Name: Keaton Whitehead
ID: 104668391

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 `indecisive QuickSort` takes $O(n)$ time on a list of length n.

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

   Proof by Induction:

   - Base Case: n = 1 which if there is only one element in the array it is sorted
   - We assume that the formula is true for all f(n), because it is a recursive function. After this, the two arrays will be sorted, assuming the `Partition` method works correctly. We can also assume that the pivot will be selected as either the best or the worst position in an alternating manner. Now we will look through the intermediate steps of the `Partition` function where the array is split up into 4 regions, with x being the pivot.
     - Region 1: values that are $\leq$ x (between location of p and i)
     - Region 2: values that are $>$ x (between location of i+1 and j-1)
     - Region 3: unprocessed values (between locations j and r-1)
     - Region 4: values that are $\leq$ x (location r)
   - Given these 4 regions we will us a loop invariant to cycle through the for loop for any location k. Here are the loop invariants:
     - If p $\leq$ k $\leq$ i then A[k] $\leq$ x
     - If i+1 $\leq$ k $\leq$ j-1 the A[k] $>$ x
     - If k=r the A[k]=x
       * `Initialization` - This invariant holds true before the loop begins
       * `Maintenance` - The invariant holds at the (i - 1)th iteration, which will also hold true for the ith iteration
       * `Termination` - This proves correctness of Partition
   - Partition is the same for both the worst and best cases, with the exception that the selection of the pivot is different. Assuming that each iteration will alternate between switching from the worst case to the best case we can say that indecisive QuickSort is correct.

(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`.

To understand the recurrence relation for this version of QuickSort we need to first understand the recurrence relations that occurs for both the worst and best cases. To begin we will look at the best case first. For the best case we get a recurrence relation of:

$$T(n) = 2T(n/2) + \theta(n)$$

This is written in a format that can be solved using the `Master Theorem`. We can let $a = 2$ and $b = 2$ in the equation of:

$$T(n) = aT(n/b) + f(n)$$

Given the rules of the Master Theorem, we get the second case which tells us that the runtime is $\theta(nlogn)$.//
For the worst case we get the recurrence relation of:

$$T(n) = T(n-1) + T(1) + \theta(n)$$

$$T(n) = T(n-1) + \theta(n)$$

We can solve this using `unrolling`.

$$T(n) = T(n-1) + \theta(n)$$

$$T(n) = T(n-2) + T(n-1) + \theta(n)$$

$$T(n) = T(n-3) + T(n-2) + T(n-1) + \theta(n)$$

$$...$$

$$T(n) = \theta(1) + ... + \theta(n-2) + \theta(n-1) + \theta(n)$$

Thus the worst-case running time of QuickSort is $\theta(n^2)$

Now since we know that partition switches from the worst case to the best case we know that the runtime of the program will alternate between these two runtimes. Now I don't know how to prove this any other way than just through words, but if we look at both of the recurrences we can notice how the partition is splitting up the array. So for the best case it is being split in half (i.e. T(n) = 2T(`n/2`) +

$\theta(n)$) and the worst case is only partitioning the array by one element (i.e. T(n) = T(n-1) + $\theta(n)$). Clearly the best case is going to split off more elements than the worst case will ever be able to. So even though it does alternate between the two cases, the function will be partitioned mainly in the best case. So as we go into to having bigger data sets the running time will be very similar to `nlogn` because the the n-1 partitions will not affect the partitioning as much.

Name: Keaton Whitehead
ID: 104668391

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 4 **Spring 2019, CU-Boulder**

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

   - Divide the n values into $\frac{n}{2}$ pairs.
   - 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 $(25, 19, 9, 8, 2, 26, 21, 26, 31, 26, 3, 14)$.
   (25|19)————(9|8)————-(2|26)————-(21|26)————-(31|26)————-(3|14)
   ————(25|9)————————————(26|26)————————————(31|14)
   ————————————(25|26) ————————————(31*Skip this comparison)
   ————————————————————(26|31)————————————————-
   ————————————————————(31)————————————————————

   Here you can see that each iteration where there are only n/2 comparisons. Each (A|B) denotes a comparison of A and B. The result of which element was greater is then passed down to the next iteration. Note that on the third iteration there is nothing to compare 31, so sit gets skipped and it is just passed on to the next iteration. This is also because we are only allowed to do n/2 comparisons on each level so on the third iteration we can only do 3/2 comparisons which is 1.5 so it just moves on to the next iteration.

(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 operations.

(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`.

- For this function to work we must assume that k >0 AND k != length(A).
- We can assume that A[1,...,n] is a list of numbers, $1 \leq k \leq n$ where both k and n are natural numbers.
- This immediately implies the precondition of SomeSort(A[1,...,n]) because A[1,...,n] is an array of numbers with n as a natural number

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

- If k $\leq$ 0, the for loop will loop from 0 to n. As we call Merge Sort from the first iteration of the for loop we have i= 0 and i + k -1 = 0 + 0 -1. So this sets the right index of the sort to a negative value which does not exist, therefore, causing an error.
- If k $\geq$ n then the for loop will not loop at all because the for loop runs from 0 to n - k. If k = length(A) then the for loop will run from 0 to 0 resulting in no code being executed inside of the for loop.

(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.

- First we assume that k $>$ 0 AND k != length(A) is true.

(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.