Enunciados de las prácticas

Programación II

Grado de Ingeniería de Sistemas de Información

Conocimientos y destrezas a adquirir por los estudiantes

Se espera que, una vez finalizadas las prácticas, el estudiante sea capaz de:

1. Manejar bucles (tanto simples como anidados) para resolver problemas.
2. Manejar Arrays multidimensionales.
3. Implementar una estructura de datos.
4. Utilizar estructuras de datos.
5. Implementar algoritmos recursivos.
6. Manejar expresiones lambda.
7. Construir software mediante desarrollo dirigido por pruebas (TDD).

Material a entregar por los estudiantes

En el repositorio de entrega deberá aparecer el siguiente material:

1. El `makefile` o el proyecto Maven, con, al menos, las opciones de compilar y generar el Javadoc.
2. Fichero `README.md` en formato `Markdown` con las notas oportunas tanto para los usuarios del programa como para los desarrolladores.
3. El fichero `.gitignore` del repositorio.

En la práctica del grafo, en lugar del fichero `makefile`, es necesario entregar el proyecto Maven.

Criterios de evaluación

Tanto el peso que tiene en la evaluación de la asignatura como el umbral de los distintos aspectos de la práctica (documentación, uso de herramientas de soporte al desarrollo y producto funcionando) están publicados en la guía docente. Los pesos de las prácticas serán los que se muestran a continuación:

1. Práctica 1: 5% del total de las prácticas.

2. Práctica 2: 5% del total de las prácticas.
3. Práctica 3: 5% del total de las prácticas.
4. Práctica 4: 30% del total de las prácticas.
5. Práctica 5: 55% del total de las prácticas.

Se aplicarán los siguientes criterios de evaluación:

1. El producto debe funcionar correctamente para alcanzar la puntuación de aprobado.
2. Tendrán importancia en la evaluación los siguientes aspectos del código: legibilidad, eficiencia, extensibilidad, etc.
3. Se valorará tanto la documentación en Javadoc como el `README.md` y, en caso de que sean necesarios, los comentarios adicionales, por ejemplo, aquéllos que están dentro de los cuerpos de los métodos.
4. Se valorará la gestión de repositorios. Para ello, el profesor podrá consultar los *commits* realizados a los repositorios utilizados.

En la práctica del grafo se valorará que las pruebas sean las adecuadas para el sistema. Deberán estar documentadas y su código deberá satisfacer los criterios de calidad expuestos en los puntos anteriores para el software a desarrollar.

Forma de entrega

Los materiales deberán ser subidos a un repositorio remoto (al que debe tener acceso el profesor). **NO** se corregirá material que no haya sido subido al repositorio.

Las fechas de las entregas serán publicadas por el profesor en el portal del alumno. **NO** se corregirá a partir del último *commit* dentro del plazo de entrega.

NO se permitirán formatos propietarios, por ejemplo, RAR.

Licencia

El estudiante será responsable de asegurar que las entregas se ajustan a la licencia explicada a continuación y que ha respetado las licencias de los recursos que haya reutilizado.

Licencia Apache versión 2

El proyecto estará bajo licencia Apache versión. Se deberá incluir un fichero, llamado `LICENSE` con el siguiente contenido, disponible en la página Web de la licencia:

```
Copyright [año] [nombre del propietario del copyright]
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
```

Enunciados de las prácticas del método de Montecarlo

Práctica 1. Obtención de una aproximación iterativa al número pi

Implemente un programa iterativo que permita obtener una aproximación al número pi mediante el método de Montecarlo.

El algoritmo estará implementado en la clase `Matematicas.java` de acuerdo con la siguiente estructura:

```java
package mates;

public class Matematicas{
    /**
     * Genera una aproximación al número pi mediante el método de
     * Montecarlo. El parámetro `pasos` indica el número de puntos
     * generado.
     */
    public static double generarNumeroPiIterativo(long pasos){
        return 0; // Este código hay que cambiarlo.
    }
}
```

El programa principal para mostrar el resultado es el siguiente:

```java
package aplicacion;

import mates.Matematicas;

public class Principal{
    public static void main(String[] args){
        System.out.println("El número PI es " +
            Matematicas.generarNumeroPiIterativo(Integer.parseInt(args[0])));
    }
}
```

Práctica 2. Obtención de una aproximación recursiva al número pi

Implemente un programa recursivo con la misma funcionalidad que el de la práctica 1 utilizando también simulación por Montecarlo.

Práctica 3. Obtención de una aproximación al número pi mediante expresiones lambda.

Implemente un programa mediante expresiones lambda con la misma funcionalidad que la práctica 1 utilizando también simulación por Montecarlo.

Práctica 4. Cálculo de la distancia de edición

Implemente en Java la distancia de edición entre dos palabras según lo expuesto las diapositivas de Dan Jurafsky: https://web.stanford.edu/class/cs124/lec/med.pdf. No es necesario que almacene la traza para transformar una palabra en otra.

El algoritmo estará implementado en la clase `CalculadoraDeDistancias.java` de acuerdo con la siguiente estructura:

```
public class CalculadoraDistancias{
    /**
     * Calcula la distancia de edición, tal y como la expone
     * Jurafsky entre las cadenas `s1` y `s2`.
    */
    public static int calcularDistancia(String s1, String s2){
        return 0; // Este código hay que cambiarlo.
    }
}
```

Práctica 5. Cálculo del camino más corto entre dos vértices

Implemente, siguiendo un desarrollo dirigido por pruebas, una estructura de datos de grafo según el siguiente esquema:

```
package pr2;

public class Graph<V>{

    //Lista de adyacencia.
    private Map<V, Set<V>> adjacencyList = new HashMap<>();

    /*****************************************************************
     * Añade el vértice `v` al grafo.
     *
```

```
     * @param v vértice a añadir.
     * @return `true` si no estaba anteriormente y `false` en caso
     *         contrario.
     ************************************************************************/
    public boolean addVertex(V v){
        return true; //Este código hay que modificarlo.
    }

    /************************************************************************
     * Añade un arco entre los vértices `v1` y `v2` al grafo. En
     * caso de que no exista alguno de los vértices, lo añade
     * también.
     *
     * @param v1 el origen del arco.
     * @param v2 el destino del arco.
     * @return `true` si no existía el arco y `false` en caso contrario.
     ************************************************************************/
    public boolean addEdge(V v1, V v2){
        return true; //Este código hay que modificarlo.
    }

    /************************************************************************
     * Obtiene el conjunto de vértices adyacentes a `v`.
     *
     * @param v vértice del que se obtienen los adyacentes.
     * @return conjunto de vértices adyacentes.
     ************************************************************************/
    public Set<V> obtainAdjacents(V v) throws Exception{
        return null;  //Este código hay que modificarlo.
    }

    /************************************************************************
     * Comprueba si el grafo contiene el vértice dado.
     *
     * @param v vértice para el que se realiza la comprobación.
     * @return `true` si `v` es un vértice del grafo.
     ************************************************************************/
    public boolean containsVertex(V v){
        return true; //Este código hay que modificarlo.
    }

    /************************************************************************
     * Método `toString()` reescrito para la clase `Grafo.java`.
     * @return una cadena de caracteres con la lista de
     * adyacencia.
     ************************************************************************/
```

```
    @Override
    public String toString(){
        return ""; //Este código hay que modificarlo.
    }

    /**
     * Obtiene, en caso de que exista, el camino más corto entre
     * `v1` y `v2`. En caso contrario, devuelve `null`.
     *
     * @param v1 el vértice origen.
     * @param v2 el vértice destino.
     * @return lista con la secuencia de vértices del camino más corto
     * entre `v1` y `v2`
     */
    public List<V> shortestPath(V v1, V v2){
        return null; // Esto código hay que modificarlo.
    }

}
```

El código debe pasar, al menos, la siguiente prueba:

```
    @Test
    public void shortestPathFindsAPath(){
        System.out.println("\nTest shortestPathFindsAPath");
        System.out.println("---------------------------");
        // We build the graph
        Graph<Integer> g = new Graph<>();
        g.addEdge(1, 2);
        g.addEdge(1, 5);
        g.addEdge(2, 3);
        g.addEdge(3, 4);
        g.addEdge(5, 4);

        // We build the expected path
        List<Integer> expectedPath = new ArrayList<>();
        expectedPath.add(1);
        expectedPath.add(5);
        expectedPath.add(4);
        //We check if the returned path is equal to the expected one.
        assertEquals(expectedPath, g.shortestPath(1, 4));
    }
```