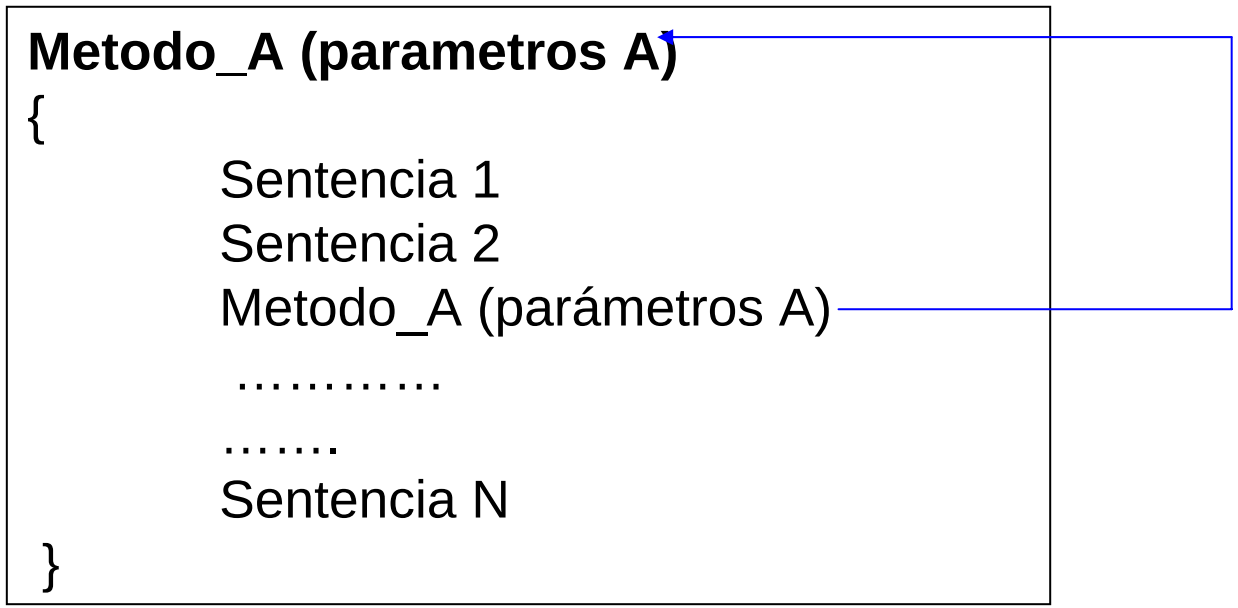
Recursividad

4.1 Definición.

- Se dice que algo es recursivo si se define en función de sí mismo o a sí mismo.
- Un objeto (problemas, estructuras de datos) es recursivo si forma parte de sí mismo o interviene en su propia definición.
- Consiste en definir la funcionalidad de un método u operación, dependiendo de un llamado interno al mismo método.
- Permite definiciones más simples de operaciones que iterativamente serían más complejas.
- Un método es recursivo si contiene invocaciones a sí mismo.
- Una llamada a un método recursivo puede generar una o más invocaciones al mismo método, que a su vez genera otras.

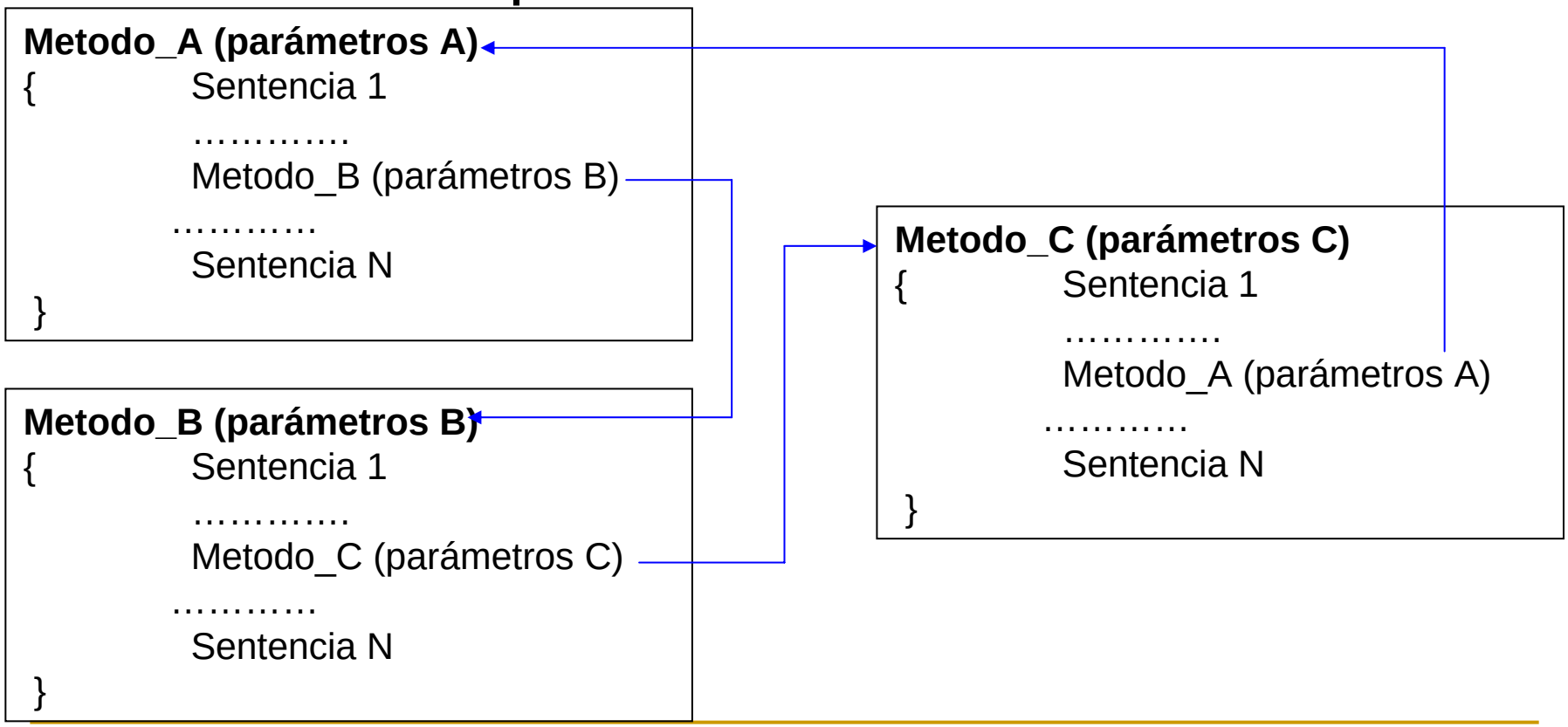
4.1.1 Recursividad directa

Un método hace referencia a sí mismo dentro de su definición.



4.1.2 Recursividad Indirecta o mutua

El método contiene llamadas a otros métodos que, finalmente, acaban provocando a su vez una llamada al primero.



4.1.3 Consideraciones de la recursividad

En una definición recursiva deben existir obligatoriamente:

- Caso(s) base(s): No provocan un nuevo cálculo recursivo, finalizando la recursión, con lo que se puede obtener una solución.
- Caso(s) recursivo(s): Son aquéllos que provocan la realización de nuevos cálculos recursivos (sobre datos más “pequeños”).

Para que un algoritmo recursivo no se convierta en infinito debe contener una condición que garantice la terminación del algoritmo recursivo en un tiempo finito.

4.1.4 Solución Iterativa o Recursiva

- No hay problemas intrínsecamente iterativos o recursivos.
- Sí hay problemas para los cuales la solución iterativa o recursiva es más sencilla, más natural, más fácil de ver.
- En algunos casos, el algoritmo recursivo es mucho más compacto y simple que el iterativo.
- Solución recursiva : costo de ejecución relativamente alto:
  - Número de Llamadas, manejo de la pila.
  - Costo en tiempo y en espacio de memoria por cada registro de activación.
- Sólo se suele elegir la solución recursiva cuando:
  - La penalización en tiempo y espacio no es significativa.
  - La solución recursiva es más elegante, legible y/o fácil de mantener.

4.2 Tipos de recursividad

- Recursividad Simple
- Recursividad Múltiple
- Recursividad de cola
- Recursividad con parámetros acumuladores
- Backtracking

4.2.1 Recursividad Simple

- La definición de la función incluye una única llamada recursiva.
- Cuando se aplica, se realiza una sola llamada recursiva en cada uno de los niveles que constituyen la aplicación.
- En la aplicación existen dos fases:
  - Niveles de anidamiento de llamadas recursivas hasta que se producen las condiciones de tope o condición de salida.
  - Vuelta atrás de la recursión con resultados intermedios en la expresión y tope en el valor que cumple con la condición de inicio.
- Ejemplo: Calcular el Factorial

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

4.2.2 Recursividad Múltiple

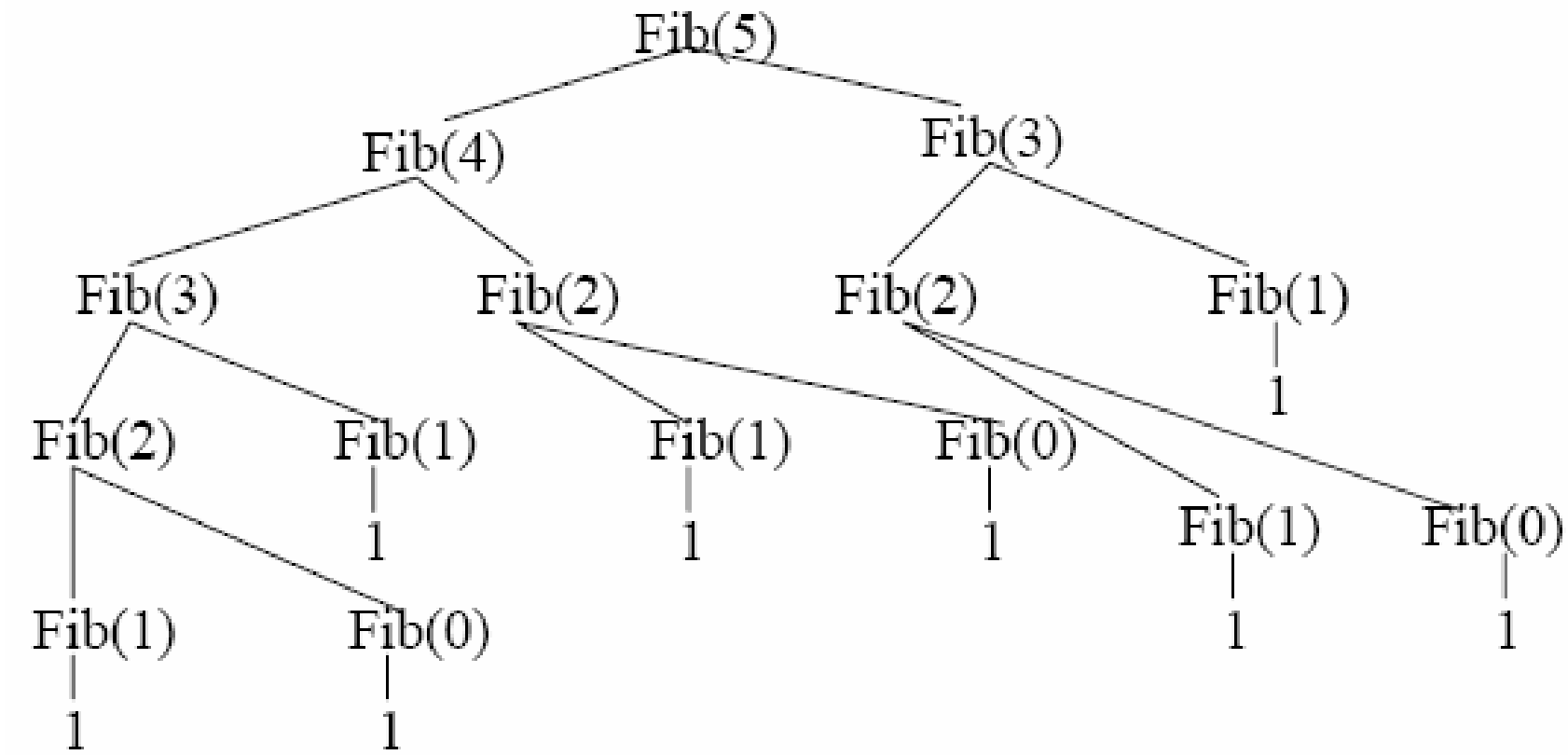
- En cada aplicación de la función se pueden producir varias aplicaciones de la misma función.
- Hay llamadas recursivas en varios puntos de la función.
- El seguimiento de una función recursiva múltiple describe un "árbol" de niveles de anidamiento de las llamadas recursivas efectuadas.

Ejemplo: Calcular la serie de Fibonacci.

```
public static int Fibo (int num) {
    if (num<=1) return (num);
    else return (Fibo(num-1)+Fibo(num-2));
}
```

4.2.2 Recursividad Múltiple (Ejemplo)

Árbol de llamadas recursivas (Fibonacci(5))



4.2.3 Recursividad final o de cola

- Todo el trabajo se hace antes de cada llamada recursiva. También llamada recursividad de Touretzky.
- Puede considerarse un tipo de recursividad complementaria a otros tipos de recursividad
- Al no existir cálculos intermedios con los valores devueltos por cada aplicación no se necesita el procedimiento de vuelta atrás visto en otros tipos de recursividad.
- La última aplicación devuelve el valor final resultante.
- Se acerca a los procesos iterativos y elimina una cantidad importante de operaciones sobre la pila de la recursividad.

```
public int CifrDere(int num,int pos) {
    if  (pos==1)   return(num %10);
     else  return (CifrDere(num / 10, pos-1));
}
```

Para CifrDere(192837465,6) el resultado es 8

4.2.4 Recursividad con parámetros acumuladores

- Se asienta sobre conceptos como la recursividad de cola y la acumulación de valores intermedios.
- El proceso de aplicación es el siguiente:
  - Se implementa una función principal no recursiva, con una regla de correspondencia a una función auxiliar (con recursividad de cola), que posee los mismos parámetros que la principal además de parámetros adicionales.
  - La función auxiliar toma el argumento del parámetro adicional como valor inicial y que, habitualmente, es el mismo que el devuelto en el caso base de la recursividad normal .
  - A partir de aquí se van haciendo llamadas de recursividad de cola que "acumulan" valores al valor inicial hasta llegar al resultado final.
- Se puede decir que la función principal indica el valor de partida mientras que la función auxiliar realiza el resto del proceso mediante la definición hecha con recursividad de cola.

4.2.5 Recursividad Backtracking (vuelta atrás)

- Se comporta como una búsqueda exhaustiva y sistemática del resultado deseado.
- En este caso no se conoce una regla de recursividad que permita obtener, mediante su aplicación, la solución que se persigue, por lo que se deben recorrer todas las posibles alternativas de solución.
- El procedimiento general es descomponer el proceso de tanteo de una solución en tareas parciales. Cada tarea parcial se expresa frecuentemente en forma recursiva.
- Construye tareas básicas y las inspecciona para determinar si conducen a la solución del problema. Si una tarea no conduce a la solución, prueba con otra tarea básica. Puede llegar a una solución o determinar que no hay solución.

Ejemplo: Problema de la vuelta del caballo (Ajedrez).

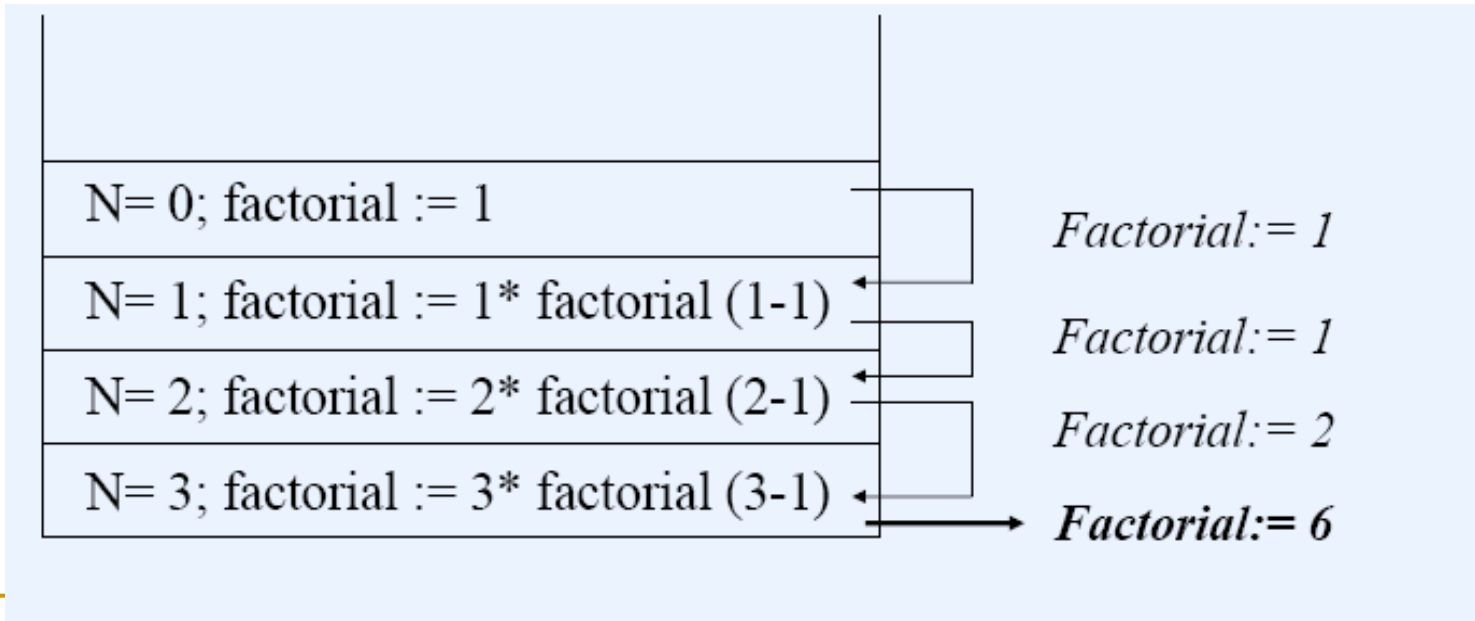
4.3 Mecánica de recursión.

- Los parámetros y variables locales toman nuevos valores en cada llamada (no se trabaja con los anteriores).
- Cada vez que se llama un método, el valor de los parámetros y variables locales se almacenan en la pila de ejecución. Cuando termina la ejecución se recuperan los valores de la activación anterior.
- El espacio necesario para almacenar los valores en memoria (pila) crece en función de las llamadas.
- Con cada llamada recursiva se crea una copia de todas las variables y constantes que estén vigentes, y se guarda esa copia en la pila.
- Además se guarda una referencia a la siguiente instrucción a ejecutar.
- Al regresar se toma la imagen guardada (registro de activación) en el tope de la pila y se continúa operando. Esta acción se repite hasta que la pila quede vacía.

4.3.1 Ejemplo de mecánica de la recursividad

```
public int factorial(int N) {
If (N==0) return(1); //Estado Base
    else return(N*factorial(N-1); //Sol. recursiva
}
```

Estado de la pila:



4.3.2 Funcionamiento de la recursión

- Cada llamada recursiva se trata igual que una llamada normal:
  - Nueva copia de todas las variables locales.
  - Nueva copia de todos los parámetros del método.
  - Las variables miembro son las mismas.
- Cada sucesiva llamada requiere, pues, un conjunto de variables locales y parámetros.
- Se hace uso de la Pila del sistema para guardar el registro de activación de cada llamada o invocación.

4.4 Transformación de algoritmos recursivos a iterativos

- Toda solución recursiva a un problema tiene un equivalente iterativo.
- Podemos simular en nuestro programa lo que haría la pila del sistema.
- Existe un método bien definido para convertir un programa recursivo en iterativo.
- Toda solución iterativa a un problema tiene un equivalente recursivo:
  - Cualquier bucle puede convertirse en un método recursivo.
  - Caso base: final del bucle. Caso general: cada paso (la llamada recursiva modifica el parámetro igual que cada paso del bucle).

4.4.1 Procedimiento general para emular recursión

1. Definir una pila para cada variable local y cada parámetro o argumento, y una pila para almacenar direcciones de retorno.
2. En la sentencia n donde se haga la llamada recursiva al subprograma:
   - Meter en las respectivas pilas los valores actuales de las variables locales y argumentos.
   - Meter en la pila de direcciones la dirección de la siguiente sentencia.
   - Inicializar los argumentos con el valor actual de ellos y empezar la ejecución desde el principio del procedimiento o subprograma.
3. Vuelta de la ejecución después de la llamada recursiva:
   - Si la pila de direcciones está vacía, devolver control al programa invocador.
   - Sacar de las pilas los valores del tope. Llevar la ejecución a la sentencia extraída de la pila de direcciones.

4.5 Recursividad en el diseño.

- Una de las técnicas más importantes para el diseño de algoritmos es la técnica llamada “divide y vencerás”.
- Consiste esta técnica en transformar un problema de tamaño n en problemas más pequeños, de tamaño menor que n.
- De modo que dando solución a los problemas unitarios se pueda construir fácilmente una solución del problema completo.
- Un algoritmo “divide y vencerás” puede ser definido de manera recursiva, de tal modo que se llama a sí mismo aplicándose cada vez a un conjunto menor de elementos. La condición para dejar de hacer llamadas es, normalmente, la obtención de un solo elemento.

4.5.1 Consideraciones de diseño

Para encontrar una solución recursiva a un problema:

- Buscar el caso base (solución inmediata)
- Buscar el caso general:
  - Solución parcial
  - Damos por supuesto que está resuelto el problema de tamaño "inmediatamente inferior"
  - Nos aseguramos de que las sucesivas llamadas recursivas que disminuyen el tamaño del problema nos acaban llevando al caso base.

4.5.1 Consideraciones de diseño (continuación)

- Si no se controla bien el caso base, el caso general se invoca a sí mismo de manera indefinida
- ¿Cuándo se detiene la cadena de llamadas?
- Desbordamiento de la pila del sistema:
  - Recuérdese: cada llamada a función implica habilitar espacio en la pila del sistema, para la dirección de retorno, variables locales y parámetros
  - Al hacer esto "hasta el infinito"... la pila del sistema se llena
  - (el tamaño de la memoria no es infinito).
  - Error en tiempo de ejecución; conocido como stack overflow.

4.6 Complejidad de los algoritmos recursivos.

Para estudiar la complejidad de los algoritmos recursivos se emplean sobre todo los tres métodos siguientes:

- La inducción matemática.
- Expansión de recurrencias. Consiste en sustituir la recurrencia por su igualdad hasta llegar a un caso conocido (más utilizado por su sencillez).
- Usar soluciones generales ya conocidas y compararlas con las que se quieren obtener (Ecuaciones en diferencias finitas).

4.6.1 Costo de las llamadas recursivas

Cada llamada a una función exige:

- Instrucciones (de la máquina) para saltar al punto adecuado del código.
- Habilitar espacio en la pila del sistema, y crear en él las variables locales y los parámetros.
- Inicializar esas variables locales y parámetros.
- En algunos casos, eso implica crear objetos y por tanto llamar a constructores.
- "Anotar" en esa pila la dirección de retorno ("de dónde se viene").
- Esto consume tiempo y espacio.

Ejemplo: para calcular recursivamente el factorial de 5, se hacen 5 llamadas a función y se generan 5 registros de activación en la pila del sistema.