CSE 331 – Section 2: Loop Reasoning

Loop Invariants

```
{ { P } }
{ { inv: … } }
while (B) { S }
{ { Q } }
```

For the Hoare Triple to be valid:

- The loop invariant must hold initially (`P → inv`)
- The loop body must break and re-establish the loop invariant
  `({ { B ^ inv } } S1 … Sn { { inv } })`
- The loop invariant must imply Q when the loop-condition is false
  `(!B ^ inv → Q)`

1. Proofs. The following method is supposed to return true if its integer argument n is a power of 2 (i.e., 2^k for some k) and false otherwise. Write a suitable invariant and appropriate assertions to prove that it works correctly.

Hint: an extra variable i is included in the code. It is not part of the algorithm, but it might be helpful in your proof. If it is not useful, you can ignore it.

```
/**
 *  isPowerOfTwo takes an integer that is greater than or
 *  equal to zero and returns whether the given number is
 *  the result of some powers of two
 *
 *  @param n Input number.
 *  @requires n >= 1
 *  @return true if n = 2^k for some integer k.
 */
   public boolean isPowerOfTwo(int n) {

   { pre:                                         }
   int i = 0; // extra variable that might be useful

   { inv:                                         }

     while(n%2 == 0) {

     {                                             }
     i = i + 1; // extra variable

     {                                             }
     n = n / 2;

     {            }
     } // end of loop
```

```
    {                                                 }
    boolean answer = (n==1);

    { post:                                           }
    return answer; }
```

2. Find loop invariant and fill out the blanks

```
{R:                     }
public void square(x)
  i = 1
  j = 1

  {P:                                    }
  {I:                                    }
  while(i < x)

    {I ^ B:                                }
    j = j + 2i + 1
    i = i + 1

    {I:                                    }

  {I ^ !B:                                 }
  => {Q:                                   }

  return j
```

3. Indicate whether the proof of correctness fails because:
   A. the invariant does not hold initially
   B. the invariant does not hold after the loop body is executed
   C. the invariant does not imply the post-condition upon termination of the loop

(No explanation necessary)

```
{{ 0 < n <= A.length }}
void reverse(int[] A, int n) {
  i = -1;

  j = n;

{{ i = -1 and j = n }}
{{ Inv: A[0] = A[n-1]$_1$, ..., A[i] = A[n-1-i]$_1$

        and A[j] = A[n-1 j]$_1$, ..., A[n-1] = A[0]$_1$ and j = n-1-i }}

  while (i < j) {

    {{ A[0] = A[n-1]$_1$, ..., A[i] = A[n-1-i]$_1$

       and A[j] = A[n-1-j]$_1$, ...,A[n-1] = A[0]$_1$ and j = n-1-i }}

    i = i + 1;

    {{ A[0] = A[n-1]$_1$, ..., A[i-1] = A[n-1-i+1]$_1$

       and A[j] = A[n-1-j]$_1$, ..., A[n-1] = A[0]$_1$ and j-1 = n-1-i }}

    j = j - 1;

    {{ A[0] = A[n-1]$_1$, ..., A[i-1] = A[n-1-i+1]$_1$

       and A[j+1] = A[n-1-j-1]$_1$, ..., A[n-1] = A[0]$_1$ and j = n-1-i }}

    swap A[i], A[j];

    { A[0] = A[n-1]$_1$, ..., A[i] = A[n-1-i]$_1$

    and A[j] = A[n-1-j]$_1$, ..., A[n-1] = A[0]$_1$ and j = n-1-i }}

  }

{{ A[0] = A[n-1]$_1$, ..., A[n-1] = A[0]$_1$ }}

}
```

4. Fill in an implementation of the following method, `sortedInsert.` It takes one array `dst` and element `src` that will be inserted into `dst` maintaining the sorted order.

Assume `dst` has enough indices to store `src` on top of the original elements in `dst` where n represents the original number of elements inserted into `dst` previously (this is because you'll likely need a way to reference the number of previous inserted elements in `dst` while allowing `dst.length` to be large enough so you can insert `src` without an `IndexOutOfBounds` exception occurring).

Hint: You may need to write another loop other than the one we have given. If you include any other loops in your implementation, you must provide the loop invariant for that loop as well.

```
{{ n >= 0 and n = dst.length - 1 }}

void sortedInsert(int[] dst, int src, int n) {

    int i = 0;

    {{ Inv: dst[0], ..., dst[i-1] < src and dst is sorted }}

    while(______________________________________________________) {










    }





}
```