Quickcheck 07: Solutions

Name:

Consider the following sorting algorithm in pseudocode:

```
1: function sort(A)
2:     for i = 1 to A.length − 1 do                ▷ indices are inclusive in this pseudocode.
3:         for j = 0 to i − 1 do
4:             if A[j] ≥ A[i] then
5:                 x = A[i]
6:                 Shift every element from j to i − 1 right by one
7:                 A[j] = x
8:                 break
```

(a) Which type of sorting does the above algorithm do (e.g., selection sort, insertion sort, merge sort, quick sort)? What is the worst-case input and worst-case big-$\mathcal{O}$ bound for this algorithm?

Solution:

Insertion sort. The worst-case runtime is $\mathcal{O}(n^2)$.

An array in reverse sorted order and an array in sorted order are both worst-case inputs for this algorithm.

When the array is in reverse sorted order, every element will need to be inserted at the beginning by shifting every element over to the right. This takes $\mathcal{O}(n)$ time and happens for every element, so the algorithm is $\mathcal{O}(n^2)$.

See part (c) for an explanation of why an array in sorted order is also $\mathcal{O}(n^2)$. The main difference is that a sorted array has $\mathcal{O}(n)$ comparisons and $\mathcal{O}(1)$ moves per iteration, while a reverse sorted array has $\mathcal{O}(1)$ comparisons and $\mathcal{O}(n)$ moves per iteration.

(b) Is this sorting algorithm stable? If yes, explain why. If not, what would you change to make it stable?

Solution:

The sorting algorithm is not stable. To make it stable, in the inner loop, the if condition (line 4) should be `if` $A[j] > A[i]$.

This is because the loop should insert $A[i]$ *after* all the elements that are equal to $A[i]$ (but before any elements that are greater) in order keep a stable order after sorting. The algorithm inserts $A[i]$ before the first element that is greater than *or equal* to $A[i]$, which would reverse the order of equal elements after sorting.

(c) Explain why this algorithm runs in $\mathcal{O}(n^2)$ time when given an already-sorted array.

Solution:

The inner for loop starts at the beginning of the array and looks for a place to insert $A[i]$ (line 3). Even though $A[i]$ is already in the correct place (at index i), the algorithm doesn't know this until it has compared it with every element before it (from index 0 to $i-1$). Both loops are $\mathcal{O}(n)$ so the algorithm is $\mathcal{O}(n^2)$ even for a sorted array.

(d) How would you change the algorithm so it runs in $\mathcal{O}(n)$ time instead when given an already-sorted array?

Solution:

Reverse the inner for loop so it runs from $j = i-1$ to 0 (line 3), and check if $A[j] \leq A[i]$ (line 4). Then, insert $A[i]$ *after* $A[j]$ instead of *before* by shifting every element from $j+1$ to $i-1$ right by one (line 6) and placing $A[i]$ at $A[j+1]$ (line 7).

(Note that this is also a stable sorting algorithm. It would be unstable if we used $A[j] < A[i]$.)

If the array is already sorted, $A[i-1] \leq A[i]$, so the loop will stop after the first iteration and run in $\mathcal{O}(1)$ time. The algorithm will then run in $\mathcal{O}(n)$ time. (This is why insertion sort is $\mathcal{O}(n)$ in the best case! Its inner for loop runs in reverse.)