CSCI 2110 Data Structures and Algorithms

Module 5: Recursion

DALHOUSIE UNIVERSITY

Srini Sampalli

What is recursion?

In most programs that we have developed so far, we have seen methods calling other methods.

Suppose we write a program in which a method calls itself.

Such a method is referred to as a **recursive method** and the process is called **recursion.**

RECURSION (in programming) = A METHOD CALLING ITSELF

Recursive Definition

A recursive definition is a self-referential definition – it defines an entity by means of the entity itself.

Example 1:

- ***Onion (non-recursive definition) – vegetable with concentric leaf bases.***
- ***Onion (recursive definition) – consists of a leaf base enclosing an onion.***

Example 2:

- ***Canadian Citizen: A person born in Canada or who has acquired citizenship by naturalization, or whose parent is a Canadian citizen.***

Recursive Humour

- Definition: ***Recursion***

  See "Recursion"

- Another definition: ***Recursion***

  If you still don't get it, see "Recursion"

- Yet another:

  To understand recursion, you must understand recursion.

- Google search for Recursion:

  Did you mean "Recursion"?

- An instruction manual on how to use the instruction manual.

Visual Recursion: Droste Effect

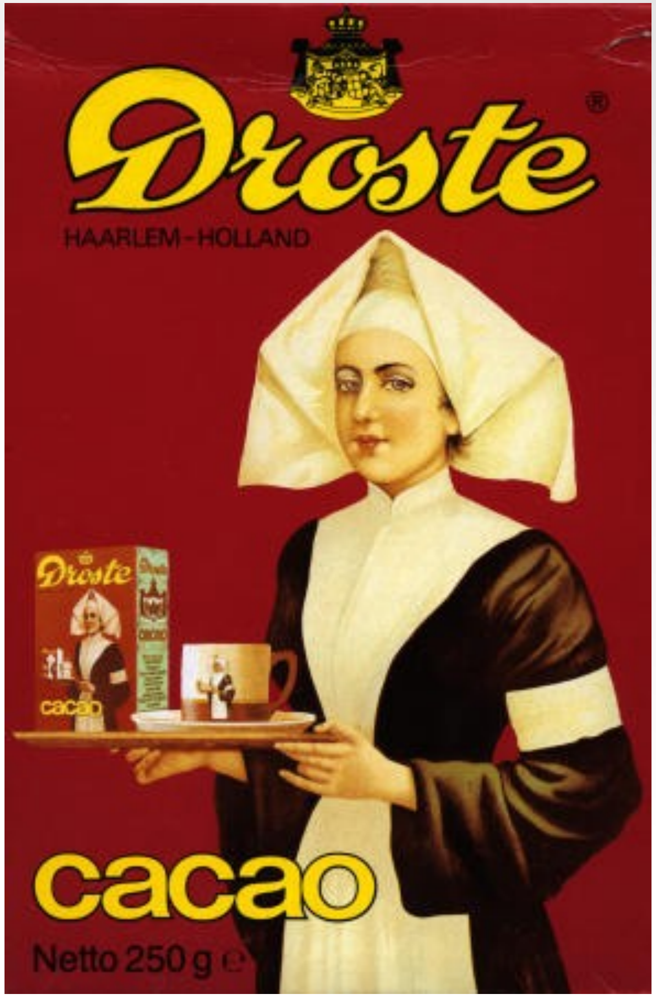


Image source: Wikipedia

Example 3: Linked List

- Non-recursive definition: A linked list consists of a set of nodes in which each node contains a reference to the next node.
- Recursive definition:

  A linked list is either empty,

  or consists of a node that contains a reference to a linked list.

Example 4: Unordered List

- Non-recursive definition: An unordered list is a sequential collection of items.
- Recursive definition:

  An unordered list is either empty,

  or consists of a first item followed by an unordered list.

Example 5: Ordered List

- Non-recursive definition: A sequential collection of items which are sorted in increasing key values.
- Recursive definition: An ordered list is either empty,

  or consists of a first entry followed by an ordered list,

  and the value of the key in the first item is less than the values of all the keys in the following ordered list.

Why recursion?

- Recursion is a powerful problem-solving paradigm that can be applied to many data structures.
- It drastically reduces the number of lines of code in a program.
- Recursion helps to solve large and unwieldy problems in terms of smaller and easily solvable problems.
- For many problems, recursion may be the only **optimal** solution.
- Recursion is also very useful in animation, graphics, gaming and wall-paper design.

Writing Recursive Programs

- A recursive program implements the recursive definition.
- A recursive program is implemented using recursive methods. As we mentioned before, a recursive method is one that calls itself.

Recursion Example 0:

```
//Endless.java This class has a recursive method.
public class Endless{
	public static void message(){
		System.out.println(“This is a recursive method”);
		message();
	}
	public static void main(String[] args){
		message();
	}
}
```

The method message() displays “This is a recursive method” and then calls itself.

Obviously, this recursive method goes into an infinite loop because the code doesn’t stop it from repeating.

Recursion Example 1:

Suppose that we have some way to control the number of times the method repeats.

Let us pass an integer argument into the method and decrement it inside the method.

```
//This class has a recursive method, which displays a message n times
public class Endless{
        public static void message(int n){
                if (n==0)
                        return;
                else{
                        System.out.println("This is a recursive method");
                        message(n-1);
                }
        }
        public static void main(String[] args){
                message(5);
        }
}
```

→ exit strategy a.k.a. base condition

→ glue condition

Recursion Example 1:

This program will display "This is a recursive method" five times.

We have been able to control the recursive method just like controlling the number of iterations in a loop.

Program trace of Example 1:

```
Call message(5) →
        Display "This is a recursive method"
        Call message(4) →
                Display "This is a recursive method"
                Call message(3) →
                        Display "This is a recursive method"
                        Call message(2) →
                                Display "This is a recursive method"
                                Call message(1) →
                                        Display "This is a recursive method"
                                        Call message(0) →
                                                Exit (back to main)
```

Number of calls to the method = Depth of recursion

Writing recursive methods (revisited)

- Every recursive method must have two conditions:
  - Base condition: a condition at which the method does not recurse anymore (exit condition)
  - Glue condition: a condition that is self-referential (method calls another version of itself)

  Note: there could be more than one base and glue conditions.

- The main work in writing a recursive method is to produce a clear recursive definition with a base condition and a glue condition.
- A recursive method keeps calling itself until it hits the base condition, at which point, it exits the method.

Recursion Example 2:

What will this program display?

```
public class Recursion2{
        public static void cheers(int n){
                if (n==1)
                        System.out.println(“Hurray!”);
                else{
                        System.out.print(“Hip ”);
                        cheers(n-1);
                }
        }
        public static void main(String[] args){
                cheers(4);
        }
}
```

$$\begin{tikzcd}
\text{cheers}(4) \arrow[dr] & & & \\
& \text{Cheers}(3) \arrow[dr] & & \\
& & \text{Cheers}(2) \arrow[d] & \\
& & \text{Cheers}(1) &
\end{tikzcd}$$

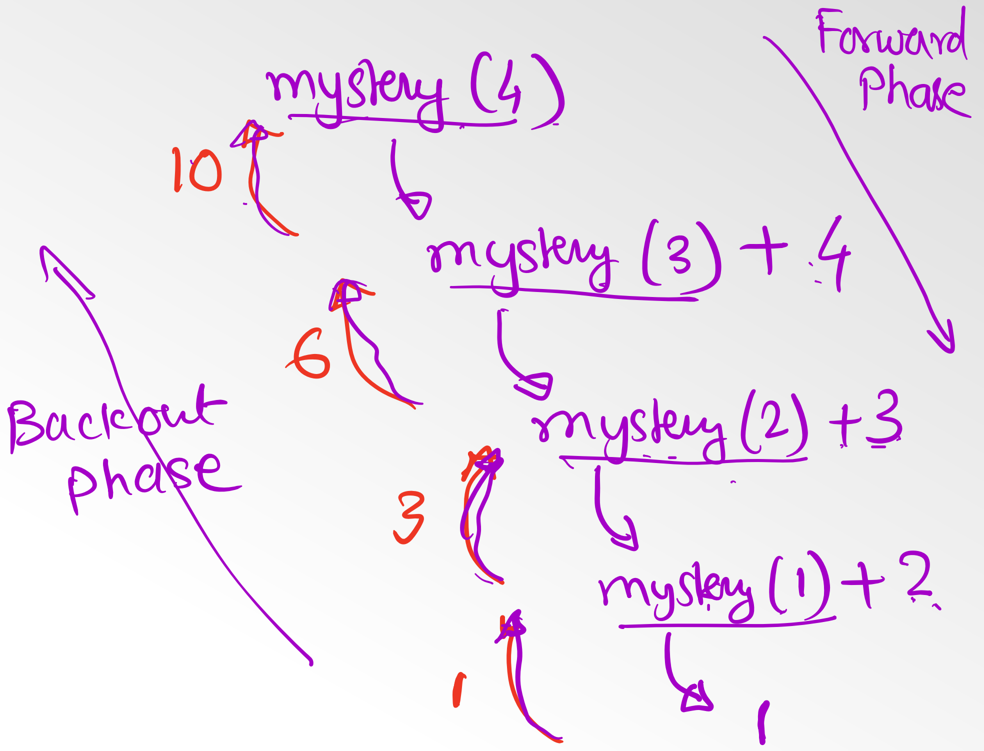
$\boxed{\text{Hip Hip Hip Hurray!}}$

Recursion Example 3:

What will this program display?

```
public class Recursion3{
        public static int mystery(int n){
                if (n<=1)
                        return 1;
                else
                        return (mystery(n-1)+n);
        }
        public static void main(String[] args){
                System.out.println(mystery(4));
        }
}
```

$\boxed{10}$



Method calls are stored in a stack

Example 4: Factorial of an integer

$$4! = 4 \times \underbrace{3 \times 2 \times 1}$$
$$= 4 \times 3!$$

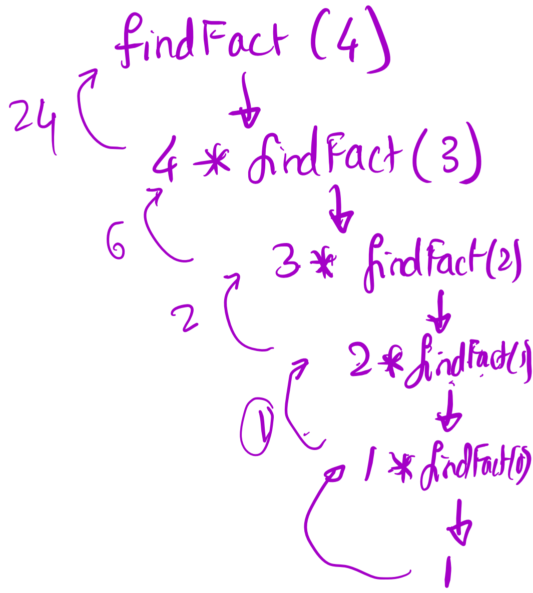
n! = 1 X 2 X 3 X …. X n
0! = 1

→ $n! = 1$ if n is 0
$n! = n * (n-1)!$ if $n > 0$

Recursive method to find factorial

```
public static int findFactorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * findFactorial(n-1);
}
```

Call to the method and visualization



Non-recursive or iterative method

```
public static int findFactorial(int n)
{
    int result = 1;
    while (n > 0)
    {
        result = result * n;
        n = n-1;
    }
    return result;
}
```

Example 5: Finding the nth number in the Fibonacci sequence

$$\boxed{\begin{matrix} \text{fibo}(0) = 0 \\ \text{fibo}(1) = 1 \end{matrix} \Big\} \text{ Exit} \quad \text{fibo}(n) = \text{fibo}(n-1) + \text{fibo}(n-2)}$$

Fibonacci series: $\underset{\text{0th number}}{0}$, $\underset{\text{1st number}}{1}$, 1, 2, 3, 5, 8, 13, 21, ….

Find the nth Fibonacci number

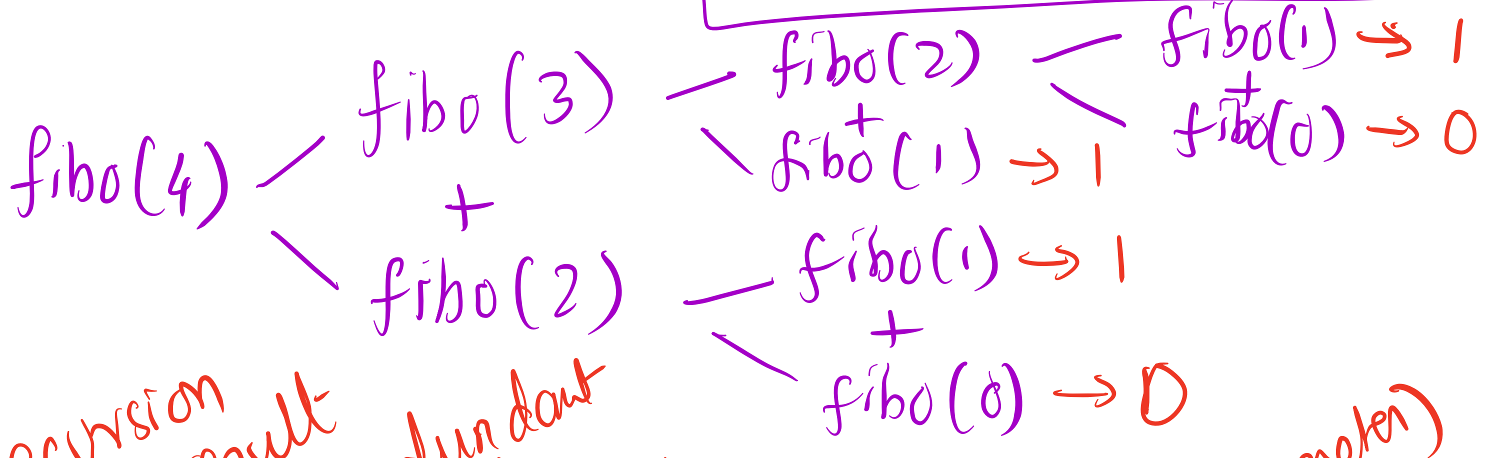
Recursive method to find fibo(n)

```
public static int fibo(int n) {
    if (n==0)
        return 0;
    else if (n==1)
        return 1;
    else
        return fibo(n-1) +
               fibo(n-2);
}
```

Iterative method to find fibo(n)

```
public static int fibo(int n){
if (n==0) return 0;
if (n==1) return 1;
int a = 0, b = 1, count = 2,
        sum = 0;
while (count <= n)
{
    sum = a + b;
    a = b;
    b = sum;
}   count++;
} return sum;
```

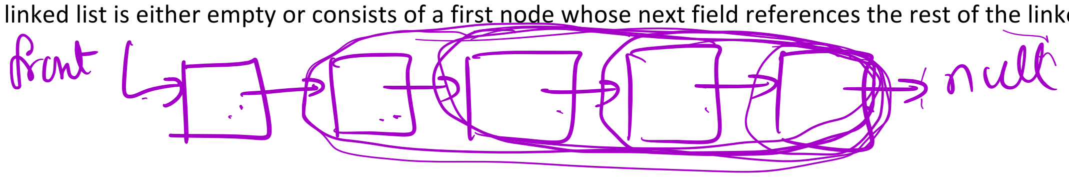
Call to the method and visualization



Recursion may result in redundant calls (multiple calls to the method for the same parameter)

Example 6: Linked List operations using recursion

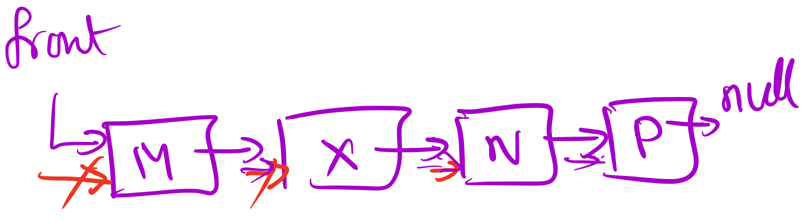
A linked list is either empty or consists of a first node whose next field references the rest of the linked list.



Count the number of nodes in a linked list

1 + 1 + 1 + 1 + 1 + 0

```
public int countNodes(Node<T> front) {
    if (front == null)
            return 0;
    else
        return 1 + countNodes(front.getNext());
}
```



Print the contents of the list (forwards)

M X N P

```
public void printForwards(Node<T> front) {
    if (front == null)
            return;
    else {  System.out.print(front.getData() + "\t");
            printForwards(front.getNext());
         }
}
```

Print the contents of the list (reverse)

P N X M

```
public void  printBackwards(Node<T> front) {
    if (front == null)
            return;
    else
      {
        printBackwards(front.getNext()); ←
        System.out.print (front.getData() + "\t");
      }
}
```

$$\begin{tikzcd}3 \arrow[d] & 4 \arrow[d] \\ a & b\end{tikzcd}$$

$$\begin{tikzcd}3^4 \arrow[r] & 3 * 3^3 \arrow[d] \\ & 3*3^2 \arrow[d] \\ & 3*3^1 \arrow[d] \\ & 3*3^0 \arrow[d] \\ & 1\end{tikzcd}$$

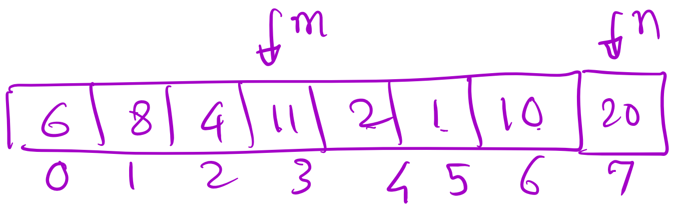
EXAMPLE 7: Computing a to the power of b with recursion

```
// Recursion Example 7
import java.util.Scanner;
public class Recursion7
{
        public static int power(int a, int b)
        {
            if (b==0)
                    return 1;
            else
                    return a * power(a, b-1);
        }
        public static void main(String[] args)
        {
                Scanner keyboard = new Scanner(System.in);
                System.out.println("Computation of a to the power of b");
                System.out.print("Enter positive integers a and b: ");
                int a = keyboard.nextInt();
                int b = keyboard.nextInt();

                System.out.println(a + " to the power " + b + " is " + power(a,b));
        }
}
```

Example 8: Summing a range of array elements

Suppose we have an array of integers

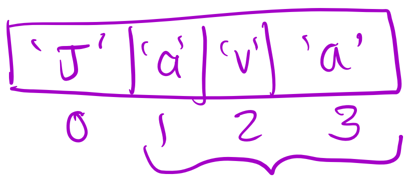


```
int [] numbers = {6, 8, 4, 11, 2, 1, 10, 20}
int sum = rangeSum(numbers, 3, 7); → sum should be 11+2+1+10+20 = 44
```

Assumption: m & n are within bounds, & m<=n

```
public static int rangeSum(int[] a, int m, int n)
{
    if (m==n)
            return a[m];
    else
            return a[m] + rangeSum(a, m+1, n);
}
```

Example 9: Count the number of occurrences of a given character ch in a String s. For example, count("Java", 'a') should return 2.



$$\begin{tikzcd}
 & \text{Check char at index 0} \arrow[dl] \arrow[dr] & \\
\text{add to count} & & \text{do not add to count}
\end{tikzcd}$$

Then call the method on the substring starting at index 1.

```
import java.util.Scanner;
public class Recursion10
{
        public static int countChar(String s, char ch)
        {
            if (s.length() == 0)
                        return 0;
            else
            {
                if (s.charAt(0) == ch)
                    return 1 + countChar(s.substring(1), ch);
                else
                    return 0 + countChar(s.substring(1), ch);
            }

        }
        public static void main(String[] args)
        {
                Scanner keyboard = new Scanner(System.in);
                System.out.print("Enter a line: ");
                String line = keyboard.nextLine();
                System.out.print("Enter a character: ");
                String ch = keyboard.nextLine();
                char c = ch.charAt(0);
                System.out.println(ch + " appears in " + line + " " + countChar(line, c) + " times");

        }
}
```

Example 10: Binary Search Algorithm

Inputs: Array of ints a, int key, int lo, int hi

Pseudo code:

```
mid ← (lo+hi)/2
✓ if (key == a[mid]) return mid;
  if (key < a[mid])
      call binarySearch on left half
  if (key > a[mid])
      call binarySearch on right half
✓ if (lo > hi) return -1;
```

```
public class RecBinarySearch
{
    private static int binarySearch(int[] a, int key, int lo, int hi)
        {

            int mid = (lo+hi)/2;
            if (lo > hi)
                  return -1;
            else if (key == a[mid])
                  return mid;
            else if (key < a[mid])
                  return binarySearch(a, key, lo, mid-1);
            else
                  return binarySearch(a, key, mid+1, hi);

        }
        public static int binarySearch(int[ ] a, int key)
        {
                return binarySearch(a, key, 0, a.length - 1);
        }

        public static void main(String[] args)
        {
            int[] myArray = {10, 20, 30, 40, 50, 60, 70};
            System.out.println(binarySearch(myArray,30));
            System.out.println(binarySearch(myArray,100));

        }
}
```

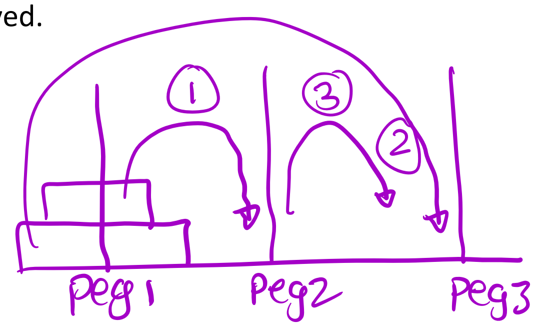
wrapper method

Example 11: TOWERS OF HANOI

The Towers of Hanoi game uses three pegs and a set of discs that are stacked on one of the pegs. The discs must be moved from the first peg to the third peg following these rules:

- Only one disk can be moved at a time.
- A disk cannot be placed on top of a smaller disc.
- All discs must be stored on a peg except while being moved.

Solution: Number of discs = 2
Move disc 1 to peg 2
Move disc 2 to peg 3
Move disc 1 to peg 3



Solution: Number of discs = 3

Move 2 discs from peg 1 to peg 2 with peg 3 as the temporary peg.
Move the third disc to peg 3.
Move 2 discs from peg 2 to peg 3 with peg 1 as the temporary peg.

Move disc 1 to peg 3
Move disc 2 to peg 2
Move disc 1 to peg 2
Move disc 3 to peg 2
Move disc 1 to peg 2
Move disc 2 to peg 3
Move disc 1 to peg 3

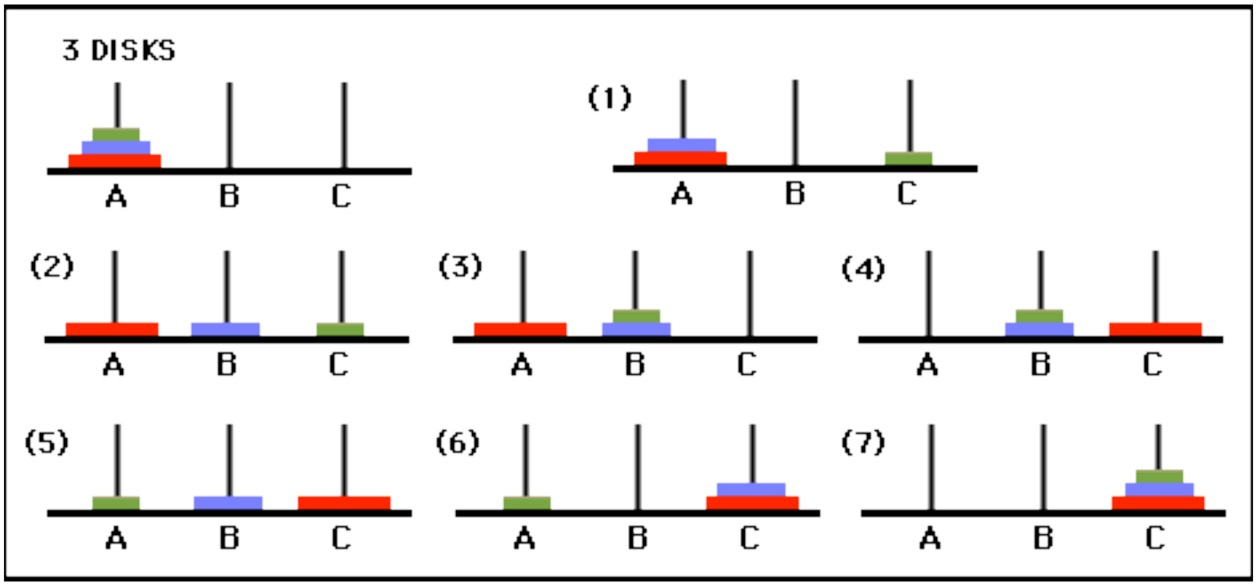
The complexity increases as the number of discs increases.

The following statement describes the overall solution to the problem:

Move n discs from peg 1 to peg 3 using peg 2 as the temporary peg.

The following algorithm can be used as the basis for a recursive method.

```
To move n discs from peg 1 to peg 3, using peg 2 as the temporary peg:
if n > 0 then
    Move n-1 discs from peg 1 to peg 2, using peg 3 as the temporary peg.   ← step 1
    Move the remaining disc from peg 1 to peg 3                             ← step 2
    Move n-1 discs from peg 2 to peg 3, using peg 1 as the temporary peg.   ← step 3
```



TOWER OF HANOI – SOLUTION FOR n=3

```
import java.util.Scanner;
public class Hanoi{
        public static void moveDiscs(int n, int from, int to, int temp)
        {
            if (n > 0)
            { moveDiscs(n-1, from, temp, to);
              System.out.println ("Move disc " + n +
            " from peg " + from + " to peg" + to);
              moveDiscs(n-1, temp, to, from);
            }

        }
        public static void main(String[] args){
                Scanner keyboard = new Scanner(System.in);
                System.out.print("Enter the number of discs: ");
                int numDiscs = keyboard.nextInt();
                moveDiscs(numDiscs, 1, 3, 2);
        }
}
                               from  to   temp
```