Review of Recursion

- What is a Recursive Method?
- The need for Auxiliary (or Helper) Methods
- How Recursive Methods work
- Tracing of Recursive Methods

What is a Recursive Method?

- A method is recursive if it calls itself either directly or indirectly.
- Recursion is a technique that allows us to break down a problem into one or more simpler sub-problems that are similar in form to the original problem.
- Example 1: A recursive method for computing x!

```
long factorial (int x) {
   if (x == 0)
      return 1;      //base case
   else
      return x * factorial (x – 1);  //recursive case
}
```

- This method illustrates all the important ideas of recursion:
  - A base (or stopping) case
    - Code first tests for stopping condition (is x = = 0 ?)
    - Provides a direct (non-recursive) solution for the base case (0! = 1).
  - The recursive case
    - Expresses solution to problem in 2 (or more) smaller parts
    - Invokes itself to compute the smaller parts, eventually reaching the base case

What is a Recursive Method?

- Example 2: count zeros in an array

```
int countZeros(int[] x, int index) {
    if (index == 0)
            return x[0] == 0 ? 1: 0;
    else if (x[index] == 0)
        return 1 + countZeros(x, index - 1);
    else
        return countZeros(x, index - 1);
}
```

The need for Auxiliary (or Helper) Methods

- Auxiliary or helper methods are used for one or more of the following reasons:
  - To make recursive methods more efficient.
  - To make the user interface to a method simpler by hiding the method's initializations.
- Example 1: Consider the method:

```
public long factorial (int x){
  if (x < 0)
       throw new IllegalArgumentException("Negative argument"
  else if (x == 0)
        return 1;
  else
       return x * factorial(x - 1);
}
```

- The condition x < 0 which should be executed only once is executed in each recursive call. We can use a private auxiliary method to avoid this:

The need for Auxiliary (or Helper) Methods

```
public long factorial(int x){
   if (x < 0)
      throw new IllegalArgumentException("Negative argument");
   else
      return factorialAuxiliary(x);
}

private long factorialAuxiliary(int x){
   if (x == 0)
      return 1;
   else
      return x * factorialAuxiliary(x - 1);
}
```

The need for Auxiliary (or Helper) Methods

- Example 2: Consider the method:

```
public int binarySearch(int target, int[] array, int low, int high) {
   if(low > high)
      return -1;
   else {
      int middle = (low + high)/2;
      if(array[middle] == target)
         return middle;
      else if(array[middle] < target)
         return binarySearch(target, array, middle + 1, high);
      else
         return binarySearch(target, array, low, middle - 1);
   }
}
```

- The first time the method is called, the parameter low and high must be set to 0 and array.length – 1 respectively. Example:

```
int result = binarySearch (target, array, 0, array.length -1);
```

- From a user's perspective, the parameters *low* and *high* introduce an unnecessary complexity that can be avoided by using an auxiliary method:

The need for Auxiliary (or Helper) Methods

```
public int binarySearch(int target, int[] array){
   return binarySearch(target, array, 0, array.length - 1);
}

private int binarySearch(int target, int[] array, int low, int high){
   if(low > high)
      return -1;
   else{
      int middle = (low + high)/2;
      if(array[middle] == target)
         return middle;
      else if(array[middle] < target)
         return binarySearch(target, array, middle + 1, high);
      else
         return binarySearch(target, array, low, middle - 1);
   }
}
```

- A call to the method becomes simple:

```
int result = binarySearch(target, array);
```

The need for Auxiliary (or Helper) Methods

- Example 3: Consider the following method that returns the length of a MyLinkedList instance:

```
public int length(Element element){
   if(element == null)
      return 0;
   else
      return 1 + length(element.next);
}
```

- The method must be invoked by a call of the form:

```
list.length(list.getHead());
```

- By using an auxiliary method, we can simplify the call to:

```
list.length();
```

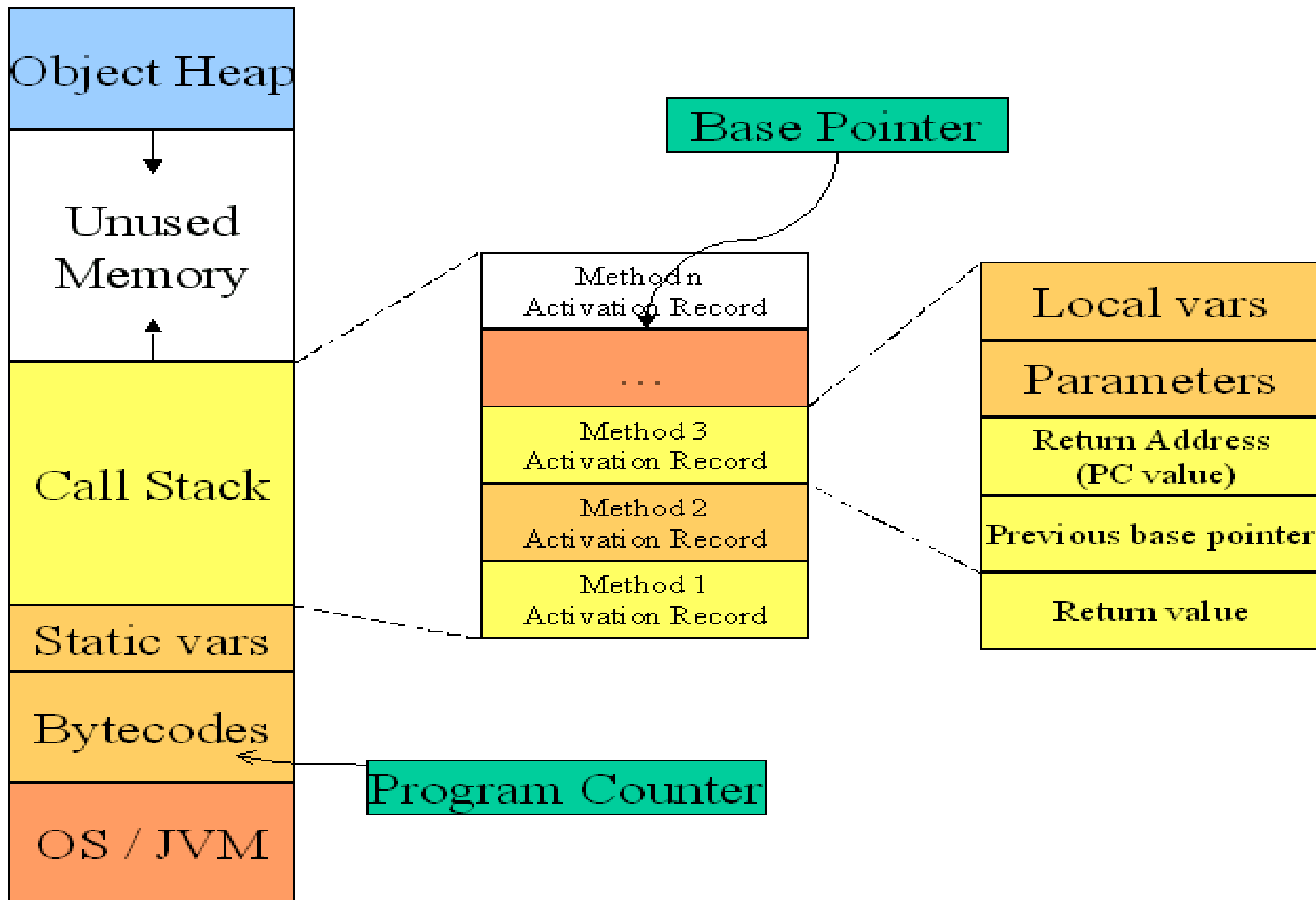
The need for Auxiliary (or Helper) Methods

```
public int length(){
   return auxLength(head);
}

private int auxLength(Element element){
   if(element == null)
      return 0;
   else
      return 1 + auxLength(element.next);
}
```

How Recursive Methods work

- Modern computer uses a stack as the primary memory management model for a running program.
- Each running program has its own memory allocation containing the typical layout as shown below.



How Recursive Methods work

- When a method is called an Activation Record is created. It contains:
  - The values of the parameters.
  - The values of the local variables.
  - The return address (The address of the statement after the call statement).
  - The previous activation record address.
  - A location for the return value of the activation record.
- When a method returns:
  - The return value of its activation record is passed to the previous activation record or it is passed to the calling statement if there is no previous activation record.
  - The Activation Record is popped entirely from the stack.
- Recursion is handled in a similar way. Each recursive call creates a separate Activation Record" As each recursive call completes, its Activation Record is popped from the stack. Ultimately control passes back to the calling statement.

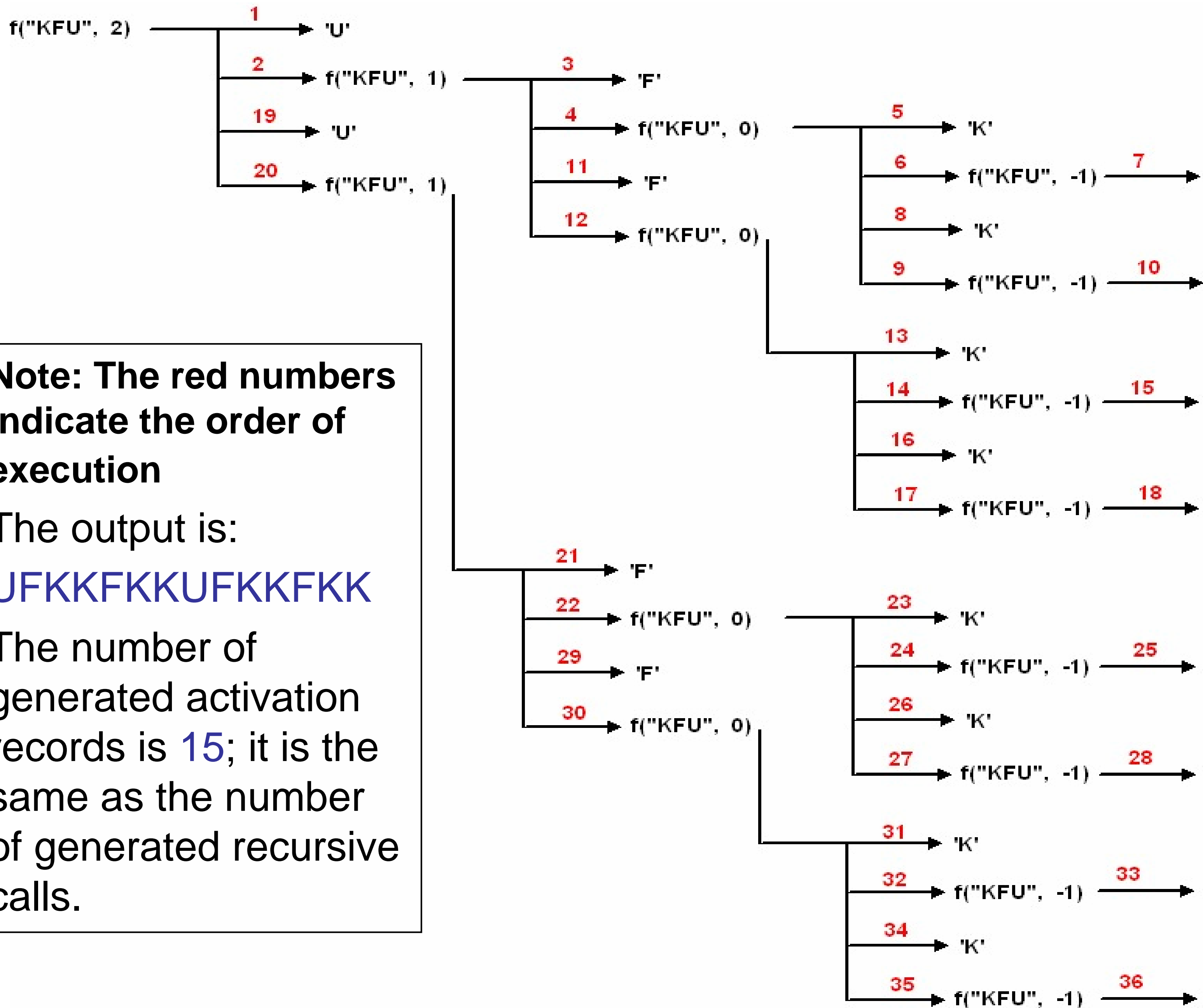
Tracing of Recursive Methods

- A recursive method may be traced using the recursion tree it generates.
- Example1: Consider the recursive method f defined below. Draw the recursive tree generated by the call f("KFU", 2) and hence determine the number of activation records generated by the call and the output of the following program:

```
public class MyRecursion3 {
   public static void main(String[] args){
      f("KFU", 2);
   }

   public static void f(String s, int index){
      if (index >= 0) {
         System.out.print(s.charAt(index));
         f(s, index - 1);
         System.out.print(s.charAt(index));
         f(s, index - 1);
      }
   }
}
```

Tracing of Recursive Methods



- **Note: The red numbers indicate the order of execution**
- The output is: UFKKFKKUFKKFKK
- The number of generated activation records is 15; it is the same as the number of generated recursive calls.

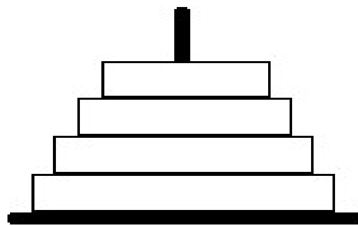
Tracing of Recursive Methods

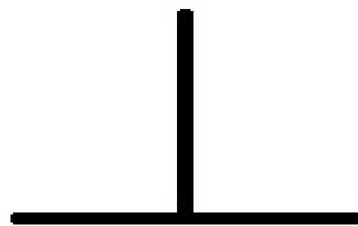
- Example2: The Towers of Hanoi problem:
- A total of n disks are arranged on a peg A from the largest to the smallest; such that the smallest is at the top. Two empty pegs B and C are provided.
- It is required to move the **n** disks from peg A to peg C under the following restrictions:
  - Only one disk may be moved at a time.
  - A larger disk must not be placed on a smaller disk.
  - In the process, any of the three pegs may be used as temporary storage.
- Suppose we can solve the problem for **n – 1** disks. Then to solve for **n** disks use the following algorithm:

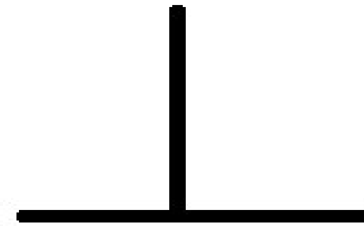
  Move **n – 1** disks from peg A to peg B

  Move the nth disk from peg A to peg C

  Move **n – 1** disks from peg B to peg C







Tracing of Recursive Methods

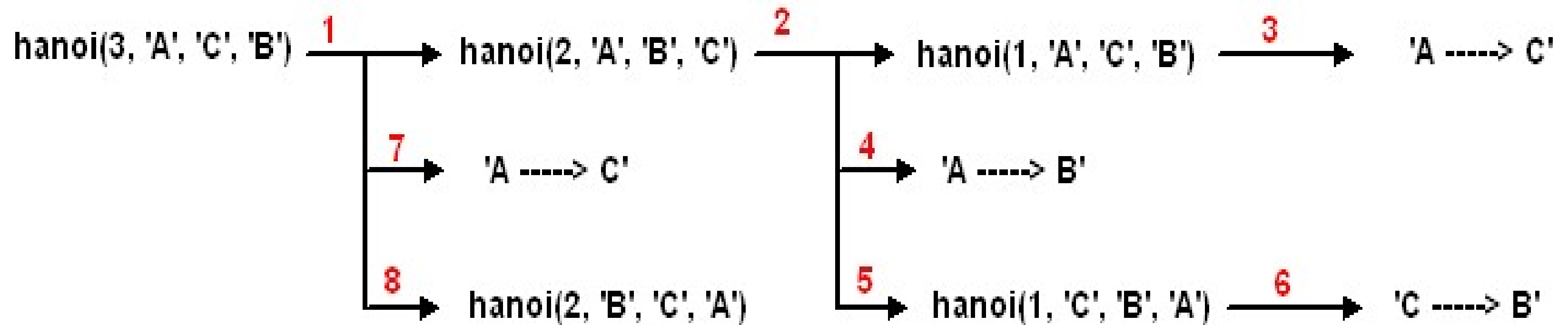
- This translates to the Java method hanoi given below:

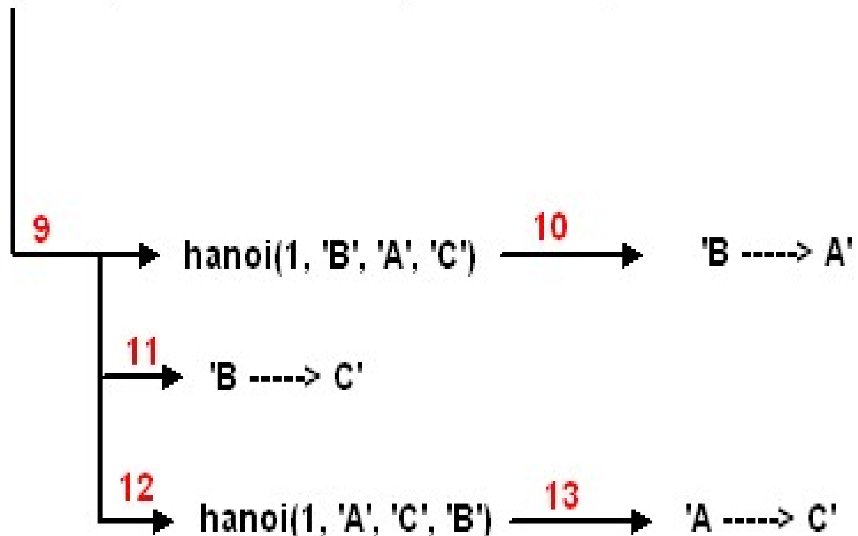
```
import java.io.*;
public class TowersOfHanoi{
   public static void main(String[] args) throws IOException {
      BufferedReader stdin =
           new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Enter the value of n: " );
      int n = Integer.parseInt(stdin.readLine());
      hanoi(n, 'A', 'C', 'B');
   }

   public static void hanoi(int n, char from, char to, char temp){
      if (n == 1)
         System.out.println(from + " --------> " + to);
      else{
         hanoi(n - 1, from, temp, to);
         System.out.println(from + " --------> " + to);
         hanoi(n - 1, temp, to, from);
      }
   }
}
```

Tracing of Recursive Methods

- Draw the recursion tree of the method hanoi for n = = 3 and hence determine the output of the above program.





output of the program is:

A -------> C
A -------> B
C -------> B
A -------> C
B -------> A
B -------> C
A -------> C