Clarifications about trees:

Def. *(Height of a Tree)*

The height of a tree is the number of **edges** along the **longest path** from the root node to a **leaf** node

The height of an empty tree is -1
The height of a tree with one node is 0

Def. *(Depth of a Node)*

The depth of a node is the number of **edges** along the **path** from the root node to the **specified node**

Def. *(Full Binary Tree)*

A binary tree is called a full binary tree if every node has 2 children except for the leaf nodes (nodes on the last row)

Def. *(Complete Binary Tree)*

A binary tree is called a complete binary tree if every row in the tree is completely filled excepted for the last row. The nodes in the last row should be as far left as possible.

What is recursion?

- U-sub, By-Parts, Partial fraction decomposition, are all techniques used to solve integrals
- Recursion is a technique used to solve problems
- Recursion is the process in which a function calls itself

How does recursion work?

- Suppose I had a function **measure()** that uses a ruler of size 30cm (standard ruler) to measure the length of sticks!
- Now I have a very long stick (hehe) that I want to measure, so I call the **measure()** function to measure it!
- How can you do that?!? What if the stick is longer than 30cm? We only have a 30cm ruler!!!
- We can measure the stick with the following technique:

    1. If the stick can be measured, then good! Measure it! We are done, skip the rest of the steps.
    2. The stick is too long! Break the stick into smaller pieces and measure the smaller sticks!
    length1 = measure(smaller stick 1) and length 2 = measure(smaller stick 2)
    3. Combine the solution of the smaller problems to solve the bigger problem!
    fullLength = length1 + length2

- Who breaks the problem down and combines the solution?

→ A recursive function is responsible for all of the following:

1. Being able to solve simple problems
2. Being able to break complex problems into smaller problems and solving the smaller problems (by **calling itself** on the smaller problems to solve them)
3. Being able to combine the solutions to the smaller problems to formulate the solution for the larger complex problem

Therefore, we can measure a stick of any length with just a 30cm ruler! Amazing!

This is exactly what we want! The result of recursion is the ability to solve any problem of any size from just knowing how to solve a simple small problem!

Recursive Example:

Write a function *int fib(int n)* that returns the n-th fibonacci number
Recall that the fibonacci sequence is defined in the following manner

$$\text{fib(n)} = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

Soln.

```
int fib(int n) {
   if (n == 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

Two basic types of recursion (There are more though)

The difference between the following types is the way in which they break the problem down

- **Divide and Conquer** reduces the size of the problem by a factor of input size n (e.g. Cut problem of size n in half, $[n] \implies [n/2] + [n/2]$)

- **"n-1" Recursion** reducces the size of the problem by a constant factor (e.g. Cut problem of size n by a constant factor 1, $[n] \implies [1] + [n-1]$)

Creating a recursive algorithm

- We create a recursive algorithm by following the following "Recursion Recipe" created by Professor Nick Cheng at the University of Toronto

Nick's Recursion Recipe

1. Determine when the input is so **small** that the solution is easy to find
2. Describe the solution to the problem when the input is **small**
3. Take a *leap of faith*. **BELIEVE!!!**
   Suppose that for any input that is not **small**, you already had the solution to the same problem with any smaller input
4. For input that is not **small**, use the solution(s) to the same problem with smaller inputs to construct the solution to the original problem
   → Combining the smaller solutions to form a solution to the big problem can get rather tricky. It may require some creativity.
   → When writing code, a **solution to the same problem with smaller input** is a **recursive call to the function we are writing**

Too Long Didn't Read the Recipe

- Find all base cases
- Figure out the inductive step (how do I get from problem size n to a smaller problem of size maybe $n - 1$)
- Believe that your code works! Verify it by testing it.

Example (Easy) - Linked list insert at tail

Problem: Write a function **insertTail** that inserts a new node into a linked list at the tail position. The function will be provided a pointer to the head of the linked list and another pointer to the new node that needs to be inserted. The function should return the head of the linked list after insertion.

Q: What does our function do?
→ Given the head of a linked list and a node to insert, insert the node into the linked list at the tail and return the head of the linked list

Q: What is our input?
→ A linked list

Q: What is the size of our input n?
→ The size of the input n is the number of nodes in our linked list

Base case: n = 0
Given the head of an empty linked list and a node to insert, we want to return the node itself

Q: Why don't we have a base case n = 1, n = 2? n = 3? n = 4? ...
→ We can most definitely have those bases cases, but we want to implement as few bases cases as we can, while still maintaining the ability to solve small problems. More base cases won't do harm, but it means that you have to write extra code for no reason! **Why do more when you can do less?**

IMPORTANT NOTE: Sometimes, you will need more than 1 base case! Such an example is the fibonacci function above. In the case that you don't implement enough base cases, your program will NOT WORK! This is why it is VERY IMPORTANT to test your code!

Inductive Hypothesis:

Suppose that for any linked list of size (n - 1) or less, a call to **insertTail** will insert a given node into the linked list at the tail and return the head of the linked list

Inductive Step: n > 0

Given the head of a linked list with size > 0 and a node to insert
- Call the **insertTail** function to insert the node into the **smaller** (size n - 1) linked list that starts with the next node of head (i.e. The head of the new linked list is head→next)
- The **return value** from this call should be the **new head** of the smaller linked list
- Therefore, replace the value of **head→next** with the return value of the function call

Code.

```
struct node *insertTail(struct node *head, struct node *newNode) {
    // Base case (n = 0) empty list
    if (head == NULL) {
        newNode→next = NULL;
        return newNode;
    // Inductive step (n > 0)
    } else {
        head→next = insertTail(head→next, newNode);
```

```
        return head;
    }
}
```

Example (Hard) - Depth First Search (Iterative)

Yikes. Doing it using loops is so hard in C that I'm going to use pseudo-code for this...

G is a graph with verticies and edges

s is the start node that we will begin DFS on

```
DFS(G, s):
    # Keep track of all visited nodes
    visited = []
    # Stack used to determine which nodes to visit next
    stack = []
    # Insert the start node onto the stack
    stack.push(s)
    # Continue to visit nodes until there are no more nodes to visit
    while (not stack.is_empty()):
        # Remove a node from the stack for consideration
        s = stack.pop()
        # Make sure the node has not been visited before
        if s not in visited:
            # Visit node s
            visit.add(s)
            # Put all the the neighbours onto the stack
            for each_neighbour of node s:
                stack.push(each_neighbour)
```

Pseudo-Code without comments:

```
DFS(G, s):
    visited = []
    stack = []
    stack.push(s)
    while (not stack.is_empty()):
        s = stack.pop()
        if s not in visited:
            visit.add(s)
            for each_neighbour of node s:
                stack.push(each_neighbour)
```

Example (Hard) - Depth First Search (Recursive)

The recursive solution is much more easier to write! This is one of the reasons why we use recursion!

```
DFS(G, s, visited):
    # visit the node
    visited.add(s)
    # DFS the non-visited neighbours
    for each_neighbour of node s:
        if each_neighbour not in visited:
            DFS(G, each_neighbour, visited)
```

Example (Hard) - Height Balanced Trees

Problem: Write a function **isBalanced** such that given a pointer to the root of a tree, determine whether or not the tree is balanced or not. (You will learn more about balanced trees in CSCB63).

Note: Balanced Binary Trees are NOT ON THE EXAM (unless paco adds them to the lecture notes).

Balanced Tree: A tree is balanced if and only if for every node in the tree, the heights of the left and right subtree differ by at most 1

Q: What does our function do?
→ Given a pointer to the root of a tree, finds out if its balanced or not. The return value... is unknown for now... Why? Returning whether or not the tree is balanced is not enough information, you cannot combine the smaller solution to solve the bigger problem. You won't know this fact for now, but you'll figure it out once you reach the inductive step and can't combine the solution.

Q: What is our input?
→ The pointer to the root of a tree

Q: What is the size of our input n?
→ The number of nodes in the tree

Base case: n = 0
Given a tree with no nodes, the tree is indeed balanced

Inductive Hypothesis

Suppose that for any tree of size (n - 1) or less, a call to **isBalanced** will determine whether or not the tree is balanced or not

Induction Step n > 0

Given the root of a tree with size > 0
- Check if the left subtree is balanced with a call to **isBalanced** with the left subtree
- Check if the right subtree is balanced with a call to **isBalanced** with the right subtree
- Check if the heights of the left and right subtrees only differ by at most 1
- If all of the checks pass, then the tree is indeed balanced

Where do we get the height of the subtrees from?
→ This is why returning "just" the balance fact is not enough, you will also have to return the height of the tree to be able to use it for combining solutions

Code.

```
// Given tree starter code.
typedef struct tree {
   struct tree left;
   struct tree right;
}

// Return data type that you defined.
typedef struct treeData {
```

```
    int balanced;
    int height;
}

struct treeData *isBalanced(struct tree *root) {
    // Create the return value
    struct treeData *p = (treeData *) calloc(1, sizeof(treeData));
    // Check for enough memory
    if (p == NULL) { return NULL; }
    // Base case (n == 0)
    if (root == NULL) {
        p→balanced = 1;
        p→height = 0;
        return p;
    // Induction step (n > 0)
    } else {
        struct treeData leftData = isBalanced(root→left);
        struct treeData rightData = isBalanced(root→right);
        // Bad Data
        if (leftData == NULL || rightData == NULL) {
            free(leftData);
            free(rightData);
            return NULL;
        }
        // Check that it's balanced
        if (leftData→balanced && rightData→balanced &&
                    (abs(leftData→height - rightData→height) <= 1)) {
            p→balanced = 1;
        } else {
            p→balanced = 0;
        }
        // Note there is no built-in max function in C, so you have to write a helper
        // yourself
        p→height = max(leftData→height, rightData→height) + 1;
        // Free the unneeded data
        free(leftData);
        free(rightData);
        return p;
    }
}
```

Practice Questions:

Hint: Some questions are very complex and may need helper functions that are recursive

1. Given an array of integers and its length, find the sum of the numbers in the array

2. Given an array of integers and its length, find the average of the numbers in the array

3. Given an array of integers, its length and another integer x, find the number of integers greater than or equal to x

Questions 1-3 from Professor Nick Cheng's CSCA48-2018 list of "easy" questions

4. Given an array of integers and its length, find the second smallest element in the array

5. Given an array of integers and its length, find the sum of the maximum element and the minimum element in the array

Questions 4-5 from Professor Nick Cheng's CSCA48-2018 list of "hard" questions

6. Given an array of integers its length and some integer n, find the n-th smallest number in the array in $\mathcal{O}(n)$ average time? (Hint. Augmentation of quicksort and usage of divide and conquer)

Question 6 from Professor Vassos Hadzilacos' (brutal) CSCC73-2019 course