CSE373: Data Structures and Algorithms

# *Induction and Its Applications*

Steve Tanimoto
Winter 2016

This lecture material is based on materials provided by Ioana Sora at the Politechnic University of Timisoara.

---

## *Lecture Outline*

- **Proving the Correctness of Algorithms**
  - Preconditions and Postconditions
  - Loop Invariants
  - Induction – Math Review
  - Using Induction to Prove Algorithms

---

## *What are key parts of an algorithm ?*

- An algorithm is described by:
  - Input data
  - Output data
  - ***Preconditions***: specifies restrictions on input data
  - ***Postconditions***: specifies what is the result
- Example: Binary Search
  - 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 postcondition

---

## *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 input (which satisfies the precondition) imply that the output satisfies the postcondition***

  - This is easy to prove for simple sequential algorithms
  - This can be complicated to prove for repetitive algorithms (containing loops or recursion)
    - use techniques based on ***loop invariants*** and ***induction***

---

## *Example – a sequential algorithm*

```
Swap1(x,y):
    aux := x
    x := y
    y := aux
```

**Precondition:**

`x = a and y = b`

**Postcondition:**

`x = b and y = a`

**Proof:** ***the list of actions applied to the input (which satisfies the precondition) imply the output satisfies the postcondition***

```
1. Precondition:
   x = a and y = b
2. aux := x => aux = a
3. x : = y => x = b
4. y := aux => y = a
5. x = b and y = a is
   the Postcondition
```

## *Example – a repetitive algorithm*

Algorithm Sum_of_N_numbers

Input: integer `N`, and
`a`, an array of N numbers
Output: `s`, the sum of the `N` numbers in `a`

```
s:=0;
k:=0;
while (k<N) do
  s:=s+a[k];
  k:=k+1;
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*

- 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

## *Example: Loop invariant for Sum of n numbers*

**`Algorithm Sum_of_N_numbers`**

**Algorithm Sum_of_N_numbers**

**Input: integer `N`, and**
**`a`, an array of `N` numbers**
**Output: `s`, the sum of the `N` numbers in `a`**

```
s:=0;
k:=0;
while (k<N) do
  s:=s+a[k];
  k:=k+1;
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 2 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.

We also must show **Termination:** that the loop terminates.

When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

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

- Induction hypothesis: s = sum of the first k 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) => hypothesis true before entering the loop

## *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] => s is the sum of the first k+1 numbers

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

- Induction hypothesis: s = sum of the first k numbers

3. *Termination: When the loop terminates, the hypothesis implies the correctness of the algorithm*

The loop terminates when k=n

This implies s = sum of first k=n numbers

Thus the postcondition of the algorithm is satisfied.

Q.E.D. (Quod Erat Demonstrandum; we are done.)

---

## *Loop invariants and induction*

- **Proving loop invariants is a form of 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 induction - Review*

- Let $(\forall n \geq c)T(n)$ be a theorem that we want to prove. It includes a constant c and a natural parameter n.
- Proving that T holds for all natural values of n greater than or equal to c is done by proving following two conditions:

1. T holds for n=c
2. For every n>c if T holds for n-1, then T holds for n

Terminology:

| | |
|---|---|
| **T(c)** | is the Base Case |
| **T(n-1)** | is the Induction Hypothesis |
| **T(n-1) => T(n)** | is the Induction Step |
| **$(\forall n \geq c)$T(n)** | is the Theorem being proved. |

---

## *Mathematical induction - review*

- **Strong Induction**: a variant of induction where the inductive step builds up on *all* the smaller values
- Proving that T holds for all natural values of n greater than or equal to c is done by proving following two conditions:

1. T holds for n=$c_1$, $c_1$+1, …, $c_m$
2. If for every k from $c_1$ up to n-1, it is true that T(k), then T(n)

---

## *Mathematical induction – Example1*

- Theorem: *The sum of the first n natural numbers is n*(n+1)/2*

  $(\forall n \geq 1)T(n) \Leftrightarrow (\forall n \geq 1)\ \sum_{k=1}^{n} k = n(n+1)/2$

- Proof: by *induction* on n

1. Base case: If n=1, s(1)=1=1*(1+1)/2
2. Inductive step: We assume that s(n)=n*(n+1)/2, and prove that this implies s(n+1)=(n+1)*(n+2)/2, for all n≥1

s(n+1)=**s(n)**+(n+1)=**n*(n+1)/2**+(n+1)=(n+1)*(n+2)/2

---

## *Mathematical induction – Example2*

- Theorem: *Every amount of postage that is at least 12 cents can be made from 4-cent and 5-cent stamps.*
- *Proof: by induction on the amount of postage*
- *Postage (p) = m * 4 + n * 5*
- Base cases:
  - Postage(12) = 3 * 4 + 0 * 5
  - Postage(13) = 2 * 4 + 1 * 5
  - Postage(14) = 1 * 4 + 2 * 5
  - Postage(15) = 0 * 4 + 3 * 5

## *Mathematical induction – Example2 (cont)*

- Inductive step: We assume that we can construct postage for every value from 12 up to k. We need to show how to construct k + 1 cents of postage. Since we have proved base cases up to 15 cents, we can assume that k + 1 ≥ 16.
- Since k+1 ≥ 16, (k+1)−4 ≥ 12. So by the inductive hypothesis, we can construct postage for (k + 1) − 4 cents: (k + 1) − 4 = m * 4+ n * 5
- But then k + 1 = (m + 1) * 4 + n * 5. So we can construct k + 1 cents of postage using (m+1) 4-cent stamps and n 5-cent stamps

---

## *Correctness of algorithms*

- Induction can be used for proving the correctness of repetitive algorithms:
  - Iterative algorithms:
    - Loop invariants
      - Induction hypothesis = loop invariant = relationships between the variables during loop execution
  - Recursive algorithms
    - Direct induction
      - induction hypothesis = assumption that each recursive call itself is correct (often a case for applying *strong* induction)

---

## *Example: Correctness proof for Decimal to Binary Conversion*

```
Algorithm Decimal_to_Binary

Input: n, a positive integer
Output: b, an array of bits, the bin repr. of n,
        starting with the least significant bits

t:=n;
k:=0;
while (t>0) do
  b[k]:=t mod 2;
  t:=t div 2;
  k:=k+1;
end
```

**It is a repetitive (iterative) algorithm, thus we use loop invariants and proof by induction**

---

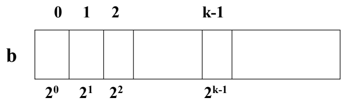
## *Example: Loop invariant for Decimal to Binary Conversion*

```
Algorithm Decimal_to_Binary

Input: n, a positive integer
Output: b, an array of bits, the bin repr. of n

t:=n;
k:=0;
while (t>0) do
  b[k]:=t mod 2;
  t:=t div 2;
  k:=k+1;
end
```

**At step k, b holds the k least significant bits of n, and the value of t, when shifted by k, corresponds to the rest of the bits**



---

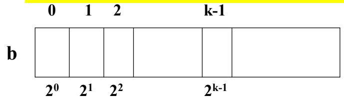
## *Example: Loop invariant for Decimal to Binary Conversion*

```
Algorithm Decimal_to_Binary

Input: n, a positive integer
Output: b, an array of bits, the bin repr. of n

t:=n;
k:=0;
while (t>0) do
  b[k]:=t mod 2;
  t:=t div 2;
  k:=k+1;
end
```

**Loop invariant: If m is the integer represented by array b[0..k-1], then n=t*$2^k$+m**



---

## *Example: Proving the correctness of the conversion algorithm*

- **Induction hypothesis=Loop Invariant**: If m is the integer represented by array b[0..k-1], then n=t*$2^k$+m
- **To prove the correctness of the algorithm, we have to prove the 3 conditions:**
  1. ***Initialization: The hypothesis is true at the beginning of the loop***
  2. ***Maintenance: If hypothesis is true for step k, then it will be true for step k+1***
  3. ***Termination: When the loop terminates, the hypothesis implies the correctness of the algorithm***

## *Example: Proving the correctness of the conversion algorithm (1)*

- Induction hypothesis: If m is the integer represented by array b[0..k-1], then $n=t*2^k+m$

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

   $k=0$, $t=n$, $m=0$(array is empty)

   $n=n*2^0+0$

## *Example: Proving the correctness of the conversion algorithm (2)*

- Induction hypothesis: If m is the integer represented by array b[0..k-1], then $n=t*2^k+m$

2. *If hypothesis is true for step k, then it will be true for step k+1*

At the start of step k: assume that $n=t*2^k+m$, calculate the values at the end of this step

If t is even then: t mod 2=0, m unchanged,

$t:=t / 2, k:=k+1 \Rightarrow (t / 2) * 2^{(k+1)} + m = t*2^k+m=n$

If t is odd then: t mod 2 =1, b[k+1] is set to 1, $m=m+2^k$ ,

$t:=(t-1)/2, k:=k+1 \Rightarrow (t-1)/2*2^{(k+1)}+m+2^k=t*2^k+m=n$

## *Example: Proving the correctness of the conversion algorithm (3)*

- Induction hypothesis: If m is the integer represented by array b[0..k-1], then $n=t*2^k+m$

3. *When the loop terminates, the hypothesis implies the correctness of the algorithm*

The loop terminates when t=0 =>

$n=0*2^k+m=m$

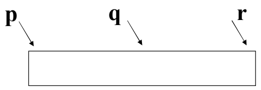
n==m, proved

## *Proof of Correctness for Recursive Algorithms*

- In order to prove recursive algorithms, we have to:
  1. Prove the partial correctness (the fact that the program behaves correctly)
     - *we assume that all recursive calls with arguments that satisfy the preconditions behave as described by the specification, and use it to show that the algorithm behaves as specified*
  2. Prove that the program terminates
     - any chain of recursive calls eventually ends and all loops, if any, terminate after some finite number of iterations.

## *Example - Merge Sort*

```
MERGE-SORT(A,p,r)
   if p < r
        q = (p+r)/2
        MERGE-SORT(A,p,q)
        MERGE-SORT(A,q+1,r)
        MERGE(A,p,q,r)
```



**Precondition:**

Array A has at least 1 element between indexes p and r ($p \leq r$)

**Postcondition:**

The elements between indexes p and r are sorted

## *Example - Merge Sort*

- MERGE-SORT calls a function MERGE(A,p,q,r) to merge the sorted subarrays of A into a single sorted one
- The proof of MERGE (which is an iterative function) can be done separately, using loop invariants
- We assume here that MERGE has been proved to fulfill its postconditions (can do it as a distinct exercise)

**MERGE (A,p,q,r)**

**Precondition:** A is an array and p, q, and r are indices into the array such that $p \leq q < r$.

The subarrays A[p .. q] and A[q+1 .. r] are sorted

**Postcondition:** The subarray A[p..r] is sorted

## *Correctness proof for Merge-Sort*

- Number of elements to be sorted: n=r-p+1
- **Base Case**: n = 1
  - A contains a single element (which is trivially “sorted”)
- **Inductive Hypothesis:**
  - Assume that MergeSort correctly sorts n=1, 2, ..., **k** elements
- **Inductive Step:**
  - Show that MergeSort correctly sorts n = **k + 1** elements.
  - First recursive call n1=q-p+1=(k+1)/2 ≤ k => subarray A[p .. q] is sorted
  - Second recursive call n2=r-q=(k+1)/2 ≤ k => subarray A[q+1 .. r] is sorted
  - A, p, q, r fulfill now the precondition of Merge
  - The postcondition of Merge guarantees that the array A[p .. r] is sorted => postcondition of MergeSort

## *Correctness proof for Merge-Sort*

- **Termination:**
  - To argue termination, we have to find a quantity that decreases with every recursive call: the length of the part of A considered by a call to MergeSort
  - For the base case, we have a one-element array. the algorithm terminates in this case without making additional recursive calls.

## *Correctness proofs for recursive algorithms*

```
RECURSIVE(n) is
    if (n=small_value)
        return simple_answer
    else
          RECURSIVE(n1)
          …
          RECURSIVE(nr)
          some_code
```

*$n_1$, $n_2$, … $n_r$ are some values smaller than n but bigger than small_value*

- **Base Case**: Prove that RECURSIVE works for n = small_value
- **Inductive Hypothesis:**
  - Assume that RECURSIVE works correctly for n=small_value, ..., **k**
- **Inductive Step:**
  - Show that RECURSIVE works correctly for n = **k + 1**

## *Lecture Summary*

- Proving that an algorithm is totally correct means:
  1. Proving that it will ***terminate***
  2. Proving that the list of ***actions*** applied to the input (satisfying the ***precondition***) imply the output satisfies the ***postcondition.***
- How to prove ***repetitive algorithms*** correct:
  - ***Iterative*** algorithms: use ***Loop invariants***, Induction
  - ***Recursive*** algorithms: use induction using as hypothesis the recursive call

## *Bibliography*

- Weiss, Ch. 1 section on induction.
- Goodrich and Tamassia: Induction and loop invariants; see, e.g., http://www.cs.mun.ca/~kol/courses/2711-w09/Induction.pdf)
- Erickson, J. Proof by Induction. Available at: http://jeffe.cs.illinois.edu/teaching/algorithms/notes/98-induction.pdf