

Programación II - 2015

Docentes:

- Lic. Martin Pustilnik
- Ing. Matias Roman Velazquez Hess

EMail: ungsmpprogramacion2@hotmail.com.ar

Enviar el mail que se utilizara para la materia a esa dirección.

Bibliografía de referencia:

- Estructura de datos en Java: Mark Allen Weiss
- Piensa en java: Bruce Eckel
- Programación orientada a objetos: David J. Barnes
- Se deja copia en la red

Modalidad: Tres instancias de evaluación:

- TP1 (de una o dos personas)
- Parcial
- TP2 (de una o dos personas)
- Y entrega de ejercicios con *

Aprobación:

Todas las notas ≥ 4

parcial ≥ 5

Programación I con final aprobado (o promocionada).

Todas las instancias tienen recuperatorio.

Promoción:

Aprobación y promedio ≥ 7

Consultas:

Jueves 18hs, oficina 2018, ICI, Modulo 2.

Material obligatorio:

Un Pendrive por alumno

Material on-line:

Bibliografía

<http://1drv.ms/1rmBqFF>

Programa de materia

<http://1drv.ms/1A8Umua>

Clases y Prácticas

<http://1drv.ms/1A8Uzxi>

Durante el transcurso del cuatrimestre se enviara el material adicional de la cursada.



Acumuladores booleanos

Un acumulador booleano utiliza las propiedades de la lógica para acumular un valor de verdad (verdadero o falso) durante la vida de un algoritmo.

En general estos algoritmos son una implementación de una función booleana.

Sean a y b dos variables booleanas. Las propiedades lógicas para **and(&&)** y **or(||)** son:

	a	b	$A \mid\mid b$	$a \&\& b$
(1)	f	f	f	f
(2)	f	v	v	f
(3)	v	f	v	f
(4)	v	v	v	v

Son muy útiles cuando se quiere chequear una propiedad booleana en una estructura, por ejemplo, en una lista de valores.

Ejemplo1a:

Queremos ver si toda una lista de números es par:

```
boolean esPar(lista)
for (int i=0, i++, i < lista.size())
    if !(par(lista[i]))
        return false
return true
```

esPar Version1a sin acumuladores

```
boolean esPar(lista)
boolean ret = true
for (int i=0, i++, i < lista.size())
    ret = ret && par(lista[i])
return ret
```

esPar Version2a con acumuladores



Veamos que sucede con la lista [2,3,10] para el ejemplo1a:

i	Sin acumuladores (valor de retorno)	Con acumuladores (valor de ret)	Obs
0	v	v	Ret = v && par(2) Ret = v && v
1	f	f	Ret = v && par(3) Ret = v && f
2	(no llega a este punto)	f	Ret = v && par(10) Ret = f && v

Por ahora, las dos implementaciones son muy parecidas.

Ejemplo1b:

Ahora también queremos devolver la lista con todos sus elementos multiplicados por 2.

```
boolean esPar(lista)
for (int i=0, i++, i < lista.size())
    if !(par(lista[i]))
        return false
    lista[i] = lista[i] * 2
return true
```

esPar Version1b sin acumuladores

Pero esto no siempre funciona. Si no llegamos hasta el final de la lista, no multiplicaremos el resto de la lista por 2.

Veamos como lo podemos arreglar.

```
boolean esPar(lista)
for (int i=0, i++, i < lista.size())
    if !(par(lista[i]))
        return false
for (int i=0, i++, i < lista.size())
    lista[i] = lista[i] * 2
return true
```

esPar Version1c sin acumuladores

Pero esto no siempre funciona. Al multiplicar por 2, todas las listas quedan pares! Pero no es cierto que la lista original sea par. Ej [1,3,5] es impar, mientras que [2,6,10] es par!



```
boolean esPar(lista)
copiaLista = lista.clonar()
for (int i=0, i++, i < lista.size())
    lista[i] = lista[i] * 2
for (int i=0, i++, i < copiaLista.size())
    if !(par(copiaLista[i]))
        return false
return true
```

esPar Version1d sin acumuladores

Finalmente funciono, pero a que costo?

```
boolean esPar(lista)
boolean ret = true
for (int i=0, i++, i < lista.size())
    ret = ret && par(lista[i])
    lista[i] = lista[i] * 2
return ret
```

esPar Version2b con acumuladores

Comparación de codificar con acumuladores, respecto de hacerlo sin:

Sin acumuladores	Con acumuladores
Código mas largo	
	Mas declarativo
Promueve la introducción de errores	
Difícil de modificar/agregar código.	
El código termina antes en algunos casos. Pero demostraremos que ese “ahorro” no tiene un peso significativo en la mayoría de los casos.	Se recorre siempre hasta el final.

**Ejemplo2:**

Queremos ver si algún número de la lista es par:

En este caso utilizaremos el or para acumular.

```
boolean algunPar(lista)
for (int i=0, i++, i < lista.size())
    if (par(lista[i]))
        return true
return false
```

algunPar Version1 sin acumuladores

```
boolean algunPar(lista)
boolean ret = false
for (int i=0, i++, i < lista.size())
    ret = ret || par(lista[i])
return ret
```

algunPar Version2 con acumuladores

Regla general para acumulación booleana

cuando la hipótesis es:

que la propiedad se cumple, ret comienza valiendo true y la acumulación es con “and”.

cuando la hipótesis es:

que la propiedad no se cumple, ret comienza valiendo false y la acumulación es con “or”.

Otros atajos

En lugar de preguntar por

```
If (variable == true)
    Return true
}else{
    Return false
```

Podemos simplemente devolver

```
return variable
```

**Ejercicio:**

Realizar una función que dada una lista de números devuelva verdadero si:

La lista tiene todos los números mayores que 8

La lista tiene algún número menor que 23

Utilizando acumuladores boléanos

```
public class Ppal {  
  
    public static void main(String[] args) {  
        Integer [] lista1 = {9,10,11};           //cumple  
        Integer [] lista2 = {90,100,110};         //no cumple  
  
        System.out.println(todosMayor8(lista1) && algunoMenor23(lista1));//true  
        System.out.println(todosMayor8(lista2) && algunoMenor23(lista2));//false  
    }  
  
    static public boolean todosMayor8(Integer [] lista){  
        boolean ret = true;  
        for (int i = 0; i < lista.length; i++){  
            ret = ret && lista[i] > 8;  
        }  
        return ret;  
    }  
  
    static public boolean algunoMenor23(Integer [] lista){  
        boolean ret = false;  
        for (int i = 0; i < lista.length; i++){  
            ret = ret || lista[i] < 23;  
        }  
        return ret;  
    }  
}
```

Nota:

Cuando una propiedad se quiere probar sobre toda la lista, se utiliza “and”

Cuando propiedad se quieren demostrar sobre algún elemento de la lista, se utiliza “or”

Programación II

Práctica 00: Acumuladores booleanos

Versión del 15/03/2014

Como vimos en clase un **acumulador** booleano toma en realidad dos valores.

Se denomina acumulador, porque suma un resultado parcial al resultado final de la función.

*1) Implementar con acumuladores una función booleana que recibe una lista de números, que sea verdadera si todos los numeros son mayores a 10.

```
boolean mayor10(int[] lista){  
}
```

Cuantificadores

Cuando queremos probar una propiedad P para todo un conjunto de datos:

$$\{\forall x \in \text{lista} / P(x)\} \equiv \text{true}$$

Utilizaremos la hipótesis de `ret = true` y la acumulación será de la forma:

$$\text{ret} = \text{ret} \text{ and } P(x)$$

Cuando queremos probar una propiedad P para un solo elemento:

$$\{\exists x \in \text{lista} / P(x)\} \equiv \text{true}$$

Utilizaremos la hipótesis de `ret = false` y la acumulación será de la forma:

$$\text{ret} = \text{ret} \text{ or } P(x)$$

*2) Implementar la función `mayor10_par`, que recibe una matriz, y es verdadera cuando:

Para todas las filas F_i , $\exists x$ en F_i tal que $x > 10$

Para todas las columnas C_j , $\exists y$ en C_j tal que y es par.

```
boolean mayor10_par(int[][] matriz){  
}
```

3) Implementar la función `mayorX_Y`, que recibe una matriz, y es verdadera cuando:

todas las filas X_i son mayores que alguna columna Y_j

Una fila es mayor que una columna cuando la suma de sus números es mayor.

Por ejemplo, sea la matriz m:

	Colu 0	Colu 1	Colu 2
Fila 0	10	20	30
Fila 1	10	20	30
Fila 2	10	20	30

la fila 0, que suma 60, no es mayor que la columna2, que suma 90.

Pero la fila 0, que suma 60, es mayor que la columna0, que suma 30.

Como ocurre lo mismo para las filas 1 y 2, `mayorX_Y` es verdadera para m.

Complejidad computacional y asintótica

Complejidad computacional

Indica el esfuerzo que hay que realizar para aplicar un algoritmo y lo costoso que éste resulta.

La eficiencia suele medirse en términos de consumo de recursos:

Espaciales: cantidad de memoria que un algoritmo consume o utiliza durante su ejecución → Complejidad espacial

Temporales: tiempo que necesita el algoritmo para ejecutarse → Complejidad temporal

Nos centraremos en complejidad temporal por ser el recurso más crítico

Complejidad Asintótica

Consiste en el cálculo de la complejidad temporal de un algoritmo en función del tamaño del problema, n , prescindiendo de factores constantes multiplicativos y suponiendo valores de n muy grandes.

No sirve para establecer el tiempo exacto de ejecución, sino que permite especificar una cota (inferior, superior o ambas) para el tiempo de ejecución de un algoritmo

Gráfico de complejidad asintótica (cuando n tiende a infinito), para las funciones más conocidas:

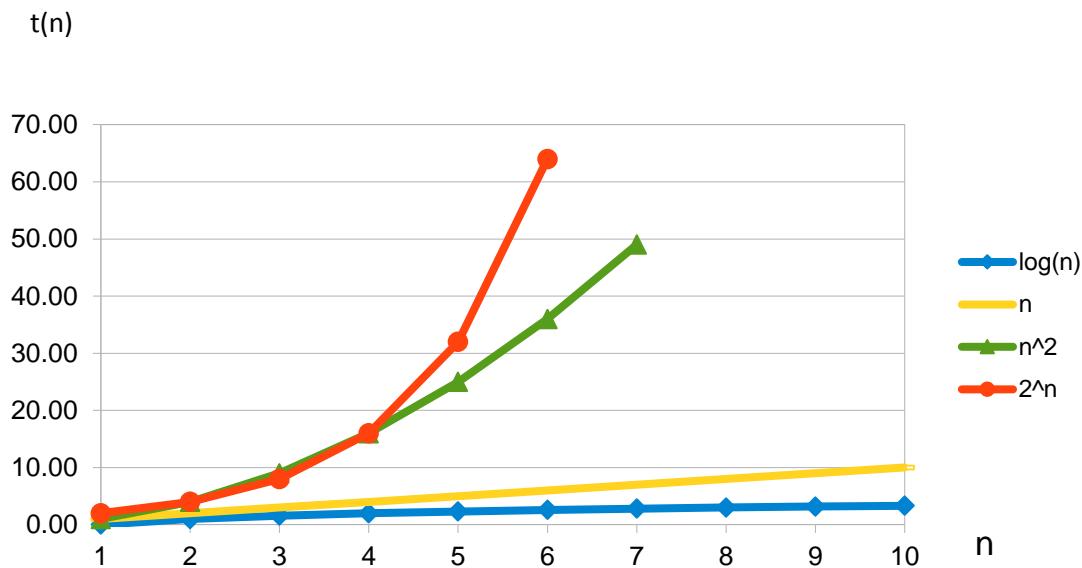
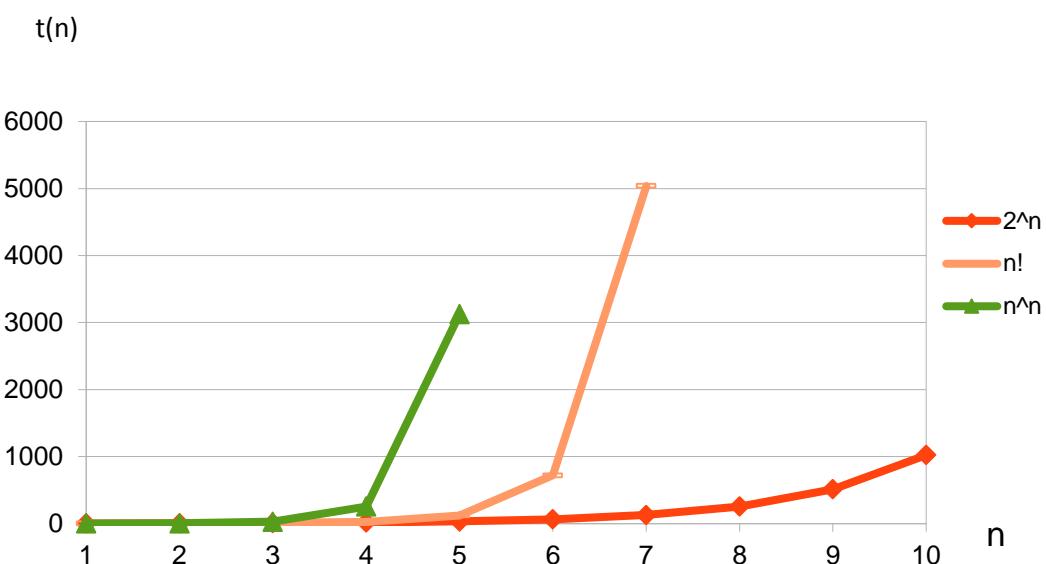


Grafico1: Familias de funciones





Se puede utilizar la complejidad computacional para comparar dos implementaciones del mismo algoritmo y quedarnos con la implementación que tenga mejor complejidad asintótica.

Ejemplo de complejidad asintótica

Ejemplo en pseudocódigo:

Buscar el máximo valor de un array

maximoArray(A)	operaciones
INICIO	2
maxActual ← A[0]	2 + n
PARA i ← 1 HASTA n – 1 HACER	2(n – 1)
SI A[i] > maxActual	2(n – 1)
ENTONCES maxActual ← A[i]	2(n – 1)
{ incrementar contador i }	2(n – 1)
DEVOLVER maxActual	1
FIN	Total 7n – 1

El orden asintótico de maximoArray es:

$$t(n) = 7n - 1$$

Las cotas

Una cota se define como un número M tal que cualquier elemento del conjunto es menor o igual que M.

Ejemplo1

Para el conjunto {2,5,6} podemos dar varias cotas:

$$M_1 = 6$$

$$M_2 = 7$$

Notar que todo el conjunto es menor o igual a 6 pero también es menor o igual a 7.

En el caso de una función, $f(n)$, una cota está dada por otra función a la que llamaremos $g(n)$, tal que para todo n :

$$g(n) \geq f(n)$$

Ejemplo2:

Como se ve en el Gráfico1, podemos acotar $f(n) = \log(n)$ por $g(n) = n$, porque para todo $n \in \text{Nat}$:

Cota1: $n > \log(n)$

Pero también podríamos acotar $\log(n)$

Cota2: $n^2 > \log(n)$

Siempre que sea posible, vamos a dar la cota más ajustada, en este caso, Cota1.

La notación O

Se puede dar una definición formal de la complejidad:

$$f \in O(g) \Leftrightarrow \exists n_0, c \text{ tales que para todo } n > n_0 \\ f(n) < c g(n)$$

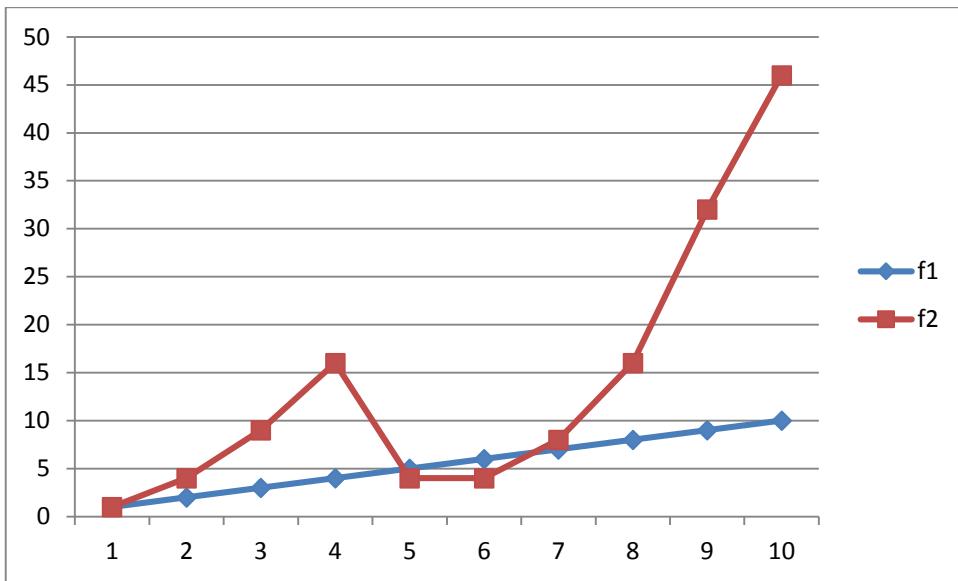
Donde n , es el tamaño de la instancia que es parámetro del algoritmo.
y n_0 significa, "el primer valor de n " para el cual se cumple.

Ejemplo1:

Queremos probar que $f_1 \in O(f_2)$

Es decir, que cuando n tiende a infinito $f_2 > f_1$

Es decir, queremos **acotar** f_1 con f_2 .



1) Buscamos n_0

El primer candidato es 2, ya que $f_2(2) > f_1(2)$. Sin embargo $f_2(6) < f_1(6)$.

Entonces no es cierto que "para todo $n > 2$ $f_2(n) > f_1(n)$ ".

Si $n_0 = 7$, si podemos afirmar que "para todo $n > 7$ $f_2(n) > f_1(n)$ ".

Siempre que el f_1 y f_2 crezcan a la velocidad que sugiere el grafico.

2) En este caso podemos usar $c = 1$

Ejemplo2:

Ahora vamos a acotar `máximoArray` utilizando la definición de O

Sea $f(n) = 7n-1$

Esto lo sabemos porque contamos las operaciones a mano.

Sea $g(n) = n$

$g(n)$ en este punto es una conjetura.

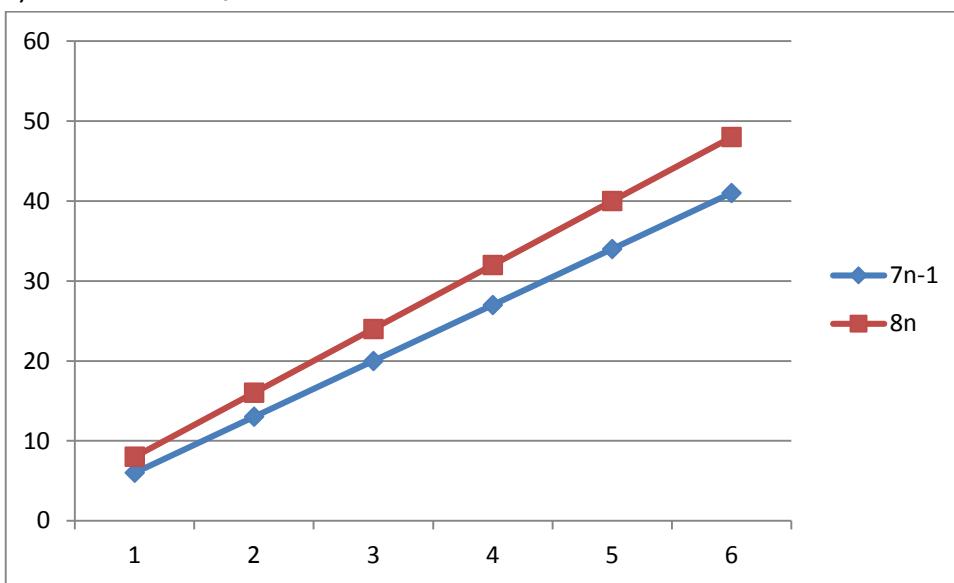
Si queremos demostrar que $7n-1 \in O(n)$ tenemos que encontrar un c y un n_0 tal que:

$$7n-1 < cn \quad // f(n) < cg(n)$$

para todo $n > n_0$

1) Claramente $c > 6$, porque de lo contrario, no es cierto que $7n-1 < 6n$

2) Buscamos el n_0



El candidato es $n_0 = 1$.

Como f y g son lineales, sabemos que crecen a la misma velocidad y que por lo tanto no van a volver a cruzarse.

Por lo tanto para todo $n > 1$ $8n > 7n-1$

Con (1) y (2), demostramos que $7n-1 \in O(n)$

Donde $8n$ es una cota de $7n-1$

Existen diferentes notaciones para la complejidad asintótica

Una de ellas es la notación O , que permite especificar la cota superior de la ejecución de un algoritmo.

La sentencia “ $f(n)$ es $O(g(n))$ ” significa que la tasa de crecimiento de $f(n)$ no es mayor que la tasa de crecimiento de $g(n)$

La notación “ O ” sirve para clasificar las funciones de acuerdo con su tasa de crecimiento

Proporciona una cota superior para la tasa de crecimiento de una función

	$f(n)$ es $O(g(n))$	$g(n)$ es $O(f(n))$
$g(n)$ crece más	Sí	No
$f(n)$ crece más	No	Sí
Igual crecimiento	Sí	Sí

Propiedades de $O(f(n))$:

Reflexiva: $f(n) \in O(f(n))$

Transitiva: si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$, entonces $f(n) \in O(h(n))$

Eliminación de constantes: $O(c \cdot f(n)) = O(f(n))$, para todo c .
 $O(\log_a n) = O(\log_b n)$, para todo a y b .

Suma de órdenes: $O(f(n)+g(n)) = O(\max(f(n), g(n)))$

Producto de órdenes: $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

Álgebra de Ordenes

Reglas y Álgebra de Ordenes

$$1) \quad O(1) \leq O(\log n) \leq O(n) \leq O(n^k) \leq O(k^n) \leq O(n!) \leq O(n^n)$$

$$2) \quad O(f) + O(g) = O(f + g) = O(\max\{f, g\})$$

$$3) \quad O(f) \cdot O(g) = O(fg)$$

$$4) \quad O(k) = O(1) \text{ para todo } k \text{ constante.}$$

Ejemplo: $O(2n)$

$$\begin{aligned} &= O(2) * O(n) \quad // \text{ por regla 3} \\ &= O(1) * O(n) \quad // \text{ por regla 4} \\ &= O(n) \end{aligned}$$

$$5) \quad \sum_{i=1}^k O(f) = O(\sum_{i=1}^k f) = O(kf)$$

Si k es constante, entonces vale $O(f)$

$$6) \quad \sum_{i=1}^k i = \frac{k*(k+1)}{2} = O(k^2)$$

El análisis asintótico de algoritmos determina el tiempo de ejecución en notación “O”

Para realizar el análisis asintótico

- Buscar el número de operaciones primitivas ejecutadas en el peor de los casos como una función del tamaño de la entrada
- Expresar esta función con la notación “O”

Ejemplo:

- Se sabe que el algoritmo *maximoArray* ejecuta como mucho $7n - 1$ operaciones primitivas
- Se dice que *maximoArray* “ejecuta en un tiempo $O(n)$ ”

Como se prescinde de los factores constantes y de los términos de orden menor, se puede hacer caso omiso de ellos al contar las operaciones primitivas

Análisis del caso mejor, peor y medio

El tiempo de ejecución de un algoritmo puede variar con los datos de entrada.

Ejemplo (ordenación por inserción):

```

INICIO
i ← 1
MIENTRAS (i < n)
    x ← T[i]
    j ← i - 1
    MIENTRAS (j > 0 Y T[j] > x)
        T[j+1] ← T[j]
        j ← j - 1
    FIN-MIENTRAS
    T[j+1] ← x
    i ← i + 1
FIN-MIENTRAS
FIN

```

Si el vector está completamente ordenado (mejor caso) → el segundo bucle no realiza ninguna iteración → Complejidad $O(n)$.

Si el vector está ordenado de forma decreciente, el peor caso se produce cuando x es menor que $T[j]$ para todo j entre 1 e $i - 1$.

En esta situación la salida del bucle se produce cuando $j = 0$.

En este caso, el algoritmo realiza $i - 1$ comparaciones.

Esto será cierto para todo valor de i entre 1 y $n - 1$ si el orden es decreciente.

El número total de comparaciones será:

$$\sum_{i=1}^{n-1} (i - 1) = \frac{n \cdot (n - 1)}{2} \in O(n^2)$$

La complejidad en el caso promedio estará entre $O(n)$ y $O(n^2)$.

Para hallarla habría que calcular matemáticamente la complejidad de todas las permutaciones de todos valores que se pueden almacenar en el vector.

Se puede demostrar que es $O(n^2)$.

Ejercicio1

Demostrar que $\sum_{i=0}^n i = (n+1) * n / 2$

Veamos el caso particular

$$\sum_{i=0}^{10} i = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$$

Si agrupamos los números de a pares de la siguiente manera:

$$10 + 1; 9 + 2; 8 + 3; 7 + 4; 6 + 5$$

Vemos que hay 5 grupos que suman 11, es decir

$$5 \text{ grupos} = 10/2$$

$$\text{Que suman } 11 \Rightarrow 10/2 * 11 = 55$$

Para n

$$\begin{array}{cccccc} n & n-1 & n-2 & \dots & n/2 + 1 \\ 1 & 2 & 3 & \dots & n/2 \end{array}$$

Se puede observar que hay $n/2$ grupos y que todos suman $n+1$, la suma es $(n+1) * n/2 =$

$$\frac{1}{2}n^2 + \frac{1}{2}n < 2n^2$$

$$2n^2 \text{ está en } O(2) O(n^2)$$

$$O(2) O(n^2) \text{ está en } O(n^2)$$

Ejercicio2 Combinatoria

En una habitación hay **n** personas que se quieren saludar entre si.

Las reglas para saludarse son:

- 1) Cada persona puede saludar solo una persona a la vez.
 - 2) El saludo es simétrico: Si a saluda a b, se considera que b saluda a a en el mismo saludo.
 - 3) Cada saludo demora 1 segundo
 - 4) Si hay n personas, puede haber hasta $n/2$ saludos simultáneos.
- a) Cuanto tiempo(en segundos) se necesita que todos queden saludados?
- Ayuda: Hacer una tabla para el caso de 4 y de 8 personas.
- b) Cuantos saludos hay en total?
- c) Como cambia a) si cambiamos la regla 4) de manera que solo puede haber un saludo a la vez?
- d) Como cambia a) si cambiamos la regla 1) de manera que cada persona puede saludar a mas de una persona a la vez?
- e) Como cambia a) si quitamos la regla 1) y además, cada persona que es saludada sale de la habitación.

Ayuda: Hacer una tabla para el caso de 4 y de 8 personas.

	1	2	3	4
1	x	1	3	2
2		x	2	3
3			x	1
4				x

Tabla1: Tabla para 4 personas

Como se ve en la tabla1, para 4 personas se necesitan 3 segundos.

La notación O en dos variables

Ejemplo

Sea f un algoritmo trivial que suma dos vectores v_1 y v_2

¿Cuáles son las variables que definen el tamaño de la entrada?

Claramente la complejidad no puede depender solo del tamaño de v_1 o del tamaño de v_2 .

Es necesario contemplar una cota que dependa de

Sea n_1 el tamaño de v_1

Sea n_2 el tamaño de v_2

Se puede dar una definición formal de la complejidad para dos variables:

$$f_{n,k} \in O(g_{n,k}) \Leftrightarrow \exists n_0, k_0, c \text{ tales que para todo } n > n_0 \text{ y para todo } k > k_0 \\ f_{(n,k)} < c g_{(n,k)}$$

Donde n y k son el tamaño de la instancia que es parámetro del algoritmo.
y n_0 significa, "el primer valor de n " para el cual se cumple.

Ejercicio3

Demostrar por definición que la complejidad de f esta en $O(n + k)$

Programación II

Práctica 01: Complejidad algorítmica

Notación O

$f \in O(g) \Leftrightarrow \exists n_0, c$ tales que para todo $n > n_0$
 $f(n) < c g(n)$

Donde n , es el tamaño de la instancia que es parámetro del algoritmo.

Reglas y Álgebra de Ordenes

- 1) $O(1) \leq O(\log n) \leq O(n) \leq O(n^k) \leq O(k^n) \leq O(n!) \leq O(n^n)$
- 2) $O(f) + O(g) = O(f + g) = O(\max\{f, g\})$
- 3) $O(f) \cdot O(g) = O(fg)$
- 4) $O(k) = O(1)$ para todo k constante.

Ejemplo: $O(2n)$
 $= O(2) * O(n) \quad //$ por regla 3
 $= O(1) * O(n) \quad //$ por regla 4
 $= O(n)$

5) $\sum_{i=1}^k O(f) = O(\sum_{i=1}^k f) = O(kf)$
Si k es constante, entonces vale $O(f)$

6) $\sum_{i=1}^k i = \frac{k*(k+1)}{2} = O(k^2)$

Calcular el Orden de complejidad algorítmica para el peor caso y para el caso promedio de los siguientes ejercicios.

***Ejercicio1** Determinar la complejidad para el peor de caso de los siguientes algoritmos

a)

```
Void función(int n)
    if n == 1
        for (i = 1; i<n; i++)
    else
        for (i = 1; i< $n^2$ ; i++)
```

b)

```
Void función1(int x)
    if f(x)
        g(x)
    else
        h(x)
```

***Ejercicio2 Burbujeo**

a)

```
for(int i = 0; i < n; i++) {
    for(int z = 0; z < n; z++) {
        if(vector[z] > vector[z + 1]) {
            aux = vector[z];
            vector[z] = vector[z + 1];
            vector[z + 1] = aux;
        }
    }
}
```

b) ¿Como cambia el orden si cambiamos $z = 0$ por $z = i$, en el segundo “for”?

Variables de complejidad

Antes de continuar identificaremos la o las variables que representan el tamaño de los datos de entrada al algoritmo.

La función $f()$ que me mide la complejidad, estará en función de dichas variables.

Ejemplo1

```
Void función1(k, h)
    for (i=0, i < k, i++)
        h++
```

Los candidatos a variables son k y h , pero como $h++$ está en $O(1)$, la complejidad no depende de h . Así que la variable de complejidad es k y esto surge de que el ciclo se repite k veces.

Variable: k

La complejidad de funcion1 es $O(k)$

```
Void función2(k, h)
    for (i=0, i <  $2^k$ , i++)
        h++
```

Notar que el ciclo no termina en k pasos.

La complejidad de funcion2 es $O(2^k)$

Ejemplo2

```
Void función3(k, h)
    for (i=0, i < k, i++) { }

    for (i=0, i < h, i++) { }
```

El primer ciclo depende de k, pero el segundo ciclo depende de h.

Entonces que variable usaremos?

En este caso se necesitan las dos

Pero cual es la complejidad? $O(k)$ o $O(h)$. Como saber si $k > h$ o si $h > k$?

La realidad es que en general no se sabe.

Entonces tendremos que poner alguna de las siguientes expresiones

- (1) $O(k + h)$
- (2) $O(\max(k, h))$

La segunda expresión es más precisa.

Supongamos como ejemplo que $k > h$. claramente $k + h$ sigue siendo mayor que k .

Entonces como puede ser que la complejidad sea $O(k)$

Vamos a buscar una cota:

Como $k > h$, sabemos que $2k > k + h$

Entonces la complejidad es $O(2k)$, pero por el álgebra de la complejidad es lo mismo que $O(k)$

$O(2k) = O(2)$ $O(k) = O(1)$ $O(k) = O(k)$

Ejercicio3 Test de primalidad

```
1)
boolean esPrimo1(n) {
    int i = 1
    int divisores = 0
    while i < n
        if divide(i,n)
            divisores ++
        i++
    return (divisores < 2)
}
```

2) Para la parte 2 utilizar la siguiente proposición P que dice:

Solo hace falta chequear los divisores hasta \sqrt{n} para ver si esPrimo(n)

Demostración

Demostrarímos por absurdo que eso no es necesario.

Asumiremos que No P es cierto.

Hipótesis(No P): Vamos a suponer que existe un divisor mayor que \sqrt{n} que hace falta ver.

Sea k un divisor de n tal que $k > \sqrt{n} \Rightarrow$

Existe q, otro divisor, pues $q = n / k$

Si $q \leq \sqrt{n} \Rightarrow$ no era necesario ver k.

Abs

Con q me alcanza para el test.

Si $q > \sqrt{n} \Rightarrow k * q > \sqrt{n} * \sqrt{n}$

Pero $\sqrt{n} * \sqrt{n} = n \Rightarrow$

$k * q > n$

Abs

$\Rightarrow P$ es cierto: No hace falta ver divisores mayores que \sqrt{n}

```
boolean esPrimo2(n) {
    int i = 1
    int divisores = 0
    while i < Math.SQRT(n) //por propiedad1
        if divide(i,n)
            divisores ++
        i++
    return (divisores < 2)
}
```

3) Utilizar la siguiente propiedad:

La cantidad de primos en los primeros n números no es mayor que $n / \ln(n)$.

Solo en este caso \ln esta en base e, en lugar de base 2 como en el resto de las prácticas.

Por ej: $\ln_e 32 = 3,46$; mientras que $\ln_2 32 = 5$

Implementar `esPrimo3`, el cual recibe la lista de números primos menores que \sqrt{n} y n como parámetros.

Calcular el nuevo Orden de complejidad.

Ayuda:

Calcular la lista de primos a mano para realizar los ejemplos.

La complejidad tiene que ser mejor que el ejercicio 2

```
boolean esPrimo3(listaPrimos, n)
```

Opcional 4) Implementar algún otro test de primalidad y calcular el orden.

http://es.wikipedia.org/wiki/Test_de_primalidad

Ejercicio4 Fósforos

Se tiene una caja de fósforos con n fósforos nuevos.

Cada vez que quiera utilizar uno, el procedimiento es el siguiente:

Tomo un fosforo de la caja. Si está quemado, tomo otro, y así hasta encontrar uno nuevo.

Luego utilizo el fosforo y lo mezclo junto con los otros fósforos usados en la caja

- a) Cual es la complejidad de encontrar un fosforo sin quemar dado que ya consumí la mitad de la caja?
- b) Cual es la complejidad de consumir n fósforos?

*Ejercicio5 Definición de O

Utilizando la definición de O ($f \in O(g) \Leftrightarrow \exists n_0, c$ tales que para todo $n > n_0 \Rightarrow f(n) < c g(n)$),
 Encontrar n_0 y c para justificar el orden de los siguientes tiempos de ejecución.
 Decidir en qué Orden(el mas chico) están.

- a) $n^2 - n^2 - 10$
- b) $n^2 + 2n + 100$
- c) $n^3 + 2n^2 + 10$
- d) $\sqrt{n} + \log n + 1000$
- e) $n^n + n^{100} + 10$

Ejercicio6 Varias variables

Implementar un algoritmo que recorra y muestre una matriz de n filas y k columnas.
 Calcular la complejidad de dicho algoritmo utilizando la definición de O para dos variables

$$f_{n,k} \in O(g_{n,k}) \Leftrightarrow \exists n_0, k_0, c$$
 tales que para todo $n > n_0, k > k_0 \Rightarrow f(n,k) < c g(n,k)$

Ayuda: Nombrar las variables antes de calcular la complejidad

Ejercicio7 Combinatoria

En una habitación hay n personas que se quieren saludar entre sí.
 Las reglas para saludarse son:

- 1) Cada persona puede saludar solo una persona a la vez.
- 2) El saludo es simétrico: Si a saluda a b, se considera que b saluda a a en el mismo saludo.
- 3) Cada saludo demora 1 segundo
- 4) Si hay n personas, puede haber hasta $n/2$ saludos simultáneos.

- a) Cuánto tiempo(en segundos) se necesita que todos queden saludados?

Ayuda: Hacer una tabla para el caso de 4 y de 8 personas.

- b) Cuántos saludos hay en total?

- c) ¿Cómo cambia a) si cambiamos la regla 4) de manera que solo puede haber un saludo a la vez?

- d) ¿Cómo cambia a) si cambiamos la regla 1) de manera que cada persona puede saludar a más de una persona a la vez?

- e) ¿Cómo cambia a) si quitamos la regla 1) y además, cada persona que es saludada sale de la habitación.

Ayuda: Hacer una tabla para el caso de 4 y de 8 personas.

Bibliografía:

[Cormen1990]: <http://www.amazon.com/exec/obidos/ISBN=0262031418/none01A/>

[Cormen2001]: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford: Introduction to Algorithms. Sección 31.8: "Primality testing", pp.887–896. MIT Press & McGraw-Hill, 2a edición, 2001. (ISBN 0-262-03293-7.)

Programación II

Tipos Abstractos de Datos(TAD)

Definición

Un Tipo Abstracto de Datos es un conjunto de valores y de operaciones definidos mediante una especificación independiente de cualquier representación.

$$\text{TAD} = \text{valores} + \text{operaciones}$$

La manipulación de un TAD sólo depende de su especificación, nunca de su implementación.

Especificación / implementación

Dada una especificación de TAD hay muchas implementaciones válidas.

Un cambio de implementación de un TAD es transparente a los programas que lo utilizan

Reusabilidad

Se puede:

- Implementar un TAD's sólo a partir de la especificación, sin saber para qué se van a usar.
- Utilizar los TAD's sólo conociendo la especificación.
- Cambiar la implementación un TAD utilizado a otra más eficiente

Abstracción:

- Destacar los aspectos relevantes del objeto.
- Ignorar los aspectos irrelevantes del mismo

Abstracción funcional:

crear procedimientos y funciones e invocarlos mediante un nombre donde se destaca qué hace la función y se ignora cómo lo hace. El usuario sólo necesita conocer la especificación de la abstracción (el qué) y puede ignorar el resto de los detalles (el cómo).

Ocultamiento de información:

Ocultar *decisiones de diseño* en un programa susceptible de cambios con la idea de proteger a otras partes del código si éstos se producen.

Proteger una decisión de diseño supone proporcionar una interfaz estable que proteja el resto del programa de la implementación (susceptible de cambios).

Encapsulamiento:

Mantener todas las características, habilidades y responsabilidades de un objeto por separado.

Especificación comportamientos esperados inesperados:

Una buena práctica de diseño es ser declarativo con los nombres(hacer lo que se dice) y ser coherente respecto de los efectos secundarios de nuestros.

Por ejemplo, avisar si tendremos aliasing. O si el Tad queda alterado de forma no deseable luego de cierta operación.

Invariante de representación

Definición: Afirmación sobre un objeto que debe ser cierta en todo momento.

Durante la implementación es necesario establecer que instancias van a representar estados válidos de TAD y cuales no.

Durante la especificación solamente describimos la interfaz del TAD (operaciones), pero no decimos que otras cosas tienen que valer en todo momento.

Durante la implementación debemos garantizar que esas operaciones no corrompan el estado interno, es decir, que dichas operaciones no violen el invariante de representación.

Ejercicios**Ejercicio1 Especificar el TAD Nat**

Nat son los números naturales de matemática

Especificación:

```
Nat(Integer n) {}           // Constructor.  
void sumar(Nat n) {}
```

Invariante de representación: $n \geq 0$

Ejercicio2 Completar la implementación del TAD Fecha según la siguiente especificación

Asumir que los días van del 1 al 31

```
Fecha(Integer dia, Integer mes, Integer año) {}           // Constructor.  
void sumar(Fecha f) {}  
Integer dia() {}  
Integer mes() {}  
Integer año() {}
```

Invariante de representación:

$$\begin{aligned} 31 &\geq \text{dia} \geq 1 \\ 12 &\geq \text{mes} \geq 1 \\ 9999 &\geq \text{año} \geq 1 \end{aligned}$$
NOTA

Todas las clases de un TAD deben implementar el método `toString` y un test que compruebe algún caso de uso.

```
public class Fecha {  
  
    private int dia;  
    private int mes;  
    private int año;  
  
    Fecha(int dia, int mes, int año){  
        if (dia>31){  
            throw new RuntimeException("El dia debe estar entre 1 y  
            31");  
        }  
        //...  
        this.dia = dia;  
        this.mes = mes;  
        //...  
    }  
  
    public int dia(){  
        return dia;  
    }  
  
    public int mes(){  
        return mes;  
    }  
  
    public int año(){  
        return año;  
    }  
}
```

```
public void sumar(Fecha fecha) {
    dia = dia + fecha.dia;
    if (dia > 31) {
        mes = mes + 1;
        dia = dia - 31;
    }
    //...
}

@Override
public String toString() {
    return dia + "/" + mes + "/" + año;
}

public static void main(String[] args) {
    // TODO Auto-generated method stub
    Fecha fechal = new Fecha(1,1,1);
    Fecha fecha2 = new Fecha(31,1,1);

    fechal.sumar(fecha2);

    System.out.println(fechal.toString());
}

}
```

Verificación del Invariante

Algo que es un poco confuso sobre los invariantes es que a veces, estos dejan de cumplirse. Por ejemplo, en la mitad de sumar,

`dia = dia + fecha.dia`, el invariante no se cumple.

Este tipo de violación al invariante es aceptable; de hecho, usualmente es imposible modificar un objeto sin violar un invariante al menos por un momento. Normalmente el requerimiento es que todo método que viole un invariable debe restablecerlo antes de terminar.

TAD Mesa

Con la siguiente especificación “Una mesa se compone de una tabla y unas patas”, realizando abstracciones, implementaremos el TAD Mesa

Considerar

-Un Tad tiene tantas clases como sean necesario. No necesariamente un TAD es una clase.

 Un TAD → Varias clases

-Hay que abstraer lo que queremos modelar y saber despreciar el resto de los atributos de la realidad.

En este caso modelaremos

-Una mesa

-Una Pata

Y no modelaremos

-Una Tabla

La justificación en este caso es el nivel de granularidad que nos interesa tener. Como no hay mas hipótesis sobre el dominio del problema, por ahora mantendremos esta abstracción.

```
public class Pata {}

public class Mesa {      // TAD
    Pata [] patas;      // Estructura de datos
    public Mesa(Integer cantPatas){
        patas = new Pata[cantPatas];
    }
}
```

Si bien es cierto que los TAD tienen estructuras de datos dentro, no es cierto que un TAD es una estructura de datos.

La otra cosa que notamos es que no hay forma de testear si esto esta funcionando.

Agregaremos código para eso:

```
public String toString(){
    return "Cantidad de patas " + patas.length;
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Mesa mesa = new Mesa(4);  
        System.out.println(mesa.toString());  
    }  
}
```



The screenshot shows an IDE interface with several tabs at the top: 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active, displaying the output of a Java application named 'Test'. The output shows the string 'Cantidad de patas 4'.

```
<terminated> Test (2) [Java Application] C:\Program Files\Java\jre7\  
Cantidad de patas 4
```

Ejercicio3:

- Cual es el irep de Mesa?
- Modificar el TAD mesa de manera que considere:

-De que material esta hecha la mesa

Ayuda: Considerar agregar otra clase

-Que puedo cambiar la cantidad de patas en el ciclo de vida de mesa

Ayuda: Considerar utilizar otra estructura de datos

Programación II

Práctica 02: Tipos Abstractos de Datos(TAD)

Versión del 01/04/2014

Introducción

En las clases teóricas se estudiaron las ventajas que tienen los TADs. A continuación las repasamos:

Abstracción:

- Destacar los aspectos relevantes del objeto.
- Ignorar los aspectos irrelevantes del mismo

Abstracción funcional:

crear procedimientos y funciones e invocarlos mediante un nombre donde se destaca qué hace la función y se ignora cómo lo hace. El usuario sólo necesita conocer la especificación de la abstracción (el qué) y puede ignorar el resto de los detalles (el cómo).

Ocultamiento de información:

Ocultar *decisiones de diseño* en un programa susceptible de cambios con la idea de proteger a otras partes del código si éstos se producen.

Proteger una decisión de diseño supone proporcionar una interfaz estable que proteja el resto del programa de la implementación (susceptible de cambios).

Encapsulamiento:

Mantener todas las características, habilidades y responsabilidades de un objeto por separado.

Especificación de comportamientos inesperados:

Una buena práctica de diseño es ser declarativo con los nombres(hacer lo que se dice) y ser coherente respecto de los efectos secundarios de nuestros.

Por ejemplo, avisar si tendremos aliasing. O si el Tad queda alterado de forma no deseable luego de cierta operación.

Para todos los ejercicios se debe escribir el ***invariante de representación(irep)*** antes de comenzar la implementación.

Como ejemplo un irep de los Números naturales podría ser:

Las instancias validadas del TAD Nat, que representa a los números naturales son:
Los números positivos

Ejercicio1:Números naturales(Nat)

En algunos casos se necesita modificar el comportamiento de TADs que ya existen.
Realizaremos una implementación de los números naturales(N) basándonos en Integer como el tipo soporte.

Como Nat se define alrededor de Integer y semánticamente son similares, se dice que **Nat envuelve**(redefine) a Integer.

Esto se realiza principalmente, para modificar el comportamiento del tipo base, sin modificar al tipo base en si mismo.

En este caso, no queremos números negativos.

Especificación

```
Nat(Integer n) {}           // Constructor. n≥0
sumar(Nat n) {}
```

Implementar Nat

Notas: Ocultamiento de información

Implementar también `toString()` de manera de poder mostrar los resultados.
Cualquier función o variable que utilice la clase salvo las peticiones en la implementación, deben ser privadas.

***Ejercicio2a: Conjunto de enteros**

Definir el TAD Conjunto, que se comporta como el conjunto de la teoría de conjuntos.
No se puede utilizar **Set** ni ninguna de sus subclases para implementarlo.
Es decir, no queremos envolver Set, queremos definirlo basado en otros tipos primivitos.

Sujerimos utilizar la clase **Array**(class **Arrays**) para el nuevo Tad.

Especificación

```
Conjunto() {}                      // Constructor1
Conjunto(IntegertamañoMAX) {}       // Constructor2
Integertamaño() {}
void Agregar(Integer i) {}
Integer iesimo(Intergerindice) {}   // indice< tamaño()

Void union(Conjunto c) {}           // union1: Destructiva
Conjunto union2(Conjunto c) {}       // union2: No debe tener Aliasing!
Void interseccion(Conjunto c) {}    // interseccion 1: Destructiva
Conjunto interseccion2(Conjunto c) {}
```

Donde **tamañoMAX** determina el tamaño máximo al que puede crecer el conjunto, si se utiliza ese constructor.

```
Integertamaño()
    returnconj.size();               // NO devolver tamañoMAX ¡!
}
```

Para evitar Aliasing, “union2” e “interseccion2” deben devolver un nuevo conjunto, que no refiera al conjunto de la clase.

Nota1: Encapsulamiento

Siempre que sea posible, se deben utilizar las funciones de la clase en lugar de preguntar por sus variables internas o privadas(**this**).

En este caso, utilizar **tamaño()**, en lugar de **this.vector.size()** siempre que sea posible.

Nota2: Reutilización: implementar union/interseccion intentando utilizar iesimo/agregar.

Siempre que sea posible, se deben reutilizar los métodos de la propia clase para implementar nuevos métodos.

2b) Calcular la complejidad de

```
Void union(Conjunto c) {}           // union1: Destructiva
Conjunto union2(Conjunto c) {}       // union2: No debe tener Aliasing!
Void interseccion(Conjunto c) {}    // interseccion 1: Destructiva
Conjunto interseccion2(Conjunto c) {}
```

Asumiendo que:

El peor caso de agregar esta en $O(n)$

Donde n es el tamaño del conjunto mas grande.

2c)Calcular la complejidad de

```
Void union(Conjunto c) {}           // union1: Destructiva
Void interseccion(Conjunto c) {}     // interseccion 1: Destructiva
```

Con una mejor cota.

Para ello utilizar que:

- n1 es el tamaño de *this*
- n2 es el tamaño de *c*

Por ejemplo, la complejidad del siguiente código

If *this.pertenece(c.iesimo(n))*, es $O(n_1 + n_2)$ porque

Pertenece esta en $O(n_1)$

Iesimoesta en $O(n_2)$

Que pasa cuando $n_1 == n_2$?

***Ejercicio3:Pila de enteros**

- a) Implementar la Pila básica, sin envolver el tipo Stack.

Especificación

```
PilaBasica(){}           // Constructor1
Void agregar (Integer i){} // Agrega en la cima.
Integer quitar(){}        // Quita de la cima.
Integer cima(){}          // Devuelve la cima sin quitarla.
```

- b) Implementar la Pila extendida extendiendo la Pila básica.

ClassPilaExtendidaextendsPilaBasica...

Especificación(extensión)

```
void ordenar(){}
void MezclarOrdenado(PilaExtendida p){}
Pila Extendida MezclarOrdenado(PilaExtendida p){}
```

MezclarOrdenado() mezcla y p con la pila interna y luego ordena la pila interna
Pila Extendida MezclarOrdenado(PilaExtendida p) hace lo mismo, pero devuelve una referencia a otra pila (no genera aliasing).

- c) Calcular el Orden de complejidad de los métodos ordenar() y MezclarOrdenado()

Ejercicio4: Polinomio de raíces y coeficientes enteros

En matemáticas, un polinomio es una expresión matemática constituida por un conjunto finito de variables (no determinadas o desconocidas) y constantes (números fijos llamados coeficientes), utilizando únicamente las operaciones aritméticas de suma, resta y multiplicación, así como también exponentes enteros positivos.

Los coeficientes del polinomio serán números con a_i distinto de cero y $n \in \mathbb{N}$, entonces un polinomio, P , de grado n en la variable x es un objeto de la forma

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0 x^0.$$

Por ejemplo

$$P(x) = -1x + 2x^3$$

$$P(2) = -2 + 2*2^3 = 14$$

a) Implementar el TAD Poli con la siguiente interfaz

poli()

```
void agregarTermino(coeficiente, posicion): Si el termino ya existía, se redefine
int evaluar(x)
int raiz(ini, fin) : que devuelve la 1er raíz que se encuentre ini y fin, asumiendo que ini y fin tienen
diferente signo y que no son raíz.
```

irep: Toda instancia de p de Poli cumple:

-Ningún termino esta repetido.

-Todos los términos y raíces son enteras.

pseudocódigo de raiz

```
medio = (ini + fin) / 2
mientras evaluar(medio) != 0 y ini < fin
    si evaluar(ini) > evaluar(fin)
        si evaluar(medio) > 0
            ini = medio
        sino
            fin = medio
    sino
        si evaluar(medio) > 0
            fin = medio
        sino
            ini = medio

medio = (ini + fin) / 2           //parte entera
```

b) Considerar la complejidad de obtener una raíz:

¿Cual es la variable?: ini o fin?

En este caso podríamos decir que el algoritmo itera en función de fin - ini, ya que los números entre 0 y ini no son examinados

Sea $k = \text{fin} - \text{ini}$ la variable de la complejidad

Que pasa en cada ciclo con k?

Veamos que hay cuatro posibilidades

En cualquier caso o bien $\text{ini} = (\text{ini} + \text{fin}) / 2$ o bien $\text{ini} = (\text{ini} + \text{fin}) / 2$, por lo tanto k vale la mitad luego de cada iteración

Si m es la cantidad de iteraciones y $k > 1$ porque paramos cuando $\text{ini} == \text{fin}$, en el peor de los casos, si no encontre la raiz antes:

$$\begin{aligned}k/2^m &< 1 \\k &< 2^m \\\log k &< m\end{aligned}$$

La complejidad es $\log k$

Calcular la complejidad de evaluar

Ejercicio5: Diccionario de int(DiccInt)

Un Diccionario es una generalización del concepto de conjunto, en la cual cada elemento que pertenece al conjunto (denominado clave) tiene asociado un valor:

- Los elementos del Diccionario son pares (clave, valor).
- No pueden existir claves repetidas.
- Sin embargo si que pueden existir valores repetidos.
- Los elementos se localizan mediante su clave.

Especificación

```
DiccInt(){}           // Constructor1
Void agregar (Integer i, String s){}      //
String obtener(Integer i){}                //
Boolean Pertenece(Integer i){}             //
```

Implementar el TAD DiccInt sin utilizar Map.

Ejercicio5: Agenda

Implementar una agenda basada en el TAD diccionario

La agenda deberá tener una clave basada en el dni

Y un significado basado en: String nombre, Integer teléfono y String dirección

Ayuda: Considerar utilizar otro TAD como abstracción del significado del diccionario.

Se puede modificar el TAD Diccionario, de manera que

```
Void agregar (Integer i, Significado s){}
```

Ejercicio6: Abstracción

Re-implementar el TAD Conjunto utilizando **sólo** el TAD Pila

Considerar que las pilas admiten elementos repetidos.

- a) Implementar solamente los métodos agregar e unión (además de la definición de la clase)
- b) Calcular la complejidad de dichos métodos.

No se puede modificar el TAD Pila.

Ayuda: Utilizar la interfaz de Pila y el siguiente diseño:

Class Conj

```
Pila datos; // no agregar mas variables
Public Conj()
```

Ejercicio 6: Matriz infinita de booleanos

El departamento de matemáticas de la UNGS nos pidió ayuda para implementar el TAD “Matriz infinita de booleanos”

La implementación(trivial) actual es la siguiente:

```
public class MIB {  
  
    private int i;  
    private int j;  
    private Boolean [][] mat;  
  
    public MIB(int i, int j){  
        this.i=i;  
        this.j=j;  
        mat = new Boolean[i][j];  
    }  
  
    Boolean leerValor(int i, int j){  
        return mat[i][j];  
    }  
    public void setearValor(int i, int j, Boolean x){  
        mat[i][j]=x;  
    }  
}
```

} El problema de esta implementación, es que ocupa demasiada memoria.

Por ejemplo “MIB x = new MIB(100000,200000)” arroja:

java.lang.OutOfMemoryError: Java heap space

Lo que se solicita es:

- a) Implementar la función test() que implemente el siguiente testeo:

```
MIB x = new MIB(100000,200000);  
x.setearvalor(5000,3000,true);  
System.out.Println(x.leervalor(5000,3000)); // debe devolver true  
System.out.Println(x.leervalor(5000,3001)); // debe devolver false
```

- b) Hacer un diseño de un TAD que represente la matriz de manera mas eficiente.

Ayuda: Se puede asumir que la mayoría de los valores van a ser **false**.

- c) Calcular el orden de complejidad para los métodos **leerValor** y **setearValor** de la implementación trivial.
- d) Calcular el orden de complejidad para los métodos **leerValor** y **setearValor** de la implementación mejorada.

Ejercicio 7: Abstracción: Monopolio

Se desea modelar el clásico juego Monopolio, pero con la cantidad de casilleros definida por el usuario. Para ello se tiene un tablero de **n** posiciones y **2** jugadores.

Instancia: para $n = 40$



Sistema de juego simplificado:

Se pide por única vez el tamaño del tablero(n), y los precios de cada casilla.

En cada turno cada jugador tira un dado (un número de 1 a 5).

El tablero es circular, de manera que cuando se llega al final, se vuelva a dar otra vuelta.

Cada jugador comienza con \$1000.

Cuando el jugador cae en una casilla, pueden pasar dos cosas:

- 1) Que la casilla no se haya comprado aun. En ese caso el jugador esta obligado a comprarla y de ahí en mas la casilla pertenecerá a dicho jugador
- 2) Que la casilla se encuentre comprada.
Si la casilla pertenece al jugador que compro la casilla no se hace nada.
Si la casilla pertenece al otro jugador, se debe pagar un alquiler, igual al 1% del monto de compra de dicha casilla.

El juego termina cuando alguno de los dos jugadores se queda sin dinero.

Esto puede suceder por alguno de los siguientes motivos:

- 1) El jugador no puede comprar una casilla.
- 2) El jugador no puede pagar el alquiler.

Diseño:

Diseñar e implementar el TAD Juego de “Monopolio” (Mono)

Es obligatorio que el TAD Mono utilice al menos otro TAD como TAD soporte.

Es decir:

- Que Mono no podrá estar implementado únicamente sobre los tipos primitivos de Java (ni java.util).
- Que Mono utilice varias clases para ser implementado.

Ayuda: Ver cuales de las siguientes clases son necesarias para el TAD Mono, cuales no, y por que:

- Jugador
- Dado
- Tablero
- Reglamento
- Mono

Interfaz obligatoria:

```
Mono (n)          //tamaño del tablero, donde n > 1
String ganador () //devuelve el nro de jugador ganador o 0 si no hay
                  //ganador por el momento
Void agregarCasilla(int casilla,double precio)
                  //Asigna el precio a la casilla
void jugar()      // tira los dos dados
String ver()       // muestra el estado del tablero y los jugadores
```

Ejemplo del código principal:

```
Mono mono = new Mono(6);           // Instancia: n == 6

Mono.agregarCasilla(0,100);        //La Casilla 0 vale $100
Mono.agregarCasilla(3,10);         //La Casilla 3 vale $10
while mono.ganador() == ""{        //como máximo hacerlo 1000 veces
    mono.jugar();                 // Los 2 jugadores "tiran" los dados
    System.out.println(mono.ver()); //Se muestra un resumen del
tablero
}
System.out.println(mono.ganador());
```

Implementación

Implementar un test que al menos pruebe el ejemplo del código principal.

***Ejercicio8: Abstracción: Un TAD basado en otro TAD**

Por lo general realizaremos diseños utilizando otro diseños ya probados, como soporte.

Por ejemplo, diseñaremos un Conjunto utilizando el TAD Pila como soporte.

```
public class ConjPilaInt {  
    PilaInt datos;  
  
    public ConjPilaInt(){  
        datos = new PilaInt();  
    }  
  
    public Integer iesimo(int i){  
        int indice =0;  
        Integer elem=null;  
        PilaInt aux = new PilaInt();  
  
        while (indice<=i){  
            elem = datos.tope();  
            aux.agregar(datos.quitar());  
            indice++;  
        }  
  
        while (!aux.vacia()){  
            datos.agregar(aux.quitar());  
        }  
  
        return elem;  
    }  
  
    public void agregar(Integer elem){  
        if (!pertenece(elem))  
            datos.agregar(elem);  
    }  
  
    boolean pertenece(Integer elem){  
        boolean ret = false;  
        PilaInt aux = new PilaInt();  
  
        while (!datos.vacia()){  
            ret =ret || datos.tope() == elem;  
            aux.agregar(datos.quitar());  
        }  
  
        while (!aux.vacia()){  
            datos.agregar(aux.quitar());  
        }  
  
        return ret;  
    }  
}
```



```
public class TestConjPila {  
  
    public static void main(String[] args) {  
        ConjPilaInt c = new ConjPilaInt();  
  
        c.agregar(2);  
        c.agregar(8);  
        c.agregar(8);  
        c.agregar(9);  
  
        System.out.println(c.iesimo(2) + " "  
                           + c.iesimo(1) + " " + c.iesimo(0));  
    }  
  
}
```



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:
<terminated> TestConjPila [Java Application] C:\Program Files\Java
2 8 9

Observaciones:

Como la Pila admite elementos repetidos, pero el conjunto no, hay que adaptar el conjunto para no utilizar directamente el método Pila.agregar().

Para ello es necesario chequear que el elemento no este repetido.

```
public void agregar(Integer elem){  
    if (!pertenece(elem))  
        datos.agregar(elem);  
}
```

Para implementar iesimo es mas fácil; pero de todas maneras es necesario “reconstruir” la estructura de datos, porque la operación de lectura de la pila es destructiva.

```
while (!aux.vacia()){  
    datos.agregar(aux.quitar());  
}
```

Ejercicios:

- 1) Implementar el TAD ConjInt sobre el arreglo estático(Integer []) de java.
Al que llamaremos ConjArrayInt

Ayuda: Considerar redimensionar el array si es necesario

Ejemplo

```
int[] a = {1, 2, 3};  
// hago una copia de a con un elemento mas  
a = Arrays.copyOf(a, a.length + 1);  
for (int i : a)  
    System.out.println(i);
```

- 2) Calcular la complejidad de

ConjPilaInt.agregar
ConjPilaInt.iesimo

ConjArrayInt.agregar
ConjArrayInt.iesimo

Bibliografía:

[Cormen1990]:<http://www.amazon.com/exec/obidos/ISBN=0262031418/none01A/>

[Cormen2001]: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford: Introduction to Algorithms. Sección 31.8: "Primality testing", pp.887–896. MIT Press & McGraw-Hill, 2a edición,2001. (ISBN 0-262-03293-7.)

Algoritmos recursivos

Un **algoritmo recursivo** es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva.

El código 1a implementa una versión recursiva de la función Factorial $N \rightarrow N$
El código 1b implementa una versión iterativa de la misma función.

Generalmente, si la primera llamada al subprograma se plantea sobre un problema de tamaño u orden n , cada nueva ejecución recurrente del mismo se planteará sobre problemas, de igual naturaleza que el original, pero de un tamaño menor que n .

De esta forma, al ir reduciendo progresivamente la complejidad del problema que resolver, llegará un momento en que su resolución sea más o menos trivial (o, al menos, suficientemente manejable como para resolverlo de forma no recursiva).

En esa situación diremos que estamos ante un **caso base** de la recursividad.

```
static public int factorialRecur(int n) {  
  
    if (n == 1) {  
        return 1; // caso base  
    } else {  
        return n * factorialRecur(n-1);  
    }  
}
```

Código 1a: factorial recursivo

```
static public int factorialIter(int n) {  
    int ret = n;  
    for (int i=n-1; i>1; i--) {  
        ret = ret * i;  
    }  
    return ret;  
}
```

Código 1b: factorial iterativo

Las claves para construir un subprograma recursivo son:

- (1) Cada llamada recursiva se debería definir sobre un problema de menor complejidad (algo más fácil de resolver).
- (2) Ha de existir al menos un caso base para evitar que la recurrencia sea infinita.

Si ocurren (1) y (2) tenemos garantizado que el programa finalizara en algún momento.

En el caso de `factorialRecur()`, cada llamada recursiva se realiza con un problema de tamaño $n-1$.

El caso base ocurre cuando $n = 1$.

Iterativo Vs Recursivo

Las versiones recursivas suelen ser mas breves(menos líneas de código), pero mas lentas (tiempo de ejecución).

Y las versiones iterativas suelen ser mas declarativas(esta claro como se resuelve el problema), pero se ven en problemas a la hora de recorrer estructuras de naturaleza recursiva, por ejemplo, un árbol binario.

Por lo tanto, no hay una regla general que indique cuando resolver un problema de una manera u otra.

Tipos de recursión

Hay dos tipos de recursión.

Una recursión es lineal, si existe una única llamada recursiva, por ejemplo, en factorialRecur() .

La recursión es no lineal si existen dos o mas llamadas recursivas, como por ejemplo, en fibonacciRecur() (Ver Código 2a).

```
static int fibonacciRecur(int n){    O( (1+√5)/2 )  
    if (n <= 1)  
        return n;           // f0 = 0 y f1= 1  
    else  
        return fibonacciRecur(n - 1) + fibonacciRecur(n - 2);  
    }
```

Código 2a: recursión no lineal

Ejercicio1: Como demostrar que fibonacciRecur() finaliza?

- (1)Todas las llamadas recursivas se invocan con un valor < n
- (2)Tiene un caso no recursivo en el cual el programa finaliza.

Ejercicio2: Como será la versión iterativa?

```
static int fibonacciIter(int n){    // O(n)  
    int n0 = 0;  
    int n1 = 1;  
    int n2 = n0 + n1;  
  
    if (n==0) { return n0; }  
    if (n==1) { return n1; }  
  
    if (n>2) {  
        for (int i=2; i<n; i++){  
            n2 = n1 + n0;  
            n0 = n1;  
            n1 = n2;  
        }  
    }  
    return n2;  
}
```

Código 2b: fibonacci iterativo

Observaciones:

El código 2a y el código 2b hacen lo mismo?

Lamentablemente no hay una formula general, para demostrar que dos códigos hacen lo mismo.

Afortunadamente para fibo, si se puede demostrar que ambos códigos son equivalentes.

Pero la demostración escapa al alcance de la materia.

Sin embargo, en la clase de complejidad demostrarímos que versión es la que tarda más tiempo en finalizar.

Ejercicio1: El problema de las n torres

El problema de las n torres es una variación del problema de las 8 reinas propuesto por el ajedrecista alemán Max Bezzel en 1848¹.

En el juego del ajedrez la torre amenaza a aquellas piezas que se encuentren en su misma fila o columna (Figura1a).

El juego de las n torres consiste en colocar sobre un tablero de ajedrez ocho torres sin que estas se amenacen entre ellas. (Figura1b).

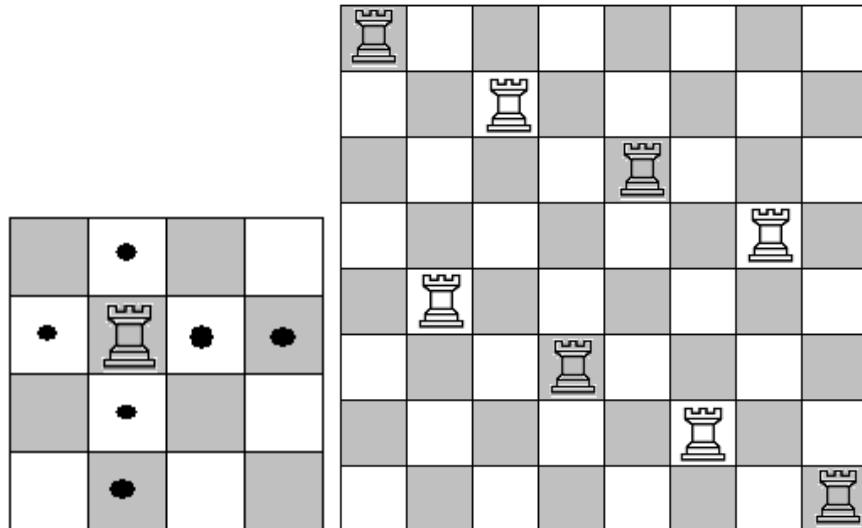


Figura1a: Movimientos posibles de una torre en un tablero de 4x4.

Figura 1b: Una posible solución en un tablero de 8x8

¹ http://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas

Planteamiento del Problema

Como cada torre puede amenazar a todas las torres que estén en la misma fila, cada una ha de situarse en una fila diferente.

Como generar una solución

Podemos generar un vector, donde índice representa una fila y el valor una columna.

Por ejemplo, el vector **(2,3,4,2)** significa que:

La torre1 esta en la columna2

La torre2 esta en la columna3

La torre3 esta en la columna4

La torre4 esta en la columna2

Por tanto el vector “solución” para $n = 4$, correspondería a una permutación de los 4 primeros números enteros.

Como se puede apreciar esta solución es incorrecta (ver Figura2) ya que estarían la torre1 y la 4 en la misma columna.

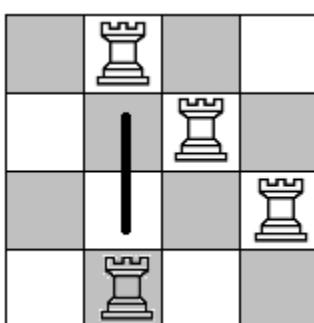


Figura2: Ejemplo de dos torres amenazadas

a) Resolver el problema de las n torres:

Implementar un algoritmo de fuerza bruta iterativo que devuelva todas soluciones.

Comentar hasta qué valor de n se pudo ejecutar de manera que el problema tarde menos de 10 minutos.

b) Calcular el **orden asintótico** exacto de 1a) .

Soluciones

1) Solución Iterativa

Esta solución se basa en generar una tabla con todas la combinaciones de números de 1 a n

Ejemplo1: n = 2

Periodicidad		Es solución (Cantidad)
2	1	
Columna1	Columna2	
1	1	
1	2	Si
2	1	Si
2	2	
Tamaño $2^2 = 4$ filas		(2! = 2)

Tabla1

Como se puede observar en Tabla1 y Tabla2, la cantidad de combinaciones de n números diferentes en una lista de longitud n resulta de combinar todos los números en todas las posiciones.

En total hay n^n combinaciones posibles.

En el caso de no poder repartir el mismo número, como en el caso de las torres o si queremos saber la cantidad de formas de sentar n personas en n sillas (la persona no se puede repetir en mas de una silla) resulta de permutar los números.

En total hay $n!$ combinaciones posibles.

Ejemplo2: n =3

Periodicidad			Es solución (cantidad)
9	3	1	
Columna1	Columna2	Columna3	
1	1	1	
1	1	2	
1	1	3	
1	2	1	
1	2	2	
1	2	3	si
1	3	1	
1	3	2	si
1	3	3	
2	1	1	
2	1	2	
2	1	3	si
2	2	1	
2	2	2	
2	2	3	
2	3	1	si
2	3	2	
2	3	3	
3	1	1	
3	1	2	si
3	1	3	
3	2	1	si
3	2	2	
3	2	3	
3	3	1	
3	3	2	
3	3	3	
Tamaño $3^3 = 27$ filas			$(3! = 6)$

Tabla2

2) Solución recursiva

Esta idea resulta más intuitiva ya que el algoritmo refleja las n^n llamadas recursivas

La complejidad está en $O(n * n^n)$ porque cuando el algoritmo termina se realizaron n^n llamadas recursivas de funciones, que por separadas son $O(n)$

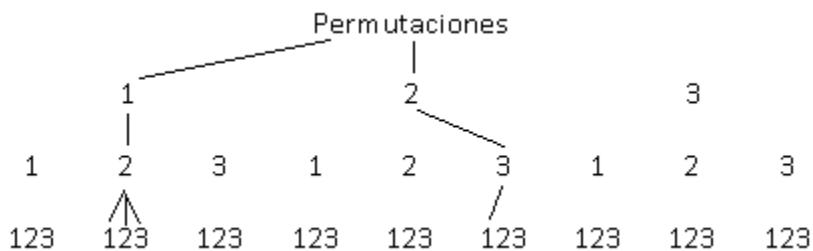
Nota: $n * n^n = n^{n+1}$

3) Solución recursiva con Back Traking²

La idea del Back Traking(o vuelta atrás) es aún más intuitiva.

El objetivo es no mirar los caminos que –se saben- no van a generar ninguna solución.

Una vez que se recortan dichas llamadas, las llamadas restantes coinciden con el Árbol de permutaciones:



Árbol de permutaciones: Se marcan las permutaciones (1,2,1); (1,2,2); (1,2,3); (2,3,1)

La complejidad está en $O(n * n!)$ porque cuando el algoritmo termina se realizaron $n!$ llamadas recursivas de funciones, que por separadas son $O(n)$

Nota: $n * n! = (n+1)!$

² http://es.wikipedia.org/wiki/Vuelta_atr%C3%A1s

Código Fuente

```
public class NTorres {

    private int n;
    private int [][] soluciones;
    private int nn; // todas las combinaciones posibles con n
    private int acum = 0;
    private int cont = 1;
    private int cantLlamadas = 0;

    public NTorres(int n){ //invariante: n > 0
        this.n = n;
        this.nn = (int)Math.pow(n, n);
        soluciones =new int [nn][n];
    }

    public void iterativo(){

        int i;
        int nn = (int)Math.pow(n, n);

        int indice = 0;

        while (indice<n){

            for (i=0;i<nn;i++){
                soluciones [i][indice] = siguiente(indice,i);
            }

            indice++;
            acum = 0;
            cont = 1;

        }

        for (i=0;i<nn;i++){
            if (esSolucion(soluciones[i],n)){
                imprimir(soluciones[i]);
            }
        }
    }

}
```

```
public void recursivo(boolean bk){
    int [] v = new int [n];

    cantLlamadas= 0;

    if (bk){
        recursivoBK(v,0);
    }else{
        recursivoFB(v,0);
    }

    System.out.println("cantLlamadas:"+cantLlamadas);
}

public void recursivoBK(int [] v, int i){
    int j;

    if (esSolucion(v,n)){
        imprimir(v);

    }else{
        for (j=0;j<n;j++){
            v[i]=j+1;
            if (esSolucion(v,i+1)){
                recursivoBK(v,i+1);
                cantLlamadas++;
            }
        }
    }
}

public void recursivoFB(int [] v, int i){
    int j;

    if (i==n){
        if (esSolucion(v,n)){
            imprimir(v);
        }
    }else{
        for (j=0;j<n;j++){
            v[i]=j+1;
            recursivoFB(v,i+1);
            cantLlamadas++;
        }
    }
}
```

```
private void imprimir(int [] v1){
    for (int j=0;j<n;j++){
        System.out.print( v1[j] ) ;
    }System.out.println("");
}

public boolean esSolucion(int [] v1, int hasta){
    int [] v2 = new int [n];
    int ret=0;
    int j;

    for (j=0;j<n;j++){
        if (v1[j]-1>=0){
            v2[v1[j]-1]++;
        }
    }

    for (j=0;j<n;j++){
        if (v2[j]>0){
            ret++;
        }
    }
    return ret>=hasta;
}

private int siguiente( int indice, int i){
    int paso;
    paso = (int)(nn / Math.pow(n, indice+1));
    if (acum < paso){
        acum++;
    }else{
        acum = 1;
        cont++;
    }

    if (cont > n){
        cont = 1;
    }
}
return cont;
}

public String toString(){
    String ret = "";
    int nn = (int)Math.pow(n, n);
    for (int j=0;j<nn;j++){
        for (int i=0;i<n;i++){
            ret = ret + soluciones[j][i] ;
        }ret = ret + "\n";
    }
    return ret;
}
}
```

Programación II

Práctica 03: Recursividad

Versión del 10/08/2014

Como vimos en clase un **algoritmo recursivo** es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva.

Veamos el concepto con un ejercicio ejemplo:

Dados dos números enteros n y m , construir una función recursiva que devuelva el producto de ambos, calculando el mismo con sumas sucesivas. Esto es $n*m=n+n+n+n+...+n$, m veces.

La solución recursiva sería:

```
int producto(int multiplicando, int multiplicador ){
    if (multiplicador==1)
        return multiplicando; // caso base
    else
        return multiplicando + producto(multiplicando, multiplicador-1);
}
```

Para un ejemplo de `multiplicando = 10, multiplicador = 3`, las llamadas serían:

`producto(10,3) =`

`10 + producto(10,2) =`

`10 + 10 + producto(10,1) = // producto(10,1) entra en el caso base`

`10 + 10 + 10 =`

¿Cómo sería este problema resuelto de manera iterativa?

```
int producto(int multiplicando, int multiplicador){
    int ret = 0;
    for (int x=0; x<multiplicador; x++){
        ret = ret + multiplicando;
    }
    return ret;
}
```

Ejercicio1 Resolver mediante algoritmos recursivos los siguientes ejercicios

- 1) Sumar los elementos de un array de enteros. Recordar que la suma es asociativa!
- 2) Buscar el mínimo elemento en un array de enteros.
- 3) Determinar si una matriz de enteros es simétrica.

Las matrices simétricas son cuadradas y tienen ese nombre debido a que presentan simetría respecto de su diagonal principal.

Una matriz $A = (a_{ij})$ es simétrica si cumple que $a_{ij} = a_{ji}$ para $i = 1..n$ y $j = 1..n$

Ejemplos: $A = \begin{bmatrix} 1 & 4 \\ 4 & 3 \end{bmatrix}$ es simétrica, $B = \begin{bmatrix} 1 & 4 & 6 \\ 4 & 3 & 8 \\ 6 & 8 & 7 \end{bmatrix}$ es simétrica

Observación: utilizar una función igualFC que dada una matriz y un índice devuelva verdadero si los valores de la fila indicada por índice y de la columna indicada por índice son iguales uno a uno. Para esto use acumuladores booleanos.

- 4) Un algoritmo que pase de base 10 a base 2

Ayuda: construir primero la versión iterativa.

- 5) Búsqueda binaria dentro de un vector

Ayuda: Pasar las posiciones *desde hasta* de la búsqueda como parámetros, además de los parámetros necesarios para la búsqueda.

- 6) Implementar para el TAD PilaBasica de enteros, dos métodos, uno que devuelva la suma de los elementos de la pila y otro que cambie el signo de todos los elementos de una pila.

```
public int suma() {....}  
  
public void cambioSigno() {....}
```

Ejercicio2 Búsqueda en vectores

- a) Calcular la complejidad de buscar un elemento en un vector desordenado
- b) Para el ejercicio b, utilizar las formulas propuestas en la práctica.
Calcular la complejidad de buscar un elemento en un vector ordenado

Ayuda: Utilizar la búsqueda binaria. Utilizar el método de desenrollar para conjeturar el Orden.

Ejercicio3 Fibonacci

```
int Fib(int n){  
    if (n <= 1)  
        return 1;  
    else  
        return Fib(n - 1) + Fib(n - 2);  
}
```

- a) Conjeturar su orden de complejidad aproximado.

Ayuda: Anotar en una tabla, cuanto tarda Fib para n = 1, n = 2, n = 3 y n = 4; y así conjeturar el O(Fib).

- b) Escribir una versión iterativa, con un orden de complejidad mejor.

Opcional b) ¿Hay implementaciones de Fibonacci mejores que lineales?
http://es.wikipedia.org/wiki/Suces%C3%ADn_de_Fibonacci

Ejercicio4

Cinco personas van al cine, de cuantas maneras diferentes se pueden sentar si están todas juntas?

Escribir un algoritmo que imprima todas las maneras de sentarse.

Ejercicio5

La función de Ackermann es una función recursiva encontrada por [Wilhelm Ackermann](#) en 1926, una función matemática, con un crecimiento extremadamente rápido.

$$A(m, n) = \begin{cases} n + 1, & \text{si } m = 0; \\ A(m - 1, 1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m - 1, A(m, n - 1)), & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

Implemente el algoritmo de la función de Ackermann, utilice long en lugar de int.

Probar que: ackermann(3,1) da 13, ackermann(3,2) da 29, ackermann(3,3) da 61, ackerman(3,4) da 125, ackerman(3,5) da 253, ackerman(3,6) da 509, ackerman(3,7) da 1021, ackerman(3,8) da 2045, ackerman(3,9) da `java.lang.StackOverflowError`.

Ayuda para codificar funciones parciales:

La regla: $A(m-1, 1)$ si $m > 0$ y $n = 0$

Se puede escribir como

If ($m > 0$ && $n == 0$)
 $A(m-1, 1)$

Ejercicio6 Quick Sort

Quick Sort funciona particionando el vector que va a ser ordenado [Cormen \[1990\]](#). Luego, de manera recursiva ordena cada partición. En Partition (Figura 1), uno de los elementos del vector es elegido como pivote. Los valores menores al pivote son colocados a la izquierda de él, mientras que los mayores son colocados a la derecha.

```

int function Partition (Array A, int Lb, int Ub);
begin
  select a pivot from A[Lb]...A[Ub];
  reorder A[Lb]...A[Ub] such that:
    all values to the left of the pivot are <= pivot
    all values to the right of the pivot are >= pivot
  return pivot position;
end;

procedure QuickSort (Array A, int Lb, int Ub);
begin
  if Lb < Ub then
    M = Partition (A, Lb, Ub);
    QuickSort (A, Lb, M - 1);
    QuickSort (A, M + 1, Ub);
  end;

```

FIGURA 1: QUICKSORT

En la Figura 2a el pivote seleccionado es 3. Luego se intercambian los numeros, de manera de tener los menores a 3 en la parte izquierda, y los mayores en la parte derecha (Figura 2b). Por ultimo, se ejecuta Quick Sort recursivamente hasta llegar a la Figura 2c.

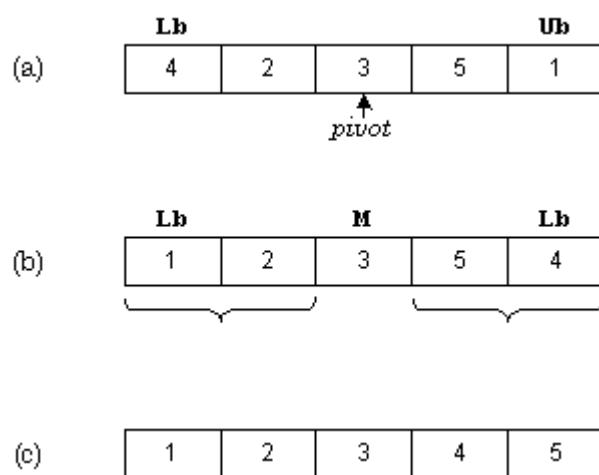


FIGURA 2: EJEMPLO

Para calcular el caso promedio, asumir que el pivote en general estará en el medio. Para calcular el peor caso, asumir que el pivote en general estará en la punta.

Tipos de Recursividad

Pueden distinguirse distintos tipos de llamada recursivas dependiendo del número de funciones involucradas y de cómo se genera el valor final. A continuación veremos cuáles son.

RECUSIÓN LINEAL

En la recursión lineal cada llamada recursiva genera, como mucho, otra llamada recursiva.

RECUSIÓN NO LINEAL

Alguna llamada recursiva puede generar más de una llamada a la función. En este caso estamos hablando de una recursión no lineal.

Un ejemplo de algoritmo de este tipo es el algoritmo Merge-Sort u Ordenamiento por mezcla.

MergeS-Sort utiliza la técnica de "**divide y conquistarás**", que consiste en resolver instancias más chicas del problema y luego componerlas para formar una instancia más grande.

Fue desarrollado en 1945 por John Von Neumann. Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

1. Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso:
2. Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
4. Mezclar las dos sublistas en una sola lista ordenada.

El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

1. Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
2. Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.

A continuación se describe el algoritmo en Java:

```
private static int inputCopy[];
private static int input[] = {10, 6, 4, 2, 8, 0, 14, 12};

private static void mergeSort(int start, int end) {
    int mid = (start + end) / 2;
    if(start < end) {
        /** DIVIDE: Tomar la 1er mitad*/
        mergeSort(start, mid);
        /** DIVIDE: Tomar la 2da mitad*/
        mergeSort(mid+1, end);
        /** CONQUISTAR: Rearmar(merge) el arreglo*/
        merge(start, mid, end);
    }
}

private static void merge(int start, int mid, int end) {
    //Buscamos el comienzo de cada arreglo
    int firstArrStart = start, secondArrStart = mid + 1;
    //Copiamos a una estructura auxiliar antes del merge
    for(int i = start ; i <= end ; i++)
        inputCopy[i] = input[i];

    while(secondArrStart <= end && firstArrStart <= mid)
        if(inputCopy[firstArrStart] >= inputCopy[secondArrStart])
            input[start++] = inputCopy[secondArrStart++];
        else
            input[start++] = inputCopy[firstArrStart++];

    while(firstArrStart <= mid)
        input[start++] = inputCopy[firstArrStart++];

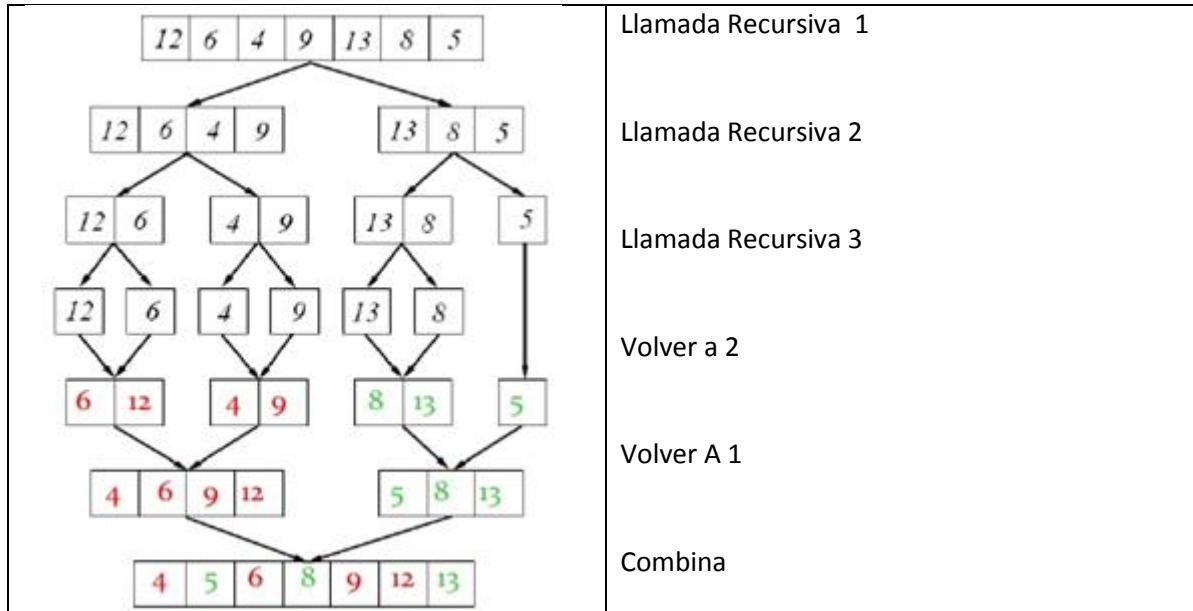
    while(secondArrStart <= end)
        input[start++] = inputCopy[secondArrStart++];
}
}
```

```

public static void main(String[] args) {
    System.out.println("INPUT Unsorted : " + Arrays.toString(input));
    inputCopy = new int[input.length];
    mergeSort(0, input.length-1);
    System.out.println("OUTPUT Sorted   : " + Arrays.toString(input));
}

```

Ejemplo de Ordenamiento por Mezcla



Referencias:

<http://geeksonjava.com/algorithms/sort/merge-sort.php>

<http://informatica.utedm.cl/~mcast/PROGRAMACION/20101/recursividad/FP.RP04.pdf>

<http://www.lcc.uma.es/~pepeg/modula/temas/tema11.pdf>

Programación II

Arboles Binarios(AB)

Definición

Un árbol consta de un conjunto finito de elementos, denominados **nodos**, y un conjunto finito de líneas dirigidas, denominadas **enlaces**, que conectan los nodos.

Hay tres tipos de nodos:

- **Nodo Raíz**: El único nodo que no tiene padre
- **Nodo Hoja**: Los nodos que no tiene hijos
- **Nodos Internos**: Los nodos que no son hojas

Intuitivamente el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están organizados entre sí a través de ramas.

En la Figura1 se pueden ver los tres tipos de nodos:

- Raíz: A
- Internos: B,C,D
- Hojas: E,F,G,H,I

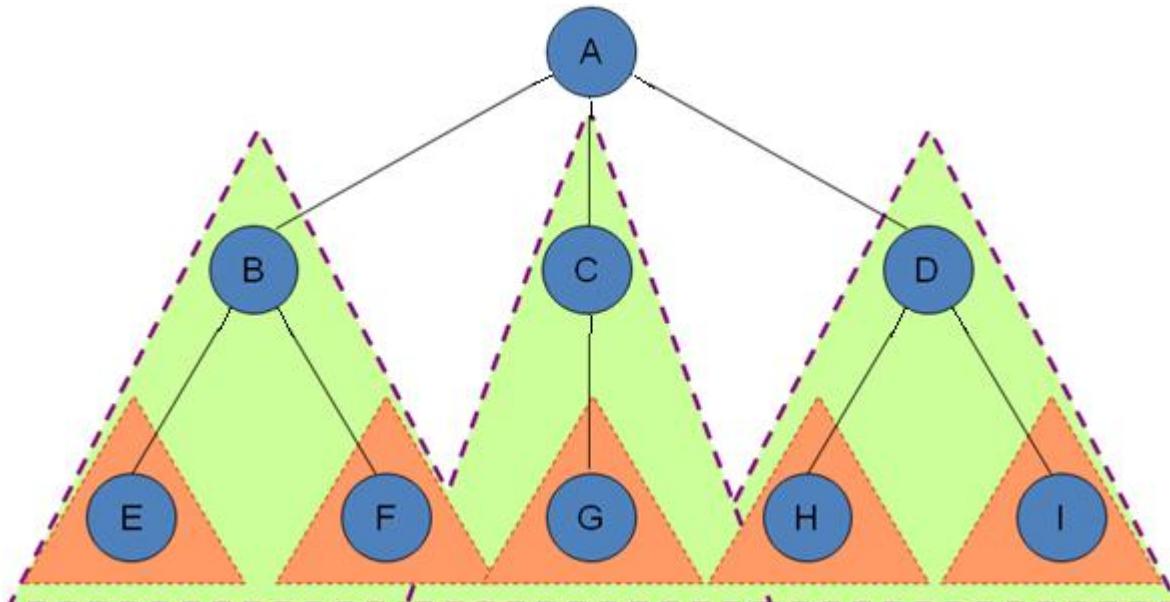


Figura1: Árbol de grado 3, con 9 nodos

La **altura** de un árbol se define como la distancia de la hoja más lejana a la raíz.

La altura del árbol de la Figura1 es 3.

Grado es el número máximo de hijos que tienen los nodos del árbol.

Así, en el ejemplo anterior el árbol es de grado 3

Árbol binario

Definición 1:

Es un árbol de grado 2.

Definición 2:

Un Árbol binario es aquel que:

- *es vacío, ó*
- *está formado por un nodo cuyos subárboles izquierdo y derecho son a su vez árboles binarios.*

El árbol de la Figura1 no es binario por ser de grado 3.

El de la Figura2 es binario por ser grado 2.

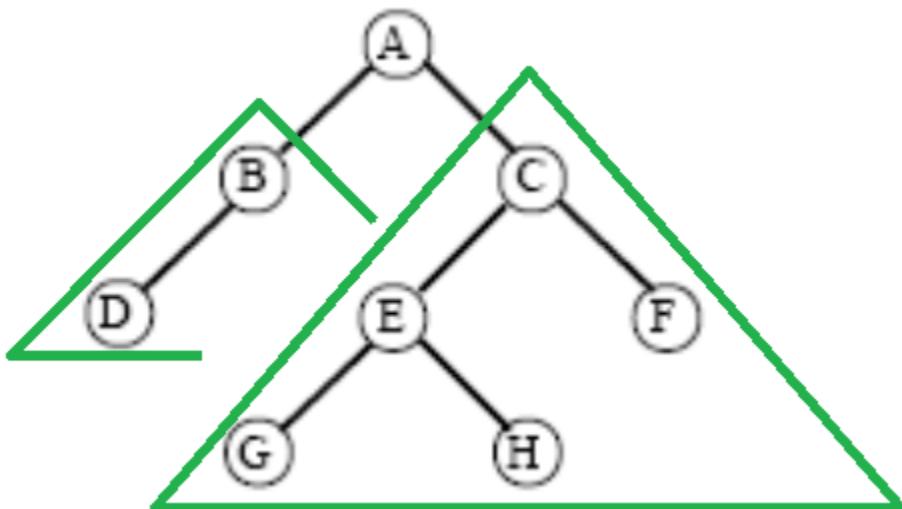


Figura2: Árbol binario y los primeros dos subárboles.

Todo árbol de mas de un nodos tiene al menos un subárbol.

En la Figura2 marcamos en verde los primeros subárboles:

El subárbol izquierdo(A y B)

El subárbol derecho(C,E,F,G,H)

Subárboles:

Por la naturaleza recursiva de los AB, todo subárbol sigue siendo árbol.

La altura de un subárbol es a lo sumo la altura del árbol padre - 1.

Existen algunos tipos especiales de árboles binarios en función de ciertas propiedades. Por ejemplo

Árbol binario balanceado es aquel en el que en todos sus nodos se cumple la siguiente propiedad: La altura(subárbol_izquierdo) - altura(subárbol_derecho) es menor o igual a 1.

En la Figura 3 se muestran dos ejemplos:

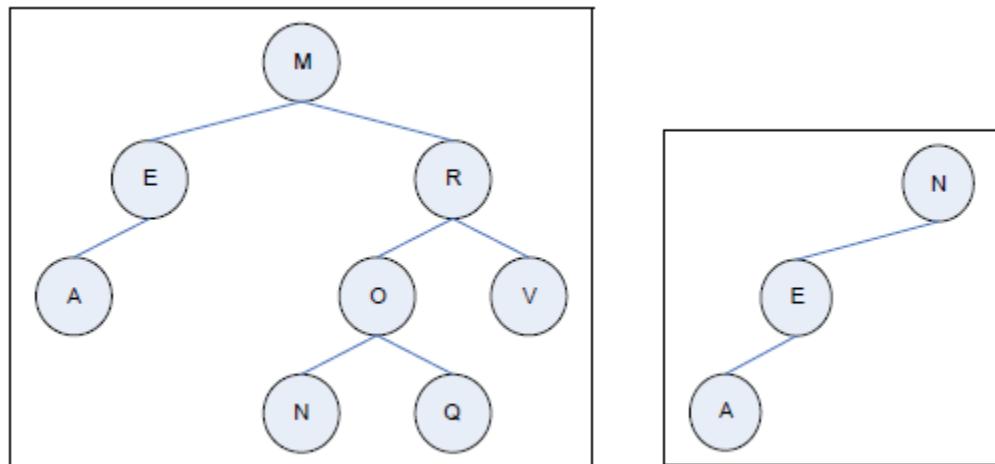


Figura3a: Árbol balanceado. Figura3b: Árbol desbalanceado

Árbol binario completo es aquel en el que todos los nodos tienen dos hijos y todas las hojas están en el mismo nivel. Se denomina completo porque cada nodo, excepto las hojas, tiene el máximo de hijos que puede tener.

Todo árbol binario completo (Ver Figura4) tiene $2^{\text{altura}} - 1$ nodos

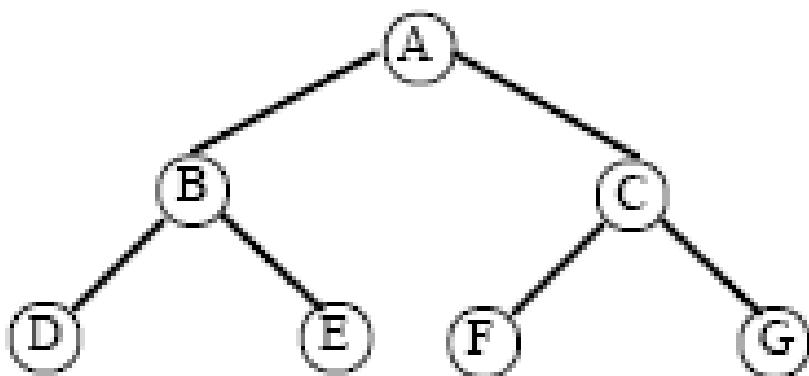


Figura4: Árbol binario completo con $2^3 - 1$ nodos

Implementación de Árbol binario

Un árbol solo necesita un puntero al nodo raíz.

Cada nodo contiene tres valores (Ver Figura5):

- La información
- Un puntero al nodo izquierdo
- Un puntero al nodo derecho

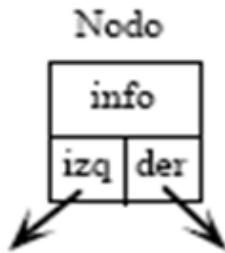


Figura5: Nodo

```
class NodoArbol{
    int info;
    private NodoArbol izq;
    private NodoArbol der;

    public NodoArbol (int info){
        this.info = info;
        izq=der=null;
    }
    void agregar(Nodo n) {}
}

Class Arbol
{
    private NodoArbol raíz;
    //No tenemos constructor!
    public int buscar (int valor) {} //devolvemos -1 como valor default
    public void agregar (int valor){
        NodoArbol nuevo = new NodoArbol(valor)
        if raíz == null raíz =nuevo
        }else{ raíz.agregar(nuevo)}
    }
    public void borrar (int valor) {}
}
```

Invariante de representación

Se pueden llegar a todos los nodos desde la raíz

Cada nodo tiene un solo parente

No tiene ciclos

Recorrido de Árbol binario

Recorrer un árbol consiste en acceder una sola vez a todos sus nodos.

Esta operación es básica en el tratamiento de árboles y permite, por ejemplo, imprimir toda la información almacenada en el árbol

- Imponiendo la restricción de que el subárbol izquierdo se recorre siempre antes que el derecho, esta forma de proceder da lugar a tres tipos de recorrido, que se diferencian por el orden en el que se realizan estos tres pasos.

Preorden: primero se accede a la información del nodo, después al subárbol izquierdo y después al derecho (Ver Figura 6).

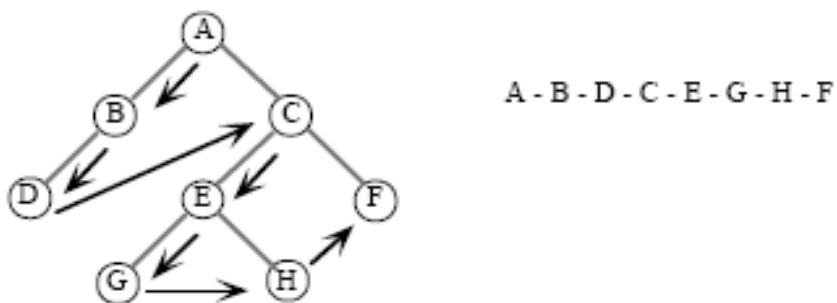


Figura 6: Preorden



Inorden: primero se accede a la información del subárbol izquierdo, después se accede a la información del nodo y, por último, se accede a la información del subárbol derecho (Ver Figura7).

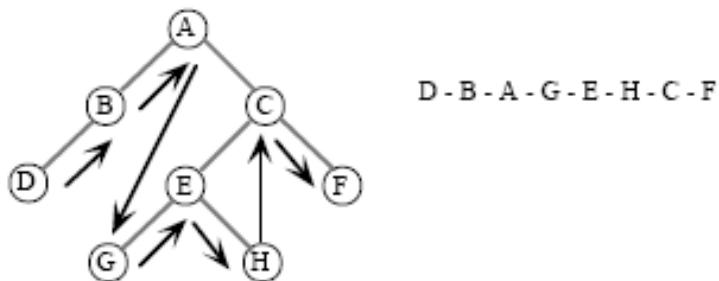


Figura7: Inorden

Postorden: primero se accede a la información del subárbol izquierdo, después a la del subárbol derecho y, por último, se accede a la información del nodo (Ver Figura8).

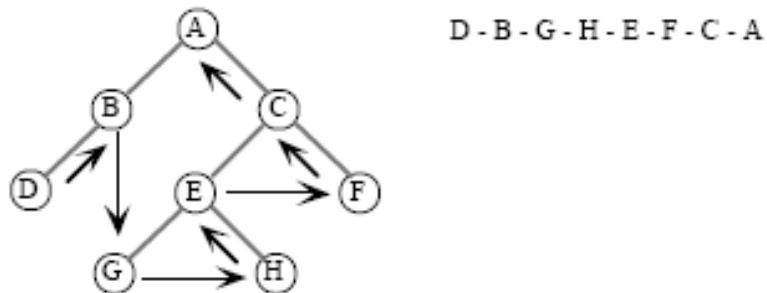


Figura8: Postorden

Si agregamos los nodos más chicos a la izquierda y los más grandes a la derecha;

Y recorremos un árbol *Inorden*, obtenemos los nodos de manera ordenada (Ver Figura9):

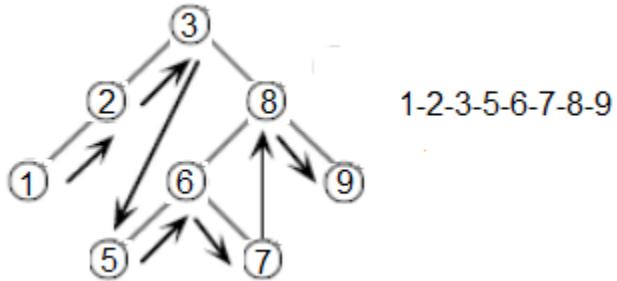


Figura9: Nodos ordenados

Ejercicio1:

Cual será el orden de complejidad de insertar un nodo para el peor caso si:

- a) El árbol no está completo
- b) El árbol está completo

Ejercicio2:

Cual será el orden de complejidad de buscar un nodo para el peor caso si:

- a) El árbol no está completo
- b) El árbol está completo

Programación II

Árboles binarios de búsqueda (ABB)

Definición

Un árbol binario de búsqueda(ABB) a es una estructura de datos de tipo árbol binario en el que para todos sus nodos, el hijo izquierdo, si existe, contiene un valor menor que el nodo padre y el hijo derecho, si existe, contiene un valor mayor que el del nodo padre.
También vale la propiedad para a.izq y a.der.

Es de búsqueda porque:

Los nodos están ordenados de manera conveniente para la búsqueda (Ver Figura1)

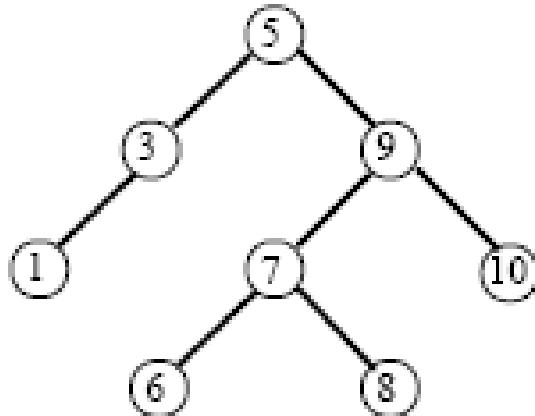


Figura1: Ejemplo de un ABB

Invariante de representación

ABB(a) \Leftrightarrow

a es un AB tal que

-Todos los nodos de a.izq son menores que a.info

-Todos los nodos de a.der son mayores que a.info

-ABB(a.izq)

-ABB(a.der)

**Búsqueda**

1. Si el valor del nodo actual es igual al valor buscado, lo hemos encontrado.
2. Si el valor buscado es menor que el del nodo actual, deberemos inspeccionar el subárbol izquierdo.
3. Si el valor buscado es mayor que el del nodo actual, deberemos inspeccionar el subárbol derecho

Para continuar la búsqueda en el subárbol adecuado se aplica el mismo razonamiento.

Por lo tanto, el método Buscar () en versión iterativa podría ser el siguiente:

```
public Nodo buscar (Nodo raíz, int dato){  
    Nodo actual = raíz;  
    while (actual.dato != dato) {  
        if (dato < actual.dato)  
            actual = actual.izq;  
        else  
            actual = actual.der;  
  
        if (actual == null)      return null;  
    }  
    return actual;  
}
```

Inserción

La operación de inserción de un nuevo nodo en un árbol binario de búsqueda consta de tres fases básicas:

1. Creación del nuevo nodo
2. Búsqueda de su posición correspondiente en el árbol. Se trata de encontrar la Posición que le corresponde para que el árbol resultante siga siendo de búsqueda.
3. Inserción en la posición encontrada. Se modifican de modo adecuado los enlaces de la estructura

La versión recursiva es muy similar al buscar. Una vez que se llega a una hoja, se inserta el elemento ahí:

Arbol:

```
public void insertar(int info) {  
    NodoABBint nuevo = new NodoABBint(info);  
    if (raiz == null){  
        raiz = nuevo;  
    }else{  
        raiz.insertar(nuevo);  
    }  
}
```

Nodo:

```
public void insertar(NodoABBint nuevo) {  
    if (info > nuevo.info){  
        if (izq == null)  
            izq = nuevo;  
        else  
            izq.insertar(nuevo);  
    }else{  
        if (der == null)  
            der = nuevo;  
        else  
            der.insertar(nuevo);  
    }  
}  
  
public String toString(){ // inOrder  
    String ret ="";  
    if (izq != null) ret = ret + izq.toString();  
    ret =ret + info + " ";  
    if (der != null) ret = ret + der.toString();  
    return ret;  
}
```

Una versión iterativa podría ser la siguiente:

```
Public void insertar (int dato){  
    NodoArbo nuevonodo= new NodoÁrbol(dato);  
    if (this == null)      //raiz  
        this = nuevonodo;  
    else {  
        NodoÁrbol actual = raíz;  
        NodoÁrbol p;  
  
        while(true)  {  
            p= actual;  
            if (dato < actual.datos) {//ir a la izquierda  
                actual = actual.izq;  
                if (actual == null ){  
                    p.izq = nuevonodo;  
                    return; }  
            }else {           //ir a la derecha  
                actual = actual.der;  
                if (actual == null) {  
                    p.der = nuevonodo;  
                    return; }  
            }  
        }  
    }  
}
```

A continuación se muestra como queda el Árbol luego de insertar cada nodo:

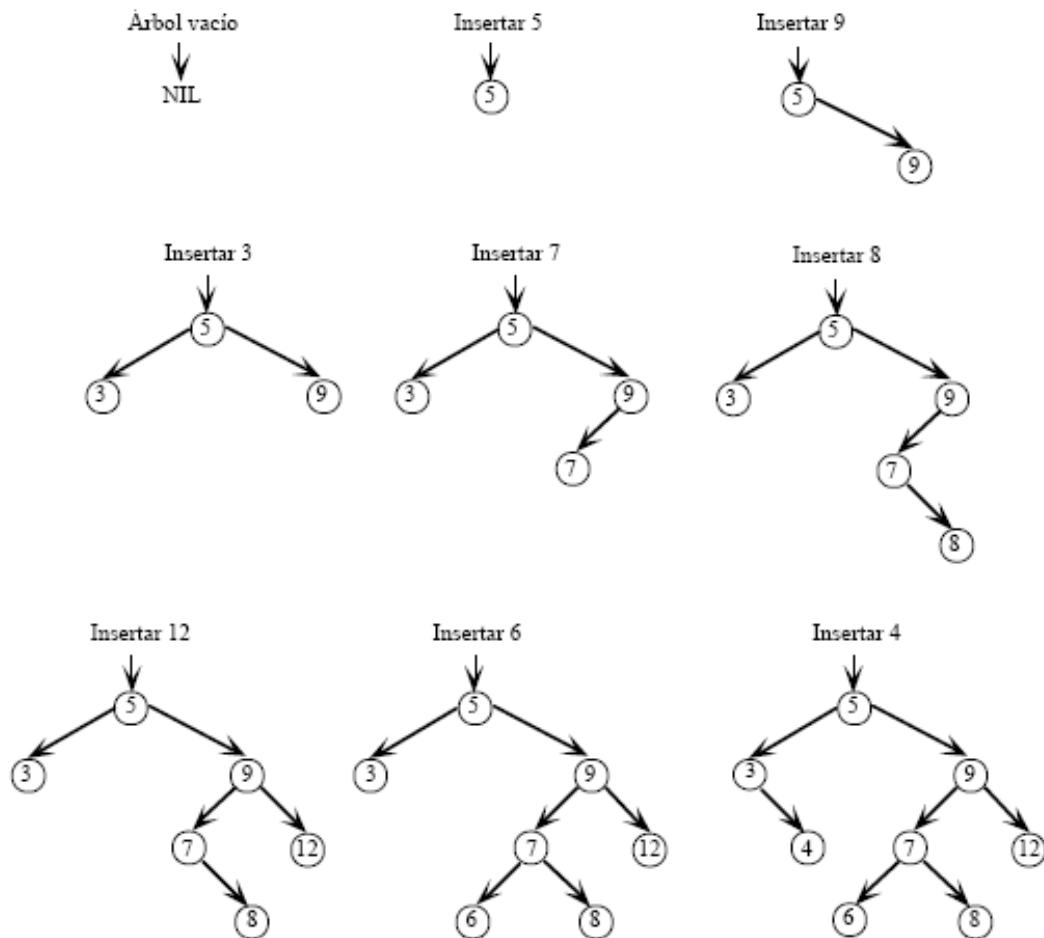


Figura2: Inserción en un ABB de 8 nodos.

Eliminar

Existen cuatro distintos escenarios:

1. Intentar eliminar un nodo que no existe.
 - No se hace nada, simplemente se regresa FALSE.
2. Eliminar un nodo hoja.
 - Caso sencillo; se borra el nodo y se actualiza el apuntador del nodo padre a NULL.
3. Eliminar un nodo con un solo hijo.
 - Caso sencillo; el nodo hijo se convierte en el padre.
4. Eliminar un nodo con dos hijos.
 - Caso complejo, es necesario mover más de una referencia.
 - Se busca el máximo de la rama izquierda o **el mínimo de la rama derecha**.

Eliminar, Caso2 y caso3 (Figura3a y Figura3b):

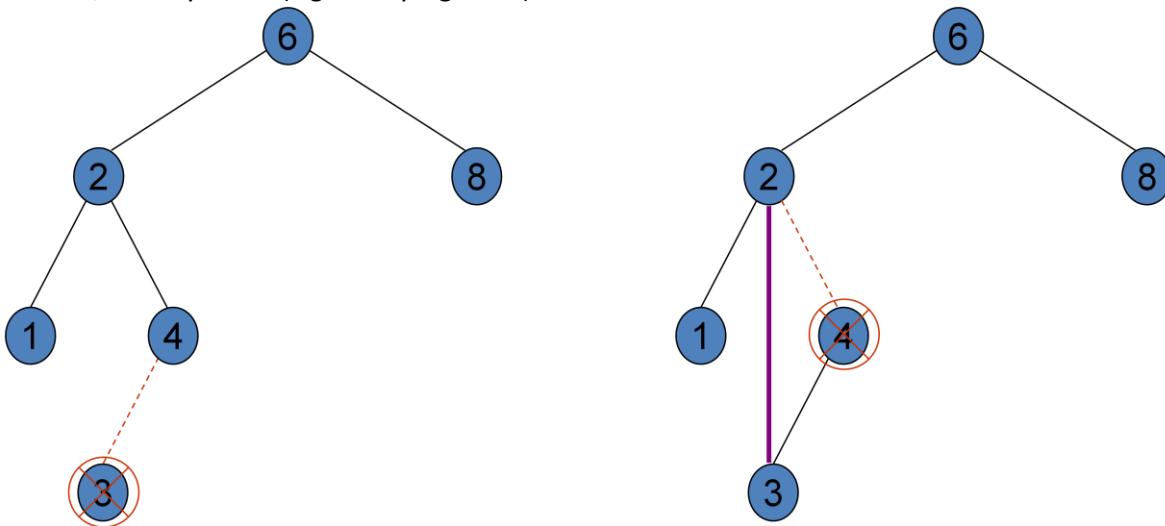


Figura3a: Eliminar Nodo hoja y Figura3b: Eliminar nodo con un hijo

Eliminar, Caso4 (Figura4):

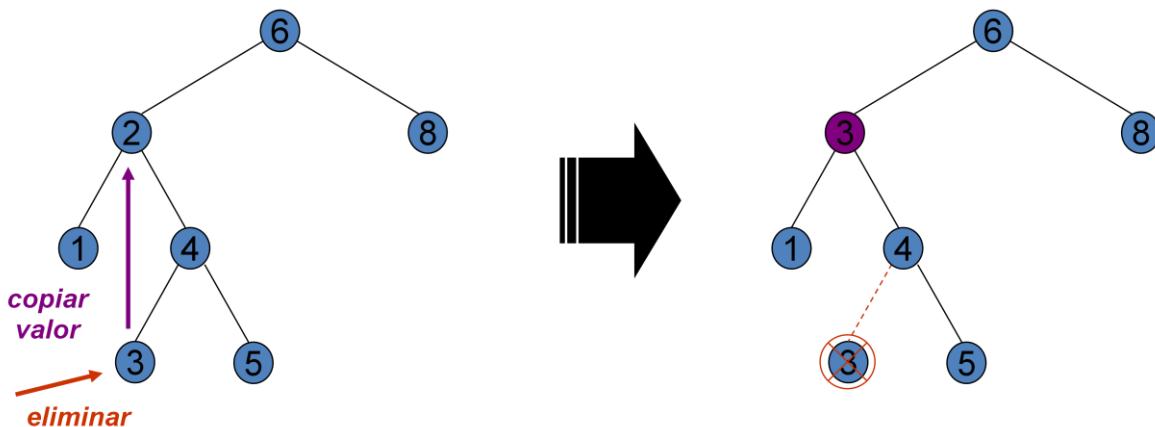


Figura 4: Eliminar nodo con dos hijos (Ejemplo1)

Remplazar el dato del nodo que se desea eliminar con el dato del nodo más pequeño del subárbol derecho

Después, eliminar el nodo más pequeño del subárbol derecho (caso fácil)

Eliminar, Caso4 (Figura5):

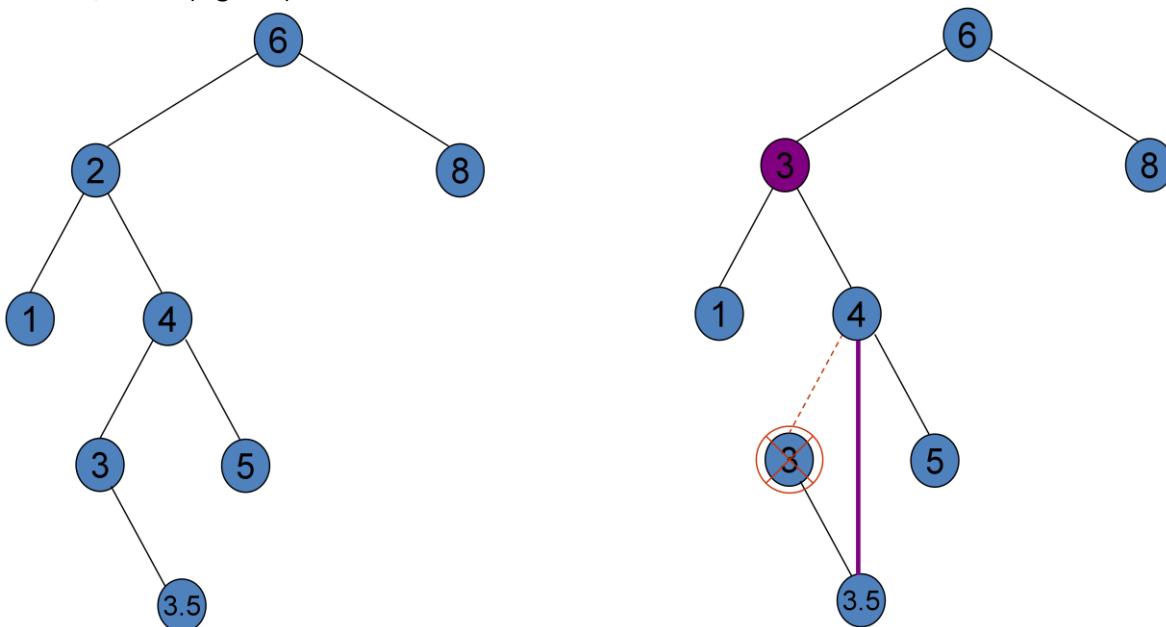


Figura5: (Ejemplo2)

Varios ejemplos de eliminación de un nodo (Figura6):

- a. Si el nodo a borrar no tiene hijos, simplemente se libera el espacio que ocupa
- b. Si el nodo a borrar tiene un solo hijo, se añade como hijo de su padre (p), sustituyendo la posición ocupada por el nodo borrado.
- c. Si el nodo a borrar tiene los dos hijos se siguen los siguientes pasos:
 - i. Se busca el máximo de la rama izquierda o **el mínimo de la rama derecha**.
 - ii. Se sustituye el nodo a borrar por el nodo encontrado.

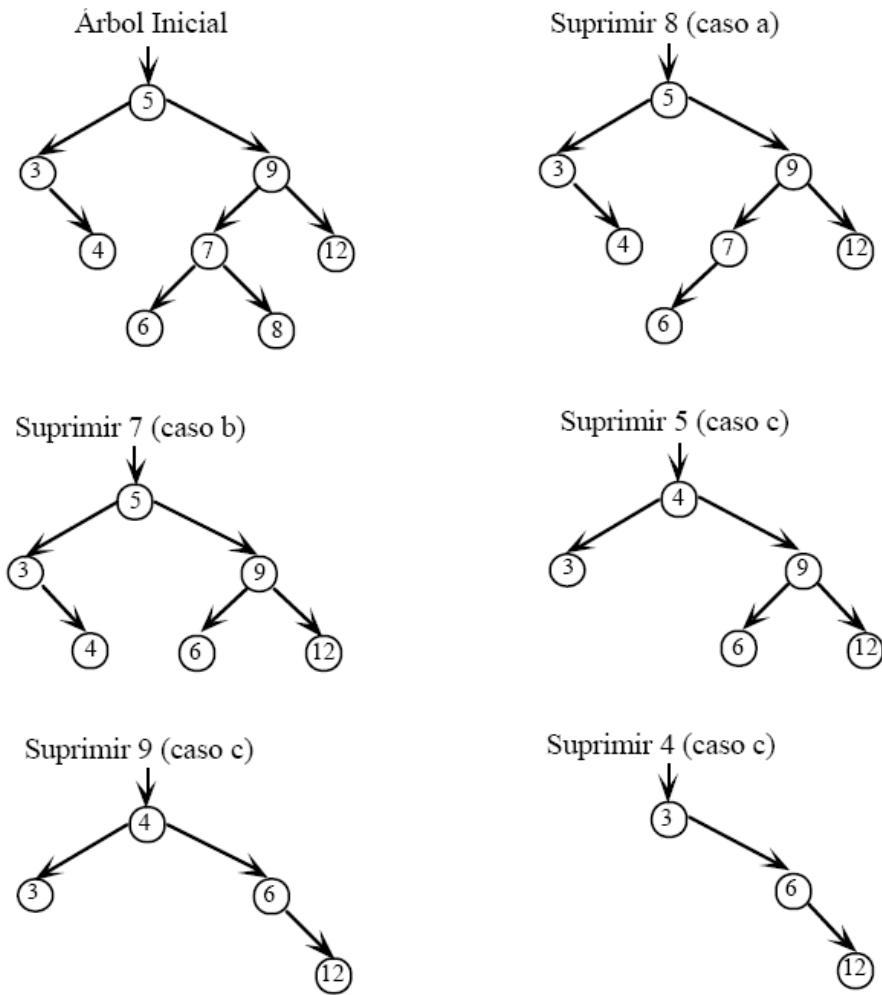


Figura6: Representación grafica

Ejercicio1:

Cual será el orden de complejidad de insertar un nodo para el peor caso sí:

- a) El Árbol no está balanceado
- b) El Árbol esta balanceado

Ejercicio2:

Cual será el orden de complejidad de buscar un nodo para el peor caso sí:

- a) El Árbol no está balanceado
- b) El Árbol esta balanceado

¿Cual es la mejora respecto del AB?

Bibliografía

Luis Joyanes Aguilar ; Ignacio Zahonero Martínez. Programación en Java 2 : Algoritmos, estructuras de datos y programación orientada a objetos. McGraw-Hill, 2002, ISBN 84-481-3290-4

Sara Baase, Allen Van Gelder. Algoritmos Computacionales: Introducción al análisis y diseño. Addison Wesley, 2002, ISBN 9702601428

Programación II

Práctica 04: Arboles

Versión del 01/04/2014

Ejercicio1: Arbol Binario de enteros (ABint)

A medida que realizamos TADs mas complejos, es necesario definir TADs auxiliares como soporte del TAD que se quiere definir.

Vamos a especificar el árbol binario especificado en la clase teórica.

EspecificaciónTAD ABint

```
ABint() {} // Constructor1
void agregar(Integer i) {}
Integer buscar(Integer i) {} //devuelve 0 si no encuentra
```

EspecificaciónTAD ABNodo

```
ABNodo(Integer i){} // Constructor1
void agregar(Integer i) {}
Integer buscar(Integer i) {} //devuelve 0 si no encuentra
```

- a) Implementar el TAD ABint
- b) Cual es el orden de complejidad de agregar()?

Ejercicio2: Arbol Binario de búsqueda de enteros (ABBint)

a) Implementar ABBint según la especificación vista en la clase teórica

Utilizar ABNodo.

EspecificaciónTAD ABBint

```
ABBint() {} // Constructor1
void agregar(Integer i) {}
Integer buscar(Integer i) {} //devuelve 0 si no encuentra
```

b) Cual es el orden de complejidad de agregar() y de buscar () ?

c) Si el ABBint esta **balanceado**, cual es el orden de complejidad de agregar() y de buscar () ?

Ayuda: Para la justificación del calculo del orden para el ítem c, utilizar los apuntes de las clases teóricas (o de la bibliografía [Cormen2001]).

Ejercicio3

Implementar los siguientes métodos en el TAD AB

```
Int cantidadNodos()
Int altura()
boolean invariante()
```

Que dado una instancia de AB, devuelve verdadero si se cumple el invariante de AB.

- Que el árbol no tenga ciclos
- Que todos los nodos sean alcanzables desde la raíz

Ayuda:

Considerar agregar variables y estructuras auxiliares para “contar” los nodos.
Cada vez que se agregue un nodo, dicho nodo debe ser considerado.

```
boolean balanceado()
```

Ejercicio4

Implementar los siguientes métodos en el TAD ABB

boolean invariante() // reutilizar AB.invariante

-Que además de ser un AB, tiene que estar ordenado.

void balancear()

ABB balancear() // Devuelve un nuevo árbol.

Que luego de ejecutar balancear, el método balanceado debe devolver **true**.

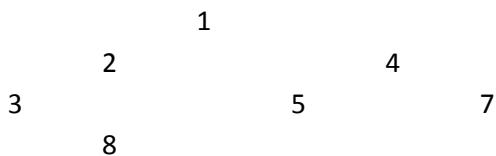
Ejercicio5

Implementar los siguientes métodos en el TAD AB

Void emilinar()

int ramaMasCorta()

Dado una instancia de AB devuelve la longitud de la rama mas corta comenzando desde la raíz y llegando hasta una hoja.



Ejemplo1

En el Ejemplo1 la rama mas corta es

1,4,7 o 1,4,5

Ejercicio6: Árbol-n-ario de búsqueda de enteros (ANBint)

A diferencia del árbol binario, que tiene hasta dos nodos por nivel, el árbol n-ario tiene hasta n nodos por nivel.

Además, el nodo padre tiene que ser menor que todos los nodos hijos.

EspecificaciónTAD ANBint

```
ANBint(Integer n) {} // Constructor: hasta n hojas por nivel
voideliminar(Integer i) {}
voidagregar(Integer i) {}
Integerbuscar(Integer i) {} //devuelve 0 si no encuentra
```

Ejemplo para un árbol de orden 3($n = 3$).

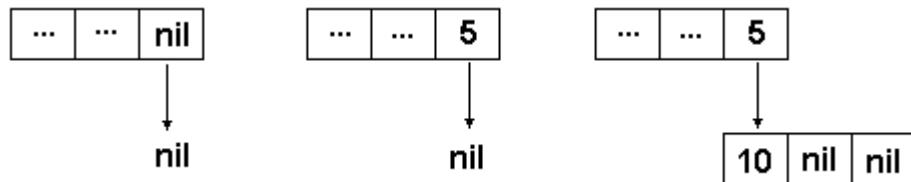


Figura 1

En la Figura1 agregamos el 5, lo cual completa el nodo y genera un nuevo nodo vacío.

Luego agregamos e 10 al comienzo del nuevo nodo.

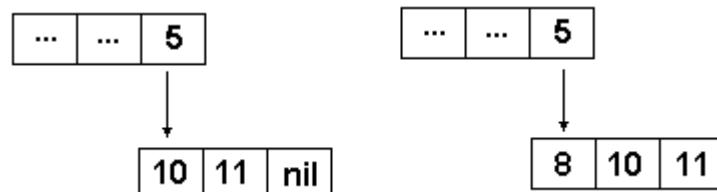


Figura2

En la Figura2 agregamos el 11.

Luego, cuando se agrega el 8, desplazamos el 10 y el 11 para poner el 8 al comienzo.

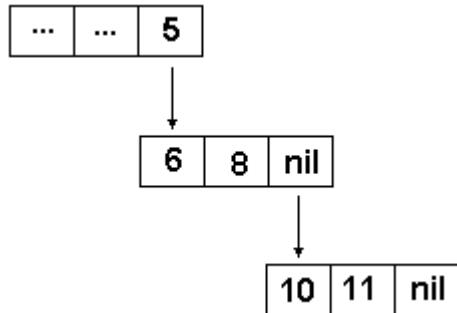


Figura3

En la Figura3 cuando agregamos el 6, tenemos que hacer dos cosas:

- 1) Generar un nuevo nodo
- 2) Repartir 6,8, 10 y 11 entre los dos nodos disponibles.

En general se intenta dejar la mitad de los elementos en cada nodo.

En este caso tenemos cuatro elementos, así que dejaremos dos elementos en cada nodo.

Implementar el TAD ANBint, calcular el orden de las operaciones.

Ayuda: El orden de complejidad tendrá dos variables: La cantidad de nodos y la cantidad de nodos por nivel.

Complejidad en algoritmos recursivos

Hasta ahora vimos como calcular el orden de complejidad en un algoritmo iterativo.

Ahora explicaremos una técnica para calcularlo también en algoritmos recursivos.

¿Cual es la complejidad de anidar1?

```
void anidar1(int tamano){  
    for (int i = 0;i < tamano;i++){}  
}
```

¿Cual es la complejidad de anidar2?

```
void anidar2(int tamano){  
    for (int i = 0;i < tamano;i++){  
        for (int j = 0;j < tamano;j++){}  
    }  
}
```

¿Y si queremos generalizarlo?

```
void anidarGenerico(int tamano,int anidamiento){  
    if (anidamiento > 0){  
        for (int i = 0;i < tamano;i++){  
            anidarGenerico(tamano,anidamiento - 1);  
        }  
    }  
}
```

¿Como calculamos la complejidad de anidarGenerico?

Utilizaremos la técnica de **desenrollar** para ayudarnos a intuirlo.

Desenrollar un algoritmo consiste en reemplazar la llamada recursiva por el código equivalente.

Se deben renombrar las variables de manera que no se repitan.

Esto se realiza para una instancia en particular.



Por ej: `tamaño == 100` y `anidamiento == 2`

Entonces

```
if (anidamiento > 0){  
    for (int i = 0;i < tamano;i++){  
        anidarGenerico(tamano,anidamiento - 1);  
    }  
}
```

Es equivalmente a

```
if (2 > 0){  
    for (int i = 0;i < 100;i++){  
        anidarGenerico(100,2 - 1);  
    }  
}
```

Es equivalmente a

```
if (2 > 0){  
    for (int i = 0;i < 100;i++){  
        //reemplazamos anidarGenerico(100,2 - 1) por:  
        if (1 > 0){  
            for (int i = 0;i < 100;i++){  
                anidarGenerico(100,1 - 1)  
            }  
        }  
    }  
}
```

Equivale a

```
if (2 > 0){  
    for (int i = 0;i < 100;i++){  
        //reemplazamos anidarGenerico(100,2 - 1) por:  
        if (1 > 0){  
            for (int i = 0;i < 100;i++){  
                //reemplazamos anidarGenerico(100,1 - 1) por:  
                if (0 > 0){  
                    ...  
                }  
            }  
        }  
    }  
}
```

Calculamos el orden de complejidad para esta instancia:

$$100 * 100 = 100^2 \quad // \text{tamano}^{\text{anidamiento}}$$

Podemos intuir que el orden es:

$$\prod^{\text{anidamiento}} \text{tamano} \\ = O(\text{tamano}^{\text{anidamiento}})$$

Se puede demostrar por una técnica algebraica llamada Inducción.

Ejercicio1: Búsqueda binaria en un vector ordenado

```
int búsquedaBinaria (int valor, int inicio, int fin )  
    posición = (inicio + fin) / 2  
    if v[posicion] = valor  
        return posición  
    else  
        If v[posicion] > valor  
            return búsquedaBinaria(valor, inicio, posicion)  
        else  
            return búsquedaBinaria(valor, posicion, fin)
```

Queremos probar que el orden de complejidad es $O(\log n)$

Donde n es el tamaño de v .

Lo primero que hay que notar es que las posiciones que se descartan del vector, en cada llamada recursiva son siempre la mitad.

Es decir, nos quedamos siempre con la mitad del arreglo en cualquier caso.

En la 1er llamada consideramos $n/2^1$ posiciones.

En la 2da llamada consideramos $n/2^2 = \frac{1}{4}$ posiciones.

En la 3da llamada consideramos $n/2^3 = 1/8$ posiciones.

...

En general, luego de m llamadas, consideramos $n/2^m$ posiciones del arreglo.

La cantidad de posiciones consideradas converge a 1(en este caso el algoritmo termina) si y solo si

$$n/2^m = 1 \quad \text{si y solo si}$$

$$n = 2^m \quad \text{si y solo si}$$

$$\log n = \log(2^m) \quad \text{si y solo si}$$

$$\log n = m$$

Esto quiere decir, que la cantidad de llamadas es $\log n$

Entonces el algoritmo esta en $O(\log n)$.

Analisis de complejidad de Fibonacci

Agregaremos reglas adicionales a las ya vistas para el análisis de complejidad.

Regla1: Las asignaciones y comparaciones son O(1)

Regla2: El O de un IF es el tiempo de la rama mas larga:

$$O(IF \text{ condición Then } f \text{ Else } g) = O(\max(\text{condición}, f, g))$$

Según las **reglas** las **propiedades** y el **álgebra** de O, evaluaremos el Orden de complejidad de fibonacci iterativo:

```
static int fibonacciIter(int n) {
    int n1 = 1;                                // O(1) Regla1
    int n2 = 1;
    int n3 = n1 + n2;                          // O(1)

    if (n==1) { return n1; }                     // O(1) Regla1 y 2
    if (n==2) { return n2; }

    if (n>2) {
        for (int i=2; i<n; i++) {
            n3 = n2 + n1;                      // 10 * O(1) por ahora
            n1 = n2;                            // O(n) Álgebra
            n2 = n3;                            // O(1) Regla1
        }
    }
    return n3;
}
```

Código 1a: fibonacci iterativo

Vamos a sumar el orden:

$$10 * O(1) + O(n) * 4 * O(1) =$$

$$\times \text{álgebra y prop} \quad O(1) + O(n) =$$

$$\times \text{álgebra y prop} \quad O(n)$$

Ahora vamos a conjeturar el orden de fibonacciRecur

```
static int fibonacciRecur(int n){  
    if (n <= 1)  
        return 1; // O(1)  
    else  
        return fibonacciRecur(n - 1) + fibonacciRecur(n - 2);  
}
```

Código 1b:

Lo primero que notamos, es que no hay reglas ni álgebra definidas para calcular la cantidad de operaciones de las llamadas recursivas.

Por lo tanto utilizaremos la técnica de desenrollaremos el código para conjeturar el O.

Lo primero que haremos será definir una instancia del problema. En este caso n.

Notar que si se elige un n muy chico no se ejecutarán muchos casos recursivos, mientras que si se elige un n muy grande, el desenrollo quedará muy extenso.

Desenrollaremos fibo para n = 4.

Paso1: reemplazar las variables

```
static int fibonacciRecur(int n = 4){  
    if (4 <= 1)  
        return 1;  
    else  
        return  
            fibonacciRecur(4 - 1) +  
            fibonacciRecur(4 - 2);  
}
```

Paso2: simplificamos las expresiones

```
static int fibonacciRecur(int n = 4){  
    if (4 <= 1)  
        return 1;  
    else  
        return  
            fibonacciRecur(3) +  
            fibonacciRecur(2);  
}
```

Es importante notar, que el código del paso2 es equivalente a ejecutar `fibonacciRecur(4)`, pues solo reemplaza las variables por los valores que toman.

Paso3: reemplazamos las llamadas recursivas *fibonacciRecur(3)* y *fibonacciRecur(2)*, por su equivalencia en código.

```
static int fibonacciRecur(int n = 4) {  
    if (4 <= 1)  
        return 1;  
    else  
        return  
            //reemplazamos fibonacciRecur(3) por: Nro1  
            if (3 <= 1)  
                return 1;  
            else  
                return  
                    fibonacciRecur(3 - 1) +  
                    fibonacciRecur(3 - 2);  
    }  
    //reemplazamos fibonacciRecur(2) por: Nro2  
    if (2 <= 1)  
        return 1;  
    else  
        return  
            fibonacciRecur(2 - 1) +  
            fibonacciRecur(2 - 2); // fibonacciRecur(0) !  
    }  
}
```

Nuevamente, ejecutar esto, es equivalente a *fibo(4)*

Como aun quedan llamadas recursivas, aplicamos nuevamente el desenrollo.

Aplicamos el Paso3 para *fibonacciRecur(2)*, *fibonacciRecur(1)* y *fibonacciRecur(0)*:

```
static int fibonacciRecur(int n = 4) {  
    if (4 <= 1)  
        return 1;  
    else  
        return  
            //reemplazamos fibonacciRecur(3) por: Nro3  
            if (3 <= 1)  
                return 1;  
            else  
                return  
                    //reemplazamos fibonacciRecur(2)+ por: Nro4  
                    if (2 <= 1)  
                        return 1;  
                    else  
                        return  
                            fibonacciRecur(2 - 1) +  
                            fibonacciRecur(2 - 2);  
            }  
            //reemplazamos fibonacciRecur(1) por: Nro5  
            if (1 <= 1)  
                return 1;  
            else  
                ...  
        }  
    }  
    //reemplazamos fibonacciRecur(2) por: Nro6  
    if (2 <= 1)  
        return 1;  
    else  
        return  
            //reemplazamos fibonacciRecur(1)+ por: Nro7  
            if (1 <= 1)  
                return 1;  
            else  
                ...  
        }  
        //reemplazamos fibonacciRecur(0) por: Nro8  
        return 1;  
    }  
}
```

Ahora Aplicamos por ultima vez el Paso3 para *fibonacciRecur(1)* y *fibonacciRecur(0)*:

```
static int fibonacciRecur(int n = 4) {
    if (4 <= 1)
        return 1;
    else
        return
            //reemplazamos fibonacciRecur(3) por: Nro1
            if (3 <= 1)
                return 1;
            else
                return
                    //reemplazamos fibonacciRecur(2)+ por: Nro2
                    if (2 <= 1)
                        return 1;
                    else
                        return
                            //reemplazamos fibonacciRecur(1)+ por: Nro3
                            return 1+
                                //reemplazamos fibonacciRecur(0) por: Nro4
                                return 1;

    }
    //reemplazamos fibonacciRecur(1) por: Nro5
    if (1 <= 1)
        return 1;
    else
        ...
    }
    //reemplazamos fibonacciRecur(2) por: Nro6
    if (2 <= 1)
        return 1;
    else
        return
            //reemplazamos fibonacciRecur(1)+ por: Nro7
            if (1 <= 1)
                return 1;
            else
                ...
    }
    //reemplazamos fibonacciRecur(0) por: Nro8
    return 1;
}
```

Ahora, como no tenemos mas llamadas recursivas, es un algoritmo iterativo (para fibo(4)).

A continuación se muestra un árbol que representa las llamadas recursivas para fibo(5) :

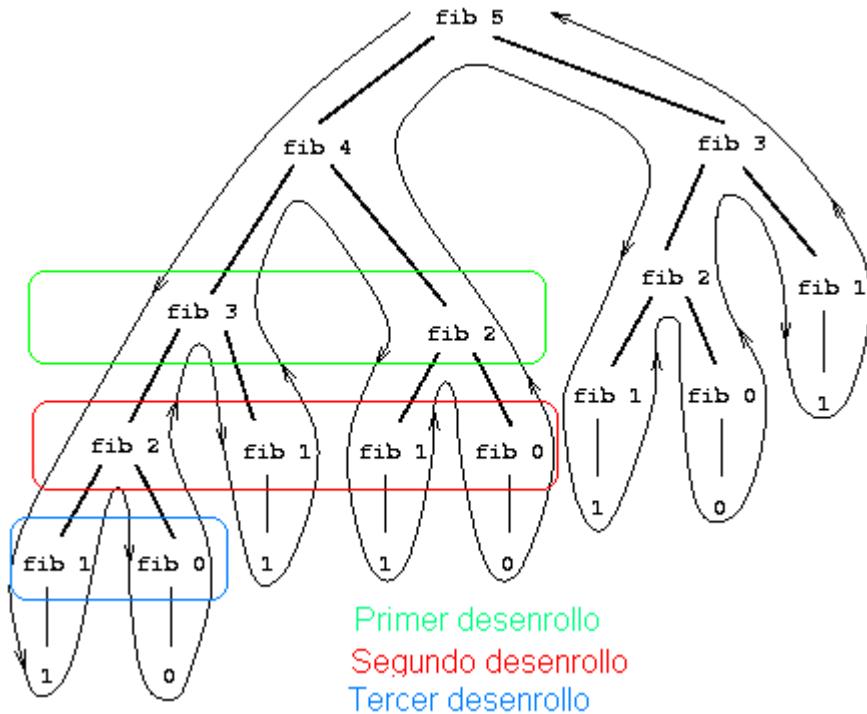


Figura2: Árbol de recursiones y desenrollos

Por último tendremos que contar las llamadas recursivas y conjeturar el O

Del desenrollo y el árbol se puede deducir la siguiente tabla:

Fibo	Cantidad de recursiones	Conjeturas: fibo(n) esta en O(?)
2	2	n
3	4	n+1
4	8	n * 2
5	14	n ^ 1,7
6	14 + 8 + 2 = 24	2 ^ n
7	14 + 24 + 2 = 40	1,6 ^ (n+1)

Tabla1: Conjeturas de ordenes de complejidad

A continuación graficaremos las conjeturas y el orden exacto de complejidad

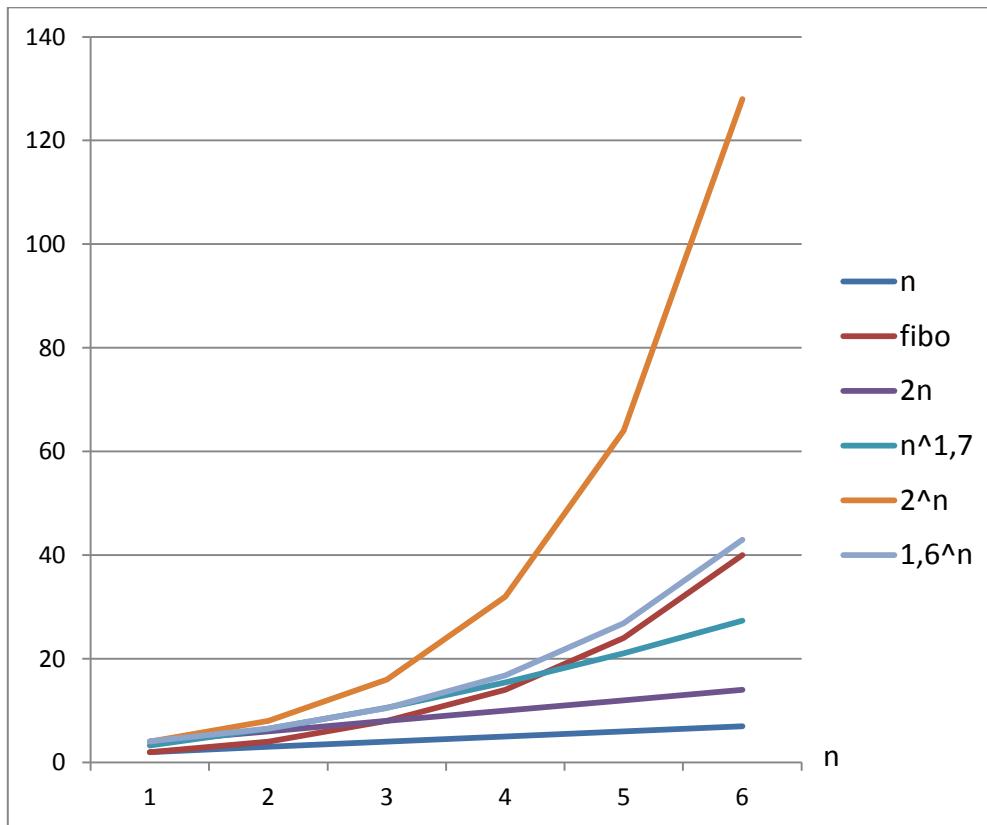


Gráfico1: O(fibo) y conjeturas

El orden de complejidad exacto es: $O\left[\left(\frac{1+\sqrt{5}}{2}\right)^n\right]$

Que aproximadamente es: $O(1,618\dots^n)$

Y a efectos prácticos asumiremos $O(2^n)$

Referencias:

<http://www.dccia.ua.es/dccia/inf/asignaturas/LPP/2010-2011/teoria/tema4.html>

<http://www.lcc.uma.es/~av/Libro/CAP1.pdf>

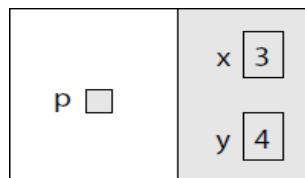
Programación II

Objetos en Java

Definición

Objeto: Conjunto de **datos** y **métodos** relacionados.

Los objetos se alojan en una parte de la RAM reservada al proceso denominada **memoria dinámica**, y son referenciados por las variables del programa o por otros objetos:



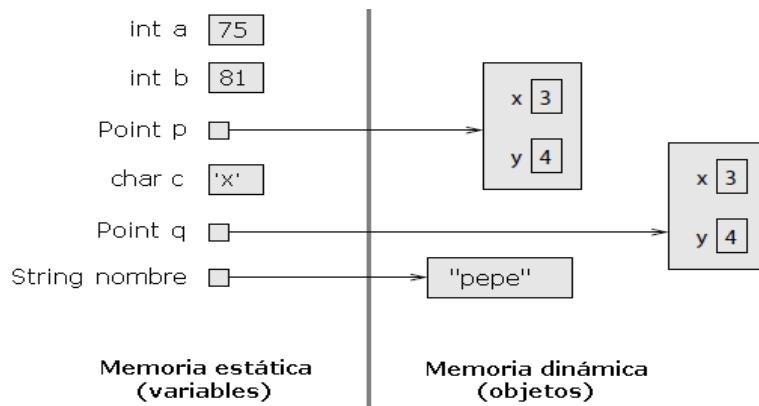
Memoria dinámica

Las variables se alojan en **memoria estática**.

1. Para los tipos básicos, las variables contienen el valor del tipo.
2. Para los objetos, las variables contienen referencias a memoria dinámica.

El **ciclo de vida** de las variables es distinto de acuerdo al tipo de memoria.

1. Las variables (en memoria estática) desaparecen de la memoria cuando se cierra el bloque en el cual fueron declaradas.
2. Los objetos se mantienen en memoria dinámica mientras haya alguna variable que los refiera.

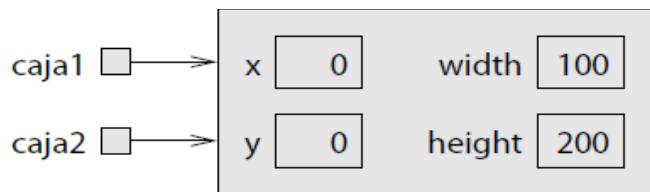


Aliasing

Consideremos este código:

```
Rectangle caja1 = new Rectangle(0, 0, 100, 200);  
Rectangle caja2 = caja1;
```

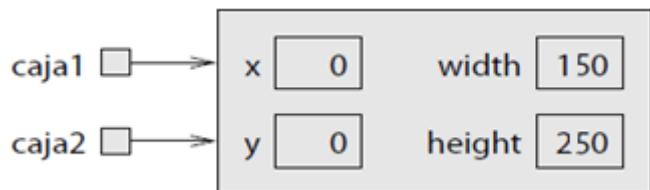
Esto genera que ambas variables referencien al mismo objeto:



Ahora, supongamos que invocamos al método grow() que ensancha y alarga el rectángulo en cada sentido, con las dimensiones especificadas.

```
System.out.println(caja2.width);  
caja1.grow(50,50);  
System.out.println(caja2.width);
```

Si bien llamamos a un método que modificaba a caja1, se modificó también caja2.



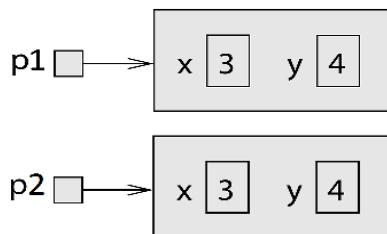
Esto se debe a que las dos variables referencian al mismo objeto. A esto se lo conoce como **aliasing**.

Comparando objetos

Consideremos este código:

```
Point p1 = new Point(3, 4);
Point p2 = new Point(3, 4);
```

Tenemos dos referencias que apuntan a objetos **iguales** pero no **idénticos**:

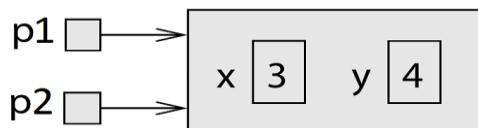


Cuál sería el resultado de la comparación `p1==p2`?

Consideremos este otro código:

```
Point p1 = new Point(3, 4);
Point p2 = p1;
```

Es decir, tenemos el siguiente diagrama:



Cuál sería el resultado de `p1==p2` ahora?

Al igual que con los Strings, la forma de comparar objetos es a través del método `equals()`.

El método `equals()`

Cada clase de Java cuenta con una implementación de este método que compara dos instancias de esa clase.

Cuando definimos nuestras propias clases, es buena práctica definir nuestro propio método `equals` para que se puedan comparar dos objetos de nuestra clase.

Eclipse implementa automáticamente este método con código por defecto!

Clases

Las clases son el “molde” a partir de la cual se crean los objetos, dado que especifican qué variables de instancia contienen y a qué métodos responden.

Los datos y métodos asociados a un tipo de objeto se definen en la clase, con lo cual podemos decir que la clase **representa** un tipo de objeto.

1. Las variables de instancia especifican los **valores** que contienen todas las instancias de la clase.
2. Los métodos de la clase especifican el **comportamiento** de las instancias de la clase, en función de las variables de instancia y otros parámetros.

Conceptos generales

Variables de instancia: Datos incluidos dentro de las instancias de una clase. Se tiene un juego de variables de instancia por cada objeto.

Variables de clase (static): Datos asociados con la clase, comunes a todas las instancias de la clase. Se tiene un único juego de variables de clase para todos los objetos.

Métodos de instancia: Funciones que se ejecutan sobre instancias de la clase. Dentro del código del método accedemos a la instancia en cuestión (**parámetro implícito**) por medio de la referencia **this**.

Métodos de clase: Funciones asociadas con la clase, y que no se ejecutan sobre instancias de la clase. Dentro del código del método no se puede acceder a variables de instancia, porque no hay parámetro implícito.

Miembros públicos: Datos y métodos disponibles por usuarios de la clase, que el objeto “expone” hacia el exterior.

Miembros privados: Datos y métodos disponibles sólo por los métodos de la misma clase, habitualmente utilizados para representar el estado interno de la instancia.

Miembros protegidos: Datos y métodos disponibles sólo por los métodos de la clase y de sus subclases.

Clase Fecha

Ejemplo: supongamos que queremos definir una clase para representar fechas. Esencialmente, una fecha es un día de un mes de un año.

```
class Fecha
{
    int dia;
    int mes;
    int anio;
}
```

¿Cómo hacemos ahora para crear objetos de tipo Fecha?

Constructores

Cuando se crea un objeto de una clase se llama a un método especial llamado **constructor**.

El constructor se encarga de **inicializar** las variables de instancia.
Veamos un ejemplo de constructor para la clase Fecha.

```
Fecha()
{
    this.dia=1;
    this.mes=1;
    this.anio=2000;
}
```

Nótese que el constructor **no devuelve ningún valor**.

La palabra `this` es una referencia al objeto que está siendo "construido".

Cuando escribimos `new Fecha()` en una expresión, se crea un objeto de tipo Fecha y se llama a este método sobre el objeto para inicializarlo.

Más de un constructor

Podemos definir más de un constructor, siempre con parámetros distintos.

Por ejemplo:

```
Fecha(int d, int m, int a)
{
    this.dia = d;
    this.mes = m;
    this.anio = a;
}
```

Este constructor es un **constructor trivial** en el que pasamos un valor para cada variable de instancia.

Es muy común contar con este tipo de constructores.

Métodos de clase (o estáticos)

Hagamos ahora un método de la clase Fecha que nos diga si un año es o no bisiesto.

```
static boolean bisiesto(int anio) { //static denota que es un método de clase
    if (anio % 4 == 0 && anio % 100 != 0)
        return true;
    else if (anio % 400 == 0)
        return true;
    else
        return false;
}
```

En este caso el resultado del método **no depende de ningún objeto**, ya que un año es o no es bisiesto, sin importar de “a quién” se le pregunte.

El anterior es un **método de clase** ya que no depende de alguna instancia en particular.

En un método de clase no existe el parámetro implícito this y el método no se invoca “sobre” un objeto.

Por ejemplo, para listar los años bisiestos hasta el año 2000, podemos usar este método con la siguiente sintaxis:

```
for (int a = 1; a <= 2000; a++)
    if (Fecha.bisiesto(a)) // ← Llamado al método de clase!
        System.out.println("El año " + a + " es bisiesto");
```

El valor null

Cuando declaramos una variable del tipo de algún objeto y no le damos ningún valor inicial, éste, por defecto, es null.

Las siguientes declaraciones son equivalentes:

```
Point p1;
Point p2=null;
```

Si tratamos de acceder a un miembro o llamar a un método de un objeto null, se genera una excepción: NullPointerException.

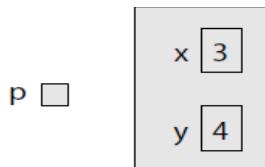


Garbage Collector

El valor null nos sirve también para borrar referencias, por ejemplo:

```
Point p = new Point(3, 4);  
p = null;
```

En este punto hay un objeto en memoria que no es referenciado por nadie, porque p ahora no apunta a nada:



Los objetos que dejan de ser referenciados por alguna variable, son liberados por el **garbage collector**.

El operador new reserva memoria para un nuevo objeto, y esa memoria está en uso hasta que el garbage collector la recupere.

Esto no ocurre así para las variables de los tipos nativos (int, double, boolean), porque no se almacenan en memoria dinámica.

Cuando el contexto en el cual se creó la variable (función, ciclo, etc) se termina, la memoria estática se libera y la variable desaparece.

Programación II

Herencia en Java

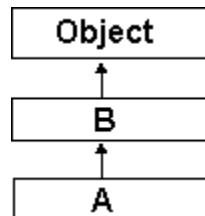
Definición

Se llama **subclase** a una clase A derivada de otra clase B. La clase B se llama la **superclase** o **clase base**.

Al derivar una clase A desde otra clase B, la clase A **hereda** automáticamente todos los miembros (datos y métodos) de la clase B.

Con excepción de la clase Object, todas las clases en Java tienen exactamente una superclase. Esta restricción se conoce con el nombre de **herencia simple**.

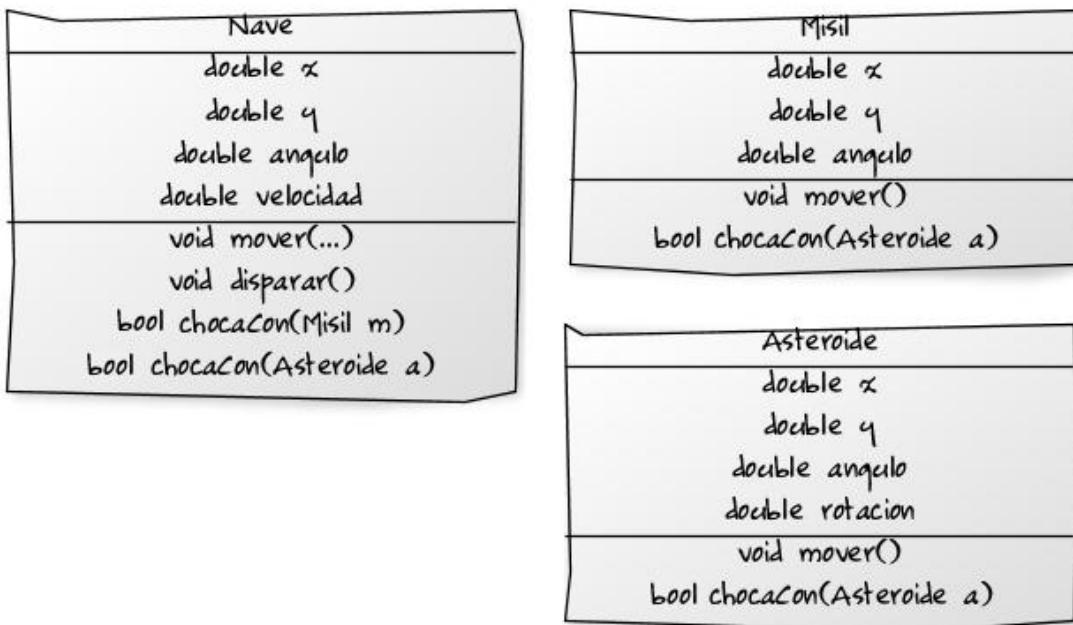
Si para una clase no se declara explícitamente su superclase, se adopta implícitamente la clase Object como superclase.



Ejemplo

Estamos implementando un videojuego en el que dos naves intentan destruirse mutuamente.

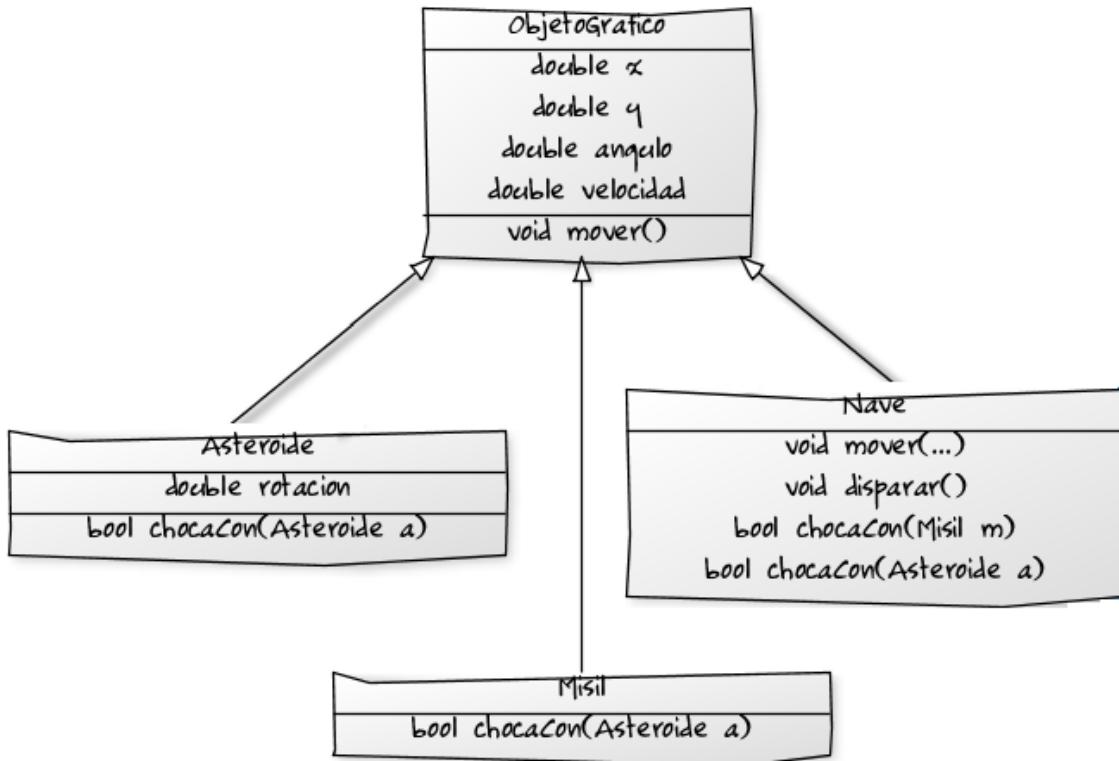
1. Es un juego de dos jugadores (dos personas o bien una persona contra la computadora), cada uno controla una **nave**.
2. Las naves pueden disparar **misiles**. Cuando un misil impacta contra una nave, la nave pierde una unidad de energía. Cuando una nave pierde todas sus unidades de energía, es destruida y pierde el juego.
3. En el espacio de juego hay asteroides moviéndose libremente. Si un misil choca con un asteroide, el misil desaparece. Si una nave choca con un asteroide, pierde una unidad de energía.

Ejemplo1

Las tres clases tienen **datos y métodos repetidos!**

1. Los tres tipos de objetos tienen posición (x; y) en el plano y ángulo.
 2. Los tres se mueven de acuerdo con su posición y su ángulo, y además en un caso se debe tener en cuenta la velocidad.
 3. Hay muchos métodos para chequear si un objeto choca contra otro!
- Más aún, el código de todos ellos seguramente será muy similar.

Cuando sucede esto, es importante preguntarse si estos objetos no son en realidad **miembros de una misma clase**, cada uno con sus características.

Ejemplo2**Sintaxis de la herencia**

```

Class ObjetoGrafico
private double x
private double y
public double x(){ return x}
public double y(){ return y}
...
  
```

Class Asteroide **extends** ObjetoGrafico

Class Nave **extends** ObjetoGrafico

Nave y Asteroide heredan el métodos `x()` e `y()`, los cuales acceden a los atributos privados `x` e `y` respectivamente.

Polimorfismo

La función principal ahora tiene una lista de ObjetoGrafico, que mantiene todos los objetos en la pantalla.

```
ArrayList<ObjetoGrafico> objetos;
Nave nave;
Asteroide asteroide;

objetos.add(nave)
objetos.add(asteroide)
```

Ejemplo2

El polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.

En el ejemplo2, las referencias a ObjetoGráfico son polimórficas, dado que pueden ser de cualquiera de las tres subclases.

Cuando se ejecuta un método sobre un ObjetoGráfico, no es relevante en ese momento de qué tipo es realmente el objeto.

El método mover va tomando la forma del ObjetoGrafico que se este referenciando.

Signatura(Firma)

La firma de una función se compone de 4 componentes que la hacen única:

-Nombre

-Tipo de retorno(Ej. String, Boolean, etc.)

-Modificadores(Ej. Public, Protected, Final, etc.)

-Parametros. Para cada parámetro:

 Nombre

 Posición

 Tipo

Por ejemplo

- (1) Public String toString(), tiene distinta signatura que
- (2) Private String toString()

Mientras que (1) sobrescribe Object.toString().

En el caso (2) solamente se sobrecarga (1).

Sobreescritura (*overriding*) de métodos

Un método de instancia en una subclase con la misma **signatura** y el mismo **tipo de retorno** que un método de instancia de la superclase **sobrescribe** (overrides) el método original.

La subclase Nave **sobrescribe** el método mover() de la clase base.

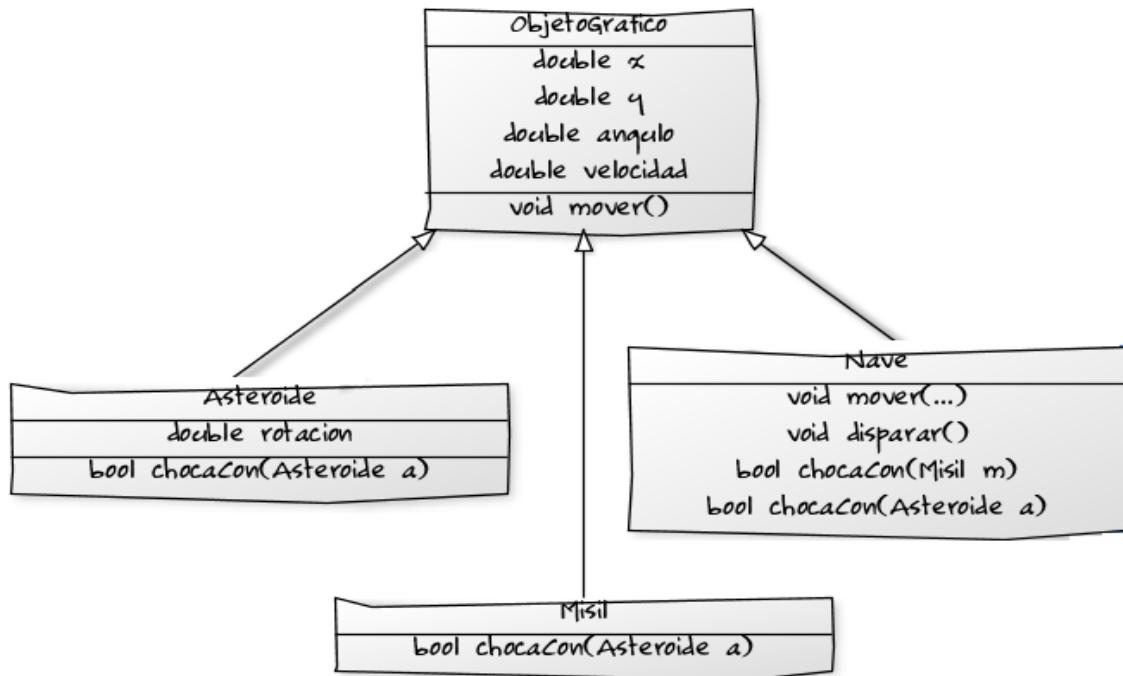
Cuando se llama a este método sobre un ObjetoGráfico, se ejecuta el método sobrescrito en la subclase en cuestión.

Se puede llamar al método de la superclase desde el método sobrescrito, en este caso con ObjetoGráfico.mover().

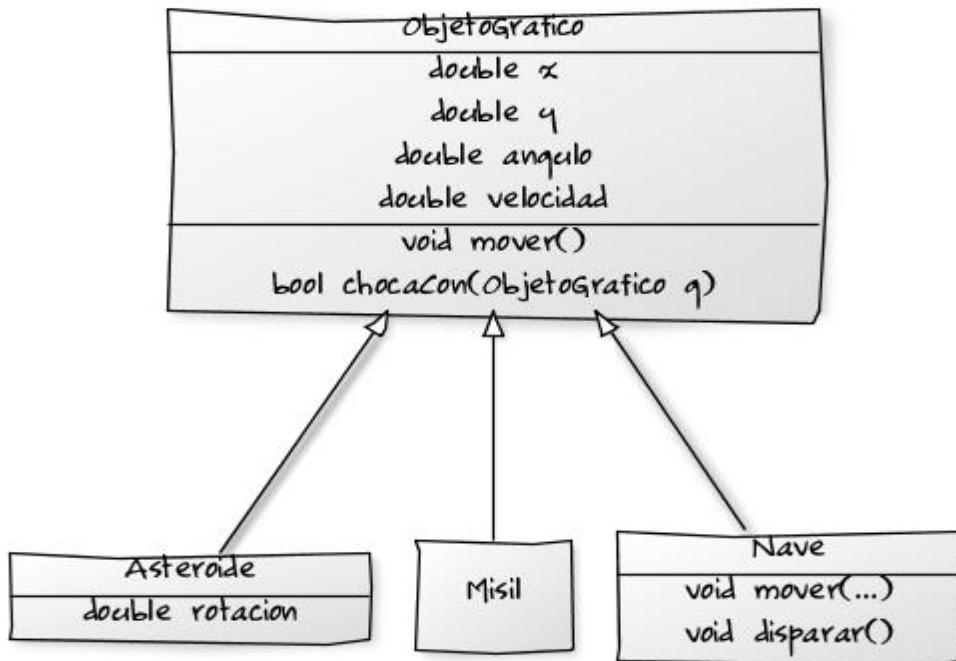
Cuando se sobrescribe un método, se puede poner antes la **anotación “Override”**, para indicarle al compilador que nuestra intención es sobrescribir un método de la superclase.

```
@Override  
void mover()  
{  
    ...  
}
```

Si el método no existe en ninguna de las superclases, se genera un error.

Ejemplo3

¿Podemos hacer algo mas?

Ejemplo4



Sobrecarga

La sobrecarga se refiere a la posibilidad de tener dos o más funciones con el mismo nombre, con la misma funcionalidad pero con diferente signatura. Es decir, dos o más funciones con el mismo nombre con parámetros diferentes. El compilador usará una u otra dependiendo de los parámetros usados.

Ejemplo: Articulo.java

Algunos métodos en una clase pueden tener el mismo nombre. Estos métodos deben contar con diferentes argumentos. El compilador decide qué método invocar comparando los argumentos. Se generara un error si los métodos tienen definidos los mismos parámetros.

```
public class Articulo {  
    private float precio;  
    public void setPrecio() {  
        precio = 3.50;  
    }  
    public void setPrecio(float nuevoPrecio) {  
        precio = nuevoPrecio;  
    }  
}
```

Sobreescritura Vs. Sobre-Carga

Sobreescritura	Sobrecarga
Diferentes clases que se heredan	Misma clase
Misma signatura	Diferente signatura
Diferente Algoritmo	Mismo Algoritmo para otro caso.
Ej: int sumar(int x, int y)	void sumar(int x, int y). El resultado se acumula sobre x.

Clases abstractas

Una **clase abstracta** es una clase que no se puede **instanciar** (no se pueden crear objetos de esa clase), y se utiliza como base para subclases concretas.

En nuestro ejemplo, ObjetoGráfico puede ser una clase abstracta, dado que no tenemos objetos en la aplicación de esa clase.

Un **método abstracto** es un método que no tiene implementación, y que debe sobrescribirse en las subclases.

```
public abstract class ObjetoGráfico
{
    ...
    abstract void dibujar();
}
```

Interfaces

Una **interface** es un tipo de referencia (similar a una clase) que contiene solamente constantes, signaturas de métodos y tipos anidados.

En una interface no hay código y no puede ser instanciada! Es similar a una clase abstracta en la que todos los métodos son abstractos.

Se dice que una clase **implementa** una interface, y se utiliza la palabra **implements** en el código para hacerlo explícito

```
public class Persona implements Serializable
{
    ...
}
```

Clases abstractas Vs. Interfaces

Las clases abstractas pueden contener **métodos implementados**.

Las interfaces no contienen código.

Una clase **implementa** una interface (palabra clave *implements*), mientras que una subclase **extiende** (palabra clave *extends*) una clase base.

Una clase puede heredar de una única clase base, mientras que puede implementar más de una interface.

Una interface especifica un **contrato** que dice qué métodos debe implementar la clase. En cambio, el mecanismo de herencia permite **compartir código común** a varias clases.

Programación II

Herencia en Java (Parte2)

Composición

En anteriores ejemplos se ha visto que una clase tiene datos miembro que son instancias de otras clases. Por ejemplo:

```
class Circulo {  
    Punto centro;  
    int radio;  
    float superficie() {  
        return 3.14 * radio * radio;  
    }  
}
```

Esta técnica en la que una clase se compone o contiene instancias de otras clases se denomina composición. Es una técnica muy habitual cuando se diseñan clases. En el ejemplo diríamos que un Circulo tiene un Punto (centro) y un radio.

Herencia

Pero además de esta técnica de composición es posible pensar en casos en los que una clase es una extensión de otra. Es decir una clase es como otra y además tiene algún tipo de característica propia que la distingue. Por ejemplo podríamos pensar en la clase Empleado y definirla como:

```
class Empleado {  
    String nombre;  
    int numEmpleado , sueldo;  
  
    static private int contador = 0;  
  
    //constructor no-args  
    //Empleado() { }  
  
    Empleado(String nombre, int sueldo) {  
        this.nombre = nombre;  
        this.sueldo = sueldo;  
        numEmpleado = ++contador;  
    }  
  
    public void aumentarSueldo(int porcentaje) {  
        sueldo += (int)(sueldo * porcentaje / 100);  
    }  
  
    public String toString() {  
        return "Num. empleado " + numEmpleado + " Nombre: " + nombre +  
               " Sueldo: " + sueldo;  
    }  
}
```

Con esta representación podemos pensar en otra clase que reuna todas las características de Empleado y añada alguna propia. Por ejemplo, la clase Ejecutivo. A los objetos de esta clase se les podría aplicar todos los datos y métodos de la clase Empleado y añadir algunos, como por ejemplo el hecho de que un Ejecutivo tiene un presupuesto.

Así diríamos que la clase Ejecutivo extiende o hereda la clase Empleado. Esto en Java se hace con la cláusula extends que se incorpora en la definición de la clase, de la siguiente forma:

```
class Ejecutivo extends Empleado {  
    int presupuesto;  
    void asignarPresupuesto(int p) {  
        presupuesto = p;  
    }  
}
```

Con esta definición un Ejecutivo es un Empleado que además tiene algún rasgo distintivo propio. El cuerpo de la clase Ejecutivo incorpora sólo los miembros que son específicos de esta clase, pero implícitamente tiene todo lo que tiene la clase Empleado.

A Empleado se le llama clase base o superclase y a Ejecutivo clase derivada o subclase.

Los objetos de las clases derivadas se crean igual que los de la clase base y pueden acceder tanto sus datos y métodos como a los de la clase base. Por ejemplo:

```
Ejecutivo jefe = new Ejecutivo( "Armando Mucho" , 1000 );  
jefe.asignarPresupuesto(1500);  
jefe.aumentarSueldo(5);
```

Nota: La discusión acerca de los constructores la veremos un poco más adelante.

Atención!: Un Ejecutivo ES un Empleado, pero lo contrario no es cierto.

```
Empleado e = new Empleado ( "Esteban " , 100 ) ;
```

Si escribimos:

```
e.asignarPresupuesto(5000); // error
```

Se producirá un error de compilación pues en la clase Empleado no existe ningún método llamado asignarPresupuesto.

Redefinición de métodos. El uso de *super*.

Además se podría pensar en redefinir algunos métodos de la clase base pero haciendo que métodos con el mismo nombre y características se comporten de forma distinta.

Por ejemplo podríamos pensar en rediseñar el método `toString` de la clase `Empleado` añadiendo las características propias de la clase `Ejecutivo`. Así se podría poner:

```
class Ejecutivo extends Empleado {  
    int presupuesto;  
  
    void asignarPresupuesto(int p) {  
        presupuesto = p;  
    }  
  
    public String toString() {  
        String s = super.toString();  
        s = s + " Presupuesto: " + presupuesto;  
        return s;  
    }  
}
```

De esta forma cuando se invoque `jefe.toString()` se usará el método `toString` de la clase `Ejecutivo` en lugar del existente en la clase `Empleado`.

Observese en el ejemplo el uso de `super`, que representa referencia interna implícita a la clase base (superclase). Mediante `super.toString()` se invoca el método `toString` de la clase `Empleado`

Inicialización de clases derivadas

Cuando se crea un objeto de una clase derivada se crea implicitamente un objeto de la clase base que se inicializa con su constructor correspondiente.

Si en la creación del objeto se usa el constructor no-args, entonces se produce una llamada implicita al constructor no-args para la clase base.

Pero si se usan otros constructores es necesario invocarlos explicitamente.

En nuestro ejemplo dado que la clase método define un constructor, necesitaremos también un constructor para la clase Ejecutivo, que podemos completar así:

```
class Ejecutivo extends Empleado {  
    int presupuesto;  
  
    Ejecutivo () {}           //constructor no-args  
  
    Ejecutivo (String n, int s) {  
        super(n,s);  
    }  
  
    void asignarPresupuesto(int p) {  
        presupuesto = p;  
    }  
  
    public String toString() {  
        String s = super.toString();  
        s = s + " Presupuesto: " + presupuesto;  
        return s;  
    }  
}
```

Observese que el constructor de Ejecutivo invoca directamente al constructor de Empleado mediante *super(argumentos)*.

En caso de resultar necesaria la invocación al constructor de la superclase debe ser la primera sentencia del constructor de la subclase.

Referencias:

<http://www.arrakis.es/~abelp/ApuntesJava/Herencia.htm>

Programación II

Herencia en Java (Parte3)

El modificador de acceso *protected*

El modificador de acceso *protected* es una combinación de los accesos que proporcionan los modificadores *public* y *private*.

protected proporciona acceso público para las clases derivadas y acceso privado (prohibido) para el resto de clases.

Por ejemplo, si en la clase Empleado definimos:

```
class Empleado {  
    protected int sueldo;  
    . . .  
}
```

Entonces desde la clase Ejecutivo se puede acceder al dato miembro sueldo, mientras que si se declara *private*, no.

Up-casting y Down-casting

Siguiendo con el ejemplo de los apartados anteriores, dado que un Ejecutivo ES un Empleado se puede escribir la sentencia:

```
Empleado emp = new Ejecutivo("Máximo Dueño" , 2000);
```

Aquí se crea un objeto de la clase Ejecutivo que se asigna a una referencia de tipo Empleado.

Esto es posible y no da error ni al compilar ni al ejecutar porque Ejecutivo es una clase derivada de Empleado.

A esta operación en que un objeto de una clase derivada se asigna a una referencia cuyo tipo es alguna de las superclases se denomina "***up-casting***".

Cuando se realiza este tipo de operaciones, hay que tener cuidado porque para la referencia emp no existen los miembros de la clase Ejecutivo, aunque la referencia apunte a un objeto de este tipo.

Así, las expresiones:

```
emp.aumentarSueldo(3); // 1. ok. aumentarSueldo es de Empleado  
emp.asignarPresupuesto(1500); // 2. error de compilación
```

En la primera expresión no hay error porque el método aumentarSueldo está definido en la clase Empleado.

En la segunda expresión se produce un error de compilación porque el método asignarPresupuesto no existe para la clase Empleado.

Por último, la situación para el método `toString` es algo más compleja. Si se invoca el método:

```
emp.toString(); // se invoca el metodo toString de Ejecutivo
```

El método que resultará llamado es el de la clase `Ejecutivo`. `toString` existe tanto para `Empleado` como para `Ejecutivo`, por lo que el compilador Java no determina en el momento de la compilación qué método va a usarse.

Sintácticamente la expresión es correcta. El compilador retrasa la decisión de invocar a un método o a otro al momento de la ejecución.

Esta técnica se conoce con el nombre de *dinamic binding o late binding*. En el momento de la ejecución la JVM comprueba el contenido de la referencia `emp`.

Si apunta a un objeto de la clase `Empleado` invocará al método `toString` de esta clase. Si apunta a un objeto `Ejecutivo` invocará por el contrario al método `toString` de `Ejecutivo`.

Operador cast

Si se desea acceder a los métodos de la clase derivada teniendo una referencia de una clase base, como en el ejemplo del apartado anterior hay que convertir explícitamente la referencia de un tipo a otro.

Esto se hace con el operador de **cast** de la siguiente forma:

```
Empleado emp = new Ejecutivo("Máximo Dueño" , 2000);  
Ejecutivo ej = (Ejecutivo)emp; // se convierte la referencia de tipo  
ej.asignarPresupuesto(1500);
```

La expresión de la segunda línea convierte la referencia de tipo `Empleado` asignándola a una referencia de tipo `Ejecutivo`.

Para el compilador es correcto porque `Ejecutivo` es una clase derivada de `Empleado`. En tiempo de ejecución la JVM convertirá la referencia si efectivamente `emp` apunta a un objeto de la clase `Ejecutivo`.

Si se intenta:

```
Empleado emp = new Empleado("Javier Todudas" , 2000);  
Ejecutivo ej = (Ejecutivo)emp;
```

No dará problemas al compilar, pero al ejecutar se producirá un error porque la referencia `emp` apunta a un objeto de clase `Empleado` y no a uno de clase `Ejecutivo`.

Otro Ejemplo de Up-Casting

El upcasting permite tomar decisiones en tiempo de ejecución.

```
public class Dibujo {
    private Figura figura;
    public Dibujo(){ }

    // Ejecturamos esIsosceles solo en Triangulo!
    public void setFigura(String figuraSTR){
        if (figuraSTR == "triangulo"){
            figura = new Triangulo(figuraSTR);
            System.out.println(((Triangulo) figura).esIsosceles());
        }

        if (figuraSTR == "cuadrado") figura =
            new Cuadrado(figuraSTR);
    }

}

public class Figura {
    private String nombre;
    public Figura(){ }

    public Figura(String nombre){
        this.nombre = nombre;
    }
}

public class Cuadrado extends Figura{
    public Cuadrado(String nombre){
        super(nombre);
    }
}

public class Triangulo extends Figura{

    public Triangulo(String nombre){
        super(nombre);
    }
    boolean esIsosceles(){
        return true;
    }
}

public class Test {
    public static void main(String[] args) {
        Dibujo d = new Dibujo();
        d.setFigura("triangulo");

    }
}
```

La clase Object

En Java existe una clase base que es la raíz de la jerarquía y de la cual heredan todas aunque no se diga explícitamente mediante la cláusula extends.

Esta clase base se llama Object y contiene algunos métodos básicos.

La mayor parte de ellos no hacen nada pero pueden ser redefinidos por las clases derivadas para implementar comportamientos específicos.

Los métodos declarados por la clase Object son los siguientes:

```
public class Object {  
    public final Class getClass() { . . . }  
    public String toString() { . . . }  
    public boolean equals(Object obj) { . . . }  
    public int hashCode() { . . . }  
    protected Object clone() throws CloneNotSupportedException  
    { . . . }  
    public final void wait() throws ...  
    public final void wait(long millis) throws ...  
    public final void wait(long millis, int nanos) throws ...  
    public final void notify() throws ...  
    public final void notifyAll() throws ...  
    protected void finalize() throws Throwable { . . . }  
}
```

Las cláusulas *final* y *throws* se verán más adelante.

Como puede verse *toString* es un método de Object, que puede ser redefinido en las clases derivadas.

Los métodos *wait*, *notify* y *notifyAll* tienen que ver con la gestión de threads de la JVM. El método *finalize* ya se ha comentado al hablar del recolector de basura.

Para una descripción exhaustiva de los métodos de Object se puede consultar la documentación de la API del JDK.

La cláusula final

En ocasiones es conveniente que un método no sea redefinido en una clase derivada o incluso que una clase completa no pueda ser extendida.

Para esto está la cláusula final, que tiene significados levemente distintos según se aplique a un dato miembro, a un método o a una clase.

Para una clase, final significa que la clase no puede extenderse. Es, por tanto el punto final de la cadena de clases derivadas.

Por ejemplo si se quisiera impedir la extensión de la clase Ejecutivo, se pondría:

```
final class Ejecutivo {  
    . . .  
}
```

Para un método, *final* significa que no puede redefinirse en una clase derivada. Por ejemplo si declaramos:

```
class Empleado {  
    . . .  
    public final void aumentarSueldo(int porcentaje) {  
        . . .  
    }  
    . . .  
}
```

Entonces la clase Ejecutivo, clase derivada de Empleado no podría reescribir el método aumentarSueldo, y por tanto cambiar su comportamiento.

Para un dato miembro, final significa también que no puede ser redefinido en una clase derivada, como para los métodos, pero además significa que su valor no puede ser cambiado en ningún sitio; es decir el modificador final sirve también para definir valores constantes.

Por ejemplo:

```
class Circulo {  
    . . .  
    public final static float PI = 3.141592;  
    . . .  
}
```

En el ejemplo se define el valor de PI como de tipo float, estático (es igual para todas las instancias), constante (modificador final) y de acceso público.

Herencia simple

Java incorpora un mecanismo de herencia simple. Es decir, una clase sólo puede tener una superclase directa de la cual hereda todos los datos y métodos.

Puede existir una cadena de clases derivadas en que la clase A herede de B y B herede de C, pero no es posible escribir algo como:

```
class A extends B , C .... // error
```

Este mecanismo de herencia múltiple no existe en Java.

Java implanta otro mecanismo que resulta parecido al de herencia múltiple que es el de las interfaces que se verá más adelante.

Referencias:

<http://www.arrakis.es/~abelp/ApuntesJava/Herencia2.htm>

Programación II

Clases paramétricas<G>

Definición

Un TAD o clase paramétrica recibe el tipo(o género G) de clase como parámetro formal. Se utiliza para generalizar el comportamiento de un TAD T, y tiene varias consecuencias inmediatas:

- Mayor abstracción
- Reutilización de código

Si G es un tipo primitivo, generalmente funcionan todas las operaciones de T.
Si G es otro TAD, es necesario también redefinir ciertas funciones que tiene todo TAD¹ de java:
-Equals()
-HashCode()

Sintaxis y semántica

Vamos a mostrar un ejemplo de cómo hacer una clase paramétrica y analizaremos los resultados del método imprimir.

```
Class Ejemplo<G>
G x;
G y;
Public Ejemplo(G valor){
    X = valor;
    Y = valor;
}
public String imprimir(){
    return x.toString() + y.toString(); //implementación1
}
```

¹ Todo TAD de java es un Objeto. Object implementa Equals y HashCode. Por lo tanto es necesario redefinir dichas clases para no tener comportamientos inesperados.

Main

```
Ejemplo ej1 = new Ejemplo(7)<Integer>;
Ejemplo ej2 = new Ejemplo(7)<String>

System.out.println(Ej1.imprimir());
System.out.println(Ej2.imprimir());
```

Ejercicio 1

Implementar la clase Ejemplo y ver que se imprime para cada ejemplo(ej1 y ej2).

Por una limitación de Java, no poder convertir T a String, la siguiente expresión no compila:

```
return (String)(x + y); //implementación2
```

Sin embargo, analizar que pasaría si imprimir() utilizara la **implementación2**.

Análisis

Para el ejemplo1, la suma es de enteros, mientras que para el ejemplo2 la suma es de Strings.
El cast a string "(String)" es necesario para cumplir con el género de imprimir().

Extendiendo G para que sea comparable

En algunos casos, necesitamos que G sea comparable.

Por ejemplo podríamos querer generalizar una agenda, donde G representa a una persona de la agenda.

La forma de declarar esa intención en java es agregando "extends Comparable<G>":

```
public class Agenda<P extends Comparable<P>> {
```

Luego, en la implementación, se podría ser la siguiente manera:

```
public P devolverLaPersonMaschica (P personal, P persona2) {  
    P ret;  
    if (personal.compareTo(persona2) == 1) { // personal < persona2  
        ret = personal;  
    } else {  
        ret = persona2;  
    }  
    return ret;  
}
```

Donde la clase persona, tiene

```
public class Persona implements Comparable<Persona> {  
  
    private int dni;  
  
    @Override  
    public int compareTo(Persona p){  
        int ret = 0;  
        if (dni == p.dni){ret= 0;}  
        if (dni < p.dni){ret= -1;}  
        if (dni > p.dni){ret= 1;}  
        return ret;  
    }  
}
```

Main

```
Persona persona1 = new Persona("Juan");  
  
Persona persona2 = new Persona("Pedro");  
  
Agenda a = new Agenda<Persona>();  
  
System.out.println(a.devolverLaPersonMaschica(persona1,persona2));
```

Persona tiene que implementar la interfaz Comparable.

Esto se explicara mas adelante, en la clase de interfaces.

Los tipos primitivos ya implementan dicha interfaz.

Variables de tipo

Como G puede tomar el valor de cualquier tipo, **a los parámetros de una clase se los los denomina variables de tipo.**

Si se tienen dos variables de tipo G1 y G2 y una clase con dos variables tal que

G1 e1 //e1 es de tipo G1

G2 e2 //e2 es de tipo G2

¿Cuando e₁y e₂son iguales?

Caso		Expresión	¿Cuando e ₁ y e ₂ son iguales? //e ₁ .equals(e ₂)
1	G1 == Int G2 == Int	Int e ₁ Int e ₂	e ₁ == e ₂
2	G1 == Int G2 == String	Int e ₁ String e ₂	Nunca
3	G1 == E ₁ G2 == E ₂	E ₁ e ₁ E ₂ e ₂	1) e ₁ y e ₂ tienen que tener el mismo tipo. Es decir: E ₁ == E ₂ 2) e ₁ == e ₂
4	G1 == E ₁ [] G2 == E ₂ []	E1[]e ₁ E2[]e ₂	1) e ₁ , e ₂ tienen que tener el mismo tamaño 2) E ₁ == E ₂ 3) Para cada elemento e _i de e ₁ y e _j de e ₂ : e _i == e _j

Como todas clases son de tipo Object y Object implementa Equals(), todas las clases responden a ese método².

Sin embargo no podemos comparar objetos (salvo los tipos primitivos), con el Equals() por defecto, porque esto tendrá un comportamiento indeseable.

Con lo cual tenemos que sobreescribir Equals() para cada Objeto que creemos, que deba ser comparado.

```
public class Ejemplo
    @Override
    public boolean equals(Ejemplo e)
```

² Por propiedades de objetos

Programación II

Conjunto paramétrico<T>

Introducción

Extenderemos la clase Conjunto para que sea paramétrica.

Sintaxis

También agregaré la sintaxis para exigir que los elementos del Conjunto sean comparables.

De esta forma, los métodos como “máximo()” podrán funcionar sin problemas.

Conjunto definido sobre el tipo T

Definición de la clase y operaciones básicas:

```
public class Conjunto<T extends Comparable<T>>{
    private ArrayList<T> conjunto;

    public Conjunto() {
        conjunto = new ArrayList<T>();
    }

    public void agregar(T elem) {
        if (!conjunto.contains(elem)) {
            conjunto.add(elem);
        }
    }

    public T iésimo(int i) {
        return conjunto.get(i);
    }

    public int tamaño() {
        return conjunto.size();
    }

    public void eliminar(int i) {
        T elem = conjunto.remove(i);
    }
}
```

A continuación agregaremos operaciones basadas en las operaciones básicas. En particular implementaremos “máximo()”, donde se hace obligatorio que los elementos de Conjunto sean comparables entre si.

```
public void union(Conjunto c) { //conjunto UNION c
    for (int i=0;i<c.tamano();i++) {
        agregar((T)c.iesimo(i));
    }
}

public boolean pertenece(T elem){
    return conjunto.contains(elem);
}

public void interseccion(Conjunto c) {
    for (int i=0;i<tamano();i++) {
        if (!c.pertenece(iesimo(i))){
            eliminar(i);
        }
    }
}

public T maximo() {
    T max = null;

    if (tamano()>0) {
        max = conjunto.get(0);
    }

    for (int i=0;i<tamano();i++) {

        if (max.compareTo(conjunto.get(i)) < 0) {
            //max < conjunto.get(i)
            max=conjunto.get(i);
        }
    }
    return max;
}
}
```

CompareTo() devuelve tres valores posibles:

a.compareTo(b)	Valor
a > b	1
a = b	0
a < b	-1

Por eso si “max.compareTo(conjunto.get(i)) < 0”, significa que:
Max < conjunto.get(i)

Programación II

Práctica 08: Tipos paramétricos

Versión del 31/03/2015

Introducción

Como se vio en las clases teóricas, siempre que sea posible, es mejor que el TAD reciba el tipo como parámetro, a trabajar con un tipo en particular.

Por ejemplo, el TAD Conjunto<**Genero**> es más general que el TAD ConjInt de la práctica anterior.

A **Género** se la denomina variable de tipo porque su dominio son los tipos de datos. Genero puede ser cualquier Objeto predefinido de java(Integer, Boolean, Char, etc), pero también puede ser un Objeto o clase¹ del usuario.

Siguiendo con el ejemplo:

- (1) Conjunto = new Conjunto<Integer>
- (2) Conjunto = new Conjunto<Nat> // Recordar el TAD Nat de la practica anterior.

La implementación de Conjunto<G> va a ser muy similar que la de ConjInt para el caso (1). El caso (2) se verá en las siguientes prácticas.

¹ Sintácticamente la variable de tipo es una clase o objeto en lugar de un TAD que puede involucrar una o más clases.

Ejercicio1: Generalizar

Generalizar los TAD de la práctica de TAD haciendo que reciban el tipo como parámetro.

Por ejemplo, El Árbol binario de búsqueda cuya especificación era:

Especificación TAD ANBint

```
ABBint() {}
void eliminar(Integer i) {}
void agregar(Integer i) {}
Integer buscar(Integer i) {} //devuelve 0 si no encuentra
```

De ahora en adelante tiene que ser:

```
public class ABB<T extends Comparable<T>> //El tipo T tiene que ser
                                                comparable.
```

```
ABB() {} //constructor de ABB de tipo T
void eliminar(T t) {}
void agregar(T t) {}
Nodo buscar(T t) {} //devuelve null si no encuentra t
```

Ejercicio2: Tupla

Acceder directamente a las variables privadas de un TAD es una mala práctica.
Entre otras cosas permite que los que utilizan el TAD estropeen su consistencia (Invariantes de representación).

De aquí en adelante utilizaremos métodos para acceder y modificar el estado del TAD.

Especificación

```
Tupla(T1 x, T2 y) {} // T1 y T2 son variables de tipo
T1 getX() {}
T2 getY() {}
void setX(T1 x) {}
void setY(T2 y) {}
```

- Implementar Tupla<T1,T2>
- Implementar el método Bool mismoGenero() que devuelve True si T1 == T2
Ayuda: Investigar la palabra clave *instanceof*.

Ejercicio3: Coordenada

Vamos a especializar la Tupla en una Coordenada.

Como queremos poder sumar, las Coordenas no van a poder ser paramétricas.

Especificación

```
public class Coordenada extends Tupla

public Coordenada(Integer x, Integer y) {...}

public void sumar(Integer x, Integer y) {...}
```

- a) Implementar Coordenada

Ayuda: Utilizar la palabra clave **super** cuando sea necesario.

- b) Modificar la implementación de Coordenada, de manera que pueda operar tanto con int como con float.

Ejercicio4: Diccionario sobre ArrayList de Tuplas

- a) Re-Implementar el TAD Diccionario, de manera que la clave y el significado sean genéricos.

```
public class Diccionario<CLAVE,SIGNIFICADO> {
    ArrayList<Tupla<T1,T2>> datos;
}
```

- b) Re-Implementar el TAD Diccionario sobre conjunto de Tuplas

```
public class Diccionario<CLAVE,SIGNIFICADO> {
    Conjunto<Tupla<T1,T2>> datos;
}
```

Que implementación tiene mejor abstracción? Porque?

Programación II

Variables de tipo

Variables de tipo

Como G puede tomar el valor de cualquier tipo, **a los parámetros de una clase se los denomina variables de tipo.**

Si se tienen dos variables de tipo G1 y G2 y una clase con dos variables tal que

G1 e1 //e1 es de tipo G1

G2 e2 //e2 es de tipo G2

¿Cuando e₁y e₂ son iguales?

Caso		Expresión	¿Cuando e ₁ y e ₂ son iguales? //e ₁ .equals(e ₂)
1	G1 == Int G2 == Int	Int e ₁ Int e ₂	e ₁ == e ₂
2	G1 == Int G2 == String	Int e ₁ String e ₂	Nunca
3	G1 == E ₁ G2 == E ₂	E ₁ e ₁ E ₂ e ₂	1) e ₁ y e ₂ tienen que tener el mismo tipo. Es decir: E ₁ == E ₂ 2) e ₁ == e ₂
4	G1 == E ₁ [] G2 == E ₂ []	E ₁ []e ₁ E ₂ []e ₂	1) e ₁ y e ₂ tienen que tener el mismo tamaño 2) E ₁ == E ₂ 3) Para cada elemento e _i de e ₁ y e _j de e ₂ : e _i == e _j

Como todas clases heredan de Object y Object implementa Equals(), todas las clases responden a ese método¹.

Sin embargo no podemos comparar objetos (salvo los tipos predefinidos), con el Equals() por defecto, porque esto tendrá un comportamiento indeseable.

Con lo cual tenemos que sobreescibir Equals() para cada Objeto que creemos, que deba ser comparado.

```
public class Ejemplo
```

```
@Override
public boolean equals(Object e)
```

Si es de tipo Ejemplo realizar el siguiente cast: Ejemplo e2 = (Ejemplo)e

Por compatibilidad con las versiones viejas de java, equals recibe un Object en lugar de un Ejemplo y no puede ser paramétrico.

¹ Por propiedades de objetos

Programación II

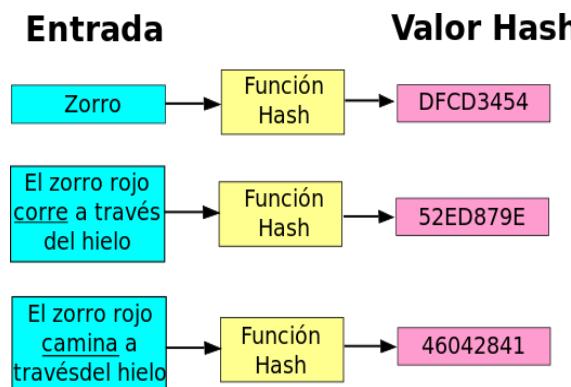
Hash

Una función hash H es una función computable mediante un algoritmo, que tiene como entrada un conjunto de elementos, que suelen ser cadenas, y los convierte (mapea) en un rango de salida finito, normalmente cadenas de longitud fija. Es decir, la función actúa como una proyección del conjunto U sobre el conjunto M .

$$H: U \rightarrow M$$
$$x \rightarrow h(x),$$

Al conjunto U se le llama dominio de la función hash. A un elemento de U se le llama **preimagen** o dependiendo del contexto **clave** o **mensaje**.

Al conjunto M se le llama imagen de la función hash. A un elemento de M se le llama **valor hash**, **código hash** o simplemente **hash**.



Normalmente el conjunto M tiene un número elevado de elementos y U es un conjunto de cadenas con un número más o menos pequeño de símbolos. Por esto se dice que estas funciones resumen datos del conjunto dominio.

La idea básica de un valor hash es que sirva como una representación compacta de la cadena de entrada. Por esta razón decimos que estas funciones **resumen** datos del conjunto dominio.

Ejemplo

```
public class Tad1 {  
    public static int cont;  
  
    public Tad1() {    cont++;    }  
  
    @Override  
    public int hashCode() {  
        return (cont);  
    }  
}  
  
public class test {  
    public static void main(String[] args) {  
        Tad1 a = new Tad1();  
        Tad1 b = new Tad1();  
  
        System.out.println(a. hashCode());  
    }  
}
```

Colisión

Se dice que se produce una **colisión** cuando dos entradas distintas de la función de hash producen la misma salida. De la definición de función hash podemos decir que U, el dominio de la función, puede tener infinitos elementos. Sin embargo M, el rango de la función, tiene un número finito de elementos debido a que el tamaño de sus cadenas es fijo. Por tanto la posibilidad de existencia de colisiones es intrínseca a la definición de función hash. Una buena función de hash es una que tiene pocas colisiones en el conjunto esperado de entrada. Es decir, se desea que la probabilidad de colisión sea muy baja.

Se dice que la función hash es inyectiva cuando cada dato de entrada se mapea a un valor hash diferente. En este caso se dice que la función hash es perfecta. Para que se dé, es necesario que la cardinalidad del conjunto dominio sea inferior o igual a la cardinalidad del conjunto imagen. Normalmente sólo se dan funciones hash perfectas cuando las entradas están preestablecidas.

Ejemplo: Mapear los días del año en números del 1 al 366 según el orden de aparición.

Formalización:

$$k_1 \neq k_2 \quad \text{implica} \quad h(k_1) \neq h(k_2)$$

Cuando no se cumple la propiedad de inyectividad se dice que hay **colisiones**.

Hay una **colisión** cuando $k_1 \neq k_2$ y $h(k_1) = h(k_2)$



```
public class Persona {  
    public int dni;  
  
    public Persona (int dni){      dni = dni; }  
  
    @Override  
    public int hashCode() {  
        return (dni);  
    }  
}
```

¿Cuando hay colisiones en el TAD Persona?

Ejercicio

Hacer el TAD generala que modele la tirada de 6 dados e implemente una función de hash a partir de los valores de los dados.

Hacer que la función no tenga colisiones.

Programación II

Como escribir el método equals en Java¹

Este artículo describe una técnica para sobrescribir el método equals, que preserve el contrato de equals inclusive cuando las subclases de una clase concreta agreguen nuevos campos.

Los errores más comunes

La clase `java.lang.Object` define el método `equals`, sin embargo es un método sorprendentemente difícil de sobrescribir de manera correcta.

Esto resulta problemático, porque la igualdad es la base de muchas otras cosas.

Por ejemplo, una mala implementación de la igualdad, para una clase `C`, impediría colocar exitosamente, instancias de `C` en una colección:

-Sean `c1` y `c2` dos elementos iguales de tipo `C`.

-`c1.equals(c2)` devolvería `true`. Sin embargo podríamos tener el siguiente comportamiento:

```
Set<C> hashSet = new java.util.HashSet<C>();  
hashSet.add(c1);  
hashSet.contains(c2); // returns false!
```

Existen cuatro errores comunes que pueden causar un comportamiento inconsistente del método `equals` cuando se sobrescribe:

1. Definir `equals` con aridad incorrecta
2. Cambiar `equals` son cambiar `hashCode`
3. ¡Definir `equals` basado en campos que pueden cambiar!
4. Hacer una implementación de `equals` que no se comporte como una relación de equivalencia

¹ Título original: "How to Write an Equality Method in Java" por Martin Odersky, Lex Spoon, and Bill Venners

1. Definir equals con aridad(signatura) incorrecta

Consideremos agregar la igualdad a la siguiente clase, que representa puntos:

```
public class Point {  
  
    private final int x;  
    private final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    // ...  
}
```

Una manera obvia pero incorrecta podría ser el Código_{1a}:

```
public boolean equals(Point other) {  
    return (this.getX() == other.getX() && this.getY() == other.getY());  
}
```

Código_{1a}: Equals₁

¿Qué hay de malo con esta implementación? A primera vista parece ser correcta:

```
Point p1 = new Point(1, 2);  
Point p2 = new Point(1, 2);  
Point q = new Point(2, 3);  
  
System.out.println(p1.equals(p2)); // true  
System.out.println(p1.equals(q)); // false
```

Código_{1b}: Ejemplo₁

Sin embargo los problema comienzan cuando queremos agregar puntos en una colección (ver Código_{1b} y Código_{1c}):

```
import java.util.HashSet;  
  
HashSet<Point> coll = new HashSet<Point>();  
coll.add(p1);  
  
System.out.println(coll.contains(p2)); // false
```

Código_{1c}: equals₁ falla

¿Como puede ser que coll parece no contener p₂, siendo que contiene p₁, y que p₁ y p₂ son iguales?

El motivo va a ser mas claro luego de ver Código_{1d}:

```
Object p2a = p2;  
System.out.println(p1.equals(p2a)); // false
```

Código_{1d}: Equals₁ falla

¿qué esta mal? De hecho, la versión de Equals dada en Código_{1a} no sobrescribe Object.equals, pues tiene diferente aridad.

Object.equals tiene la siguiente aridad:

```
public boolean equals(Object other)
```

Como el Código_{1a} recibe un Point en lugar de un Object, no sobrescribe Object.equals, en cambio, es solo una **sobrecarga**.

Por lo tanto, tanto en el Código_{1c} como en el Código_{1d} se utiliza la versión Object de equals.

Una versión mejorada de equals podría ser la siguiente:

```
@Override public boolean equals(Object other) {  
    boolean result = false;  
    if (other instanceof Point) {  
        Point that = (Point) other;  
        result = (this.getX() == that.getX())  
                && this.getY() == that.getY();  
    }  
    return result;  
}
```

Código_{2a}: equals₂

Ahora equals tiene el tipo correcto.

2. Cambiar equals sin cambiar hashCode

Sin embargo si repetimos el test del Código_{1c}, **probablemente** continúe fallando:

```
Point p1 = new Point(1, 2);  
Point p2 = new Point(1, 2);  
  
HashSet<Point> coll = new HashSet <Point>();  
coll.add(p1);  
  
System.out.println(coll.contains(p2)); // false (probablemente)
```

Código_{2b}: equals₂ tambien falla

¿Porque decimos probablemente?

La colección utilizada es un HashSet.

Eso significa que cada objeto obtendrá una posición de hash, basada en su código de hash.

En este caso, la asignación se realizara respecto del hashCode default(Object.hashCode) de p1 y p2.

En la mayoría de los casos se espera que p1 y p2 obtengan diferentes posiciones y por lo tanto el test falle.

Lo que fallo acá fue que redefinimos equals, pero no redefinimos hashCode.

El problema es que el hashCode utilizado para Point viola el contrato que tiene Object:

Si dos objetos son iguales respecto de equals, hashCode tiene que devolver la misma posición para ambos objetos.

Podríamos utilizar la siguiente definición de hashCode para la clase Point:

```
public class Point {  
  
    private final int x;  
    private final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    @Override public boolean equals(Object other) {  
        boolean result = false;  
        if (other instanceof Point) {  
            Point that = (Point) other;  
            result = (this.getX() == that.getX())  
                    && this.getY() == that.getY();  
        }  
        return result;  
    }  
  
    @Override public int hashCode() {  
        return (41 * (41 + getX()) + getY());  
    }  
}
```

Código₃: hashCode

Multiplicando las coordenadas por el número primo 41 nos da una distribución bastante uniforme del código.

Código₃ es una de las muchas posibilidades para hashCode. Además, la función se calcula rápidamente y devuelve valores acotados.

Redefinir hashCode arregla el problema de la igualdad para clases como Point. Sin embargo también se pueden presentar otros problemas.

3. Definir equals basado en campos que pueden cambiar!

Consideremos la siguiente variación de la clase Point:

```
public class Point {  
  
    private int x;  
    private int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setX(int x) { // Problematica  
        this.x = x;  
    }  
  
    public void setY(int y) {  
        this.y = y;  
    }  
  
    @Override public boolean equals(Object other) {  
        boolean result = false;  
        if (other instanceof Point) {  
            Point that = (Point) other;  
            result = (this.getX() == that.getX())  
                    && this.getY() == that.getY();  
        }  
        return result;  
    }  
  
    @Override public int hashCode() {  
        return (41 * (41 + getX()) + getY());  
    }  
}
```

Código_{4a}: Point sin el modificador **final**.

La única respecto del Código₃ es que el Código_{4a} no tiene el modificador **final**.

Ese decir, podemos modificar el punto las veces que queramos.

Consideremos el Código_{4b}:

```
Point p = new Point(1, 2);  
  
HashSet<Point> coll = new HashSet<Point>();  
coll.add(p);  
  
System.out.println(coll.contains(p)); // true  
  
p.setX(p.getX() + 1);  
  
System.out.println(coll.contains(p)); // false (probablemente)
```

Código_{4b}: equals/hashCode fallan

Esto se ve extraño. ¿A donde se fue p?

Lo que sucedió fue que al modificar p, su hashCode dejó de corresponder con hashCode utilizado al ingresar p en el HashSet.

Luego, al preguntar por p, este probablemente ya no se encuentre en la misma posición del hash, por lo tanto, no será encontrado.

La moraleja de este último ejemplo es no hacer depender la comparación entre objetos que van a ser utilizados dentro de un hash, de campos que pueden ser modificados.

Si es necesario, se deben utilizar otros campos para implementar equals/hashCode .

4. Hacer una implementación de equals que no se comporte como una relación de equivalencia

El contrato de Object.equals especifica que equals tiene que implementar una relación de equivalencia para todos los objetos que no sean null.

Para ser una relación de equivalencia se deben cumplir las siguientes propiedades(para cualquier valor no nulo de x, y,z):

-**Reflexividad:** x.equals(x) tiene que devolver true.

-**Simetria:** x.equals(y) == y.equals(x)

-**Transitividad:** Si x.equals(y) y y.equals(z), entonces x.equals(z)

Para cualquier valor no-null de x, x.equals(null) debe devolver false.

Hasta ahora la definición de equals nos viene funcionando.

¿Pero que sucede cuando agregamos subclases?

Consideremos la clase ColoredPoint, que extenderá la clase Point:
ColoredPoint sobrescribirá Point.equals, de manera de considerar los nuevos atributos agregados.

```
public enum Color {  
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET;  
}  
  
public class ColoredPoint extends Point { // Problem: equals not  
// symmetric  
  
    private final Color color;  
  
    public ColoredPoint(int x, int y, Color color) {  
        super(x, y);  
        this.color = color;  
    }  
  
    @Override public boolean equals(Object other) {  
        boolean result = false;  
        if (other instanceof ColoredPoint) {  
            ColoredPoint that = (ColoredPoint) other;  
            result = (this.color.equals(that.color)  
                      && super.equals(that));  
        }  
        return result;  
    }  
}
```

Código 5a: ColoredPoint

Notar que ColoredPoint no necesita sobrescribir Point.hashCode

Esto se debe a que ColoredPoint.equals(Ver Código 5a) es más estricto que Point.equals, por lo tanto:

-Se sigue cumpliendo el contrato de hashCode

-Para cualquier par ColoredPoint iguales respecto ColoredPoint.equal, Point.hashCode devolverá la misma posición.

Si trabajamos únicamente con instancias de ColoredPoint, ColoredPoint.equal funcionará bien.

Sin embargo, si mezclamos ColoredPoint y Point:

```
Point p = new Point(1, 2);  
  
ColoredPoint cp = new ColoredPoint(1, 2, Color.RED);  
  
System.out.println(p.equals(cp)); // true  
  
System.out.println(cp.equals(p)); // false
```

Código 5b: ColoredPoint.equal y Point.equal violan la simetría

Aca hay dos interpretaciones posibles:

O bien las dos clases son comparables

En este caso, ambas funciones deberían haber devuelto true

O bien las dos clases no son comparables

En este caso, ambas funciones deberían haber devuelto false

En cualquier caso Código 5b viola el principio de **simetría**:

- p.equals(cp) != cp.equals(p)

El método canEqual

La solución es implementar el método canEqual a cada clase que implemente equals.

canEqual , para la semántica que queremos, garantiza que solo sean comparables clases del mismo tipo.

Versión final de Point y ColoredPoint:

```
public class Point {  
  
    private final int x;  
    private final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    @Override public boolean equals(Object other) {  
        boolean result = false;  
        if (other instanceof Point) {  
            Point that = (Point) other;  
            result = (that.canEqual(this)  
                      && this.getX() == that.getX()  
                      && this.getY() == that.getY());  
        }  
        return result;  
    }  
  
    @Override public int hashCode() {  
        return (41 * (41 + getX()) + getY());  
    }  
  
    public boolean canEqual(Object other) {  
        return (other instanceof Point);  
    }  
}
```

Código_{6a}: versión final de Point

```
// No longer violates symmetry requirement
public class ColoredPoint extends Point {
    private final Color color;

    public ColoredPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof ColoredPoint) {
            ColoredPoint that = (ColoredPoint) other;
            result = (that.canEqual(this)
                      && this.color.equals(that.color)
                      && super.equals(that));
        }
        return result;
    }

    @Override public int hashCode() {
        return (41 * super.hashCode() + color.hashCode());
    }

    @Override public boolean canEqual(Object other) {
        return (other instanceof ColoredPoint);
    }
}
```

Código_{6b}: Versión final de ColoredPoint

Se puede demostrar que Código_{6a} y Código_{6b} no violan el contrato de equals (relación de equivalencia).

Referencias

<http://www.artima.com/lejava/articles/equality.html>

Comparación de tres implementaciones de Dicc<CLAVE,SIGNIFICADO>

DiccArray

```
public Tupla(T1 e1, T2 e2){  
    this.e1 = e1;  
    this.e2 = e2;  
}  
  
@Override  
public boolean equals(Object t2){  
    boolean ret =true;  
  
    if (!(t2 instanceof Tupla))  
        ret = false;  
    else{  
        // es importante no asumir <T1,T2> para t2,pues podria ser <T3,T4>  
        Tupla<T1,T2> t3 = (Tupla)t2;  
  
        ret = ret && t3.e1.equals(e1) && t3.e2.equals(e2);  
    }  
  
    return ret;  
}  
  
@Override  
public String toString(){  
    String ret = e1.toString() + "," + e2.toString() + ";"  
    return ret;  
}  
}  
  
public class TuplaDicc<T1,T2> extends Tupla<T1,T2> {  
  
    public TuplaDicc(T1 e1, T2 e2){  
        super(e1,e2);  
    }  
  
    @Override  
    public boolean equals(Object t2){  
        boolean ret =true;  
  
        if (!(t2 instanceof Tupla))  
            ret = false;  
        else{  
            // es importante no asumir <T1,T2> para t2,pues podria ser  
            <T3,T4>  
            Tupla<T1,T2> t3 = (Tupla)t2;  
  
            ret = ret && t3.e1.equals(e1); //&& t3.e2.equals(e2);  
        }  
    }  
}
```

```

        return ret;
    }

}

public class DiccT<CLAVE,SIGNIFICADO> {
    ArrayList<TuplaDicc<CLAVE,SIGNIFICADO>> datos;

    public DiccT(){
        datos = new ArrayList<TuplaDicc<CLAVE,SIGNIFICADO>>();
    }

    public void agregar(CLAVE c, SIGNIFICADO s){
        if (!existe(c)) { // <-- deberia se Conjunto x abstraccion
            TuplaDicc<CLAVE,SIGNIFICADO> t = new
            TuplaDicc<CLAVE,SIGNIFICADO>(c,s);
            datos.add(t);
        }
    }

    public boolean existe(CLAVE c){

        SIGNIFICADO s = null;

        //el equals solo pregunta por c
        if (datos.contains(new TuplaDicc<CLAVE,SIGNIFICADO>(c,s))){
            return true;
        }else{
            return false;
        }
    }

    @Override
    public String toString(){
        String ret = datos.toString();

        return ret;
    }
}

public class Test {

    public static void main(String[] args) {
        Tupla<String,String> t1 = new Tupla<String,String>("a1","b1");
        Tupla<String,String> t2 = new Tupla<String,String>("a1","b1");
        Tupla<String,String> t3 = new Tupla<String,String>("a2","b2");
        Tupla<String,Integer> t4 = new Tupla<String,Integer>("a2",3);

        System.out.println(t1.equals(t2)); //true
        System.out.println(t1.equals(t3)); //false
        System.out.println(t1.equals(t4)); //false
    }
}

```

```
System.out.println("Test dicc"); //false

DiccT<String, String> d = new DiccT<String, String>();
d.agregar("a1", "b1");
d.agregar("a2", "b2");

System.out.println(d.existe("a1")); //true
System.out.println(d.existe("a3")); //false
System.out.println(d.toString());

}

}
```

Consola

```
true
false
false
Test dicc
true
false
[a1,b1;, a2,b2;]
```

DicHash

```
public class Tupla<T1,T2> {  
    T1 e1;  
    T2 e2;  
  
    public Tupla(T1 e1, T2 e2){  
        this.e1 = e1;  
        this.e2 = e2;  
    }  
  
    @Override  
    public boolean equals(Object t2){  
        boolean ret =true;  
  
        if (!(t2 instanceof Tupla))  
            ret = false;  
        else{  
            // es importante no asumir <T1,T2> para t2,pues podria ser <T3,T4>  
            Tupla<T1,T2> t3 = (Tupla)t2;  
  
            ret = ret && t3.e1.equals(e1) && t3.e2.equals(e2);  
        }  
  
        return ret;  
    }  
  
    @Override  
    public String toString(){  
        String ret = e1.toString() + "," + e2.toString() + ";"  
  
        return ret;  
    }  
  
}  
public class TuplaDicc<T1,T2>  
extends Tupla<T1,T2> {  
  
    public TuplaDicc(T1 e1, T2 e2){  
        super(e1,e2);  
    }  
  
    @Override  
    public boolean equals(Object t2){  
        boolean ret =true;  
  
        if (!(t2 instanceof Tupla))  
            ret = false;  
        else{  
            // es importante no asumir <T1,T2> para t2,pues podria ser <T3,T4>  
            Tupla<T1,T2> t3 = (Tupla)t2;  
  
            ret = ret && t3.e1.equals(e1); //&& t3.e2.equals(e2);  
        }  
    }  
}
```

```

        }

        return ret;
    }

    @Override
    public int hashCode() {
        return e1.hashCode();
    }
}

public class DiccT<CLAVE,SIGNIFICADO> {
    HashSet<TuplaDicc<CLAVE,SIGNIFICADO>> datos;

    public DiccT(){
        datos = new HashSet<TuplaDicc<CLAVE,SIGNIFICADO>>();
    }

    public void agregar(CLAVE c, SIGNIFICADO s){

        TuplaDicc<CLAVE,SIGNIFICADO> t = new
        TuplaDicc<CLAVE,SIGNIFICADO>(c,s);

        datos.add(t);
    }

    public boolean existe(CLAVE c){

        SIGNIFICADO s = null;

        //el equals solo pregunta por c
        if (datos.contains(new TuplaDicc<CLAVE,SIGNIFICADO>(c,s))){
            return true;
        }else{
            return false;
        }
    }

    @Override
    public String toString(){
        String ret = datos.toString();

        return ret;
    }
}

public class Persona {
    String nombre;

    public Persona(String n){
        nombre =n;
    }
}

```

```

@Override
public boolean equals(Object p2){
    boolean ret =true;

    if (!(p2 instanceof Persona))
        ret = false;
    else{
        // es importante no asumir <T1,T2> para t2,pues podria ser <T3,T4>
        Persona p3 = (Persona)p2;

        ret = ret && p3.nombre.equals(this.nombre);
    }

    return ret;
}

@Override
public int hashCode() {
    return nombre.hashCode();
}
}

public class Test {

    public static void main(String[] args) {
        Tupla<String,String> t1 = new Tupla<String,String>("a1","b1");
        Tupla<String,String> t2 = new Tupla<String,String>("a1","b1");
        Tupla<String,String> t3 = new Tupla<String,String>("a2","b2");
        Tupla<String,Integer> t4 = new Tupla<String,Integer>("a2",3);

        System.out.println("test tupla");
        System.out.println(t1.equals(t2)); //true
        System.out.println(t1.equals(t3)); //false
        System.out.println(t1.equals(t4)); //false

        DiccT<String,String> d = new DiccT<String,String>();
        d.agregar("a1", "b1");
        d.agregar("a2", "b2");

        System.out.println("test diccHash");
        System.out.println(d.existe("a1")); //true
        System.out.println(d.existe("a3")); //false
        System.out.println(d.toString());

        DiccT<Persona,String> d2 = new DiccT<Persona,String>();
        d2.agregar(new Persona("a1"), "b1");
        d2.agregar(new Persona("a2"), "b2");

        System.out.println(d2.existe(new Persona("a1"))); //true
        System.out.println(d2.existe(new Persona("a3"))); //false
        System.out.println(d2.toString());
    }
}

```

```

    }
}

Consola

test tupla
true
false
false
test diccHash
true
false
[a1,b1;, a2,b2;]
true
false
[diccHash.Persona@bf0,b1;, diccHash.Persona@bf1,b2;]

```

TreeSet

```

public class DiccT<CLAVE,SIGNIFICADO> {
    TreeSet<TuplaDicc<CLAVE,SIGNIFICADO>> datos;

public DiccT(){
    datos = new TreeSet<TuplaDicc<CLAVE,SIGNIFICADO>>();
}

public void agregar(CLAVE c, SIGNIFICADO s){

    TuplaDicc<CLAVE,SIGNIFICADO> t = new
    TuplaDicc<CLAVE,SIGNIFICADO>(c,s);

    datos.add(t);
}

...

public class Tupla<T1,T2> implements Comparable<Tupla<T1,T2>> {

T1 e1;
T2 e2;

public Tupla(T1 e1, T2 e2){
    this.e1 = e1;
    this.e2 = e2;
}

@Override
public boolean equals(Object t2){
    boolean ret =true;

```

```

        if (!(t2 instanceof Tupla))
            ret = false;
        else{
// es importante no asumir <T1,T2> para t2,pues podria ser <T3,T4>
            Tupla<T1,T2> t3 = (Tupla)t2;

            ret = ret && t3.e1.equals(e1) && t3.e2.equals(e2);

        }

        return ret;
    }

@Override
public String toString(){
    String ret = e1.toString() + "," + e2.toString() + ";";

    return ret;
}

@Override
public int compareTo(Tupla<T1, T2> t2) {
    // TODO Auto-generated method stub
    String tmp1 = "";
    Integer tmp2 = 0;

    if (t2.e1 instanceof String){
        tmp1 = (String)t2.e1;
        return tmp1.compareTo((String)this.e1) ;
    }

    if (t2.e1 instanceof Integer){
        tmp2 = (Integer)t2.e1;
        return tmp2.compareTo((Integer)this.e1) ;
    }

    return 0;
}
}

```

Resumen

HashSet

ideal

pero no vimos hash

TreeSet

funciona

pero necesita compareTo hardcodeado

ArrayList

funciona

pero no tiene abstraccion correcta