Lecture 10

# Recursion: Calling itself again

**Last time**:

When you have your own objects added into a data structure and want to sort them, you need to make sure that they are mutually comparable. In Java, this can be accomplished by implementing either *Comparable* interface or *Comparator* interface. In some cases, both!

- Your class can implement *Comparable* interface. In that case, you are to override *`compareTo(Object e)`* method in your class.
- You can also create separate classes that implement *Comparator* interface. The Comparator interface has only one method, *`compare(Object o1, Object o2)`*, to be overridden.
- This enables you to have different ways of sorting objects.

It is important to remember that there is a strong relationship between *`compareTo()`* and *`equals()`* methods.

**If x.compareTo(y) == 0, then x.equals(y) is true.**

Additionally, if the *`equals()`* method is overridden, then the *`hashCode()`* method also needs to be taken care of accordingly.

**If x.equals(y) is true, then x.hashCode() == y.hashCode().**

This time, let's explore a different way of thinking, called Recursion.

## Adding positive integers from 1 to n

1 + 2 + 3 + 4 + … + n

How would you solve this problem in Java?

```
public int sum(int n) {
     int result = 0;
     for(int i = 1; i <= ___n_____; i++)
          result = ___result + i_________;
     return result;
}
```

Looking at it more closely….

1 + 2 + 3 + 4 + … + n = n + (1 + 2 + 3 + 4 + … + (n-1))

We can say that sum(n) = n + sum(n-1).
Now, we can rewrite our solution differently.

```
public int recursiveSum(int n) {
     if(n == ____1______) return n;  // base case
     else return n + recursiveSum(_____n-1_________);
}
```

### Recursive definition

A definition contains a call to itself. Recursive programming has a method that calls itself.

A method call is a transfer of control to the start of the method. And, this can take place from within the method as well as from outside.

### Base case

However, we cannot call the method itself recursively to infinity (*Infinite recursion*).
The condition that leads to a recursive method returning without making another recursive call is the base case.

Every recursive method has a base case or base cases!

## Step 1: Conceptual View

```
public int recursiveSum(int n) {
     if(n == ____1____) return n;  // base case
     else return n + recursiveSum(___n-1_______);
}
```

```
recursiveSum(5)                                                  15
   5+recursiveSum(4)                                       10+5 = 15
      4+recursiveSum(3)                               6+4 = 10
         3+recursiveSum(2)                       3+3 = 6
            2+recursiveSum(1) = 1    1+2 = 3
```

**Bookkeeping**

For a recursive method, there is a lot of bookkeeping information that needs to be stored. Here, there are five pending calls till it hits its base case.

These pending calls may be very deeply nested. All of these pending calls are pushed onto the call stack. As the call hits the base case, then there is a return value and from that point, we pop the call from the stack to calculate and return a value.

**The question is "Is recursion efficient?"**

Usually, recursion is used because it simplifies a problem conceptually, not because it is more efficient.

**Example 1: Fibonacci Numbers**

In 1170, Leonardo Pisano, who is also known as Fibonacci, was born in Pisa, Italy.

A sequence of numbers such that each number is the sum of the previous two numbers in the sequence, starting the sequence with 0 and 1.

0, 1, 1, 2, 3, 5, 8, 13, 21, ___, ___ , ...

Let's define this recursively.

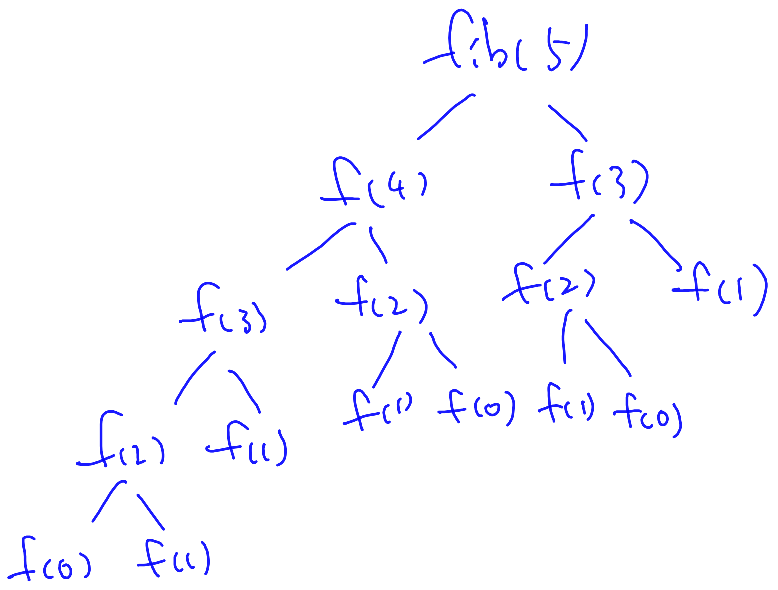
What is the base case(s)?

fib(0) fib(1)

What is the recursive case?

f(n-2) + f(n-1)

Assuming that we are trying to calculate fibonacci(5), what would happen conceptually? (**recursion tree**)



**Step 2: Implementation View**

Remember the binary search we discussed in lecture note 4?

Our goal was to find a given value's location (index) in an **ordered** array using the fewest number of comparisons. The solution was to divide the array in half and see which half has the given value and do the same thing again and again till we find (or do not find) the value.

Here is the code that we implemented at that time.

```
public int binarySearch(int[] data, int key) {
      int lowerBound = 0;
      int upperBound = data.length - 1;
      int mid;

      while(true) {
            if(lowerBound > upperBound) return -1; // not found
            mid = lowerBound + (upperBound - lowerBound)/2;
            if(data[mid] == key) return mid; // found
            else {
                  if(data[mid] < key) // right half
                        lowerBound = mid + 1;
                  else   // left half
                        upperBound = mid - 1;
            }
      }
}
```

Let's take a look at the binary search again.

Do you see it has a recursive nature?

What is the base case(s)?

What is the recursive case?

```
_private_int find(int[] data, int key, _int lB_, _int uB_) {
    if(lowerBound > upperBound) return -1; // base (not found)
    int mid = lowerBound + (upperBound - lowerBound)/2;

    if(data[mid] == key) return mid; // base case when found
    else { // recursive case
        if(data[mid] < key)
            return find(data, key,_mid+1_, _uB_);
        else
            return find(data, key,_lB_, _mid-1_);
    }
}
```

The method calls itself over and over but, each time with a smaller range of arrays than before till it hits one of the base cases.

If we were to make this to be a public method, it would look more complicated for users to specify lowerBound and upperbound. In fact, a user of this recursive method may not know what should be initial lowerBound and upperBound values.

To remove this kind of burden or confusion from users, we can provide the following public method to users and have an initial call to the private helper method in it.

```
public int find(int[] data, int key) {
    return find(data, key, _0_, _data.length_);
}
```

**Summary: characteristics of Recursive Methods**

- It calls itself
- When it calls itself, it does so to solve a smaller version of the same problem.
- There is a base case (or base cases) that does not call itself anymore.