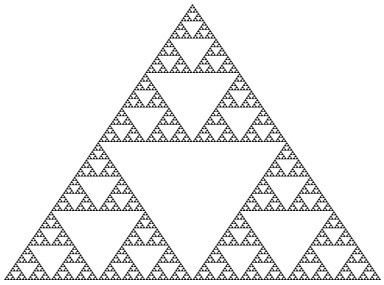
Recursividad

Los términos **recurrencia**, **recursión** o **recursividad** hacen referencia a una técnica de definición de conceptos (o de diseño de procesos) en la que el concepto definido (o el proceso diseñado) es usado en la propia definición (o diseño).

Un ejemplo paradigmático sería el del triángulo de Sierpinski en el que cada triángulo está compuesto de otro más pequeños, compuestos s su vez de la misma estructura recursiva (de hecho en este caso se trata de una estructura fractal)



Otro caso de estructura recursiva son las denominadas *Matryoshkas* (o muñecas rusas): donde cada muñeca esconde en su interior otra muñeca, que esconde en su interior otra muñeca que …, hasta que se llega a una muñeca que ya no escode nada.

En nuestro caso nos preocuparemos de los **métodos (funciones o acciones) recursivos**: aquéllos en los que, dentro de las instrucciones que los forman, contienen una llamada a sí mismos.
Como siempre, la parte más compleja no será a nivel de programación, sino a nivel de diseño: dado un problema, ser capaz de encontrar una solución recursiva del mismo. Por tanto, deberemos ser capaces de **pensar recursivamente**.
Algunos de los problemas que veremos ya los sabéis resolver iterativamente y es bueno comparar las soluciones recursivas que veremos con las iterativas que podéis realizar por vuestra cuenta.

1. Llamadas a funciones

Antes de empezar con las llamadas recursivas, recordaremos brevemente cómo funcionan las llamadas entre funciones y cómo éstas modifican el flujo de ejecución.
Consideremos el siguiente ejemplo, que ya vimos en el tema anterior:

```
 1 /*
 2  * File: SquareRoot.java
 3  * --------------------
 4  * This program calculates the square root of a
 5  * given positive integer
 6  */
 7
 8 import acm.program.ConsoleProgram;
 9
10 public class SquareRoot extends ConsoleProgram {
11
12   public int squareRoot(int n) {
13     int lower = 0;
14     while ((lower + 1) * (lower + 1) <= n) {
15       lower = lower + 1;
16     }
17     return lower;
18   }
19
20   public void run() {
21     int n = readInt("Enter a natural number: ");
22     int root = squareRoot(n);
23     println("The root is " + root);
24   }
25 }
```

Lo que vamos a considerar ahora es cómo se ejecutan las líneas, en función de las llamadas entre funciones:

- La ejecución comienza la línea 21, que contiene la llamada a la función `readInt`. Se congela la ejecución del método `run` y se ejecuta el código de `readInt`
- Poco podemos decir de la ejecución de `readInt` ya que no disponemos de su código, pero a grandes rasgos, después de escribir el mensaje y esperar la entrada del usuario, una vez éste ha entrado un número entero, se devuelve la ejecución a la línea

21 (en la que habíamos congelado la ejecución), asignando el valor devuelto por `readInt` a n
- La ejecución pasa entonces a la línea 22, dónde se llama al método `squareRoot`. Se vuelve a congelar la ejecución de `run` y se pasa a ejecutar la línea 13
- Después de unas cuantas vueltas (dependiendo del valor de n) , se sale del bucle y se ejecuta la línea 17, volviendo al punto dónde nos habíamos congelado la ejecución de `run`.
- …

¿Qué pasaría si, desde una función, llamáramos a la propia función? Pues que el punto de ejecución pasaría a la primera instrucción de la función y que, cuando dicha llamada retornase, continuaríamos la ejecución en el punto en el que nos hubiéramos quedado.

2. Pensar recursivamente: Los textos palíndromos

Una palabra (o texto) es palíndroma si se lee igual de izquierda a derecha que de derecha a izquierda.
Por ejemplo: "Dábale arroz a la zorra el abad" es, tal y como podéis comprobar, un texto palíndromo[1]
Lo que queremos será un programa tal que, dado un texto, nos diga si es palíndromo o no.
El programa principal básicamente consistirá en:

- Pedir los datos al usuario. Como se tratará de un texto, la forma natural de hacerlo será con el método `readLine`
- Eliminar los espacios de la cadena de entrada. Para ello crearíamos un método `removeSpaces` tal que, dado un `String`, devuelva otro, con los espacios borrados[2].
- Llamar a la función que comprueba si el texto entrado es palíndromo. Llamaremos a esta función `isPalindrome`, y será una función que recibirá como parámetro un `String` y devolverá un `boolean`.
- Finalmente, dependiendo del valor devuelto por la función anterior, se indicará si el texto es palíndromo o no.

Si escribimos esto en Java, tendremos:

[1] Para simplificar, al introducir el texto obviaremos los posibles acentos ortográficos que pudieran tener las palabras.
[2] Para programarlo os podéis inspirar en el método `removeVocals` del tema anterior.

```
1 /* CheckPalindrome.java
2  * --------------------
3  * Checks whether the entered text is palindrome.
4  */
5
6 import acm.program.ConsoleProgram;
7
8 public class CheckPalindrome extends ConsoleProgram {
9
10   public String removeSpaces(String text) {
11     // Ejercicio
12   }
13
14   public boolean isPalindrome(String text) {
15      // Se detallará más adelante
16   }
17
18   public void run() {
19     String text = readLine("Enter text to check: ");
20     text = removeSpaces(text);
21     if ( isPalindrome(text) ) {
22       println("Text is palindrome.");
23     } else {
24       println("Text is not palindrome.");
25     }
26   }
27 }
```

Una solución iterativa

Antes de intentar solucionar el problema de forma recursiva, vamos a ver cómo procederíamos a hacerlo con los conocimientos que tenemos, es decir, mediante una solución iterativa.

¿Qué es lo que hemos de hacer? Básicamente comprobar que:

- el primer carácter de la cadena (posición 0) coincide con el último (posición longitud-1), y
- el segundo (posición 1), coincide con el penúltimo (posición longitud-2), y …
- … hasta llegar a la mitad de la cadena[3]

Para ello haremos un bucle que vaya generando las parejas a comparar. En el momento de encontrar dos caracteres diferentes, ya podemos dar por acabada la comprobación (ya que sabemos que no lo

[3] ¿Por qué solamente hasta la mitad? Considerad los casos de textos de longitud par e impar al justificarlo.

es). Si al final no hemos encontrado ninguna diferente, sabemos que se trata de un palíndromo. Es decir, se trata de un esquema de búsqueda.

```
1 public boolean isPalindrome(String text) {
2   int i = 0;
3   int length = text.length();
4   int maxPair = length / 2;
5   while ( i < maxPair &&
6           text.charAt(i) == text.charAt(length-1-i) ) {
7     i = i + 1;
8   }
9   // If not found, isPalindrome is true
10  return ( i == maxPair );
11 }
```

Si comparamos la descomposición del problema en subproblemas tenemos:

- problema inicial: ver si un texto es palíndromo
- una serie de problemas más sencillos: comparar parejas de caracteres.

Pensando una solución recursiva

Una solución recursiva del problema consistiría en una descomposición en la que, para comprobar si una texto es palíndromo, nos surja como subproblema la necesidad de saber si una parte de él lo es.

Recordad las muñecas rusas: tenemos una muñeca (saber si un texto es palíndromo) que al abrirla (descomponer el problema) contiene dentro otra muñeca más pequeña (saber si una parte del texto es un palíndromo).

En este caso, la descomposición que podemos hacer es la siguiente:para comprobar si un texto es palíndromo o no:

- miramos si los caracteres de los extremos (primero y último) son iguales
- comprobamos si el texto formado por los caracteres no extremos es palíndromo

Como puede observarse, en la nueva descomposición, uno de los subproblemas que nos aparece es el mismo que el original (**pero sobre un texto más pequeño**).

La necesidad de los casos simples

Si la única posibilidad de solucionar un problema fuera aplicar la regla recursiva, ¡nunca acabaríamos de resolverlo!, ya que siempre nos

quedaría un subproblema que resolver (al que aplicaríamos la regla recursiva, que nos daría otro subproblema, que nos daría …, hasta el infinito).
Es por ello que **deberá haber** casos que no necesiten de aplicar la regla recursiva, sino que se pueden resolver directamente. Son como la muñequita rusa más pequeña, que ya no contiene más muñequitas dentro. A estos casos que pueden resolverse directamente se les conoce como casos simple, en contraposición de los otros que se denominan casos recursivos.
En este caso, ¿cuándo podemos considerar que un texto es palíndromo sin necesidad de comprobar nada más?

- si un texto es vacío, podemos considerar que es palíndromo
- si un texto consiste en solamente una letra, también

Juntando todo, ya tenemos todos los ingredientes que necesitamos para programar nuestra versión recursiva de la función:

```
1 public boolean isPalindrome(String text) {
2   if ( text.length() <= 1) {
3     return true;
4   } else {
5     char   first = text.charAt(0);
6     char   last  = text.charAt(text.length()-1);
7     String inner = removeExtrems(text);
8     return (first == last) && isPalindrome(inner);
9 }
10
11 public String removeExtrems(String text) {
12   // Ejercicio para el lector
13 }
```

Es cierto que en esta versión recursiva, dentro del método `removeExtrems`, creamos muchas copias de trozos de la cadena. Cuando más adelante veamos más detalles de la clase `String`, veremos que hay formas de evitar dichas copias. Tened presente que todavía nos queda mucho por avanzar y que no podemos pretender tener en cuenta todos los detalles.

Resumiendo

Intentemos resumir en una tabla las intuiciones que obtenemos pensando en la imagen de las muñecas rusas y su equivalente en cuanto a la solución recursiva de un problema[4]:

[4] Como toda metáfora la coincidencia no es exacta, pero puede ayudarnos a tener intuiciones.

Muñecas rusas	Solución recursiva
Una muñeca puede abrirse para ver qué es lo que hay en su interior.	Un problema de descompone en varios subproblemas .
Al abrir una muñeca grande encontramos muñecas más pequeñas en su interior	Al descomponer un problema grande (casos recursivos) encontramos subproblemas que tienen la misma estructura que el problema inicial y trabajan sobre datos más pequeños
La muñeca más pequeña ya no contiene otras muñecas	Existen casos simples cuya solución no requiere descomponerlos más.
Sólo hay dos tipos de muñecas (las que contienen otras en su interior y las más pequeñas que no las contienen).	Entre los casos simples y los recursivos tengo todas las posibilidades cubiertas

3. Recursividad usando índices

Una forma de no tener que ir generando múltiples copias de partes del `String` en el caso del texto capicúa, consiste en plantear la recursividad no sobre el `String`, sino sobre unos índices que nos indiquen el subconjunto de elementos sobre los que estamos trabajando.
Como tanto los vectores como los `Strings` permiten acceder directamente a una posición, plantaremos este ejemplo sobre vectores y quedará como ejercicio hacer los cambios necesarios para que funcione sobre `Strings`.

¿Cómo referirse a un subvector?

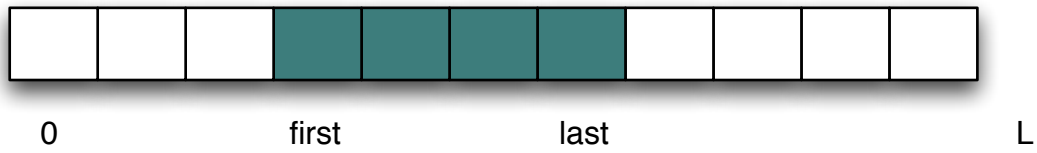
Supongamos que el `String text`, que ya no tiene espacios, almacena el texto del que queremos ver si es palíndromo, y `textChars` el `char[]` correspondiente, es decir:

```
char[] textChars = text.toCharArray();
```

Además denominaremos L al número de elementos del vector, es decir:

```
int L = textChars.length;
```

Por tanto, el vector está formado por las posiciones 0 a `L-1`.
Para referirnos a un subvector de elementos consecutivos, uno podría usar dos índices: uno para el primer elemento del segmento y otro para el último, es decir[5]:

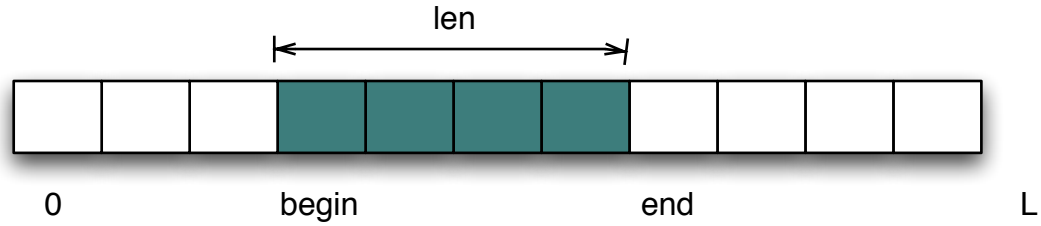


Pero la experiencia ha demostrado que elegir los índices de esta manera complica algunos algoritmos. Por ejemplo, para representar un subvector vacío hemos de hacer que último sea más pequeño que primero, el número de elementos del segmento es último-primero+1, lo que es algo incómodo.
Existen **dos formas típicas** para referirnos a un subvector:

- Dos índices, uno que se refiera al primer elemento de subvector (`inf`) y otro que se refiera al primer elemento fuera del subvector (`sup`).
- Un índice que indique el primer elemento del subvector (`begin`) y el número de elementos a considerar (`len`)[6].

[5] Fijaos que la posición L cae fuera del vector.

Es decir,



En este caso se cumple:

- `len = begin – end`
- el subvector vacío: `len = 0` y `begin = end`
- el subvector total: `begin = 0 y end = len = L`

Vectores palíndromos

La función recursiva que realizaremos tendrá la siguiente forma:

```
1 public boolean isPalindromeArray(char[] textChars,
2                                  int    begin,
3                                  int    end) {
4
5   // Checks whether the subarray from begin to
6   // end-1 of textChars is palindrome
7 }
```

¿Cuál será la descomposición en este caso? Para ver si los caracteres desde `begin` a `end-1` de `textChars` forman un palíndromo hemos de:

- comprobar si los caracteres extremos son iguales, es decir hemos de comparar `textChars[begin]` y `textChars[end-1]`
- comprobar que el subvector sin los extremos, es palíndromo, es decir, hemos de hacer la llamada recursiva con los parámetros `textChars, begin+1` y `end-1`

Como ya vimos anteriormente dicha descomposición solamente es posible si el subvector tiene longitud al menos dos. En los casos de longitudes cero (vacío) o uno (un solo carácter) sabemos que la función ha de retornar cierto.
Si lo juntamos todo, nos queda:

[6] Si recordáis, esta es la forma que se utiliza para indicar los elementos de un char[] a tener en cuenta cuando construimos un vector.

```
1 public boolean isPalindromeArray(char[] textChars,
2                                   int    begin,
3                                   int    end) {
4
5   // Checks whether the subarray from begin to
6   // end-1 of textChars is palindrome
7
8   if ( begin == end || begin == end-1) {
9     return true;
10  else {
11    return textChars[begin] == textChars[end-1] &&
12           isPalindromeArray(textChars, begin+1, end-1);
13  }
14 }
```

Teniendo en cuenta que `end-begin` da el número de elementos en el segmento, la condición del caso simple puede expresarse también como (`end-begin <= 1`). Fijaos que en la llamada recursiva el tamaño del intervalo es más pequeño.
Con esta nueva versión, el método `run` quedará ahora como:

```
1 public void run() {
2   String text = readLine("Enter text to check: ");
3   text = removeSpaces(text);
4   boolean isPal = isPalindromeArray(text.toCharArray(),
5                                     0,
6                                     text.length());
7   if (isPal) {
8     println("Text is palindrome.");
9   } else {
10    println("Text is not palindrome.");
11  }
12 }
```

Se deja como ejercicio la implementación usando como parámetros `begin` y `len`, es decir, el índice del primer elemento del subvector y su tamaño.

4. Un ejemplo sobre números: la exponenciación

Vamos a aplicar nuestra idea de buscar recurrencias a aun problema, esta vez sobre números: dados dos números a≥0 y b≥0, diseñar una función que calcule la exponenciación a^b.

Solución iterativa

Dado que a^b no es más que realizar b multiplicaciones de a, hacemos un bucle tal que, a cada vuelta acumule (multiplicando) sobre una variable (que inicialmente valdrá 0) el producto de esa variable por a. Al final de b iteraciones habremos multiplicado b veces por a y, por tanto, tendremos la exponenciación buscada.

Vamos a buscar una solución recursiva

Recordad, la estrategia consiste en encontrar una descomposición de la exponenciación que, a su vez, incluya una exponenciación. Una vez encontrada, deberemos buscar también como mínimo un caso en que se pueda dar el resultado sin necesidad de aplicar la recurrencia (si no, el programa nunca acabaría). Si con esos casos tenemos cubiertas todas las posibilidades, hemos acabado.
En nuestro caso podemos ver que:

- $a^b = a * a^{b-1}$, si $b \geq 1$
- cuando $b = 0$, $a^0 = 1$

Es decir,

```
1 public long exp(long a, long b) {
2   if ( b == 0 ) {
3     return 1;
4   } else {
5     return a * exp(a, b-1);
6   }
7 }
```

Hemos usado `long` en vez de `int` pues la exponenciación genera números potencialmente grandes y con `int` podríamos tener problemas de precisión.
Esta solución funciona pero, cuando `b` es grande, es bastante ineficiente: a cada llamada se reduce `b` en una unidad, por lo que para

calcular la exponenciación hemos de hacer tantas llamadas como valor tiene el exponente[7]. ¿Podemos hacerlo mejor?

Restar es lento, dividir es más rápido (para llegar a cero)

Si en vez de restar uno al exponente, lo dividimos por dos (usando división entera), tendremos que hacer menos llamadas hasta legar al caso simple. Por ejemplo, si empezamos con b=10, tendremos:

- Restando: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
- Dividiendo: 10, 5, 2, 1, 0

Incluso en este caso (con un valor bajo de b) la diferencia es significativa. Pensad que si duplico el valor de b, en el primer caso se duplica el número de vueltas, mientras que en el segundo solamente se añade una vuelta más[8].

¿Podemos encontrar una recursión dividiendo?

Como queremos dividir[9], tendremos que encontrar una relación entre el a^b original y $E^{b \text{ div } 2}$ de la llamada recursiva (dónde el E es una expresión que debemos determinar).
Una posible forma de proceder es buscar una relación entre b y (b div 2), y luego aplicar algo de aritmética.
Al dividir por dos, tenemos dos posibilidades, dependiendo de si b es par o impar[10]:

- Si b es par, la división entera es exacta, por lo que tenemos que b es exactamente $2 * (b \text{ div } 2)$
- Si b es impar, la división entera pierde el valor del resto de la división, por tanto b es $2 * (b \text{ div } 2)+1$

Si aplicamos el análisis anterior a la fórmula a^b, tenemos:

- Si b es par, $a^b = a^{2*(b \text{ div } 2)} = (a^2)^{b \text{ div } 2}$
- Si b es impar, $a^b = a^{2*(b \text{ div } 2)+1} = a * (a^2)^{b \text{ div } 2}$

Es decir, en ambos casos, para calcular a^b hemos de calcular $(a^2)^{b \text{ div } 2}$, tan sólo que en caso de que b sea impar, deberíamos multiplicar el resultado de la llamada recursiva por a antes de devolverlo. Es decir:

---

[7] Técnicamente se dice que el coste temporal del algoritmo es lineal respecto del valor del exponente. Más en la asignatura de Algorítmica y Complejidad.
[8] En este caso diremos que el coste temporal del algoritmo es logarítmico respecto del valor del exponente.
[9] Cuando trabajamos con enteros usamos la división entera a div b. En Java no existe un operador especial por lo que cuando a y b son enteros, a/b es la división entera entre a y b.
[10] Recordad que estamos tratando con aritmética de enteros por lo que la división no siempre es exacta.

```
1 public long exp2(long a, long b) {
2   if ( b == 0) {
3     return 1;
4   } else {
5     long result = exp2(a*a, b/2);
6     if ( b%2 == 1) {
7       result = a * result;
8     }
9     return result;
10  }
11 }
```

En este caso, es b quién decrece a cada llamada recursiva, ya que cuando $b > 0$, se tiene que $b/2 < b$ (usando división entera).

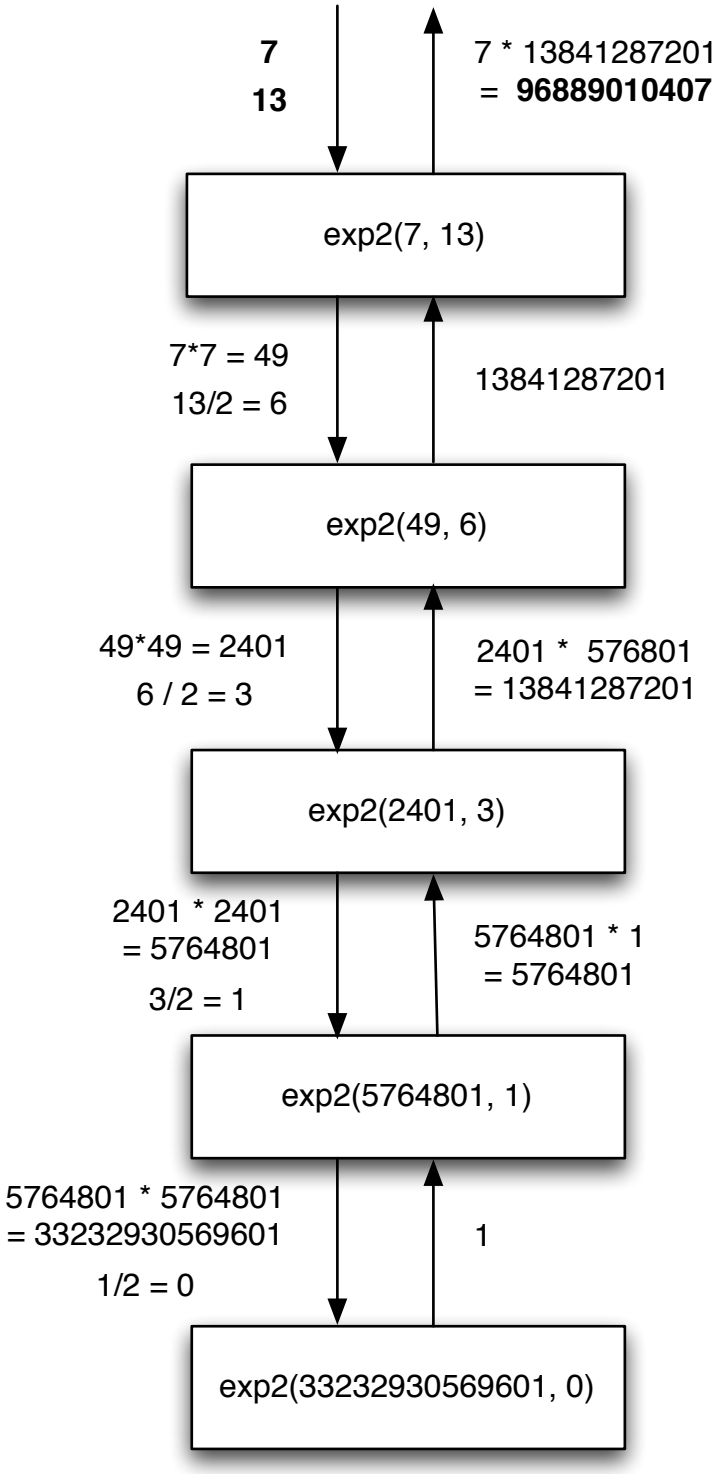
Trazando las llamadas

Aunque hayamos realizado un fino razonamiento sobre la corrección de la función anterior, a veces necesitamos un pequeño ejemplo de ejecución para ver que, efectivamente, funciona. Vamos a ejecutar paso a paso el cálculo de 7^{13}.

Observando el diagrama podemos observar que la ejecución de un algoritmo recursivo consiste en dos bucles:

- uno que va desde la llamada inicial hasta el caso simple y que va modificando los parámetros
- uno que va desde la solución del caso simple hasta regresar de la llamada inicial y que va propagando la solución

Fijaos que conseguimos estos dos bucles sin explicitarlos de manera alguna (con un `while`, o un `for`).



Espacio usado por un algoritmo recursivo

Si miramos la traza de las llamadas, veremos que, para que la ejecución de la función pueda recorrer el camino de vuelta, en algún lugar se deberán guardar los valores de los parámetros y de las variables locales de las llamadas anteriores. Este lugar es la **pila de ejecución**. Su nombre responde a que, de la misma manera que en una pila de platos, solamente podemos colocar cosas encima de la pila, y acceder al elemento de la cima (que es el último que hemos introducido).

Debido a que en la ejecución de un método recursivo podemos tener que realizar muchas llamadas hasta alcanzar los casos simples, puede darse la situación de que se llena la pila de ejecución y se produce el error de **desbordamiento de pila** (*Stack Overflow*).

5. Raíz cuadrada (esta vez "exacta")

Vamos a realizar el diseño de una función para calcular la raíz cuadrada de un número $x \geq 0$, pero usando números en coma flotante[11].

Una primera cuestión a tener en cuenta es que, debido a que hay errores de precisión, ya que el número de decimales que se toman en cuenta es finito, no podemos comprobar si el resultado es igual a la raíz cuadrada sino que es una aproximación *suficientemente buena*.

Relacionado con lo anterior, un algoritmo del tipo: comenzamos en 0.0 y vamos incrementado poco a poco (por ejemplo en incrementos de 0.00001) hasta encontrar la raíz cuadrada no son aplicables[12]. Además, estamos buscando una solución recursiva. ¿Qué hacemos?

Reformular el problema

A veces, cuando un problema se nos resiste, la solución consiste en replantear el problema en función de otro que sí sabremos resolver. Desgraciadamente no existe una regla que podamos seguir para encontrar la reformulación que nos garantice encontrar el *¡ajá!* adecuado para cada problema. Lo único que podemos hacer es estudiar soluciones, meditar sobre ellas, intentar generalizarlas y aplicarlas a otras situaciones.

En este caso el ¡ajá! consiste en considerar lo siguiente[13]:

- tal y como se ha comentado antes, no podemos buscar exactitud, sino una buena aproximación
- una buena aproximación quiere decir que lo que encontraremos está suficientemente cerca
- que esté suficientemente cerca quiere decir que está dentro de un intervalo que incluye la solución
- intervalo que incluye la solución, …
- ummm, intervalo, ¡intervalo! , ¡claro!, ¡aja! ☺

---

[11] En el primer tema vimos el cálculo de la raíz cuadrada entera, es decir, del número más alto que, elevado al cuadrado, es menor que el número dado. Es decir, la raíz cuadrada entera r de un número n cumple $r^2 \leq n < (r+1)^2$

[12] No porque no logren la solución, sino porque el tiempo que emplearían en ello sería astronómico.

[13] También podríamos habernos acordado teorema de Bolzano sobre funciones continuas.

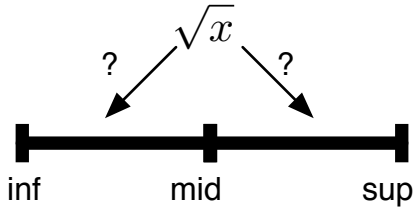
Podemos reformular el problema de la siguiente manera: dado el número de que queremos calcular la raíz (x), la precisión que queremos en la solución ($epsilon$), y los extremos de un intervalo que la contiene (inf y sup), diseñar una función que devuelva la raíz cuadrada con la precisión requerida.
Formalicemos un poco las relaciones que se han de cumplir entre estos valores:

- **Entrada**: $inf \leq \sqrt{x} \leq \text{sup} \equiv inf^2 \leq x \leq sup^2$
- **Salida**: $\left|\sqrt{x} - r\right| \leq epsilon \cong |x - r^2| \leq epsilon$

Solución recursiva

¿Cómo descompondremos el problema en subproblemas de manera que alguno de ellos tenga la misma forma que el problema original? La idea será la siguiente:

- si tenemos ya la precisión adecuada, ya hemos encontrado la solución.
- si aún no la tenemos, calculamos el punto medio del intervalo y nos planteamos en si la raíz está en el subintervalo de la izquierda o en el de la derecha, por lo que ya tenemos la llamada recursiva a realizar. Visualmente:



Programando la solución

```
1 public double squareRoot(double x,
2                           double epsilon,
3                           double inf,
4                           double sup) {
5
6   double mid = (inf + sup) / 2;
7
8   if ( Math.abs(mid * mid - x) <= epsilon ) {
9     return mid;
10  } else if ( mid * mid < x ) {
11    return squareRoot(x, epsilon, mid, sup);
12  } else {
13    return squareRoot(x, epsilon, inf, mid);
14  }
15 }
```

Dos cosas a mencionar en el código anterior:

- El uso de la función `Math.abs` para calcular el valor absoluto. La clase `Math`, del paquete `java.lang`, contiene funciones algunas matemáticas de uso común, como la que permite calcular el valor absoluto. Como todas las clases del paquete `java.lang` no hace falta hacer un `import` para usarlas (lo mismo sucede con la clase `String`).
- Fijaos en la forma de alinear los `if`, aunque el `else` de la línea 12 se corresponde con el `if` de la 10 se alinea con el de la línea 8. De esta manera visualmente vemos que la estructura de control se corresponde en el fondo con tres posibilidades (dos con la condición explicitada y una que es la que se toma cuando ambas fallan).

La llamada inicial

Hasta el momento no ha habido demasiados problemas para encontrar los parámetros a pasar en la primera llamada. Para esta función, tendremos que pensar un pelín más. De las cuatro parámetros de la función, dos están claros: x y épsilon. El problema viene para inf y para sup, ya que se ha de cumplir $inf^2 \leq x \leq sup^2$.

- para inf no hay demasiado problema dado que $x \geq 0$, lo que garantiza que podemos coger siempre $inf = 0$
- para sup está un pelín más complicado. Una posible idea sería coger x, ya que normalmente $x^2 \geq x$, pero ello solamente es

cierto si $x \geq 1$. En caso de no serlo, podemos coger el mismo 1.0 como un valor mayor que x

En resumen:

```
1 public double squareRoot(double x, double epsilon) {
2   return squareRoot(x, epsilon, 0.0, Math.max(1.0, x));
3 }
```

Dos comentarios:

- el uso de la función `Math.max`, para decidir si usamos un valor u otro para *sup* (también se podría usar un condicional pero usar max o min en estos casos es habitual).
- el método tiene el mismo nombre que el que tenía 4 parámetros. En Java es posible tener dos métodos con el mismo nombre, siempre que se puedan distinguir a partir del número de parámetros o del tipo de los mismos. A esto se le llama **sobrecarga**[14].

[14] Fijaos que esto mismo pasa con los métodos `print` y `println`, que se llaman igual independientemente de que lo que escribamos sea un entero, un carácter, un String, etc.

6. Búsqueda binaria[15] en un vector ordenado

Intentemos aplicar esta idea de los intervalos que se van dividiendo por la mitad a un problema *parecido*: la búsqueda en un vector de enteros ordenado.
Similitudes:

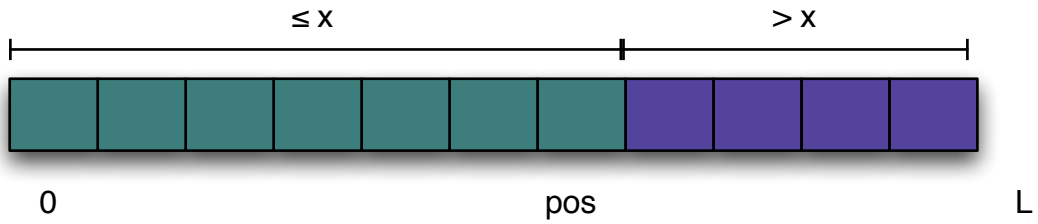
- al igual que los números reales en el intervalo [inf, sup], los elementos del vector están ordenados.
- buscamos un elemento dentro del intervalo (inicialmente el intervalo es el vector desde las posiciones 0 hasta la L-1).

Diferencias:

- en el caso de la raíz cuadrada sabemos con seguridad que la solución existe, es decir, existe un elemento que es la raíz cuadrada. En el caso de la búsqueda, el elemento podría no existir dentro del vector.

Reformulación del problema

De cara a tratar con el problema de qué devolver si el elemento a buscar no se encuentra en el vector, reformularemos el problema de la siguiente manera: dado un vector de enteros ordenado crecientemente y un elemento x, que puede o no pertenecer al vector, encontrar la posición **pos** del vector tal que marque la separación entre los que son $\leq x$ y los que son $> x$.
Es decir:



Casos particulares:

- si x pertenece al vector, ocupará la posición pos
- si x aparece varias veces en el vector, se dará la posición de más a la derecha
- si todos los elementos del vector son $> x$, el valor devuelto es -1
- si todos los elementos del vector son $< x$, el valor devuelto es L-1
- si x no pertenece al vector, pero hay elementos $\leq x$ y $> x$, se devuelve una posición del vector entre 0 y L-1

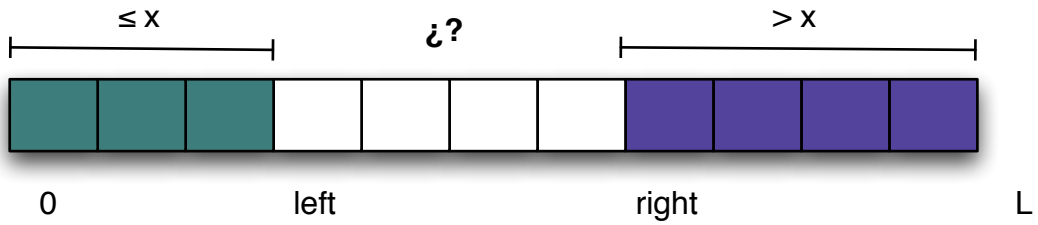
[15] También denominada búsqueda dicotómica.

Planteamiento de la solución recursiva

La estrategia consistirá en considerar que el vector está dividido en tres zonas:

- la izquierda, que contiene los elementos que sabemos que son $\geq x$
- la central, que es la que desconocemos, y que queremos hacer cada vez más pequeña
- la derecha, que contiene los elementos que sabemos que son $> x$

Gráficamente:



Dos cosas a tener en cuenta:

- fijaos en que los tres intervalos están definidos de la manera recomendada, es decir, la posición del primer elemento del intervalo y la posición del primero que no pertenece a él
- cualquier zona puede estar vacía (por ejemplo inicialmente las zonas de la derecha y de la izquierda lo están, ya que no sabemos nada)[16].
- se cumple: $0 \leq left \leq right \leq L$

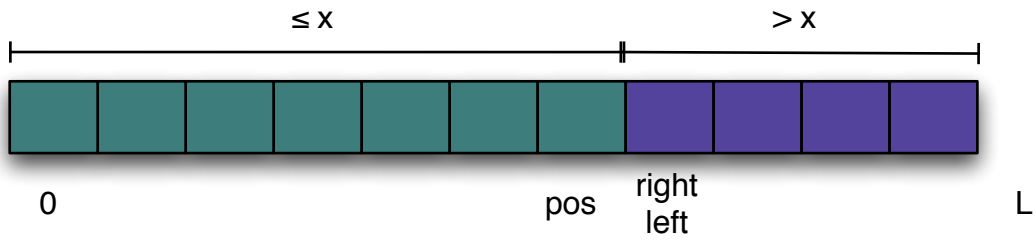
A nivel de Java, la función recursiva tendrá la forma:

```
1 public int binarySearch(int[] v,
2                          int   x,
3                          int   left,
4                          int   right) {
5   ¿?
6 }
```

Caso simple

¿Cuándo podemos dar la respuesta sin tener que hacer nada más? Cuando la zona intermedia de los interrogantes está vacía, es decir, cuando left=right, ya sabemos la solución left-1.

[16] Como ejercicio podéis considerar los valores de left y right para los casos particulares mencionados en el apartado anterior.

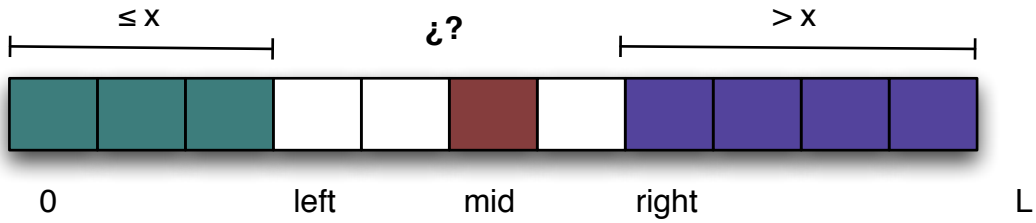


Caso recursivo

Ahora es cuando aplicaremos la estrategia que hemos utilizado antes, es decir, dividiremos el intervalo por la mirad y estudiaremos qué podemos hacer. Para calcular la posición media, haremos la misma división que antes, tan solo que ahora se tratará de una división entera, es decir:

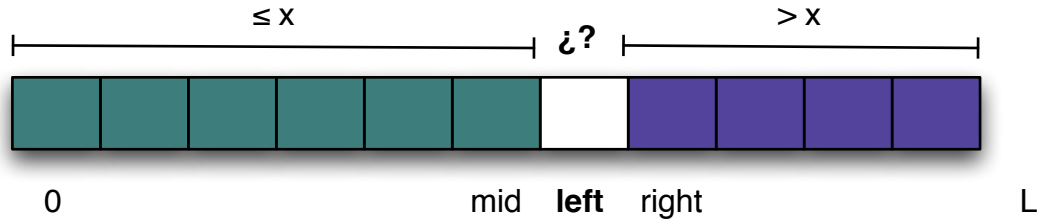
$$mid = (left + right) \text{ div } 2;$$

Gráficamente:

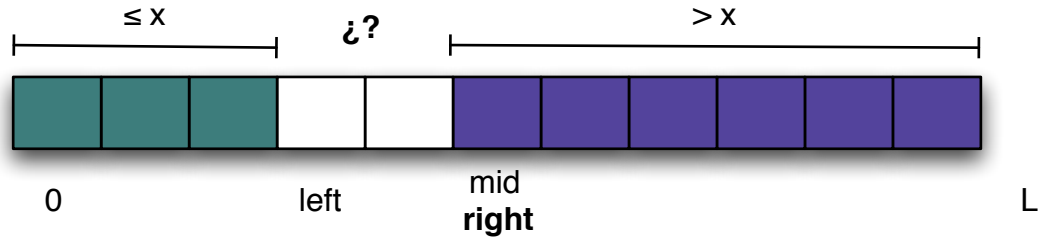


Lo que hemos de hacer depende de si el valor que hay en la posición mid es $\leq x$ o $> x$.

- Si $v[mid] \leq x$, podemos extender la zona izquierda hasta incluir la posición mid, es decir:



- Si $v[mid] > x$, ahora lo que podemos extender es la zona de la derecha, es decir:



Dónde el índice marcado en negrita es el que se pasa en la llamada recursiva.

Una vez diseñada la solución, simplemente queda programarla en Java e indicar los valores de la llamada inicial. Al principio la zona

desconocida ocupa todo el vector, por lo que la llamada inicial se hace con left = 0 y right = L

```
 1 public int binarySearch(int[] v,
 2                          int   x,
 3                          int   left,
 4                          int   right) {
 5   int pos;
 6   if ( left == right ) {
 7     pos = left-1;
 8   } else {
 9     int mid = (left + right) / 2;
10     if ( v[mid] <= x) {
11       pos = binarySearch(v, x, mid+1, right);
12     } else {
13       pos = binarySearch(v, x, left, mid);
14     }
15   }
16   return pos;
17 }
18
19 public int binarySearch(int[] v, int x) {
20   return binarySearch(v, x, 0, v.length);
21 }
22
23 public boolean contains(int[] v, int x) {
24   int pos = binarySearch(v, x);
25   return pos != -1 && v[pos] == x;
26 }
```

Las llamadas se hacen sobre intervalos más pequeños

Lo único que nos queda es comentar que cuando hacemos la llamada recursiva, lo hacemos sobre un intervalo más pequeño. Lo primero a tener en cuenta es que en los casos recursivos $left < right$ (ya que siempre se cumple que $left \leq right$ y además sabemos que $left \neq right$.

- Caso `v[mid]<=x`: Queremos demostrar que

$$right - left > right - (mid + 1)$$
$$-left > -(mid + 1)$$
$$left < mid + 1$$
$$left \leq mid$$
$$left \leq (left + right) \text{ div } 2$$

Lo cual es cierto para $right > left$ ya que

$$(left + right) \text{ div } 2 \geq (left + left) \text{ div } 2 = left$$

- Caso `v[mid]>x`: Ahora se trata de probar que

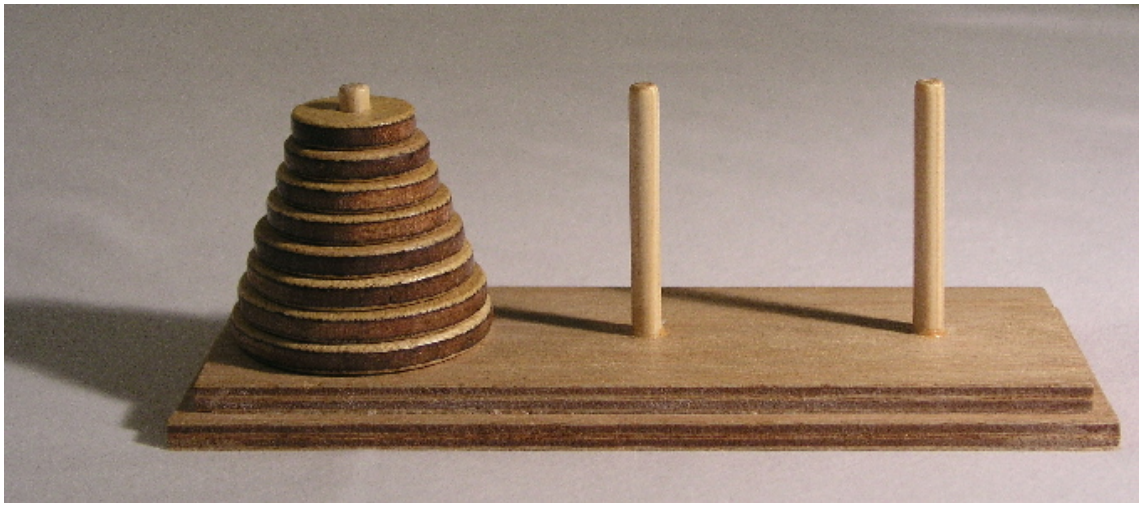
$$right - left > mid - left$$
$$right > mid$$
$$right > (left + right) \text{ div } 2$$

Lo que es cierto para $right > left$ ya que

$$right = (right + right) div\ 2 > (left + right) div\ 2$$

Es decir, en ambos casos el intervalo es menor.

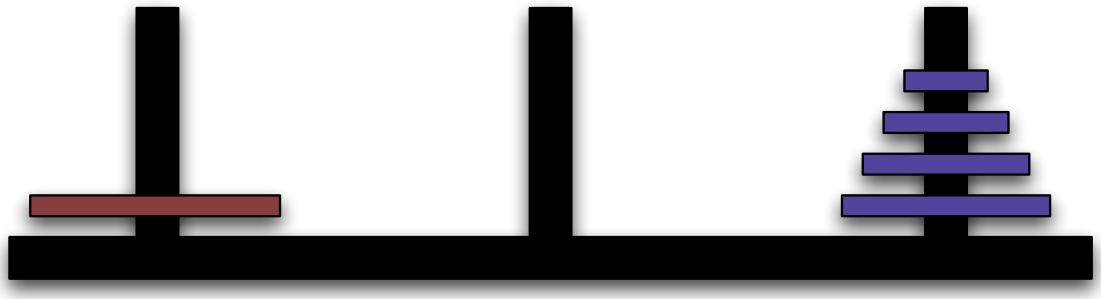
7. Recursividad múltiple: las torres de Hanói



El puzle de las Torres de Hanói consiste en hacer pasar los discos, que inicialmente están en la primera torre, a la segunda, usando la tercera como auxiliar. Eso sí, cumpliendo dos reglas:
solamente puede moverse el disco superior de una torre (el que está encima de todo)
no podemos poner un disco sobre otro de menor tamaño
Lo que se pide es una función que escriba los movimientos a realizar para resolver un puzle con un número de discos dado.
Para referirnos a un movimiento solo tendremos que indicar las torres origen y destino, ya que solamente podrá moverse el disco superior de la torre origen y quedará colocado sobre todos los que ya existen en la torre destino.

Planteamiento recursivo

Una forma de encontrar la solución consiste en estudiar las condiciones bajo las cuales puedo mover el disco más grande:



- ha de ser el único que haya en la torre origen
- la torre destino ésta ha de estar vacía
- todos los discos han de estar en la torre auxiliar

¿Lo veis? Para poder mover el disco más grande de los n desde la torre origen a la destino, he de haber movido previamente los n-1 discos desde la torre origen a la auxiliar.
Una vez movidos los n-1 discos que lo obstruyen y movido el más grande a la torre destino, hemos de mover los n-1 discos que están en la torre auxiliar a la torre destino, usando como torre auxiliar la torre origen inicial.

Casos simples

¿Cuándo puedo solucionar el problema directamente? Cuando solamente he de mover un disco, no he de apartar nada no usar la torre auxiliar: lo muevo directamente.

Escribiendo la solución en Java

La función tendrá tres parámetros:

- numDisks, que será el número de discos a mover
- from, nombre de la torre de partida
- to, nombre de la torre de destino
- using, nombre de la torre auxilar

```
1 public void solve(int numDisks,
2                   String from,
3                   String to;
4                   String using) {
5
6   if ( numDisks == 1 ) {
7     println(“Move disk from “ + from + “ to “ + to);
8   } else {
9     solve(n-1, from, using, to);
10    println(“Move disk from “ + from + “ to “ + to);
11    solve(n-1, using, to, from);
12  }
13 }
```

Visualización de las llamadas

Por ejemplo, para 3 discos que se quieren mover de la torre “A” a la “B” usando “C” como auxiliar haríamos la llamada

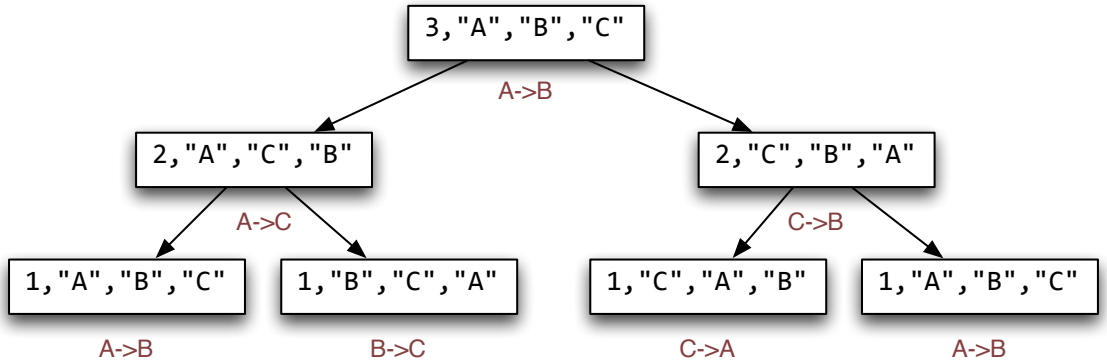
```
solve(3, “A”, “B”, “C”)
```

que escribiría:

```
Move disk from A to B
Move disk from A to C
Move disk from B to C
Move disk from A to B
```

```
Move disk from C to A
Move disk from C to B
Move disk from A to B
```

Si hacemos un gráfico de las llamadas que se producen nos queda el siguiente árbol[17]:



Dónde además de indicar dentro de los cuadrados los parámetros que se pasan en la llamada, he indicado de la forma Origen->Destino, fuera de las llamadas, lo que se escribe en la pantalla.

Simplificando la solución

De cara a pensar la solución ha sido conveniente considerar que el caso base es cuando numDisks es 1. Pero si miramos la solución obtenida vemos que podríamos simplificar las cosas considerando como caso más simple cuando numDisks es 0. ¡En tal caso, no hay que hacer nada para mover los 0 discos!
El código de la función quedaría ahora como:

```
1 public void solve2(int numDisks,
2                    String from,
3                    String to;
4                    String using) {
5
6   if ( numDisks >= 1 ) {
7     solve2(n-1, from, using, to);
8     println(“Move disk from “ + from + “ to “ + to);
9     solve2(n-1, using, to, from);
10  }
11 }
```

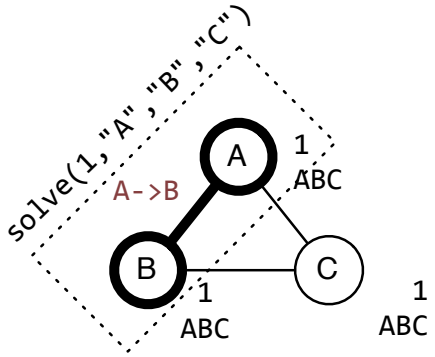
[17] En este caso no mostraremos los resultados ya que las llamadas a la función no devuelven nada, simplemente se escribe en la salida. Para que ocupe menos espacio, solamente mostramos el valor de los tres parámetros.

Recursividad múltiple

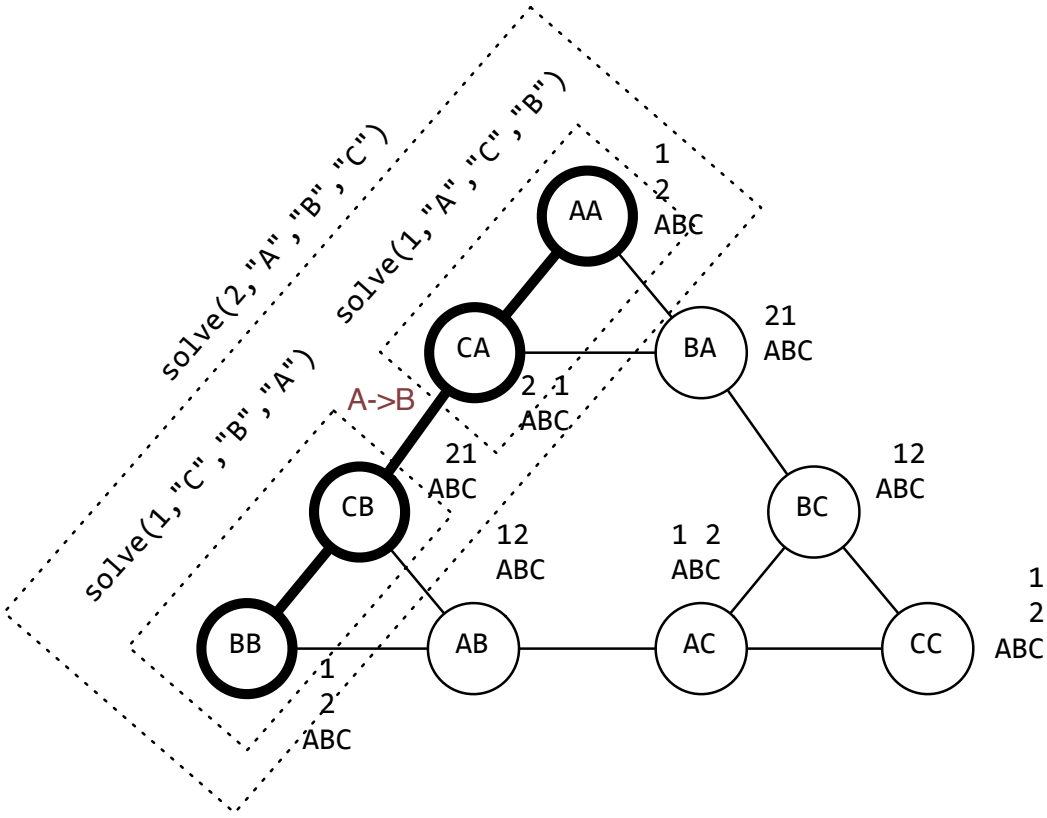
Fijaos que en este caso la descomposición del caso no simple ha dado lugar a dos llamadas recursivas. No hay ningún problema. Mientras las llamadas se hagan sobre datos más pequeños, no hay limitación alguna en su cantidad.

Como curiosidad final

Un aspecto curioso, que de alguna manera conecta de forma palindrómica con el inicio del tema es que si dibujamos las configuraciones posibles de los discos como vértices de un grafo y los unimos cuando es posible pasar de una a otra a través de un movimiento válido, obtenemos el triángulo de Sierpinski con el que comenzábamos el tema.
Para un disco (caso simple):



Para dos discos (caso recursivo):



8. Bibliografía

- Para la parte de recursividad, "Programación Metódica", de J.L.Balcázar, Ed. McGraw-Hill (2001).
- Para el funcionamiento de las llamadas a función, capítulo 5 del libro "*The Art and Science of Java (Preliminary Draft)" de Eric S. Roberts.* Lo tenéis disponible en sakai.