NAME: VIVEK KOODLI UDUPA

CONCEPTUAL PROBLEM

(1) Consider a singly linked list of n nodes, 0,1,.. n-1. Write a recursive function to delete all the even-nodes.

```
class Node :
    def __init__(self, data = None, next = None) :
        Self.data = data
        Self.next = next
    def getNextNode (self) :
        return self.next
    def setNextNode (Self, newNext) :
        Self.next = newNext
    def getData (Self) :
        return self.data
Class LinkedList :
    def __init__ (Self) :
        Self.head = None
    def push (self, data) :
        newNode = Node (data)
        newNode.setNextNode (self.head)
        self.head = newNode
```

NAME: VIVEK KOODLI UDUPA

```
def printLis (self):
    current = Self.head
    while current :
        print (current.data, "  ", end = "  ")
        current = current.getNextNode ()
    print ()
```

#Recursive Function

```
def deleteEven (Self, prev = None, current = None):

    if (prev == None and current == None):
        prev = None
        Current = Self.head
    if (current == None):
        return
    elif (current.data % 2 == 0) :
        if (prev == None):
            Self.head = current.getNextNode ()
            prev = Self.head
        else :
            prev.setNextNode (current.getNextNode())
        if (current.getNextNode () == None):
            current = None
        else
            current = current.getNextNode ()
    else: prev = current
          current = current.getNextNode ()
    Self.deleteEven (prev, current)
```

Recursive Function Call

NAME: VIVEK KOODLI UDUPA

Algorithm to remove even Numbers from Linked List

Step 1 : Define two variables "previous" and "current" to Keep track of Nodes. Initialize previous to None and current to Head of the Linked List.

Step 2 : Check if we are at the end of the List.
IF NOT, continue

Step 3 : Check if the current Node is even by using the Modulus 2. Even numbers return Zero.

Step 4 : If the number is even, then we delete the Node by NOT POINTING AT IT.
Change the previous pointer to point at the Next item.

Step 5 : If the Number was odd in step 3, continue to the next node. without skipping over the node.

Step 6 : Recursively perform steps 2 through 5 until the end of Node is reached.

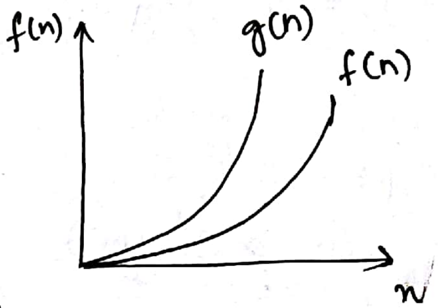
Step 7 : Once all the Nodes are checked and even nodes are deleted, print the final result.

Name: Vivek Koodli Udupa

② Let $f(n) = n^{3/2}\log(n)$ and $g(n) = 3n^2$. Prove that $f(n)$ is $O(g(n))$ but $f(n)$ is not $\Theta(g(n))$

Part I) $f(n)$ is $O(g(n))$ is true if $f(n) \le c\, g(n)$ for $c > 0$ and $n \ge n_0$.

Basically this means that $g(n)$ is the upper-bound of $f(n)$



given $f(n) = n^{3/2}\log(n)$ $\qquad g(n) = 3n^2$

$= n^{1.5}\log(n)$ $\qquad g(n) = 3.(n^{1.5})(n^{0.5})$

$$f(n) \le c.g(n)$$

$$\cancel{n^{1.5}}\log(n) \le 3c.(\cancel{n^{1.5}})(n^{0.5})$$

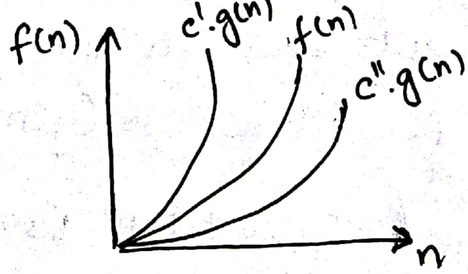
$$\log(n) \le 3c\,n^{0.5} \quad \text{— ①}$$

Equation ① is true for all $c > 0$ and $n \ge 1$ $\therefore$ $f(n)$ is $O(g(n))$

part II) $f(n)$ is not $\Theta(g(n))$

$f(n)$ is $\Theta(g(n))$ if $c'.g(n) \le f(n) \le c''.g(n)$

If we can show that $f(n) < c''.g(n)$ then $f(n)$ will not be $\Theta(g(n))$



Name: Vivek Koodli Udupa

Since $f(n)$ is bounded by $c'.g(n)$ and $c''.g(n)$, and we proved in Equation (1) that $c'.g(n) \geq f(n)$ we must show that ~~$f(n) \not<$~~ $f(n) \not> c''.g(n)$

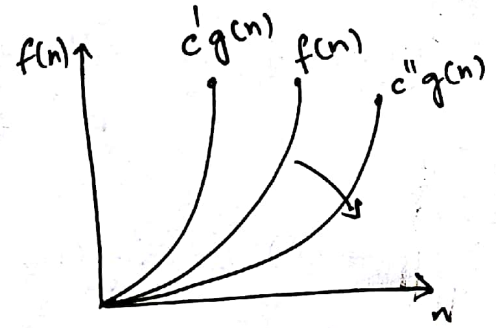
$f(n) = n^{1.5}\log(n)$ $\qquad\qquad$ $g(n) = 3n^2$

$$c''.g(n) = c''.3n^2$$

To prove:

$f(n) \not> c''.g(n)$ [$f(n)$ IS NOT GREATER THAN $c''.g(n)$]

$$\cancel{n^{1.5}}\log(n) \not> 3c''.(\cancel{n^{1.5}})(n^{0.5})$$



$$\log_2(n) \not> 3c''.\sqrt{n}$$

$$\log_2(n) < 3c''.\sqrt{n} \quad — (2)$$

If we substitute $n=4$ and $c''=1$ in (2)

$$\log_2(4) < 3.(1).\sqrt{4}$$

$$\underline{\underline{2 < 6}}$$

we can see that $f(4)=2$ and $c''g(4)=6$. ~~Thus~~ ~~w)~~ $f(n)$ went below its lower bound $c''.g(n)$. Thus we can say $f(n)$ is not $\Theta(g(n))$

Thus we proved that $f(n)$ is $O(g(n))$ and $f(n)$ IS NOT $\Theta(g(n))$

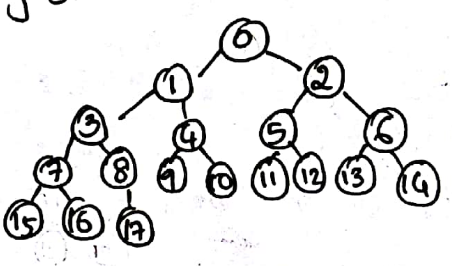
NAME: VIVEK KOODLI UDUPA

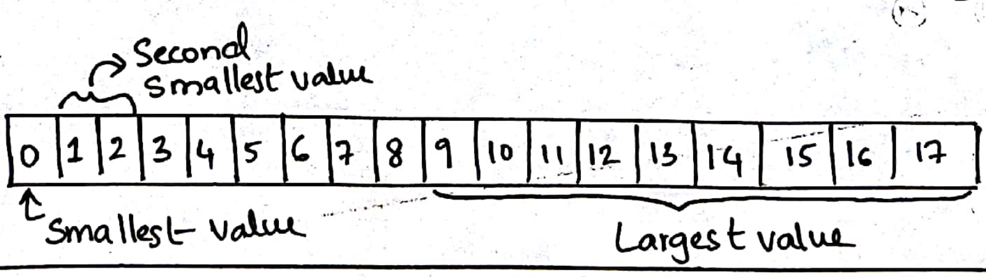
(3.) Consider a binary heap with 18 elements. We know that the element with the smallest key appears in position 0. of the array.

(a) In what location could the element with the second smallest key appear?

In a Minimum binary heap, the smallest element is always the root. The second smallest element can be either of the children of the root node. Thus in the array, second smallest number appears either in <u>position 1 or position 2</u>.

(b) In a Minimum binary heap, the largest element will be at any one of the leafs of the binary tree.
∴ for a tree of size n, largest element will be at any of <u>the last n/2 position</u>.
∴ For an array of 18 elements, Largest will be in one of the following position.
[9, 10, 11, 12, 13, 14, 15, 16, 17]





Note: Index starts at zero.
∴ 18 elements = 0 to 17

Name: Vivek Koodli Udupa

④ Construct an algorithm to reverse the contents of a Stack S of n integers. You may use an auxilary stack and one auxilary variable.

To reverse the contents of a stack of n integers, we will ~~fo~~ use the following algorithm.

Step 1 : Take a stack S with n integers. Set count to zero.

Step 2 : pop the last entry of the Stack ~~in~~ and store that Value in the auxilary variable V.

Note: Stack follows First In Last Out order.

Step 3 : pop rest of the elements of the Stack into the auxilary Stack T.

Step 4 : Now Stack T has (n-1) values in reverse order. V has one Value. Now set another variable to keep count of the (~~pop~~) pop done on stack T. When the pop count = Main count, pop the element from V. All these pop's are pushed into main Stack S.

Step 5 : Incriment the main count.

Step 6 : Follow Step 2 through Step 5 ~~s~~ till the ~~st~~ reversed stack is Stored in Stack S. This takes (n-1) iterations

Name: Vivek Koodli Udupa

Python implementation:

```
class Stack :
    def __init__(self) :
        Self.items = []

    def push(self, data) :
        Self.items.insert(0,data)

    def pop(Self) :
        return self.items.pop(0)

    def show(self) :
        n = self.size()
        for i in range(n) :
            print(self.items[i], "  ", end="  ")
        print()

    def size(self):
        return len(self.items)

#Functions
def reverse(r) :
    #Initialize a Temporary stack
    T= Stack()

    n = r.size()
    count = 0
    while (count < n-1) :
        r,T,v = Switch1(n,r,T)
        r,T = Switch2(n, count, r, T, v)
```

PTO

Name: Vivek Koodli Udupa

```
        count = count + 1
    return r

~~def switch1(n, orig, T)~~
def switch1(n, S, T):
    for sc in range(n):
        if (sc == 0):
            V = S.pop()
        else:
            T.push(S.pop())
    return S,T,V

def switch2(n, count, S,T, V):
    for sc in range(n):
        if (sc == count):
            S.push(V)
        else:
            S.push(T.pop())
    return S,T
```

In the above code, "count" is used to count the total iterations.
S → original stack T → auxilary stack. V → auxilary int Variable
Function switch1 puts elements from S into T and V
Function switch2 puts eliments back into S.
we will consider a small example for better understanding the algorithm.

P.T.O

Name : Vivek Koodli Udupa

Example: We will consider a simple array of 5 elements [0, 1, 2, 3, 4]. We will use the algorithm discussed to reverse the above mentioned array.

(~~Remember~~) Note: Stack follows first In Last out so popping elements from S and pushing them into T will look like:

S = [0, 1, 2, 3, 4] After the operation S = [ ]
T = [ ] T = [4, 3, 2, 1, 0]

Count	Original Stack (S)	Auxilary (V)	Temp. Stack (T)
0	0 1 2 3 4 $\boxed{1}$ 2 3 4 0	0	4 3 2 1
1	$\boxed{2}$ 3 4 1 0	1	~~0~~ ~~8~~4 3 2
2	$\boxed{3}$ 4 2 1 0	2	0 1 4 3
3	4 3 2 1 0	3	0 1 2 4
	$\Downarrow$ $\boxed{4\ 3\ 2\ 1\ 0}$	Successfully Reversed!!	

As we can see above, in 4 (count = 3) iterations, we reversed the original stack. We need to pop all n elements from S to T and then back from T to S. And we must do this (n-1) times

$\therefore$ Complexity = ~~O(n)~~ $O(n * n * (n-1)) =$ ~~$O(n^3)$~~ ~~$O(n^2)$~~

$\boxed{\cong O(n^3)}$

Name: Vivek Koodli Udupa

5 You are given an array A of non-negative integers and you need to determine whether there are two indices i & j such that $A[i] = 3 \times A[j]$.

a) Describe an algorithm with worst case running time $O(n^2)$

The most basic approach to the given problem would be to have two loops and iterate through every element and check for the given condition.

This method takes $O(n^2)$ as there are 2 loops iterating through all n- elements.

b) Describe another algorithm with worst-case running time better than $O(n^2)$

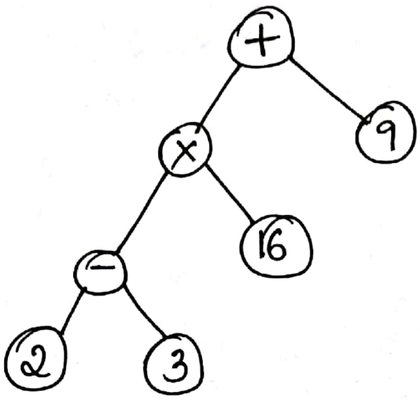
Given that the ~~(arry)~~ array contains only non-negative integers, we can sort the array. Sorting ~~takes~~ has a complexity of ~~O(logn)~~ $O(n \log n)$ [Merge Sort / quick Sort].

With a sorted array, we can skip all the indices prior to the index in consideration and check for the remaining values. This way the complexity is definitely below $O(n^2)$ and on or above $O(n \log n)$.

Name: Vivek Koodli Udupa

(6) The following is the pre-order traversal of a binary expression tree.

$$+ \times 16 - 2 \; 3 \; 9$$



$$((2-3) \times 16) + 9$$

$$(-1 \times 16) + 9$$

$$-16 + 9$$

$$= \underline{\underline{-7}}$$