15-122: Principles of Imperative Computation

Recitation 8

Josh Zimmerman

There is a good visualization of both Selection Sort and Quicksort at
`http://www.cs.usfca.edu/~galles/visualization/flash.html`.

Quicksort

Quicksort is a fast ($O(n \log n)$ on average, $O(n^2)$ worst case) sorting algorithm. The general idea behind quicksort is that we start with an array we need to sort, and then select a *pivot* index. (There are many different methods of picking a pivot index: some strategies always pick index 0 or the largest possible index in the array, some choose a randomly generated index, and some randomly generate three indices, look at the values at those indices, and pick the index of the median value. The different strategies affect the performance of quicksort in different ways.)

After we pick a pivot index, we rearrange the array so that everything to the to the left of the pivot index in the array is less than or equal to `A[pivot_index]` and everything on the right of the pivot index in the array is greater than `A[pivot_index]`. Then, we call quicksort recursively on the portion of the array that is less than or equal to the pivot and on the portion that is greater than the pivot. If the list is empty or has one element, we just return the list since it's already sorted.

Quicksort can be hard to understand at first, so let's look at an example.

Start with the array `[4, 2, 1, 0]`. We randomly select `1` as our pivot index. We switch `A[0]` and `A[1]`, which gives us `[2, 4, 1, 0]`.

We split the array so everything on the left is less than or equal to the pivot. After we've done that, we have `[2, 0, 1, 4]`. We then swap the pivot, 2, back to the place where we now know it belongs in the final array – index 2. Therefore we have `[1, 0, 2, 4]` Then, we recursively sort the two portions of the array: `[1, 0]` and `[4]`.

First, we sort `[1, 0]`. We must choose `0` as our pivot index, switching index 0 with index 0 keeps the array exactly the same: Our switch then results in the array `[1, 0]`. Then, we can swap the pivot, `1`, to the place where it belongs in the final array – index 1.

Then, we recursively sort the array `[0]`. It has only one element, so it's sorted already.

Next, we recursively sort the array `[]` (everything in `[0, 1]` that's larger than `1`). The empty array is already sorted.

Now, we know that `[0, 1]` is sorted, so we can sort `[4]`.

`[4]` is already sorted, so we know our whole array, `[0, 1, 2, 4]` is now sorted.

Now, let's look at the code for quicksort and see what we can say about its correctness:

```
1 int partition(int[] A, int lower, int pivot_index, int upper)
2 //@requires 0 <= lower && lower <= pivot_index;
3 //@requires pivot_index < upper && upper <= \length(A);
4 //@ensures lower <= \result && \result < upper;
5 //@ensures ge_seg(A[\result], A, lower, \result);
6 //@ensures le_seg(A[\result], A, \result, upper);
7 {
8  // Hold the pivot element off to the left at "lower"
9  int pivot = A[pivot_index];
10  swap(A, lower, pivot_index);
11
12  int left = lower+1; // Inclusive lower bound (lower+1, pivot's at lower)
13  int right = upper; // Exclusive upper bound
14
15  while (left < right)
16   //@loop_invariant lower+1 <= left && left <= right && right <= upper;
17   //@loop_invariant ge_seg(pivot, A, lower+1, left); // Not lower!
18   //@loop_invariant le_seg(pivot, A, right, upper);
19   {
20    if (A[left] <= pivot) {
21     left++;
22    } else {
23     //@assert A[left] > pivot;
24     swap(A, left, right-1); // right-1 because of exclusive upper bound
25     right--;
26    }
27   }
28  //@assert left == right;
29
30  swap(A, lower, left-1);
31  return left-1;
32 }
33
34 void sort(int[] A, int lower, int upper)
35 //@requires 0 <= lower && lower <= upper && upper <= \length(A);
36 //@ensures is_sorted(A, lower, upper);
37 {
38  if (upper - lower <= 1) return;
39  int pivot_index = lower + (upper - lower)/2; // Pivot at midpoint
40
41  int new_pivot_index = partition(A, lower, pivot_index, upper);
42  sort(A, lower, new_pivot_index);
43  //@assert is_sorted(A, lower, new_pivot_index + 1);
44  sort(A, new_pivot_index + 1, upper);
45 }
```

Questions

1. What would go wrong if the `partition` function ignored `pivot_index` and picked a new pivot?
2. Why is swapping the pivot with `left-1` the right thing? Why is `left` wrong? Why is `left-1` safe?
3. Prove that the that `partition` function is correct.
4. Could you change `partition` (code and/or loop invariants) in order to justify one of the postconditions on line 5 or 6 being `gt_seg` or `lt_seg`?
5. Using the `rand` library in C0, modify this code to select a random pivot.