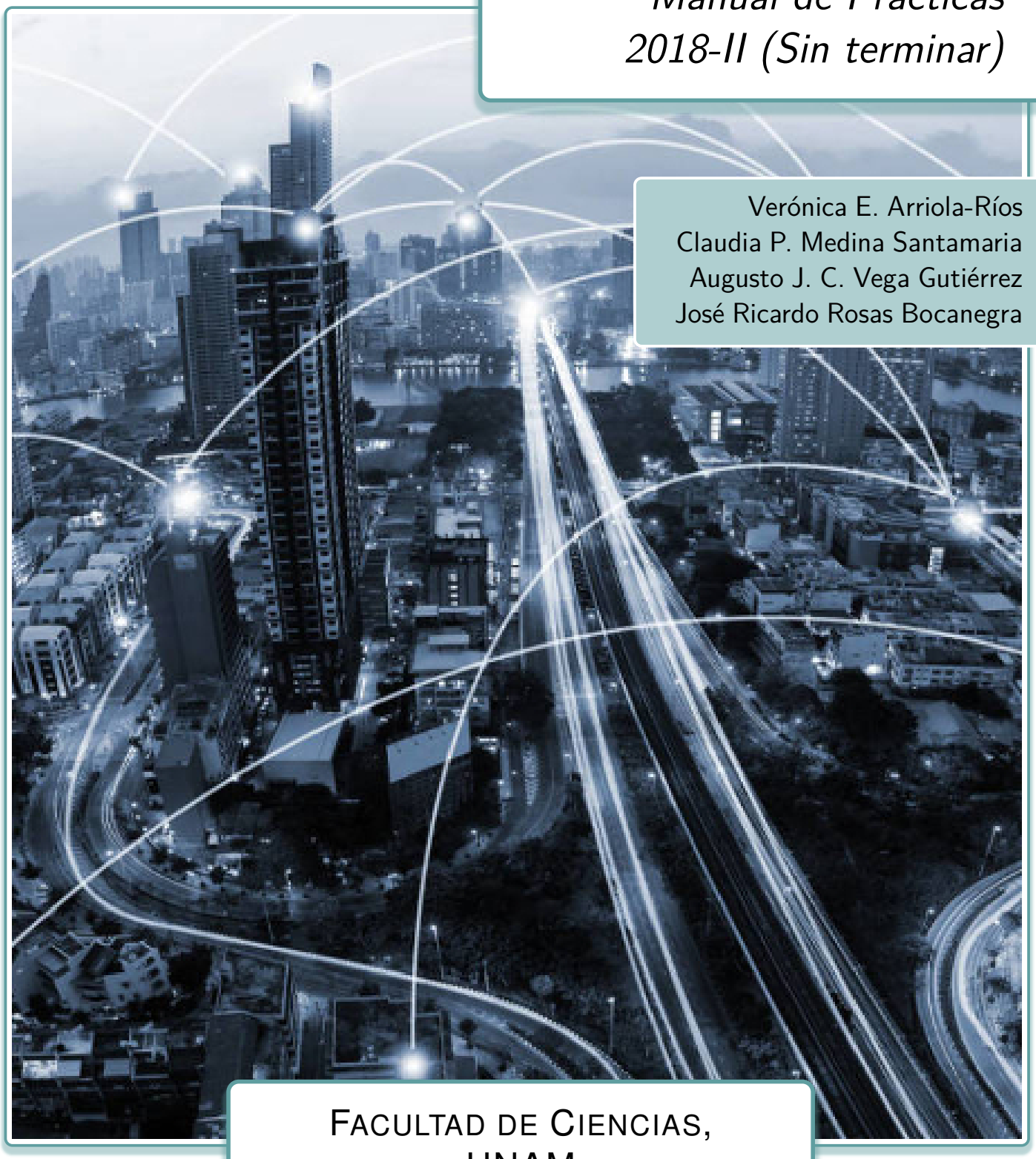
Estructuras de Datos

Manual de Prácticas
2018-II (Sin terminar)



Verónica E. Arriola-Ríos
Claudia P. Medina Santamaria
Augusto J. C. Vega Gutiérrez
José Ricardo Rosas Bocanegra

Facultad de Ciencias,
UNAM

Índice general

Parte I
Prácticas

1 | 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 1.1: 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.

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 1.1 *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 1.2 *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
```

Aparecerán varios errores pues el código no está completo.

Actividad 1.3 *Consigue que la clase compile. Agrega los enunciados `return` que hagan falta, aunque sólo devuelvan `null` ó `0`. Tu clase no ejecutará nada útil, pero será sintácticamente correcta. Por ejemplo, puedes hacer esto con el método `lee`:*

```
1   public T lee(int i) {
```

```
2       return null;
3   }
```

Al invocar `ant compile` ya no deberá haber errores y el directorio `build` habrá sido creado. Dentro de `build` se encuentran los archivos `.class`.

Actividad 1.4 *Intenta compilar el código utilizando el comando:*

```
1   $ ant
```

Esto intentará generar una distribución de tu código, pero para ello es necesario que pase todas las pruebas de `JUnit`, así que de momento te indicará que éstas fallaron. Para ejecutar únicamente las pruebas puedes llamar:

```
1   $ ant test
```

Esta tarea genera reportes en el directorio `reportes` donde puedes revisar los detalles sobre la ejecución de las pruebas, particularmente cuáles fallaron.

Para cuando termines esta práctica `ant` habrá creado el directorio `dist/lib`. Éste contendrá al archivo `Estructuras-<timestamp>.jar`. Si fueras a distribuir tu código, éste es el archivo que querrías entregar. Para los fines de este curso, más bien querremos el código fuente.

Actividad 1.5 *Cuando comentas tu código siguiendo el formato de `javadoc` es posible generar automáticamente la documentación de tus clases en formato `html`. Ejecuta la tarea:*

```
1   $ ant docs
```

Esto creará el directorio `docs`, con la documentación.

Actividad 1.6 *Para remover todos los archivos que fueron generados utiliza:*

```
1   $ ant clean
```

Asegúrate de ejecutar esta tarea antes de entregar tu práctica. Incluso remueve los archivos de respaldo de `emacs`. Ojo, no remueve los que llevan `#`. Puedes remover estos a mano o intenta modificar el archivo `build.xml` para que también los elimine, guíate por lo que ya está escrito.

DESARROLLO

Agrega el código necesario para que los métodos funcionen según indica la documentación. Cada vez que programes alguno asegúrate de que pase sus pruebas correspondientes de `JUnit`.

1. Obseva que los métodos `lee` y `asigna` deben ser programados para pasar cualquier prueba, pues son los métodos de acceso a la información, sin los cuales no es posible probar a los demás. Inicia con éstos.

   Para el método `lee` observa que tu clase promete entregar un objeto de tipo `T`, pero el arreglo interno (el atributo `buffer`) contiene `Object`. Desafortunadamente el sistema de tipos de `Java` no permite crear arreglos de un tipo genérico. Por ello será necesario utilizar un casting de la forma siguiente:

   ```
   1    T e = (T)(buffer[i]);
   ```

   Asegúrate de únicamente realizar este casting cuando estés seguro de que el objeto es de tipo `T`, de lo contrario `Java` te lo creerá, compilará correctamente, ejecutará el código y *cualquier cosa puede pasar*, desde excepciones tipo `ClassCastException` y errores extraños hasta que nunca se de cuenta.

2. En el método `asignaCapacidad` debes copiar los elementos del arreglo original cuando cambies el `buffer`. Por intereses académicos, es necesario que realices esta tarea con un ciclo, sin ayuda del API de `Java`. TIP: recuerda utilizar una *variable local* para referirte al arreglo recién creado, al final actualiza la *variable de clase*.

3. Para el método `aseguraCapacidad` calcula una fórmula que te permita cumplir con la condición indicada en el tamaño ($2^{\mathtt{i}}\mathtt{inc} > \mathtt{n}'$). Puedes utilizar las funciones `Math.log`, `Math.pow` y `Math.ceil` para realizar el cálculo, utiliza castings a `int` cuando sea necesario.

PREGUNTAS

1. ¿Cuál fue la fórmula que utilizaste para calcular $\mathtt{nn}$ en el caso en que es necesario redimensionar el arreglo?

2. Explica ¿cuál es el peor caso en tiempo de ejecución para la operación `aseguraCapacidad`?

3. ¿Qué problema se presenta si, después de haber incrementado el tamaño del arreglo en varias ocasiones, el usuario remueve la mayoría de los elementos del `Vector`, quedando un gran espacio vacío al final? ¿Cómo lo resolverías?

2 | 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.

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 2.1 *Define, con estos datos, la función de fibonacci.*

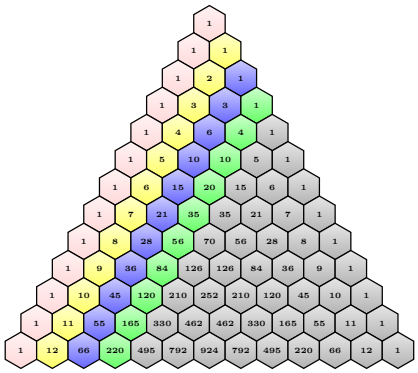


Figura 2.1 Triángulo de Pascal. Autor:M.H. Ahmadi

Triángulo de Pascal.

En la figura Figura 2.1 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:

$$Pascal_{ij} = \begin{cases} 1 & si\ j = 0\ ó\ j = i \\ Pascal_{(i-1)(j-1)} + Pascal_{(i-1)(j)} & \text{En cualquier otro caso.} \end{cases} \tag{2.1}$$

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 realizará 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.

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:

```
1 100
2 250
3 300
5 575
```

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:

```
1 1 1

2 1 1
2 2 1

3 1 1
3 2 2
3 3 1
```

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.

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 2.2.

```
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
```

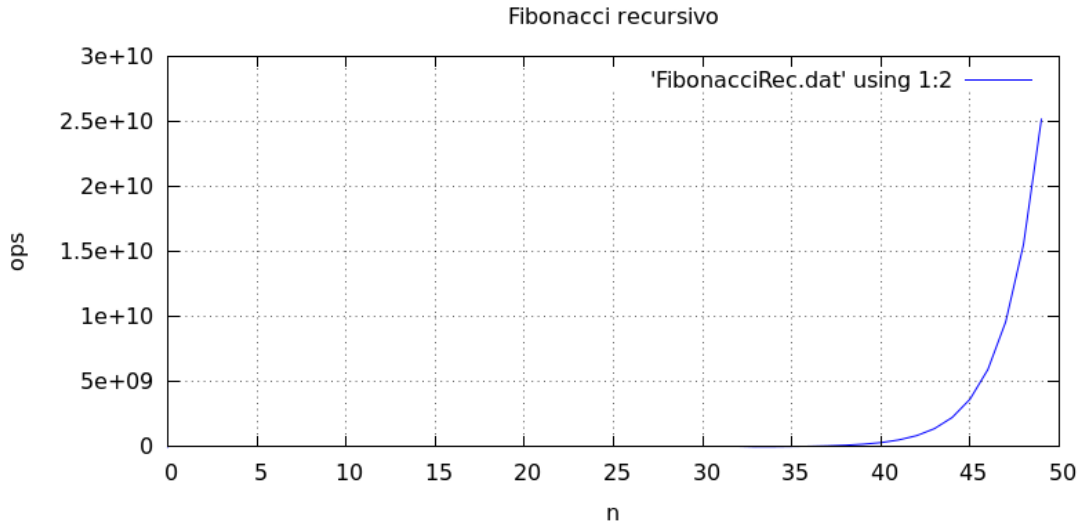


Figura 2.2 Complejidad en tiempo al calcular el n-ésimo coeficiente de la serie de Fibonacci en forma recursiva.

```
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 2.3.

```
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
```

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

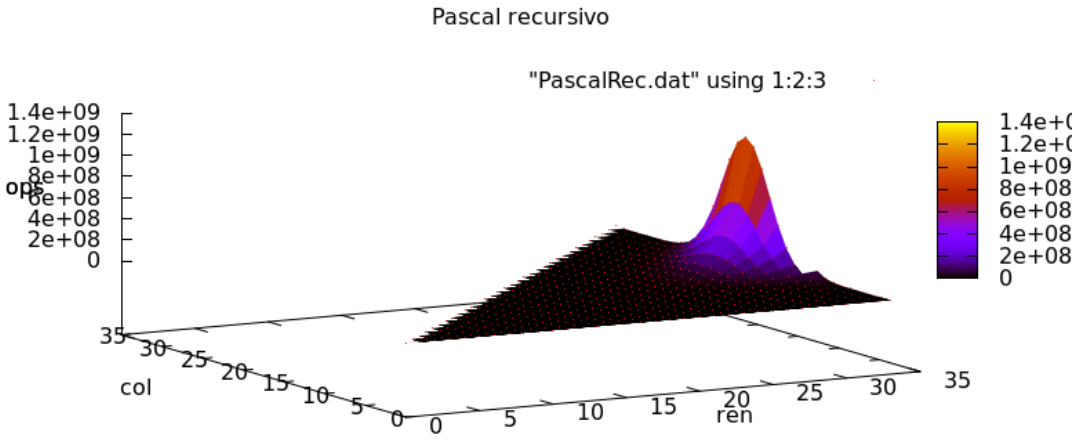


Figura 2.3 Complejidad en tiempo al calcular el coeficiente del triángulo de Pascal para el renglón y columna dados.

los cálculos.

2. Programa los métodos indicados en la interfaz. Las pruebas unitarias te ayudarán a verificar tus implementaciones de Fibonacci y Pascal. En particular, nota que los métodos estáticos se implementan en la interfaz, sólo son auxiliares para escribir los datos en el archivo. El archivo se debe abrir para agregar *append*, de modo que los datos se acumulen entre llamadas sucesivas al método, revisa la documentación de `PrintStream` y `FileOutputStream`, te ayudarán mucho en esta parte.

3. 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. Auquí utilizarás una técnica un poco más avanzada que sirve para optimizar varias cosas. Necesitarás crear una función auxiliar 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 puedes 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.
   Para la función factorial esto se vería como:

```
1 public class ComplejidadFactorial {
```

```
2     private long contador;
3     public int factorial(int n) {
4       contador = 1;
5       if (n < 0) throw new IndexOutOfBoundsException();
6       if (n == 0) return 1;
7       return factorialAux(n);
8     }
9     private int factorialAux(int n) {
10      operaciones++;
11      if (n == 1) return 1;
12      else return factorialAux(n - 1);
13    }
14  }
```

4. Crea un método `main` que mande llamar los métodos programados para diferentes valores de sus parámetros y que guarde los resultados en archivos de texto.

5. 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.

6. 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.

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

3 | Polinomio de direccionamiento

Meta

Que el alumno domine el manejo de información almacenada en arreglos multidimensionales.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Almacenar un arreglo de n dimensiones en uno de una sola dimensión.

Antecedentes

Vectores de Iliffe

Los arreglos multidimensionales son aquellos que tienen más de una dimensión, los más comunes son de dos dimensiones, conocidos como matrices. Este tipo de arreglos en Java se ven como arreglos de arreglos, a los cuales se llama *vectores de Iliffe*.

Existen dos momentos fundamentales en la creación de arreglos:

1. Declaración: en esta parte no se reserva memoria, solo se crea una referencia.

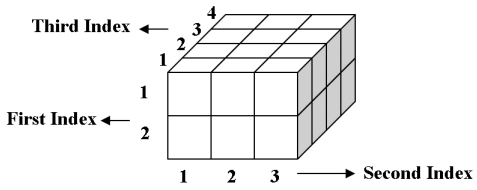
```
1 int [][] arreglo;
2 float[][][] b;
```

2. Reservación de la memoria y la especificación del número de filas y columnas.

```
1 //matriz con 10 filas y 5 columnas
```

```
arreglo = new int[10][5];

//cubo con 13 filas, 25 columnas y 4 planos
b = new float [13][25][4];
```



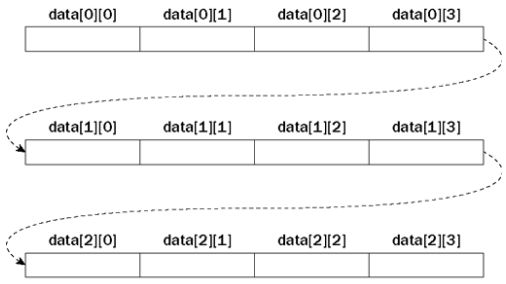
Three-dimensional array with twenty four elements

Nota: Se puede declarar una dimensión primero, pero siempre debe de ser en orden, por ejemplo `arreglo = new int[][10];` es erróneo pues las filas quedan indeterminadas. Esta libertad permite crear arreglos de forma irregular, como:

```
int[][][] arreglo = new int[3][][];
arreglo[0] = new int[2][];
arreglo[0][1] = {1,2,3};
arreglo[2] = new int[1][2];
// Se ve como:
// {{{1,2,3}, null}, null, {0,0}}
```

Polinomio de direccionamiento

Dado que la memoria de la computadora es esencialmente lineal es natural pensar en almacenar los elementos de un arreglo multidmensional en un arreglo de unidimensional. Por ejemplo, consideren un arreglo de tres dimensiones:



The array elements are stored in contiguous locations in memory.

Generalicemos el ejemplo anterior a uno de n-dimensiones, donde el tamaño de cada dimensión i esta dada por δ_i, entonces tenemos un arreglo n-D: $\delta_0 \times \delta_1 \times \delta_2 \times ... \times \delta_n$.

Entonces la posición del elemento $a[i_0][i_1][...][i_n]$ esta dada por:

$$p(i_0, i_1, ..., i_n) = \sum_{i=1}^{n} f_j i_j \tag{3.1}$$

donde

$$f_j = \begin{cases} 1 & \text{si } j{=}n \\ \prod_{k=j+1}^{n} \delta_k & \text{si } 1 \leqslant j < n \end{cases} \tag{3.2}$$

DESARROLLO

La práctica consiste en implementar los métodos definidos en la interfaz `IArreglo`, la cual convierte un arreglo de enteros de n-dimensiones en uno de una dimensión. El constructor de la clase que implemente la interfaz deberá tener como parámetro un arreglo de $\mathtt{int}$s que representen las dimensiones del arreglo. Por ejemplo para crear un arreglo tridimensional ($3 \times 10 \times 5$), invocamos el constructor de la siguiente forma:

```
1  Arreglo a = new Arreglo(new int [] {3,10,5});
```

Observa que todas las dimensiones deben ser mayores que cero.

PREGUNTAS

1. Explica la estructura de tu código, explica en más detalle tu implementación del método `obtenerIndice`.
2. ¿Cuál es el orden de complejidad de cada método?

4 | Colección abstracta

Meta

Que el alumno aplique la reutilización de código mediante el mecanismo de herencia e interfaces propuesto por el paradigma orientado a objetos en `Java`.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Escribir código general en una clase padre, sin conocer detalles sobre la implementación de las clases descendientes.
- Utilizar la definición, mediante una interfaz, de un tipo de dato abstracto, para programar funciones generales, sin conocer detalles sobre la implementación de los objetos que utiliza.

Antecedentes

Para reducir la cantidad de trabajo en las prácticas siguientes, se utilizarán las ventajas del paradigma orientado a objetos. Concretamente, se programará una biblioteca con varias estructuras de datos y las operaciones comunes a todas ellas serán implementadas en una clase padre. En esta práctica se trata de completar tantos métodos como sea posible programar eficientemente, aún sin haber programado ninguna de esas estructuras.

Para adquirir algo de habilidad creando código profesional, este paquete trabajará cumpliendo un conjunto de especificaciones dictadas por la API[1] de `Java`.

Actividad 4.1 *Para familiarizarte con el ambiente de trabajo, revisa la documentación*

[1] Interfaz de programación de aplicaciones.

de las clases `Collection<E>` e `Iterator<E>` de `Java`.

Todas las estructuras que programaremos implementarán `Collection<E>`. No te preocupes, no duplicaremos la labor de las estructuras que ya vienen programadas en `Java`. La mayoría de nuestras estructuras ofrecerán características distintas a las versiones de la distribución oficial. Más aún, por motivos didácticos y ligeramente nacionalistas, nuestras clases tendrán nombres en español[2].

Actividad 4.2 *Revisa la documentación del paquete `java.util`. ¿Qué estructuras de datos encuentras incluidas?*

La primer clase a programar de nuestro paquete se llama `ColeccionAbstracta<E>` e implementa la intefaz `Collection<E>`. Todas nuestras estructuras heredarán de ella, por lo que el trabajo de esta práctica nos ahorrará mucho código en las venideras. Como `ColeccionAbstracta<E>` no sabe aún cómo serán guardados los datos, no podrá implementar todos los métodos de `Collection<E>`; de ahí que será de tipo `abstracto`. Para poder trabajar, hará uso del único conocimiento que tiene de estructuras tipo `Collection<E>`: que todas ellas implementan el método `iterator()`, que devuelve un método de tipo `Iterator<E>`.

Dado que el iterador recorre la estructura (sea cual sea ésta), otorgando acceso a cada uno de sus elementos una única vez, es posible implementar varios de los métodos de `Collection<E>` haciendo uso de este objeto.

DESARROLLO

1. Crea una clase llamada `ColeccionAbstracta<E>` que implemente la interfaz `Collection<E>`, dentro del paquete `estructuras`.
2. Implementa únicamente los métodos listados a continuación. Una sugerencia es que añadas todas las firmas de los métodos y hagas que devuelvan `0` o `null` para verificar que tu clase compile. Observa que la clase `Conjunto<E>` fue provista como ejemplo de clase hija. Una vez que agregues los métodos, `Conjunto<E>` deberá compilar sin problemas, esto será necesario para que las pruebas unitarias funcionen.
   - `public boolean contains(Object o)`
   - `public Object[] toArray()`
   - `public <T> T[] toArray(T[] a)`
   - `public boolean containsAll(Collection<?> c)`
   - `public boolean addAll(Collection<? extends E> c)`

[2]Aunque los métodos seguirán teniendo nombres en inglés, pues así lo requieren las interfaces.

- `public boolean remove(Object o)`
- `public boolean removeAll(Collection<?> c)`
- `public boolean retainAll(Collection<?> c)`
- `public void clear()`

PREGUNTAS

1. ¿Qué estructuras de datos incluye la API de `Java` dentro del paquete que importas, `java.util`?

2. ¿Cuál crees que es el objetivo de la interfaz `Collection`? ¿Por qué no hacer que cada estructura defina sus propios métodos?

3. ¿Qué métodos permite la interfaz `Collection` que su funcionalidad sea opcional? ¿Qué deben hacer estos métodos opcionales si no se implementa su funcionalidad? ¿Por qué crees que son opcionales?