1) Utilizando terminología del paradigma de la Programación Funcional, explique la evaluación de la siguiente expresión escrita en Clojure e indique el resultado: 1. / 2

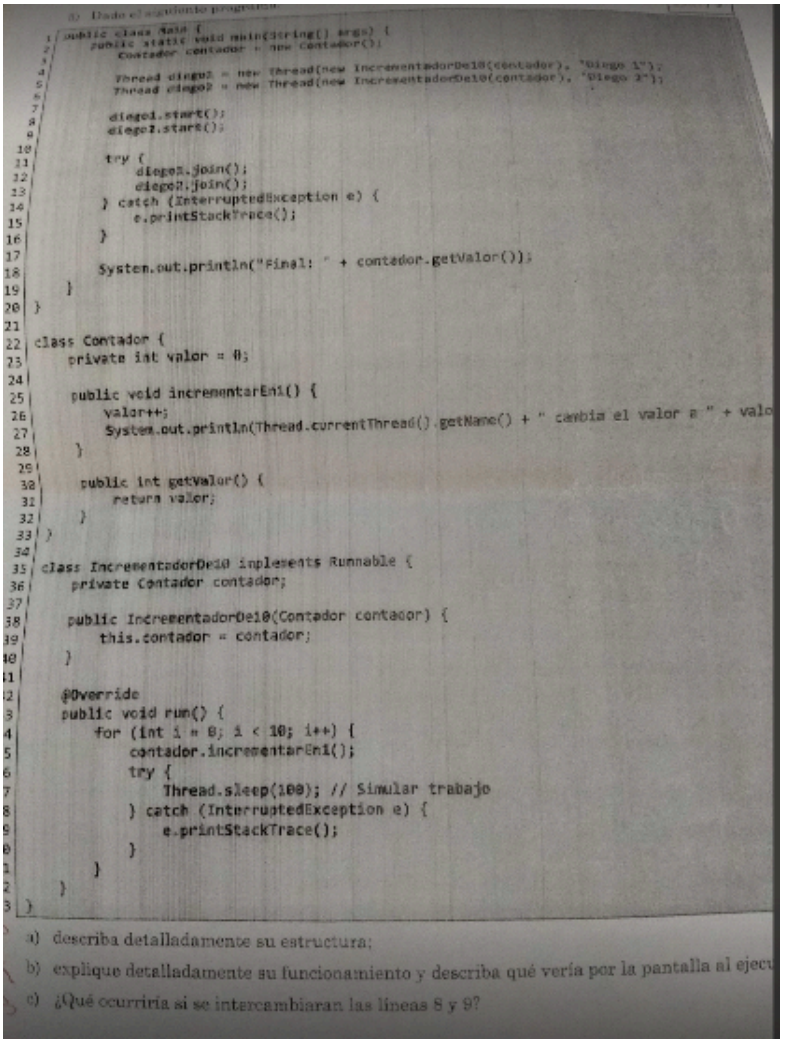
a)

```
(filter #(> % 5) (sort (map #(Float/parseFloat %) ["1" "8" "6" "1.4" "3.5" "7"])))
```

b) Justifique cuál es el error algorítmico que afecta negativamente la eficiencia del código anterior.

a) En este código tenemos: (map #(Float/parseFloat %) que se encarga de transformar la cadena de textos en números del tipo Float. Por otra parte sort, que se encarga de ordenar los números resultantes en forma ascendente. Por último filter #(> % 5) que selecciona y filtra a la cadena, los números menores a 5, dejando los mayores a este (osea, quita de la cadena a los que no cumplen la condición puesta)

b) Es el orden de las tareas lo que hace que este código pierda eficiencia, ya que el hecho de ordenar los elementos y pasarlos a float, para luego recién filtrar la cadena con los números deseados, hace que este código tenga una alteración innecesaria en su eficiencia temporal. Debería primero filtrar los elementos de la cadena y luego recién ahí ordenar y convertirlos en float.



```
public class Main {
    public static void main(String[] args) {
        Contador contador = new Contador();

        Thread diego1 = new Thread(new IncrementadorDe10(contador), "Diego 1");
        Thread diego2 = new Thread(new IncrementadorDe10(contador), "Diego 2");

        diego1.start();
        diego2.start();

        try {
            diego1.join();
            diego2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final: " + contador.getValor());
    }
}

class Contador {
    private int valor = 0;

    public void incrementarEn1() {
        valor++;
        System.out.println(Thread.currentThread().getName() + " cambia el valor a " + valo
    }

    public int getValor() {
        return valor;
    }
}

class IncrementadorDe10 implements Runnable {
    private Contador contador;

    public IncrementadorDe10(Contador contador) {
        this.contador = contador;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            contador.incrementarEn1();
            try {
                Thread.sleep(100); // Simular trabajo
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

a) describa detalladamente su estructura;

b) explique detalladamente su funcionamiento y describa qué vería por la pantalla al ejecu

c) ¿Qué ocurriría si se intercambiaran las líneas 8 y 9?

a) El programa comienza con una función main, donde de manera concurrente se crean dos tareas que son ejecutadas en hilos separados. Por otra parte tenemos a las clases contador e incrementador que son invocadas en el main, y que se encargan de realizar el conteo e incremento de en valor (inicialmente establecido en 0). A su vez se implementa a Runable que se encarga de realizar la evaluación y ejecución del tiempo de incremento.
b) El programa comienza con la ejecución de los hilos Diego 1 y 2 (Y sus tareas) de manera independiente. La ejecución se iniciará en cero y se interrumpe en 100 milisegundos. En ese intervalo, el valor se irá incrementando desde 0 hasta un valor menor al 10 establecido, y allí finalizara su ejecución. De esta manera, y como dijimos antes, en hilos separados, se mostrará los valores de los incrementos que tomarán Diego 1 y 2, hasta finalizar su ejecución y mostrar el valor final que estos tomaron.
c) No, no cambiaría ni afectará en nada el resultado ya que los hilos/threads Diego 1 y Diego 2 trabajan en paralelo y de manera independiente. La disposición original llama primero a Diego 1 y después a Diego 2 , pero esto solo “asegura” que el programa principal espere que ambos hilos terminen con sus tareas antes de continuar. Así que cambiar el orden de estas referencias ni a la ejecución ni al resultado del programa, ya que corren de manera independiente desde el momento de su ejecución.