1. Caso

Unidad Didáctica X

Caso

*El trabajo diario en **SI Andalucía** puede parecer monótono desde fuera. Siempre ante el ordenador, un día tras otro, resolviendo los problemas que dan las aplicaciones e intentando conseguir programas útiles para los clientes. Alguien podría pensar que es algo aburrido, pero si le preguntan a **Víctor** podrán observar que él lo ve de otro modo.*
*Realmente cada **aplicación** es un reto al que se enfrentan como un verdadero equipo en el que cada cual tiene asignada una tarea concreta. En cada uno de los trabajos que realizan aprenden algo, sobre la programación o sobre las diferentes formas de trabajar en cualquiera de los ámbitos profesionales de la sociedad actual.*
*El encanto de la **programación** está precisamente en la realización de un profundo estudio de cada tarea a programar, lo que nos permite ver el problema desde una posición privilegiada y conseguir de este modo enriquecernos como individuos cada vez más preparados.*
*En cada aplicación que se programa, se adquieren nuevos conocimientos y en algunas de ellas se utilizan técnicas que nunca antes se habían utilizado. Esto le está ocurriendo a **Víctor** que cuando cree que lo ha visto todo y lo único que queda es practicar para mejorar sus capacidades, aparecen cosas nuevas y esto no acaba nunca.*

Unidad Didáctica X

Concepto de recursividad

*Algo así le ha ocurrido con la **recursividad**, un mecanismo que parece que no aporta nada y que le resulta muy complicado de entender.*
***Carmen** le dice que hay que tener cuidado, porque si se utiliza de forma incorrecta puede dejar al ordenador realizando operaciones de forma interna indefinidamente, sin posibilidad de concluirlas, lo que supondrán el inevitable "cuelgue" del equipo.*

*Pero es un mecanismo muy **potente** con el que se consigue que determinados programas mejoren su rendimiento y se escriban de forma más simple y clara, facilitando la actualización y modificación del código. Como siempre a **Víctor** le gusta que **Carmen** le explique las cosas. Siempre lo hace de forma muy clara y sencilla, con ejemplos claros basados en su experiencia. Normalmente suele acertar y conseguir que **Víctor** entienda el concepto.*

¿Has oído alguna vez la máxima de que en la definición no debe usarse lo que se intenta definir? Parece lógico **que si me piden definir lo que es un ordenador, decir que es una especie de ordenador que permite**..., no aporta mucha información, no es la mejor manera de empezar la definición.

Aunque esa máxima es muy lógica y sensata a la hora de hablar en nuestra vida diaria, ya puedes ir olvidándola en el caso de la programación. Al igual que en matemáticas, donde existen definiciones por recurrencia:

- Se define alguna propiedad para el primer número Natural, cero.
- Suponiendo que la propiedad está definida para un número Natural cualquiera, llamado n, usamos esa definición para definirla también para el siguiente número Natural, n+1
- De esta forma queda definida para todo el conjunto de los números Naturales.

En programación, usamos el término **recursividad para referirnos a procedimientos o métodos que contienen en su implementación llamadas a sí mismo, es decir un método es recursivo si para ejecutarse se llama a sí mismo**.

El aspecto de la declaración de un método recursivo es algo como lo que sigue:

```
<modificadores y tipo devuelto>metodoRecursivo(<lista de argumentos>){
      sentencia;
         ...
      sentencia;
         metodoRecursivo(<lista de argumentos>);
         sentencia;
       ...
       sentencia;
}
```

Hemos resaltado en negrita la llamada que el método se hace a sí mismo.

Pero, si para ejecutar a un método hay que hacer una llamada al mismo método que a su vez hará una llamada al mismo método, y así sucesivamente, ¿no habremos entrado en un **bucle infinito** que hará que nunca terminemos?

No, si las cosas se hacen bien, aunque evidentemente, es posible hacerlas mal.

¿Qué debemos tener en cuenta a la hora de saber si un método recursivo está bien definido?

- Además de la llamada al propio método, **deben existir otras sentencias en la definición del**

método que establezcan al menos un caso base, es decir, un caso para el que se conoce la solución, y para el que la llamada al método puede devolver un valor o terminar la ejecución sin necesidad de volver a invocar de nuevo al propio método.

- **Cada nueva llamada al propio método debe reducir el problema**, es decir, debe acercarnos más al caso base.
- **La solución debe ser correcta para los casos no base**. Es decir, debemos ser capaces de expresar correctamente la solución de un caso a partir de la soluciones de casos menos complejos, para poder construir la solución de cualquier caso a partir de las soluciones del caso base.

Autoevaluación

¿Qué debemos tener en cuenta a la hora de saber si un método recursivo está bien definido? Señala la afirmación correcta.

- a) Debemos ser capaces de expresar correctamente la solución de un caso a partir de la soluciones de casos menos complejos, para poder construir la solución de cualquier caso a partir de las soluciones del caso base.
- b) Debemos ser capaces de expresar correctamente la solución de un caso a partir de la soluciones de casos más complejos
- c) La solución puede no ser correcta para los casos no base.
- d) Todas las anteriores son correctas.

Comprobar

Respecto a la recursividad, señala la afirmación correcta.

- a) La solución recursiva es más elegante, en el sentido de que genera un código más breve, y más simple de entender, es muy intuitiva.
- b) La recursividad es una poderosa herramienta en programación y ofrece una alternativa a las soluciones iterativas complejas de algunos algoritmos.
- c) La afirmación a y b son ciertas.
- d) Ninguna de las anteriores es cierta.

Comprobar

2.1. Ejemplo

Unidad Didáctica X

Ejemplo

El **concepto** suele entenderse mejor con el **ejemplo** del cálculo del factorial de un número. El factorial de un número entero positivo n se representa como n! (que se lee como "n factorial")

Sabemos que dado un número entero positivo, el factorial se puede definir de dos formas:

Definición Iterativa

Como el producto de todos los números enteros comprendidos entre 1 y el propio número:

n!= n*(n-1)*(n-2)*...*3*2*1. Además, por convenio, para extender la definición también al cero, se define **0!=1**

Para calcular el factorial de esta forma, tenemos que repetir un ciclo en el que se vaya acumulando el producto del número por el anterior, hasta que lleguemos a uno.

Definición Recursiva

Como el producto del propio número por el factorial del número anterior.

Es evidente que (n-1)! = (n-1)* (n-2)*...*3*2*1, lo que nos lleva a poder escribir que **n!=n*(n-1)!, siendo además 0!=1** igual que antes. Acabamos de hacer una definición por recurrencia del factorial de un número. "El factorial de un número vale 1 si el número es cero, y el producto del propio número por el factorial del anterior, en otro caso."

Para calcular el factorial de esta forma, es como si le encargáramos la tarea a una persona, que no sabe calcular el valor final del factorial de n, pero sabe que es n * (n-1)!, y le pregunta a otra persona el valor de (n-1)! Esta persona, tampoco sabe calcular el valor final de (n-1)!, pero sabe que es (n-1)*(n-2)!, y le pregunta a su vez a otra persona el valor de (n-2)!, y así sucesivamente hasta que llegamos a preguntarle a una persona el valor de 0! Y esa persona responde que 0! =1. A partir de ese momento, la persona que le preguntó tiene toda la información para calcular 1!, lo calcula, y le responde a la persona que le preguntó, que completa el cálculo de 2!, y así sucesivamente, hasta llegar a la persona que calcula (n-1)!, que completa el cálculo y le proporciona su valor a quien le preguntó, que multiplica ese valor por n y obtiene el valor de n!.

DEMO: Visualiza los estados de la pila al usar el algoritmo Factorial

Al igual que tenemos dos definiciones del factorial, una iterativa y otra recursiva, el problema lo podemos resolver mediante un método que sea iterativo, ajustándose a la primera definición o mediante un método recursivo, que se ajuste a la segunda. La solución recursiva en este caso es más elegante, en el sentido de que genera un código más breve, y más simple de entender, es muy intuitiva.

La recursividad es una poderosa herramienta en programación y ofrece una alternativa a las soluciones iterativas complejas de algunos algoritmos.

¿Cuáles son entonces las razones para usar la recursividad?

1. **Los problemas resultan más fáciles de resolver que con estructuras iterativas.**
2. **Proporciona soluciones más simples.**
3. **Proporciona soluciones elegantes.**

Autoevaluación

¿Qué debemos tener en cuenta a la hora de saber si un método recursivo está bien definido? Señala la afirmación correcta.

a) Además de la llamada al propio método, deben existir otras sentencias en la definición del método que establezcan al menos un caso base.
b) Cada nueva llamada al propio método debe reducir el problema
c) La solución debe ser correcta para los casos no base.
d) Todas las anteriores son correctas.

Comprobar

Respecto a programación, ¿cómo usamos el término recursividad? Señala la afirmación correcta.

a) Para referirnos a la propiedad de reutilizar las variables y métodos.
b) Para referirnos a procedimientos o métodos que contienen en su implementación llamadas a otros métodos, es decir un método es recursivo si al ejecutarse llama a otros métodos.
c) Para referirnos a procedimientos o métodos que contienen en su implementación llamadas a sí mismo, es decir un método es recursivo si para ejecutarse se llama a sí mismo.
d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica X

Recursión frente a iteración

*A **Victor** le parece que esto de la recursividad son ganas de complicar las cosas, ya que todos los ejemplos que le ha puesto **Carmen** se pueden programar utilizando otras instrucciones, sin necesidad de que un método se llame a sí mismo.*
Eso es cierto pero hay casos en los que es posible prescindir del complejo anidamiento de varios bucles y sustituirlo por una llamada recursiva de modo que se simplifique el código y resulte más sencilla su programación.
***Carmen** dice que apenas utiliza la recursividad, pero hay algunas tareas que habitualmente las emplea con métodos recursivos; porque ya las tiene programadas y porque simplifica el código haciéndolo más fácil de entender. **Carmen** dice que puede programar utilizando en la mayoría de los casos instrucciones iterativas, pero debe tener en cuenta que para programar no se debe renunciar a nada.*

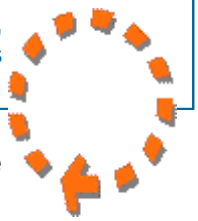
En el caso del factorial hemos visto que había dos posibles soluciones, una iterativa y otra recursiva.

¿Siempre va a existir una solución iterativa o hay casos en los que obligatoriamente tendremos que usar recursividad?

No sólo no existen casos en los que sea obligatoria, sino que **siempre existe una solución iterativa**, **y ésta siempre será más eficiente**, es decir, necesitará consumir menos recursos de memoria y de CPU, y obtendrá la solución en menos tiempo.

Lo que ocurre es que a veces, por la naturaleza recursiva del problema, no es fácil encontrar la solución iterativa, o resulta complicada y difícil de entender.

La recursividad se debe usar cuando sea realmente necesaria, es decir, cuando no exista una solución iterativa simple. Cuando las dos soluciones son fácilmente expresables, siempre será preferible la solución iterativa.



Por otro lado, la **recursividad requiere hacer una asignación dinámica de memoria,** que no es posible en todos los lenguajes de programación. (Sí lo es en Java y en otros muchos).

PARA SABER MÁS:
Si deseas conocer más detalles sobre la asignación estática y dinámica de memoria en programación, visita este interesante enlace donde se explica de una forma bastante clara. Nota. La asignación estática y dinámica de memoria está definida a partir de la página 23 del documento.
Organización de Memoria en Tiempo de Ejecución **[Versión en caché]**

Autoevaluación

Señala la afirmación correcta. Respecto a la solución interactiva, podemos afirmar que:

- a) Con una solución iterativa sólo tendremos un método ejecutándose, por lo que sólo necesitaremos memoria para una copia de cada una de las variables que usa ese método.
- b) Con una solución iterativa no hay que hacer ninguna operación de almacenar en la pila los datos de la ejecución para posteriormente reanudarla.
- c) Las soluciones iteractivas requieren mucho menos tiempo de CPU y de memoria que las recursivas.
- d) Todas las anteriores son correctas.

Comprobar

Señala la afirmación correcta. Respecto al método recursivo, podemos afirmar que:

a) Cada vez que se hace una nueva llamada al método recursivo, es necesario guardar en la pila (stack) todos los datos de la llamada anterior que aún no ha terminado.

b) Cada vez que se hace una nueva llamada al método recursivo, es necesario almacenar en la memoria todos los valores de los registros del microprocesador.

c) Las soluciones recursivas requieren mucho más tiempo de CPU y de memoria que las iterativas.

d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica X

¿Recursivo o iterativo?

Cada vez que se hace una nueva llamada al **método recursivo**, es necesario guardar en la pila (stack) todos los datos de la llamada anterior que aún no ha terminado (todas las variables de memoria que tenga definidas el método), lo que se conoce como **entorno volátil** de esa ejecución.

Esto incluye almacenar en la memoria todos los valores de los registros del microprocesador, para poder restablecerlos posteriormente y continuar por el mismo punto de la ejecución de esa llamada que lo habíamos dejado.

Como puede haber muchas llamadas sucesivas, éstas se van almacenando en la pila, de forma que se irán retirando de la pila en el orden contrario a como se introdujeron.

El último en entrar es el primero en salir. (Estructura LIFO: Last-IN, First-Out)

Todo ese tiempo de CPU dedicado a:

- guardar los datos de cada ejecución en la pila,
- cargar los de la nueva llamada para cuando al final ésta termine
- ir de nuevo sacando esos valores de la pila en orden contrario a como se introdujeron
- para restaurarlos en la CPU y completar la llamada,

es la causa responsable de que las soluciones recursivas requieran mucho más **tiempo** de cpu y mucha más memoria.

Con una solución **iterativa** sólo tendremos un método ejecutándose, por lo que sólo necesitaremos memoria para una copia de cada una de las variables que usa ese método. Por otro lado, no hay que hacer ninguna operación de almacenar en la pila los datos de la ejecución para posteriormente reanudarla, ya que la llamada al método iterativo se ejecuta hasta finalizar, sin tener que intercambiarse con ninguna otra llamada.

En este sentido, como veremos a continuación en el apartado siguiente, el factorial no es el mejor ejemplo que se puede mostrar de uso adecuado de una solución recursiva, ya que existe una solución iterativa igualmente clara y simple, que siempre va a ser más eficiente.

PARA SABER MÁS:
En este enlace podrás profundizar sobre la programación recursiva e iterativa en java.
Programación Recursiva en Java **[Versión en caché]**

Autoevaluación

Señala la afirmación correcta. Respecto a la recursividad frente a la iteración, podemos afirmar que:

a) Si existe una solución recursiva a un problema, no siempre existe una solución iterativa.
b) A veces, por la naturaleza recursiva del problema, no es fácil encontrar la solución iterativa, o resulta complicada y difícil de entender.
c) La solución recursiva necesitará consumir menos recursos de memoria y de CPU, y obtendrá la solución en menos tiempo.
d) Todas las respuestas anteriores son correctas.

Comprobar

¿Cuáles son entonces las razones para usar la recursividad? Señala la afirmación correcta.

a) Los problemas resultan más fáciles de resolver que con estructuras iterativas.
b) Proporciona soluciones más simples.
c) Proporciona soluciones elegantes.
d) Todas las anteriores son correctas.

Comprobar

Unidad Didáctica X

Ejemplos de uso de la recursividad

*Como era de esperar **Víctor** aún no está convencido de que esto sea una herramienta de programación que realmente facilite el trabajo, más bien lo ve como algo que dificulta bastante el diseño de algoritmos, aunque reconoce que la codificación posterior se puede simplificar considerablemente.*

*Y **Carmen** decide pasar a la acción y demostrarle que este mecanismo puede ser útil en determinadas situaciones.*
*Para ello le plantea algunos **ejemplos** en los que el uso de la recursividad facilita y mejora la programación. Le advierte que los primeros son interesantes para entender el concepto, aunque no sean adecuados para utilizar la recursividad como mejor opción frente a la iteración. **Carmen** considera que le va a venir muy bien a su compañero practicar un poco con estos ejemplos y el entorno de programación en el que debe trabajar durante los próximos meses, de este modo su aprendizaje será más sólido y rápido.*

¿Verdad que algún ejemplo de programa recursivo ayudaría a entender mejor estos conceptos?

Hemos venido contando las características a tener en cuenta sobre la recursividad, su definición, y algunas consideraciones generales sobre su uso, pero aún no hemos visto ningún ejemplo. ¡Va siendo hora de remediarlo!

Con los ejemplos de los apartados siguientes pretendemos centrar tu atención sobre los siguientes asuntos:

- Hay **ejemplos que ayudan por su sencillez y claridad a comprender el concepto de recursividad**, pero que no son adecuados para resolver recursivamente, porque hay una solución iterativa simple que es igualmente clara y más eficiente (factorial, Fibonacci y máximo común divisor)
- Hay **ejemplos en los que, a pesar de que existe la solución iterativa, ésta es difícil de imaginar, y la recursividad aporta una solución simple y elegante**. Estos casos serán apropiados para el uso de recursividad. (Torres de Hanoi, Ordenación por QuickSort)
- Hay un tipo especial de problemas, denominados **algoritmos de Vuelta atrás o backtraking,** que consisten en explorar de forma sistemática todos los caminos posibles para alcanzar la solución, y si algún camino nos lleva a un punto en el que no es posible alcanzar la solución deseada, se vuelve atrás hasta alcanzar un punto en el que es posible continuar explorando nuevas alternativas. Es el caso de problemas para encontrar la salida de un laberinto, o para calcular los movimientos de una ficha en el tablero de ajedrez

Te recomendamos que compiles y ejecutes todos los ejemplos usando el **IDE NetBean**s, con el propósito de que te vayas familiarizando con su uso, y que a partir de ahora, cualquier programa que escribas lo hagas usando ese entorno de desarrollo.

Unidad Didáctica X

Factorial de un número

Éste es el primer ejemplo de recursividad que vamos a ver. ¿Por qué lo elegimos como el primero?

Porque es el más simple de todos, y el que más ayuda a aclarar los conceptos.

Presentaremos tanto una solución iterativa, que ya se te pedía en un ejercicio de unidades anteriores, como una solución recursiva.

Primero te recomendamos que veas el código de la versión iterativa. Como puedes observar, esta solución es totalmente clara, y al ser más eficiente, será preferible a la solución recursiva.

Descarga el archivo FactorialIterativa.java

A continuación aparece el enlace a la versión recursiva. Como ves, es una solución elegante, que ayuda a comprender el concepto de recursividad, pero que no aporta claridad ni sencillez, por lo que al ser menos eficiente que la solución iterativa, no sería adecuado usar recursividad en este caso.

Descarga el archivo FactorialRecursiva.java

Unidad Didáctica X

Función Fibonacci

¿Empiezas a ver claras las diferencias entre un buen uso de la recursividad y un mal uso? El ejemplo de la sucesión de Fibonacci pone de manifiesto más aún, esas diferencias.

Al igual que en el caso del factorial, existe una solución iterativa que es bastante sencilla y clara, por lo que la recursividad lo único que aporta es un mayor parecido con la definición de la función. Pero en este caso tenemos el agravante de que cada llamada a la función genera dos nuevas llamadas recursivas, con lo que la eficiencia decrece considerablemente. Se produce una explosión exponencial del número de llamadas recursivas, repitiéndose además cálculos de forma innecesaria. Así, resulta aún más desaconsejable el uso de la recursividad en este caso.

La función Fibonacci calcula el valor del término n-ésimo de la sucesión de Fibonacci, en la que el primer término es 0, el segundo 1 y a partir de ahí, cada nuevo término se calcula como la suma de los dos inmediatamente anteriores. Puede también inicializarse los dos primeros términos con otros valores, pero lo que no cambia es la forma de obtener los sucesivos términos.

Por tanto, la función Fibonacci se puede definir como:

- Fibonacci(n) = n, si n=0 ó n=1
- Fibonacci(n)= Fibonacci(n-1) +Fibonacci(n-2), si n >1

Esta definición de la función es muy clara e intuitiva, y el algoritmo recursivo que la implementa es muy sencillo de hacer, pero muy ineficiente. Si intentas calcular el término 40 de la sucesión, verás cómo tarda un tiempo considerable en resolverlo, y si introduces un valor más alto, posiblemente tendrás la sensación de que el ordenador se ha bloqueado, debido a lo que tarda. Ese tiempo, no obstante, será mayor o menor dependiendo de la potencia de tu ordenador.

Descarga el archivo FibonacciRecursiva.java

DEMO: Visualiza las llamadas producidas en el cálculo recursivo de la serie Fibonacci

A continuación presentamos el código de la solución iterativa, que también es claro y fácil de programar, aunque quizás no tan elegante, pero desde luego es mucho más eficiente. Al ejecutarlo comprueba el tiempo que tarda en ejecutarse para el caso de que el número sea 100. Prácticamente es instantáneo. Sin embargo, la solución recursiva, ya tardaba mucho más de lo razonable para números mayores de 40. Claramente, la solución iterativa es mucho más eficiente.

Descarga el archivo FibonacciIterativa.java

Unidad Didáctica X

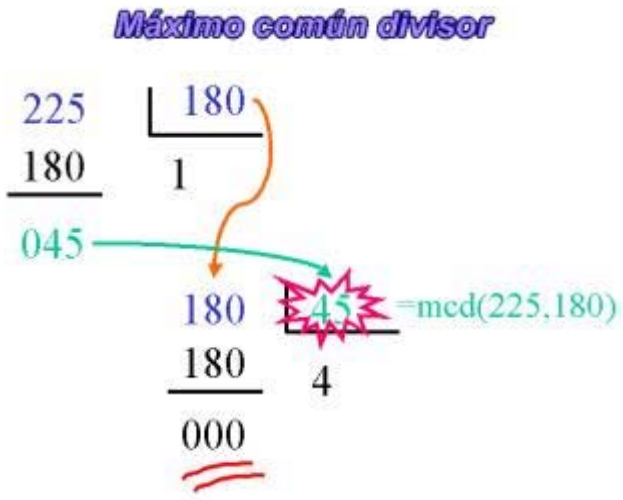
Máximo común divisor

¿Necesitas algún ejemplo más para aclararte las ideas?

Éste es otro caso en el que la solución iterativa sería más eficiente, y por tanto preferible: el cálculo del **máximo común divisor** de dos números por el algoritmo de **Euclides**. No aporta por tanto ninguna novedad, salvo la de comprobar que casi cualquier algoritmo iterativo puede tener una versión recursiva, y mostrar un ejemplo más de problema recursivo. La versión no recursiva de este algoritmo ya se proponía como ejercicio en unidades anteriores, por lo que aquí nos limitaremos a presentar la solución recursiva.

El cálculo del máximo común divisor de dos números enteros por el algoritmo de Euclides, consiste en:

1. Dividir el mayor entre el menor.
2. Repetir los pasos siguientes hasta que obtengamos un resto cero.
   - Si el resto de la división es 0, el máximo común divisor es el divisor.
   - En caso contrario, seguiremos dividiendo, tomando como dividendo el divisor, y como divisor el resto de la división anterior.



Por tanto, el máximo común divisor será el último resto no nulo de esas divisiones sucesivas, que también es el divisor de la última división para la que se ha obtenido resto 0.

Descarga el archivo MaximoComunDivisor.java

Unidad Didáctica X

Torres de Hanoi

¿Pero hay algún caso en el que la recursividad sea realmente útil o al menos aconsejable?

Es posible que ya empieces a dudarlo, pero los hay.

En este apartado presentamos un **problema** netamente recursivo. Como ya dijimos, sería posible encontrar una solución iterativa, pero no es ni mucho menos evidente.

Lo interesante de este ejemplo es que a primera vista parece bastante complicado de resolver, pero al cambiar el punto de vista y enfocarlo de un modo recursivo, la solución es de una sencillez y una elegancia increíbles, por lo que se aprecia que la solución recursiva sí aporta claridad a la solución, y resulta muy adecuada en este caso.

Planteemos en qué consiste el problema:

En realidad se trata de un juego consistente en resolver un **problema**, adornado con una bonita **historia** a modo de leyenda por quienes lo enunciaron por primera vez. Merece la pena que le eches un vistazo al enlace que aparece más abajo para conocer la historia completa. Aquí expondremos un enunciado más directo, pero menos hermoso:

El juego consiste en pasar todos los anillos que se encuentran en un poste a otro, usando un tercer poste como auxiliar, en el menor número de movimientos posible y cumpliendo unas sencillas reglas:

- No se puede mover más de un anillo en cada movimiento
- Nunca puedes soltar un anillo fuera de un poste
- No se puede soltar un anillo sobre otro de menor diámetro.

Posición **INICIAL** para 5 anillos. Hay que cambiar todos los anillos a otro poste.

Posición **FINAL**, en la que todos los anillos han cambiado de poste.

Nuestro programa deberá solicitarnos el número de anillos que hay inicialmente en el poste origen, e indicar

los movimientos que debemos realizar para trasladarlos, siguiendo las reglas, al poste destino, ayudándonos del poste auxiliar.

Es evidente que para un anillo bastaría un movimiento, para dos anillos serían necesarios tres movimientos, para 3 anillos siete, y **en general, para n anillos serían necesarios como mínimo 2^n - 1 movimientos**. El problema se va complicando de forma exponencial según se van añadiendo anillos, de forma que parece difícil poder encontrar una solución general.

Pero si lo enfocamos desde un punto de vista **recursivo**, la cosa cambia. El problema de mover n anillos desde el poste origen al destino, usando el poste auxiliar, podemos descomponerlo en tres subproblemas:

- Mover los n-1 anillos superiores del poste origen al auxiliar.
- Mover el anillo que queda (un anillo) del poste origen al destino.
- Mover los n-1 anillos del auxiliar al destino.

Las reglas impiden mover n-1 anillos de golpe, pero podemos ver cada uno de esos 3 pasos como una nueva llamada recursiva al método con menos anillos, lo que nos acerca cada vez más al caso base.

En este ejemplo el caso base se da cuando el número de anillos es 1. En este caso es evidente que basta con escribir el único movimiento que hay que hacer: "Mover de origen a destino".

En el siguiente enlace puedes encontrar el código para probarlo, pero te recomendamos que no introduzcas un número de anillos muy alto, ya que puede tardar bastante en encontrar la solución. Ten en cuenta que cada nuevo anillo requiere el doble de movimientos (el doble más uno, para ser exactos)

Descarga el archivo TorresDeHanoi.java

Y la solución a este problema puedes comprenderla mejor con la siguiente animación:

DEMO: Visualiza la resolución de las torres de hanoi

PARA SABER MÁS:
En el siguiente enlace encontrarás la historia de las torres de Hanoi, junto a unas estupendas applets java interactivas, que te permiten ver en funcionamiento el problema, jugando tú mismo si lo deseas o viendo la solución en movimiento.
Torres de Hanoi **[Versión en caché]**
Otro enlace interesante puede ser éste, en el que aparece la historia un poco más adornada, pero de forma breve.
Breve Historia del Juego de las Torres de Hanoi **[Versión en caché]**

Unidad Didáctica X

Ordenación Rápida o QuickSort

¿Recuerdas el algoritmo que te proponíamos como tarea en la unidad anterior que consistía en ordenar una lista de 4 números leídos desde teclado? Seguramente habrás visto que hacerlo "por el método de la fuerza bruta", consistente en ir comparando cada uno de los cuatro números con todos los demás, resulta más largo y más laborioso de lo esperado. Pues imagina si lo que tenemos que ordenar en vez de ser una lista de 4 números es una lista de 40.000 números, o los 600.000 registros de la base de datos de clientes de una compañía financiera.

Evidentemente, habrá que recurrir a usar algún método de ordenación más adecuado que los anteriores, y sobre todo, será importante elegirlo bien para que el tiempo necesario para realizar la ordenación sea el menor posible, elegir uno que sea eficiente.

Existen bastantes algoritmos de ordenación, y muchos de ellos pueden definirse de forma recursiva, pero entre ellos destaca el **algoritmo de ordenación rápida** o QuickSort. Es probablemente el algoritmo de ordenación más rápido que se conoce. Este algoritmo fue desarrollado por C.A.R. Hoare en 1960, quien lo definió de forma recursiva, ya que es inherentemente recursivo. El tamaño del código es sorprendentemente pequeño para la alta velocidad y eficiencia de este algoritmo. A groso modo consiste en lo siguiente:

- Eliges un elemento cualquiera de la lista, al que llamaremos pivote.
- Buscas la posición que le corresponde al pivote en la lista ordenada.
  - Para ello se utilizan dos índices: `contadorIzquierdo` y `contadorDerecho`, que nos van a indicar inicialmente la posición del primer elemento de la lista y la del último, respectivamente
  - Recorres la lista simultáneamente con `contadorIzquierdo` y `contadorDerecho`: por la izquierda con `contadorIzquierdo` (desde el primer elemento hacia el último), y por la derecha con `contadorDerecho` (desde el último elemento hacia el primero).
  - Cuando `lista[contadorIzquierdo]` sea mayor que el pivote y `lista[contadorDerecho]` sea menor los intercambias.
  - Repites esto hasta que se crucen los índices.
  - El punto en que se cruzan los índices es la posición adecuada para colocar el pivote, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).
  - Al finalizar este procedimiento el pivote queda en una posición en que todos los elementos a su izquierda son menores que él, y los que están a su derecha son mayores.
- En este momento el pivote separa la lista en dos sublistas, la primera con todos los elementos menores y la segunda con todos los elementos mayores
- Realizas esto de forma recursiva para cada sublista mientras éstas tengan más de 1 elemento.
- Una vez terminado este proceso todos los elementos estarán ordenados.

Se utilizan versiones iterativas para mejorar su rendimiento (ya sabemos que los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos). A pesar de todo, en este caso la solución recursiva sigue siendo altamente eficiente y mucho más clara.

En el siguiente enlace puedes encontrar el fichero con el código fuente de un programa que introduce una serie de números aleatorios en un vector del tamaño que se haya indicado desde teclado, escribe todos los números en el orden en que aparecen inicialmente en el vector, los ordena usando Ordenación rápida o QuickSort y muestra los elementos ordenados.

☐ **Descarga el archivo QuickSort.java**

No te preocupes si todavía no lo entiendes todo. Hay algunas cuestiones de **sintaxis**, como la declaración de arrays y su uso, o el paso de parámetros a métodos, que todavía no se han explicado formalmente.

Pero al igual que una persona que pretende perfeccionar su idioma en el

extranjero no puede pretender que le hablen usando sólo las palabras y expresiones que entiende, al aprender un lenguaje de programación también es frecuente que en los ejemplos, si se quiere que sean más o menos reales y útiles, es necesario que aparezcan algunos elementos que todavía no se han explicado.

DEMO: Visualiza un ejemplo de quicksort

Pero de la misma forma que irse a Inglaterra a "sumergirse en el inglés" ayuda a aprender el idioma, incluso aquellas expresiones que nunca se habían oído, sumergirse en el lenguaje de programación ayuda a facilitar el aprendizaje de los elementos nuevos que aparecen, siendo más fácil cuando llega la hora de introducirlos formalmente.

PARA SABER MÁS:
En este interesante enlace verás el funcionamiento del algoritmo de ordenación rápida mediante una animación. Está en inglés, pero no te preocupes, ejecuta la animación de forma que veas la resolución de una manera gráfica que te ayudará a comprender perfectamente el funcionamiento del algoritmo.
Ordenación Rápida o QuickSort
Este otro enlace se te explica perfectamente el funcionamiento del QuickSort en español.
Ordenamiento Rápido (QuickSort) **[Versión en caché]**
Si quieres profundizar sobre la figura de Charles Antony Richard Hoare pulsa este enlace.
C. A. R. Hoare **[Versión en caché]**

Unidad Didáctica X

Algoritmos de vuelta atrás (Backtraking)

*Las técnicas de programación son algo muy útil para un programador y conocer **soluciones adecuadas** a determinados problemas aporta, en el peor de los casos, un considerable ahorro de tiempo de diseño y codificación, ya que habitualmente la técnica de la recursividad supone una importante reducción de instrucciones en el programa.*

***Carmen** es una persona práctica y busca siempre **soluciones simples** que faciliten la programación y el mantenimiento de las aplicaciones, pero también es consciente de que existen soluciones a determinados problemas que han sido desarrolladas tras largos estudios y análisis de expertos. Así que también es inteligente hacer uso de esas técnicas de los grandes programadores. Pero para ello al menos hay que conocerlas y saber cómo funcionan.*

¿Te has planteado alguna vez si un ordenador, que es capaz de resolver en poco tiempo el más complicado de los laberintos, o que es capaz de ganar jugando al campeón del mundo de ajedrez, es **inteligente**?

La respuesta es que no lo es. Alguien dijo hace ya bastantes años que los ordenadores sólo son tontos rápidos. Tan rápidos que parecen inteligentes, pero no son capaces de hacer más que lo que se les indica que hagan en un programa. El inteligente es la persona que diseña ese programa, en todo caso, y a veces ni eso.

¿Y como un tonto, por rápido que sea, es capaz de ganar al ajedrez al campeón del mundo? ¿No requiere inteligencia el ajedrez?

Básicamente "haciendo trampas". El humano mejor entrenado es capaz de visualizar en su cabeza un número limitado de posibilidades que se abren ante un movimiento, y tiene un tiempo limitado para rechazarlas o aceptarlas. Además, si decide iniciar un camino, y hace un movimiento, no puede volverse atrás, aunque luego compruebe que es erróneo (Pieza tocada, pieza movida).

Mientras tanto,

- el programa le permite al ordenador comprobar, a una velocidad de varios millones de instrucciones por segundo, absolutamente todas las posibilidades, hasta el final, de cada nueva posición.
- **Puede comprobarlas todas, y si alguna lleva a una posición de desventaja, volver atrás y probar otra alternativa**.
- Y cuando decimos todas, son todas, incluso aquellas más absurdas, que el jugador inteligente de ajedrez ni siquiera considera.
- Es como si al ordenador se le permitiera mover todas las fichas, las suyas y las del contrario, tantas veces como quisiera, antes de volver a colocar el tablero como estaba y realizar su movimiento.
- El humano, en el mismo tiempo, habrá tenido que imaginarse 6 ó 7 jugadas, las que a su juicio más prometen, que seguramente no habrá podido analizar hasta el final, a una velocidad de cálculo infinitamente inferior y elegir una, antes de agotar su tiempo.
- La inteligencia radica en dedicar el tiempo a comprobar justamente las posibilidades interesantes, e ignorar las absurdas o irrelevantes sin siquiera pensar en ellas.

¿Y como conseguir que el programa haga todas esas comprobaciones?

Eso se consigue mediante algoritmos de vuelta atrás. Estos problemas son típicamente recursivos también, ya que necesitan almacenar en la pila las opciones que se van ensayando, y se van retirando de la pila las opciones ya comprobadas para volver a un punto anterior, antes de ensayar nuevas posibilidades. Esto es algo que se consigue fácilmente usando **recursividad**.

- Cada nuevo camino o posibilidad que se explora, viene representada por una llamada recursiva, que se carga en la pila.
- Si en esa llamada no se encuentra la solución, el algoritmo finaliza y se descarga de la pila, situando el problema justo en el punto en el que se encontraba antes de haber ensayado esa posibilidad, y listo para ensayar la siguiente.

En el siguiente apartado no vamos a aprender a hacer un programa de ajedrez, pero podrás ver en funcionamiento un problema de vuelta atrás relacionado con el ajedrez: El salto del caballo.

Unidad Didáctica X

Salto del caballo

¿Eres aficionado al ajedrez? Si es así, es posible que disfrutes más de este ejercicio. ¿Sabes de qué va? Consiste sólo en encontrar una solución que nos permita mover el caballo de forma que recorra todo el tablero pasando por cada posición una sola vez.

Es un algoritmo típico de vuelta atrás. ¿Se podría implementar iterativamente?

Sí, pero sería necesario:

- "anotar" todos los puntos en los que hemos realizado un movimiento,
-