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:

   - `Theorem:` The `randomized QuickSort` will sort a given array.
   - `Assumptions:`
     - Let p = first value of A
     - Let q = position of the pivot
     - Let r = last value of A
     - Let n >1
     - `Partition` will alternate between the best and worse cases.
     - The first iteration of the function execute the `best case Partition` functionality.
     - That both best and worse case Partitioning functions will always provide a correctly sorted list.
   - `Base Case:` n = 1. If there is only one element in the array it is certainly sorted. Thus it will return a sorted list.
   - `Inductive Hypothesis:` The `random QuickSort` works currently for all iterations on A with any lists of length n.
   - `Inductive Step:` Let A be a list of length n. Suppose the Inductive Hypothesis (`IH`) is true. We can break this up into multiple cases:
     - `Case 1: The list is not sorted`
       Given the assumption that the QuickSort will provide a correctly sorted list and that it is a recursive function that calls on two QuickSorts on a subarray on both sides of a given pivot. We know that The QuickSorts will return a correctly sorted list where the first call will return a sorted subarray A[p..q-1], another subarray from the second QuickSort will return a sorted subarray A[q+1..r], and we will have a pivot located at A[q] This will ultimately return a sorted list because A[p..q-1] $\leq A[q] \leq A[q+1..r]$.

So if the current iteration has a list that is not sorted, then the program has not yet finished sorting all of the possible iterations of partitions and QuickSorts that were called.

- `Case 2: The list is sorted`
  If the list is sorted that means that for ever partition called, there was two calls to QuickSort that were on subarrays on either side of the pivot point. From our assumption of the alternating partitions from best to worse case we can safely assume that this will occur every iteration. Once all of the recusive iterations are ran, the all the QuickSort functions will have reached a base case and returned a correctly sorted subarray. There fore constructing the array with may subarrays that were properly sorted from the function calls will result in a sorted array and finish the program.

- Thus by Induction, we have shown that randomized QuickSort is correct and will return a completely sorted array.

(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 spliting the array 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 efficiency of the best case partitioning. It is very likely that the actual run time will be greater than `nlogn` because the worst case partition will slow it down. Therefore we can say that this algorithm will have a running time of $\Omega$(n).

Name: Keaton Whitehead
ID: 104668391



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 iteration 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 after doing one comparison. Resulting in `31` as the max value in the array.

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

- The Asymptotic runtime of this algorithm can be written as so:

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

- We can use this equation because the array gets split by n/2 which is why we have $T(n)$. The $n/2$ is the amount of work that has to be done, which in this case is the amount of comparisons that occur on each iteration. We can use the `Master Theorm` to calculate the asymptotic runtime. Here we have a = 1, b = 2 and k = 1. We get the case where $a < b^k$ because indeed $1 < 2^1$ which tells us we have a runtime of $O(n)$.

(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.

- Assuming that we just ran the above algorithm, we can use the greatest value to quickly determine the 2nd largest number in the array in at most $log_2 n$ additional operations. In the case above, we know that 31 is the greatest number and that the second biggest number would be compared to 31 because the function is taking the max of every comparison and comparing them. Therefore a 2nd max would eventually be compared to 31. Knowing this, we can use it to our advantage. Instead of going through the entire array, we just have to compare all of the values that were compared to 31 with a temporary value. Here we set the temp to the max from the comparison of the temp and every value that is compared to 31. Because we can ignore all other cases we are only looking at the path that 31 took with its comparisons. Since 31 is the max and always moves on to the next iteration the depth of the path 31 follows is the depth of the function. Since this function is always splitting the array in half it is an upside down fully balanced tree partially, there are some cases where one node only has one children while the other side is full, but this happens not too frequently so it is safe to assume that it acts like a full balanced tree. Knowing this we can determine that to find the 2nd max value in this array will only be $log_2 n$ because a full balanced tree has a depth of $log_2 n$. What we said before 31 traverses the depth of the entire tree so the worst case for finding the 2nd max will be an additional $log_2 n$ operations.

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

- *I changed the 10th array value to 27 to demonstrate the worst case*

  (25|19)————(9|8)————-(2|26)————-(21|26)————-(31|27)————-(3|14)
  ————(25|9)————————(26|26)————————(31|14)
  ————————(25|26) ————————(31*Skip this comparison)
  ————————————(26|31)————————————-
  ————————————(31)————————————

- We have found the max which is 31. Now we wish to find the 2nd max
  ————————————-(26|31)————————————

- Set temp value to 26.
  ————————(31 *skipped this comparison)————————
  –

- Do nothing because there was no comparison
  ————————————-(31|14)————————————

- $14 < 26$ so do nothing
  ————————————-(31|27)————————————

- Set temp value to 27 because $26 < 27$

- We have reached the beginning of the function and there is no more comparisons left to be made. Return temp which is = 27. Looking back we can see that we only did 3 comparisons to temp and we had a total n of 12 so if we use our equation for additional operations we get $log_2(12) = 3.58496$. So we did 3 operations and based on our calculations we were had 3.59 operations so this works.

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 correctly sort the array we have to assume that A has elements that are at most k positions away from their correct index.
- We can also assume that MergeSort works correctly.
- $Len(A) > k$ so $n > k$
- $k > 1$

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

`Proof:`

`Loop invariant:` An array B has an element `j` at position `l` that belongs at position `p` is more than `k` positions away from the correct position `p` in the subarray of that instance of MergeSort. This will result in sorting an array but it will not be in correct order.

`Initialization:`
Before the body of the first iteration of the loop executes:
- i = 0.
- n = length(B)
- B is still not sorted because we have not entered the for loop and executed the MergeSort function to begin the sorting.

`Maintenance:` Let $i > 0$. Suppose the loop invariant holds true just before the body of the (i-1)st iteration of the loop executes. MergeSort is called and a subarray is created containing elements in B[i..k-1]. This will be sorted in the given scope of B[i] to B[(i+k-1)]. Item `l` will then be sorted according to the scope of `B[i]` to `B[(i+k-1)]`. With each iteration of the for loop `i` will increase by one. This will in turn move the scope of the MergeSort by one as both outer regions of the scope are dependent on `i`. It is likely that an element `j` has a correct position `p` that has been already passed by `i`, and since the element `j` has a distance from it's point `p` that is $\geq k - 1$, that does not lie in the scope of the MergeSort, because again its' scope is from $B[i..k-1]$, so the element `j` will never be able to be moved to the correct location P, because index `i` has already passed it.

`Termination:` Since there exists an element `j` that has positions `p` and has been sorted by MergeSort but is not at index`p` when the loop terminates our loop invariant remains true.

(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.

`Proof:`

`Loop invariant:`
An array B has an element `j` at position `l` that belongs at position `p` is at most `k` positions away from the correct position `p` in the subarray of that instance of MergeSort. This will result in sorting an array but it will not be in correct order.

`Initialization:`
Before the body of the first iteration of the loop executes:
- i = 0.
- n = length(B)
- B is still not sorted because we have not entered the for loop and executed the MergeSort function to begin the sorting.

`Maintenance:` Let $i > 0$. Suppose the loop invariant holds true just before the body of the (i-1)st iteration of the loop executes. MergeSort is called and a subarray is created containing elements in B[i..k-1]. This will be sorted in the given scope of B[i] to B[(i+k-1)]. Item `l` will then be sorted according to the scope of `B[i]` to `B[(i+k-1)]`. With each iteration of the for loop `i` will increase by one. This will in turn move the scope of the MergeSort by one as both outer regions of the scope are dependent on `i`. Element `j` has a correct position `p` that has not been passed by `i` yet because `j` has a position that is at most $\texttt{k} - 1$ indexes away from `p`. So the element `j` will be able to move to the correct location `p`, because index `i` has not already passed index `p`.

`Termination:` Since there exists an element `j` that has positions `p` and has been sorted by MergeSort and is at index`p` when the loop terminates our loop invariant remains 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`.

We know that the worst case for MergeSort is $O(nlogn)$. MergSort is working based off of k and is being called n-k+1 times.
$T(n) = (n - k + 1)O(klogk)$
$\approx$ (n-k+1)$O(klogk) = nklogk - k^2logk + klogk > nklogk$
SomeSort is $O(nklogk)$

Name: Keaton Whitehead
ID: 104668391



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 run-time of Append for a dynamic array that adds $n/2$ more space when it becomes full.

   According to the equation of Accounting Method we have:
   $Work = spaceadded + itemsinserted - itemswithdrawn$
   and
   $Amortizedcosts = WorkDone/ofAppends$
   With this we get:
   $Work = (n/2) + (3n/2) - (3n/2) = n/2$
   $Amortizedcosts = (n/2)/(n/2) = 1$
   Therefore, we get $O(1)$

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

$Work = (n^2) + (n + n^2) - (n + n^2) = n/2$
$Amortizedcosts = (n^2)/(n^2) = 1$
Therefore, we get $O(1)$

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

$Work = (C * \Theta(1)) + (\Theta(n + C) - (\Theta(n + C))$
$Work = (C * \Theta(1)) + 2(\Theta(n + C))$
$C \sum_{i=1}^{n} i + 2(\Theta(n + C))$
$\sum_{i=1}^{n} = C((n(n + 1))/2) = n^2$
$Amortizedcosts = (n^2)/(n) = n$
Therefore, we get $O(n)$