1 | Complejidad

Meta

Que el alumno visualice el concepto *función de complejidad computacional*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Medir la complejidad en número de operaciones de un método de manera experimental.
- Comparar el desempeño entre las versiones iterativas y recursivas de un método.
- Reportar formalmente los resultados de sus experimentos.

Código Auxiliar 1.1: Complejidad

https://github.com/computacion-ciencias/ed-complejidad-cs

Antecedentes

Sucesión de Fibonacci.

La sucesión de Fibonacci fue descubierta por Fibonacci en relación a un problema de conejos. Supongamos que se tiene una pareja de conejos y cada mes esa pareja cría una nueva pareja. Después de dos meses, la nueva pareja se comporta de la misma manera. Entonces, el número de parejas nuevas nacidas a_n en el n-ésimo mes es $a_{n-1} + a_{n-2}$, ya que nace una pareja por cada pareja nacida en el mes anterior y cada pareja nacida hace dos meses cria una nueva pareja. Por convención consideremos $a_0 = 0$ y $a_1 = 1$.

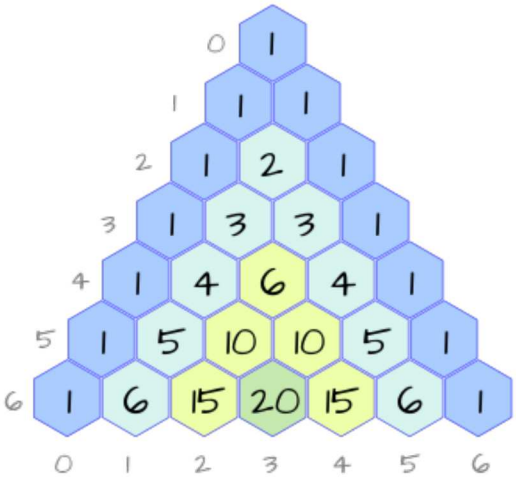

Figura 1.1 Triángulo de Pascal mostrando la enumeración de sus dos coeficientes.

Actividad 1.1

Define matemáticamente, con estos datos, la función de Fibonacci.

Entregable: La definición matemática en tu reporte.

Triángulo de Pascal.

En la figura Figura 1.1 se muestran algunos términos del Triángulo de Pascal.

Matemáticamente, podemos definir el elemento Pascal_{ij} que corresponde al elemento en la fila i, columna j de la siguiente manera:

$$\mathrm{Pascal}_{ij} = \begin{cases} 1 & \text{si } j = 0 \text{ ó } j = i \\ \mathrm{Pascal}_{(i-1)(j-1)} + \mathrm{Pascal}_{(i-1)(j)} & \text{En cualquier otro caso.} \end{cases} \tag{1.1}$$

Parte I

Desarrollo

La práctica consiste en implementar métodos que calculen el n-ésimo número de Fibonacci y el ij-ésimo elemento del triángulo de Pascal, al tiempo que estiman el número de operaciones realizadas. Esto se programará en forma recursiva e iterativa. Se deberá

implementar la interfaz `IComplejidad` en una clase llamada `Complejidad`. Se entregan pruebas unitarias para ayudar a verificar que estas funciones estén bien implementadas. Adicionalmente, deberán llevar la cuenta del número de operaciones estimadas en un atributo de la clase para generar, en la parte siguiente, un reporte ilustrado sobre el número de operaciones que realiza cada método.

Ejercicios

1. Crea la clase `Complejidad`, que implemente `IComplejidad`. Agrega las firmas de los métodos requeridos, si hay un valor de regreso distinto de `void` devuelve `null` ó `0` según se requiera y asegúrate de que el código compile, aunque aún no realize los cálculos.

2. Abre el archivo `UnitTestComplejidad.cs` en el directorio `ComplejidadNPruebas`. Lee el código. Observa que cada método marcado con la anotación `[Test]` se ejecuta como una prueba unitaria. Las expresiones `Assert.That` y `Is.EqualTo` se utilizan para verificar que el código devuelva el valor esperado. Por lo demás, el archivo contiene la definición de una clase común y corriente.

   Agrega cuatro métodos que prueben el funcionamiento de los cuatro métodos que programarás para calcular Pascal y Fibonacci, uno por método, **adicionalmente a los que ya están ahí**. Elige un número y calcula a mano la respuesta correcta, tus pruebas deberán mandar ejecutar el código y comparar su resultado con la respuesta que calculaste mediante un método. Si no utilizas la anotación `[Test]` y algún método de `NUnit` como `Assert.That` y sus restricciones como `Is.EqualTo` tus métodos no funcionarán con este sistema, así que no olvides estos elementos. En tu reporte indica cuáles son los métodos que agregaste.

   Es verdad, aún no tienes el código que calcula los números de las sucesiones, pero igualmente puedes programar las pruebas pues ya sabes cómo deberán comportarse las funciones una vez que estén hechas. De este modo podrás verficar que tu código sea correcto conforme lo vayas programando.

   Para saber más sobre la programación de pruebas unitarias revisa la documentación oficial del paquete `NUnit`.[1]

3. Programa los métodos indicados en la interfaz. **No uses las fórmulas matemáticas directas para calcular los elementos de las series**, el objetivo de esta práctica es medir la complejidad de las implementaciones según las definiciones de los elementos de las series, ya sea calculándolos con ciclos (iterativo) o recursivamente. Asegúrate de cumplir con las precondiciones y postcondiciones especificadas en la documentación de la interfaz.

[1] La documentación se pude consultar en https://docs.nunit.org/articles/nunit/writing-tests/attributes.html

Las pruebas unitarias te ayudarán a verificar tus implementaciones de Fibonacci y Pascal. Puedes compilar y ejecutar las pruebas desde el directorio `ed-complejidad-cs` con la siguiente instrucción:

```
dotnet test ComplejidadNPruebas/ComplejidadNPruebas.csproj
```

4. Agrégale un atributo a la clase para que cuente lo siguiente:

   Iterativos El número de veces que se ejecuta el ciclo más anidado. Observa que puedes inicializar el valor del atributo auxiliar al inicio del método y después incrementarlo en el interior del ciclo más anidado.

   Recursivos El número de veces que se manda llamar la función. Aquí utilizarás una técnica un poco más avanzada que sirve para optimizar varias cosas. Necesitarás crear una función auxiliar (*privada*) que reciba los mismos parámetros. En la función original revisarás que se cumplan las precondiciones de los datos e inicializarás la variable que cuenta el número de llamadas recursivas. La función auxiliar es la que realmente realizará la recursión. Ya no revises aquí las precondiciones, pues ya sólo depende de ti garantizar que no la vas a llamar con parámetros inválidos. Incrementa aquí el valor del atributo contador, deberá incrementarse una vez por cada vez en que mandes llamar esta función.
   A continuación se ilustra la idea utilizando la función factorial: (OJO: tu código no es igual, sólo se ilustra el principio).

```
/// <summary>
/// Ejemplo de cómo contar el número de llamadas a la
/// implementación recursiva de la función factorial.
/// </summary>
namespace Análisis;

public class ComplejidadFactorial
{

    /// <summary>
    /// Número de operaciones realizadas en la última
    /// llamada a la función.
    /// </summary>
    private long _contador;

    /** Valor del contador de operaciones después de la
     * última llamada a un método. */
    public long Contador { get => _contador; }

    /** n! */
    public long Factorial(int n)
    {
        if (n < 0) throw new ArgumentOutOfRangeException();
        _contador = 0;
        return FactorialAux(n);
```

```
26      }
27
28      private long FactorialAux(int n)
29      {
30          _contador++;
31          if (n == 0) return 1;
32          else return n * FactorialAux(n - 1);
33      }
34
35      /** Imprime en pantalla el número de llamadas a la funció
            ↪ n
36      * para varios parámetros. */
37      static void Main(String[] args)
38      {
39          ComplejidadFactorial c = new ComplejidadFactorial();
40          for(int n = 0; n < 22; n++)
41          {
42              long f = c.Factorial(n);
43              Console.WriteLine
44              ($"Para␣{n}!={f}␣se␣realizaron␣{c.Contador}␣
                    ↪ operaciones");
45          }
46      }
47  }
```

5. Edita el método `Main` en el archivo `ComplejidadMain/Main.cs` en la clase `UsoComplejidad`. Este es otro proyecto que vive en la misma solución. Observa que la clase `UsoComplejidad` también se encuentra en el espacio Análisis, por lo tanto podrás mandar llamar los métodos anteriormente. Llámalos para diferentes valores de sus parámetros e imprime en pantalla el resultado y el número de operaciones que realizaron. Podrás ejecutarlo con el comando

```
dotnet run --project ComplejidadMain/ComplejidadMain.csproj
```

6. No olvides agregar los comentarios para generación de documentación automática en todos los métodos que programaste. Si estás utilizando `Visual Studio Code` estos comentarios aparecerán cuando la IDE te envíe mensajes referentes a tus métodos o atributos. Asegúrate de que todo tu código esté indentado correctamente.

Parte II

Gnuplot

Gnuplot es una herramienta interactiva que permite generar gráficas a partir de archivos de datos planos. Para esta práctica, los datos deben ser guardados en un archivo de este

tipo y graficados con gnuplot. Supongamos que el archivo donde se guardan es llamado **datos.dat**.

Por ejemplo, para el método de Fibonacci el archivo **datos.dat** tendría algo semejante al siguiente contenido:

Listado 1.1: data/FibonacciIt.dat

```
0       1
1       1
2       2
3       3
4       4
5       5
```

donde la primer columna es el valor del argumento y la segunda, el número de operaciones. Observa que la separación entre valores es un tabulador.

Para el método de Pascal el archivo **datos.dat** tendría algo semejante al siguiente contenido:

Listado 1.2: data/PascalRec.dat

```
0       0       2

1       0       2
1       1       2

2       0       2
2       1       4
2       2       2

3       0       2
3       1       6
3       2       6
3       3       2
```

donde la primer columna es el valor del renglón, la segunda es la columna y la tercera, el número de operaciones. Observa que cada vez que cambies de renglón, debes dejar una linea en blanco para indicarle a `gnuplot` cuándo cambia el valor en el eje x. No dejes espacios de más, porque `gnuplot` les asocia significado.

Gráficas en 2D

Al iniciar el programa `gnuplot` aparecerá un *promt* y se puede iniciar la sesión de trabajo. A continuación se muestra cómo crear una gráfica 2D. Deberás obtener algo como la Figura 1.2.

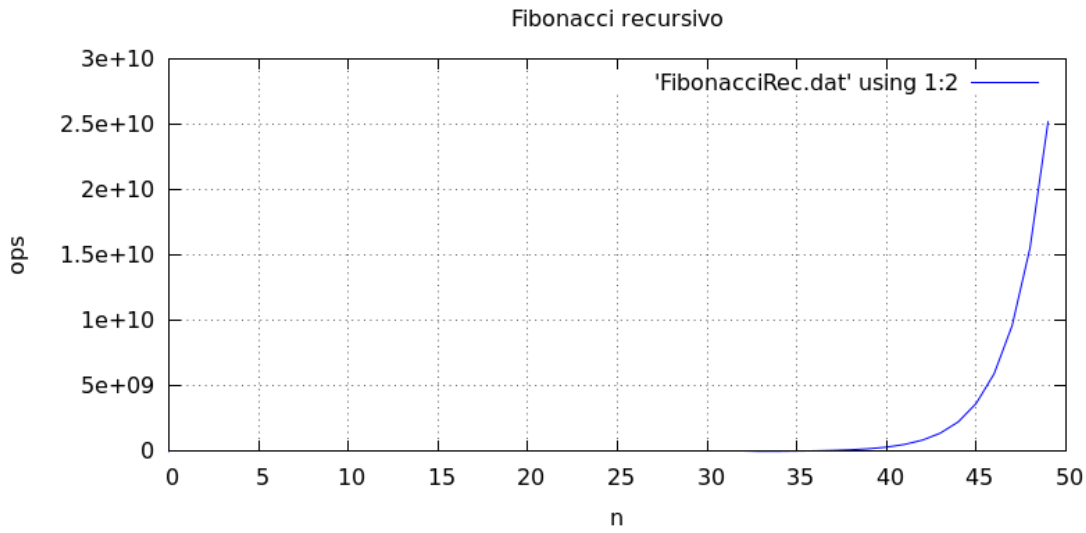

Figura 1.2 Complejidad en tiempo al calcular el n-ésimo coeficiente de la serie de Fibonacci en forma recursiva.

```
gnuplot> set title "Mi gráfica"      //Título para la gráfica
gnuplot> set xlabel "Eje X: n"       //Título para el eje X
gnuplot> set ylabel "Eje Y: ops"     //Título para el eje Y
gnuplot> set grid "front";           // Decoración
gnuplot> plot "datos.dat" using 1:2 with lines lc rgb 'blue' //
    ↪ graficamos los datos
gnuplot> set terminal pngcairo size 800,400  //algunas caracterí
    ↪ sticas de la imagen que se guardará
gnuplot> set output 'fib.png'        //nombre de la imagen que se
    ↪ guardará
gnuplot> replot                      //lo graficamos para que se
    ↪ guarde en la imagen
```

Gráficas en 3D

A continuación se muestra cómo crear una gráfica 3D. Observa que, en este caso, el archivo de datos requiere tres columnas. Deberás obtener algo como la Figura 1.3.

```
gnuplot> set title "Mi gráfica"
gnuplot> set xlabel "Eje X"
gnuplot> set ylabel "Eje Y"
gnuplot> set zlabel "Eje Z"
gnuplot> set pm3d
gnuplot> splot "datos.dat" using 1:2:3 with dots
gnuplot> set terminal pngcairo size 800,400
gnuplot> set output 'Pascal.png'
gnuplot> replot
```

1. Complejidad

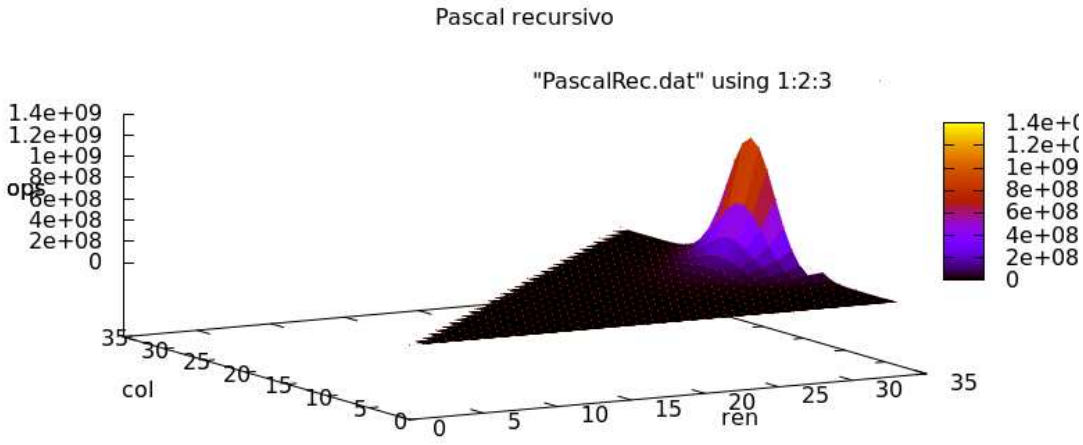

Figura 1.3 Complejidad en tiempo al calcular el coeficiente del triángulo de Pascal para el renglón y columna dados.

Ejercicios

1. Nota que la interfaz `IComplejidad` tiene algunos métodos estáticos. Estos se implementan en la interfaz, son auxiliares para escribir datos en un archivo. La cadena `archivo`, que reciben como parámetro, contiene la ubicación del archivo. Necesitarás crear un objeto de tipo `StreamWriter` para crear y acceder el archivo en el disco duro, observa que el método estático `EscribeLineaVacía` en la misma interfaz ya está programado; sigue la misma idea para escribir las líneas con los resultados.

   Esta técnica de escribir en un archivo los resultados línea por línea es común para la creación de bitácoras (*logs*).

2. Modifica tu método `Main` en la clase code `UsoComplejidad` para que, además de llamar los métodos programados para diferentes valores de sus parámetros, también guarde los resultados en archivos de texto utilizando los métodos que agregaste en `IComplejidad`. Podrás ejecutarlo con el comando

   ```
   dotnet run --project ComplejidadMain/ComplejidadMain.csproj
   ```

3. Para el método de Fibonacci, genera las gráficas n(entrada) vs número de operaciones y haz un análisis de lo que sucede. ¿Cuál es el orden de complejidad? Justifica.

4. Para el método recursivo del cálculo del triángulo de Pascal, genera una gráfica en 3-D en donde el parámetro renglón se encontrará en el eje X, el parámetro columna

se encontrará en el eje Y, el número de operaciones en el eje Z y haz un análisis de lo que sucede. ¿Cuál es el orden de complejidad? Justifica.

5. Entrega tu código limpio y bien documentado.

Preguntas

1. Para las versiones recursivas:
   - ¿Cuál es el máximo valor de n que pudiste calcular para Fibonacci sin que se alentara tu computadora? ¿cuánto se tardó? (Puede variar un poco de computadora a computadora (± 3), así que no te esfuerces en encontrar un valor específico).
   - ¿Cuál es el máximo valor de ren que pudiste calcular para el triángulo de Pascal sin que se alentara tu computadora? ¿cuánto se tardó?
2. Justifica a partir del código ¿cuál es el orden de complejidad para cada uno de los métodos que programaste?
3. Escribe un reporte en un archivo `.pdf` con tus gráficas generadas y las respuestas a las preguntas anteriores.