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.

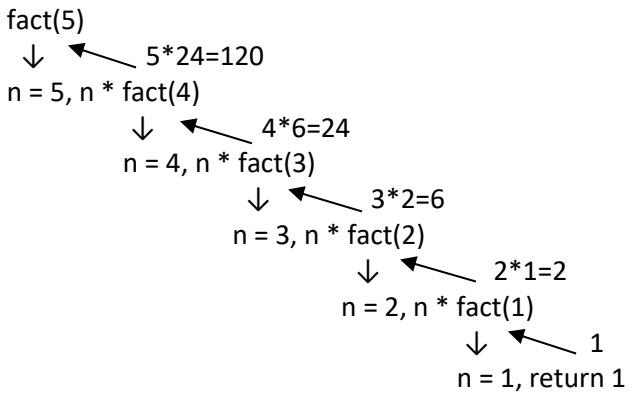
3. Solve the recursive part logically. Take your problem and break it down into parts, where one part is basically the same problem, but smaller than the original problem. For example:
   n items → n-1 items
   array of n items → array of n-1 items
   linked list of n items → linked list of n-1 items
   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 · fact(n-1), (e.g., $4! = 4 \cdot 3!$)
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;
}
```

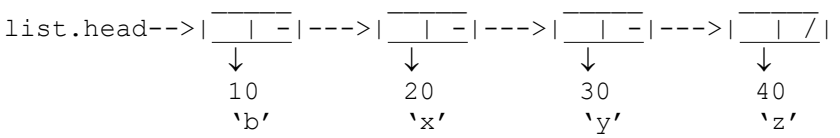
Draw an execution tree (showing the calls and returns) of the execution of fact(5):



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

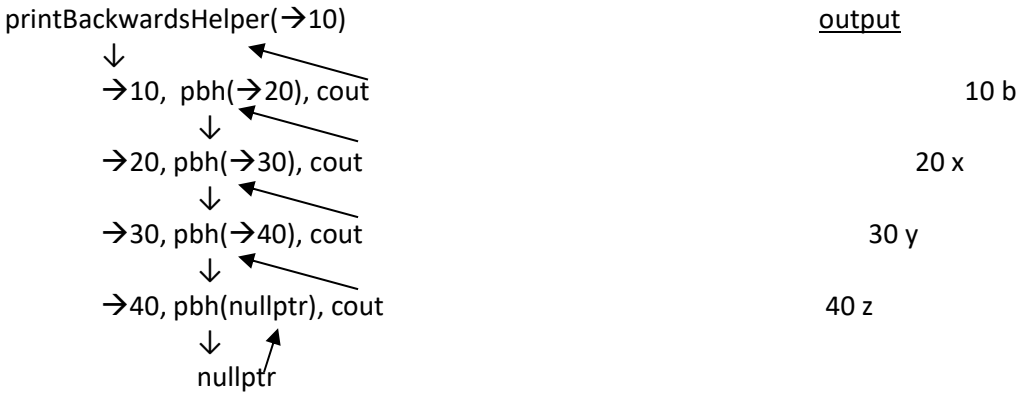
Example – Displaying a linked list backwards

```
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
};
```




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

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

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



Example – 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., (with removed punctuation):

dad	
madam	
Dammit I'm mad	(data would be dammitimmad)
A man, a plan, a canal, Panama	(data would be amanaplanacanalpanama)
I'm a lasagna hog, go hang a salami	And so on … (remove blanks, punctuation, make lowercase)
Doc, note, I dissent, I diet on cod	
Are we not drawn onward to new era?	

Solve it as a practice problem.