CSE 331 - Section 2 - Loop Reasoning

1. Fill in the proof of correctness for the following code. You may use both forwards and/or backwards reasoning.

```
{{ n >= 0 and i >= 0 and i + n <= A.length }}
int moveFront(int[] A, int i, int n, int x) {
      int L = i;
      {{ ____________________________________________________ }}
      int R = i + n;
      {{ ____________________________________________________ }}
      {{ Inv: A[i], ..., A[L-1] <= x < A[R], ..., A[i+n-1] }}
      while (L != R) {
            {{ ____________________________________________________ }}
            if (A[L] > x) {
                  {{ ____________________________________________________ }}
                  swap(A[L], A[R - 1]);
                  {{ ____________________________________________________ }}
                  R--;
                  {{ ____________________________________________________ }}
            } else {
                  {{ ____________________________________________________ }}
                  L++;
                  {{ ____________________________________________________ }}
            }
            {{ ____________________________________________________ }}
      }
      {{ ____________________________________________________ }}
      {{ A[i], ..., A[L-1] <= x < A[L], ..., A[i+n-1] }}
      return L-1;
}
```

2. Indicate whether the proof of correctness fails because (1) the invariant does not hold initially, (2) the invariant does not hold after the loop body is executed, or (3) 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 }}
}
```

3. 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 (__________________________________________) {









     }











}
```