Problemas de Recursividad

Problema 1.

El factorial de un número **entero no negativo** ($\geq \mathbf{0}$), denotado como $n!$, se define como $\prod_{i=1}^{n} i = 1 * 2 * \ldots * n$ cuando $n > 0$, y $0! = 1$.
Por ejemplo $6! = 1 * 2 * 3 * 4 * 5 * 6 = 720$.

- Diseñad una método recursivo que lo calcule
- Implementadlo en Java (junto con un programa que lo utilice)
- Mostrad cómo se ejecutan las llamadas en el caso de calcular el factorial de 3

Problema 2.

Para calcular el máximo común divisor de dos números **enteros positivos (>0)** puedo aplicar el **algoritmo de Euclides**, que consiste en ir restando el más pequeño del más grande hasta que queden dos números iguales, que serán el máximo común divisor de los dos números.
Por ejemplo, si comenzamos con el par de números 412 y 184, tendríamos:

412	228	44	44	44	44	44	36	28	20	12	8	**4**
184	184	184	140	96	52	8	8	8	8	8	4	**4**

Es decir, m.c.d.(412, 184) = 4

Problema 3.

Diseñad un método recursivo tal que, dado un vector de números enteros, retorne la suma de sus elementos.
Para poder hacer recursividad, usaremos un índice que indique el trozo de vector a sumar en cada llamada.
Es decir, el método a diseñar tendrá la forma:

```
1 public int sum(int[] elems, int pos) {
2   ¿?
3 }
```

Diseñad este método así como el que lo utiliza para calcular la suma de todo el vector. Tened en cuenta cómo hacemos para referirnos a un intervalo dentro de un vector. ¿Qué pasa si el vector está vacío (es decir, cuando `elems.length` vale cero)?
Usando el método recursivo, implementad el método que lo usa para calcular la suma de todo el vector, es decir:

```
4 public int sum(int[] elems) {
5   return sum(elems, ¿?);
6 }
```

Nota: Podéis considerar dos descomposiciones del vector, una en la que la zona que vais sumando es el prefijo del vector y otra en la que trabajáis sobre el sufijo del mismo.

Problema 4.

Diseñad un método recursivo que escriba al revés la cadena que se le pasa como parámetro. Para poder hacer recursividad añadiremos un parámetro adicional para poder variar la subcadena sobre la que se trabaja sin necesidad de crear cadenas auxiliares.
Dicho método será de la forma:

```
1 public void printReversed(String text, int index) {
2   ¿?
3 }
```

Haced dos versiones del mismo:

- una en la que la subcadena sobre la que trabaja la función sea el **prefijo** de la cadena original
- otra en la que sea el **sufijo**.

En ambos casos implementad la función que llama a la función recursiva diseñada, es decir:

```
4 public void printReversed(String text) {
5   printReversed(text, ¿?);
6 }
```

Mostrad las secuencias de llamadas que se producen para la cadena “abcd” en ambos casos.
Nota: No se considera una solución válida invertir la cadena y luego escribirla.

Problema 5.

El ejemplo de la exponenciación mostrado en los apuntes, permite la siguiente descomposición:

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

Acabad de diseñar la solución recursiva que la emplea, implementar la solución en Java y hacer el mismo diagrama de llamadas para el caso de 7^{13}.

Nota: Es muy interesante que intentéis resolver un mismo problema de varias maneras y comparéis entre sí las diferentes soluciones.

Problema 6.

Ya que estamos, diseñad un método tal que dada una cadena, retorne la cadena invertida (es decir, el primer carácter del resultado será el último de la cadena dada, etc.). Dicho método tendrá la forma:

```
1 public String invert(String text) {
2   ¿?
3 }
```

Podéis encontrar diferentes formas de descomponer la cadena en varias cadenas, por ejemplo:

- el primer carácter y el sufijo correspondiente a la cadena menos el primer carácter
- el último carácter y el prefijo correspondiente a la cadena original menos el último carácter
- las dos mitades de la cadena

PISTA: El método substring de la clase String os puede ser de gran ayuda.

La soluciones anteriores tienen el problema de que se crean muchas cadenas intermedias. Una forma de evitarlo es considerar que en vez de un String tenéis un char[], que podéis modificar sin necesidad de crear vectores adicionales. En este caso deberéis añadir uno o más parámetros para demarcar los trozos dentro del char[] sobre los que trabajáis en cada momento.

Problema 7.

Diseñad un método tal que, dados dos vectores de enteros, retorne un booleano indicando si son iguales, es decir, si tienen los mismos valores en las mismas posiciones.
Para poder hacerlo recursivamente deberéis, como ya es habitual, hacer otro método que incluya uno o más índices para indicar los subvectores sobre los que se trabaja.
Indicad qué llamada se hace al método recursivo para resolver el problema inicial.

Problema 8.

Diseñad un método tal que calcule el máximo de un vector **no vacío** de números enteros. De forma similar al problema 4, implementad el

método que llama al que habéis definido recursivamente para que se calcule el máximo de todo el vector.

Problema 9.

El algoritmo chino de multiplicación. Diseñad un método que multiplique dos números enteros usando las siguientes equivalencias:

$$x * y = (2 * x) * \left(\frac{y}{2}\right) = \begin{cases} (2 * x) * (y \; div \; 2), & si \; y \; es \; par \\ (2 * x) * (y \; div \; 2) + x, & si \; y \; es \; impar \end{cases}$$

Una cuestión a considerar es la siguiente: la expresión (2*x) puede calcularse de manera no recursiva. Una posibilidad es usar:

- 2 * x = x + x
- 2 * x también puede implementarse (y en realidad el código que genera el compilador es lo que hace) desplazando un bit la representación binaria de x. En el tema de Archivos veremos cómo usar los desplazamientos de bits en Java.

Problema 10.

Dado un vector de números enteros ordenado decrecientemente, diseñad un método tal que compruebe si el valor de alguno de los elementos del vector coincide con su índice.
Podéis hacer dos versiones:

- una que trabaje con el prefijo (o sufijo) del vector. Esta solución apenas utiliza la información sobre la ordenación del vector
- otra que, usando dos índices, sea capaz de descartar a cada llamada la mitad del vector. Os podéis inspirar en la búsqueda binaria

En ambos casos implementad los métodos que hacen la llamada inicial al que habéis diseñado recursivamente dando valores iniciales a los índices.
Pista: podéis pensar qué relación tiene este problema con la búsqueda dicotómica y, si la encontráis, obtendréis la solución.

Problema 11.

Un problema parecido al anterior se puede plantear cuando el vector de enteros está ordenado crecientemente y no contiene valores repetidos.
El razonamiento en este caso es más complicado que en el caso anterior (obviamente cuando se intenta hacer la versión que, a cada paso divide la longitud del intervalo donde buscar por la mitad).

Pista: la idea de la solución consiste en darse cuenta de que los valores crecen como mínimo tanto como los índices. Esto es cierto porque el vector no contiene elementos repetidos.

Problema 12.

La sucesión de Fibonacci viene definida por la siguiente recurrencia:

$$f_{n+2} = f_n + f_{n+1}$$

con valores iniciales $f_0 = 0$ y $f_1 = 1$.
Diseñad e implementad un método recursivo para calcular el enésimo término de la sucesión y mostrad el árbol de llamadas que se produce al calcular f_4 con vuestra solución.

Problema 13.

Mostrad cómo se va modificando el vector

```
int[] v = {7, 8, 1, 3, 5, 4, 1, 6}
```

cuando se ordena utilizando el algoritmo de QuickSort explicado en los apuntes. Mostrad por separado las llamadas que se realizan a la función `partition`.

Problema 14.

Hay figuras que pueden describirse recursivamente. Una de las más conocidas es el triángulo de Sierpinski, del que ya hemos hablado en el apartado de recursividad. Lo que haremos en este problema es definir un método recursivo que dibuje en la pantalla dicho triángulo.
Para ello procederemos paso a paso haciendo sucesivas aproximaciones al problema.

Fase 1

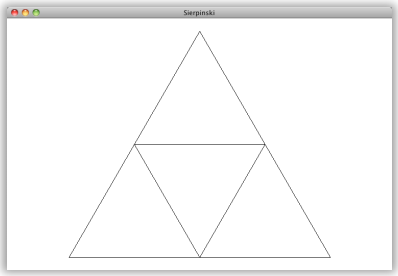
Inicialmente haremos una función tal que dibuje un triángulo equilátero (con el lado inferior alineado horizontalmente) tal que su vértice inferior izquierdo esté en las posición marcada por las coordenadas `x` e `y`, y la longitud del lado sea `side`, es decir:

```
4 public void addTriangle(double x, double y, double side) {
5   ¿?
6 }
```

Haced un programa principal que haga que la altura del triángulo sea el 90% de la altura de la pantalla y que esté centrado.

Fase 2

Utilizando la función anterior haced un método que usando el método anterior (dibujando tres triángulos) realice el siguiente dibujo:



Para posicionar la figura, usaremos los mismos parámetros que en el caso anterior, es decir, las coordenadas del vértice inferior izquierdo y el lado del triángulo total.
Tened en cuenta que la figura parece descompuesta en cuatro triángulos pero solamente hemos de dibujar los tres triángulos externos.
Fijaos en que la descomposición en tres triángulos pequeños ocupa el mismo espacio que el triángulo de la fase 1.

Fase 3

Ahora es cuando haremos recursividad. Para ello definiremos el método:

```
7 public void sierpinski(double x, double y, double side, int level)
  {
8   ¿?
9 }
```

Dónde `x`, `y` y `side` representan lo mismo que en los casos anteriores, y `level` es el parámetro que indicará las llamadas recursivas a hacer. Cuando `level` valga 1, el método sierpinski dibujará un único triángulo. Cuando sea mayor que uno, lo que hará es la misma descomposición que hemos hecho en la fase 2, pero en vez de llamar al método que dibuja un triángulo, llamará recursivamente al método sierpinski decrementando `level` en una unidad.
El resultado de llamar al método con un `level` inicial de 10 es: