Proving the Correctness of Algorithms

Lecture Outline

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

What does 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 precondition imply the postcondition***

  - This is easy to prove for simple sequential algorithms
  - 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

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

Precondition:

`x = a and y = b`

Postcondition:

`x = b and y = a`

Proof:

1. `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`

Example – a repetitive algorithm

Algorithm sum_of_numbers

Input: A, an array of n numbers
Output: s, the sum of the n numbers in A

```
s:=0;
k:=0;
While (k<n) do
  k:=k+1;
  s:=s+A[k];
end
```

Proof:

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_numbers

Input: A, an array of n numbers
Output: s, the sum of the n numbers in A

s:=0;
k:=0;
While (k<n) do
   k:=k+1;
   s:=s+A[k];
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.

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:*

   k=0, S=0

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

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

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

At the start of step k: assume that S is the sum of the first k numbers, calculate the value of S at the end of this step

K:=k+1, s:=s+A[k+1]

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 =>s= sum of first k=n numbers, proved

Loop invariants and 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.

Mathemetical Induction Review

- Let T be a theorem that we want to prove. T includes a natural parameter n.
- Proving that T holds for all natural values of n is done by proving following two conditions:

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

Terminology:
T= Induction Hypothesis
1= Base case
2= Inductive step

Correctness of algorithms

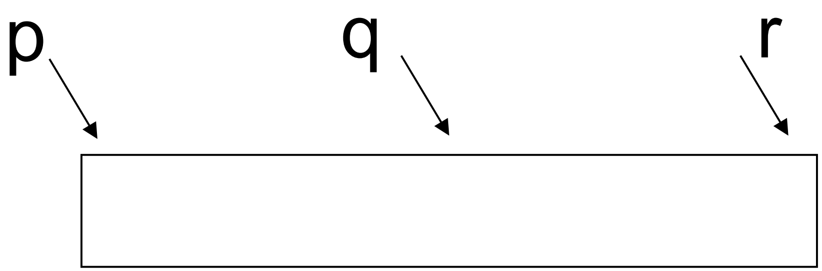
- Using induction to prove
  - Loop invariants
    - Induction hypothesis = loop invariant = relationships between the variables during loop execution
  - Recursive algorithms

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<=r)

Postcondition:

The elements between indexes p and r are sorted

Example - Merge Sort

MERGE (A,p,q,r)

Precondition: A is an array and p, q, and r are indices into the array such that p <= q < r. The subarrays A[p..q] and A[q+1.. r] are sorted

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

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1+1] and R[1..n2+1] be new arrays
for i=1 to n1
  L[i] = A [p+i-1]
for j=1 to n2
  R[j] = A [q +j]
L[n1] = infinity
R[n2] = infinity
i = 1
j = 1
for k = p to r
   if L[i]<=  R[j]
       A[k] = L[i]
       i=i+1
   else A[k] = R[j]
       j=j+1
```

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 subarray A[p .. r] is sorted

Correctness Proof for Merge-Sort

- **Termination**:
  - To argue termination, we find a quantity that decreases with every recursive call. One possibility is 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 Proof for Merge

- **Loop Invariant:**
- At the start of each iteration of the **for k loop**, the subarray A[p.. k-1] contains the k-p smallest elements of L[1..n1+1] and R[1 .. n2+1], in sorted order. Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
Let L[1..n1+1] and R[1..n2+1] be new arrays
for i=1 to n1
   L[i] = A [p+i-1]
for j=1 to n2
   R[j] = A [q +j]
L[n1] = infinity
R[n2] = infinity
i = 1
j = 1
for k = p to r
   if L[i]<=  R[j]
       A[k] = L[i]
       i=i+1
   else A[k] = R[j]
       j=j+1
```

Correctness Proof for Merge (1)

- **Initialization:**
  - Prior to the first iteration of the loop, we have k = p, so that the subarray A[p .. k -1] is empty. This empty subarray contains the k-p = 0 smallest elements of L and R, and since i = j = 1, both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Correctness Proof for Merge (2)

- **Maintenance:**
  - To see that each iteration maintains the loop invariant, let us first suppose that L[i]<= R[j] . Then L[i] is the smallest element not yet copied back into A. Because A[p .. k – 1] contains the k - p smallest elements, after L[i] is copied into A[k], the subarray A[p .. k] will contain the k-p+1 smallest elements. Incrementing k (in the for loop update) and i reestablishes the loop invariant for the next iteration. If instead L[i] > R[j] , then R[j] is copied into A[k], and the loop invariant is maintained in a similar way.

Correctness Proof for Merge (3)

- **Termination**:
  - At termination, k=r+1. By the loop invariant, the subarray A[p .. k–1], which is A[p .. r], contains the k-p= r-p+1 smallest elements of L[1.. n1+1] and R[1.. n2+1], in sorted order. The arrays L and R together contain n1+n2+ 2= r- p+3 elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

- Proving correctness of algorithms, Induction:
  - [CLRS] – chap 2.1, chap 2.3.1