Chapter 10: Recursive Problem Solving

Objectives

Students should

- Be able to explain the concept of recursive definition
- Be able to use recursion in Java to solve problems

Recursive Problem Solving

To solve a problem using recursive problem solving techniques, we break such a problem into identical but smaller, or simpler, problems and solve smaller problems to obtain a solution to the original one. For example, we can find the summation of integers from 0 to a positive integer *n* by finding the summation of integers from 0 to a positive integer *n-1* first then add to that result the integer *n* to obtain the result of the original problem. Finding the summation of integers from 0 to a positive integer *n-1* is considered a smaller problem than from 0 to *n*. Also, we know that if *n* equals 0, the result is just zero.

Mathematically, we can write the summation that we want as:

$$s(n) = s(n-1) + n$$

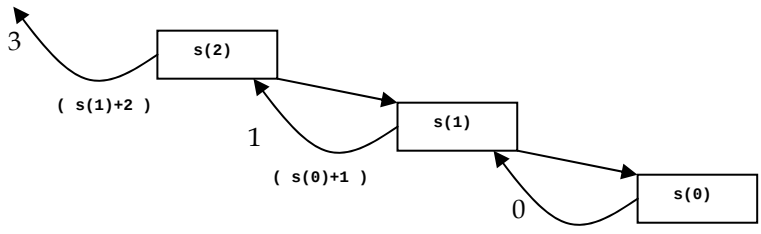
where $s(n)$ is the summation of integers from 0 to n for any positive integer n. Also, we know that $s(0) = 0$.

A Java method for finding such a summation could be written in a recursive fashion as in the following code segment.

```
public static int s(int n){                                    1
    if(n==0) return 0;                                         2
    return s(n-1)+n;                                           3
}                                                              4
```

In the body of *s()*, we can see that the method call itself but the input parameter used is smaller every time. If *n* equals 2, **`s(2)`** calls **`s(1)`**, wait for the value to be returned from the method, and adds *n*, which is now 2, to the returned value before returning the result to the caller. In a similar fashion, once **`s(1)`** is called, it invokes **`s(0)`**, wait for the value to be returned from the method, which is 0, and adds 1 to the returned value before returning the result to the **`s(2)`**.

The invocation explained can be depicted in the following picture.



Another example is recursively finding the factorial of *n*, *n*!. The definition of factorial might be described as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times \ldots \times 1 & \text{if } n > 0 \end{cases}$$

or, more precisely, as:

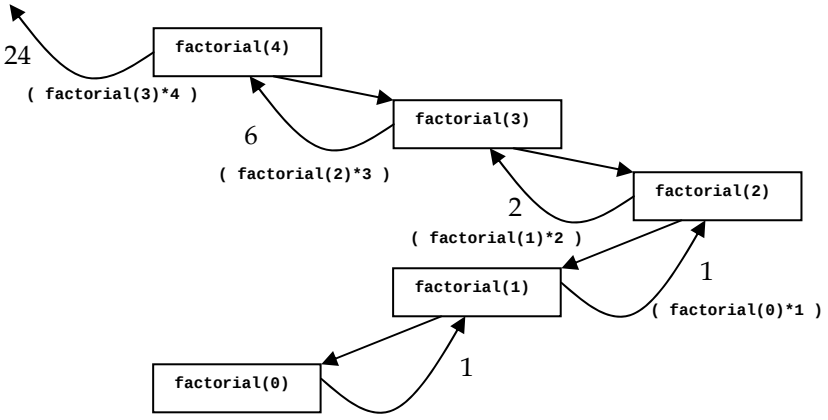
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

Again, *n*! can be solved recursively from (*n*-1)! and we know that 0! is 1. A Java method that can find *n*! recursively can be written as the following code segment.

```
public static int factorial(int n){                    1
    if(n==0) return 1;                                 2
    return factorial(n-1)*n;                           3
}                                                      4
```

As we can see from line 3, given an input *n*, the method call itself but with an input one which is one smaller than the original. When the input reaches 0, the method just returns a value without further recursively calling itself.

The picture below depict the method invocation of **`factorial(4)`**.



Solving the two problems shown here can also be done using iterative constructs such as *for* loops. You should try writing the methods shown above using iterative approach and compare them with the recursive approach in various aspects, such as their lengths, their implementation complexity, as well as the difficulty in coming up with the solutions via both approaches.

Recursive Method Design

A recursive method must have two parts. The first part determines the case where the recursive method invocation terminates. Cases where this part occurred are called the *base cases*. The other part recursively calls itself, but with simpler parameters. Cases where this part occurred are called the *recursive cases*. Each time the method is recursively called the parameters must be made simpler and must move towards the base cases. Failure to terminate recursive methods either from the missing of the base cases or the recursion never falls into the base cases results in an *infinite recursion* when the method is called during its associated program execution.

Example: Fibonacci Numbers

The *Fibonacci numbers* form a sequence of integer defined recursively by:

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

for every non-negative integer *n*.

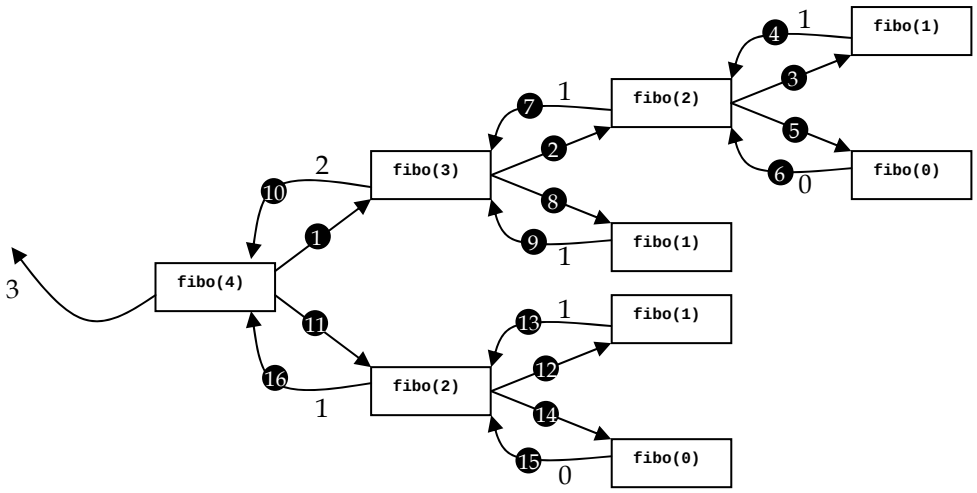
Therefore, the Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, … . The Fibonacci numbers model many things in nature, such as branching in trees and arrangement of pine cones.

The following Java program prints out the first 20 Fibonacci numbers with the use of a recursive method *fibo()*.

```
public class FiboDemo                                                    1
{                                                                        2
    public static void main(String[] args)                              3
    {   final int n = 20;                                                4
        for(int i=0;i<20;i++)                                            5
            System.out.print(fibo(i)+",");                               6
        System.out.println();                                            7
    }                                                                    8
    public static int fibo(int n){                                       9
        if(n<=0) return 0;                                              10
        if(n==1) return 1;                                              11
        return fibo(n-1)+fibo(n-2);                                     12
    }                                                                   13
}                                                                       14
```

```
C:\WINDOWS\system32\cmd.exe
C:>javac FiboDemo.java

C:>java FiboDemo
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,

C:>_
```

The following picture depicts the invocation of *fibo()* in finding **`fibo(4)`**. The numbers listed in solid circles indicate the order of method invocations and value returning.



Costs of Recursion

A recursive method accomplishes its task by successively calling itself. Therefore, there are many invocations of method involved. As we have discussed in Chapter 8, the mechanism of method invocation consists of steps such as copying values of the input variables to the local variables of the method, and copying the returned values from inside the method to the caller. These steps make method invocation relatively computationally expensive compared to evaluating expressions and looping through sets of statements using iterative constructs.

Furthermore, each time a recursive call is made, a certain amount of memory must be allocated. For a recursive method that makes very deep recursions, a large amount of memory is required.

Consider *fibo()* in the last example. We can see that to find **`fibo(4)`**, 8 method invocations are made, and the recursion goes 3 level-deep. For an input value of more than 30, there are more than 1 million method invocations made, and the depth can be more than 30 levels.

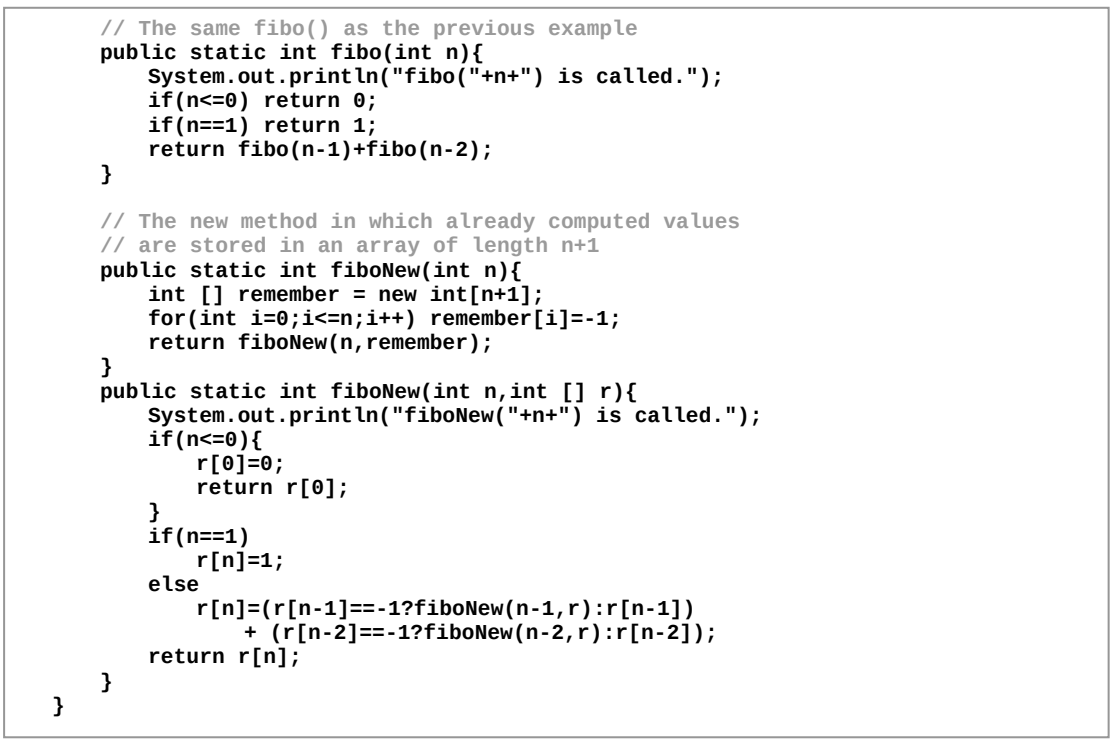
Does this mean we should avoid recursive algorithms? No, it does not. Sometimes, the easiest and the least error-prone ways to write programs for solving some problems are recursive methods. Sometimes, an iterative approach is much more difficult than the recursive ones. The examples that we have discussed so far might not serve as a good example to this claim. However, solving the "Towers of Hanoi" problem, presented in the last section, should present convincing example. Try solving it using an iterative approach.

Example: Fibonacci Numbers Revisited

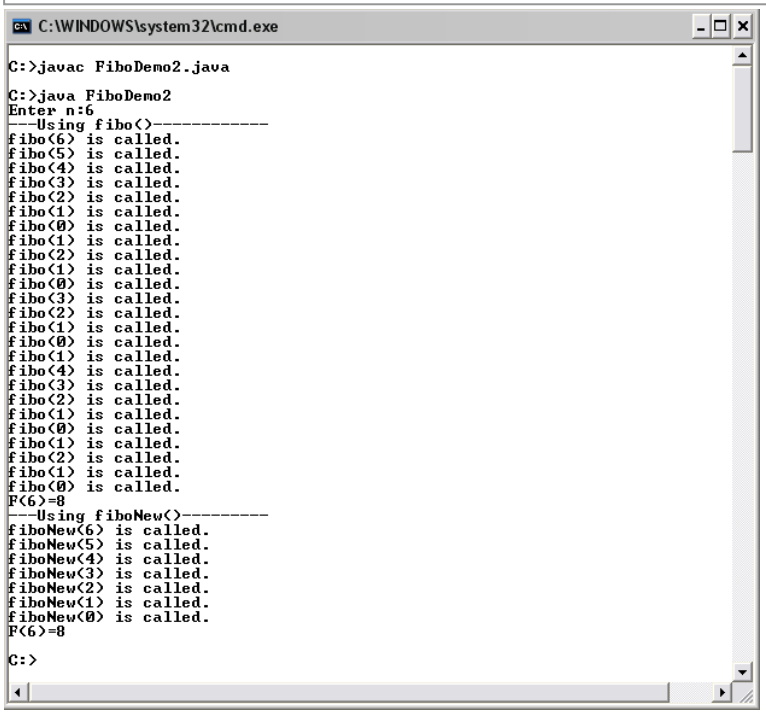
Let's revisit the method *fibo()* presented previously. Most of the time, calculating the n^{th} Fibonacci number involves redundant method invocations. For example, from the diagram showing method invocation in **`fibo(4)`**, we can see that **`fibo(2)`** and **`fibo(0)`** are called twice, while **`fibo(1)`** is called three times. An alternative implementation that should save some numbers of method invocation is to introduce a variable storing previously computed values of Fibonacci numbers. If desired Fibonacci number has been computed earlier, the program should read from the stored values instead of making a method call.

The following program computes the n^{th} Fibonacci number using the method fibo() which is the same as in the previous example, and the method *fiboNew()* in which an array of **`int`** is used for remembering the Fibonacci numbers that have already been computed. A *println()* statement is added into both *fibo()* and *fiboNew()*, so that it prints a message whenever the method is called. The number of message printed determines how many times each method is called. (To count how many times each method is called, we could introduce static variables used for storing the counts of method invocations. However, static variables have not been discussed until the next chapter.)

```
import java.io.*;
public class FiboDemo2
{
    public static void main(String[] args) throws IOException
    {   BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter n:");
        int n = Integer.parseInt(stdin.readLine());
        System.out.println("---Using fibo()-----------");
        System.out.println("F("+n+")="+fibo(n));
        System.out.println("---Using fiboNew()---------");
        System.out.println("F("+n+")="+fiboNew(n));
    }
    // continue on the next page
```



```
        // The same fibo() as the previous example
        public static int fibo(int n){
            System.out.println("fibo("+n+") is called.");
            if(n<=0) return 0;
            if(n==1) return 1;
            return fibo(n-1)+fibo(n-2);
        }

        // The new method in which already computed values
        // are stored in an array of length n+1
        public static int fiboNew(int n){
            int [] remember = new int[n+1];
            for(int i=0;i<=n;i++) remember[i]=-1;
            return fiboNew(n,remember);
        }
        public static int fiboNew(int n,int [] r){
            System.out.println("fiboNew("+n+") is called.");
            if(n<=0){
                r[0]=0;
                return r[0];
            }
            if(n==1)
                r[n]=1;
            else
                r[n]=(r[n-1]==-1?fiboNew(n-1,r):r[n-1])
                    + (r[n-2]==-1?fiboNew(n-2,r):r[n-2]);
            return r[n];
        }
    }
```

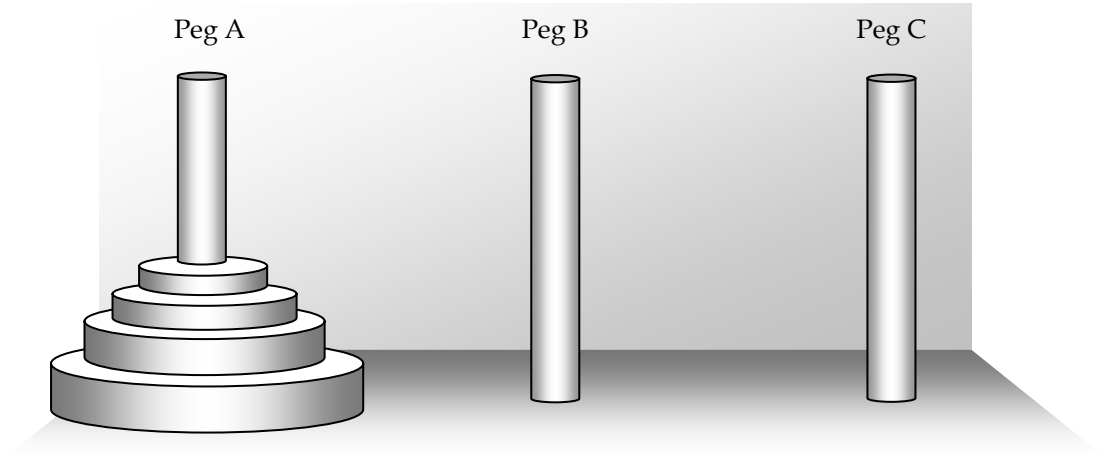


```
C:\WINDOWS\system32\cmd.exe

C:>javac FiboDemo2.java

C:>java FiboDemo2
Enter n:6
---Using fibo()------------
fibo(6) is called.
fibo(5) is called.
fibo(4) is called.
fibo(3) is called.
fibo(2) is called.
fibo(1) is called.
fibo(0) is called.
fibo(1) is called.
fibo(2) is called.
fibo(1) is called.
fibo(0) is called.
fibo(3) is called.
fibo(2) is called.
fibo(1) is called.
fibo(0) is called.
fibo(1) is called.
fibo(4) is called.
fibo(3) is called.
fibo(2) is called.
fibo(1) is called.
fibo(0) is called.
fibo(1) is called.
fibo(2) is called.
fibo(1) is called.
fibo(0) is called.
F(6)=8
---Using fiboNew()---------
fiboNew(6) is called.
fiboNew(5) is called.
fiboNew(4) is called.
fiboNew(3) is called.
fiboNew(2) is called.
fiboNew(1) is called.
fiboNew(0) is called.
F(6)=8

C:>
```

From the picture, we can see that finding the 6th Fibonacci number using *fibo()* requires more than three times as many method invocations as it is required in the case of using *fiboNew()*.

Example: The Towers of Hanoi

The Towers of Hanoi is a puzzle invented in the late nineteenth century by the French mathematician Édouard Lucas. The setting of this puzzle consists of three pegs mounted on a board together with disks of different sizes. Initially, these disks are placed on the first peg (peg A) in order of their sizes, with the largest one on the bottom and the smallest one on the



top, as shown in the picture below. The goal of this puzzle is to move all disks from their initial locations on Peg A to PegB. However, a valid move in this puzzle must obey two rules:

- Only one disk can be moved at a time, and this disk must be top disk on a tower.
- A larger disk cannot be placed on the top of a smaller disk.

Here, we would like to develop a program that finds the moves necessary to complete the puzzle with n disks, where n is a positive integer.

Let's try making the list of the necessary moves for some small values of n.

Starting from n = 1, the only move needed is to move the disk from Peg A to Peg B.

For n = 2, let's call the smaller disk Disk1 and the other disk Disk2. The moves needed are:

- Move Disk1 from Peg A to Peg C.
- Move Disk2 from Peg A to Peg B.
- Move Disk1 from Peg C to Peg C.

For n = 3, the steps needed to complete the puzzle start from performing the moves required to move the top two disk from Peg A to Peg C using a similar set of moves used in solving the puzzle with n=2, but from Peg A to Peg C instead of to the final destination, Peg B. Then, move the biggest disk from Peg A to Peg B. Finally, we can again use the set of moved used in moving two disks from one peg to another peg to move the two disks left on Peg C to Peg B. Therefore, the puzzle is solved for n = 3.

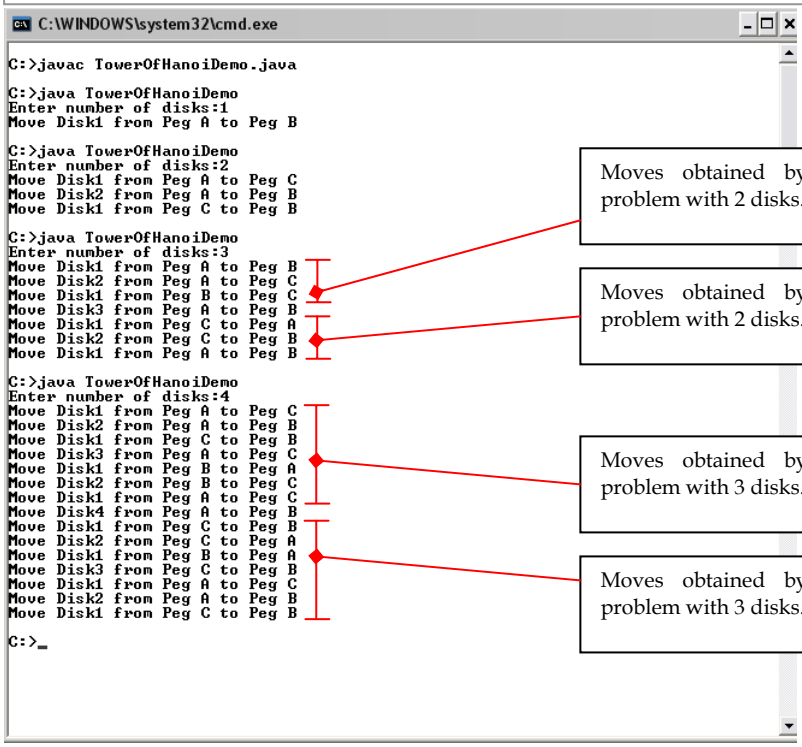
It is easy to notice that to solve the puzzle for any n, the move list starts from the moves required to move the top n-1 disks from one peg to another, which is from Peg A to Peg C, then a move that takes the largest disk from Peg A to Peg B, and, finally, another set of moves required to move n-1 disks from one peg to another which, this time, is from Peg C to Peg B. Therefore, for any n, the problem can be solved recursively, starting from the case where there is only one disk.

Naming the disks with Disk1, Disk2, to Diskn, the following Java program, making use of a recursive method, produces the required move list in the following format:

```
Move [disk] from [origin peg] to [destination peg].
```

Here is the program listing.

```
import java.io.*;
public class TowerOfHanoiDemo
{
    public static void main(String[] args) throws IOException
    {   BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter number of disks:");
        int n = Integer.parseInt(stdin.readLine());
        move(n,"A","B","C");
    }
    public static void move(int n,
        String orgPole,String destPole,String otherPole){
        String step;
        if(n<=1){
            step = "Move Disk1 from Peg "+orgPole+" to Peg "+destPole;
            System.out.println(step);
        }else{
            move(n-1,orgPole,otherPole,destPole);
            step = "Move Disk"+n+" from Peg "+orgPole+" to Peg "+destPole;
            System.out.println(step);
            move(n-1,otherPole,destPole,orgPole);
        }
    }
}
```



Exercise

1. What is the output when *main()* is executed?

```
public static void main(String[] args)
{
    System.out.println(f(5));
}
public static int f(int n){
    if(n<=0) return 1;
    return f(n-1)+2*f(n-2);
}
```

2. What is the output when *main()* is executed?

```
public static void main(String[] args)
{
    System.out.println(g(4));
}

public static int g(int n)
{
    if(n<=0) return 1;
    return 2*g(n-1)+ f(n+1);
}

public static int f(int n)
{
    if(n<=0) return 1;
    if(n==1) return 2;
    return f(n-1)-f(n-2);
}
```

3. Implement the following method using a recursive approach.

```
public static int f(int n)
{
    int a = 0;
    for(int i=1;i<=n;i++){
        a += 2*i;
    }
    return a;
}
```

4. Implement the following method using a recursive approach.

```
public static int f(int n)
{
    int a = 1;
    int b = 1;
    for(int i=1;i<=n;i++){
        a += ++b;
        b = a;
    }
    return a;
}
```

5. Implement the following method using a recursive approach.

```
public static int f(int n)
{
    int a=0,b=1,c=0;
    if(n==1) return b;
    for(int i=2;i<=n;i++){
        a = b+c;
        c = b;
        b = a;
    }
    return a;
}
```

6. Implement the following method using a recursive approach.

```
public static double f(int n)
{
    double a=2.0,b=2.0,c=2.0;
    if(n==1) return b;
    for(int i=2;i<=n;i++){
        a = b+0.5*c+i;
        c = b;
        b = a;
    }
    return a;
}
```

7. Write a Java method for calculating the following function at a given non-negative integer n. Based on your implementation, plot the number of method invocation made as a function of n ranging from 1 to 10.

$$f(n) = \begin{cases} 1 & \text{if } n = 0,1,2 \\ f(n-2) + n \times f(n-3) & \text{if } n = 3,4,... \end{cases}$$

8. Write a Java method for calculating the following function at a given non-negative integer n.

$$f(n) = \begin{cases} \sum_{k=0}^{n} k & ;n = 0,1,2 \\ \frac{1}{2} f(n-1) + \frac{1}{2} f(n-3) & ;3 \le n < 10 \\ \sum_{k=n-3}^{n-1} (f(k) - (-1)^k f(k-1)) & ;n \ge 10 \end{cases}$$

9. Write a recursive Java method that calculates the sum of every **`int`** element in an input array of **`int`**.

10. Write a recursive Java method that finds the smallest value in an input array of **`int`**.

11. Write a recursive Java method that returns the index of the smallest value in an input array of **`int`**.

12. The mathematical constant e is the base of the natural logarithm. It is called *Euler's number* or *Napier's constant*. This constant is the sum of the infinite series defined below.

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ...$$

Write a Java program to approximate the value of e. Utilize the recursive approach as appropriate. Note that 0! equals 1. Also, you may use the following method to find the value of n!.

```
public static double factorial(int n)
{
    if(n<=0) return 1;
    return n*factorial(n-1);
}
```

13. Write a recursive method that receives an **`int`** value as its input, and prints the associated digits on screen one digit on each line, starting from the left. You are not allowed to directly convert the input **`int`** value to *String*. Assume that the input value is always positive.

14. The greatest common divisor (gcd) of two integers is the largest integer that divides both of them. For example, the gcd of 74 and 111 is 37.

    a. Write a method that finds the gcd of two input integers using an iterative approach.
    b. Repeat part a. using an recursive approach based on:

$$\gcd(a,b) = \begin{cases} b & \text{if } a\%b = 0 \\ \gcd(b, a\%b) & \text{otherwise} \end{cases}$$

15. A *palindrome* is a word, phrase, number or other sequence of units (such as a strand of DNA) that has the property of reading the same in either direction where the punctuations and spaces are generally ignored. For example, "civic", "level", "Was it a cat I saw?", and "A man, a plan, a canal: Panama" are palindromes.

    Write a recursive Java method that returns **`true`** if the *String* passed to the method is a palindrome. Otherwise, it returns **`false`**.