Chapter 5

Recursion

Basically, a method is *recursive* if it includes a call to itself.

```
if simplest case
    Solve directly
else
    Make a recursive call to a simpler case
```

Consider recursion when

1. **Simplest case(s) can be solved directly;**
2. **Complex cases can be solved in terms of simpler cases of the same form.**

Example 1: Factorials

Given a non-negative integer *n*,

***n*!, Read "*n* factorial", is the product of all integers between *n* and 1, inclusive.**

3! = 3 * 2 * 1 = 6

5! = 5 * 4 * 3 * 2 * 1 = 120

1! = 1

0! = 1 // by definition

For *n* > 1, we can calculate *n*! in terms of (*n* – 1)!

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 ← **calculate directly**

We can then work back up:

2! = 2 * 1! = 2 * 1 = 2

3! = 3 * 2! = 3 * 2 = 6

4! = 4 * 3! = 4 * 6 = 24

5! = 5 * 4! = 5 * 24 = 120

```
/**
 *  Returns the factorial of an integer. The worstTime(n)
 *  is O(n).
 *
 *  @param n – the integer whose factorial is returned.
 *  @return n!
 *  @throws IllegalArgumentException – if n is less than 0.
 *
 */
public static long factorial (int n)
{
      if (n < 0)
            throw new IllegalArgumentException( );
      if (n <= 1)
            return 1;
      return n * factorial (n - 1);
} // method factorial
```

Execution frames allow you to see what happens during the execution of a recursive method.

Each frame is a box with information (such as parameter values) related to one call to the method.

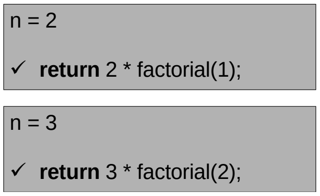
For example, here are the execution frames generated, step-by-step, after an initial call of factorial (3)**. At each step, the top frame pertains to the call being executed.**

Step 0:

n = 3

✓ **return** 3 * factorial(2);

Step 1:



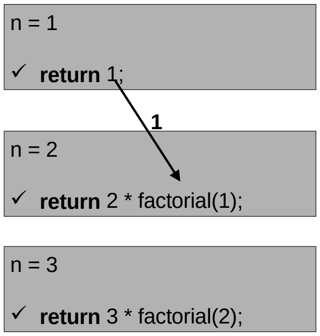
n = 2

✓ **return** 2 * factorial(1);

n = 3

✓ **return** 3 * factorial(2);

Step 2:



n = 1

✓ **return** 1;

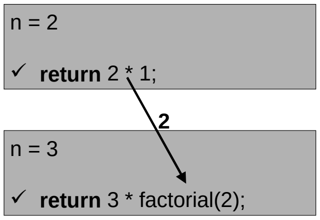
n = 2

✓ **return** 2 * factorial(1);

n = 3

✓ **return** 3 * factorial(2);

Step 3:



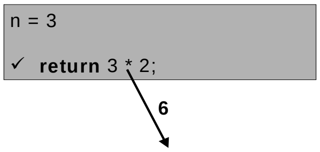
n = 2

✓ **return** 2 * 1;

n = 3

✓ **return** 3 * factorial(2);

Step 4:



n = 3

✓ **return** 3 * 2;

Analysis:

The key to estimating execution time and space of the factorial **method is the number of recursive calls to** factorial.

Number of recursive calls

= Maximum height of execution frames – 1

= *n – 1*

so worstTime(*n*) is linear in *n*.

In general, worstTime(*n*) depends on only two things:

1. **The number of loop iterations as a function of *n*;**
2. **The number of recursive calls as a function of *n*.**

In each call to factorial**, some information must be saved (return address, value of *n*).**

Because there are *n* – 1 recursive calls, the number of variables needed in any trace of this method is linear in *n*.

In other words, worstSpace(*n*) is linear in *n*.

averageTime(*n*)? averageSpace(*n*)?

Any problem that can be solved recursively can also be solved iteratively.

An *iterative* method is one that has a loop statement.

```
/**
 *  Returns the factorial of an integer. The worstTime(n)
 *  is O(n).
 *
 *  @param n – the integer whose factorial is returned.
 *  @return n!
 *  @throws IllegalArgumentException – if n is less than 0.
 *
 */
public static long factorial (int n) {
      int product = n;
      if (n < 0)
            throw new IllegalArgumentException( );
      if (n == 0)
           return 1;
      for (int i = n-1; i > 1; i--)
           product = product * i;
      return product;
} // method factorial
```

The number of loop iterations is *n* – 1.

So worstTime(*n*) is linear in *n*.

The number of variables used in any trace of this function is 3.

So worstSpace(*n*) is constant.

To trace the execution of the recursive factorial function:

http://www.cs.lafayette.edu/~collinsw/factorial/factorialins.html

Example 2: Converting from Decimal to Binary

The argument is a non-negative decimal integer.

The return value is the binary representation of that integer.

Run the following applet to see the user’s view:

http://www.cs.lafayette.edu/~collinsw/writebinary/binary.html

For the binary equivalent of 34:

The rightmost bit is 34 % 2 = 0;

The other bits are the binary equivalent of 34 / 2, which is 17.

For the binary equivalent of 17:

The rightmost bit is 17 % 2 = 1;

The other bits are the binary equivalent of 17 / 2, which is 8.

34 % 2 = 0 → 0
34 / 2 = 17

17 % 2 = 1 → 1
17 / 2 = 8

8 % 2 = 0 → 0
8 / 2 = 4

4 % 2 = 0 → 0
4 / 2 = 2

2 % 2 = 0 → 0
2 / 2 = 1 → 1

Read bits from bottom to top:

100010

We need to calculate the binary equivalent of *n* / 2 before we append *n* % 2. Otherwise, the string returned will be in reverse order.

Stop when *n* = 1 or 0, and return *n* as a string.

```
/**
 *  Returns a String representation of the binary
 *  equivalent of a specified integer.  The worstTime(n)
 *  is O(log n).
 *
 *  @param n – an int in decimal notation.
 *  @return the binary equivalent of n, as a String
 *  @throws IllegalArgumentException, if n is less than 0
 */
public static String getBinary (int n) {
      if (n < 0)
            throw new IllegalArgumentException( );
      if (n <= 1)
            return Integer.toString (n);
      return getBinary (n / 2) + Integer.toString (n % 2);
} // method getBinary
```

Analysis:

For $n > 0$, the number of recursive calls is the number of times that n can be divided by 2 until $n = 1$.

According to the analysis done in Chapter 3, that number is floor($\log_2 n$).

So worstTime(n) is logarithmic in n.

WHAT ABOUT worstSpace(n)?

Because worstSpace(n) is proportional to the number of recursive calls, worstSpace(n) is logarthmic in n.

Exercise: Use execution frames to trace the execution of

`getBinary` **(20);**

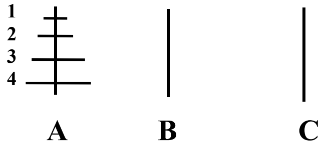
recall that each time a recursive call is made, a new execution-frame box is "stacked" on top of the other execution-frame boxes.

Example 3: Towers of Hanoi

Given 3 poles (A, B, C) and *n* disks of increasing size (1, 2, 3, …, *n*), move the *n* disks from pole A to pole B. Use pole C for temporary storage.

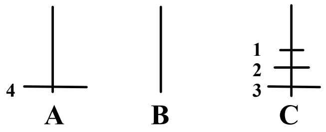
1. **Only one disk may be moved at any time.**
2. **No disk may ever be placed on top of a smaller disk.**
3. **Other than the prohibition of rule 2, the top disk on any pole may be moved to either of the other poles.**

Initially, with 4 disks:



Instead of trying to figure out where to move disk 1, let's look at the picture just before disk 4 is moved:

Just before disk 4 is moved:



So we will be able to move 4 disks from one pole to another if we are able to figure out how to move 3 disks from one pole to another (Aha!).

To move 3 disks …

If *n* = 1, move disk 1 from pole ‘A’ to pole ‘B’.

Otherwise,

1. **Move *n – 1* disks from ‘A’ to ‘C’, with ‘B’ as a temporary.**
2. **Move disk *n* from ‘A’ to ‘B’.**
3. **Move *n – 1* disks from ‘C’ to ‘B’, with ‘A’ as a temporary.**

For the sake of generality, use variables instead of constants for the poles:

orig = ‘A’
dest = ‘B’
temp = ‘C’

Here is the strategy to move n disks from orig to dest:

If n = 1, move disk 1 from orig to dest.

Otherwise,

move n-1 disks from orig to temp.

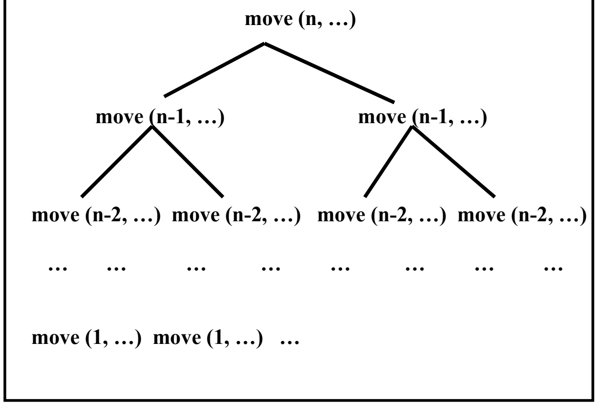
Move disk n from orig to dest.

Move n-1 disks from temp to dest

```
/**
 * Determines the steps needed to move disks from an origin to a destination.
 * The worstTime(n) is O(2^n).
 *
 * @param n the number of disks to be moved.
 * @param orig the pole where the disks are originally.
 * @param dest the destination pole
 * @param temp the pole used for temporary storage.
 * @return a String representation of the moves needed.
 * @throws IllegalArgumentException if n is less than or equal to 0.
 */
public static String move (int n, char orig, char dest, char temp) {
   final String DIRECT_MOVE =
                    "Move disk " + n + " from " + orig + " to " + dest + "\n";
   if (n <= 0)
        throw new IllegalArgumentException( );
   if (n == 1)
        return DIRECT_MOVE;
   return move (n - 1, orig, temp, dest) + DIRECT_MOVE +
          move (n - 1, temp, dest, orig) ;
} // method move
```

Analysis:

worstTime(*n*) ≈ # of calls to move



move (n, …)

move (n-1, …) **move (n-1, …)**

move (n-2, …) **move (n-2, …)** **move (n-2, …)** **move (n-2, …)**

… … … … … … … …

move (1, …) **move (1, …)** …

There are n levels in this tree.

The number of calls to move at level 0 is 1 = 2^0

The number of calls to move at level 1 is 2 = 2^1

The number of calls to move at level 2 is 4 = 2^2

...

The number of calls to move at level n-1 is 2^{n-1}

The total number of calls to move is:

$$1 + 2 + 4 + \ldots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i$$

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

See example 6 of Appendix 2 for a proof by mathematical induction.

This shows that worstTime(n) is $O(2^n)$, and, because 2^n - 1 disks must be moved, worstTime(n) is $\Omega(2^n)$, that is, 2^n is also a lower bound. We conclude that worstTime(n) is $\theta(2^n)$.

To trace the execution of this recursive move **function:**
http://www.cs.lafayette.edu/~collinsw/hanoi2/hanoiins.html

Exercise: Re-write the move method to print out the steps needed to move n disks. The heading is

```
public static void move (int n,
                         char orig,
                         char dest,
                         char temp)
```

Example 4:

Searching an Array

We assume that the array elements are in a class that implements the Comparable **interface:**

```
public interface Comparable
{
    int compareTo(Object obj);
}
```

The int **returned by**

x.compareTo (y)

is < 0, if x **is less than** y**;**

is = 0, if x **is equal to** y**;**

is > 0, if x **is greater than** y**.**

Sequential search of an array for an element key**:**

Start at Index 0: Compare each element in the array to key **until success (**key **found) or failure (end of array reached).**

```
/**
 *  Determines whether an array contains an element equal
 *  to a given key. The worstTime(n) is O(n).
 *
 * @param a the array to be searched.
 * @param key the element searched for in the array a.
 * @return the index of an element in a that is equal to key, if
 *             such an element exists; otherwise, -1.
 * @throws ClassCastException, if the element class does
 *              not implement the Comparable interface.
 */
public static int sequentialSearch (Object[ ] a, Object key) {
   for (int i = 0; i < a.length; i++)
         if ((Comparable) a [i].compareTo (key) == 0)
               return i;
   return –1;
} // sequentialSearch
```

The worstTime(*n*) is linear in *n*.

The averageTime(*n*) is linear in *n*.

Binary Search:

During each iteration, the size of the subarray searched is divided by 2 until success or failure occurs.

Note: The array must be sorted!

The basic idea:

Compare a [middle index] to key:

< 0: Search a [middle index + 1 … a.length – 1];

> 0: Search a [0 … middle index – 1];

= 0: Success!

```
/**
 * Searches the specified array for the specified object using
 * the binary search algorithm.  The array must be sorted into
 * ascending order according to the natural ordering of
 * its elements prior to making this call.  If it is not sorted,
 * the results are undefined.  If the array contains multiple
 * elements equal to the specified object, there is no
 * guarantee which one will be found.  The worstTime(n) is
 * O(log n).
 *
 * @param a - the array to be searched.
 * @param first - smallest index in the region of the array now
 *              being searched
 * @param last - the largest index in the region of the array
 *             now being searched.
 * @param key the value to be searched for.
```

```
 * @return index of the search key, if it is contained in the array;
 *         otherwise, <tt>(-(<i>insertion point</i>) - 1)</tt>.  The
 *         <i>insertion point</i> is defined as the point at which the
 *         key would be inserted into the array: the index of the first
 *         element greater than the key, or <tt>a.length</tt>, if all
 *         elements in the array are less than the specified key.
 *         Note that this guarantees that the return value will be
 *         &gt;= 0 if and only if the key is found.
 *
 * @throws ClassCastException if the array contains elements
 *       that are not <i>mutually comparable</i> (for example,
 *        strings and integers), or the search key is not mutually
 *        comparable with the elements of the array.
 */
public static int binarySearch (Object[ ] a, int first, int last,
                                Object key)
```

---

a [0]   Andrew
a [1]   Brandon
a [2]   Chris
a [3]   Chuck
a [4]   Geoff
a [5]   Jason
a [6]   Margaret
a [7]   Mark
a [8]   Matt
a [9]   Rob
a [10] Samira

Search for “Matt”, “Jeremy”, “Amy”, “Zach”

What if 6 is returned?

What if -6 is returned?

```
if (first <= last)
{
    int mid = (first + last) / 2;
    Comparable midVal = (Comparable)a [mid];
    int comp = midVal.compareTo (key);
    if (comp < 0)
        return binarySearch (a, mid + 1, last, key);
    if (comp > 0)
        return binarySearch (a, first, mid - 1, key);
    return mid;  // key found
} // if first <= last
```

Suppose the array is:

a [0] Andrew
a [1] Brandon
a [2] Chris
a [3] Chuck
a [4] Geoff
a [5] Jason
a [6] Margaret
a [7] Mark
a [8] Matt
a [9] Rob
a [10] Samira

Search for “Mark”.

Search for “Carlos”.

```
public static int binarySearch(Object[ ] a, int first, int last,
                                                    Object key)
{
      if (first <= last)
      {
          int mid = (first + last) / 2;
          Comparable midVal = (Comparable)a [mid];
          int comp = midVal.compareTo (key);
          if (comp < 0)
              return binarySearch (a, mid + 1, last, key);
          if (comp > 0)
              return binarySearch (a, first, mid - 1, key);
          return mid;  // key found
      } // if first <= last
      return -first - 1; // key not found; belongs at a [first]
} // method binarySearch
```

Binary Search Tester Applet:

http://www.cs.lafayette.edu/~collinsw/binary/binaryins.html

Analysis: Let *n* = size of initial region to be searched.

Unsuccessful search:

Keep dividing *n* by 2 until *n* = 0.

Unsuccessful Search

The number of divisions will be:

floor($\log_2 n$) + 1

Unsuccessful Search

So worstTime(*n*) is logarithmic in *n*.

Unsuccessful Search

The averageTime(*n*) is also logarithmic in *n* because, for an unsuccessful search, the algorithm terminates only when *n* = 0.

Successful Search:

Worst case: Keep dividing *n* by 2 until *n* = 1. Then first = last = middle.

Successful Search

The worstTime(*n*) is logarithmic in *n*.

Successful Search

The averageTime(*n*) is logarithmic in *n*. (See Concept Exercise 5.7.)

See Lab 8 for an iterative version that is:

1. **Slightly faster;**
2. **In the Java collections framework.**

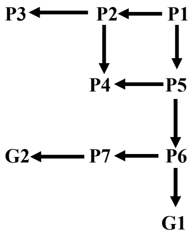
Exercise: Provide the successive values of `first`, `middle` (if calculated) and `last` if the above array `a` is searched for "Jeremy."

Example 5:

Backtracking

Backtracking **is the strategy of trying to reach a goal by a sequence of chosen positions, with re-tracing in reverse order of positions that cannot lead to the goal.**

Strategy: Try to go west; if unable to go west, try to go south; if unable to go south, backtrack (until you can go south). Do not go to any position that has been shown not to lead to a goal. The goal is either G1 or G2. Start at P1.



When a position is visited, it is marked as (potentially) being on a path to the goal. If we discover otherwise, the marking must be undone, so that position will never again be visited. For example, P4 is not visited from P5.

We will solve this maze-search problem within the general framework of backtracking, which can easily be utilized in other applications.

The Application **interface and the** Backtrack **class are the same for any backtracking project.**

```
import java.util.*;

public interface Application {

     // Returns true if it pos is a legal position and not a dead end.
     boolean isOK (Position pos);

     // Marks pos as possibly being on a path to a goal position.
     void markAsPossible (Position pos);

     // Returns true if pos is a goal position.
     boolean isGoal (Position pos);
```

```
     // Marks pos as not being on a path to a goal position.
     void markAsDeadEnd (Position pos);


     // Returns a string representation of this Application.
    String toString();


    // Returns an iterator over the positions directly
    // accessible from pos.
    Iterator<Position> iterator (Position pos);

} // interface Application
```

In any class that implements the Application interface, there will be an embedded iterator class with the usual methods: hasNext( ) and next( ).

Here are the method descriptions for the BackTrack class:

```
// Initializes this BackTrack object from app.
public BackTrack (Application app)


// Returns true if a solution going through pos was
// successful.
public bool tryToReachGoal (Position pos)
```

The only field in the BackTrack class is app, OF TYPE Application.

The definition of the BackTrack class starts with:

```
import java.util.*;

public class BackTrack {

    Application app;


    public BackTrack (Application app) {

        this.app = app;

    } // constructor
```

The tryToReachGoal method: First construct an iterator from pos of all positions immediately accessible from pos.

Then loop until success has been achieved or no more iterations are possible.

Each loop iteration considers several possibilities for the new position, pos,

Generated by the iterator:

1. pos **a goal! Return true.**
2. **Might be on path to goal; then mark, and see if a goal can be reached from the current value of** pos.
    a. **Yes? Return true;**
    b. **No? Mark** pos **as dead end. Iterate again if hasNext(); return false if iterator at end.**

```
public boolean tryToReachGoal (Position pos) {
     boolean success = false;
      Iterator<Position> itr = app.iterator (pos);
      while (!success && itr.hasNext()) {
               pos = itr.next();
               if (app.isOK (pos)) {
                    app.markAsPossible (pos);
                    if (app.isGoal (pos))
                         success = true;
                    else {
                         success = tryToReachGoal (pos);
                         if (!success)
                              app.markAsDeadEnd (pos);
                    } // goal not yet reached
               } // a legal, not-dead-end position
      } // while
      return success;
} // method tryToReachGoal
```

A user supplies:

- **A specific application**
- **What “position” means in the application**
- **A way to iterate from a given position**

Specific Application

Maze Searching

Maze searching: **1 = Corridor;**
0 = Wall

start → 1 1 1 0 1 1 0 0 0 1 1 1 1
1 0 1 1 1 0 1 1 1 1 1 0 1
1 0 0 0 1 0 1 0 1 0 1 0 1
1 0 0 0 1 1 1 0 1 0 1 1 1
1 1 1 1 1 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 ← finish

Iterator choices: north, east, south, west

Marked as possible = 9; dead end = 2

To watch a solution being created:

http://www.cs.lafayette.edu/~collinsw/maze/MazeApplet.html

To retrieve the project so you can run it:

http://www.cs.lafayette.edu/~collinsw/cs103/proj1/maze.html

Solution: 9 = Path; 2 = dead end

9 9 9 0 2 2 0 0 0 2 2 2 2
1 0 9 9 9 0 2 2 2 2 2 0 2
1 0 0 0 9 0 2 0 2 0 2 0 2
1 0 0 0 9 2 2 0 2 0 2 2 2
1 1 1 1 9 0 0 0 0 1 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 0
0 0 0 0 9 9 9 9 9 9 9 9 9

All that need to be developed are the Position class, the class that implements the Application interface and a tester.

For this application, a position is simply a row and column, so:

```
protected int row,
                   column;


public Position (int row, int column)
{
      this.row = row;
      this.column = column;
} // two-parameter constructor

...
```

The definitions of the Position methods are straightforward. For example:

```
public int getRow( )
{
    return row;
} // method getRow( )
```

For this application, Maze.java implements the Application interface, with a grid to hold the maze.

```
public class Maze implements Application {

    protected final byte WALL = 0;
    protected final byte CORRIDOR = 1;
    protected final byte PATH = 9;
    protected final byte DEAD_END = 2;

    protected Position finish;

    protected byte[ ][ ] grid;     // "hard-wired" or read in
```

Here, for example, is the definition of the method isOK:

```
public boolean isOK (Position pos)
{
    if (pos.getRow() >= 0 &&
         pos.getRow() < grid.length &&
         pos.getColumn() >= 0 &&
         pos.getColumn() < grid [0].length &&
          grid [pos.getRow()][pos.getColumn()] ==
                                    CORRIDOR)
         return true;
   return false;
 } // method isOK
```

Finally, we define the MazeIterator class embedded in the Maze class.

The MazeIterator **class has** row **and** column **fields to keep track of where the iterator is, and a** count **field to keep track of how many times the iterator has advanced (at most 3: north to east, east to south, and south to west).**

```
// Postcondition: This MazeIterator object has been
//                        initialized from pos.
public MazeIterator (Position pos) {
     row = pos.getRow();
     column = pos.getColumn();
     count = 0;
} // constructor


// Postcondition: true has been returned if this
//                       MazeIterator object can still advance.
//                       Otherwise, false has been returned.
public boolean hasNext()
{
     return count < MAX_MOVES  // = 4;
} // method hasNext
```

```
// Precondition: count < MAX_MOVES (= 4).
// Postcondition: the choice for the next Position has
//                        been returned.
public Position next() {
     Position nextPosition = new Position();
     switch (count++) {
          case 0: nextPosition = new Position (row-1, column);
                     break;           // NORTH
          case 1: nextPosition = new Position (row, column+1);
                    break;            // EAST
          case 2: nextPosition = new Position (row+1, column);
                    break;            // SOUTH
          case 3: nextPosition = new Position (row, column-1);
                                          // WEST
     } // switch;
     return nextPosition;
} // method next
```

Exercise: Recall the solution when the order was north, east, south, west:

9 9 9 0 2 2 0 0 0 2 2 2 2
1 0 9 9 9 0 2 2 2 2 2 0 2
1 0 0 0 9 0 2 0 2 0 2 0 2
1 0 0 0 9 2 2 0 2 0 2 2 2
1 1 1 1 9 0 0 0 0 1 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 0
0 0 0 0 9 9 9 9 9 9 9 9 9

Re-solve with the order north, east, west, south:

start → 1 1 1 0 1 1 0 0 0 1 1 1 1
1 0 1 1 1 0 1 1 1 1 1 0 1
1 0 0 0 1 0 1 0 1 0 1 0 1
1 0 0 0 1 1 1 0 1 0 1 1 1
1 1 1 1 1 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 ← finish

Hint: Only one ‘1’ remains.

The Cost of Recursion

Whenever a method is called, some information is saved to prevent its destruction in case the call is recursive. This information is called an *activation record*.

An activation record is an execution frame without the statements. Each activation record contains:

1. **The return address;**
2. **A copy of each argument;**
3. **A copy of each of the method's other local variables.**

After the call has been completed, the previous activation record's information is restored and the execution of the calling method resumes. The saving and restoring of these records takes time.

Basically, an iterative method will be slightly faster than its recursive counterpart. But for problems such as backtracking and towers of Hanoi, the recursive methods take a *lot less time* to develop than the iterative versions!