1. Loop invariants

Given that

```
 l < r:
```

1. small < r not an invariant: if all items in the partition range are smaller than the pivot, after the last loop iteration small = r.

2. small < large: not an invariant for the same reason as above.

3. small <= large: this is an invariant: before the loop executed small < large (because l < r) and after the last iteration, small = large.

4. for all i such that l <= i < small, arr[i] < pivot: this is a loop invariant. Before the loop is executed, there are no such i. After each iteration, items to the left of small are less than the pivot.

5. for all j such that large <= j <= r, arr[j] >= pivot: this is a loop invariant. Before the loop is executed, large = r and the value at this index is the pivot, so arr[j] = pivot. After each iteration, items at the index large and to the right are greater or equal to the pivot.

6. for all j such that large < j <= r, arr[j] >= pivot: this is also a loop invariant. It is a weaker statement than 5 so if 5 is true then 6 is true.

2. Correctness of selection sort

Algorithm

```
void  selectionSort(int arr[], int len){
   int i;
   int j;
   int temp;
   int pos_greatest;

   for( i = len - 1; i > 0; i--){
      pos_greatest = 0;
      for(j = 0; j <= i; j++){
         if( arr[j] > arr[pos_greatest]){
            pos_greatest =  j;
         }//end if
      }//end inner for loop
      temp = arr[i];
      arr[i] = arr[pos_greatest];
      arr[pos_greatest] = temp;
   }//end outer for loop
}//end selection sort
```

Outline of the proof

We just formalise what we know the selection sort is doing, namely: at every passage through the inner loop, we find the largest element in the range [0,...,i]; the outer loop swaps that element to position i and decrements i. So the first passage through the outer loop swaps the largest element in the range [0,...,n-1] to position n-1, the next one swaps the largest element in the range [0,...,n-2] to position n-2 etc. There are many properties which stay invariant for the outer loop, and may be ours is not the most obvious choice, but let's say that the invariant of the outer loop is that the part of the array to the right of the loop counter is sorted in ascending order and that the numbers to the right are greater or equal to the numers remaining in the unsorted part. When we are finished this means that the whole array is sorted in ascending order.

Invariant of the inner loop

First we need to formulate precisely and prove the loop invariant of the inner loop: *All numbers in the array up to and not including the position j are are less or equal to the number at pos_greatest (for all k < j, arr[k] <= arr[pos_greatest]).*
Proof:

- true before the first iteration (no numbers before position 0)
- Suppose it is true at the start of iteration j, namely for all k < j, arr[k] <= arr[pos_greatest]. Now we look at arr[j]. If it is greater than arr[pos_greatest] we reset pos_greatest to be j and it now holds the index of the greatest number between 0 and j (inclusive). Otherwise we just increment j and again pos_greatest holds the index of the greatest number between 0 and j (inclusive). So in either case we have for all k <= j, arr[k] <= arr[pos_greatest]. After we increment j this becomes again for all k < j, arr[k] <= arr[pos_greatest].

When the inner loop terminates, i < j. So the invariant of the inner loop implies that for all k <= i, arr[k] <= arr[pos_greatest], in other words, pos_greatest is the index of the greatest item between 0 and i (inclusive).

Invariant of the outer loop

Invariant: *elements at positions strictly to the right of i to len-1 are sorted in ascending order and are greater than the elements to the left of i. More formally: for all k with i < k <= len-1, arr[k] <= arr[k+1]<=... <= arr[n-1] and for all m with 0 <= m <= i, arr[m] <= arr[k].*

Proof that this property is an invariant of the outer loop:

- before the first passage through the loop: there are no k with i < k <= len - 1, so the statement is trivially true.
- Suppose that the property is true before the iteration when i equals say x. We need to show that the same property will be true after the iteration when i will be equal x-1.
  Before the iteration we had:
  (1) for all k with x < k, arr[k] <= arr[k+1]<=... <= arr[n-1]
  (all numbers strictly to the right of x are ordered)
  and
  (2) for all k with x < k and for all m with m <= x, arr[m] <= arr[k].
  (all numbers to the left of x including x are smaller than the numbers on the right)
  All that happens in the iteration is: the inner loop executes and finds the index of the greatest number between 0 and x (inclusive). That number is then swapped to the position x. From (2) we know that this number is smaller than all numbers to the right of x and from (1) we know that those numbers are ordered in ascending order, so now we have
  (1') for all k with x <= k, arr[k] <= arr[k+1]<=... <= arr[n-1]
  (all numbers to the right of x including x are ordered)
  We also know that the number at x is the greatest in the range from 0 to x, so we have (2') for all k with x <= k and for all m with m < x, arr[m] <= arr[k]
  (all numbers to the left of x are smaller than the numbers on the right)
  Finally the outer loop decrements i so we have to reformulate (1') and (2') for the new value of i which is x-1: (1) for all k with len-1 >= k > i, arr[k] <= arr[k+1]<=... <= arr[n-1]
  (all numbers strictly to the right of i are ordered)
  (2) for all k with len - 1 >= k > i and for all 0 <= m <= i, arr[m] <= arr[k].

And finally...

When i = 0 the invariant of the outer loop becomes
arr[0] <= arr[1] <= ... <= arr[len-1] so the array is sorted.