1 | Complejidad

Meta

Que el alumno visualice el concepto de "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.

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 $a_1 = 1$.

Actividad 1.1

Define, con estos datos, la función de fibonacci.

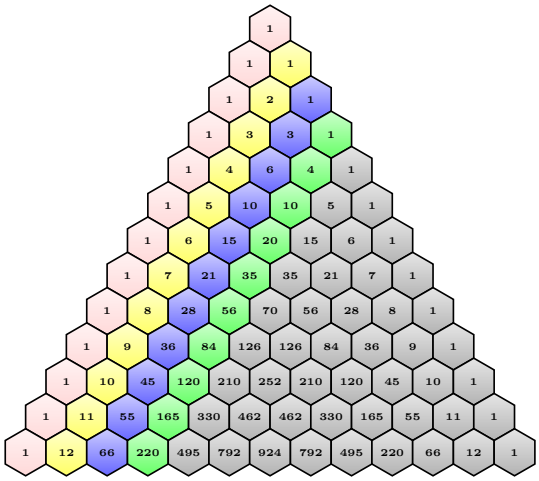

Figura 1.1 Triángulo de Pascal. Autor:M.H. Ahmadi

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:

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

Parte I

Desarrollo

La práctica consiste en implementar métodos que calculen el triángulo de pascal y el n-ésimo número de fibonacci, 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 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 y asegúrate de que compile, aunque aún no realize los cálculos.

2. Abre el archivo `ComplejidadTest.java`. Lee el código. Observa que cada método marcado con la anotación `@Test` se ejecuta como una prueba unitaria. La expresión `assertEquals` se utilizar 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 programaste para calcular Pascal y Fibonacci. 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.

   Es verdad, aún no tienes el código 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 `org.junit`.

3. Programa los métodos indicados en la interfaz. Las pruebas unitarias te ayudarán a verificar tus implementaciones de Fibonacci y Pascal. Compila tu código utilizando el comando `ant` en el directorio donde se encuentra el archivo `build.xml`, si compila correctamente las pruebas se ejecutarán automáticamente.

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).

```
1 /** Ejemplo de cómo contar el número de llamadas a la
2  * implementación recursiva de la función factorial. */
```

```java
public class ComplejidadFactorial {

  /* Número de operaciones realizadas en la última
   * llamada a la función. */
  private long contador;

  /** Valor del contador de operaciones después de la última
   *  llamada a un método. */
  public long leeContador() {
    return contador;
  }

  /** n! */
  public int factorial(int n) {
    contador = 1;
    if (n < 0) throw new IndexOutOfBoundsException();
    if (n == 0) return 1;
    return factorialAux(n);
  }

  private int factorialAux(int n) {
    operaciones++;
    if (n == 1) return 1;
    else return factorialAux(n - 1);
  }

  /** Imprime en pantalla el número de llamadas a la función
   * para varios parámetros. */
  public static void main(String[] args) {
    ComplejidadFactorial c = new ComplejidadFactorial();
    for(int n = 0; n < 50; n++) {
      int f = c.factorial(n);
      System.out.format
        ("Para n=%d se realizaron %d operaciones",
         n, c.leeContador());
    }
  }
}
```

5. Crea un método `main` en una clase code `UsoComplejidad` que mande llamar los métodos programados para diferentes valores de sus parámetros y que guarde los resultados en archivos de texto. Podrás ejecutarlo con el comando `ant run`.

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.

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.

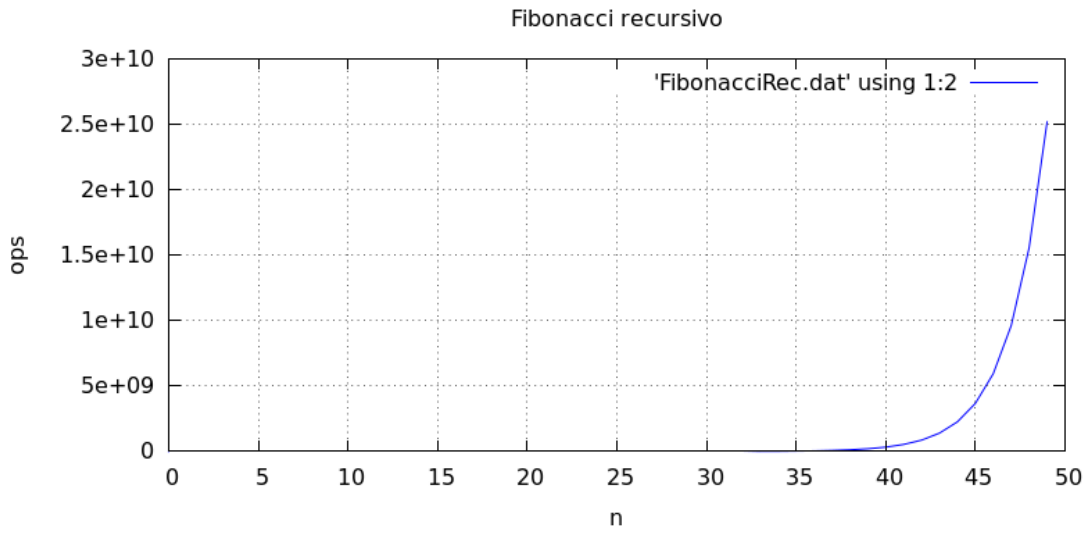

Figura 1.2 Complejidad en tiempo al calcular el n-ésimo coeficiente de la serie de Fibonacci en forma recursiva.

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.

```
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"
```

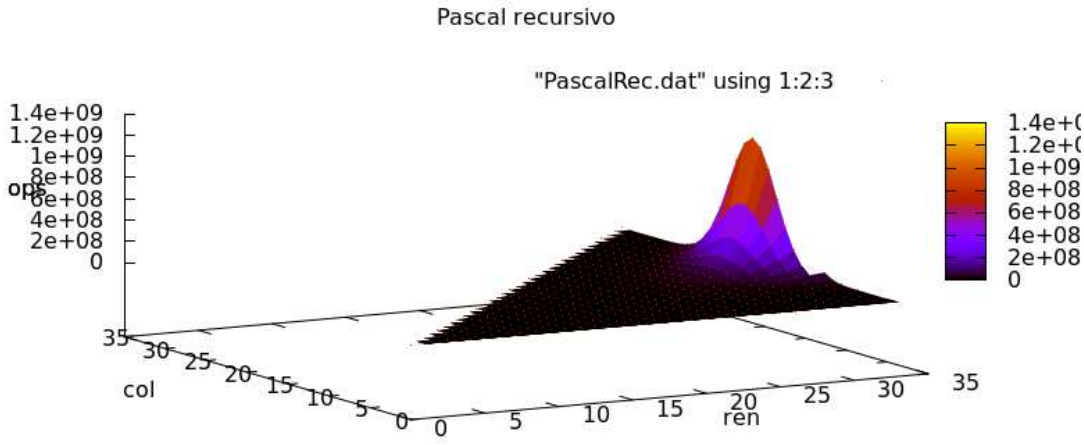

Figura 1.3 Complejidad en tiempo al calcular el coeficiente del triángulo de Pascal para el renglón y columna dados.

```
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
```

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 `PrintStream` para crear y acceder el archivo en el disco duro [1]. Cada vez que el método sea llamado, el archivo se debe abrir para agregar (modo *append*), de modo que los datos se acumulen entre llamadas sucesivas al método, al terminar debes cerrarlo. Revisa la documentación de `PrintStream` y `FileOutputStream`, te ayudarán mucho en esta parte.

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

[1] Si no estás seguro de cómo utilizar un `PrintStream` puedes consultar el tutorial https://docs.oracle.com/javase/tutorial/essential/io/charstreams.html.

guarde los resultados en archivos de texto utilizando los métodos que agregaste en `IComplejidad`. Podrás ejecutarlo con el comando `ant run`.

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 tus resultados en un reporte en un archivo `.pdf`, junto con tu código limpio y empaquetado.

Preguntas

1. Para las versiones recursivas:

   - ¿Cuál es el máximo valor de n que pudiste calcular para fibonacci sin que se alentara tu computadora? (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?

2. Justifica a partir del código ¿cuál es el orden de complejidad para cada uno de los métodos que programaste?

3. Escribe un reporte con tus gráficas generadas y las respuestas a las preguntas anteriores.

2 | Vector

Meta

Que el alumno domine el manejo de información almacenada arreglos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Transferir información entre arreglos cuando la capacidad de un arreglo ya no es adecuada.
- Diferenciar entre el tipo de dato abstracto *Vector* y su implementación.

Antecedentes

Un arreglo en la computadora se caracteriza por:

1. Almacenar información en una región contigua de memoria.
2. Tener un tamaño fijo.

Ambas características se derivan del sistema físico en el cual se almacena la información y sus limitaciones. Por el contrario, un *tipo de dato abstracto* es una entidad matemática y debe ser independiente de el medio en que se almacene. Para ilustrar mejor este concepto, se pide al alumno programar una clase `Vector` que obedezca a la definición del tipo de dato abstracto que se incluye a continuación, utilizando arreglos y aquellas técnicas requeridas para ajustar las diferencias de comportamiento entre ambas entidades.

Los métodos de manipulación que no devuelven ningún valor no pueden ser definidos estrictamente como funciones, por ello a menudo se refiere a ellos como *subrutinas*. Para resaltar este hecho se utiliza el símbolo $\xrightarrow{?}$ al indicar el valor de regreso.

Definición 2.1: Vector

Un *Vector* es una estructura de datos tal que:

1. Puede almacenar n elementos de tipo T.
2. A cada elemento almacenado le corresponde un *índice* i con $i \in [0, n-1]$. Denotaremos esto como $V[i] \to e$.
3. Para cada índice hay un único elemento asociado.
4. La capacidad máxima n puede ser incrementada o disminuida.

Nombre: $Vector$.
Valores: $\mathbb{N}$, T, con $null \in T$.
Operaciones: Sea inc una constante con $inc \in \mathbb{N}, inc > 0$ y this el vector sobre el cual se está operando.

Constructores :

Vector(): $\emptyset \to Vector$

Precondiciones: $\emptyset$

Postcondiciones :

- Un $Vector$ es creado con $n = inc$.
- A los índices $[0, n-1]$ se les asigna $null$.

Métodos de acceso :

lee(this, i) $\to$ e: $Vector \times \mathbb{N} \to T$

Precondiciones :

- $i \in \mathbb{N}, i \in [0, n-1]$

Postcondiciones :

- $e \in T$, e es el elemento almacenado en $Vector$ asociado al índice i.

leeCapacidad(this): $Vector \to \mathbb{N}$

Precondiciones: $\emptyset$

Postcondiciones: Devuelve n

Métodos de manipulación :

asigna(this, i, e): $Vector \times \mathbb{N} \times T \xrightarrow{?} \emptyset$

Precondiciones :

- $i \in \mathbb{N}, i \in [0, n-1]$
- $e \in T$

Postcondiciones :

- El elemento e queda almacenado en el vector, asociado al índice i. Nota: dado que el elemento asociado al índice es único, cualquier elemento que hubiera estado asociado a i deja de estarlo.

asignaCapacidad(this, n'): $Vector \times \mathbb{N} \xrightarrow{?} \emptyset$

Precondiciones: $n' \in \mathbb{N}, n' > 0$

Postcondiciones :

- A n se le asigna el valor n'.
- Si $n' < n$ los elementos almacenados en $[n', n-1]$ son eliminados.

- Si $n' > n$ a los índices $[n, n' - 1]$ se les asigna $null$.

aseguraCapacidad(this, n'): $Vector \times \mathbb{N} \xrightarrow{?} \emptyset$

Precondiciones: $n' \in \mathbb{N}, n' > 0$

Postcondiciones :

- Si $n' < n$ no pasa nada.
- Si $n' > n$: sea $nn = 2^i inc$ tal que $nn > n'$, a n se le asigna el valor de nn.
  El significado de esta fórmula surge de la heurística siguiente: habrá menos cambios de tamaño si, cada vez que se requiere más espacio, se duplica el tamaño actual. Esta fórmula surge entonces de duplicar virtualmente el tamaño del arreglo tantas veces como sea necesario hasta que el índice n' quepa en el arreglo.

Esta definición puede ser traducida a una implementación concreta en cualquier lenguaje de programación, en particular, a `Java`. Dado que `Java` es un lenguaje orientado a objetos, se busca que la definición del tipo abstracto de datos se vea reflejada en la interfaz pública de la clase que le corresponde, mientras que los detalles de implementación se vuelven privados. El esqueleto que se muestra a continuación corresponde a esta definición. Obsérvese cómo las precondiciones y postcondiciones pasan a formar parte de la documentación de la clase, mientras que el dominio y el rango de las funciones están especificados en las firmas de los métodos. Igualmente, el argumento `this` es pasado implícitamente por `Java`, por lo que no es necesario escribirlo entre los argumentos de la función; otros lenguajes de programación, como `Python`, sí lo solicitan.

Actividad 2.1

Revisa la documentación de la clase `Vector` de `Java`. ¿Cuáles serían los métodos equivalentes a los definidos aquí? ¿En qué difieren?

Compilando con ant

El código en este curso será editado en `Emacs` y será compilado con `ant`. El paquete para esta primera práctica incluye un archivo `ant` con las instrucciones necesarias.

Actividad 2.2

Abre una consola y cambia el directorio de trabajo al directorio que contiene a `src`. Intenta compilar el código utilizando el comando:

```
1   $ ant compile
```