Practice Problem Set 5

.

Problem 1

Execute Quicksort$(A, 1, 8)$ on the input array $[4, 2, 5, 8, 1, 7, 3, 6]$ by always selecting the last element in the array as the random pivot. Illustrate the steps of the recursion. Clearly indicate the parameters of all recursive calls to Quicksort.

Problem 2

Answer the following:

- What is the primary difference between the **return value** in the pseudo-code for the basic partitioning algorithm from week 3 versus the partitioning algorithm used by Quicksort?
- Suppose you accidentally implemented Quicksort so that it used the trivial Partitioning algorithm (the one that does not run in place). Would the runtime of Quicksort change or be the same? Justify your answer.
- Suppose you implemented Randomized-Select so that it used the in-place partitioning algorithm. Would the runtime of Randomized-Select change? Justify your answer.
- Compare the pseudo-code for Randomized-Select and Quicksort. Both algorithms make a call to a Partition algorithm that runs in time $\Theta(n)$, and both algorithms make recursive calls. Describe why the recurrences for the runtime of each algorithm are different from each other, and how they result in different expected runtimes.

Problem 3:

Below is an attempt at a new version of the QuickSort algorithm.

Step 0:. If $s \geq f$ return. Otherwise proceed with the following steps:
Step 1:. Use the **Select** algorithm to find the element of rank $\lfloor n/3 \rfloor$ in array A. Call this element m
Step 2:. Loop through A to find the **index** of element m. Call this index r
Step 3: Partition the array A using element at index r as the pivot
Step 4. Quicksort(A, s, r-1)
Step 5: Quicksort(A,r+1,f)

Does this algorithm correctly sort the input in array A?
What is runtime recurrence for this algorithm?
Determine the runtime of this algorithm in big-oh notation.

Problem 4:

Suppose we pick *three* random elements from the array, and then use the median of those three numbers as the pivot. If the array has size ≤ 2 then we pick either element

as the pivot. Execute this variation of quicksort on the array $[4, 2, 5, 6, 1, 7, 3, 8]$ where the three random numbers are always chosen as the first 3 elements in the array.

What is the best-case runtime of this new Quicksort?

What is the worst-case runtime of this new Quicksort?

Do you think that it is *less likely* that the worst-case scenario occurs?

Problem 5:

A new Quicksort algorithm is designed as follows: suppose we pick *two* pivots from A instead of just one pivot. We order the pivots as $x_1 < x_2$. We partition the elements in the input array into those that are less than x_1, those that are between x_1 and x_2, and those that are larger than x_2. Quicksort is called recursively on each of the three subarrays. For this new version of Quicksort, write a recursion for the best-case runtime and the worst-case runtime and justify the runtime.

Problem 6

Suppose we update how Quicksort finds a pivot. We now use the **Select** algorithm to find the *median* element of the array$A$, and this median is used as the pivot. What is the new overall runtime of Quicksort? What is the drawback of this new technique?

Problem 7

Discuss the differences in the following algorithms when the input is either random, sorted, or sorted in reverse: Quicksort, Insertion Sort, Selection Sort, Bubble Sort, Merge-sort. Which algorithm performs the best when the input is random?

Problem 8

For each of the following sorting algorithms, explain which run *in-place* and which do not: Quicksort, Insertion Sort, Selection Sort, Bubble Sort, Merge Sort.

Problem 9:

Below is the pseudocode for a recursive algorithm called PartialSort(A, s, f, k), which takes as input an array A indexed between s and f and a rank k. The algorithm calls the in-place Partition(A, s, f) algorithm from class, which returns the index of the random pivot. It also calls the usual Quicksort(A, s, f) algorithm.

Execute the code on the array $[4, 13, 2, 14, 10, 5, 8, 9, 1, 7, 3, 11, 6, 12, 15]$ using $k = 5$ and random pivots $12, 3, 9, 7, 5$.

By examining the pseudocode, describe its output. Finally, write a new version of the algorithm that produces the same output, but is not directly recursive. You may use the **Select** algorithm and the **Quicksort** algorithm. Your algorithm must run in time $O(n + k \log k)$ where k is the rank of the parameter.

```
PartialSort(A,s,f,k)
    if s < f
       p = Partition(A,s,f)
       r = p - s+1
```

```
if k > r
   PartialSort(A,p+1,f,k-r)
   Quicksort(A,s, p-1)
else if k < r
   PartialSort(A,s,p-1,k)
else if k=r
   Quicksort(A,s,p-1)
```

Hands-on Application:

Try running a simple implementation of Quicksort. You can find many implementations online, in various programming languages. There are also many online Python interpreters that you can use. Make sure that you use a version of Quicksort that uses an **in-place** partitioning algorithm.

Now do the following simple update to Quicksort: Add a new base-case that tests if the subarray to be sorted has less than 5 elements in it. If it does, use Insertion Sort instead to sort the subarray. Test the new version on large arrays. Does it seem faster than the original?