1    def avg(A):
2        
3        Pre: A is a non-empty list of numbers
4        Post: Returns the average of the elements of A
5        
6        sum = 0
7        i = 0
8        while i < len(A):
9            sum += A[i]
10           i += 1
11        return sum / len(A)

1    def iter_exp_slow(a, b):
2        
3        Pre: a, b are natural numbers (but can relax domain of a)
4        Post: returns a^b (exponentiation)
5        
6        x = 1
7        y = b
8        while y > 0:
9            x *= a
10           y -= 1
11        return x

1    def iter_exp_fast(a, b):
2        
3        Same specifications as iter_exp_slow
4        
5        x = 1
6        y = b
7        m = a
8        i = 0 // Note: i isn't necessary for the program, but helps with the proof
9        while y > 0:
10           if y % 2 == 1:
11              x *= m
12              y = (y-1) / 2
13           else:
14              y = y / 2
15              m = m * m
16              i += 1
17        return x
def in_place_partition(A, x):
    # Pre: A is a list (of numbers)
    # Post: A is "partitioned" with respect to x.
    # That is, the order of elements of A is changed so that
    # for some index i, A[0..i-1] contains only elements <= x,
    # and A[i..len(A)-1] contains only elements > x.
    #
    divider = 0
    checked = 0
    while checked < len(A):
        if A[checked] <= x:
            swap A[checked], A[divider]
            divider += 1
        checked += 1

Loop invariant for in_place_partition

As we saw in class, a good loop invariant is

$$Inv(A, checked, divider) : checked \leq len(A) \land A[0..divider-1] \leq x \land A[divider..checked-1] > x,$$

where “$\leq x$” here means every element in the list slice is $\leq x$, and similarly for the “$> x$”.

We first note that the first time the loop is reached, the invariant holds; both divider and checked are equal to 0, so checked $\leq len(A)$, and the two list slices in the invariant are both empty, and hence the invariant is vacuously true.

Now consider a single iteration of the loop. Let $A_0, c_0, d_0$ and $A_1, c_1, d_1$ be the values of the corresponding variables $A$, checked, and divider, before and after the iteration runs, respectively. We assume that $Inv(A_0, c_0, d_0)$ holds, and want to prove that $Inv(A_1, c_1, d_1)$ also holds.

We consider the two paths through the loop separately.

- Let’s do the easier one first: assume $A_0[c_0] > x$. In this case, only the variable checked changes; we get that $A_1 = A_0$, $d_1 = d_0$, and $c_1 = c_0 + 1$. I’ll leave checking the first part of the invariant ($c_1 \leq len(A)$) as an exercise, as it’s very similar to a previous example from lecture. The second part of the invariant certainly holds, because our assumption $Inv(A_0, c_0, d_0)$ tells us that $A_0[0..d_0-1]$ contains only elements $\leq x$, and $A_1 = A_0$ and $d_1 = d_0$, i.e., these values are unchanged.

What about the third part: is every number in $A_1[d_1..c_1-1]$ greater than $x$? Since $A_1 = A_0$ and $d_1 = d_0$, we know from our assumption that $A_1[d_1..c_0-1]$ are all $> x$. Since $c_1 = c_0 + 1$, the only element we need to check is $A_1[c_1-1] = A_0[c_0]$. But on this program path, we explicitly made this check at line 12: we know that $A_0[c_0] > x$.

- The other program path is quite similar. If $A_0[c_0] \leq x$, then $d_1 = d_0 + 1$, $c_1 = c_0 + 1$, and two elements of $A$ are swapped (note that this happens before the index variables are incremented), so $A_1[c_0] = A_0[d_0]$ and $A_1[d_0] = A_0[c_0]$, and all other elements of $A$ are unchanged.

Since the reasoning is similar to the first case, we won’t do a “full” proof here; instead, we’ll do a sketch and let you fill in some details. For the second part of the invariant, we need to check $A_1[0..d_1-1] = A_1[0..d_0]$. Since this iteration didn’t change $A[0..d_0-1]$ (so $A_0$ and $A_1$ agree on this slice), the only element to check is $A_1[d_0]$. You can justify this because of the return path we’re on.

For the third part of the invariant, we need to check $A_1[d_1..c_1-1] = A_1[d_0 + 1..c_0]$. Note that the elements in $A[d_0 + 1..c_0 - 1]$ didn’t change, so the only element we need to check here is $A_1[c_0]$. You can justify this because of the loop invariant assumption.

Finally, we should show that when the loop terminates (assuming it does, for now), we can use the loop invariant to prove the postcondition. But that’s pretty easy; combining the loop invariant with the fact that for the loop not to run, we cannot have checked $< len(A)$, this means that checked = len(A), and by the invariant, $A[0..divider-1]$ are all $\leq x$, and $A[divider..len(A)-1]$ are all $> x$, and this is exactly what’s required by the postcondition. Note that a real partition implementation for quicksort would also return divider.