TP - Unidad 01

Complejidad Algorítmica Temporal y Espacial

Ejercicio 1

Se pide implementar en Java la **clase MyTimer** (sin uso de otras bibliotecas relativas al tiempo, from scratch). La misma resultará de gran utilidad para calcular la complejidad temporal de algoritmos **empíricamente.**

La clase MyTimer representa la duración de un **intervalo temporal cerrado-abierto [*start*, *end*)** donde ***start*** pertenece al intervalo pero ***stop*** no.

Ej:

[40 ms, 40 ms) => duración 0 ms

[40 ms, 41 ms) => duración 1 ms

[40 ms, 42 ms) => duración 2 ms

[40 ms, 39 ms) => inválido

Básicamente, dicha clase MyTimer tendrá:

- el constructor MyTimer() que da inicio al mismo (el valor de inicio se calcula automáticamente)
- el método stop() detiene el timer y da fin al intervalo, es decir, dicho valor ya no es parte del mismo. (el valor de fin se calcula automáticamente).
- El método toString() que devuelve la duración del intervalo en ms y además el detalle de su duración en días, horas, minutos y segundos con fracción se segundos con 3 decimales. Ver ejemplo.

Para la detección automática del tiempo actual en ms sólo se puede usar: **System.currentTimeMillis() o System.nanoTime().** Cualquier problema que se detecte deberá lanzar **RuntimeException**.

Caso de Uso:

```
public static void main(String[] args) {
   MyTimer timer = new MyTimer();
   // bla bla bla ….. aca se invocaría el algoritmo cuyo tiempo de ejecución quiere medirse
  timer.stop();
  System.out.println(timer);
}
```

Salida esperada ms con 3 decimales

(93623040 ms) 1 día 2 hs 0 min 23,040 s

Ejercicio 2

El problema de API Timer es que resulta difícil chequear si funciona correctamente. Por ejemplo, generar una duración de **93623040 ms** para verificar que imprime 1 día 2 hs … y no 26 hs ….

Podríamos agregar

```
public static void main(String[] args) {
   MyTimer timer = new MyTimer();
   // bla bla bla ….. aca se invocaría el algoritmo cuyo tiempo de ejecución quiere medirse
  esperar(100000);
   timer.stop();
   System.out.println(timer);
}
private static void esperar(long ms) {
try {
            Thread.sleep(ms);
      }
catch (Exception e) {
         e.printStackTrace();
       }
}
```

Para esperar por un tiempo más largo, pero resultaría engorroso si quisiéramos probar si funciona correctamente por una duración mayor a unos pocos minutos.

Agregarle entonces los siguientes 2 métodos:

- el constructor **MyTimer(long inicio)** cuyo parámetro indica el comienzo del intervalo en ms (es decir, si se usa este constructor el inicio no se calcula automáticamente sino se usa el proporcionado).
- el método **stop(long fin)** detiene el timer y da fin al intervalo (no se calcula automáticamente). Como dijimos, el valor proporcionado como parámetro no es parte del intervalo.

Caso de Uso:

```
MyTimer t1= new MyTimer();
MyTimer t2= new MyTimer(long ms);
// bla bla bla
t1.stop();
// bla bla bla
t2.stop(long ms);
System.out.println(t1);
System.out.println(t2);
t1= new MyTimer();
// bla bla bla
t1.stop(long ms);
t2= new MyTimer(long ms);
// bla bla bla
t2.stop();
```

Como se observa, las 4 combinaciones son posibles. Es decir, podría comenzar y detenerse automáticamente; comenzar y finalizar con valor proporcionado por el usuario; comenzar automáticamente, pero finalizar con valor proporcionado por el usuario; comenzar con valor proporcionado por el usuario y finalizar automáticamente.

Esto da gran versatilidad de uso, pero sin embargo puede introducir casos no correctos de generación de intervalos. Ej: se inicia automáticamente, pero se lo detiene en un momento anterior al comienzo.

Contemplar todos los casos de errores posibles y **lanzar RuntimeException**, cuando sea necesario.

Ejercicio 3

3.1) Migrar la clase MyTimer anterior para que sea un proyecto Maven. Será nuestra primera versión.

```
  <groupId>ar.edu.itba.eda</groupId>
  <artifactId>TimerFromScratch</artifactId>
  <version>1</version>
```

Colocar explícitamente versión de Java esperada. Hay 2 formas para hacer eso en el pom.xml

a) Acá pondríamos 11 o la versión que deseemos y preferentemente su encoding.

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>
```

b) Agregarlo como plugin

```
<build>
<plugins>
 <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.0</version>
    <configuration>
         <release>11</release>
         <encoding>UTF-8</encoding>
    </configuration>
</plugin>
</plugins>
</build>
```

Probar alguna de esas opciones.

Probar con las fases **clean, compile (compiler:complie), package (jar:jar), install**.

3.2) Agregar al pom, en la parte de plugins

```
<build>
<plugins>
    <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-source-plugin</artifactId>
          <executions>
                    <execution>
                              <id>attach-sources</id>
                              <goals>
                                        <goal>jar</goal>
                              </goals>
                    </execution>
          </executions>
<version>3.2.1</version>
   </plugin>
 </plugins>
  </build>
```

Y generar mvn:install. ¿Qué genera?

Ejercicio 4

Existe una biblioteca denominada **joda-time**, la cual implementa lo necesario para implementar la clase **MyTimer**. En particular contiene las clases **Instant y Period** que resultan de gran utilidad.

4.1) Dado que es una **biblioteca externa**, para manejar mejor los versionados, crear un nuevo proyecto Maven y agregan dicha dependencia. Para diferenciarlo del proyecto anterior, llamarlo **TimerJoda al artefacto que generará. Será nuestra segunda versión.**

```
<groupId>ar.edu.itba.eda</groupId>
  <artifactId>TimerJoda</artifactId>
  <version>2</version>
...
```

Completar en el pom.xml la dependencia joda-time con la versión última estable.

Usar todo lo que hemos seteado en el pom.xml del ejercicio anterior.

4.2) Implementar la misma funcionalidad de la clase MyTimer (ejercicio 2) pero con Joda. Chequear que sigue funcionando correctamente, según lo esperado. Nuestra clase es un wrapper que expone la funcionalidad que teníamos definida.

Ejercicio 5

5.1) Escribir una nuevo proyecto Maven llamado **AnalysisTime**, el cual utilizará "la biblioteca Timer" que hemos generado en el ejercicio 4 como TimerJoda-2.jar. Para poder usarla, debemos haber ejecutado la opción mvn install (para que se escriba en el repositorio local y la pueda encontrar).

Usaremos la biblioteca, versión 2, para medir empíricamente tiempo. Ej: deberá tener

```
public class Proof{
        public static void main(String[] args) {
                MyTimer myCrono = new MyTimer(10);
                myCrono.stop(10 + 93623040);
                System.out.println(myCrono);
        }
}
```

Colocar en el pom.xml la dependencia con nuestra **TimerJoda-2.jar**. **No debe figurar la biblioteca que usamos indirectamente Joda Time**.

5.2) Ejecutar mvn install Ejecuta correctamente dentro de Eclipse/IntelliJ?

¿Cómo es que resuelve la biblioteca TimerJoda y también joda-time? Explicar

5.3) Analizar en el file system si **AnalysisTime-1.jar** contiene a las dos dependencias.

Borrar del repositorio mvn las bibliotecas .m2/repository/ar/edu/itba/eda/TimerJoda y .m2/repository/joda-time.

¿Ejecuta correctamente ahora?

5.4) Intentar ejecutarlo desde la línea de comandos (fuera del editor, como lo haría un usuario de dicha biblioteca). Abrir una terminal y ejecutar

(apuntarlo a donde se encuentre el jar)

$ set CLASSPATH=c:\Users\lgomez\.m2\repository\ar\edu\itba\eda\AnalysisTime\1\AnalysisTime-1.jar

$ java Proof -jar AnalysisTime-1.jar

Explicar el error obtenido

5.5) Imaginemos que queremos entregarle a un cliente la biblioteca **AnalysisTime-1.jar** pero el cliente no tiene por qué tener instaladas las otras 2 bibliotecas (por eso las borramos)

Si se quiere incluir en nuestro jar las dependencias, se debe agregar el siguiente plugin. Volver a generar **mvn install**. ¿Qué genera? ¿Cuánto ocupan los archivos jars? ¿Cual contiene las dependencias?

```
   <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
         <executions>
               <execution>
                        <phase>package</phase>
           <goals>
                        <goal>single</goal>
           </goals>
                 <configuration>
                              <descriptorRefs>
                              <descriptorRef>jar-with-dependencies</descriptorRef>
                              </descriptorRefs>
                       </configuration>
            </execution>
         </executions>
```

```
</plugin>
```

5.6) Intentar ejecutarlo desde la línea de comandos (fuera del editor, como lo haría un usuario de dicha biblioteca)

$ set CLASSPATH=c:\Users\lgomez\.m2\repository\ar\edu\itba\eda\AnalysisTime\1\AnalysisTime-1-jar-with-dependencies.jar

$ java Proof -jar AnalysisTime-1-jar-with-dependencies.jar

Explicar qué ocurre.

5.6) Si bien funcionó OK, resulta engorroso pedirle al usuario que tiene que indicar cómo se llama la clase que contiene el main (en nuestro caso Proof). Si lo ejecutamos así, no funciona

$ java -jar AnalysisTime-1-jar-with-dependencies.jar

Para poder incluir automáticamente el nombre de la clase principal, que contiene el main, cambiar así el plugin anterior

```
<plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <executions>
                <execution>
                        <phase>package</phase>
                                <goals>
                                        <goal>single</goal>
                                </goals>
                                <configuration>
                                <descriptorRefs>
                                        <descriptorRef>jar-with-dependencies</descriptorRef>
                                </descriptorRefs>
                                <archive>
                                                <manifest>
                                                        <mainClass>....</mainClass>
                                                </manifest>
                                </archive>
                                 </configuration>
                </execution>
                                </executions>
  </plugin>
```

Volver a generar **mvn install**.

Re ejecutar desde línea de comandos:

$ java -jar AnalysisTime-1-jar-with-dependencies.jar

Abrir el jar (con 7-zip, winzip, etc) y analizar el contenido del archivo Manifest.mf

Ejercicio 6

A partir de la versión 8, Java incorporó una nueva clase para manejar lo mismo que Joda: java.time.**Instant** y java.time.**Duration** **(**https://www.journaldev.com/2800/java-8-date-localdate-localdatetime-instant **)**

6.1) Aunque estas clases son parte de Java y no se precisan dependencias externas, crear un nuevo proyecto Maven. Llamarlo por ejemplo, **TimerNativo y será nuestra versión 3.**

<groupId>ar.edu.itba.eda</groupId>

<artifactId>TimerNativo</artifactId>

<version>3</version>

6.2) Re implementar las mismas funcionalidades de la clase **MyTimer**, haciendo uso de las clases que vienen a partir de esta versión.

Chequear que sigue funcionando según lo esperado.

6.3) Cuál de las APIs, Joda o Java nativo, resultó más fácil de utilizar? ¿Cuál estaba mejor documentada?

Ejercicio 7

Ahora tenemos 3 versiones de MyTimer: from scratch, la que usa Joda y la que usa Java 8 nativa.

Sin embargo, resulta bastante difícil garantizar que funcionan correctamente probando todo desde un "main". A medida que agreguemos más funcionalidad, podría arruinarse algo ya implementado y chequeado hasta ahora.

Para poder ejecutar test unitarios y ejecutarlos siempre, a medida que avance el proyecto, usaremos el framework **JUnit**. Desde un proyecto Maven es muy sencillo generar lo necesario para crear test unitarios.

Agregar a la version TimerFromScratch la dependencia

```
<dependencies>
<dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.8.0-M1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Y el plugin:

```
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <version>2.22.2</version>
 </plugin>
```

Y diseñar todos los testeos posibles (por ejemplo, para el proyecto TimerJoda). Tener también en cuenta que hay que chequear si se lanzan las RuntimeException que esperábamos.

Ejecutar los tests desde mvn elegir como goal :

surefire:test

y setear la lista de clases en parámetros:

test=clase1, clase2

o

test=*

Ejercicio 8

Como se ha observado, es bastante tedioso tener que comparar strings para saber si se obtiene lo esperado. Por ejemplo, quizás nuestro testeo de unidad declare:

@Test

void aTest() {

assertEquals("(93 623 040 ms) 1 día 2 hs 0 min 23,04 s", **timer.toString()**);

}

Y esta prueba no pasa el testeo porque incluimos un espacio blanco de más o una mayúscula en vez de una minúscula. Mejorar la clase MyTimer para que ofrezca métodos **getters**. De esta manera podremos chequear si un entero/double coincide con otro entero/double, etc.

@Test

void aTestDay() {

assertEquals(1, **timer.getDays()**);

}

@Test

```
void aTest() {
assertEquals(2,  timer.getHours());
}
Etc.
```

8.1) Agregar los **getters** y diseñar el testeo de unidad completo.

Ejemplo de chequeo de validaciones de Excepciones:

```
class TheTest {
@Test
void shouldThrowExcep() {
        MyTimer t = new MyTimer();
        assertThrows(RuntimeException.class,
                () -> {
        t.stop(3);
                });
}
@Test
void shouldThrowExcepMesage() {
        MyTimer t = new MyTimer();
        Throwable except= assertThrows(RuntimeException.class,
                () -> {
        t.stop(3);
                });
        assertEquals("Bad end", except.getMessage());
}
}
```

8.2) ¿Qué es el “coverage” el testeo y por qué es tan importante? Explicar

Ejercicio 9

9.1) Verificar que existe el **TimerFromScratch-1.jar** en el repositorio local de Maven ($HOME/.m2/repositories). En particular aparecerá ar.edu.itba.eda y dentro se encuentra **TimerFromScratch-1.jar**

9.2) Escribir el proyecto mvn ComplejidadTemporalAlgoritmica. Setear en el pom lo típico que hemos analizado y declarar que la dependencia que quiere usar es la versión TimerFromScratch-1.jar.

9.2) En Campus encontrarán 2 versiones de algoritmos que calculan el máximo elemento de un arreglo de enteros (**algoA.java y algoB.java**). Tambien encontrarán el archivo **Performance.jav**a. Incluirlos códigos.

Performance genera 2 timers y los usar para detectar la performance de algoA y algoB. Ejecutarlo y completar la siguiente tabla

n	**Time(algoA) en ms**	**Time(algoB) en ms**
1000		
100 000 000		
200 000 000		
400 000 000		
600 000 000		
800 000 000		
1 000 000 000		

Ejercicio 10

- Calcular T(algoA) en función del tamaño de entrada. Luego, calcular la complejidad temporal teórica para el peor caso (O grande, Big-O notation)
- Ídem con algoB. Consultar https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html
- ¿Cuál resulto mejor?

Ejercicio 11

Se ha logrado caracterizar que $T(N)= (N+1)^2$ .Probar formalmente (encontrando la constante c) que tiene complejidad $O(N^2)$

Ejercicio 12

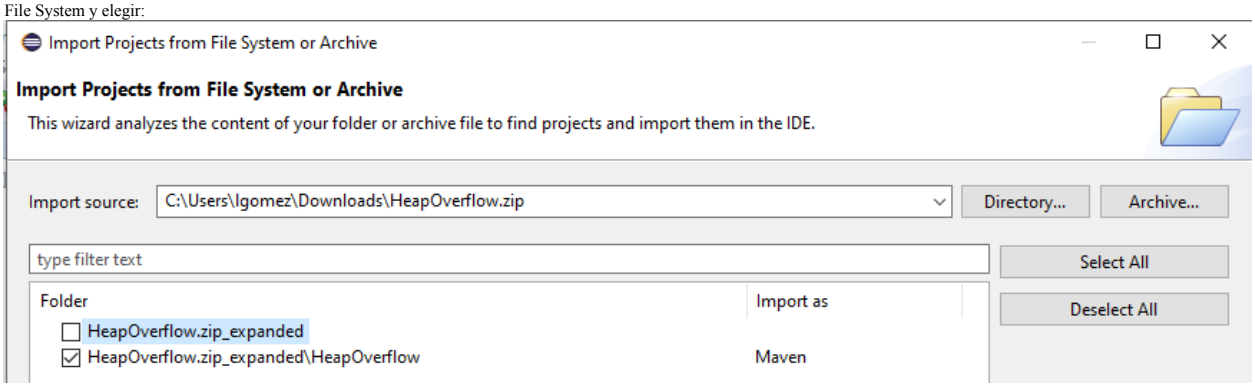
Caracterizar las siguientes complejidades O grande, para ver cuáles algoritmos serían mejores que otros.

Representar en una planilla de cálculo el gráfico del comportamiento para n= 0, 2, 4, 6, 8, 10 y 12 para las siguientes complejidades: $O(2^n)$, O(n*log2(n)), O(n), O(log2(n)), $O(^n)$, $O(n^3)$, $O(n^2)$

Ejercicio 13

Si queremos medir la complejidad espacial de un algoritmo tendremos que tener en cuenta, en el modelo de una aplicación Java, el espacio que precisa para su ejecución: Heap y Stack.

13.1) Bajar de Campus el proyecto HeapOverflow.zip que queremos que produzca heap overflow. Importarlo en el Java IDE como proyecto Maven. (en eclipse: Open Projects from File System y elegir:



13.2) Ejecutarlo y, según los recursos de su computadora, indicar en qué valor de "n" se produce heap overflow.

13.3) Cambiar explícitamente los parámetros de alocación del Heap para Java (son parámetros para la máquina virtual de Java)

Ejemplo: $ java -Xms512m -Xmx12G -jar HeapOverflow.jar

Y completar el siguiente cuadro indicando según los parámetros hasta cuanto resiste el valor de n

Parámetros	**Heap Overflow en n**
-Xms512m -Xmx1G	
-Xms512m –Xmx2G	
-Xms512m –Xmx4G	
-Xms512m –Xmx8G	
-Xms512m –Xmx12G	
-Xms512m -Xmx16G	

Ejercicio 14

14.1) Escribir una aplicación que genere solamente stack overflow.

14.2) Cambiar explícitamente los parámetros de Stack para Java (son parámetros para la máquina virtual de Java)

Ejemplo: $ java -Xss10k -jar StackOverflow.jar

Y completar el siguiente cuadro indicando según los parámetros hasta cuanto resiste el valor de n

Parámetros	**Stack Overflow**
-Xss10k	?
-Xss512m	?
-Xss1G	?

Etc	

Ejercicio 15

Escribir el método que calcule la suma de los primeros N números naturales en 2 versiones diferentes: iterativa y recursiva

15.1) Calcular la complejidad temporal y espacial para la versión iterativa **sumaIter(int n)**

15.2) Calcular la complejidad temporal y espacial para la versión recursiva **sumaRec(int n)**