Ejercicios Básicos de Autodiagnóstico.

1. [Conceptos: Árboles y Expresiones]. Escriba en notación post-fija. `a / (b * -c) / 2 + 10`. Asuma operadores y precedencia de Java. Dibuje el árbol de la expresión antes de generar su post-fija.

2. Muestre (en "papel-y-lápiz") y asumiendo que en memoria `a=100, b = 20 y c = 10` (todos de tipo `int`) y usando una pila el resultado paso a paso de la evaluación de la expresión post-fija de la pregunta anterior. Evalué en Java y compruebe su resultado.

3. [Java-Expresiones Regulares]. Asuma que un número de placa en `CR` puede ser: o a) una secuencia de entre 3 y 10 dígitos (inclusive ambos), nunca siendo el primero un cero o b) una secuencia de tres letras mayúsculas, seguida de un guion, seguida de exactamente tres dígitos. Escriba una `RE` que represente estás hileras. Use el API de Java. Su `RE` no puede tener más de 25 caracteres de largo vista como hilera.
   Ejemplos: "`109456`", "`ABC-666`" y "`666`" cumplen ser placas válidas. Pero no lo son "`600-ABC`" ni "`abC-123`". Pruebe su resultado haciendo una clase de prueba `Placas.java` en un paquete `com.eif400.diagnostico` como se muestra. Implemente el método `boolean placaCR(String hilera)` en una clase `Placas.java`. Compile y haga un `jar placas.jar` para correr su prueba. No use un IDE, trabaje en consola. Haga que esta clase funcione.

   Modelo de `Placa.java`

```java
package com.eif400.diagnostic;
import java.util.*;
import java.util.regex.*;

public class Placa{
    /**

    static public boolean placaCR(String placa){

    public static void main(String... args){
        List<String> placas = Arrays.asList(
            "109456",
            "ABC-666",
            "600-ABC",
            "666",
            "abC-123",
            "55",
            "600/ABC"
        );
        for (var placa : placas){
            System.out.format("Is %s valid? Answer: %b%n", placa, placaCR(placa) );
        }

    }
}
```

Salida esperada una vez construido el `jar` que tiene a Placa como clase de entrada.

```
DIAGNOSTICO:java -jar placas.jar
Is 109456 valid? Answer: true
Is ABC-666 valid? Answer: true
Is 600-ABC valid? Answer: false
Is 666 valid? Answer: true
Is abC-123 valid? Answer: false
Is 55 valid? Answer: false
Is 600/ABC valid? Answer: false

DIAGNOSTICO:
```

4. [Recursión]. El número `e` ≈ (2.718281828459045) se puede aproximar por esta sumatoria: $\sum_{i=0}^{n} 1/i!$. Escriba una función recursiva en `Java aproximeE(n)` que aproxime `e` usando dicha sumatoria. El número de multiplicaciones y divisiones debe ser $O(n)$. Si no es lineal en esa cantidad no se considera correcta.

5. [Recursión/Inducción]. Pruebe que la siguiente función en `Java` siempre retorna $2n + 5$ asumiendo que no haya problemas de `overflow` ni `stackoverflow`

```java
public int f(int n){
    if( n == 0 ) return 5;
    if( n == 1 ) return 7;
    return 2 * f( n - 1 ) - f( n - 2 );
}
```

6. [Estructuras Datos básicas $O(.)$]. Escriba un método en `Java` llamada `List<String> unico(List<String> a)`, que, dado una lista `a` de hileras, devuelva una lista que contenga exactamente aquellas que se encuentran solamente una vez en `a`. Debe ser $O(n)$ en tiempo de corrida para `n = a.size()`. Si no es lineal no estaría correcta.
   Ejemplo.
   Pruebe en una clase así usando assert:

```java
package com.eif400.diagnostic;

import java.util.*;

public class Unico{

    static public List<String> unico(List<String> a){

    public static void main(String... args){
        List<String> a = Arrays.asList("a", "t", "b", "a", "h", "v", "b", "t", "g");
        List<String> result = unico(a);
        Collections.sort(result);
        assert( result.equals(Arrays.asList("g", "h", "v")) );
        System.out.println("Test passes");
    }
}
```

Para correr use el flag -ea (enable assert).

7. [Java:Compilación y análisis de tipos]. Considere la siguiente clase `Problema`. Así como está sí compila. Pero si solo se descomentara toda la parte del método `foolist` comentada entonces ya no compila. Escriba un fuente `Problema.java` compile de consola y verfique que ocurre lo afirmado. Explique el por qué en forma clara y concisa.

```
import java.util.*;
public class Problema{
   public void fooArray(){
        String[] as = new String[0];
        Object[] os = as;

   }
   /* public void fooList(){
        List<String> ls = new ArrayList();
        List<Object> lo = ls;
   } */
   public static void main(String[] args){

   }
}
```

8. Considere las siguientes clases en Java. Explique qué valores se imprimiría en las líneas marcadas como "`Caso 1`)" hasta "`Caso 5)`". Justifique su respuesta en cada caso. Su solución a esta pregunta está correcta sólo si todas sus respuestas (en cada caso) son correctas.

```java
class A {
  public void foo(A a){
    System.out.println("foo(aa)");
  }
  public void foo(B b){
    System.out.println("foo(ab)");
  }
}
class B extends A {
  public void foo(A a){
    System.out.println("foo(ba)");
  }
  public void foo(B b){
    System.out.println("foo(bb)");
  }
}
public class F {
   public static void main(String[] args){
       A a  = new A();
       A b1 = new B();
       B b2 = new B();
       a.foo(b1);                          // Caso 1)
       b1.foo(b1);                         // Caso 2)
       b2.foo(b2);                         // Caso 3)
       System.out.println(b2 == (A)b2); // Caso 4)
       b2.foo((A)b2);                      // Caso 5)
   }
}
```

9. [Java OOP-Grafos]. Estudie el código adjunto en la carpeta `graph`. Implemente de manera **no recursiva** un recorrido dfs en el método `DfsResult<T> dfs(Node<T> start)`. Su método debe permitir la salida indicada abajo. Posicionado en el directorio graph y una vez que haya implementado su método y compilado su código:

   A) Dibuje a mano el grafo que se está probando
   B) Logre la salida mostrada según se le pide
   C) Repita pero ahora implemente recursivamente. Deben salir los mismos resultados

```
DIAGNOSTICO:java -cp classes com.eif400.dignostic.SimpleDigraph
*** Very Simple Demo of DFS ***
*** Visit starting at a ***
Visiting a successors=[c, b, x]
Visiting x successors=[c, x]
Visiting c successors=[d]
Visiting d successors=[]
Visiting b successors=[a, c]
Visiting c successors=[d]
*** Results of DFS started at a ***
a with number 1
x with number 2
c with number 3
d with number 4
b with number 5
c with number 6
*** Following nodes were not visited ***
z
y

DIAGNOSTICO:
```

10. [JS-Scope] Considere las siguientes declaraciones en `JS`. Explique qué valores imprimirían las líneas marcadas con "`Caso 1)`" y "`Caso 2`". Justifique su respuesta en cada caso.

```
var f = [];
for (var i = 0; i < 10; i++) {
  f[i] = function() { return i; }
}

console.log("caso 1 = " + f[5]()); // Caso 1)

var a = "a";
function g() {
  if ( !a ) { var a = "b"; }
  return a;
}

console.log("caso 2 = " + g());     // Caso 2)
```

11. [JS-Herencia prototípica] Considere las siguientes clases en `Java` (ignorando problemas de propiedades públicas). Reescriba en `JS` un modelo "análogo" usando herencia prototípica pero sin usar ES6 o superior (use JS vainilla). Debe poder explicar precisamente porqué funciona su solución.

```
class A {
    public int s;
    public A(int s){
        this.s = s;
    }
}

class B extends A {
    public int w, h;
    public B(int w, int h){
        super(2);
        this.w = w;
        this.h = h;
    }
    public int f(){
        return w * h;
    }
}
```