Recursion (Rosen, 6th edition, Section 4.3, 4.4) Carol Zander

For recursion, the focus is mostly on recursive algorithms. While recursive definitions will sometimes be used in definitions (you already saw this with the definition of logical expressions), the focus will be on recursive algorithms. In section 4.3, read through the recursive definition examples, but skip the sections on Structural Induction and Generalized Induction. In 4.4, we will not consider proving algorithms are correct which uses induction, material that will be covered later in the quarter.

What is recursion?
Recursion is a powerful problem-solving technique that uses a divide-and-conquer approach. You break a problem down into a simpler (usually smaller) version of itself and then you solve that. Continue to do this until you get down to a very simple case, called the base case.

Code-wise, it's nothing new – a recursive function is one which calls itself (must have base case – some code that will eventually get executed that doesn't call itself, otherwise you get infinite recursion). All the examples shown in these notes are coded in C++.

Steps to solving a recursive problem

1. Write down precisely what your function does. You must believe that it will do this task, even though you haven't written it yet.

2. Solve the base case (which is usually easy). For example: a sequence of ints, n = 0. Or an empty array, or array with one element. Or an empty list. Or an empty tree.

3. Solve the recursive part logically. Take your problem and break it down into parts, where one part is basically the same problem, but smaller than the original problem. For example:
   n items → n-1 items
   array of n items → array of n-1 items
   linked list of n items → linked list of n-1 items
   binary tree → its root, left subtree, right subtree
   When you think through the solution, try to use the words you wrote down in #1.

4. Code the recursive part. Be sure to think in terms of step #1 and call your function if you find that you need to solve that task. It takes some getting used to because you are using what you are writing. Pretend it already exists, that you are just calling a function like you always do.

Example – Factorial
Recursive definition for Factorial of n, fact(n) = n! (e.g., $4! = 4 \cdot 3 \cdot 2 \cdot 1$) are defined by $0! = 1$, and

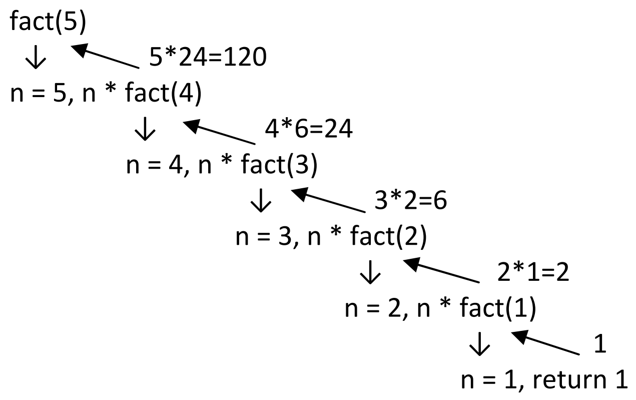
$$fact(n) = n \cdot fact(n-1)$$

for n = 0, 1, 2, … .

```
// does not take into consideration overflow
int fact(int n) {
   if (n < 0) return -1;
   if (n <= 1) return 1;
   return n * fact(n-1);
}

int main() {
   cout << fact(5) << endl;
   return 0;
}
```

Draw an execution tree (showing the calls and returns) of the execution of fact(5):



Notice that solving the problem, writing the code, has nothing to do with how it actually executes.

Example – Fibonacci numbers

Recursive definition for Fibonacci numbers $f_0, f_1, f_2, \ldots$, are defined by the equations $f_0 = 0, f_1 = 1$, and

$$f_n = f_{n-1} + f_{n-2}$$

for n = 2, 3, 4, … .

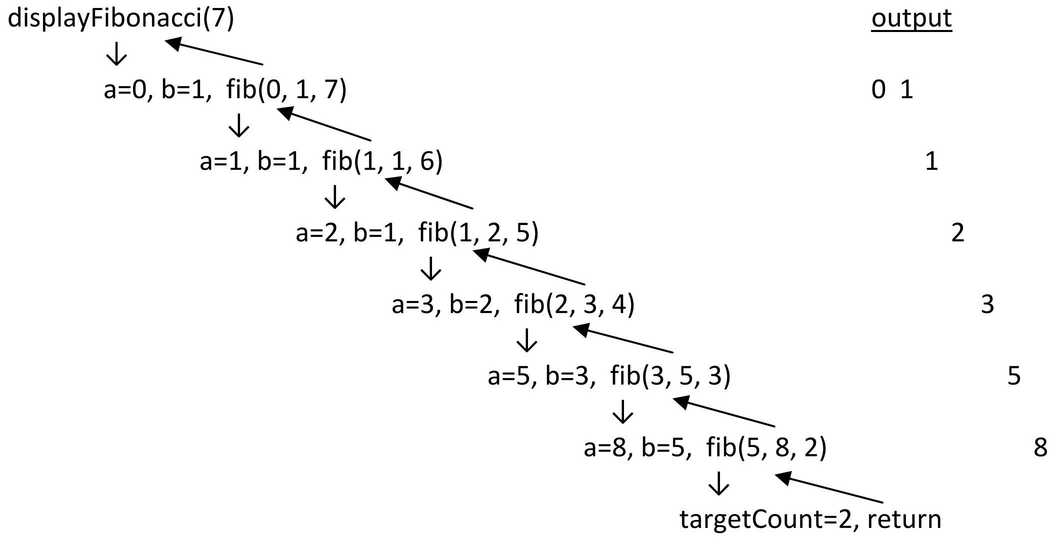
In Section 4.4, in the Recursion and Iteration section, your text shows the pseudocode for finding the fibonacci numbers using the recursive definition. The definition above essentially becomes the code. It is so inefficient though as you are doing all the same work twice. I've shown here a slicker recursive solution, based on the iterative solution:

```
void fib(int a, int b, int targetCount) {
   if (targetCount <= 2) return;
   a += b;
   cout << a << "  ";
   fib(b, a, targetCount-1);
}
```

```
void displayFibonacci(int targetCount) {
   int a = 0, b = 1;
   cout << a << "  " << b << "  ";
   fib(a, b, targetCount);
}
```

```
int main() {
   displayFibonacci(10);
   cout << endl;
   return 0;
}
```

Draw an execution tree (showing the calls and returns) of the execution of displayFibonacci(7):



<u>Example</u> – Displaying a linked list backwards

```
class List {
   friend ostream &operator<<(ostream&, const List&);

public:
   List();                                      // default constructor
   ~List();                                     // destructor
   List(const List&);                           // copy constructor
   bool insert(NodeData*);                      // insert one Node into list
   void buildList(ifstream&);                   // build a list from datafile
   void printBackwards() const;                 // print the list in reverse

   // needs many more member functions to become a complete ADT

private:
   struct Node {              // the node in a linked list
      NodeData* data;         // pointer to actual data, operations in NodeData
      Node* next;
   };

   Node* head;                                  // pointer to first node
   void printBackwardsHelper(Node*) const;      // actual backwards printer
};
```

```
                 _____      _____      _____      _____
list.head-->|__|_-|--->|__|_-|--->|__|_-|--->|__|_/|
             ↓          ↓          ↓          ↓
             10         20         30         40
             'b'        'x'        'y'        'z'
```

```
void List::printBackwards() const {                  // public interface function
   printBackwardsHelper(head);
}

void List::printBackwardsHelper(Node* current) const {     // private utility
   if (current != NULL) {
      printBackwardsHelper(current->next);
      cout << *current->data;
   }
}
```

Draw an execution tree (showing the calls and returns) of the execution of printBackwardsHelper(head):

```
printBackwardsHelper(→10)                                output
      ↓
      →10,  pbh(→20), cout                                       10 b
            ↓
      →20, pbh(→30), cout                                     20 x
            ↓
      →30, pbh(→40), cout                                  30 y
            ↓
      →40, pbh(NULL), cout                              40 z
            ↓
            NULL
```

Example – Towers of Hanoi

The Towers of Hanoi is a mathematical game or puzzle. It consists of three pegs, and a number of disks of different sizes which can be put onto a peg. The puzzle starts with the disks in a stack in order from smallest to largest on one peg, smallest at the top. The objective is to move the entire stack from one peg to another peg, obeying the following rules:

- Only one disk may be moved at a time
- One move consists of taking the top disk from one of the pegs and putting it onto another peg, on top of the other disks on that peg
- No disk may be placed on top of a smaller sized disk

The puzzle was invented by the French mathematician Édouard Lucas under the name N. Lucas de Siam in 1883. There is a legend about a temple in Benares with a dome which marked the center of the world. Within the dome, there is a large room with three posts in it surrounded by 64 golden disks. The priests of Hanoi, move these disks, in accordance with the rules of the puzzle. According to the legend, when the last move of the puzzle is completed, the universe will come to an end. (There are many variations on this legend.)

If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64}-1$ seconds or roughly 585 billion years to finish.

Solve recursion:

Step 1: moveDisks – **will** move any number of disks from a peg to another peg using extra peg as an auxiliary

Step 2: base case – one disk, easy, just move it

Step 3: recursive part – if you have a function that will move any number of disks from one to another,
Move n-1 disks from the peg they are initially on, the *from peg*, to the *auxiliary peg*
Then move the nth disk (largest disk) to the peg you want to ultimately move them to, the to peg
Move n-1 disks sitting on the *auxiliary peg* to the peg you want to ultimately move them to, *to peg*

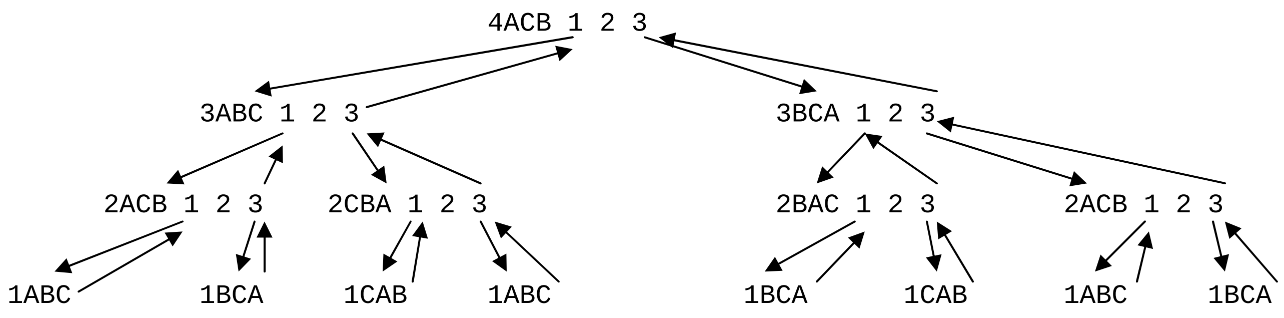
Step 4: code

```
void moveDisks (int n, char fromPeg, char toPeg, char auxPeg) {
   if (n == 1) {                                           // base case
      cout << setw(6) << n
           << "            " << fromPeg
           << "            " << toPeg << endl;
   }
   else {                                                  // recursive step
      moveDisks(n-1, fromPeg, auxPeg, toPeg);              // call this 1
      cout << setw(6) << n                                 // call this 2
           << "            "  << fromPeg
           << "            " << toPeg << endl;
      moveDisks(n-1, auxPeg, toPeg, fromPeg);              // call this 3
   }
}

int main() {
   cout << "Move Disk #      From Peg     To Peg" << endl;
   moveDisks(4, 'A', 'C', 'B');

   return 0;
}
```

Draw an execution tree (showing the calls and returns) of the execution of movedisks(4, ‘A’, ‘C’, ‘B’) and display output:



Output:
```
Move Disk #      From Peg     To Peg
     1             A            B
     2             A            C
     1             B            C
     3             A            B
     1             C            A
     2             C            B
     1             A            B
     4             A            C
     1             B            C
     2             B            A
     1             C            A
     3             B            C
     1             A            B
     2             A            C
     1             B            C
```

<u>Example</u> – Does a char array form a palindrome? You are given the char array and its size (or length).

Palindrome – reads the same forwards and backwards, e.g., remove punctuation:

dad   madam   Dammit I’m mad
A man, a plan, a canal, Panama
I’m a lasagna hog, go hang a salami
Doc, note, I dissent, I diet on cod
Are we not drawn onward to new era?

```
// given an char array, it will return whether the char array is a palindrome
bool isPalindrome(char a[], size) {
   if (size < 0) return false;
   if (size == 0 || size == 1)
      return true;                        // base case

   return (a[0] == a[size-1]              // first and last char are the same

           // remaining char array, without first and last char is a palindrome
           && isPalindrome(&a[1], size-2);
}
```