15-122: Principles of Imperative Computation

Recitation 8 Solutions

Josh Zimmerman

Quicksort

```
#use <rand>
#use "sortutil.c0"

int abs(int x)
{
  if (x <0)
    return -x;
  return x;
}

int partition(int[] A, int lower, int pivot_index, int upper)
//@requires 0 <= lower && lower <= pivot_index;
//@requires pivot_index < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures ge_seg(A[\result], A, lower, \result);
//@ensures le_seg(A[\result], A, \result+1, upper);
{
    int pivot = A[pivot_index];
    swap(A, pivot_index, upper-1);

    int left = lower;
    int right = upper-2;
    while (left <= right)
    //@loop_invariant lower <= left && left <= right+1 && right+1 < upper;
    //@loop_invariant ge_seg(pivot, A, lower, left);
    //@loop_invariant le_seg(pivot, A, right+1, upper-1);
    {
        if (A[left] <= pivot) {
            left++;
        } else { //@assert A[left] > pivot;
            swap(A, left, right);
            right--;
        }
    }
    //@assert left == right+1;
    //@assert A[upper-1] == pivot;

    swap(A, left, upper-1);
    return left;
}

void qsort(int[] A, int lower, int upper, rand_t gen)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
    if (upper - lower <= 1) return; // already sorted

    int pivot_index = lower + abs(rand(gen) % (upper-lower));
    /* pivot_index = upper-1 or pivot_index = 0 gives O(n^2) for sorted array */
    /* pivot_index = lower+(upper-lower)/2 efficiency depends on input distribution */
    /* better yet would be: median of 3 random elements */

```

```
53   int mid = partition(A, lower, pivot_index, upper);
54   qsort(A, lower, mid, gen);
55   qsort(A, mid+1, upper, gen);
56   return;
57 }
```

Practice!

1. Rank these big-O sets from left to right such that every big-O is a subset of everything to the right of it. (For instance, $O(n)$ goes farther to the left than $O(n!)$ because $O(n) \subset O(n!)$.) If two sets are the same, put them on top of each other.

Solution:

$O(4)$	$O(\log(\log(n)))$	$O(\log(n))$	$O(\log^2(n))$	$O(n)$	$O(n\log(n))$	$O(n^2+20000n+3)$	$O(2^n)$	$O(n!)$
$O(1)$				$O(4n+3)$		$O(n^2)$		

2. Using the formal definition of big-O, prove that $n^3 + 300n^2 \in O(n^3)$.

Solution: $n^3 + 300n^2 \leq n^3 + 300n^3$ for all $n > 1$. $n^3 + 300n^3 = 301n^3$. So, for all $n > 1$, $n^3 + 300n^2 \leq 301n^3$. We have $n_0 = 1$, $c = 301$ if we want to plug back in to the formal definition.

3. Using the formal definition of big-O, prove that if $f(n) \in O(g(n))$, then $k * f(n) \in O(g(n))$ for $k > 0$.

One interesting consequence of this is that $O(\log_i(n)) = O(\log_j(n))$ for all i and j (as long as they're both greater than 1), because of the change of base formula. So, it doesn't matter what base we use for logarithms in big-O notation.

Solution: Since $f(n) \in O(g(n))$, we know that there exist some $n_0 \in \mathbb{R}$ and $c \in \mathbb{R}^+$ such that $f(n) \leq c * g(n)$ for all $n > n_0$.

We can multiply both sides by k to obtain $k * f(n) \leq k * c * g(n)$ for all $n > n_0$.

So, if we set $c_1 = k * c$, then we know that $k * f(n) \leq c_1 * g(n)$ for all $n > n_0$. Thus, $k * f(n) \in O(g(n))$.

4. Let's prove that `partition` is correct. (Note: since `qsort` is a recursive function, it requires a slightly different proof format and we won't discuss it in recitation. Instead of loop invariants, we'd have to base our proof entirely on preconditions and postconditions of `qsort` and `partition`. Recursive programs generally have relatively clean proofs by induction that they're correct. If you don't know what induction is yet, that's fine.)

Solution:

Preconditions imply loop invariants.

1st loop invariant: On line 11, we set `left = lower`. So, `lower <= left`. By the preconditions, we know that `lower < upper`, so therefore `lower <= upper - 2 + 1`, and so `left <= right + 1`. Since `right = upper - 2`, we know that `right + 1 < upper`.

2nd loop invariant: since `lower == left`, this is true because the subarray of A we're considering is empty and thus `pivot` is greater than or equal to everything in it.

3rd loop invariant: `right + 1 == upper - 1`, so this must be true, since the array we're looking at is empty and thus `pivot` is less than or equal to everything in it.

Preservation of loop invariants.

Let's case on whether `A[left] <= pivot`.

Case 1: `A[left] <= pivot`. In this case, we know that `left' == left + 1` and all other variables are unchanged.

1st loop invariant: Since `lower <= left`, `lower <= left + 1`, so `lower <= left'`. By the loop exit condition (or loop guard), we know that `left <= right`. Therefore, `left + 1 <= right + 1` or `left' <= right' + 1`. Finally, `right` is unchanged, so we still have `right' + 1 < upper`.

2nd loop invariant: Since `pivot` is greater than or equal to everything in `A` from `lower` to `left - 1` (inclusive) by the loop invariant, and `A[left] <= pivot`, we know that `pivot` is greater than or equal to everything in `A` from `lower` to `left` (inclusive). Thus, `ge_seg(pivot, A, lower, left')` holds.

3rd loop invariant: `upper` and `right` are unchanged, so this is still true.

Case 2: `A[left] > pivot`. In this case, `right' == right - 1`, and `A[left]` and `A[right]` are swapped.

1st loop invariant: Since `left` is unchanged, we still know that `lower <= left`. Since `left <= right` (by the loop guard), `left <= right - 1 + 1`, so `left <= right' + 1`. Since `right' < right`, we know that `right' + 1 < upper` (since `right < upper`).

2nd loop invariant: The swap doesn't change any array element from index `lower` to index `left - 1` (inclusive), so this loop invariant is unaffected.

3rd loop invariant: First, note that `right' + 1 == right`. Next, note that the old element at index `left` was larger than pivot, and that that element is now at `A'[right]`. Since the loop invariant was true at the start of the loop, we know that `pivot` is less than or equal to every element at index at least `right + 1`. We know that `pivot` is less than the element now at `right` (since we're in this case), so the loop invariant still holds.

Loop invariants and negated loop guard imply postcondition.

First, we should show that the `//@assert` statements hold.

The negated loop guard tells us that `left > right` (and so `left >= right + 1`) The loop invariant tells us that `left <= right + 1`. Thus, `left == right + 1`.

We never touched `A[upper - 1]`, by the first loop invariant and the fact that `pivot == pivot` (so we'd never try to swap it). Thus, `A[upper - 1] == pivot`.

Now, we swap `upper - 1` and `left`. We know by the third loop invariant and the first assert statement that `pivot <= A[right + 1]` (and thus `pivot <= A[left]`) and that `pivot` will now be `A[left]`. Note that `pivot <= A[upper - 1]` now.

1st postcondition: we know that `lower <= left` by the first loop invariant. We also know that `left == right + 1` and `right + 1 < upper`, by the first loop invariant. Thus, this postcondition is true.

2nd postcondition: By the second loop invariant, we know `ge_seg(pivot, A, lower, left)` is true. But `A[\result ] == pivot`, so the postcondition must be true.

3rd postcondition: By the third loop invariant, we know `le_seg(pivot, A, right + 1, upper - 1) is true`. Further, based on the swap we did on line 30, we know that `pivot < A[upper - 1]`.

So, we know that `le_seg(pivot, A, right + 1, upper) is true`. `left == right + 1`, so the third postcondition is true.

Termination:

We only enter the loop if `left <= right`, by the loop guard.

At each iteration, we either increment `left` or decrement `right`. Therefore, we'll eventually get to a point when `left > right`. At this point, we exit the loop.

Thus, `partition` is correct.