Programming Assignment 4 Solutions

When writing a function its always a good idea to `return` as soon as one knows the answer.

```
/* le_seg: x <= A[lo,hi) */
bool le_seg(int x, int[] A, int lo, int hi)
/*@requires 0 <= lo && lo <= hi && hi <= \length(A); @*/
{
    for(int i=lo; i<hi; i=i+1)
    //loop invariant: x <= A[lo, i)
    {
        if(x>A[i])
        {
             return false;
        }
    }
    //@assert i==hi;
    //LI gives x <= A[lo, hi)
    return true;
}

/* good_partition: A[lo, i) <= A[i, hi) */
bool good_partition(int[] A, int lo, int i, int hi)
/*@requires 0 <= lo && lo <= i && i <= hi && hi <= \length(A); @*/
{
    for(int j=lo; j<i; j=j+1)
    {
        if( !le_seg(A[j], A, i, hi)) //when le_seg fails for A[j]
        {
            return false;
        }
    }
    return false;
}
```

Coding `le_seg` using Loop Invariants

And giving a correctness outline along the way ...

Both the problems above have a common pattern. Lets say we are trying to solve `le_seg(x, A, lo, hi)`, for this we need to check whether `x <= A[i]`, for i in `[lo, hi)` (by

definition). So we set up a loop on line 5. Lets look at the hint loop-invariant `LI: x <= A[lo, i)`, the loop invariant is obviously true in the INIT step. Lets look at the EXIT condition (when we have exited the loop and reached the line after loop body. The loop invariant about loop variable is `lo <= i <= hi`. Combining it with the negation of loop guard at the exit condition gives us `i==hi`, substituting this in LI gives us `x <= A[lo, hi)`, therefore we must return true after the loop (this gives us line 15). Next, we need to fill in the loop body. Ofcourse, we need to compare `x` with `A[i]`. Therefore we may put the following at line 8 above:

```
1  if( x <= A[i] )
2  {
3      //if-part: do something;
4  }
5  else
6  {
7      //@assert x > A[i];
8      //else part: do something;
9  }
```

The `else-part` above is very easy to decide. If x is strictly more than A[i] then we must `return false`, as the i^th index violates the test of `x <= A[lo, hi)`. What about the `if-part`? We definitely can not return false, because `x<=A[lo, i+1)` and its possible that for the rest of the elements we have `x <= A[i+1, hi)`. For a similar reason we can not return true also. Therefore, we must do nothing in the `if-part` above. Thats it, we find that the loop terminates and the LI holds. Therefore, we have the following loop:

```
1  if( x <= A[i] )
2  {
3      //if-part: do nothing;
4  }
5  else
6  {
7      //@assert x > A[i];
8      return false;
9  }
```

Which can be easily converted to the following, which gives us the answer.

```
1  if( x > A[i] ) {
2      return false;
3  }
```

For coding `good_partition` we need to check `A[lo, i) <= A[i, hi)`. This is equivalent to checking if `A[x] <= A[i,hi)` for each x in `[lo, i)`. The condition `A[x] <= A[i, hi)` can be

given by `le_seg(A[x], A, i, hi)`, and the rest of the argument is similar again.

Another approach to coding `le_seg`

We know that the definition of `x <= A[lo, hi)` requires us to iterate over the range of indices `{lo, lo+1, ..., hi-1}`, so that we can compare each element against `x`. This gives us the following loop:

```
1  bool le_seg(int x, int[] A, int lo, int hi) {
2      for(int i=lo; i<hi; i=i+1)
3      {
4          //we can use A[i] here.
5      }
6  }
```

Next, one has to try out some inputs and try to guess the pattern in the solution. Lets say we have `A[lo,hi) = {4, 5, 1, 2}` and `x=3`. Using the loop allows us to scan the list `A[lo, hi)` from left to right. So we try our pen-and-paper approach, and go through each element one by one. First, we check whether `x<=4`, as this is true, we move onto the next element in the list. As long as the condition `x <= (current element of A)` is satisfied we move along to the next element in the list, if we were able to arrive at the end of the list in this way then we return true. If this is not so, then the condition fails and we have `x>A[i]` for some `i`, we return false here. This idea also gives us the same code.

find_min

Recall that to find a minimum element in an array `A` of length `n` we do the following:

```
1  int minValue(int[] A, int n) {
2      int min = A[0];
3      for(int i = 0; i<n; i++) {
4          if (A[i] < min) {
5              min = A[i];
6          }
7      }
8      return min;
9  }
```

Exercise The code above has similar loop-invariants as `le_seg`, find them.

However, the `find_min` function which we are required to code requires the index of `min`. We just need to remember that index each time we set the value of `min`. Therefore, we declare a new int variable to store the `minIndex` and update it each time we set the value of `min` (lines 2, and 5 above). This

gives us the solution:

```
int find_min(int[] A, int lo, int hi) {
    int min = A[lo];
    int minIndex = lo;
    for(int i=lo; i<hi; i=i+1) {
        if(A[i]< min) {
            min = A[i];
            minIndex = i;
        }
    }
    return minIndex;
}
```

swap

We have seen this many times in ICS, the same solution works.

```
void swap(int[] A, int i, int j)
{
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

count_le

```
//partition2: partitions an array segment A[lo, hi) around the pivot element A[
// and returns the index of pivot in the partitioned array.
int partition2(int[] A, int lo, int pi, int hi)
//@requires 0 <= lo && lo <= pi && pi < hi && hi <= \length(A);
//@ensures lo <= \result && \result < hi;
// must ensure A[\result] >=  A[lo, \result)
/*@ensures le_seg(A[\result], A, \result+1, hi); @*/
{
    int pivot = A[pi];

    //create an extra copy of elements of A
    //we have Acopy[i-lo] = A[i] when i is in [lo, hi)
    int[] Acopy = alloc_array(int, hi - lo);
    for(int i = lo; i<hi; i++)
        Acopy[i-lo] = A[i];

```

```
    //the index for smaller elements is left
    int left = lo;
    //the index for larger elements is right
    //right index stores the index just after the final location of pivot.

    /*
    if lo = 0, then the pivot must go to index count_le(pivot, A, lo, hi) - 1.
    For a general value of lo, we need to shift the index above accordingly.

    The question we need to answer is, lets say we have k elements, with the
    first element occuring at index lo, and the other k-1 after it,
    what is the index of the last element?

    Clearly, the index lo element has k-1 elements after it, hence the index
    of the last element is lo+k-1. Here we need to put
    k = count_le(pivot, A, lo, hi)
    as that is the number of elements in the left half of the partitioned
    array including the pivot.
    */
    int finalPivot = (lo + count_le(pivot, A, lo, hi)) - 1;
    int right = finalPivot + 1;

    /* process elements of Acopy one by one  */
    for(int j = 0; j < hi-lo; j++)
    {
        /*while scanning the copy, make sure that the
          pivot goes to the correct place, we know that
          the pivot is stored at location pi-lo in Acopy.
        */
        if(j==(pi-lo))
        {
            A[finalPivot] = Acopy[j];
        }
        else { // non-pivot elements must go to the correct side
            if (Acopy[j] <= pivot)
            //put Acopy[j] left of pivot in A[lo, hi)
            {
                A[left] = Acopy[j];
                left = left + 1;
            }
            else
            //put Acopy[j] right of the pivot in A[lo, hi)
            {
                //@assert Acopy[j] > pivot;
                A[right] = Acopy[j];
                right = right + 1;
            }
        }
```

```
65      }
66      //return final location of pivot
67      return finalPivot;
68  }
```