CSC236H
Introduction to the Theory of Computation

- Correctness of Iterative Programs:
  - **Termination:** $Pre \Rightarrow Term$.
  - **Partial Correctness** $Pre \wedge Term \Rightarrow Post$.

Loop Termination – Review

- Associate with the loop a **loop measure** m loop measure:
  1. m decreases with each iteration of the loop;
  2. m is always a natural number at the beginning of each loop iteration
- An appropriate **loop measure** for a loop represents the **maximum** number of remaining iterations of the loop.
- **Intuitively:** if such an m exists, eventually m reaches 0, which is the smallest natural number, and therefore the loop eventually terminates.
- **Formally:** Every **strictly decreasing** sequence of natural numbers is **finite**
  So the sequence of values for m, and thus the number of iterations is finite.

```
def avg(A):
    sum = 0
    i = 0
    while i < len(A):
        sum += A[i]
        i += 1
    return sum / len(A)
```

Correctness of Iterative Programs – Loop Invariant

- **Precondition:** Pre_Cond.
- **Postcondition:** $Post\_Cond$.

```
                                        |
def iter_prog(P):                       |     def rec_prog(P):
#Pre_Cond                               |     #Pre_Cond
         .                              |          if P has "small enough" size:
         .                              |              return res_0
         .                              |              .
         .                              |              .
     while loop_cond: {                 |              .
                                        |         #P' has smaller size than P
             .                          |         rec_prog(P')
             .                          |              .
             .                          |              .
     }                                  |              .
                                        |
     #Some instructions                 |         #Some instructions
     return res                         |         return res
#Post_Cond                              |     #Post_Cond
                                        |
                                        |
```

- **Loop Invariant:** a statement that is true on entering the loop, and after every iteration.

- A good **Loop Invariant**:
  - is a formal specification of the behaviour of the loop;
  - can be applied to entail the Postcondition;
  - can be applied to prove Termination.

Tips for Identifying Loop Invariants

- Loop invariants provide a formal description of relationships between variables in the loop.
- Loop invariants are incremental $\rightarrow$ they should be about a **portion** of the values that are being processed.
- A loop invariant should hold even when the loop condition is false.

Trial-and-error process for identifying loop invariants:

1. Understand what the loop does.
   How the loop's function is related to the correctness of the program?
   Tracing the program sometimes help.
2. Formulate a candidate statement as our proposed loop invariant.
3. Check to see if the proposed loop invariant is sufficient to prove partial correctness.
4. If the answer in Step 3 is negative, then we repeat the above three steps.

- **Precondition:** $Pre\_Cond.$
- **Postcondition:** $Post\_Cond.$

```
def iter_prog(P):
#Pre_Cond
        .
        .
        .
        .
     while loop_cond: {

            .
            .
            .
     }

     #Some instructions
     return res
#Post_Cond
```

1. Precondition implies that LI(0) is true.

2. For all integers $k \geq 0$, if the guard $loop\_cond$ and $LI(k)$ are both true before an iteration of the loop, then $LI(k+1)$ is true after iteration of the loop.

3. After a finite number of iterations of the loop, $loop\_cond$ becomes false.

4. If loop iterates n times before termination, and $LI(n)$ is true, and the instructions after the loop execute, then the Postcondition holds.

Steps in proving Correctness of Iterative Programs

1. Formulate a loop invariant (**LI**).
2. Prove the LI using **Induction**:
   a) Prove that assuming the precondition holds, then the LI holds on entering the loop; **(Base Case)**
   b) Prove that if the LI holds before an iteration, then it also holds after that iteration. **(Induction Step)**
3. Use the LI to prove **partial correctness**:
   a) Proving that if the loop halts, then the postcondition follows:
   The **loop exit condition** (negation of the condition in the while loop) and the **LI** implies **postcondition**.
4. Find a **loop measure** m such that
   a) the value of m is a natural number on entering the loop, and after every iteration.
   b) the value of m decreases with every iteration.
5. Use LI to prove that the **loop measure** m actually satisfies the above conditions.

Note: In doing steps 2 to 5, it is not unusual to find that the LI from step 1 needs to be modified. So sometimes all the steps need to be revisited multiple times before a complete proof is obtained.

- **Precondition:** A is a non-empty list of numbers.
- **Postcondition:** Returns the average of the numbers in A.

```
def avg(A):
    sum = 0
    i = 0
    while i < len(A):
          sum += A[i]
          i += 1
    return sum / len(A)
```

Note: A variable v subscripted with a natural number k denotes the value of the variable v at the end of the k-th iteration of the loop (if such an iteration exists).

$LI(k)$: if the loop is executed at least k times, then

Correctness of Iterative Programs – Example

- **Precondition:** $n \in \mathbb{N}$
- **Postcondition:** Returns n^2.

```
def Sq(n):
    s = 0; d = 1; i = 0
    while i < n:
        s = s + d
        d = d + 2
        i = i + 1
    return s
```

$LI(k)$: if the loop is executed at least k times, then

Correctness of Iterative Programs – Example

- **Precondition:** $m \in \mathbb{N}$ and $n \in \mathbb{Z}$
- **Postcondition:** Returns $m \cdot n$.

```
def Mult(m, n):
    x = m; y = n; z = 0
    while x != 0:
        if (x mod 2) == 1:
            z = z + y
        x  = x // 2
        y  = y * 2
    return z
```

$LI(k)$: if the loop is executed at least k times, then $z_k = mn - x_k y_k$.

Correctness of Iterative Programs – Example

- **Precondition:** $m \in \mathbb{N}$ and $n \in \mathbb{Z}$
- **Postcondition:** Returns $m \cdot n$.

```
def Mult(m, n):
1.    x = m; y = n; z = 0
2.    while x != 0:
3.         if (x mod 2) == 1:
4.               z = z + y
5.         x  = x // 2
6.         y  = y * 2
7.    return z
```

Path 1: $(x \mod 2) == 1$

4. $z = z + y$
5. $x = x//2$
6. $y = y * 2$

$$z_{k+1} = z_k + y_k$$
$$x_{k+1} = \frac{x_k - 1}{2}$$
$$y_{k+1} = 2y_k$$

Path 2: $(x \mod 2) == 0$

5. $x = x//2$
6. $y = y * 2$

$$z_{k+1} = z_k$$
$$x_{k+1} = \frac{x_k}{2}$$
$$y_{k+1} = 2y_k$$

Correctness of Recursive Programs Containing Loops– Example

- **Precondition:** A is a non-empty list of numbers.
- **Postcondition:** Return a list containing elements in A in sorted order.

```
def MergeSort(A):
    if len(A) == 1:
        return A
    else:
        m = len(A) // 2  # integer division
        L1 = MergeSort(A[0..m-1])
        L2 = MergeSort(A[m..len(A)-1])
        i = 0
        j = 0
        C = []
        #merging L1 and L2
        while (i < len(L1)) and (j < len(L2)):
            if (L1[i] <= L2[j]):
                C.append(L1[i]) # Add L1[i] to the end of C
                i += 1
            else:
                C.append(L2[j])
                j += 1
     return C + L1[i..len(L1)-1] + L2[j..len(L2)-1] # List concatenation
```

Correctness of Recursive Programs Containing Loops– Example

- **Precondition:** A is a non-empty list of numbers.
- **Postcondition:** Return a list containing elements in A in sorted order.

```
def MergeSort(A):
    if len(A) == 1:
        return A
    else:
        m = len(A) // 2  # integer division
        L1 = MergeSort(A[0..m-1])
        L2 = MergeSort(A[m..len(A)-1])
        return merge(L1, L2)
```

- **Precondition:** A and B are non-empty lists of numbers. A and B are sorted.
- **Postcondition:** Return a list containing all elements in A and B in sorted order.

```
def merge(A, B):
    i = 0
    j = 0
    C = []
    while (i < len(A)) and (j < len(B)):
        if (A[i] <= B[j]):
            C.append(A[i])
            i += 1
        else:
            C.append(B[j])
             j += 1
     return C + A[i..len(A)-1] + B[j..len(B)-1]
```

Correctness of Recursive Programs Containing Loops– Example

- **Precondition:** A is a non-empty list of numbers.
- **Postcondition:** Return a list containing elements in A in sorted order.

```
def MergeSort(A):
    if len(A) == 1:
        return A
    else:
        m = len(A) // 2  # integer division
        L1 = MergeSort(A[0..m-1])
        L2 = MergeSort(A[m..len(A)-1])
        return merge(L1, L2)
```

- **Precondition:** A and B are non-empty lists of numbers. A and B are sorted.
- **Postcondition:** Return a list containing all elements in A and B in sorted order.

```
def merge(A, B):
    i = 0
    j = 0
    C = []
    while (i < len(A)) and (j < len(B)):
        if (A[i] <= B[j]):
            C.append(A[i])
            i += 1
        else:
            C.append(B[j])
             j += 1
     return C + A[i..len(A)-1] + B[j..len(B)-1]
```

$P(k)$: if A is a non-empty lists of numbers and $len(A) = k$, then $MergeSort(A)$ terminates, and returns a list containing all elements in A in sorted order.

Correctness of Recursive Programs Containing Loops

- **Step 1:** Prove the correctness of the loop separately.
- **Step 2:** Prove the correctness of the program like a regular recursive program, and using the loop postcondition proved in Step 1.

Exercises

- Prove the correctness of the following programs.
  (Remember to following the steps described in "Steps in proving Correctness of Iterative Programs")

- **Precondition:** x and y are natural numbers, and $x \geq y$.
- **Postcondition:** Returns $x * y$.

```
def M(x,y):
    i = 0;
    r = 0
    while i < y:
        r += x
        i += 1
    return r
```

Exercises

- **Precondition:** $n \in \mathbb{N}$
- **Postcondition:** Returns $n!$.

```
def G(n):
    i = 0; f = 1;
    while i < n:
        i = i + 1
        f = f * i
    return f
```

- **Precondition:** A is a list of natural numbers and the length of A is greater than 0.
- **Postcondition:** Returns largest number in A.

```
def H(A):
    result = 0
    i = 0
    while(i < len(A)):
        if(A[i] > result):
            result = A[i]
        i += 1
    return result
```