Laboratorio 3 – Pilas y la transformación recursivo a iterativo

Los objetivos de la práctica son trabajar los elementos siguientes:

- Implementación y uso de la estructura de datos pila (Stack).
- Utilización de dicha estructura para convertir métodos recursivos en iterativos (incluso para el caso de recursividad múltiple).

La práctica está compuesta por los ejercicios descritos a continuación, **el primero de los cuales forma parte del contenido del primer parcial y los dos últimos, del segundo parcial.**

Ejercicio 1: Definición e implementación de las pilas

En este primer apartado implementaremos la interfaz `Stack<E>` definida como:

```
package stack;

public interface Stack<E> {
    void push(E elem);
    E top();
    void pop();
    boolean isEmpty();
}
```

Tanto el método top() como el método pop() devuelven la **excepción predefinida y no comprobada** `NoSuchElementException`[1] indicando en el mensaje (que se pasa como parámetro al constructor) que la pila está vacía.

Lo que se pide es implementar dicha interfaz en la clase `LinkedStack<E>` utilizando nodos encadenados, es decir:

```
package stack;

public class LinkedStack<E> implements Stack<E> {
    // ¿?

    @Override
    public E top() {¿?}

    @Override
    public void pop() {¿?}
```

[1] Estamos siguiendo la especificación de la clase Deque<E>, que es la que se usa en Java como implementación de las pilas. Existe una clase Stack, pero está marcada como obsoleta (deprecated) y no debería utilizarse.

```
    @Override
    public void push(E elem) {¿?}

    @Override
    public boolean isEmpty() {¿?}
}
```

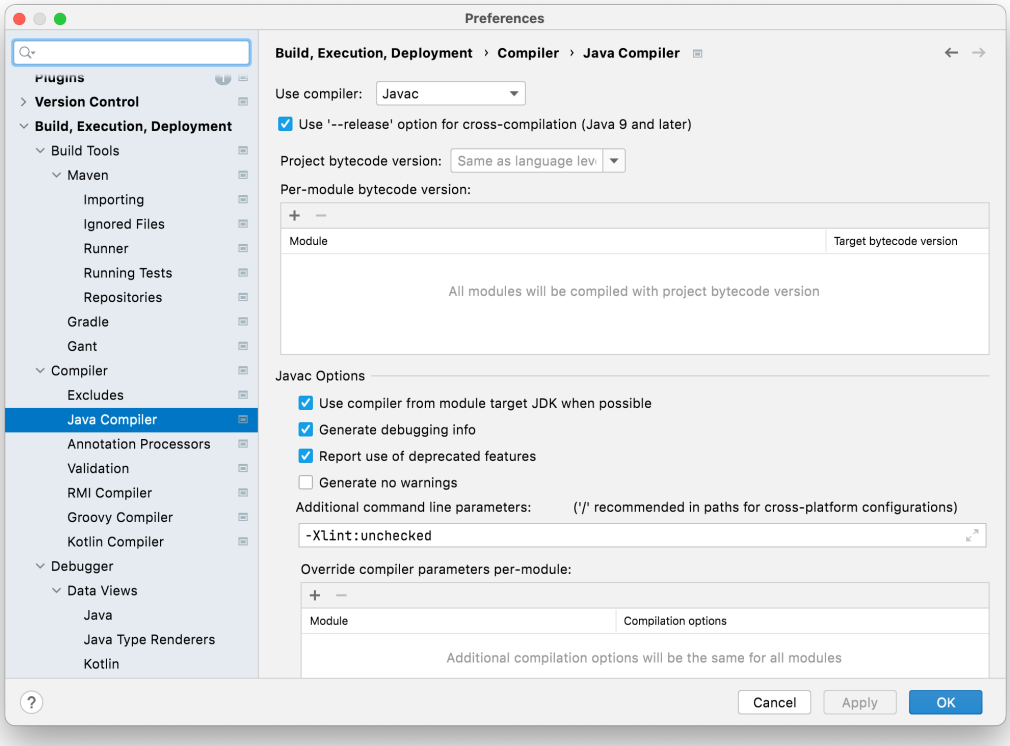
La clase que representará los nodos de la pila, y que solamente se utilizará en su interior, la definiréis como una **clase interna, privada y estática** de la clase `LinkedStack<E>`.

Obviamente, como nos podemos equivocar y querremos estar suficientemente seguros de que cuando usemos dicha implementación ésta no contiene errores, crearemos una **clase de pruebas** `LinkedStackTest` (como siempre, creando el directorio test para el código de pruebas y colocando dicha clase en el mismo paquete que la clase Stack), es decir:

```
package stack;

import static org.junit.jupiter.api.Assertions.*;

class LinkedStackTest {

    // ¿?
}
```

Recordad que los test deben **comprobar el comportamiento** de la clase, es decir, si sus métodos se ejecutan según su especificación y no su implementación (no ha de acceder a la parte interna de la clase). Por tanto debéis probar los casos de pilas vacías y pilas no vacías y comprobar que, en todos los casos, los métodos se comporten según diga su especificación. Como veréis **no podréis comprobar los métodos de forma independiente unos de otros**, ya que unos métodos construyen pilas y otros acceden a las partes de esas pilas. Comprobar la especificació es comprobar que ambos tipos de métodos funcionan de forma coherente entre sí, siguiendo la especificación.

Para conseguir que **el compilador nos ayude aún más** en los posibles errores de genéricos, añadiremos `-Xlint:unchecked` a las **opciones de compilación** (y de hecho es una muy buena costumbre **añadirlo siempre**, no solamente en esta asignatura, sino a todo proyecto Java). Para hacerlo, abrid el menú de Preferencias del IntelliJ:



Ejercicio 2: Transformación a iterativo

Para mostrar el proceso de transformación de un algoritmo recursivo a iterativo, partiremos de la siguiente definición de factorial (definida en la clase `Factorial` del paquete `factorial`):

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

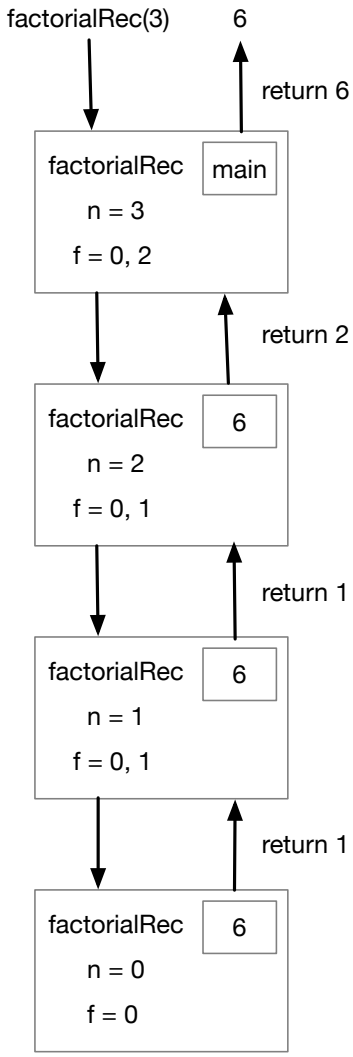
De cara a la transformación nos interesará que el código de partida:

- Defina todas las variables locales al principio de la función y las inicialice explícitamente.
- Cada llamada recursiva se guarde en una variable local.

En nuestro caso, hemos introducido la variable local f para almacenar el resultado de la llamada recursiva y la declaramos, como variable local. Recordad que como se trata de una variable local de tipo entero, Java la inicializará con valor 0, por lo que el código transformado es:

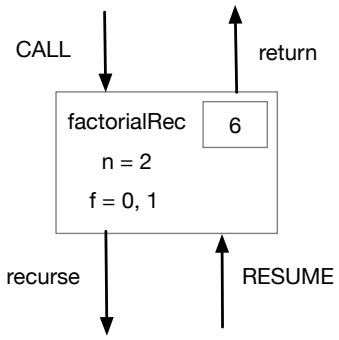
```
1. public static int factorialRec(int n) {
2.     int f = 0;
3.     if (n == 0)
4.         return 1;
5.     else {
6.         f = factorialRec(n - 1);
7.         return n * f;
8.     }
9. }
```

Si recordamos cómo se ejecuta, por ejemplo la llamada `factorialRec(2)`, mostrándolo con los diagramas que usamos en Programación 2, en los que el recuadro interno indica la línea de regreso de la llamada, veremos:



Básicamente la transformación a iterativo consiste en **explicitar los elementos de las cajas en una clase** (que llamaremos `Context`) y usar una pila para manejarlos (para recordar aquellos contextos a los que es necesario volver).

Fijaos en que cada una de las llamadas a la función `factorialRec`, es decir, cada una de las cajas, tiene dos puntos de entrada y dos puntos de salida, es decir:



Dichos puntos de entrada y salida los podemos explicitar en el propio código de la función:

```
public static int factorialRec(int n) {
    int f = 0;
    // CALL
    if (n == 0)
        return 1;
    else {
        f = factorialRec(n - 1); // recurse
        // RESUME
        return n * f;
    }
}
```

Los puntos de salida **return** no los hemos marcado ya que coinciden con la instrucción homónima y que, obviamente, representan salidas de la función, devolviendo el resultado a quien nos ha llamado.

Los otros puntos se corresponden con:

- **CALL**: punto de entrada de la llamada a la función.
- **RESUME**: punto de entrada a la función cuando la llamada recursiva nos ha devuelto el resultado y hemos de continuar con la ejecución.
- **recurse**: punto de salida de la función en la que dejamos la ejecución de ésta para ejecutar la llamada recursiva[2].

Para representar los puntos de entrada, utilizaremos el tipo enumerado `EntryPoint`, definido en la clase `Factorial` como:

[2] En este enunciado estamos tratando explícitamente del caso de llamadas recursivas aunque el uso de la pila es exactamente el mismo para todo tipo de llamadas a funciones.

```
private enum EntryPoint {
    CALL, RESUME
}
```

Si queréis saber más sobre tipos enumerados, podéis consultar el tutorial de Java, concretamente: Enum Types.

Para representar el contexto de cada llamada, es decir, de sus variables locales y parámetros, así como el punto de entrada de ésta, definiremos la clase `Context` como:

```
private static class Context {
    int n;
    int f;
    EntryPoint entryPoint;

    Context(int n) {
        this.n = n;
        this.f = 0;
        this.entryPoint = EntryPoint.CALL;
    }
}
```

Como es una clase interna y privada de la clase `Factorial`, no es necesario definir *getters* y *setters* para manipular sus variables de instancia.

Para guardar los contextos de ejecución a los que hemos de regresar (recordad que al ejecutar una función recursiva hay un recorrido que va desde la llamada inicial al caso simple y otro que transforma el resultado de este caso simple hasta obtener el resultado de la llamada inicial), utilizaremos una pila. ¿Por qué una pila? Porque el siguiente contexto al que regresaremos será el último que hemos apilado (que se corresponde con el contexto de quién nos llamó).

Por ello tendremos la variable:

```
Stack<Context> stack = new LinkedStack<>();
```

Aunque podríamos perfectamente usar la pila para devolver los resultados, hemos decidido añadir una variable, de nombre **return_,** en la que almacenaremos el resultado de la última llamada procesada[3] y, por tanto, definiremos la variable:

```
int return_ = 0;
```

[3] Esto está en concordancia con las convenciones de llamada de los procesadores x86 (https://en.wikipedia.org/wiki/Calling_convention)

Una vez establecidos los tipos y la variable que almacena la pila, la llamada inicial para calcular el factorial de n consistirá en ejecutar:

```
stack.push(new Context(n));
```

Es decir, colocaremos como primer contexto uno en el que:

- La variable n sea en valor de n que nos han pasado como parámetro.

Teniendo en cuenta el constructor de `Context`:

- El valor inicial de `f` será `0` (valor de inicialización de una variable entera).
- Como es una llamada, el punto de entrada será `CALL`.

El algoritmo iterativo no finalizará su ejecución mientras haya contextos en la pila por ejecutar, es decir:

```
while (!stack.isEmpty()) { ... }
```

Cuando la pila esté vacía, el resultado final, tal y como hemos indicado, estará almacenado en la variable `return_`, con lo que después del bucle lo único que queda es:

```
return return_;
```

Ahora viene la parte más compleja, pero si hemos entendido los diferentes elementos que hemos presentado hasta ahora, la podremos entender perfectamente. Lo primero que hemos de hacer es obtener el contexto actual:

```
Context context = stack.top();
```

Fijaos en que **no sacamos el contexto de la pila** y **obtenemos una referencia** al mismo, ello nos permitirá manipular el contexto manteniéndolo en la pila[4].

Como estamos ejecutando un contexto, hemos de saber en qué punto de ejecución, en qué punto de entrada, nos encontramos. Por ello:

```
switch (context.entryPoint) { ... }
```

Si se trata de una llamada a la función:

```
case CALL:
```

[4] Se podría haber elegido igualmente sacar el contexto de la pila, manipularlo, y volverlo a colocar, pero se ha decidido que hacerlo así oscurecía el uso real que se hace de la pila de ejecución.

Se ha de decidir si se trata de un caso simple o uno recursivo. Fijaos en que para acceder al valor del parámetro para decidir si es un caso simple o recursivo, hemos de acceder al valor de **context.n.**

```
if (context.n == 0) {
    // simple
} else {
    // recursive
}
break;
```

El break es necesario para salir del case (y dejárselo es uno de los errores que hacen que la sentencia switch sea odiada por muchos[5]).

En el caso simple, hemos de retornar el valor 1 y dar por finalizado la ejecución de la llamada actual (y por ello sacamos el contexto actual de la pila), es decir:

```
return_ = 1;
stack.pop();
```

El caso recursivo es un poco más complejo: como no cambiamos ninguna variable del contexto actual, antes de la llamada recursiva, lo único que hace falta es cambiar el punto de entrada para que, cuando volvamos a encontrar en la pila este contexto de ejecución, sepamos que se trata de RESUME, es decir:

```
context.entryPoint = EntryPoint.RESUME;
```

Finalmente, lo único que queda hacer es la llamada recursiva, lo que conseguimos apilando el contexto que la representa:

```
stack.push(new Context(context.n - 1));
```

Mostremos ahora qué sucede si el punto de entrada es el de retorno de la llamada recursiva. En este caso:

```
case RESUME:
```

En este caso hemos de guardar en la variable f el valor de la llamada recursiva, lo que conseguimos con:

```
context.f = return_;
```

[5] Java 12 introdujo, y Java 13 refinó, las llamadas expresiones switch (https://docs.oracle.com/en/java/javase/13/language/switch-expressions.html) que solventa alguno de los aspectos desagradables de la sentencia switch. Dichos cambios van en la línea de que Java incorpore más elementos provenientes de la programación funcional.

Y ahora, hemos de retornar como valor el producto de f * n, lo que se consigue con:

```
return_ = context.f * context.n;
```

Como la llamada está finalizada, sacamos el contexto de la pila, y ejecutamos el break (para salir del case).

```
stack.pop();
break;
```

Si juntamos todo el código de la implementación iterativa (y usando *switch expressions*, lo que elimina la necesidad de los break, así como el uso de *var* para escribir un código más terso[6]), queda:

```
public static int factorialIter(int n) {
    int return_ = 0;
    var stack = new LinkedStack<Context>();
    stack.push(new Context(n));
    while (!stack.isEmpty()) {
        var context = stack.top();
        switch (context.entryPoint) {
            case CALL -> {
                if (context.n == 0) { // simple
                    return_ = 1;
                    stack.pop();
                } else {
                    context.entryPoint = EntryPoint.RESUME;
                    stack.push(new Context(context.n - 1));
                }
            }
            case RESUME -> {
                context.f = return_;
                return_ = context.f * context.n;
                stack.pop();
            }
        }
    }
    return return_;
}
```

[6] En la variable `return_` hemos mantenido la declaración explícita de tipo para resaltar que es **el mismo que el que devuelve la función**.

El Fibonacci iterativo

A continuación, os presentamos la aplicación de la transformación explicada en el apartado anterior al caso de la función recursiva que calcula la serie de Fibonacci utilizando recursividad múltiple (ver `fibonacciOrig(int n)`).

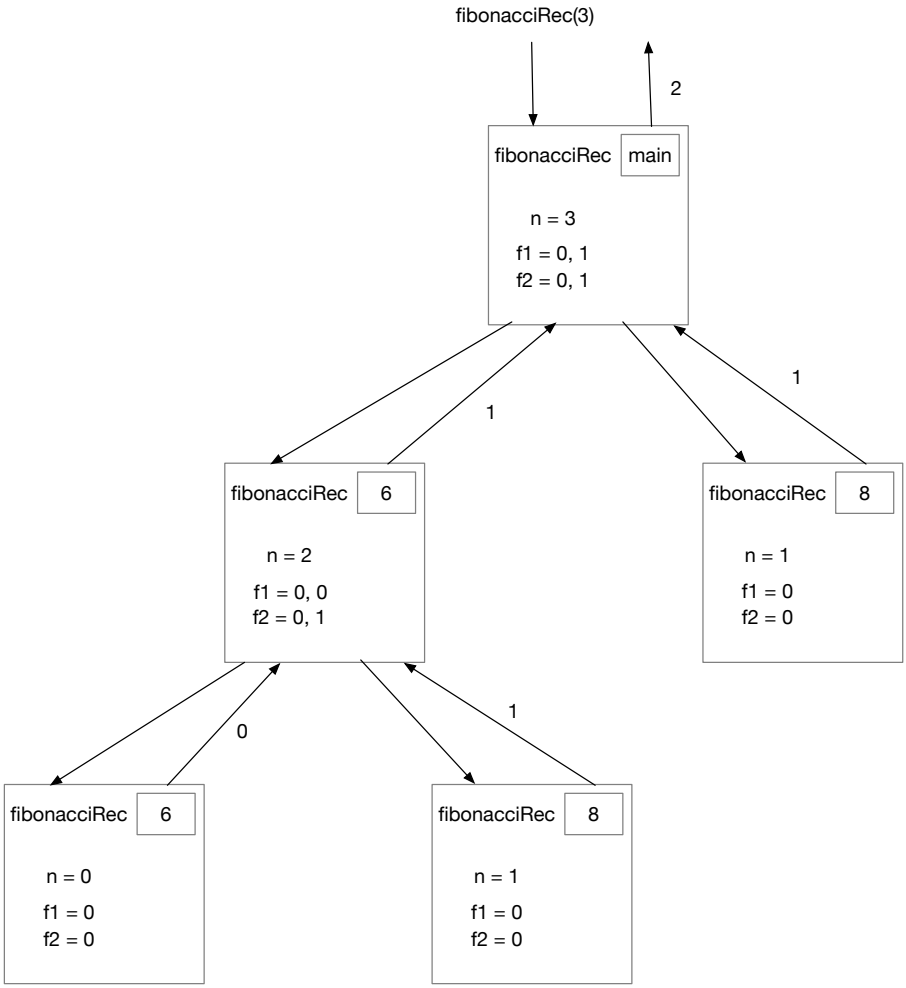
```
package fibonacci;

import stack.LinkedStack;


public class Fibonacci {

    private static class Context {
        int n;
        int f1;
        int f2;
        EntryPoint entryPoint;

        public Context(int n) {
            this.n = n;
            this.f1 = 0;
            this.f2 = 0;
            this.entryPoint = EntryPoint.CALL;
        }
    }

    private enum EntryPoint {
        CALL, RESUME1, RESUME2
    }

    public static int fibonacciOrig(int n) {
        if (n <= 1)
            return n;
        else
            return fibonacciRec(n - 1) + fibonacciRec(n - 2);
    }

    public static int fibonacciRec(int n) {
        // CALL
        if (n <= 1)
            return n;
        else {
            int f1 = fibonacciRec(n - 1);
            // RESUME1
            int f2 = fibonacciRec(n - 2);
            // RESUME2
            return f1 + f2;
        }
    }
```

```
    public static int fibonacciIter(int n) {
        int return_ = 0;
        var stack = new LinkedStack<Context>();
        stack.push(new Context(n));
        while (!stack.isEmpty()) {
            var context = stack.top();
            switch (context.entryPoint) {
                case CALL -> {
                    if (context.n <= 1) {
                        return_ = context.n;
                        stack.pop();
                    } else {
                        context.entryPoint = EntryPoint.RESUME1;
                        stack.push(new Context(context.n - 1));
                    }
                }
                case RESUME1 -> {
                    context.f1 = return_;
                    context.entryPoint = EntryPoint.RESUME2;
                    stack.push(new Context(context.n - 2));
                }
                case RESUME2 -> {
                    context.f2 = return_;
                    return_ = context.f1 + context.f2;
                    stack.pop();
                }
            }
        }
        return return_;
    }
}
```

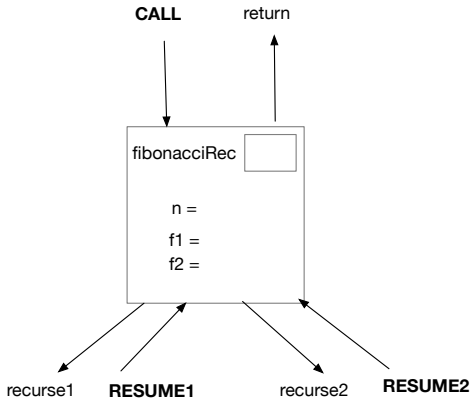
Tened en cuenta que en el caso del fibonacciRec, cada llamada que no se corresponde con un caso simple, acaba realizando dos llamadas recursivas

```
1. public static int fibonacciRec(int n) {
2.     // CALL
3.     if (n <= 1)
4.         return n;
5.     else {
6.         int f1 = fibonacciRec(n - 1);
7.         // RESUME1
8.         int f2 = fibonacciRec(n - 2);
9.         // RESUME2
10.        return f1 + f2;
11.    }
12. }
```

por lo que las llamadas tienen la estructura:



Es decir, cada llamada tiene dos recurse, un CALL y dos RESUME, es decir:



Lo que se pide en este apartado es que expliquéis de manera similar a como se ha ilustrado la versión iterativa del factorial, es decir, con diagramas, paso a paso, con extractos de código y su explicación, etc, etc, el caso de la versión iterativa de la función fibonacciRec, llamada fibonacciIter, que tenéis en el código anterior.

Ejercicio 3: El problema de las particiones de un número natural

Este problema, que se describe en el anexo que se os adjunta (y que forma parte de los apuntes de la asignatura Programación 2) tiene una solución recursiva múltiple de la que partiremos y, la tarea a realizar será la de transformarla en iterativo **usando la metodología explicada en el apartado anterior.**

Repetimos, se pide que apliquéis dicha metodología, no que busquéis una solución por internet y que la entreguéis.

El código de partida es el siguiente:

```
public static int numPartitionsRec(int sum) {
    // Precondition: sum > 0
    int count = 0;
    for (int numParts = 1; numParts <= sum; ++numParts) {
        count += numPartitionsRec(sum, numParts);
    }
    return count;
}

public static int numPartitionsRec(int sum, int numParts) {
    // Precondition: sum > 0
    int np1, np2;
    // CALL
    if (numParts <= 0 || numParts > sum) {
        return 0;
    } else if (numParts == sum) {
        return 1;
    } else {
        np1 = numPartitionsRec(sum - 1, numParts - 1);
        // RESUME1
        np2 = numPartitionsRec(sum - numParts, numParts);
        // RESUME2
        return np1 + np2;
    }
}
```

La primera de las funciones ya es iterativa, por lo que la única cosa que deberemos hacer es cambiarle el nombre y hacer que llame a la versión iterativa de la segunda, es decir:

```
public static int numPartitionsIter(int sum) {
    // Precondition: sum > 0
    int count = 0;
    for (int numParts = 1; numParts <= sum; ++numParts) {
        count += numPartitionsIter(sum, numParts);
    }
    return count;
}
```

Y ahora lo que deberemos hacer es la siguiente función:

```
private static int numPartitionsIter(int sum, int numParts) {
    // Precondition: sum > 0
    // Transformación a iterativo de la función
    // numPartitionsRec(int sum, int numParts)
}
```

Para ello deberéis crear una clase estática interna Context, definir el enumerado con los EntryPoints, etc, etc.

Es decir, en este apartado aplicaréis la metodología a un caso nuevo y explicaréis las diferencias con la solución del problema del Fibonacci que habéis descrito en el apartado anterior.

Una idea para hacer pruebas

Cuando lo que queremos es obtener una versión diferente de una función que ya tenemos (por ejemplo, buscando optimizaciones o, en este caso, pasando de una versión recursiva a una iterativa), una forma de hacer pruebas (obviamente suponiendo que la versión original ya está suficientemente comprobada) es usar la técnica de los **golden tests**, que consiste en comprobar que las dos funciones devuelven el mismo resultado para los mismos valores de los parámetros. Por ejemplo:

```
package partitions;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class PartitionsGoldenTest {

    public static final int MAX = 20;

    @Test
    void golden() {
        for (int sum = 1; sum <= MAX ; sum++) {
            var rec = Partitions.numPartitionsRec(sum);
            var ite = Partitions.numPartitionsIter(sum);
            assertEquals(rec, ite);
        }
    }
}
```

Hemos definido MAX como 20 pues, al ser ambas versiones algoritmos poco eficientes, para valores mayores el test tardaba demasiado.

Posibles extensiones

Como toda práctica, hay flecos de los que tirar para hacer posibles ampliaciones y mejorar vuestros conocimientos. Aquellos que queráis aprender más sobre el tema os proponemos:

- Comparar, usando jmh, el rendimiento de las versiones recursivas e iterativas de los algoritmos.
- Estudiar si varía el rendimiento cuando implementamos las pilas, en vez de manera enlazada, de manera contigua y con una política de redimensionado similar a las de las listas vistas en clase.
- Aplicar la transformación a otros algoritmos recursivos como el QuickSort o las Torres de Hanoi.
- Podéis definir una subclase de pila que guarde la altura máxima que ha tenido (y que disponga de un método para obtenerlo). Podéis realizar transformaciones del algoritmo de QuickSort encontrado para intentar hacer que el espacio máximo ocupado por la pila disminuya.
    - ¿Empeoran esos cambios el rendimiento?
- Un subconjunto de algoritmos recursivos simples son los llamados: algoritmos recursivos finales (tail recursive). En ellos, en los casos recursivos, el resultado de la llamada recursiva se devuelve tal cual, sin transformarlo. Por ejemplo:

```
public static int factorial(int n) {
    return factorialTR(n, 1);
}

public static int factorialTR(int n, int acc) {
    if (n == 0)
        return acc;
    else
        return factorialTR(n - 1, acc * n);
}
```

En este caso, si aplicáis la transformación que hemos explicado, podréis transformar (razonando) el código resultante, eliminando completamente la pila.
A partir de este caso particular, podéis empezar a explorar lo que se llama como Tail Call Optimization, que consiste en, reaprovechar el contexto de ejecución existente en la cima de la pila si la llamada recursiva es lo último que se realiza.

A partir de aquí, una vez entendemos que podemos representar el contexto de ejecución de una llamada como una pila, podemos explorar

elementos que, de momento[7] no existen en Java, pero sí en Python, como son los generadores y las corrutinas.

Entrega

El resultado obtenido debe ser entregado por medio de un único fichero comprimido en **formato ZIP** por cada grupo con el nombre "Lab3_NombreIntegrantes". Dicho fichero contendrá el **proyecto IntelliJ** (con las clases en los paquetes indicados por el enunciado) y el documento en **formato PDF** con el informe.

Por cierto, en la primera práctica nos hemos encontrado con entregas que contenían proyectos dentro de proyectos y otras cosas extrañas. Si no os acordáis de cómo generar un proyecto, cómo crear el directorio de test, etc., etc. consultad las presentaciones de laboratorio (tanto de Estructuras de Datos como de Programación 2) o preguntad.

Consideraciones de Evaluación

A la hora de evaluar el laboratorio se tendrán en cuenta los siguientes aspectos:

- Solución a los ejercicios planteados.
- **Razonamientos y justificaciones**.
- Calidad y limpieza del código.

[7] Aunque loom, uno de los proyectos en los que se están desarrollando elementos a incorporar en las nuevas versiones de Java, posiblemente en un futuro las incorpore.