EITANBROWN 346816549
YONI RUBIN 648831051

In questions in which you need to write a recursive function, you can write a shell function (a normal function that does not have iterations and/or recursions) that calls a recursive function.

Question 1: Define a recursive function that accepts a string and checks if it is a palindrome. The function accepts a string of characters and its length. The function returns True if the string is a palindrome otherwise returns False.

```
function IsPali(L):
   // Base case - test case for me personally
         //1 or 0 should always be true
   if length(L) <= 1:
      return true

   // Recursive case The first and last term are equal then it will do the substring of the second
term and the second to last term
   if first(L) == last(L):
      L = substring(L, 2, length(L) - 1)
         //L=123321
         //subtring returns 2332
      return IsPali(L)
   else:
      return false
```

<u>**Question 2:**</u>

The natural numbers 1 - 100 have the following attribute:
For each number N you can make the series N_1 , N_2 , … N_k in the following way.
N_1 is the given number N.
If N_i is even then N_{i+1} is $N_i/2$
If N_i is odd then N_{i+1} is 3 * N_i + 1
In this series there is a value k in which Nk = 1.

a. **Write an algorithm that inputs from the user a natural number N and if the number is between 1 and 100 the algorithm calls a recursive function that prints the series of numbers from N (N_1 , N_2 , … N_k) and prints the sum of the series. And if not, then the algorithm does not do anything.**
   For example: If N = 6 then the function will print the series:
   6 3 10 5 16 8 4 2 1
   And the printed sum will be 55.
   For example: If N = 7 then the function will print the series:
   7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
   And the printed sum will be 288.
b. **What is the run time complexity of your function?**

Run time complexity is O(1) (constant) since for every option from 1-100, there is an upper bound that acts as a constant number
For example the most amount of steps in the range of N for 1-100, is when N=100 and the amount of steps taken is 3142

```
int sum = 0;
function ImBadAtNamingThings(N, sum)
  // Check for invalid input
  if N > 100 or N < 1 then
    return sum -1 // Invalid input
  end if
  // Base case: If N is 1, add to sum, print N, and return sum
  if N == 1 then
    sum = sum + N
    print N
    return sum
  end if
  // Print the current value of N
  print N
  // Add the current value of N to the sum
  sum = sum + N
  // Recursive case: Update N based on its parity and call the function recursively
  if N % 2 == 0 then
    N = N / 2
  else
    N = 3 * N + 1
  end if
  // Recursively call the function with the updated value of N
  return ImBadAtNamingThings(N, sum)
end function
// Get the user input for N
initialN = user input
// Call the function and store the result
result = ImBadAtNamingThings(initialN, sum)
// Print the result sum if the input was valid
if result != -1 then
  print "Sum: " + result
else
  print "Invalid input"
end if
```

Question 3: Implement a queue using a stack. Implemented in pseudocode the following operations: is-empty, enqueue, dequeue using stack operations. Do not use any additional data structure but only one stack. Hint: recursion. What is the run time complexity of each of the operations?

Q=StackCreate()

IsEmpty(Q): —-> O(1)
return Q.is-empty()

---

enqueue(Q): —> O(1)
Q.push(x)

---

dequeue(Q): —> O(n)

1) **//Check if the stack (Q) is empty**
   if Q.IsEmpty():
   return Q.IsEmpty()
2) **//Pop the top element from the stack (Q)**
   Top = Q.pop()
3) **//If the stack (Q) is now empty, the popped element is the front of the queue**
   if Q.is_empty():
   return top_element
4) **Else:**
   //Recursively dequeue to get to the bottom element
   Removed = dequeue(Q)
5) **//Push the top element back to restore the original stack**
   Q.push(Top)
   // Return the dequeued element from the bottom of the stack
   return Removed

Question 4: Execute the quick sort algorithm for the following array containing the values: <3,18,2,6,1,10,5,4,7> For each iteration show the pivot value and the array at the conclusion of the iteration.

Where p is the left bound of the array, r is the right bound of the array, and q is the pivot index

qsort(A,p,r)
if (p<r)
p q r
Initial (no q yet)
3, 18, 2, 6, 1, 10, 4, 5, 7
q=partition(A,p,r) //A is array, p is currently 1 since arrays start at 1, and r is 9
//after first partition array = 3, 2, 6, 1, 4, 5, 7, 10, 18
qsort(A,p,q-1) (p=1, q-1=6 after the first partition)
qsort(A,q+1,r) (q+1=8, r=9 after the first partition)

p q r
Initial (no q yet)
3, 18, 2, 6, 1, 10, 4, 5, 7
q=partition(A,p,r) (A is array, p is currently 1 since arrays start at 1, and r is 9)
//after first run through q=7

partition(A,p,r)
pivot=A(r) //A(7)=4
i=index before p //0 which is out of bounds but that's fine
For j=p, to (and include) index before r, j=j+1 //initial value of j is 1, index before r is 8, this loop runs 7 times
If A[j] <=pivot
//first iteration A(1)(3)<A(9)(7)
//second iteration j=2 18<7 false, nothing happens
//third iteration j=3 2<7
//fourth iteration j=4 6<7
//fifth iteration j=5 1<7
//sixth iteration j=6 10<7 false, nothing happens
//seventh iteration j=7 4<7
//eighth iteration j=8 5<7
i=i+1
//first iteration i=1
//third iteration i=2
//fourth iteration i=3
//fifth iteration i=4
//seventh iteration i=5

//eighth iteration i=6
Swap (A[i],A[j])
//first iteration swaps the element with itself
//third iteration swaps 18 and 2
//3, **2, 18**, 6, 1, 10, 4, 5, 7
//fourth iteration swaps slots 3 and 4
//3, 2, **6, 18**, 1, 10, 4, 5, 7
//fifth iteration swaps slots 4 and 5
//3, 2, 6, **1, 18**, 10, 4, 5, 7
//seventh iteration swaps slots 5 and 7
//3, 2, 6, 1, **4**, 10, **18**, 5, 7
//eighth iteration swaps slots 6 and 8
//3, 2, 6, 1, 4, **5**, 18, **10**, 7
//end of loop
Swap (A[i+1],A[r])
//i=6 i+1=7
//swap A(7) and A(9)
//3, 2, 6, 1, 4, 5, 7, 10, **18**
Return i+1 //return 7

Will not be showing all the rest as it is a few pages to make a wall of repetitive text
index q is = 7 and A[q] = 7
3 2 6 1 4 5 **7** 10 18
index q is = 5 and A[q] = 5
3 2 1 4 **5** 6
index q is = 4 and A[q] = 4
3 2 1 **4**
index q is = 1 and A[q] = 1
1 2 3
index q is = 3 and A[q] = 3
1 2 **3**
index q is = 9 and A[q] = 18
1 2 3 4 5 6 7 10 **18**
Tada the sorted array

<u>Question 5</u>

<u>Given: A binary tree (not necessarily a BST) with the following transversals:</u>

Preorder: 10, 17, 8, 2, 6, 3, 0, 13, 9, 5, 4, 12, 42 ***(Root, Left, Right)***

Inorder: 2, 8, 3, 6, 17, 10, 4, 5, 12, 9,13, 0, 42 ***(Left, Root, Right)***

<u>What is the postorder transversal of the tree? (Left, Right, Root)</u>

<u>(A bit of help for you: Draw the tree and then transverse it)</u>

<u>Drawn Tree</u>

```
        10
       /   \
     17      0
    /       / \
   8      13   42
  / \      /
 2   6    9
      \    \
       3    5
           / \
          4  12
```

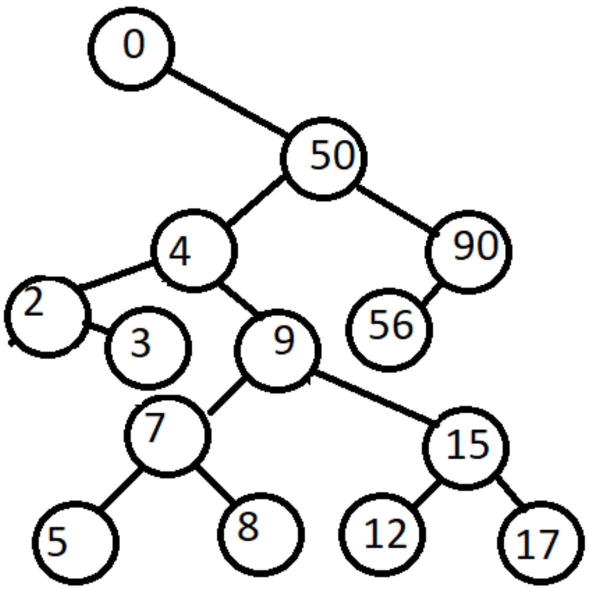
Post Order: **2, 3, 6, 8, 17, 4, 12, 5, 9, 13, 42, 0, 10**

<u>Question 6:</u>

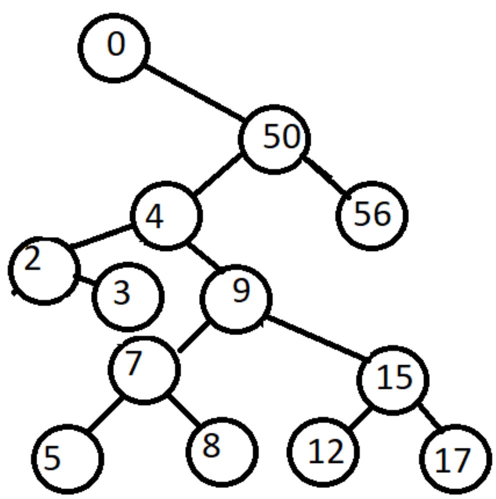
a. Build a BST from the following values: 0, 50, 4, 90, 9, 15, 17, 7, 8, 2, 5, 12, 56, 3

b. Delete from the tree that you built in part a the following values. Draw the tree after each deleted value. 90, 0, 50, 4, 5, 12

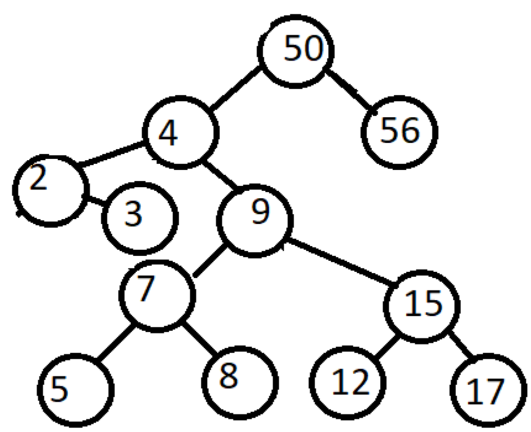
Initial tree



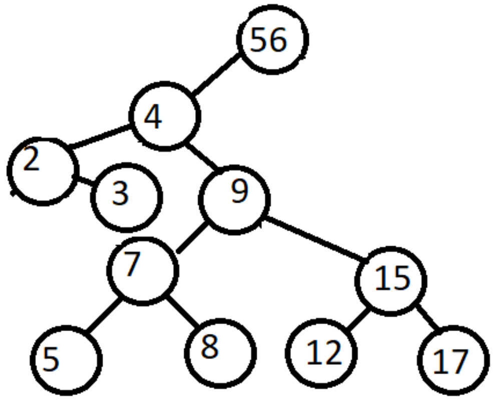
Step 1: Tree after deleting 90



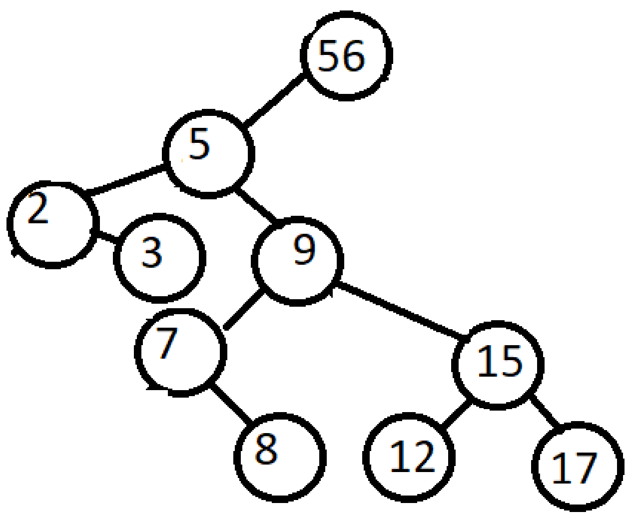
Step 2: Tree after deleting 0



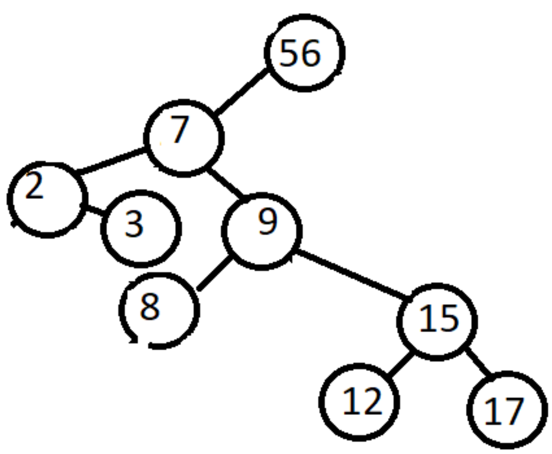
Step 3: Tree after deleting 50



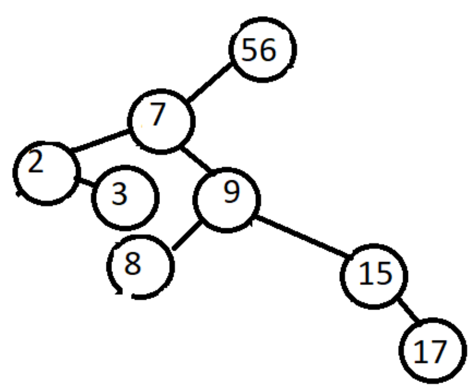
Step 4: Delete 4 (find successor to 4, replace 4 with successor, delete successor node)



Step 5: Delete 5 (same process as deleting 4)



Step 6: Delete 12



Question 7 Write a recursive function that accepts a binary tree and returns the number of leaves in the tree. What is the run time complexity of your function? Explain.

```
countLeaves(T)
If emptyTree(T) return 0
If leftT=null and rightT=null return 1 //meaning we found a node with no children
x=0
y=0
If left(T)!=NULL
        x=countLeaves(leftT)
If right(T)!=NULL
        y=countLeaves(rightT)
Return x+y
```

The function itself is O(1). The whole algorithm is θ(n) where n is the number of nodes since we need to go through every path to find the leaf/s at the end we will need to go through every node

Question 8:

<u>Given:</u>
<u>a BST containing distinct values.</u>
<u>In order to print the values in ascending order, you can call the function TreeMinimum(T) and afterwards perform n-1 calls to the Tree-Successor function. Prove that the runtime complexity of the above algorithm is Θ(n)</u>

To print the values of a BST in ascending order,
→ you start by finding the minimum value, then repeatedly find the next successor.
→ Finding the minimum takes O(h), and each successor takes O(h)
→ h=the height of the tree.
→ Since you perform these operations n times because it is once per node, the overall complexity is O(n * h).
→ In a balanced tree, this is (O(nlogn) like we said in class but in the worst case , it's O(n^2).
However, since each node is processed exactly once bec. It is ascending, the total time spent is proportional to the number of nodes, resulting in a complexity of <u>Θ(n)</u>
(assuming we have access to the parent field of each node)