Below is a write-up of the proof of the correctness of Bubble Sort, as discussed in lecture. I expect your correctness proofs to be similar in structure and level of detail if you want full credit. But first, some intuition for how we pick our loop invariants (you don't need to include this part in your proofs).

```
BUBBLE-SORT(A)
1     for i = 1 to A.length-1
2           for j = A.length downto i + 1
3                 if A[j] < A[j-1]
4                       exchange A[j] with A[j-1]
```

This is a sorting algorithm, so we are trying to prove the Sorting Problem, that is, that the final list A' is a permutation (reordering) of the original list A such that A'[1] ≤ A'[2] ≤ … ≤ A'[n].

This algorithm has two nested loops. To prove its correctness, we need to prove that when the outermost loop terminates, the list will be in sorted order. But to prove anything about the outer loop, we need to figure out what the inner loop does, and prove it, so that we can use that to prove our outer loop invariant.

As discussed in lecture, each time the outer loop iterates, we noticed that the first iteration of the outer loop places the smallest value in A[1], the second iteration places the 2nd smallest value in A[2] (without disturbing the 1st), and so on. Clearly, once you do that A.length times (or even A.length-1 times), the array is in sorted order. This is what we use as our outer loop invariant: Before the ith iteration, the smallest (i-1) elements in the array are in the first (i-1) indexes, in sorted order.

So, if each outer for loop iteration moves the ith smallest value to A[i] without disturbing A[1...i-1], then how does the inner for loop accomplish that? The basic idea is that each time an exchange happen (or doesn't), it leaves the smaller of the two values on the left. This means that once the inner loop reaches the smallest value in the unsorted part of the list (which is the ith smallest value overall), it will continually move it to the left, because there are no smaller values in that part of the list. So, at each step A[j] (the smallest value in the unsorted part of the list) will be smaller than A[j-1], causing the values to swap. Then that value will become A[j] again when the loop iterates, and the cycle will continue.

This means that we can put a bound on where the smallest value in the unsorted part of the list is at each step of the inner for loop, which becomes our inner loop invariant. The smallest value in the unsorted part must be somewhere in the list A[i...j] at the beginning of each iteration, because if not, that implies that there was a smaller value in A[j+1...A.length] that caused it to stop the chain of exchanges moving it to the left.

Now for the actual proof:

```
BUBBLE-SORT(A)
1     for i = 1 to A.length-1
2           for j = A.length downto i + 1
3                 if A[j] < A[j-1]
4                       exchange A[j] with A[j-1]
```

To show that Bubble-Sort is correct, we must show that the final array A' is a permutation (reordering) of the original array A such that A'[1] ≤ A'[2] ≤ … ≤ A'[n]. At all times, the array A must be a permutation of the original array because the algorithm never introduces any new elements to the array or removes old ones: it only exchanges existing elements. Thus, A' is a permutation of A. Now, we need to show that A'[1] ≤ A'[2] ≤ … ≤ A'[n].

First, we prove that the following loop invariant holds for the inner for loop on lines 2-4 of Bubble-Sort:

Loop invariant: Before any given iteration of the inner for loop, the minimum value in the subarray A[i...A.length] occurs within A[i...j] (at least one of them does, in the case of a tie).

Initialization: We must show the loop invariant holds before the first iteration of the loop. Before the first iteration of the loop, j = A.length. Thus, we must show that the minimum value in the subarray A[i...A.length] occurs within A[i...A.length]. This is trivially true.

Maintenance: Suppose that before a given iteration of the inner for loop, the minimum value in the subarray A[i...A.length] is within A[i...j]. We must show that before the beginning of the next iteration, the loop invariant still holds. There are three cases to consider:

Case 1: The minimum value in the subarray A[i...A.length] occurs within A[i...j-2]. Then the exchange step does not impact the minimum value, so the minimum value still occurs within A[i...j-2] after the iteration.

Case 2: The minimum value in the subarray A[i...A.length] is at A[j-1]. Then the A[j-1] ≤ A[j], since A[j-1] is the minimum, so the exchange step doesn't happen. Thus the minimum value still occurs within A[i...j-1] after the iteration.

Case 3: The minimum value in the subarray A[i...A.length] is at A[j]. Then A[j] ≤ A[j-1], since A[j] is the minimum, so an exchange occurs and the minimum is now within A[i...j-1]. If A[j] == A[j-1], then an exchange does NOT occur, but they're the same value so it doesn't matter; there is still at least one occurrence of the minimum value of A[i...A.length] within A[i...j-1].

Since under all of these cases, the minimum value of A[i...A.length] occurs within A[i...j-1] after the iteration, when j decrements at the beginning of the next iteration, the minimum value of A[i...A.length] occurs within A[i...j]. So, the loop invariant holds for the next iteration.

Termination: The inner loop terminates just before the iteration where j = i. So, by the loop invariant, the minimum value of A[i...A.length] occurs within A[i...i], which means that the

minimum value of A[i...A.length] is A[i].

Next, we prove that the following loop invariant holds for the outer for loop, on lines 1-4:

Loop invariant: At the start of each iteration, the smallest i-1 elements are in the first i-1 places in the array, in sorted order.

Initialization: Before the first iteration, i = 1. “The smallest 0 elements are in the first 0 places in the array, in sorted order” is vacuously true.

Maintenance: Suppose that before a given iteration of the loop, the smallest i-1 elements were in the first i-1 places in the array, in sorted order. The inner for loop cannot alter any elements in the subarray A[1...i-1] because it only ever alters elements A[j] and A[j-1], and during all iterations of the inner for loop, $j \geq i+1$ by the loop bounds. Thus, the smallest i-1 elements are still in the first i-1 places in the array, in sorted order, after the iteration completes. So, because A must be a permutation of the original array at all times, the smallest element in the subarray A[i...A.length] is the ith smallest element overall. By the inner for loop invariant, after the iteration, A[i] is the smallest element of the subarray A[i...A.length]. This means that the smallest i elements are in the first i places in the array, in sorted order. At the start of the next iteration, i increments, so smallest i-1 elements are in the first i-1 places in the array, in sorted order. Thus, the loop invariant holds before the next iteration.

Termination: After the end of the loop, i = A.length. So, the smallest A.length-1 elements of the array are in the first A.length-1 places, in sorted order. This means that the whole array is in sorted order, since the last element must be the maximum.

We know that $A'[1] \leq A'[2] \leq \ldots \leq A'[n]$ because we proved that the array is in sorted order after the outer for loop by the outer for loop invariant, and the algorithm ends immediately after the outer for loop terminates.

We have shown that the final array A’ is a permutation (reordering) of the original array A such that $A'[1] \leq A'[2] \leq \ldots \leq A'[n]$. Thus, Bubble-Sort solves the Sorting Problem, and is correct.