Prove the correctness of the following code, that is, prove that the statement labelled "end result" is true.

```
public static void sort(int[] arr) {
  for ( int i = 1 ; i < arr.length ; i++) {

    // loop invariant: arr[0..i-1] (inclusive) is sorted in increasing order

    int elt = arr[i]; // current element to put in place

    // shift - move elements of arr[0..i-1] that are
    // greater than elt one spot to the right
    int shift = i - 1;
    for ( ; shift >= 0 && arr[shift] > elt ;
          shift = shift - 1) {
      arr[shift + 1] = arr[shift];
    }
    // put element in place
    arr[shift + 1] = elt;
  }

  // end result: arr[0..arr.length-1] (inclusive) is sorted in
  //  increasing order

}
```

Discussion:

The key takeaway from this is to see how induction can be used to show correctness of a loop in code. The main ingredient is a statement called a *loop invariant* which, in correctly functioning code, is true at the beginning of each repetition of the loop. Induction is used to show that the invariant holds at the beginning of the first iteration, and that if the invariant holds at the beginning of some iteration, what is done in the loop body is sufficient to make the invariant still hold at the beginning of the next iteration.

Let the loop invariant be as stated: $P(i)$ is statement that `arr[0..i-1]` (inclusive) is sorted in increasing order. (Observe that when the loop ends, $i =$ `arr.length` and $P($`arr.length`$)$ is the statement that `arr[0..arr.length-1]` (inclusive) is sorted in increasing order — i.e. the whole array is sorted. Thus, if the loop invariant holds, we can combine that with the loop termination condition to conclude that the loop works.)

For proof by induction, we need to show $P(1)$ to show that the loop invariant is true at the beginning ($i = 1$ the first time through the loop), and that $P(i) \rightarrow P(i+1)$

(if the invariant holds for i, it still holds after the next repetition of the loop when i has been incremented).

Show $P(1)$. $P(1)$ is the statement that `arr[0..0]` (inclusive) is sorted in increasing order (because $i - 1 = 0$ when $i = 1$). This is just `arr[0]` — the first slot of the array — and one element is automatically in sorted order as there's nothing else for it to be out of order with respect to. So $P(1)$ is true.

Show $P(i) \to P(i+1)$. Assume $P(i)$, that is, assume `arr[0..i-1]` is sorted in increasing order. We need to show that after the next iteration through the loop, `arr[0..i]` is sorted in increasing order.

So what happens in the loop? The inner `for` shifts elements — the idea is to move bigger elements out of the way so that `arr[i]` can be inserted in the correct place in the sorted portion of the array. We need to address three things: that the order of elements originally in `arr[0..i-1]` is not changed (because they were in increasing order), that `arr[i]` is put in the right place amongst the sorted elements, and that no element gets lost because it is overwritten with another.

First, the inner `for` loop shifts elements one spot to the right — `arr[shift + 1] = arr[shift]` — keeping the order of the elements shifted. At some point the loop stops, so we have a situation where the first part of the array hasn't been touched, `arr[shift]` is the rightmost of the untouched elements, and then the elements that were in `arr[shift+1..i-1]` have been copied into `arr[shift+2..i]` with their ordering preserved. (Reasoning about what a chunk of code is doing can be tricky and often takes experience; doing some examples where you make up an array with some numbers in it and trace through the code by hand on those examples can be very helpful for gaining understanding of what is going on.) So the order of the elements originally in `arr[0..i-1]` has changed — they were in increasing order, and they still are.

Next, we need to show that `arr[i]` is put in the right place. It goes in `arr[shift+1]` when the inner `for` loop ends. So what do we know about that spot? The loop condition is `shift >= 0 && arr[shift] > elt`, which means there are two reasons why the loop might have ended: `shift` could have become -1 (and thus `shift >= 0` is no longer true), or `arr[shift] <= elt` (and thus `arr[shift] > elt` is no longer true).

Consider `shift = -1` first. In that case, the loop has shifted everything in `arr[0..i-1]` to the right and all of those elements are greater than `elt`. (If not, the loop would have ended earlier.) That means `elt` should go before everything else, so putting it in `arr[0]` is the right place.

Now consider `arr[shift] <= elt`. The loop has shifted everything in `arr[shift+1..i-1]` to the right and all of those are greater than `elt`. (If not, the loop would have ended earlier.) That means `elt` should go before those elements, but not before `arr[shift]` or anything before it — so `arr[shift+1]` is the right place.

Finally, we need to show that nothing is lost. The inner `for` loop shifts everything in `arr[shift+1..i-1]` to the right, so putting `elt` into `arr[shift+1]` is safe.

So, if `arr[0..i-1]` was sorted before the current loop iteration, `arr[0..i]` will be sorted after and the loop invariant holds. Combined with that $i =$ `arr.length` when the loop exits, we can conclude the end result: `arr[0..arr.length-1]` (inclusive) is sorted in increasing order.

Prove the correctness of the following code, that is, that `hanoi` produces a valid list of moves to move n disks from peg `from` to peg `to`. “Valid” means that only one disk is moved at a time, and it is never the case that a bigger disk is put on top of a smaller disk.

```
public static void hanoi(int n, int from, int to, int spare) {
  if (n == 1) {
    System.out.println("move disk from " + from + " to " + to);

  } else {
    hanoi(n - 1, from, spare, to);
    System.out.println("move disk from " + from + " to " + to);
    hanoi(n - 1, spare, to, from);
  }
}
```

Discussion:

The key takeaway from this is to see how induction can be used to show correctness of a recursive subroutine.

Let $P(n)$ be the statement that `hanoi(n,from,to,spare)` generates a valid list of moves for $n \geq 1$ disks from peg `from` to peg `to`.

For proof by induction, we need to show $P(1)$ and that $P(k) \to P(k+1)$.

Show $P(1)$. When $n = 1$, the solution is simply the one instruction

```
move disk from peg 'from' to peg 'to'
```

This moves the disk from peg `from` to peg `to`, only moves a single disk at a time, and as there is only one disk, it is never the case that there is a bigger disk on top of a smaller disk.

Show $P(k) \to P(k+1)$. Assume $P(k)$, that is, `hanoi(k,from,to,spare)` generates a valid list of moves to move k disks from peg `from` to peg `to`. What happens for $P(k+1)$? The code contains three steps:

```
hanoi(k, from, spare, to);
move disk from peg 'from' to peg 'to'
hanoi(k, spare, to, from);
```

By the induction hypothesis, `hanoi(k, from, spare, to)` moves k disks from peg `from` to peg `spare`. Only a single disk is moved at a time, and amongst the k disks, there is never a bigger disk on top of a smaller disk. There is, however, disk $k + 1$ on the bottom of peg `from`, but as this is bigger than everything else, putting any of the k disks on top of it will not violate the rules.

The instruction

```
move disk from peg 'from' to peg 'to'
```

only moves a single disk and, as peg `to` is empty, does not place a bigger disk on top of a smaller one. It results in the biggest disk being on peg `to`, as desired.

By the induction hypothesis, `hanoi(k, spare, to, from)` moves k disks from peg `spare` to peg `to`. This produces a valid list of moves by the same reasoning as for `hanoi(k, from, spare, to)`, and causes the other k disks end up on peg `to` as desired.

Thus all $k + 1$ disks end up on peg `to` and none of the rules of the game were violated.