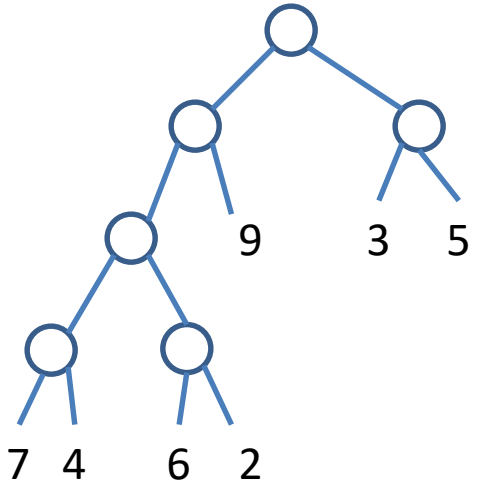
Recursion

We cover recursion in 150. Why do it again in 151?

First, good solutions to problems are often recursive. Here is a quick way to sort a list of objects: split the list in half, recursively sort each half, merge the two sorted halves back together. This is easy and runs much faster than any of the iterative sorts we have discussed.

Next, the only natural solutions to some problems are recursive. Try solving the Towers of Hanoi puzzle without recursion -- it is possible, but really hard. The recursive solution is 3 lines.

Suppose you have binary tree with numbers in the leaves. How would you sum those numbers?



This is easy with recursion -- just recursively sum the left and right subtrees and add them together.

Finally, sometimes recursive algorithms are easier to analyze than iterative algorithms. If you want to prove that what you are doing is correct, or if you want to find a good asymptotic analysis of your algorithm, recursion is often the way to go.

Clicker question: Which is the correct recursive version of the factorial function?

A

```
int fact(int n) {
    return n*fact(n-1);
}
```

B

```
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}
```

C

```
int fact(int n) {
    return n*fact(n);
    if (n <= 1)
        return 1;
}
```

D: Answers B and C are both correct

Here is something you should have taken from 150 -- the basic structure of recursive functions.

Consider this function:

```
public int factorial(int n) {
        return n*factorial(n-1);
}
```

This will never halt:

```
factorial(2) returns 2*factorial(1)
which is             2*1*factorial(0)
which is             2*1*0*factorial(-1)
which is             2*1*0*(-1)*factorial(-2)
                     etc/
```

To be successful, a recursive function must satisfy two properties:

a) It must have a non-recursive base case, i.e. some value of the argument(s) for which it can return without recursing. You should always test for this case before you recurse.
b) The recursive calls should be with arguments that make progress towards the base case. If your function has an argument n that is a non-negative integer and the base case is n=0, your recursive calls should be with arguments closer to 0 than n, such as n-1 or n-2, not n+1.

What does this function return if x is positive?

```
boolean foo(int x) {
        if (x%2) == 0
                return true;
        else
                return foo(x-1);
}
```

A. It crashes
B. It returns true if x is even, runs forever if x is odd
C. It returns true if x is even, false if x is odd
D. It always returns true

At one time recursion was considered inefficient and programmers were advised to avoid using it if it wasn't absolutely necessary. That is no longer the case. Modern processors are designed to optimize function calls in general and recursive calls in particular.

When any function is called, the processor pushes onto a stack a *frame* containing the arguments and local variables of the function. This frame isn't popped off until the function returns. If you have a function that recurses 100 times (such as a call to factorial(101), you will have 100 function frames built up on this stack, but even that isn't very much memory. It is possible for the runtime stack to run out of memory, but that is usually a signal that your recursion is wrong.

Here is an example of a nontrivial recursion. We want to write a function that takes as an argument a non-negative integer and prints the representation of this integer in other bases (such as binary -- base 2 or hexadecimal -- base 16). To start we'll print in base 10 since that is familiar. We need to print from left to right. To print 435, we will recurse to print 43, and then we will print 5. How do we get the numbers 43 and 5 out of 435? 43 is 435/10; 5 is 435%10. This works for any number -- to print 1234 we will recurse to print 123, then print 4. To print 123, we recurse to print 12, then print 3, and so forth.

Our base case will be a number small enough that we don't need to recurse. One digit numbers suffice for this: if our argument is less than 10 we will print it directly.

Here is the function for printing in base 10

```
public static void Print( int n) {
        if (n < 10)
                System.out.print(n);
        else {
                Print(n/10);
                System.out.print(n%10);
        }
}
```

If we change the 10s to 2s or 8s this will print in binary or octal. If we want to include base 16 we need digits that are greater than 10, usually written A, B, C, etc. Here is the full function for an arbitrary base up to 16:

```
static String digits = "0123456789ABCDEF";

public static void PrintDigit(int x) {
        System.out.printf( "%c", digits.charAt(x) );
}

public static void Print( int base, int n) {
        if (n < base)
                PrintDigit(n);
        else {
                Print(base, n/base);
                PrintDigit(n%base);
        }
}
```

Consider again the recursive function:

```
public static void Print( int base, int n) {
        if (n < base)
                PrintDigit(n);
        else {
                Print(base, n/base);
                PrintDigit(n%base);
        }
}
```

Note that we Print argument n by recursively Printing n/base -- the arguments are getting smaller. Our non-recursive base case is when n < base. Each step of the recursion is making progress towards its base case.

How do we know this works correctly? The base case is certainly correct: for any base b the representation of number n, which is less than b, is just n.

The correctness of the recursive case just depends on a fact from number theory: for any base b and any number n

b*(n/b) + (n%b) is the same as n.

The way we use place value to represent numbers means that writing the digit (n%b) to the right of the representation for (n/b) increases the value of n/b by a factor of b.