Name: Trevor Buck

ID: 109081318

CSCI 3104 **Profs. Grochow & Layer**

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

Quick links: 1a 1b 2a 2b 2c 2d 3a 3b 3c 3d 4a 4b 4c

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` works correctly (that is, it produces a list in which the first i elements are all $\leq x$, the $(i+1)$-st element is x, and the remaining elements are all $\geq x$, where x is the pivot and i is what it has to be) and takes $O(n)$ time on a list of length n.

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

```
def ind-Quicksort(A):                    ('→' means through)
    ind-partition(A[0→i], x, A[i+1→(length-1)])
    ind-Quicksort(A[0→i])
    ind-Quicksort(A[i+1→(length-1)])
```

Proof by induction

(IH) ind-Quicksort() sorts a list of size n

Base case $n = 1 \Rightarrow$ List is already sorted. so we know it works

Case n+1 There are 2 options:

Option 1: $A[n+1] \leq x$	Option 2: $A[n+1] > x$
$A[n+1]$ gets sorted into first list. Then that list gets recursively called. As long as 'partition' works, the array will be sorted ✓	$A[n+1]$ gets sorted into the second list and is then recursively called. " " " " ✓

Name: Trevor Buck
ID: 109081318

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

(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`.

Best case: Partition splits list into 2 equal arrays

Worst case: Partition splits the list of size 'n' into lists of size 'n-1' and '1'

$T(\text{ind. quicksort}) = \overbrace{O(n) + 2T(\tfrac{n}{2})}^{\text{best case}}$

$= O(n) + 2\left(O(\tfrac{n}{2}) + \overbrace{T(\tfrac{n-1}{2}) + 1}^{\text{worst case}}\right)$ → constant

$= O(n) + 2\left[O(\tfrac{n}{2}) + O(\tfrac{n-1}{2}) + T(\tfrac{n-1}{4})\right]$

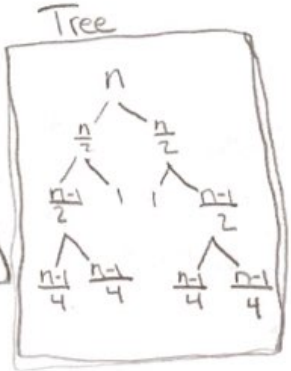
2 observations

- Work at each level = $O(n)$
- Twice as many layers as best case scenario because our worst case doesn't really help

# of Levels = $2\lg(n)$

- Total Time = $O(2n \lg n)$
- Our running time exactly doubles

Tree



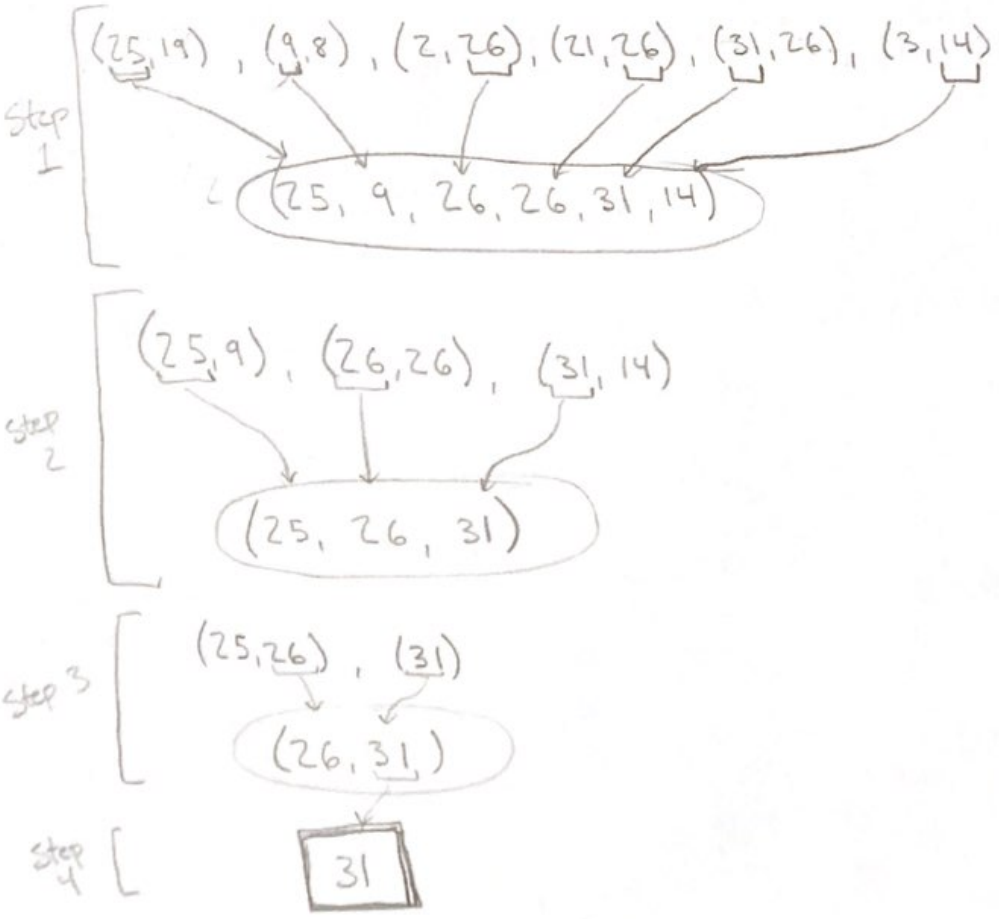
Name: Trevor Buck
ID: 109081318

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 $\frac{n}{2}$ consecutive pairs, starting from the beginning.
   - 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).



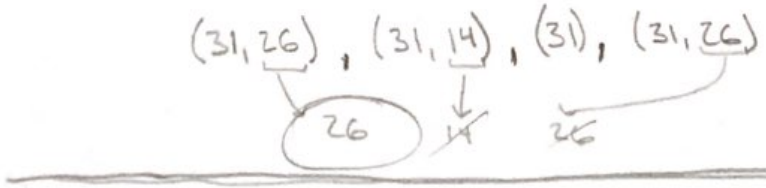
(b) (3 pts) Derive and prove a tight bound on the asymptotic runtime of this algorithm

- steps taken = $\lg(n)$
- amount of work @ each step:
  - Dividing n takes $\frac{n}{2}$ steps
  - Finding max of each pair is $\frac{n}{2}$
  - $\frac{n}{2} + \frac{n}{2} = n$
- Runtime = $n \lg n$

77

(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.

The second largest number had to be paired with the largest number at some point in the algorithm. We also know that the largest number was 'paired' with '$\log_2 n$' numbers. Which means that all we have to do is look at those numbers to see which is biggest. In our example:



[STORE IT AS a value]

SEE EXTRA PAGE FOR PSUEDO CODE

```
def secondlargest(A, max):
    secondmax = ∅
    divide into pairs(A)
    for all pairs:
        if (a == max):
            if (b > secondmax):
                secondmax = b
        if (b == max):
            if (a > secondmax):
                secondmax = a
    return secondmax
```

assume pairs are listed as (a, b)

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

First pairing: (31,26)

*Looking for pairings with '31'

- Set '26' as second largest

Second pairing (31,14)

- compare '14' and '26'
- keep 26

Third pairing: (31)

- Keep 26

Fourth pairing: (31,26)

- compare '26' and '26'
- Keep '26'

Name: Trevor Buck
ID: 109081318

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

3. Consider the following algorithm

```
SomeSort(A, k):                    n=10  k=5
    N = length(A)
    for i in [0,..,n-k]
        MergeSort(A,i,i+k-1)
```

(a) (5 pts) What assumption(s) must be true about the array A such that SomeSort can correctly sort A given k.

- Assume that MergeSort works correctly
- Assume $K \leq N$
- Assume that at every step, the minimum value of $A[i, i+1, \ldots, n-1]$ has to be located in the first 'K' positions
- Assume that minimum value fom the step above is also greater than any value from $A[0, 1, \ldots, i-1]$.

↖ This holds true if the assumption listed above is true

(b) (6 pts) Prove that your assumption(s) is/are necessary: that is, for **any** array `A` which violates your assumption(s), `SomeSort` incorrectly sorts `A`.

if $k > n$: Let $k = 11$, and $n = 10$

SomeSort(A, 11):

N = length(A) ← ie. 10

for i in [0, ..., 11] ← IMPOSSIBLE

- Let A = [2, 3, 4, 1, 5, 6], and k = 3

first sort result: [2, 3, 4, 1, 5, 6]

second sort result: [2, 1, 3, 4, 5, 6]

↳ Wrong positions

~~- let A = [1, 2, 3, 5, 6, 7, 8, 4] and k = 3~~

~~First sort = A = [1, 2, 3, 5, 6, 4, 7, 8]~~

(c) (8 pts) Prove that your assumption(s) from part a are sufficient. That is, prove the correctness of `SomeSort` under your assumption(s) from part a.

For Loop: Loop Invariant: $A[0, 1, \ldots, i+k-1]$ is sorted

Initiation: $i = 0$

*Assume Merg sorts $A[i, i+1, \ldots, i+k-1]$

Maintain:

From part A, if we know the minimum value from $A[i, i+1, \ldots, n-1]$ is found in here: $A[i, \ldots, i+k-1]$

Min will get sorted to the ith position.

* it is important to note that we also know that $A[i]$ is greater than any $A[0, 1, \ldots, i-1]$.

Termination $i = n-k \Rightarrow i+k-1 = (n-k)+k-1 = n-1$

we sort the last element in the list

(d) (5 pts) 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`.

MergeSort $= n \log_2 n$

# of loops $= n-k+1$

work at each loop $= k \log_2 k$

$\Rightarrow$ runtime $= O((n-k+1)(k \log k))$

$= O((n \cdot k \cdot \log_2 k) - (k^2 \log_2 k) + (k \log k))$

$\boxed{= O(n \cdot k \cdot \log_2 k)}$

Name: Trevor Buck
ID: 109081318

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

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

Amortized runtime =

work + deposits − withdraw, $\frac{n}{2}$

work = cost of insert + cost of copy

$= \Theta(1) + \Theta(n) = \boxed{\Theta(n)}$

$= \Theta(n) + \Theta\left(\frac{n}{2}\right) - \Theta\left(\frac{n}{2}\right)$

$= \boxed{\Theta(n)}$

(b) (6 pts) Derive the amortized runtime of Append for a dynamic array that adds n^2 more space when it becomes full.

work = cost of insert + cost of copy

$= \Theta(1) + \Theta(n) = \boxed{\Theta(n)}$

Deposits = $\Theta(n^2 - n)$ ← Extra space after squaring 'n'

Withdraw = $\Theta(n^2 - n)$ ← Fill that space

total runtime = work + deposits − withdraw

$= \Theta(n) + \Theta(n^2 - n) - \Theta(n^2 - n)$

$= \boxed{\Theta(n)}$

(c) (5 pts) Derive the amortized runtime of Append for a dynamic array that adds some constant C amount of space when it becomes full.

Work = cost of insert + cost of copy

$= \Theta(1) + \Theta(n) = \Theta(n)$

Deposit = $\Theta(c)$ ← Extra space

Withdraw = $\Theta(c)$ ← Fill space

total runtime = work + Deposit - Withdraw

$= \Theta(n) + \Theta(c) - \Theta(c)$

$= \Theta(n)$