Name: $\boxed{\text{Michael Rogers}}$

ID: $\boxed{105667404}$

CSCI 3104 **Profs. Grochow & Layer**

Problem Set 4 **Spring 2019, CU-Boulder**

1. Suppose that instead of a *randomized* `QuickSort` we implement an *indecisive* `QuickSort`, where the `Partition` function alternates between the best and the worst cases. You may assume that `IndecisivePartition` takes $O(n)$ time on a list of length n.

   (a) (5 pts) Prove the correctness of this version of `QuickSort`.

(b) (5 pts) Give the recurrence relation for this version of `QuickSort` and solve for its asymptotic solution. Also, give some intuition (in English) about how the indecisive `Partition` algorithm changes the running time of `QuickSort`.

2. Consider the following algorithm that operates on a list of n integers:

   - Divide the n values into $\frac{n}{2}$ pairss.
   - Find the max of each pair.
   - Repeat until you have the max value of the list

   (a) (2 pts) Show the steps of the above algorithm for the list $(25, 19, 9, 8, 2, 26, 21, 26, 31, 26, 3, 14)$.

(b) (3 pts) Derive and prove a tight bound on the asymptotic runtime of this algorithm

(c) (3 pts) Assuming you just ran the above algorithm, show that you can use the result and all intermediate steps to find the 2nd largest number in at most $\log_2 n$ additional steps.

(d) (2 pts) Show the steps for the algorithm in part c for the input in part a.

3. Consider the following algorithm

```
SomeSort(A, k):
    N = length(A)
    for i in [0,..,n-k]
        MergeSort(A,i,i+k-1)
```

(a) What assumption(s) must be true about the array `A` such that `SomeSort` can correctly sort `A` given `k`.

(b) Prove that your assumption(s) is/are necessary: that is, for **any** array `A` which violates your assumption(s), `SomeSort` incorrectly sorts `A`.

(c) Prove that your assumption(s) from part a are sufficient. That is, prove the correctness of `SomeSort` under your assumption(s) from part a.

(d) Assuming that the assumption(s) from part a hold on `A`, prove a tight bound in terms of `n` and `k` on the worst-case runtime of `SomeSort`.

4. A dynamic array is a data structure that can support an arbitrary number of append (add to the end) operations by allocating additional memory when the array becomes full. The standard process is to double (adds n more space) the size of the array each time it becomes full. You cannot assume that this additional space is available in the same block of memory as the original array, so the dynamic array must be copied into a new array of larger size. Here we consider what happens when we modify this process. The operations that the dynamic array supports are

   - `Indexing A[i]`: returns the `i`-th element in the array
   - `Append(A,x)`: appends `x` to the end of the array. If the array had `n` elements in it (and we are using 0-based indexing), then after `Append(A, x)`, we have that `A[n]` is `x`.

   (a) Derive the amortized runtime of Append for a dynamic array that adds $n/2$ more space when it becomes full.

(b) Derive the amortized runtime of Append for a dynamic array that adds n^2 more space when it becomes full.

(c) Derive the amortized runtime of Append for a dynamic array that adds some constant C amount of space when it becomes full.