Programación Eficiente — Trabajo Práctico Nro. 2

Maximiliano A. Eschoyez

Fecha de Entrega: Miércoles **17 de Noviembre de 2021**

Presentación del Trabajo Práctico

Conformación de los grupos

El trabajo práctico podrá realizarse en forma individual o grupal, prefiriéndose la modalidad de trabajo en equipo. Los grupos de trabajo podrán estar conformados por hasta dos (2) estudiantes, no siendo admisibles grupos con mayor número de integrantes.

Informe Escrito

Se debe presentar un informe escrito donde se describa la problemática abordada en el trabajo práctico, el desarrollo de la solución propuesta, los resultados de las mediciones pertinentes y los resultados logrados. El texto deberá ser conciso y con descripciones apropiadas. No se debe incluir el código fuente, sino los textos necesarios para realizar las explicaciones pertinentes.

Sólo se aceptarán informes en formato PDF.

Archivos Relacionados al Trabajo

Todos los archivos de código fuente, archivos utilizados en el desarrollo del trabajo y la versión digital del informe deben entregarse a través de la plataforma MiUBP (`http://mi.ubp.edu.ar/`). Además, se debe incluir una breve descripción sobre los archivos entregados y la estructura de directorios utilizada para organizar los archivos.

Sólo se aceptará la entrega en formato ZIP.

Aprobación del Trabajo Práctico

El trabajo práctico deberá estar completo el día de entrega estipulado. En caso de no presentación o presentación incompleta quedará como no aprobado. Si el docente considera que está aprobado pero tiene observaciones, se estipulará una nueva fecha de entrega para presentar las modificaciones indicadas oportunamente.

No se aceptarán informes incompletos y formatos de archivos distintos a los indicados.

Recursividad vs. Iteración

El objetivo de esta parte de Trabajo Práctico es revisar los temas abordados durante este período sobre *pruebas del sotfware basado en herramientas estándar* para comparar el comportamiento de algoritmos *recursivos* frente a su versión *iterativa*. Además, se realizará la comparación de las versiones *recursivas* para los casos de *Evaluación Ansiosa* (*eager*) y *Evaluación Perezosa* (*lazy*). Finalmente, se hará un estudio de los resultados obtenidos.

Caso 1: Recursión Simple

En este caso vamos a utilizar el cálculo del *Factorial* de un número para comparar la eficiencia entre la versión recursiva y la versión iterativa.

La definición matemática del *Factorial* de un número nos dice que:

$$n! = \begin{cases} 1 & \text{para } n = 0 \text{ o } n = 1 \\ n \cdot (n-1)! & \text{para } n > 0 \end{cases}$$

La función recursiva que resuelve esto es una codificación trivial de la función matemática y podría escribirse de esta forma:

```
1 unsigned long int Factorial (unsigned long int n) {
2   unsigned long int f = 0;
3   if ((n == 0) || (n == 1))
4     f = 1;
5   else if (n > 1)
6     f = n * Factorial(n - 1);
7   return f;
8 }
```

La versión iterativa podría escribirse de esta forma:

```
1 int Factorial (int n) {
2   int f = 1;
3   if (n < 0)
4     f = 0;
5   else if (n > 1)
6     for (int i = 2; i <= n; i++)
7       f *= i;
8   return f;
9 }
```

Testee las dos versiones para diferentes valores factoriales, trace las curvas con los tiempos medidos y responda las siguientes preguntas:

1. ¿Cuál de las dos es más eficiente?
2. ¿Cuál de las dos es más programable?
3. ¿El comportamiento de ambas versiones es lineal o exponencial?

Caso 2: Recursión Doble

En este caso vamos a utilizar el cálculo de los *Números de Fibonacci* para comparar la eficiencia entre la versión recursiva y la versión iterativa.

La definición matemática de los *Números de Fibonacci* nos dice que:

$$F_n = \begin{cases} 1 & \text{para } n = 1 \text{ o } n = 2 \\ F_{n-1} + F_{n-2} & \text{para } n > 2 \end{cases}$$

La función recursiva que resuelve esto es una codificación trivial de la función matemática y podría escribirse de esta forma:

```
int Fibonacci (int n) {
  int f = 1;
  if (n < 0)
    f = -1;
  else if (n > 2)
    f = Fibonacci(n - 2) + Fibonacci(n - 1);
  return f;
}
```

La versión iterativa podría escribirse de esta forma:

```
int Fibonacci (int n) {
  int f = 1, anterior = 1, actual = 1;
  if (n < 0)
    f = -1;
  else if (n > 2)
    for (int i = 3; i <= n; i++) {
      f = anterior + actual;
      anterior = actual;
      actual = f;
    }
  return f;
}
```

Testee las dos versiones para diferentes valores factoriales, trace las curvas con los tiempos medidos y responda las siguientes preguntas:

1. ¿Cuál de las dos es más eficiente?
2. ¿Cuál de las dos es más programable?
3. ¿El comportamiento de ambas versiones es lineal o exponencial?

Caso 3: Combinaciones

En este caso se debe explorar la implementación del cálculo de las *Combinaciones sin repetición*, cuya definición matemática es:

$$C(n,r) = \binom{n}{r} = \frac{n!}{r!(n-r)!} \quad \text{para } n \geq r$$

donde

$$\binom{n}{n} = \binom{n}{0} = 1$$

Se puede modificar la expresión matemática para que quede expuesta una versión recursiva:

$$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$$

En este caso se están presentando la versión iterativa de la solución como así también un método matemático equivalente para solucionar el problema. Se pide analizar los siguientes casos:

1. Combinaciones con Factorial Recursivo,
2. Combinaciones con Factorial Iterativo,
3. Combinaciones versión Recursiva.

Caso 4: Evaluación Ansiosa vs. Perezosa

La mayoría de los lenguajes y compiladores que utilizamos está pensado para el paso de variables por valor. Este tipo de ejecución se denomina *Evaluación Ansiosa* (*Eager Evaluation*). La evaluación ansiosa tiene como inconveniente que no se puede utilizar convenientemente la recursividad por limitación del tamaño de pila realizable y el alto consumo de recursos que requiere.

Por el contrario, los lenguajes y compiladores que instrumentan la *Evaluación Perezosa* (*Lazy Evaluation*) basan su funcionamiento en la dilación por el mayor tiempo posible de la evaluación de los cálculos necesarios. Este mecanismo permite implementar algoritmos recursivos y sobre conjuntos de datos *virtualmente infinitos* sin inconveniente y a un costo computacional razonable.

En este caso se deberá comparar el comportamiento de los algoritmos *Factorial de un número* y *Secuencia de Fibonacci* implementados con evaluación perezosa frente a los tiempos medidos con los algoritmos de los casos anteriores.

El *Factorial de un número* en Haskell se puede definir como

```
1 factorial :: Integer  -> Integer
2 factorial n
3         | n == 0 = 1
4         | n > 0  = n * factorial (n - 1)
5         | n < 0  = error "Valor negativo"
```

La *Secuencia de Fibonacci* en Haskell se puede definir como

```
1 fibonacci :: Integer -> Integer
2 fibonacci n
3          | n == 0 = 1
4          | n == 1 = 1
5          | n > 1  = fibonacci (n - 2) + fibonacci (n - 1)
6          | n < 0  = error "Valor negativo"
```

No olvide implementar los algoritmos para *Combinaciones sin repetición* en Haskell (solo versiones recursivas).

Nota

Para la realización de este estudio se aconseja desarrollar *shell scripts* que le permitan la repetibilidad de las pruebas a realizar.

Se pide que se midan los tiempos de ejecución de cada versión para diferentes valores de entrada. También se pide que se estudie el número de llamadas a las funciones para cada versión. Estos resultados deberán verse reflejados en gráficos comparativos y árboles de llamadas.

No olvide hacer las comparaciones para la compilación sin y con optimizaciones del compilador (`gcc` y `ghc` con `-O1`, `-O2` y `-O3`)

A la hora de entregar los resultados envíe el informe en formato PDF y los código fuente utilizados tanto de los programas como de los *shell scripts*.

Reutilización de Objetos

Garbage Collector

Los lenguajes que implementan el *garbage collector* fueron diseñados para disminuir la carga al programador al desligarlo de controlar la asignación/liberación de memoria. Esto hace que el programador solo deba preocuparse de pedir la creación de los objetos, utilizarlos y luego simplemente desreferenciarlos. La tarea de limpieza de la memoria queda a cargo de otro. Sin embargo, la "recolección de (memoria) basura" tiene costo computacional.

En los programas cuya memoria es bastante estática a lo largo de su ejecución, la tareas del *garbage collector* es baja. En cambio, si la tasa de asignación/liberación de memoria es muy alta, el "recolector de basura" tendrá mucho mayor trabajo, produciendo picos de consumo de recursos a la hora de realizar esta tarea.

La activación de la liberación de memoria tiene diferentes políticas y el programador puede pedir que se utilice alguna en particular. Sin embargo, no está a cargo del programador la liberación de memoria, por lo cual, se hará cuando el *garbage collector* determine que es momento apropiado.

Reutilización de Objetos

Si estamos frente a un programa que requiere gran generación/destrucción de objetos se puede optar por una política de reutilización de los objetos. En este caso, en lugar de crear un nuevo objeto cada vez que se lo necesite se reinicializará un objeto existente en desuso para adaptarlo a la nueva tarea. De esta forma se logran dos objetivos:

- se evita el costo computacional requerido por el constructor del objeto y
- se libera al *garbage collector* de su tarea de llamada al destructor y liberación de la memoria.

Por supuesto, para que la reutilización de objetos sea factible deberá mantenerse una referencia a los objetos en desuso dentro de algún tipo de colección.

Conigna

Para la realización de este estudio se aconseja trabajar sobre la IDE NetBeans utilizando el *profiler* provisto.

Se pide que estudie el comportamiento del programa "Zorros y Conejos". Este programa es un proyecto BlueJ que se entrega convertido en un proyecto con Maven para poder trabajar. El programa es una simulación que requiere una alta tasa de creación/destrucción de objetos (zorros y conejos) para estudiar el comportamiento de sus nacimientos/muertes.

Una vez estudiado el programa deberá planificar e implementar una propuesta de reutilización de objetos.

Se pide que se midan los tiempos de ejecución de cada versión. Estos resultados deberán verse reflejados en gráficos comparativos y árboles de llamadas.

Programación Funcional en Java

Desde la versión 8 de Java se pueden realizar operaciones propias de la Programación Funcional mediante las herramientas provistas en el paquete Stream.

El proyecto *Figuras Avanzadas* está implementado de forma tal que se comporta como un generador aleatorio de figuras geométricas simples y las gestiona desde una colección de tipo lista. Para poder comparar el comportamiento de un recorrido estándar sobre colecciones se implementaron las versiones con *for–each* y con *streams*.

Creación de las Figuras Geométricas

Las figuras se crean aleatoriamente por *reflection* en la clase `HerramientaDeAzar`

```
1  public FiguraGeometrica getFiguraAleatoria () {
2      FiguraGeometrica f;
3      Forma forma = Forma.values()[dado.nextInt(Forma.values().length)];
4      try {
5          Class<?> c = Class.forName("FigurasAvanzadas." +
6                              forma.toString());
7          Constructor<?> constructor = c.getConstructor(Integer.class);
8          f = (FiguraGeometrica)constructor.newInstance(getEntero(10, 500));
9          f.getPosicion().setX(getEntero(0, 800));
10         f.getPosicion().setY(getEntero(0, 600));
11     }
12     catch (Exception e) {
13         System.out.println(e);
14         throw new IllegalStateException(e.getMessage());
15     }
```

Este generador se invoca desde la clase `GestorFiguras` en versión con *for* tipo C

```
1      public void crearFigurasAleatoriamente(Integer cantidad) {
2          for (Integer i = 0; i < cantidad ; i++) {
3              try {
4                  agregarFiguraAleatoria();
5              } catch (IllegalStateException e) {
6              }
7          }
8      }
```

y en versión con *streams*

```
1      public void crearFigurasAleatoriamenteStream(Integer cantidad) {
2          Stream.generate(generador::getFiguraAleatoria)
3                .limit(cantidad).forEach(figuras::add);
4      }
```

Acciones sobre la colección

Para poder comparar recorridos sobre la lista, se aplican dos filtros. El primero, toma los elementos que poseen una superficie mayor a un valor dado y los coloca en una nueva lista. El segundo, toma los elementos de la clase `Rectangulo` mediante *type casting* y los coloca en una nueva lista.

El filtrado por superficie en versión *for–each* es

```
1  public List<FiguraGeometrica> listarMayoresA (Double superficie) {
2      // Variable bandera indica si se uso -> funciona como lacrado
3      List<FiguraGeometrica> lista = new ArrayList<>();
4      for (FiguraGeometrica f : figuras) {
5          if (f.getSuperficie() > superficie) {
6              lista.add(f);
7          }
8      }
9      return lista;
10 }
```

y en versión *streams* es

```
public List<FiguraGeometrica> listarMayoresAStream (Double superficie) {
    return figuras.stream()
                  .filter(x -> x.getSuperficie() > superficie)
                  .collect(Collectors.toList());
}
```

El filtrado por clase en versión *for–each* es

```
public List<FiguraGeometrica> listarRectangulos () {
    List<FiguraGeometrica> lista = new ArrayList<>();
    for (FiguraGeometrica f : figuras) {
        try {
            lista.add((Rectangulo)f);
        } catch (ClassCastException e) {
        }
    }
    return lista;
}
```

y en versión *streams* es

```
public List<FiguraGeometrica> listarRectangulosStream () {
    return figuras.stream()
                  .filter(f -> f.getClass() == Rectangulo.class)
                  .collect(Collectors.toList());
}
```

Actividad

Para este apartado, se solicita comparar los tiempos de ejecución, creación de objetos y cosumo de memoria para ambas implementaciones.