R$).

Correctness Proofs

- Proving (beyond “any” doubt) that an algorithm is correct.
 - Prove that the algorithm produces correct output when it terminates. Partial Correctness.
 - Prove that the algorithm will necessarily terminate. Total Correctness.
- Techniques
 - Proof by Construction.
 - Proof by Induction.
 - Proof by Contradiction.

Algorithm correctness: Multiplication of two integers

```
MULTIPLY( $a, b$ )  
1.  $x \leftarrow 0$   
2.  $y \leftarrow b$   
3. while  $y \neq 0$  do  
4.      $x \leftarrow x + a$   
5.      $y \leftarrow y - 1$   
6. return  $x$ 
```

- For the multiplication problem the set I of inputs is a set of pairs (a, b) such that a and b are integers.
- Note that if $b < 0$ then this algorithm **does not terminate**.
- So, if $I = Z \times Z$ (Z is the set of integers) then the algorithm is only **partially correct**.
- However, if we restrict I to $Z \times N$ then the algorithm is **totally correct** (N is the set of natural numbers)

Correctness of computing x^n

POWER2(x, n)

1. $p \leftarrow 1$
2. **while** $n > 0$ **do**
3. **if** n is even **then**
4. $x \leftarrow x * x$
5. $n \leftarrow n/2$
6. **else**
7. $n \leftarrow n - 1$
8. $p \leftarrow p * x$
9. **return** p

Correctness of computing x^n

POWER2(x, n)

```
1.  $p \leftarrow 1$ 
2. while  $n > 0$  do
3.     if  $n$  is even then
4.          $x \leftarrow x * x$ 
5.          $n \leftarrow n/2$ 
6.     else
7.          $n \leftarrow n - 1$ 
8.          $p \leftarrow p * x$ 
9. return  $p$ 
```

Let x_0 and n_0 be the initial values of x and n . Each time the algorithm enters the while loop we have: $p \times x^n = x_0^{n_0}$. The proof is made by induction on the number i of executions of the body of the while loop.

If $i = 0$ then $p = 1$, $x = x_0$ and $n = n_0$ so the result is obvious.

For the induction step we assume that at the end of the i -th execution we have $p \times x^n = x_0^{n_0}$. If $n = 2k$ then the new values for p , x and n are p , x^2 and k so the invariant is preserved because $p \times x^{2k} = p \times (x^2)^k$. If $n = 2k+1$ the new values for p , x and n are $p \times x$, x and $2k$. The invariant is again preserved because $p \times x^{2k+1} = (p \times x) \times x^{2k}$. When the algorithm stops $n = 0$ and so $p = x_0^{n_0}$. The algorithm always stops because n is decreasing after each pass through the loop.

Euclid's Algorithm correctness

Consider the well-known Euclid's algorithm for computing the greatest common divisor:

$$\gcd(m,n) = \text{EUCLID}(m,n)$$

EUCLID(m, n)

1. **while** *TRUE* **do**
2. $r \leftarrow m \bmod n$
3. **if** $r = 0$ **then**
4. **return** n
5. $m \leftarrow n$
6. $n \leftarrow r$

Euclid's Algorithm Complexity

- The number N of division steps of Euclid's algorithm for $m > n > 0$ is at most $5 \log_{10} n$
- It follows that the number of division steps is $O(h)$ where h is the number of digits of n .

Euclid's Algorithm correctness

- Let m_0 and n_0 be the initial values of m and n . We shall prove that each time the algorithm enters the while loop the following condition holds:

$$\gcd(m,n) = \gcd(m_0,n_0).$$

- This condition is called a *loop invariant*.
- The proof is made by induction on the number i of executions of the body of the while loop.
- If $i=0$ then the result is trivial because $m = m_0$ and $n = n_0$.
- For the induction step we assume that at the end of the i^{th} execution of the while loop we have $\gcd(m,n) = \gcd(m_0,n_0)$. It is enough to show that $\gcd(m,n) = \gcd(n,r)$. This follows trivially from the fact that r is computed as m modulo n .
- When the algorithm terminates m is divisible with n (because r is 0 !), so $\gcd(m,n) = n$. But $\gcd(m,n) = \gcd(m_0,n_0)$, so the returned value n is $\gcd(m_0,n_0)$.
To show that the algorithm terminates it is sufficient to notice that after each pass through the loop, the value of n decreases, so we shall have at most n_0 passes through the loop.

Correctness of evaluating polynomials

Given a polynomial $p(x) = p_0 + p_1x + \dots + p_nx^n = \sum_{i=0}^n p_i x^i$ and a value x the problem is to compute $p(x)$.

EVALUATE-POLY(p, n, x)

1. $y \leftarrow 0$
2. **for** $i = n, 0$ **do**
3. $y \leftarrow y * x + p[i]$
4. **return** y

We prove by induction on $i = n, \dots, 0$ that after each execution of the for loop we have: $y = \sum_{j=0}^{n-i} p_{i+j} x^j$.

For the base case, $i = n$, we have $y = p_n$ so the result is obvious.

For the induction case, we assume that after the execution of for i $y = \sum_{j=0}^{n-i} p_{i+j} x^j$ we have. After the next pass through the loop we obtain $y = x(\sum_{j=0}^{n-i} p_{i+j} x^j) + p_{i-1} = \sum_{j=0}^{n-i+1} p_{i+j-1} x^j$ and the proof is finished.

Applying this result for $i = 0$ we obtain that the algorithm evaluates correctly $p(x)$.

Mathematical Induction

Mathematical Induction

- *Mathematical induction* (MI) is an essential tool for proving the statement that proves an algorithm's correctness. The general idea of MI is to prove that a statement is true for every natural number **n**.

Basic Steps

1. **Induction Hypothesis:** Define the rule we want to prove for every n , let's call the rule **$F(n)$**
2. **Induction Base:** Proving the rule is valid for an initial value, or rather a starting point - this is often proven by solving the Induction Hypothesis $F(n)$ for $n=1$ or whatever initial value is appropriate
3. **Induction Step:** Proving that if we know that **$F(n)$** is true, we can step one step forward and assume **$F(n+1)$** is correct

Example: The sum of the first n natural numbers

Problem: If we define $S(n)$ as the sum of the first n natural numbers, for example $S(3) = 3+2+1$, prove that the following formula can be applied to any n : $S(n) = [(n+1)*n]/2$

- **Induction Hypothesis:** $S(n)$ defined with the formula above
- **Induction Base:** In this step we have to prove that $S(1) = 1$:

$$S(1) = [(1+1)*1]/2 = 2/2 = 1$$

- **Induction Step:** In this step we need to prove that if the formula applies to $S(n)$, it also applies to $S(n+1)$ as follows:

$$S(n+1) = [(n+1+1)*(n+1)]/2 = [(n+2)*(n+1)]/2$$

- This is known as an **implication** ($a \Rightarrow b$), which just means that we have to prove **b** is correct providing we know **a** is correct.

Example: The sum of the first n natural numbers

- This is known as an **implication** ($a \Rightarrow b$), which just means that we have to prove b is correct providing we know a is correct.

$$S(n+1) = S(n) + (n+1) = [(n+1)*n]/2 + (n+1)$$

$$\Rightarrow S(n+1) = (n^2 + n + 2n + 2)/2$$

$$\Rightarrow S(n+1) = (n^2 + 3n + 2)/2 = [(n+2)*(n+1)]/2$$

- Note that $S(n+1) = S(n) + (n+1)$ just means we are recursively calculating the sum.
- Example with literals: $S(3) = S(2) + 3 = S(1) + 2 + 3 = 1 + 2 + 3 = 6$

Correctness prove for Insertion Sort

Correctness prove for Insertion Sort

Input: An array $A[1 : n]$ of n numbers.

Output: Elements of $A[1 : n]$ rearranged in non-decreasing order of value.

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. $A[i + 1] = key$

Correctness prove for Insertion Sort with *loop invariants*

Use *loop invariants* to prove correctness of iterative algorithms.

A loop invariant is associated with a given loop of an algorithm, and it is a formal statement about the relationship among variables of the algorithm such that

- [**Initialization**] It is true prior to the first iteration of the loop
- [**Maintenance**] If it is true before an iteration of the loop, it remains true before the next iteration
- [**Termination**] When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Loop Invariants for Insertion Sort

INSERTION-SORT (A)

1. **for** $j = 2$ **to** $A.length$

Invariant 1: $A[1..j - 1]$ consists of the elements
originally in $A[1..j - 1]$, but in sorted order

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** $A[i] > key$

Invariant 2: $A[i..j]$ are each $\geq key$

6. $A[i + 1] = A[i]$

7. $i = i - 1$

8. $A[i + 1] = key$

Correctness prove for Insertion Sort with loop invariants

At the start of the first iteration of the loop (in lines 1- 8): $j = 2$
Hence, subarray $A[1.. j- 1]$ consists of a single element $A[1]$, which is in fact the original element in $A[1]$.

The subarray consisting of a single element is trivially sorted. Hence, the invariant holds initially.

INSERTION-SORT (A)

1. **for** $j = 2$ **to** $A.length$

Invariant 1: $A[1..j - 1]$ consists of the elements originally in $A[1..j - 1]$, but in sorted order

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** $A[i] > key$

Invariant 2: $A[i..j]$ are each $\geq key$

6. $A[i + 1] = A[i]$

7. $i = i - 1$

8. $A[i + 1] = key$

Loop Invariant 1: Maintenance & Loop Invariant 2: Termination

```
INSERTION-SORT ( A )
1.  for j = 2 to A.length
    Invariant 1: A[1..j - 1] consists of the elements
                  originally in A[1..j - 1], but in sorted order
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
        Invariant 2: A[i..j] are each ≥ key
6.        A[i + 1] = A[i]
7.        i = i - 1
8.    A[i + 1] = key
```

We assume that invariant 1 holds before the start of the current iteration.

Hence, the following holds: $A[1..j - 1]$ consists of the elements originally in $A[1..j - 1]$, but in sorted order.

For invariant 1 to hold before the start of the next iteration, the following must hold at the end of the current iteration:

$A[1..j]$ consists of the elements originally in $A[1..j]$, but in sorted order.

We use invariant 2 to prove this.

Loop Invariant 1: Maintenance & Loop Invariant 2: Termination

```
INSERTION-SORT ( A )
1.  for  $j = 2$  to  $A.length$ 
    Invariant 1:  $A[1..j-1]$  consists of the elements
                  originally in  $A[1..j-1]$ , but in sorted order
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  $A[i] > key$ 
        Invariant 2:  $A[i..j]$  are each  $\geq key$ 
6.         $A[i+1] = A[i]$ 
7.         $i = i - 1$ 
8.     $A[i+1] = key$ 
```

At the start of the first iteration of the loop (in lines 5 – 7): $i = j - 1$

Hence, subarray $A[i..j]$ consists of only two entries: $A[i]$ and $A[j]$.

We know the following:

- $A[i] > key$ (explicitly tested in line 5)
- $A[j] = key$ (from line 2)

Hence, invariant 2 holds initially.

Loop Invariant 1: Maintenance & Loop Invariant 2: Termination

```
INSERTION-SORT ( A )
1.  for  $j = 2$  to  $A.length$ 
    Invariant 1:  $A[1..j-1]$  consists of the elements
                originally in  $A[1..j-1]$ , but in sorted order
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  $A[i] > key$ 
        Invariant 2:  $A[i..j]$  are each  $\geq key$ 
6.         $A[i+1] = A[i]$ 
7.         $i = i - 1$ 
8.     $A[i+1] = key$ 
```

We assume that invariant 2 holds before the start of the current iteration.

Hence, the following holds: $A[i..j]$ are each $\geq key$.

Since line 6 copies $A[i]$ which is known to be $> key$ to $A[i+1]$ which also held a value $\geq key$, the following holds at the end of the current iteration: $A[i+1..j]$ are each $\geq key$.

Before the start of the next iteration the check $A[i] > key$ in line 5 ensures that invariant 2 continues to hold.

Loop Invariant 1: Maintenance & Loop Invariant 2: Termination

```
INSERTION-SORT ( A )  
1.  for j = 2 to A.length  
    Invariant 1:  $A[1..j-1]$  consists of the elements  
        originally in  $A[1..j-1]$ , but in sorted order  
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  
        Invariant 2:  $A[i..j]$  are each  $\geq$  key  
6.        A[i + 1] = A[i]  
7.        i = i - 1  
8.      A[i + 1] = key
```

Observe that the inner loop (in lines 5 – 7) does not destroy any data because though the first iteration overwrites $A[j]$, that $A[j]$ has already been saved in *key* in line 2.

As long as *key* is copied back into a location in $A[1..j]$ without destroying any other element in that subarray, we maintain the invariant that $A[1..j]$ contains the first *j* elements of the original list.

Loop Invariant 1: Maintenance & Loop Invariant 2: Termination

```
INSERTION-SORT ( A )  
1.  for  $j = 2$  to  $A.length$   
    Invariant 1:  $A[1..j-1]$  consists of the elements  
                originally in  $A[1..j-1]$ , but in sorted order  
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  $A[i] > key$   
        Invariant 2:  $A[i..j]$  are each  $\geq key$   
6.         $A[i+1] = A[i]$   
7.         $i = i - 1$   
8.     $A[i+1] = key$ 
```

When the inner loop terminates we know the following.

- $A[1..i]$ is sorted with each element $\leq key$
 - if $i = 0$, true by default
 - if $i > 0$, true because $A[1..i]$ is sorted and $A[i] \leq key$
- $A[i+1..j]$ is sorted with each element $\geq key$ because the following held before i was decremented: $A[i..j]$ is sorted with each item $\geq key$
- $A[i+1] = A[i+2]$ if the loop was executed at least once, and $A[i+1] = key$ otherwise

Loop Invariant 1: Maintenance & Loop Invariant 2: Termination

```
INSERTION-SORT ( A )  
1.  for  $j = 2$  to  $A.length$   
    Invariant 1:  $A[1..j-1]$  consists of the elements  
                originally in  $A[1..j-1]$ , but in sorted order  
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  $A[i] > key$   
        Invariant 2:  $A[i..j]$  are each  $\geq key$   
6.         $A[i+1] = A[i]$   
7.         $i = i - 1$   
8.     $A[i+1] = key$ 
```

When the inner loop terminates we know the following.

- $A[1..i]$ is sorted with each element $\leq key$
- $A[i+1..j]$ is sorted with each element $\geq key$
- $A[i+1] = A[i+2]$ or $A[i+1] = key$

Given the facts above, line 8 does not destroy any data, and gives us $A[1..j]$ as the sorted permutation of the original data in $A[1..j]$.

Loop Invariants 1 : Termination

```
INSERTION-SORT ( A )  
1.  for  $j = 2$  to  $A.length$   
    Invariant 1:  $A[1..j - 1]$  consists of the elements  
                originally in  $A[1..j - 1]$ , but in sorted order  
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  $A[i] > key$   
        Invariant 2:  $A[i..j]$  are each  $\geq key$   
6.         $A[i + 1] = A[i]$   
7.         $i = i - 1$   
8.     $A[i + 1] = key$ 
```

When the outer loop terminates we know that $j = A.length + 1$.

Hence, $A[1..j - 1]$ is the entire array $A[1..A.length]$, which is sorted and contains the original elements of $A[1..A.length]$.

Worst Case Runtime of Insertion Sort (Upper Bound)

INSERTION-SORT (A)

	<u>cost</u>	<u>times</u>
1. for $j = 2$ to $A.length$	c_1	n
2. $key = A[j]$	c_2	$n - 1$
3. $//$ insert $A[j]$ into the sorted sequence $A[1..j - 1]$	0	
4. $i = j - 1$	c_4	
5. while $i > 0$ and $A[i] > key$	c_5	$\sum_{2 \leq j \leq n} j$
6. $A[i + 1] = A[i]$	c_6	$\sum_{2 \leq j \leq n} (j - 1)$
7. $i = i - 1$	c_7	
8. $A[i + 1] = key$	c_8	$n - 1$

$$\begin{aligned}
 \text{Running time, } T(n) &\leq c_1 n + c_2(n - 1) + c_4(n - 1) \\
 &\quad + c_5 \sum_{j=2}^n j + c_6 \sum_{j=2}^n (j - 1) + c_7 \sum_{j=2}^n (j - 1) + c_8(n - 1) \\
 &= 0.5(c_5 + c_6 + c_7)n^2 + 0.5(2c_1 + 2c_2 + 2c_4 + c_5 - c_6 - c_7 + 2c_8)n \\
 &\quad - (c_2 + c_4 + c_5 + c_8) \\
 &\Rightarrow T(n) = O(n^2)
 \end{aligned}$$

Best Case Runtime of Insertion Sort (Lower Bound)

INSERTION-SORT (A)

	<u>cost</u>	<u>times</u>
1. for $j = 2$ to $A.length$	c_1	n
2. $key = A[j]$	c_2	$n - 1$
3. <i>// insert</i> $A[j]$ <i>into the sorted sequence</i> $A[1..j - 1]$	0	
4. $i = j - 1$	c_4	
5. while $i > 0$ and $A[i] > key$	c_5	
6. $A[i + 1] = A[i]$	c_6	0
7. $i = i - 1$	c_7	
8. $A[i + 1] = key$	c_8	$n - 1$

$$\text{Running time, } T(n) \geq c_1 n + c_2(n - 1) + c_4(n - 1) \\ + c_5(n - 1) + c_8(n - 1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$$\Rightarrow T(n) = \Omega(n)$$

Summary

- The correctness of a recursive algorithm should be shown using induction.
- The correctness of an iterative algorithm should be shown by proving initialization, maintenance, termination, and correctness for each of the loops.
- Some algorithms might contain both recursion and iteration. In such cases, both techniques should be used.
- Because the algorithm is recursive, its correctness should be shown using induction. In order to complete the induction, the loops will need to be handled by proving initialization, maintenance, termination, and correctness.

Thank you for your attention



References

- <http://cs.engr.uky.edu/~lewis/essays/algorithms/correct/correct1.html>
- <http://web.info.uvt.ro/~dzaharie/algeng/lecture3.pdf>
- <https://stackabuse.com/mathematical-proof-of-algorithm-correctness-and-efficiency/>

Exercises

Exercise

Exercise 1 Induction can be used to prove solutions for summations. Use induction to prove each of the following:

a. The *arithmetic series*:

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

b. The *geometric series*:

$$\sum_{i=0}^n x^i = \frac{x^{n+1} - 1}{x - 1} \quad (2)$$

for any real $x \neq 1$.

Exercise 2 Let

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{i=0}^{n-1} f(i) & \text{if } n > 0. \end{cases}$$

Use induction to prove that for all $n > 0$, $f(n) = 2^{n-1}$.

Exercise

* **Exercise 3** The *Fibonacci sequence* is defined as follows:

$$F_n = \begin{cases} n & \text{if } 0 \leq n \leq 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases} \quad (3)$$

Use induction to prove each of the following properties of the Fibonacci sequence:

a. For every $n > 0$,

$$F_{n-1}F_n + F_nF_{n+1} = F_{2n} \quad (4)$$

and

$$F_n^2 + F_{n+1}^2 = F_{2n+1}. \quad (5)$$

[**Hint:** Prove both equalities together in a single induction argument.]

b. For every $n \in \mathbb{N}$,

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}, \quad (6)$$

where ϕ is the *golden ratio*:

$$\phi = \frac{1 + \sqrt{5}}{2}.$$