CS210 => Assignment 3

Due: Mar. 9 2022

1. a) Algorithm: LucasNumber(n). -> Recursive
Input: Positive integer 'n'. $[0, \infty) \in \mathbb{Z}$
Output: The n^{th} Lucas Number Value.

$0 \Rightarrow 2$
$1 \Rightarrow 1$
$5 \Rightarrow$

```
if n = 0 return 2
else if n = 1 return 1
else return LucasNumber(n-1) + LucasNumber(n-2)
```

b) Algorithm: LucasNumber(n)
Input: Positive integer 'n'. $[0, \infty) \in \mathbb{Z}$
Output: The n^{th} Lucas Number value.

```
if n = 0 return 2
else if n = 1 return 1
else{
    Declare array A with size of n+1
    A[0] = 2
    A[1] = 1
    for i <- 2 to n {
        A[i] = A[i-1] + A[i-2]}
    return A[n]}
```

2. $T(n) = T(n-1) + 6n^2 + 1$ for $n > 0$; $T(0) = 1$

The numerical solution points us in the direction of $O(n^3)$ but we cannot say for sure yet.

$n = 5$
$T(5) = 185 + 150 + 1 = 336$
$T(4) = 88 + 96 + 1 = 185$
$T(3) = 33 + 54 + 1 = 88$
$T(2) = 8 + 24 + 1 = 33$
$T(1) = 1 + 6 + 1 = 8$
$T(0) = 1$

$T(n) = T(n-1) + 6n^2 + 1$
$T(n-1) = T(n-2) + 6n^2 + 1$
$T(n-2) = T(n-3) + 6n^2 + 1$
$T(n-3) = T(n-4) + 6n^2 + 1$
$T(n-4) = T(n-5) + 6n^2 + 1$
$T(n-5) = T(n-6) + 6n^2 + 1$

* At first glance this looks lik $O(n^3)$.

(2)

$$T(n) = T(n-1) + 6n^2 + 1 \quad \}1$$

$$T(n) = T(n-2) + 6(n-1)^2 + 1 + 6(n)^2 + 1 \quad \}2$$

$$T(n) = \underbrace{T(n-3)}_{\text{for } 3=n} + 6(n-2)^2 + 1 + 6(n-1)^2 + 1 + 6(n)^2 + 1 \quad \}3$$

$$T(n) = \sum_{i=1}^{n} 6(i)^2 + \sum_{i=1}^{n+1} 1$$

$$T(n) = n(n+1)(2n+1) + (n+1) \Rightarrow 2n^3 + 3n^2 + n + n^{+1} \Rightarrow 2n^3 + 3n^2 + 2n + 1$$

At first I did a few manual calculations on page 1, to Check if if my iterative/repeated Solution is correct. I used the case of $T(3) = 88$. First I began with writing out $T(n)$, $T(n-1)$, $T(n-2)$ and so on. Then I substituded a following $T(n-1)$ call into its calling $T(n)$ function for a few iterations. I noticed the $6(n-\#)^2$ term was showing up an amount of times equal to lenght of recursive call, for example $n=3$ it Showed up 3 times with the values of 3, 2, and 1 therefore I simplified it as $\sum_{i=1}^{n} 6(i)^2$. Next, I noticed the constant 1 appeared on equal amount of times as recursive calls (ex, $n=3$ then 3) plus a final recursive call $T(0) = 1$ adding another constant of 1. Therefore I simplified it as $\sum_{i=1}^{n+1} 1$. Putting these two summation together and simplifying yields

$$2n^3 + 3n^2 + 2n + 1$$

My claim is big-oh is $O(n^3)$ to prove this claim we determine constants C, $C \in \mathbb{R}$ and $n_0 \in \mathbb{Z}^+$ so that for all $n \geq n_0$ $2n^3 + 3n^2 + 2n + 1 \leq C \cdot n^3$.

I pick $C = 8$ and $n_0 = 1$: $2n^3 + 3n^2 + 2n + 1 \leq 8n^3$

$$\Rightarrow 2n^3 + 3n^2 + 2n + 1 \leq 2n^3 + 3n^3 + 2n^3 + n^3$$

$$\begin{aligned} 2n^3 &\leq 2n^3 \\ 3n^2 &\leq 3n^3 \\ 2n &\leq 2n^3 \\ 1 &\leq n^3 \end{aligned}$$

$\leftarrow$ Due to all constants being positive, and all n being positive we can say a higher power grows faster or the same therefore all claims true.

In Conclusion by the definition of big-oh the solution is $O(n^3)$.

③ 3. a) Algorithm: enqueue(e)
Input: element 'e'
Output: (none, inserts e at the front of queue)

```
If S.isEmpty() is true
    S.push(e)
else
    while S.isEmpty() is false
        T.push(S.peek())          C2    C2 × # of iterations of J
        T.pop()
C1  T.push(e)
    while T.isEmpty() is false
        S.push(T.peek())          C3    C3 × # of iterations of J+1
        T.pop()
Count = Count + 1
```

b) Algorithm: dequeue()
Input: None
Output: If the queue is empty return 0 and inform user, could use an "exit" function. Otherwise return the element at the front of the queue, oldest element added.

```
if S.isEmpty() is true
    print "Empty"  } or exit(0)      C2
    return 0
else
    X = S.peek()
    S.pop()                          C1
    Count = Count - 1
    return X
```

My algorithms use the "S" stack as the main storing stack while the "T" stack is used to help with operations.

we know the stacks are implemented as a linked list therefore we can say that push(e), pop(), size(), peek() and isEmpty() run in constant big-Oh time O(c). (Therefore they will be treated as constant.)

④ worst case analysis for enqueue(e):

$(C_2 \times \#\text{ of iterations of } J) + (C_3 \times \#\text{ of iterations of } J+1) + C_1$

$(C_2 + C_3)(\#\text{ of iterations of } J) + C_1$

$(C_{2+3})(\#\text{ of iterations of } J) + C_1$

Due to while loops the j iterations are equal to queqe size of n.

* The largest power value.

$C(n) + C_1$ $\boxed{\therefore O(n)}$

looking at our psuedocode we see that lines 1, 2, 7, 11 are outside of any loops therefore constent. The lines 5, 6, 9, 10 are within a while emply loop which means that they will iterate a number of times equal to the size of the queue to move it beetween the 2 Stacks. Therefore the most "impoetful" term is the while loop based on n (size) ∴ that is big-Oh.

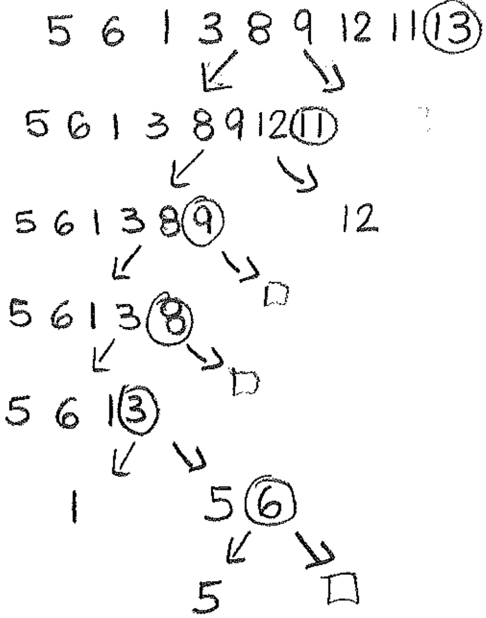
Worst case analysis for dequeue():

$C_1 + C_2$
C $\rightarrow$ $\boxed{\therefore O(C)}$

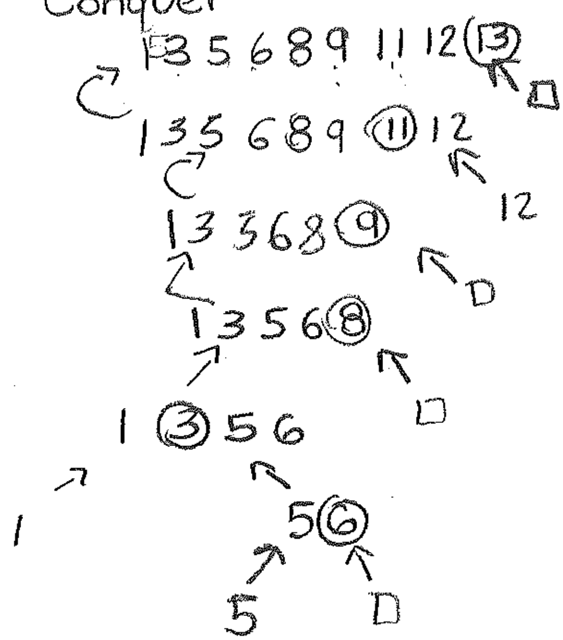
looking at this algorithm we see only 2 conditional statements which are constent on lines 1 and 4. The other lines of 2, 3, 5, 6, 7, 8 all are also constent time. Therefore due to no iteration loops we can say this algorithm is constent time.

5 a)

Divide

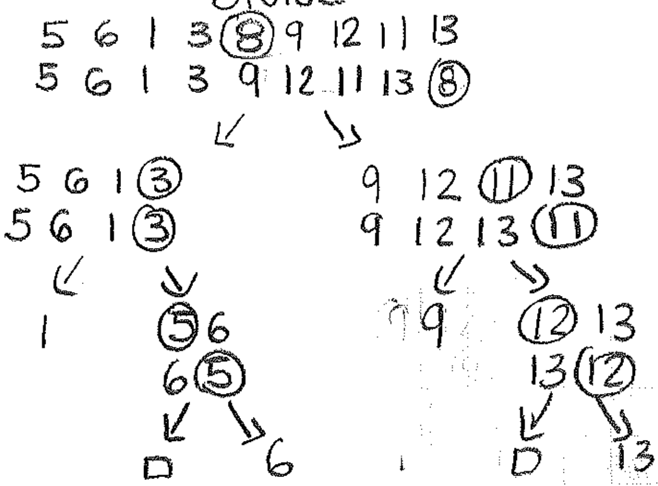


Conquer

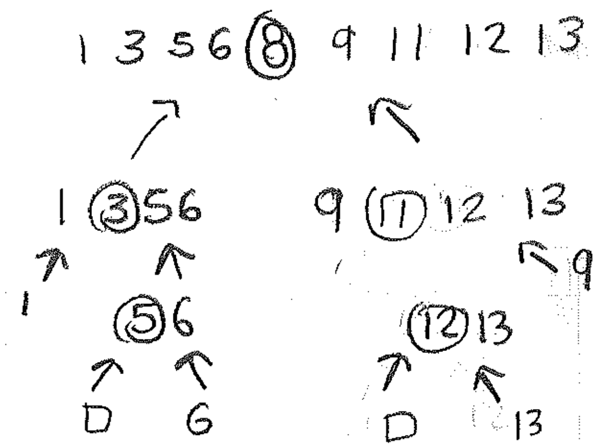


b)

Divide

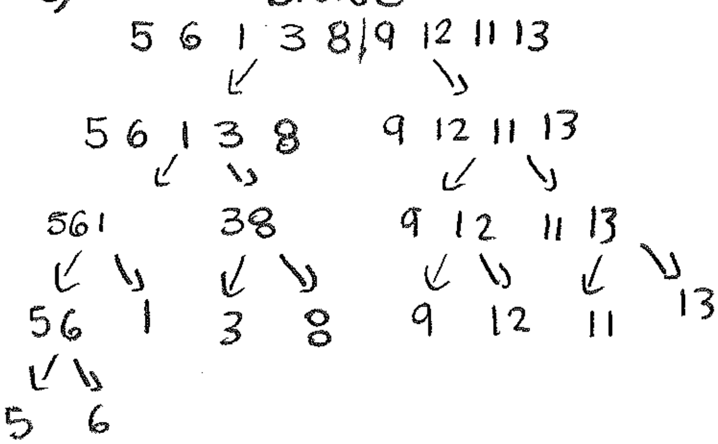


Conquer



c)

Divide



Conquer