Ficha 26

Estrategias de Resolución de Problemas: Recursividad

1.] Introducción.

Cuando en la práctica se habla de *recursividad*, se está haciendo referencia a una muy particular forma de expresar la definición de un objeto o un concepto. Esencialmente, una definición se dice *recursiva* si el objeto o concepto que está siendo definido *aparece a su vez en la propia definición*. Consideremos por ejemplo, la siguiente definición:

*Una **frase** es un conjunto de palabras que puede estar vacío, o bien puede contener una palabra seguida a su vez de otra **frase**.*

Estamos frente a una definición recursiva, puesto que el objeto definido (la *frase*) se está usando en la misma definición, al indicar que una frase puede ser *una palabra seguida a su vez de otra frase*.

Al plantear una definición recursiva debe tenerse cuidado con los siguientes elementos [1]:

1. Una definición recursiva correctamente planteada, exige que la definición *agregue conocimiento* respecto del concepto u objeto que se define. No basta con que el objeto definido aparezca en la definición, pues si sólo nos limitamos a esa regla podrían producirse definiciones obviamente verdaderas, pero sin aportar conocimiento alguno. Por ejemplo, si decimos que:

   Una frase es una frase.

   no cabe duda en cuanto a que esa afirmación es recursiva, pero de ninguna manera estamos definiendo lo que ***es*** una frase. En todo caso, lo que tenemos es una *identidad*, y no una *definición*. En cambio en la definición original que dimos de la idea de *frase*, encontramos elementos que nos permiten construir paso a paso el concepto de *frase*, a partir de la noción de *frase vacía* y la noción de *palabra*, y esos elementos son los que agregan conocimiento al concepto.

2. Por otra parte, una definición recursiva correctamente planteada debe evitar la *recursión infinita* que se produce cuando en la definición no existen elementos que permitan cerrarla lógicamente. Se cae así en una definición que, aunque agrega conocimiento, no termina nunca de referirse a sí misma. Por ejemplo, si la definición original de *frase* fuera planteada así:

   Una frase *es un conjunto que consta de una palabra seguida a su vez de una frase.*

   tenemos que esta nueva definición es recursiva y aparentemente está bien planteada, por cuanto agrega conocimiento al indicar que una frase consta de palabras y frases. Pero al no indicar que una frase puede estar vacía, la noción de frase se torna en un concepto *sin fin*: cada vez que decimos *frase*, pensamos en una palabra, y en otra *frase*, lo cual a su vez lleva a otra palabra y a una nueva *frase*, y así sucesivamente, sin solución de continuidad.

Las definiciones recursivas no son muy comunes en la vida cotidiana porque en principio, siempre existe y siempre resulta más simple pensar y entender una definición directa, sin la vuelta hacia atrás que supone la recursividad. La misma noción de frase, puede definirse sin recursividad de la forma siguiente:

Una frase es una sucesión finita de palabras, que puede estar vacía.

y esta definición resulta más natural y obvia (incluso podría no indicarse que la frase puede estar vacía, y la definición seguiría teniendo sentido).

Sin embargo, en ciertas disciplinas como la Matemática, la recursividad se usa con mucha frecuencia debido a que existen problemas cuya descripción simbólica es más compacta y consistente con recursividad que sin ella[1]. Consideremos el típico y ya conocido caso del *factorial* de un número *n*. Como sabemos, si *n* es un entero positivo o cero, entonces el *factorial* de *n* (denotado como *n!*), se puede definir sin recursividad, como sigue:

$$\mathbf{n!} = \begin{cases} \text{si } n = 0 \Rightarrow \mathbf{0! = 1} \\ \text{si } n > 0 \Rightarrow \mathbf{n! = n * (n-1) * (n-2) * \ldots * 3 * 2 * 1} \end{cases}$$

Es decir: si *n* es cero, su factorial vale uno. Pero si *n* es mayor a cero, su factorial es el producto de *n* por todos los enteros positivos anteriores a *n*, hasta el uno. De esta forma, el factorial de 4 sería:

```
4! = 4 * 3 * 2 * 1
4! = 24
```

La fórmula dada para el cálculo del factorial es sencilla, pero incluye una serie de puntos suspensivos que no siempre son bien recibidos, pues se presupone que quien lee la fórmula será capaz de deducir por sí mismo la forma de llenar el espacio de los puntos suspensivos. Eso podría no ser tan simple si la fórmula fuese más compleja.

La recursividad ofrece una alternativa de notación más compacta, evitando los puntos suspensivos. Por ejemplo, si se observa la definición para *n > 0*, tenemos:

```
n! = n * (n - 1) * (n - 2) * . . . * 3 * 2 * 1
```

En esta fórmula es claramente visible que la expresión *(n – 1) * (n – 2) * . . . * 3 * 2 * 1* es equivalente al factorial del número *(n – 1)*: se está multiplicando a *(n – 1)* por *todos los números enteros anteriores a él, hasta llegar al uno*. Por lo tanto, la definición original podría escribirse así:

```
n! = n * (n - 1)!
```

[1] La idea de un concepto o un proceso que recurre a sí mismo para completarse puede aplicarse a la extraña situación de un sueño dentro de un sueño, que fue utilizada ya por *Edgar Alan Poe* en 1849 en su poema "*A Dream Within a Dream*". Este poema a su vez, inspiró en 1976 una composición musical instrumental de la banda *The Alan Parsons Project* tambien llamada "*A Dream Within a Dream*". Y en el cine, la referencia es obvia: la película *Inception* (conocida como *El Origen* en Hispanoamérica) del año 2010, dirigida por *Christopher Nolan* y protagonizada por *Leonardo DiCaprio*, narra una atrapante historia de un equipo de espías industriales que navega en ¡tres niveles de sueños recursivos!

En otras palabras: si *n > 0*, entonces el *factorial* de *n* es igual a *n* multiplicado por el *factorial* de *(n – 1)*. Si reunimos todo en una definición global, tenemos:

$$n! = \begin{cases} \text{si } n = 0 \Rightarrow \mathbf{0! = 1} \\ \text{si } n > 0 \Rightarrow \mathbf{n! = n * (n\text{-}1)!} \end{cases}$$

y llegamos así a una *definición recursiva*: para definir el *factorial* de *n*, en la misma definición se recurre al *factorial* de *(n – 1)*. Observemos que la condición

```
si  n = 0  entonces  0! = 1
```

es la que evita la *recursión infinita*, y que la condición

```
si  n > 0 entonces  n! = n * (n – 1)!
```

provoca el paso recursivo, pero agregando conocimiento al mismo: un *factorial* es en última instancia un *producto* de un número por el *factorial* del *número precedente*.

2.] Programación recursiva.

La noción de definición recursiva vista hasta aquí sirve como punto de partida para plantear *algoritmos* en forma recursiva (de hecho, la definición recursiva del factorial que se estudió, no es más que un *algoritmo recursivo* para calcular ese factorial).

Prácticamente todos los lenguajes de programación modernos soportan la recursividad, a través del planteo de *subrutinas recursivas*. En Python, la idea es que un algoritmo recursivo puede implementarse a través de *funciones de comportamiento recursivo* [2]. En términos muy básicos, una *función recursiva* es una función que incluye en su bloque de acciones *una o más invocaciones a sí misma*.

En otras palabras, una función recursiva es aquella que se invoca a sí misma una o más veces. En esencia, entonces, la siguiente función es recursiva (aunque aclaramos que está *mal planteada*):

```
def procesar():
    procesar()
```

Si la condición para ser recursiva es invocarse a sí misma, entonces la función anterior cumple el requisito: de hecho, *lo único* que hace es invocarse a sí misma. Sin embargo, como ya se dijo, está mal planteada: aún sin saber mucho respecto de cómo trabaja la recursividad en Python, un breve análisis inmediatamente permite deducir que una vez invocada esta función provoca una *cascada* infinita de auto-invocaciones, sin realizar ninguna tarea útil. Como ya veremos, este *proceso recursivo infinito* provocará tarde o temprano una falla de ejecución por falta de memoria, y el programa se interrumpirá.

¿Por qué está mal planteada esta función? En realidad ya conocemos la respuesta: cuando se introdujo a nivel teórico el tema de las definiciones recursivas bien planteadas en la sección anterior, se indicó que estas deberían *agregar conocimiento* respecto del concepto definido, y evitar la *recursión infinita* incluyendo una condición que corte el proceso recursivo. Y bien: la función *procesar()* aquí planteada no incluye ninguno de los dos elementos. por lo que no corresponde a una definición recursiva bien planteada [2].

Para poner un ejemplo conocido, supongamos que se desea plantear una función recursiva para calcular el factorial de un número *n* que entra como parámetro. Si planteáramos una función *factorial(n)* para calcular *recursivamente* el factorial de *n*, pero lo hiciéramos a la manera incorrecta que vimos antes para *procesar()*, quedaría:

```
def factorial(n):
   return factorial(n)
```

La función *factorial* así planteada, correspondería a "definir" el factorial así:

El factorial de n es igual al factorial de n.

Y como se ve, ni la definición ni la función agregan conocimiento al concepto y son por lo tanto, incorrectas. Un segundo intento, sería:

```
def factorial(n):
    return n * factorial(n - 1)
```

En este caso, la función mostrada correspondería a definir al factorial de la siguiente forma:

```
n!  =  n * (n – 1)!
```

lo cual agrega conocimiento pero cae en *recursión infinita*, pues la invocación *factorial(n–1)* provoca una nueva llamada, y esta a su vez otra, y así en forma continua, sin definir en qué momento detener el proceso.

La *recursión infinita* se evita considerando que si *n == 0*, entonces el factorial de *n* es *1*. Una simple condición en la función y queda la versión final, ahora correcta:

```
def factorial(n):
    if n == 0:
          return 1
    else:
          return n * factorial(n-1)
```

3.] Seguimiento de la recursividad.

En el punto anterior vimos que una función recursiva bien planteada debe contener una *condición de corte* para evitar la recursión infinita, e incluir una o mas invocaciones a sí misma pero agregando algún tipo de explicitación del proceso. Ahora bien... ¿Cómo funciona todo esto? ¿Cómo hace un lenguaje de programación para ejecutar una función recursiva? Una persona poco entrenada en programación recursiva podría creer que la función *factorial()* que planteamos antes está incompleta (porque no incluye ningún ciclo para calcular el factorial), o asumir que es correcta pero no comprender nada en absoluto respecto de su funcionamiento.

Para entender como trabaja una función recursiva, lo mejor es intentar un seguimiento a partir de un esquema gráfico [1]. Supongamos que se invoca a la función *factorial()* para calcular el factorial del número 4. O sea, supongamos la siguiente invocación:

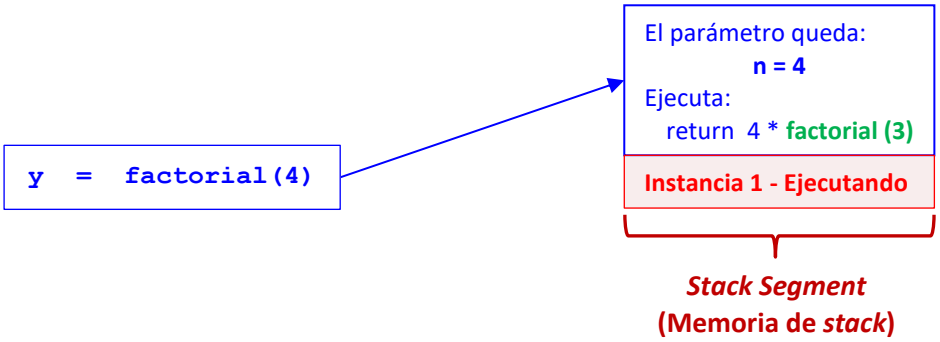
```
y = factorial(4)
```

Cuando una función es invocada (sea o no recursiva) automáticamente se asigna para ella un bloque de memoria en el segmento de memoria conocido como *Stack Segment* (o *Segmento de Pila*). Dentro de ese bloque, *la función crea y aloja sus parámetros formales y*

variables locales y luego comienza a ejecutarse. Se dice que se está ejecutando una *instancia* de la función (y en este caso, es la *primera instancia*).

Recordando que la función *factorial()* toma como parámetro una variable *n*, en la siguiente figura representamos con un rectángulo al bloque de memoria asignado a la función en esta *primera instancia de ejecución*:

Figura 1: Primera invocación a la función *factorial()*.



El valor 4 enviado como parámetro actual, es asignado en el parámetro formal *n*. Luego, la función verifica si *n* vale cero. En este caso, la condición sale por falso y ejecuta la rama *else*, que expresa:

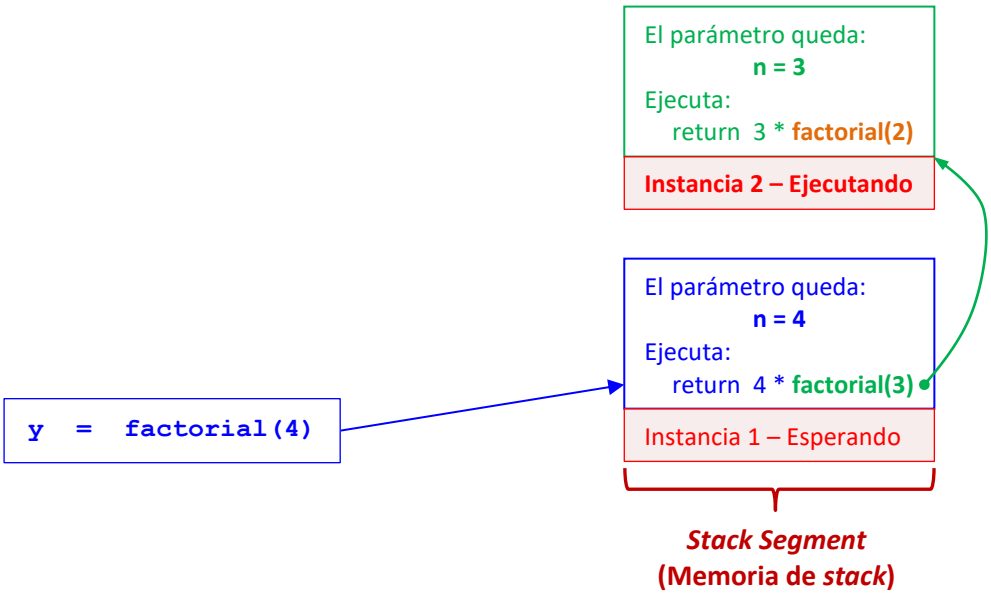
```
return  n * factorial(n - 1)
```

Como *n* vale 4, la expresión se evalúa como

```
return  4 * factorial(3)
```

Ahora bien: en esta última expresión se está invocando nuevamente a la función *factorial()*, pero ahora para calcular el factorial de 3. ***Aquí se produjo una llamada recursiva*** y el lenguaje Python actúa frente a ella de manera sencilla: trata a esa invocación recursiva como trataría a cualquier invocación normal de función: simplemente, le asigna a esa *segunda instancia* de ejecución una *nueva área* de memoria de *stack*, para que a su vez esta segunda instancia aloje en ella sus variables locales y parámetros formales. Nuevamente, mostramos un rectángulo para representar a la segunda instancia:

Figura 2: Segunda invocación (recursiva) a la función *factorial()*.



Lo importante aquí es entender que al asignar memoria para la *segunda instancia de ejecución*, la función *vuelve a crear* sus variables locales y parámetros formales, sin interferir con los ya creados para la *primera instancia*. El área de memoria de stack asignada a la *primera instancia* sigue ocupada, pero momentáneamente inactiva (la *primera instancia* de ejecución está esperando a que la *segunda* finalice).

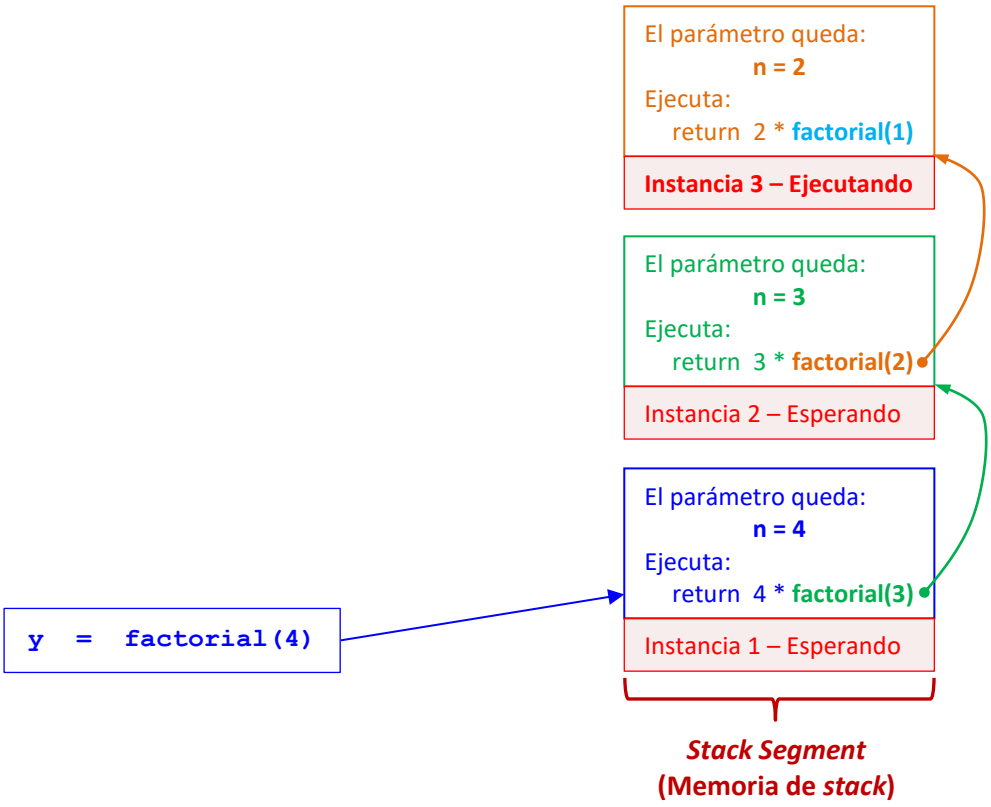
En otras palabras: la *segunda instancia* se comporta (y de hecho, *es*) como una función *diferente* de la primera, y cada una de ellas tiene *sus propias variables*. Aún cuando estas variables tienen los mismos nombres o identificadores en ambas instancias de ejecución, son variables diferentes *porque ocupan lugares distintos de la memoria*. El bloque de memoria de stack de la *primera instancia*, contiene una variable *n* cuyo valor es 4, y el bloque de memoria de stack de la *segunda instancia* contiene *otra* variable (también llamada *n*), pero con valor igual a 3. Mientras está activa (o sea, mientras se está ejecutando) la *segunda instancia*, la variable que se usa es la *n* cuyo valor es 3.

Cuando se ejecuta la *segunda instancia*, se repite el esquema que ocurrió en la *primera*, pero ahora con *n* valiendo 3. La expresión:

```
return  3 * factorial(2)
```

provoca a su vez *otra llamada recursiva*, que es alojada en *otra* área de memoria de memoria de stack. Esta *tercera instancia de ejecución* produce además una nueva creación de variables locales:

Figura 3: Tercera invocación (recursiva) a la función *factorial()*.



La *tercera instancia* lanza una nueva llamada recursiva y otra asignación de memoria de stack para la *cuarta instancia*, al hacer:

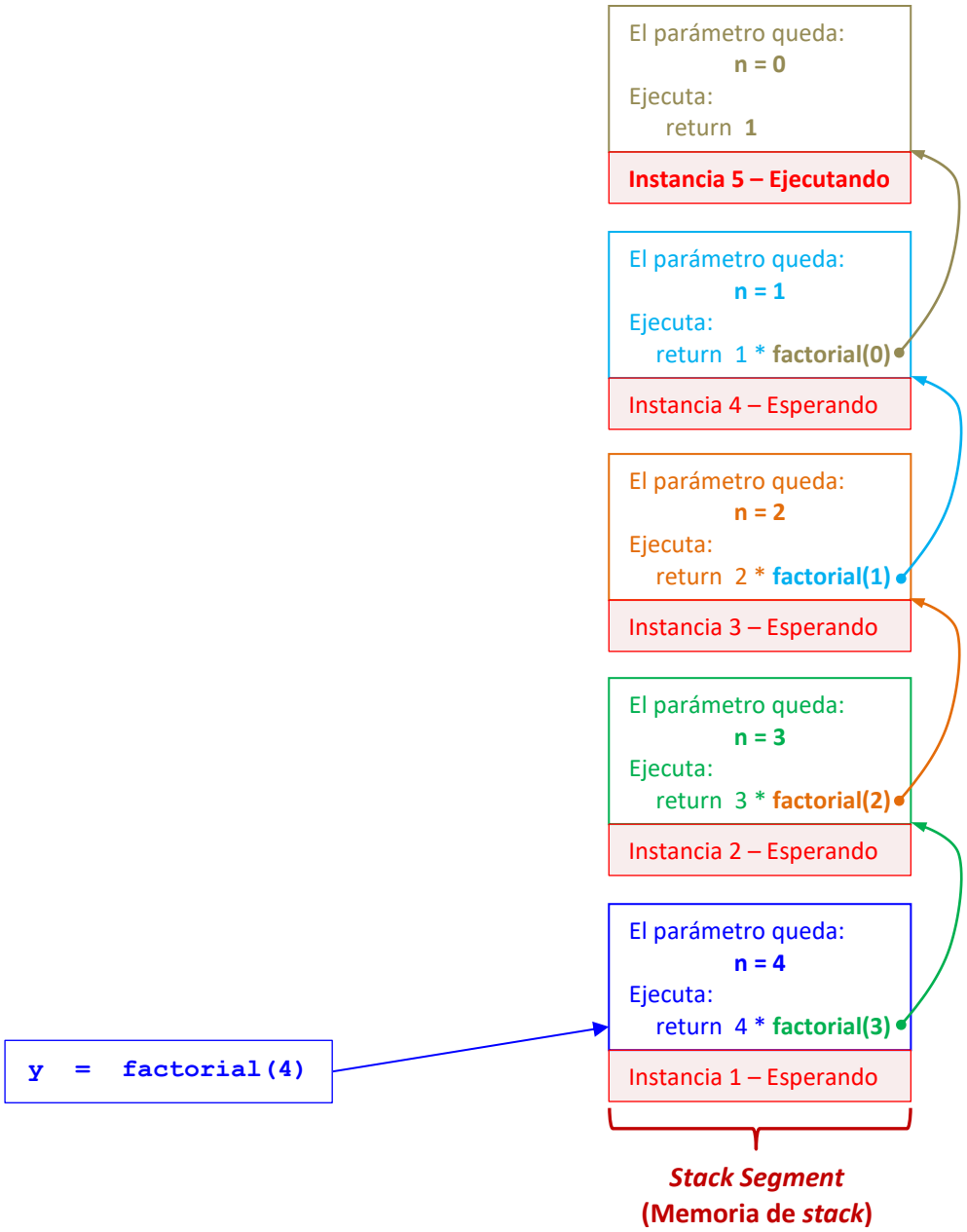
```
return 2 * factorial(1)
```

Y la *cuarta instancia* a su vez lanza una ***quinta instancia*** haciendo:

```
return 1 * factorial(0)
```

con lo que nuestro esquema de memoria de stack queda así:

Figura 4: Memoria de stack luego de cinco invocaciones recursivas a la función *factorial()*.



Observemos con mucha atención la ***quinta instancia de ejecución***: en ella, el valor de *n* es *cero*, y por lo tanto en esta ***quinta instancia*** la condición:

```
if n == 0:
    return 1
```

es evaluada en *verdadero*, ejecutando entonces la instrucción ***return 1***. Entiéndase bien: hasta aquí, había en memoria *cinco instancias* ejecutándose y/o esperando. Si bien se trataba de la misma función invocándose a sí misma varias veces, Python (y todo lenguaje

que soporte recursividad) trata a estas instancias como funciones diferentes que piden memoria por separado.

Ahora bien: cuando alguna de las funciones de esa *cascada recursiva* logra finalizar (como pasa con la ***quinta instancia*** de nuestro último gráfico), comienza a desarrollarse automáticamente un proceso conocido como *proceso de vuelta atrás* o *backtracking*.

La cuestión es simple, aunque un poco extensa de explicar: la función que logra finalizar libera su área de memoria de stack y retorna su valor hacia la instancia de ejecución que originalmente la había llamado. En nuestro caso, la ***instancia 5*** retorna el valor **1** (que es el valor del *factorial de 0*) hacia la ***instancia 4*** (que es la que había pedido el *factorial de 0*).

En la ***instancia 4*** ahora se conoce entonces cuánto vale el *factorial de 0*, por lo cual a su vez puede calcular el valor de *1 * factorial(0)* que es igual a **1**. Cuando la ***instancia 4*** calcula este valor (que es el *factorial de 1*), la ***instancia 4*** termina de ejecutarse retornando el valor **1** a la ***instancia 3*** (que fue la que pidió el *factorial de 1*).

En la ***instancia 3*** se calcula ahora el *factorial de 2*, haciendo *2 * factorial(1)* cuyo valor es **2**. Ese resultado se retorna a la ***instancia 2***, en donde a su vez se calcula *3 * factorial(2)* que vale **6**.

Por fin, el valor **6** es retornado a la ***instancia 1***, donde se calcula el valor *4 * factorial(3)* cuyo valor es **24** y es justamente el *factorial de 4* que se quería calcular originalmente. Ese resultado es devuelto al *punto original de llamada*, y la función *factorial()* termina (¡por fin!) de ejecutarse. Gráficamente, el *proceso de vuelta atrás* o *backtracking* completo puede verse en la *Figura 5* (*página 543*).

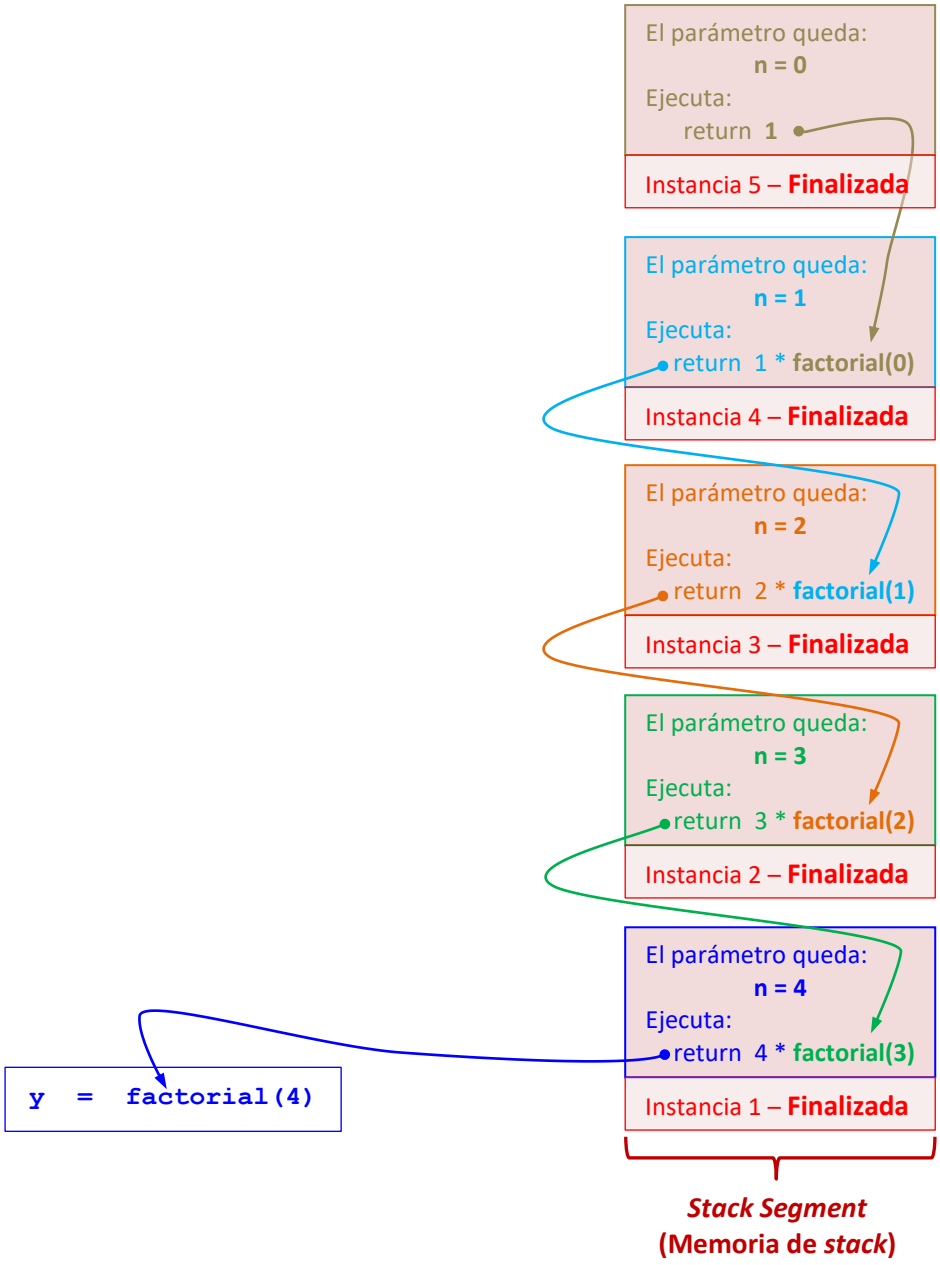
Lo importante de todo esto es que tanto el proceso de asignación de memoria de stack al comenzar el proceso recursivo hacia adelante (o *cascada recursiva*), como el proceso de *vuelta atrás* o *backtracking*, *son gestionados en forma automática por el programa*. Lo único que debe hacer el programador es *plantear en forma correcta la función*, lo cual como ya vimos, implica incluir una condición de corte y efectuar una o mas llamadas recursivas que de alguna forma vayan reduciendo poco a poco el proceso. En nuestro caso, al llamar a *factorial()* enviando como parámetro el valor *n – 1*, se va reduciendo el valor con el cual se trabaja hasta que en alguna instancia ese valor será cero y comenzará el proceso de *vuelta atrás*.

4.] Aplicación elemental de la recursión.

El proceso recursivo implica una cascada de invocaciones a nuevas instancias de la misma función, y luego otro proceso de regreso o vuelta atrás mediante el cual se van cerrando los cálculos que estuviesen pendientes en instancias anteriores.

El hecho es que no siempre resulta evidente la forma en que ocurre ese proceso de vuelta atrás, sobre todo debido a que el mismo es automático e implícito. Pero si el programador tiene en claro el mecanismo, puede aprovecharlo para lograr resolver problemas que de otro modo parecerían muy complicados. Vemos un ejemplo simple pero muy ilustrativo: Supongamos que se desea implementar una función que simplemente tome como parámetro un número *n*, y muestre *n* mensajes en la consola de salida, pero numerados desde 1 hasta *n*. Supongamos también que se nos pide que la función lo haga en forma recursiva, sin usar un ciclo.

Figura 5: Proceso de *vuelta atrás* completo y finalización de la ejecución.



Un primer intento podría verse como sigue:

```
def mostrar01(n):
    if n > 0:
        print('Mensaje numero', n)
        mostrar01(n-1)
```

Evidentemente, si *n* es cero o menor que cero, la función no tiene nada que hacer (ya que se están pidiendo "cero mensajes"). Por lo tanto, esa es la *situación trivial* o *base* que define la condición de corte: la función sólo llevará a cabo algún proceso si *n > 0*. Si *n* es mayor que cero, se muestra el primer mensaje incluyendo en él el valor actual de *n*, *y luego se activa la recursión*: se vuelve a invocar a la función *mostrar01()* pero ahora pasándole como parámetro el valor *n-1*. Al reducir en 1 el valor del parámetro en cada invocación, se garantiza que en algún momento se alcanzará el valor 0 y la cascada recursiva finalizará.

Todo parecería correcto… pero el estudiante quizás ya notó que hay un problema: así planteada, la función mostrará efectivamente una secuencia de *n* mensajes, pero numerados en *forma descendente* (de *n* hacia 1) en lugar de hacerlo en *forma ascendente* (que era lo pedido). Si *n* es 5, por ejemplo, la salida producida por esta función será:

```
Mensaje numero 5
Mensaje numero 4
Mensaje numero 3
Mensaje numero 2
Mensaje numero 1
```

En efecto: cada instancia de ejecución de la función chequea si *n* es mayor a 0. Cuando *n* vale 5 la condición es cierta, y en la rama verdadera la primera instrucción muestra el valor actual de *n*, ***antes*** de la invocación recursiva. Por lo tanto, era de esperar que el primer mensaje mostrado se numere como 5 y no como 1. En cada una de las otras instancias de ejecución ocurre lo mismo: primero se muestra el valor del parámetro *n*, y ***luego*** de lanza la recursión para los mensajes que quedan.

¿Cómo se podría solucionar el problema y hacer que los mensajes salgan numerados de menor a mayor? Sólo debemos recordar que cada vez que se invoca a la función, se almacena en la memoria de stack el valor del parámetro *n*. No es obligatorio mostrar el valor de *n* inmediatamente al entrar en la rama verdadera de la condición: *se puede hacer primero la llamada recursiva*, ir almacenando con eso en el stack la sucesión de valores (sin mostrarlos), y mostrarlos sólo cuando finaliza la cascada recursiva y a medida que se regresa con el proceso de vuelta atrás:

```
def mostrar02(n):
    if n > 0:
        mostrar02(n-1)
        print('Mensaje numero', n)
```

Por curioso que parezca, esta nueva función hace lo pedido y muestra los mensajes numerados en forma ascendente. Lo único que tuvimos que hacer fue *invertir el orden de las instrucciones de la rama verdadera*. Como cada vez que se invoca a la función se almacena en la memoria de stack el valor actual de *n*, pero estos valores se almacenan de forma que el último en llegar es el que queda disponible en la cima del stack, entonces queda en la cima el valor 1 (puede ver la *Figura 4* para recordar la forma en que se van generando las instancias de ejecución).

Como en nuestra segunda versión las llamadas recursivas se hacen antes que las visualizaciones, el resultado es que ***ninguna*** visualización se hará hasta que todas las llamadas recursivas queden almacenadas en el stack… y cuando eso ocurra, la primera instrucción *print()* en ejecutarse será la que corresponda a la cima del stack que es la que contiene al valor *n = 1*. El proceso de vuelta atrás hará que se vayan mostrando los valores desde el stack en orden inverso al de su llenado, y eso es lo que hace que los valores se muestren en orden ascendente.

La primera función *mostrar01()* que hemos analizado, tenía una estructura conocida como *procesamiento en pre-orden* o *procesamiento en orden previo*, que se caracteriza por el hecho de que el proceso que debe realizar la función (en este caso, el *print()*), *se hace antes que la invocación recursiva*. Por otra parte, la función *mostrar02()* tiene una estructura que se conoce como *procesamiento en post-orden* o *procesamiento en orden posterior*: el

proceso a realizar (nuevamente la función *print()*) se hace *después de hacer la invocación recursiva*.

Para finalizar, un detalle: el segmento de memoria que hemos llamado *Stack Segment* se designa con ese nombre por una razón de peso: la palabra en inglés *stack* se traduce como *pila* en español y ese nombre se debe a que en ese segmento los valores que se almacenan van *apilándose* uno sobre otro a medida que se van invocando funciones (con recursión o no) (vuelva a ver la *Figura 4* para observar este efecto). Y como ya vimos en una Ficha anterior, esta forma de almacenar datos se conoce como *esquema de Pila* o de *tipo LIFO* por sus iniciales en inglés: *Last In – First Out* (*Primero en llegar – Último en salir*).

El *Stack Segment* es el segmento de soporte para todos los procesos de invocaciones a funciones de cualquier lenguaje de programación. Cada vez que se invoca a una función, se reserva para ella un espacio en la cima del stack y en ese espacio la función almacena la dirección a la que se debe volver cuando finalice su ejecución, y sus variables locales. Y cuando una función finaliza, se libera el espacio de stack que la misma tenía, quedando en la cima la función desde la cual se invocó a la que acaba de terminar.

Y es natural que funcione como un *apilamiento* de datos: cuando una función termina, el flujo de ejecución del programa debe regresar al punto desde donde fue invocada la que acaba de terminar, y es justamente esa la que en ese momento estará en la cima. El *Stack Segment* constituye así un mecanismo confiable para que el programa recuerde el camino de regreso que debe seguir a medida que las funciones que se invocaron vayan finalizando.

5.] Consideraciones generales.

Llegado a este punto quizá resulte obvio que la recursividad es un proceso algo complicado de entender para quienes recién se acercan a ella. Sin embargo es también cierto que después de un poco de práctica el proceso se asimila sin inconvenientes, sobre todo si en ese período de práctica el lector se toma el trabajo de hacer un seguimiento gráfico de las funciones recursivas que plantea.

Uno de los puntos que más trabajo parece costar a los recién iniciados es el *planteo de la condición de corte* de la función (es decir, la condición para evitar la *recursión infinita*). La clave de este paso es tener en cuenta que lo que se busca determinar con esa condición es si se ha presentado lo que se conoce como un *caso base* o un *caso trivial* para el problema: un caso en el que la recursión no es necesaria y puede resolverse en forma directa (o incluso no haciendo nada) [1].

La determinación de esa condición de corte puede hacerse sin mayores problemas, simplemente respondiendo a la siguiente pregunta:

¿En qué caso o casos el problema que se quiere plantear se resuelve en forma trivial?

La respuesta a esa pregunta, aplicada al problema particular que se está enfrentando evidencia la condición de corte que se buscaba. En el caso del cálculo del factorial, la pregunta sería: ¿en qué caso el factorial de *n* se resuelve en forma trivial, sin necesidad de cálculos ni procesos recursivos? Es obvio que ese caso se da cuando *n* vale cero, pues entonces el factorial vale directamente 1 (uno). Y en nuestra versión recursiva del factorial hemos incluido entonces una condición de corte que comprueba si *n* es 0.

Por otra parte, debe observarse que la recursividad es una herramienta para usar con cautela [3] [4]: si bien es cierto que una función recursiva es extremadamente compacta en cuanto a código fuente y permite además que una definición recursiva sea llevada a una función recursiva prácticamente sin cambios (obsérvese la gran similitud que existe entre la definición algebraica recursiva del factorial y el planteo de la función para calcular el factorial), también es cierto que *la recursividad insume más memoria que la que usaría un proceso cíclico común*.

Esto se debe a que cada *instancia recursiva pide memoria de stack* para esa instancia (como se vio en los gráficos anteriores). Si la *cascada recursiva* fuera muy larga (por ejemplo, para calcular el factorial de un número grande), podría llegar a provocarse un *rebalsamiento*, *desborde* o *sobreflujo* (*overflow*) de memoria de stack. Esto significa que alguna instancia de la cascada de invocaciones *podría no tener lugar en el Stack Segment* para ejecutarse, con lo cual se produciría la interrupción o cancelación del programa que invocó a esa función. La memoria de un computador es un recurso de tamaño limitado, y el *Stack Segment* es parte de la memoria.

Por otra parte, la ejecución de una función requiere *cierto tiempo de procesamiento* desde que esta comienza y hasta que finaliza, que debe contemplarse en la estimación del tiempo total de ejecución de un proceso recursivo completo. En el caso del factorial, la versión *iterativa* (no recursiva, basada en ciclos) tendrá un tiempo de ejecución en relación directa con el tiempo que demore el ciclo *for* en finalizar, y este ciclo depende a su vez del valor de *n*. Por otra parte, la versión *recursiva* hará tantas invocaciones recursivas como sea el valor de *n*, y el tiempo total dependerá del tiempo que lleve terminar de ejecutar cada una. En este caso, se puede asumir que en ambos escenarios se tendrá un *tiempo de ejecución que depende de n en forma directa*, y por lo tanto serían *igualmente aceptables* la versión recursiva y la versión iterativa.

Notemos además que el código fuente de la versión recursiva es simple, directo y compacto. La *complejidad aparente y la estructura del código fuente* de un programa o función es otro de los elementos que se tienen en cuenta para hacer un análisis comparativo entre dos o más soluciones propuestas para un mismo problema. Pero en el caso del factorial, la versión iterativa no es mucho más compleja que la recursiva, aunque es cierto que la recursiva es más directa para comprender.

Si se tienen en cuenta los tres factores generales de análisis comparativo entre algoritmos (*consumo de memoria*, *tiempo de ejecución* y *complejidad de código fuente*) entonces si un programador debe realizar al cálculo del factorial de *n*, debería usar la *versión iterativa* (y ***no*** la *recursiva*). La decisión final de usar la versión *iterativa*, *se basa en este caso en el uso más eficiente de la memoria por parte de la versión iterativa*. El *tiempo de ejecución final está en el mismo orden de magnitud en ambos casos* y *la complejidad del código fuente es aceptable también en ambos casos*.

Por estos motivos, en general *se suele* aconsejar usar la recursividad *sólo cuando sea absolutamente necesario por la naturaleza implícitamente recursiva del problema que se enfrenta*, como es el caso del recorrido de ciertas estructuras de datos como los *árboles* o los *grafos* (que escapan a los alcances de este curso), o la generación de gráficos de *naturaleza fractal* (figuras que se forman combinando versiones más sencillas de las mismas figuras…) que resultaría prácticamente imposible de programar sin recursión desde el punto de vista de la *complejidad del código fuente*.

Algunos programadores experimentados a veces plantean primero la solución recursiva de un problema (para darse una idea de la forma mínima de resolverlo desde el punto de vista de la *complejidad del código fuente*), y luego tratan de convertir ese planteo a una solución no recursiva basada en ciclos (aunque esto no siempre es sencillo de hacer…)

Anexo: Temas Avanzados

En general, cada Ficha de Estudios podrá incluir a modo de anexo un capítulo de *Temas Avanzados*, en el cual se tratarán temas nombrados en las secciones de la Ficha, pero que requerirían mucho tiempo para ser desarrollados en el tiempo regular de clase. El capítulo de *Temas Avanzados* podría incluir profundizaciones de elementos referidos a la programación en particular o a las ciencias de la computación en general, como así también explicaciones y demostraciones de fundamentos matemáticos. En general, se espera que el alumno sea capaz de leer, estudiar, dominar y aplicar estos temas aún cuando los mismos no sean específicamente tratados en clase.

a.) Caso de análisis: La Sucesión de Fibonacci.

Para permitir un mayor dominio de las técnicas de programación recursiva, y sus ventajas y desventajas, analizaremos ahora un conocido problema: el cálculo del término *n-ésimo* de la *Sucesión de Fibonacci*.

Problema 59.) *La Sucesión de Fibonacci es una secuencia de valores naturales en la que cada término es igual a la suma de los dos anteriores, asumiendo que el primer y el segundo término valen 1. En algunas fuentes se parte de suponer que los dos primeros valen 0 y 1 respectivamente, pero eso no cambia el espíritu de la regla ni la explicación que sigue. Formalmente, el cálculo del término n-ésimo F(n) de la sucesión se puede entonces definir así (y note que la definición planteada es directamente recursiva):*

Término n-ésimo de Fibonacci

$$F(n) \quad (n \text{ entero}, n >= 0) \begin{cases} = 1 & (\text{si } n = 1 \text{ ó } n = 0) \\ = F(n-1) + F(n-2) & (\text{si } n > 1) \end{cases}$$

Se pide desarrollar un programa que permita calcular el valor del término enésimo de esta sucesión, cargando n por teclado.

Discusión y solución: Si bien el enunciado muestra la definición en forma directamente recursiva, podemos diseñar un par de funciones que calculen el valor del término n-ésimo de Fibonacci: una en forma iterativa y la otra en forma recursiva para luego poder comparar ambas soluciones (vea el proyecto *F[26] Recursión* que acompaña a esta ficha):

```
# versión iterativa
def fibonacci01(n):
    ant2 = ant1 = 1
    for i in range(2, n + 1):
        aux = ant1 + ant2
        ant2 = ant1
        ant1 = aux
    return ant1
```

```
# versión recursiva
def fibonacci02(n):
    if n <= 1:
        return 1
    return fibonacci02(n - 1) + fibonacci02(n - 2)
```

La *versión iterativa* inicializa las variables *ant2* y *ant1* con el valor 1, y usa un *for* para recorrer el intervalo *[2, n]*. En cada repetición suma los valores de *ant1* y *ant2* para obtener el siguiente término en forma progresiva, actualizando también los valores de *ant1* y *ant2* para quedarse siempre con los dos últimos calculados. Cuando el *for* finaliza, el último número almacenado en *ant1* es el que corresponde al término n-ésimo pedido.

La *versión recursiva* es la aplicación directa de la fórmula dada en la definición: la condición de corte comprueba si *n* es menor o igual a 1. En caso afirmativo, se asume que *n* vale 0 o vale 1 y en ese caso, se retorna 1 como se pide en la definición. Si *n* fuese mayor a 1, se aplica la fórmula en forma recursiva y se retorna la suma de los dos anteriores a *n*.

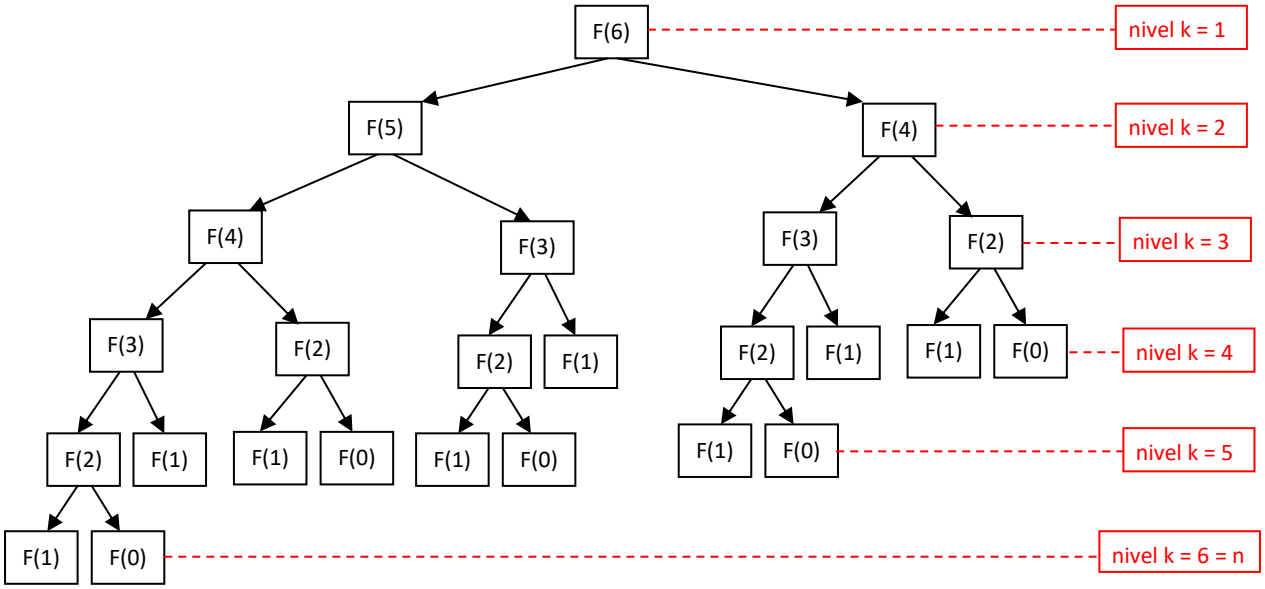
Está claro que la *versión iterativa* se *ejecuta en un tiempo proporcional a n*, en forma similar al cálculo del factorial: a medida que *n* aumenta, en la misma proporción aumenta el tiempo de ejecución ya que depende de cuántas repeticiones haga el ciclo *for*.

El *espacio de memoria empleado por la versión iterativa* es siempre el mismo: un puñado de variables locales, que no cambia aunque *n* sea mayor o menor.

Si se comienza un análisis comparativo, está muy claro que la *versión recursiva* es evidentemente más simple de entender y más compacta en cuanto a *complejidad de código fuente* que la *versión iterativa*. Pero el análisis del *tiempo de ejecución* (y en menor medida, del *consumo de memoria*) de la *versión recursiva* no es tan simple debido a la presencia de ***dos*** invocaciones recursivas.

Para intentar una aproximación intuitiva, podemos mostrar un gráfico de la estructura de la *cascada de llamadas* que implica la *versión recursiva*. Si suponemos que se desea calcular el valor de *fibonacci02(6)* (que abreviaremos como *F(6)* en el gráfico, para simplificar), se puede ver que se producirá un *árbol de llamadas* similar al siguiente (aunque comprenda, *no todo el árbol estará contenido en la memoria al mismo tiempo*):

Figura 6: Esquema del *árbol de llamadas recursivas* para *F(6)*.



Como se dijo, este es el *árbol completo* de invocaciones pero *nunca* estará totalmente contenido en memoria: de hecho, *sólo cada una de las ramas de este árbol* estará completamente en el *stack segment* en un momento dado, por lo que se puede ver que la ocupación de memoria del proceso en el peor caso, *equivale a la cantidad de niveles de la rama más larga del árbol* (en este caso, la de la extrema izquierda). Puede verse que para calcular el término *F(6)* la rama más larga tiene exactamente *6(seis) niveles*, por lo que es simple comprobar que *F(n)* requerirá *n* niveles, y de allí que el consumo de memoria será *proporcional a n: a medida que n aumente, lo hará también el espacio de memoria de stack usado, en proporción directa con n* (esto en sí mismo *ya es malo* en comparación con la *versión iterativa*… cuyo espacio de memoria se mantenía constante sin importar qué tan grande sea *n*…)

Pero los problemas verdaderamente comienzan cuando se analiza el *tiempo de ejecución*. En este caso, **sí** importa el *árbol completo*, pues debemos calcular *cuánto tiempo llevará hacer el proceso total*. De nuevo, el hecho técnico de invocar a una función lleva un tiempo constante, pero a ese tiempo debe sumarse el que lleve completar el proceso contenido en ella. En nuestra *versión recursiva*, el proceso consta básicamente de una suma que también ejecuta en tiempo constante… y por lo tanto, *todo queda reducido a saber cuántas veces se invoca la función a lo largo de toda la ejecución*.

Se puede sospechar que será un número elevado, ya que sólo para *F(6)* tenemos 25 invocaciones (el número de rectángulos o *nodos* del árbol anterior), pero además podemos ver que muchas de esas invocaciones se hacen para realizar un proceso que ya había sido completado antes: sin contar los cálculos triviales para *F(1)* y *F(0)*, vemos que dos veces se calcula *F(4)*, tres veces se calcula *F(3)* y cuatro veces se calcula *F(2)*. El árbol no sólo es denso de por sí, sino que además se pierde tiempo re-calculando valores que ya se tenían…

En definitiva: ¿cuánto tiempo llevará el proceso completo para *F(n)* en la *versión recursiva*? Intuitivamente, si se observa el árbol de llamadas recursivas para *F(6)*, puede verse con claridad que el número total de invocaciones a medida que se completa cada nivel aumenta en forma más o menos predecible:

Nivel	**Invocaciones hasta ese nivel (incluido)**
k = 1	$1 = 2^1 - 1$
k = 2	$3 = 2^2 - 1$
k = 3	$7 = 2^3 - 1$
k = 4	$15 = 2^4 - 1$
k = 5	23
k = 6	25

Al menos hasta el nivel $k = 4$, todos los niveles están completos y el número de llamadas acumulado hasta allí varía en *forma exponencial*. Y con ello podemos empezar a suponer que el tiempo de ejecución será algo de la forma b^n para alguna base *b* constante y mayor a 1.

Puede probarse que este resultado intuitivo es efectivamente correcto, y veremos que si un algoritmo/programa tiene un tiempo de ejecución exponencial (como en este caso) entonces estamos en serios problemas… Si el tiempo de ejecución de un algoritmo cambia en relación exponencial a medida que *n* crece, entonces ese algoritmo sólo será aplicable cuando *n* sea realmente muy pequeño.

Si *n* toma un valor de entre 30 y 35 la cantidad de pasos que el algoritmo supone es tan grande que incluso una computadora moderna muy veloz comenzará a verse en problemas para terminar el proceso y entregar el resultado en un tiempo prudente. Puede probar el programa *test01.py* del proyecto *[F26] Recursión* que acompaña a esta ficha, y cargar los valores 30, 31, 32, 33, 34 y 35 para darse una idea de lo que ocurre con pequeñas variaciones en el valor de *n*…

Para valores de *n* mayores a 35 y hasta 40 el proceso se torna penosamente lento, de forma que ya para n = 40 posiblemente tenga que detener el programa en forma manual (y todo mientras la *versión iterativa* finaliza en un instante).

Para *n = 50* o *n = 60* el *proceso recursivo es definitivamente inaceptable* en términos prácticos: la computadora podría estar miles o centenares de miles años haciendo el cálculo y buscando el resultado (que la *versión iterativa* obtiene en unos pocos milisegundos). Y si fuese *n = 100*, entonces será hora de rendirse: el *programa recursivo* ejecutará *durante todo el tiempo de vida que le queda al universo, y no alcanzará a calcular el valor de F(100)*. Ejecute el programa *test01.py* ya mencionado, e inténtelo si no lo cree…

Veremos más adelante en este mismo curso, que si un problema *sólo admite soluciones que se ejecutan en tiempo exponencial*, entonces ese problema se dice ***intratable*** y constituye todo un desafío para las Ciencias de la Computación. Note que el cálculo del término *n-ésimo de Fibonacci* ***no es un problema intratable***, ya que se conoce al menos un *algoritmo de tiempo de ejecución no exponencial* para resolverlo: nuestra ya conocida *versión iterativa*.

A modo de conclusión:

- La recursividad es una herramienta muy útil para el planteo de algoritmos, pero debe ser usada con cuidado y con conocimiento adecuado en cuanto a la forma de estimar el uso de recursos de tiempo y memoria.
- En general, desde el punto de vista de la *complejidad del código fuente*, la recursión permitirá escribir programas más compactos, más simples de comprender, y más consistentes con la definición formal del problema que en un planteo iterativo. Pero si se toma a la ligera el consumo de tiempo y memoria usada, el programa obtenido podría simplemente ser inaceptable en la práctica.
- Algunos programadores suelen ensayar soluciones recursivas para un problema, dado que en términos de complejidad de código fuente podría resultar más simple; y si luego descubren que esas soluciones son poco eficientes proceden a eliminar la recursión y convertir el programa en un proceso iterativo (aunque como dijimos, no siempre esto es simple de hacer)
- ¿¿¿Obtuvo ya el resultado de *F(100)*???
- A la luz de todo lo dicho, podría alegarse que entonces no es conveniente aplicar recursividad *en ningún caso*. Sin embargo, esta postura pesimista está lejos de ser cierta: si la recursión se aplica en la forma apropiada, puede dar lugar a soluciones muy eficientes (al menos en cuanto a tiempo de ejecución) y sorprendentemente compactas en cuanto a complejidad de código fuente. Ciertas estrategias de resolución de problemas (como la que se conoce como *Divide y Vencerás* (*DyV*)) se basan en un esquema recursivo aplicado con ciertas restricciones, y de acuerdo a esas restricciones pueden resolverse distintos problemas en forma muy eficiente respecto del tiempo de ejecución.
- La estrategia conocida como *Backtraking* (o *Vuelta Atrás*) es otra estrategia de planteo de algoritmos que se monta sobre un proceso recursivo para ayudar a explorar diversas

soluciones a un problema y quedarse con la mejor a medida que las instancias recursivas van finalizando. Muchos de los problemas que se resuelven con *Backtraking*, serían casi imposibles de resolver sin esa técnica.

- Sin embargo, el enfoque pesimista todavía podría alegar que las estrategias *DyV* y *Backtraking* son *técnicas diferentes* y con *entidad propia* para el planteo de algoritmos. Se podría alegar que la *recursividad aplicada en forma directa* y sin las restricciones propias de la estrategia *DyV* o el *Backtraking* no es práctica. Y de nuevo, la respuesta es que eso no es cierto. Como dijimos, muchas aplicaciones propias del recorrido de estructuras de datos no lineales (como *árboles binarios*, *árboles n-arios* y *grafos*) se resuelven en forma natural y eficiente usando *recursividad directa*, con el adicional de que esos procesos resultan muy compactos y simples de comprender en cuanto a código fuente.
- ¿¿¿Obtuvo ya el resultado de *F(100)*???
- Y por supuesto, existen áreas específicas en las ciencias de la computación en las que el empleo de la recursión en forma directa o indirecta (la *recursión indirecta o mutua*, es aquella situación en la que una función *A()* invoca a otra *B()*, pero luego *B()* invoca a su vez a *A()*) es prácticamente obligado, pues de otro modo los algoritmos serían absurdamente difíciles de plantear. Una de esas áreas es la de la generación de *figuras fractales*. En casos así, el factor de eficiencia preponderante es evidentemente, la *complejidad del código fuente*.
- Ya puede cancelar el programa del cálculo de *F(100)*. A menos que esté dispuesto a esperar hasta el fin de los tiempos….

Créditos

El contenido general de esta Ficha de Estudio fue desarrollado por el *Ing. Valerio Frittelli* para ser utilizada como material de consulta general en el cursado de la asignatura *Algoritmos y Estructuras de Datos* – Carrera de Ingeniería en Sistemas de Información – UTN Córdoba, en el ciclo lectivo 2019.

Actuaron como revisores (indicando posibles errores, sugerencias de agregados de contenidos y ejercicios, sugerencias de cambios de enfoque en alguna explicación, etc.) en general todos los profesores de la citada asignatura como miembros de la Cátedra, y particularmente los ingenieros *Analía Guzmán*, *Karina Ligorria y Cynthia Corso* que realizaron aportes de contenidos, propuestas de ejercicios y sus soluciones, sugerencias de estilo de programación, y planteo de enunciados de problemas y actividades prácticas, entre otros elementos.

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] Python Software Foundation, "Python Documentation," 2018. [Online]. Available: https://docs.python.org/3/.

[3] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.

[4] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.