Lecture 5

Loop Invariants, Correctness of Iterative Algorithm: Initialization, Maintenance, and Termination.

Algorithms & Its Description

Algorithms can solve computational problems. Computational problems have Inputs and outputs

Two Requirements:

- Given an input to algorithm, should produce the correct output
- The algorithm should use resources efficiently.

A Complete description of Algorithm has THREE parts.

- The Algorithm – expressed in concise English / Pseudocode
- A proof of Algorithm Correctness
- A derivation of the algorithm's running time

Correctness of Iterative Algorithm

Initialization, Maintenance, Termination

Linear Search : To show Output Correctness

Linear-Search (A, n, x)

Input:

- A: An Array
- n: number of elements in A
- x: the value to be searched for

Output:

- Either an index i for A[i]=x, or Not-Found if A does not contain x

```
1. FOR  i=0 to n-1
2.       IF  A[i]=x THEN RETURN  i
3. RETURN  Not-Found
```

Loop Invariant

To prove correctness, We focus on Loop Invariant

Loop Invariant is an assertion that we prove to be TRUE each time a loop starts.
To Prove correctness with Loop Invariants, we have THREE things.

- Initialization: Invariant is true before first iteration of the loop
- Maintenance: If invariant is TRUE before an iteration of the loop, it remains TRUE before the next iteration.
- Termination: When the loop terminates, the loop invariant along with the reason that the loop terminated, gives us a useful property.

```
<loop invariant>
while(test) :
        <test AND loop invariant>
        //  loop body
        <loop invariant>

<~test AND loop invariant>
```

If we can prove that
the loop invariant holds before the loop and that the loop body keeps the loop invariant true

then we can infer that

~test AND loop invariant
holds after the loop terminates

Linear Search

Linear-Search (A, n, x)

```
1. FOR  i=0 to n-1
2.       IF  A[i]=x THEN RETURN  i
3. RETURN  Not-Found
```

To Show

1. IF index i is returned THEN A[i]=x // x is in array A
2. IF Not-Found is returned THEN x is not in the array

Loop invariant

At the start of each iteration of STEP-1, IF x is present in the array A, THEN it is present in the subarray from A[i : n].

A[a:b] : array A from index a up to but **excluding** b

Loop Invariant

A[a:b] : array A from index a up to but excluding b

Loop invariant

At the start of each iteration of STEP-1, IF x is present in the array A, THEN it is present in the subarray from A[i : n].

- Initialization
  - Initially, i=0 so that the subarray in the Loop Invariant is A[0:n], which is the entire array. Thus, the Loop Invariant holds initially.
- Maintenance
  - If at the start of an iteration, x is present in the array, then it is present in the subarray A[i : n]. If we do not return, then $A[i] \neq x$. Hence, if x is in the array, then is it in the subarray A[i+1 : n]. Here i is incremented before the next iteration, so the invariant will hold again.
- Termination

"if A then B" ≡ "if not B then not A"

  - $If\ A[i] = x$. If $i > n - 1$, consider contrapositive of invariant. "If x is not present in the subarray A[n : n], then x is not present in A.
  - Here, $i > n - 1$
    - →subarray from A[i : n] is Empty
    - → x is not present in an empty subarray → $\mathbf{x\ is\ not\ present\ in\ A}$

Is this Loop Invariant unique?

› Linear-Search (A, n, x)

1. FOR i=0 to n-1

2. *IF* A[i]=x *THEN RETURN* i

3. RETURN Not-Found

› Which of the following is a valid Loop Invariant?

A. At the start of each iteration of the Step-1, A[0 : i -1] does not contain x

B. At the start of each iteration of the Step-1, A[0 : i] does not contain x

C. At the start of each iteration of the Step-1, A[i : n] does not contain x

D. At the start of each iteration of the Step-1, A[i + 1 : n] does not contain x

Loop Invariant Factorial

- Given a positive integer (n), develop an algorithm to calculate the factorial of (n). Identify a loop invariant to prove the correctness of your algorithm.
- Solution:
  - **Identify the goal of the loop and write it as a post condition**
    - The goal of the loop should be something like "Result = product of all numbers between (1) and (n)". This is indeed the definition of factorial for example the factorial of (5) is (1 x 2 x 3 x 4 x 5)

```
1  //Post condition: R = product of (1 ... n)
```

  - **Write the loop specifying the guard (loop condition)**

```
1  While (k < n)
3  {k++;}
4   //Post condition: R = product of (1 ... n)
```

Loop Invariant Factorial

- **Fix the initialization so that the loop invariant evaluate to true**
  - When (n = 1) no multiplication is needed, and the result evaluates to (1) so let us initialize the result as (R^{C++} = 1).
  - This requires us to initialize (k = 1). With (R = 1 and k = 1) the invariant holds true before entering the loop because we need to multiply the numbers between (1) and (1) which is only (1) sounds silly.

```
1  int R = 1;
2  int k = 1;
3   //Invariant: R = 1 * ... * k
4   While (k < n)
5  {k++;
6   //Invariant: R = 1 * ... * k
7  }//Post condition: R = product of (1 ... n)
```

- **Figure out how to achieve the goal by filling in the body of the loop**
  - Calculate the product of all numbers between (1) and (k), that is **R=R*K**, This statement depends on the loop index (k) and to increment it at the right place.
  - The reason why it should be after we increment the loop index (k) is because it makes **the loop invariant hold true after each iteration**
  - **For example**, if we put it before we increment the loop index (k) the loop invariant keeps true until we exit the loop, at that point the last number (n) is not included in the product which is wrong. Note also that we can put **(R = R * k)** before we increment the loop index (k) only if we change the load guard from **(k < n) to (k <= n).**

Loop Invariant

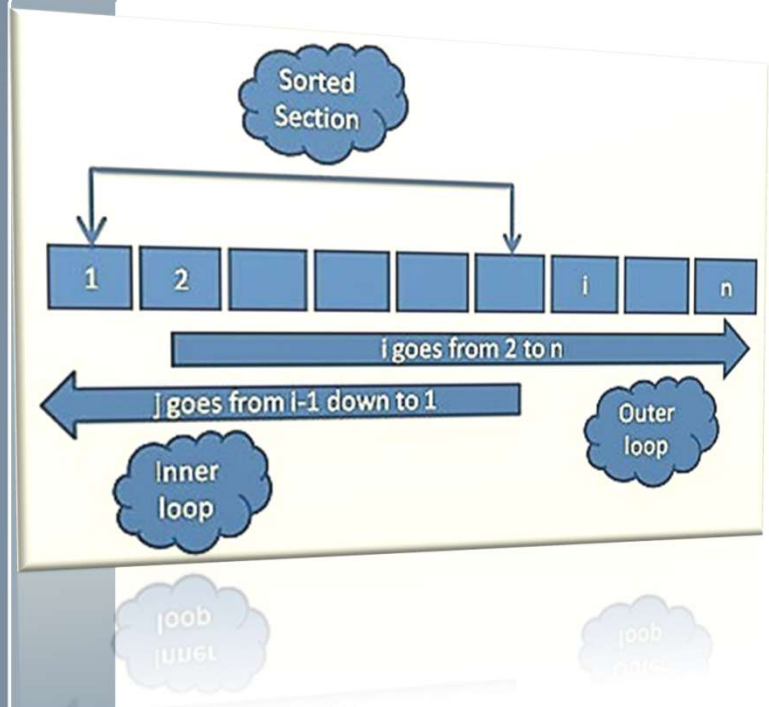
Example: Exponentiation

```
1 // compute x^n
2 int exponentiate(int x, int n){
3   int i = 0, ans = 1;  // initialisation
4   for(int i = 0; i < n; i++){
5         ans = ans * x;
6   }
7   return ans;
8 }
```

- Loop invariant: ans = x^i
  - is it relevant to what the loop is trying to accomplish?
  - does it kinda tell you the progress with your computation?

- Before the first iteration
  - i = 0, ans = 1
  - so ans = x^i
- At the start of iteration k
  - i = k, ans = x^k
  - so ans = x^i
- At the end of iteration k
  - i = k+1, ans = x^{k+1}
  - so ans = x^i
- At the very end
  - , i = n, ans = x^n
  - so ans = x^n

Example: Loop Invariant (Insertion Sort)



https://www.8bitavenue.com/loop-invariant-bubble-sort/

› Given an array (A) of (n) numbers. Assume the array starts at position (1). Develop an algorithm to sort the array in increasing order using insertion sort.

› (1) **Identify the goal of the loop and write it as a post condition**

```
//Post condition:
//A[k] <= A[k + 1] where 1 <= k < n
//A is a permutation of the input array
```

› (2) **Write the loop specifying the guard (loop condition)**

```
while (i <= n)
{
    while (j >= 1 && Cond)
    {
        j--;
    }
    i++;
}
//Post condition:
//A[k] <= A[k + 1] where 1 <= k < n
//A is a permutation of the input array
```

Condition is False

Example: Loop Invariant (Insertion Sort)

- **Fill in the loop invariant**
  - **Elements to the left of (i) are sorted.**
- **Formal way to describe the Inner Loop**
  - Pick an element A[i] from the unsorted section then save it (because its position will be destroyed by the shift operation).
  - Shift all elements to the right starting from the position of the selected element in the sorted section
  - Insert the saved element into the correct place.
- **Fix the initialization so that the loop invariant evaluate to true**
  - Elements from (1) to (i-1) are sorted. This is given by the outer loop (elements to the left of (i) are sorted)
  - Elements from (i+1) to (i) is an empty set which is trivially sorted
  - Does not apply because j+2 <= i is not true
  - Does not apply because j+2 <= i is not true
  - Trivial
- **Figure out how to achieve the goal by filling in the body of the loop**
  - Save A[i] in (key): int key = A[i]
  - Replace (cond) with (key < A[i])
  - Shift the array one element to the right: A[j+1] = A[j]
  - After the inner loop terminates, insert (key) into its position in the sorted array: A[j+1] = key

https://www.8bitavenue.com/loop-invariant-bubble-sort/

```
int i = 2;
//Outer invariant:
//Elements (A[1] .. A[i-1]) to the left of (i) are sorted
while (i <= n)
{
    //1. Save A[i]
    int key = A[i];

    int j = i-1;

    //Inner invariant:
    //A[1..j] is sorted
    //A[j+2..i] is sorted
    //key < A[j+2] where j+2 <= i
    //A[j] <= A[j+2] where j+2 <= i
    //0 <= j //2. Shift elements to the right
    while (j >= 1 && key < A[j])
    {
        //One shift each iteration
        A[j+1] = A[j];

        j--;

        //Inner invariant:
        //A[1..j] is sorted
        //A[j+2..i] is sorted
        //key < A[j+2] where j+2 <= i
        //A[j] <= A[j+2] where j+2 <= i
        //0 <= j
    }

    //3. Insert key into the right position
    A[j+1] = key;

    i++;

    //Outer invariant:
    //Elements (A[1] .. A[i-1]) to the left of (i) are sorted
}

//Post condition:
//A[k] <= A[k + 1] where 1 <= k < n
//A is a permutation of the input array
```

Loop Invariant [Conditions and Verification}

All Conditions holds true

1. Elements A[1…i-1] are sorted (we know that from the initialization stage). The inner loop implementation indicated above does not modify the first (i-2) elements which means A[1..i-2] are sorted as well so the condition holds
2. A[i...i] is only one element which is trivially sorted
3. (key < A[i] where i <= i) this condition holds true due to the fact that we are inside the inner loop otherwise the inner loop would have been terminated. Note that (key < A[i]) is one of the inner loop guards
4. (A[i-2] <= A[i] where i <= i) this condition holds true because A[i] holds the same value as A[i-1] (due to the shift operation) so the condition becomes A[i-2] <= A[i-1] and this is also true because the first (i-1) elements are sorted
5. Trivial (Unimportant)

Loop Invariant [Conditions and Verification}

All Conditions holds true

- Let us now verify that the inner loop invariant holds true after termination. The inner loop terminates when (j = 0) or (key >= A[i]). Let us start with the **first case:**
- Apply (j = 0) and for simplicity assume (i = n) to the inner loop invariant:
  1. A[1..0] is sorted. This is trivially true because it is an empty array.
  2. A[2..n] is sorted. This implies the post condition i.e.; the whole array is sorted. Elements from (2) to (n) are in place and the first element is indeed in place because point (3) below (which we will prove) says (key < A[2]) which means (key) will be inserted in A[1]
  3. (key < A[2] where 2 <= n) same reasoning as point (3) mentioned earlier in the first iteration
  4. (A[0] <= A[2] where 2 <= n) this does not apply as A[0] is not define (our array starts at (1))
  5. (0 <= 0) this is trivial (unimportant)

Loop Invariant [Conditions and Verification}

All Conditions holds true

- Let us now work on the **second case** when (key = A[i] and for simplicity i = n):
  - Combine the case assumption (key = A[n]) with point (3) (key < A[j+2]) we obtain A[n] = key < A[j+2]
  - Combining this with point (1) (A[1..j] is sorted) and point (2) (A[j+2..n] is sorted) follows that the assignment A[j+1] = key makes A[1..n] sorted

Example: Proof of Partial Correctness

› *Problem Definition:* Finding the largest entry in an integer array.

Precondition P: Inputs include

- n: a positive integer
- A: an integer array of length n, with entries A[0]*; … ;* A[n-1]

Postcondition Q:

- Output is the integer i such that 0 ≤ i < n, A[i] ≥ A[j] for every integer j such that 0 ≤ j < n
- Inputs (and other variables) have not changed

```
int FindMax(A, n)
  i = 0
  j = 1
  while (j < n) do
    if A[j] > A[i] then
      i = j
    end if
    j = j + 1
  end while
  return i
```

Sample Easy Quiz

› Linear-Search (A, n, x)

```
1. Set answer to No-Found
2. FOR each index i, going from 0 to n-1, in order:
3.     IF A[i]=x THEN Set answer to the value of i
4. RETURN answer
```

› Which of the following is the valid Loop Invariant?

A. At the start of the iteration with index i, if answer is an index, then A [answer] = x; otherwise, A[0 : i] does not contain x

B. At the start of the iteration with index i, A[0 : i] does not contain x

C. At the start of the iteration with index i, if A contains x, then A[answer]=x;

Assignment No. 2: CLO-1

Submission Date:

Question 01

```
1  void selection_sort(int arr[]){
2    for (int i = 0; i < n-1; i++){
3      int min_index = i;
4      for (int j = i+1; j < n; j++){
5        if (arr[j] < arr[min_index])
6          min_index = j;
7      }
8      swap(i,min_index);
9    }
10 }
```

Determine Loop Invariant for the given steps of selection sort?
Write detail answer for the inner and outer loop?

Thank You!!!

Have a good day