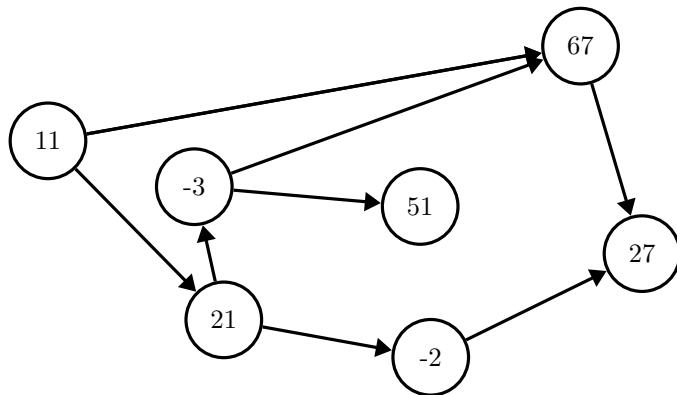


UADE



Programación 2

Facultad de Ingeniería y Ciencias Exactas

Universidad Argentina de la Empresa

Autor: Monzón, Nicolás Alberto

Versión: **1.1.15**

Última edición: 26 de mayo de 2023

Índice general

1. Fundamentos de Java	7
1.1. Variables, operaciones y tipado	7
1.1.1. Tipos de datos primitivos	7
1.1.2. Variables y valores	8
1.1.3. Asignación	9
1.1.4. Instrucciones	10
1.1.5. Comentarios	10
1.1.6. Constantes	10
1.1.7. Orden de las instrucciones	10
1.1.8. Lectura de variables	11
1.1.9. Intercambio de variables	11
1.1.10. Operaciones aritméticas	11
1.1.11. Operaciones booleanas	12
1.1.12. Operaciones de comparación	13
1.1.13. Relaciones de orden y de equivalencia	14
1.1.14. Operación y Acción	15
1.1.15. Propiedades del álgebra de Boole	16
1.1.16. Compuerta XOR	17
1.1.17. Operaciones con caracteres	17
1.1.18. Casteo implícito	17
1.1.19. Precisión	18
1.2. Condicionales y ciclos	19
1.2.1. Sentencia If	19
1.3. Sentencia Short If	24
1.3.1. Sentencia Switch	24
1.3.2. Sentencia While	24
1.3.3. Sentencia For	25
1.3.4. Break y Continue	26
1.3.5. Scopes	28
1.4. Otros conceptos importantes	28
1.4.1. Operaciones simplificadas	28
1.4.2. Función main	29
1.4.3. String	29
1.4.4. Math	31
1.4.5. Entrada y salida de datos	32
1.4.6. Casteo explícito	33
1.5. Ejercicios	34

2. Arreglos, matrices y algoritmos	35
2.1. Arreglos	35
2.1.1. Crear un arreglo con los elementos definidos	35
2.1.2. Crear un arreglo sin los elementos definidos	35
2.1.3. Arreglo vacío	36
2.1.4. Acceso y manipulación de un arreglo	36
2.1.5. Iteración sobre arreglos	36
2.1.6. Contador	37
2.1.7. Acumulador	37
2.1.8. Algoritmos de búsqueda	37
2.1.9. Algoritmos de ordenamiento	39
2.1.10. Sub-arreglo	43
2.2. Matrices	43
2.2.1. Recorrer una matriz	44
2.2.2. Matriz de dimensión cero	44
2.2.3. Matriz nula y matriz identidad	44
2.2.4. Traza de una matriz	44
2.2.5. Traspuesta de una matriz	45
2.3. Ejercicios	45
3. Programación funcional	47
3.1. Funciones	47
3.1.1. Reducciones	48
3.1.2. Composición de funciones	50
3.1.3. Bitwise	51
3.1.4. Funciones inversas	52
3.1.5. Función tirar	52
3.1.6. Función constante	53
3.1.7. Consumidores y productores	54
3.1.8. Funciones puras e impuras	54
3.1.9. Funciones partidas	54
3.2. Recursividad	54
3.3. Ejercicios	56
4. Fundamentos de POO	59
4.1. Introducción a la POO	59
4.1.1. Clases y objetos	59
4.1.2. Interfaces y clases abstractas	60
4.1.3. Records y annotations	60
4.2. Sintaxis en Java	61
4.3. Encapsulamiento	65
4.3.1. GRASP	67
4.4. Herencia y polimorfismo	68
4.5. La superclase Object	71
4.6. Clases abstractas e interfaces	72
4.7. Manejo de errores	76
4.8. Operador For-each	79
4.9. Wrappers	79
4.10. Ejercicios	80

5. Estructuras de datos lineales	82
5.1. Tipo de Dato Abstracto	82
5.2. Lista enlazada	82
5.3. Lista enlazada cíclica	86
5.4. Lista doblemente enlazada	86
5.5. Lista cíclica doblemente enlazada	86
5.6. Pilas	86
5.7. Pila - Implementación estática	87
5.8. Pila - Implementación dinámica	88
5.9. Pila - Operaciones	89
5.10. Cola	90
5.11. Cola - Implementación estática	91
5.12. Cola - Implementación dinámica	92
5.13. Cola - Operaciones	93
5.14. Cola con prioridad	94
5.15. Cola con prioridad - Implementación estática	95
5.16. Cola con prioridad - Implementación dinámica	96
5.17. Cola con prioridad - Operaciones	98
5.18. Comentarios	98
5.19. Ejercicios de TDA Pila	99
5.19.1. Métodos	99
5.19.2. Implementación y complejidad	99
5.19.3. Algoritmos	100
5.20. Ejercicios de TDA Cola	100
5.20.1. Definición	100
5.20.2. Métodos	100
5.20.3. Implementación y complejidad	101
5.21. Ejercicios de TDA Cola con prioridad	101
5.21.1. Definición	101
5.21.2. Métodos	102
5.21.3. Implementación y complejidad	102
6. Complejidad	103
6.0.1. Costo espacial	103
6.0.2. Costo temporal	104
6.0.3. Big- \mathcal{O}	106
6.0.4. Costo lineal	106
6.0.5. Jerarquía de los costos	108
6.0.6. Coste logarítmico	110
6.0.7. Costo indeterminado	110
6.0.8. Comentarios sobre costos temporales	111
6.0.9. Costos de las estructuras lineales	111
7. Conjuntos	113
7.1. Números aleatorios	113
7.1.1. Hash	114
7.1.2. Semilla	115
7.1.3. Función random	115
7.1.4. Función next	116
7.1.5. Comparación de generaciones aleatorias	116
7.1.6. TDA Conjunto	116
7.1.7. Implementación estática	117
7.1.8. Implementación dinámica	118
7.1.9. Algoritmos útiles	120

7.1.10. Cardinalidad	124
7.1.11. Conjunto de partes	124
7.1.12. Comparación con el álgebra de Boole	126
7.1.13. Comentarios	126
7.1.14. Análisis de costos	127
7.2. Ejercicios	127
8. Diccionarios	129
8.0.1. Diccionario Simple - Implementación estática	130
8.0.2. Diccionario Simple - Implementación dinámica	131
8.0.3. Diccionario Simple - Operaciones	133
8.0.4. Diccionario Múltiple - Implementación estática	134
8.0.5. Diccionario Múltiple - Implementación dinámica	137
8.1. Ejercicios	139
9. Árbol binario	141
9.1. BT - Implementación estática	142
9.2. BT - Implementación dinámica	142
9.3. Recorridos recursivos	142
9.4. Recorridos iterativos	142
9.5. Recorrido por nivel	142
9.6. Árbol sesgado, degenerado, completo y perfecto	142
9.7. Caminos	142
9.8. Árbol Binario de Búsqueda (SBT)	142
9.9. SBT - Implementación	142
9.10. Árbol AVL	142
9.11. AVL - Implementación	142
9.12. Análisis de Costos	142
10. Resumen TDAs	143
10.1. Lista enlazada	143
11. Ejemplo: Juego de Cartas	144
11.1. Creación del mazo de cartas	144
11.2. Mezcla de cartas	147
11.3. Repartir las cartas	151
11.4. Comentarios	153
12. Ejercicios resueltos	154
12.1. Series	154
12.1.1. Producto de Wallis (1655)	154
12.1.2. Problema de Basilea (1735)	155
12.1.3. Números primos	156
12.2. Matrices	158
12.2.1. Imprimir en pantalla una matriz cuadrada	158
12.2.2. Determinante matrix 2x2	159
12.2.3. Traza de una matriz	160
12.2.4. Multiplicación escalar	160
12.3. Recursividad	161
12.3.1. Suma de dígitos	161
12.3.2. Divisores de un número	161
12.3.3. Números perfectos	162
12.4. Tipos de Datos Abstractos	162
12.4.1. Pair	162
12.4.2. Vector	164

12.4.3. Qubit	166
12.5. Pila	168
12.5.1. Cantidad de elementos par	168
12.5.2. Eliminar repetidos sin auxiliares	170
12.5.3. Invertir una pila sin auxiliares	172
12.5.4. Estrategia e implementación	172
12.6. Cola	174
12.6.1. Invertir elementos sin una cola auxiliar ni estructuras auxiliares	174
12.6.2. Código	175
12.7. Cola con prioridad	176
12.7.1. Ordenar elementos de una Cola	176
12.7.2. Código	176
12.8. Conjuntos	177
12.8.1. Agenda	177
12.8.2. Booleanos	177
12.9. Diccionario Simple	180
12.9.1. Calcular la intersección de dos diccionarios	180
12.9.2. Código	180
12.10. Diccionario Múltiple	181
12.10.1. Búsqueda sobre valores	181
12.10.2. Valor común a cada clave	182
12.11. Árbol binario	182
12.11.1. Mostrar elementos no repetidos	182
12.11.2. Decidir si pudo haber sido generado con ABB	183
12.11.3. Elemento no repetido	184
12.11.4. Encontrar el elemento más repetido	184
12.11.5. Recorrer un árbol de forma iterativa	185
12.11.6. Recorrido por niveles	186
12.11.7. Decidir si todas las hojas están a la misma altura	187
12.11.8. Imprimir un camino del árbol	187
12.11.9. Diámetro de un árbol	192
12.11.10. Swap	194

Capítulo 1

Fundamentos de Java

1.1. Variables, operaciones y tipado

1.1.1. Tipos de datos primitivos

En Java contamos con 8 tipos de datos primitivos para representar datos numéricos, de caracteres o booleanos.

Tipo de dato	Rango
byte	-128 a 127
short	-32,768 a 32,767
int	-2,147,483,648 a 2,147,483,647
long	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
float	1.4E-45 a 3.4028235E38
double	4.9E-324 a 1.7976931348623157E308

Cuadro 1.1: Tipos de datos numéricos de Java y sus rangos

Para representar un número $n \in \mathbb{Z}$ utilizaremos diferentes tipos de datos según el rango que querremos abarcar. Por ejemplo, `byte` es válido para el rango $[-2^7, 2^7]$, e internamente esto es equivalente a 8 bits (1 para el signo y 7 numéricos), literalmente 1 byte. Con este razonamiento, podemos deducir los rangos asociados a los tipos de datos enteros sabiendo que para m bits, Java puede usar el rango $[-2^{m-1}, 2^{m-1}]$. Para una aproximación, puede usar la propiedad $2^n \approx 10^{\frac{n}{3}}$. Recordando que 1 byte equivale a 8 bits, los pesos de los tipos de datos enteros es:

Tipo de dato	Peso en bytes
byte	1
short	2
int	4
long	8

Cuadro 1.2: Tipos de datos enteros Java y sus pesos

El tipo de dato `long`, en particular, requiere acompañarse de la letra `L`. Por ejemplo, `1L`, `15L` y `-3L`.

De forma análoga, para representar un número $x \in \mathbb{R}$, el tipo de dato `float` ocupa 4 bytes y el tipo de dato `double` ocupa 8 bytes. Más específicamente usa el algoritmo de coma flotante, que utiliza la *aritmética de coma flotante*. Un número como `5.6` es considerado por defecto de tipo `double`, mientras que si escribimos `5.6f` se considerará de tipo `float`. Opcionalmente se puede escribir `5.6d` para hacer explícito que el valor es de tipo `double`. Esto fue considerado una buena práctica durante mucho tiempo pero hoy en día ya no lo es.

Para almacenar caracteres, utilizaremos el tipo de dato `char`, que tiene tamaño de 2 bytes. El valor binario de cada letra puede representarse como un número decimal que tiene asignada una letra en base a codificaciones como la EASCII (ASCII extendido).

El tipo de dato `boolean` permite almacenar `true` o `false`. Es importante notar que el álgebra de Boole utiliza dos elementos opuestos que no necesariamente son `true` o `false`. En este caso, se toman estos valores porque representan *valores de verdad*.

1.1.2. Variables y valores

Una *variable* es un espacio en memoria que podemos utilizar para guardar un *valor*. Dentro de un *contexto*, una variable tiene un nombre único que la representa. La sintaxis para crear una variable es:

```

1 byte a;
2 short b;
3 int c;
4 long d;
5 boolean p, q, r;
6 float x;
7 double y;
8 char c2;
```

Pueden inicializarse varias variables cuando coinciden en el tipo de dato, si se separan por comas. Las variables tienen reglas de sintaxis (de no cumplirse, el código no compilará) y convenios (de no cumplirse, el código compilará pero se considerará desprolijo) que se agrupan bajo el concepto de notación. Las reglas de sintaxis son:

1. El nombre de la variable debe comenzar con una letra o un símbolo de subrayado (underscore).
2. El nombre de la variable no puede comenzar con un número.
3. El nombre de la variable solo puede contener letras, números y símbolos de subrayado (esto da a entender que tampoco es válido espacios de por medio).
4. El nombre de la variable no debe ser una palabra reservada de Java, como `if`, `else`, `while`, `class`, entre otras.
5. Las variables en Java son sensibles a mayúsculas y minúsculas, por lo que `myVariable` y `myvariable` son dos variables diferentes.

Además, por prolividad, los nombres de las variables deben ser descriptivos. Por ejemplo, una variable `age` que represente la edad de una persona deberá ser nombrada `age` en lugar de llamarse `myVariable` o simplemente `a`. Cuando se trata de números algunos convenios son:

1. Si representa un $n \in \mathbb{N}$ se usa `n`, `m`, `o`, etc.
2. Si representa un $n \in \mathbb{Z}$ o los lados de un polígono, se usa `a`, `b`, `c`, etc.
3. Si representa un $n \in \mathbb{R}$ se usa `x`, `y`, `z`, etc.
4. Si representa ángulos se pueden usar los nombres de letras griegas como `alpha`, `beta`, `gamma`, etc.
5. Si se utilizan contadores (haremos énfasis sobre este tema en las siguientes secciones) se utilizan `i`, `j`, `k`, etc.
6. Para caracteres simplemente la letra `c` o la palabra `character`.
7. Para booleanos `p`, `q`, `r` (por las proposiciones en lógica).

Además, es posible que por necesidad de crear múltiples variables, se vea utilizado notaciones con subíndices. Por ejemplo, para múltiples valores en N lo conveniente es usar n_1 , n_2 , n_3 , etc.

Cuando una variable tiene múltiples palabras, se puede optar por alguna de las siguientes opciones: CamelCase, SnakeCase, PascalCase, KebabCase, UpperCase, LowerCase, etc. En Java, se usa CamelCase para las variables, pero con la primera letra en minúscula. Por ejemplo, `isEmpty` es válido, mientras que `IsEmpty` no lo es.

Cuando declaramos una variable, en lugar de dejar los bits basura de la memoria, Java colocará un valor por defecto. En general no podremos acceder a este valor hasta introducir el concepto de POO (Programación Orientada a Objetos).

Tipo de dato	Valor por defecto
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

Cuadro 1.3: Tipos de datos primitivos en Java y sus valores por defecto

1.1.3. Asignación

Ejemplos:

```

9 byte a;
10 a = 1;
11
12 short b = 5;
13 int c = 6 + 1;
14 long d = -3L;
15
16 boolean p = false;
17 boolean q = true;
18
19 float x = 1.3f;
20 double y = 1.3d;
21 char c2 = '@';

```

La asignación de variables sigue la estructura `variable = valor;`. Esto guarda en la variable puesta a la izquierda, lo puesto en la derecha. Este formato se respeta aún conceptualmente en funciones (Si tenemos una función `asignar(a, b)` se puede asumir que esa asignación es de b como valor guardado en a como variable. Si bien esto no es obligatorio, si es un estándar).

Por otro lado, tenemos la posibilidad de inicializar una variable y al mismo tiempo asignarle un valor cuando usamos la estructura `tipo variable = valor;`. El tipo puede ser o no primitivo. Además el valor puede ser calculado, no necesariamente necesita ser un valor explícito.

Se pueden asignar valores de a varios a la vez. Aquí dos ejemplos:

```

22 int a, b, c;
23 a = b = c = 1;
24
25 int d, e, f = e = d = 2;

```

1.1.4. Instrucciones

Las instrucciones en Java tienen un formato del tipo `instrucción;`. El ; es necesario para terminar una instrucción, pero no es necesario que se encuentre siempre al final de la *sentencia*, sino que puede encontrarse más adelante en el código. Los saltos de línea también se permiten siempre y cuando la expresión mantenga el formato establecido. Ejemplo:

```
26 int a = 9
27 ;
28
29 int b =
30 2;
```

1.1.5. Comentarios

Los comentarios permiten ignorar secciones del código. Contamos con comentarios de una línea (usaremos `//`) o de múltiple línea (usaremos `/* */`). Ejemplos:

```
31 // Ejemplo de comentario
32 // de una linea
33
34 /*
35 Ejemplo de comentario
36 de
37 multiples lineas.
38 */
```

1.1.6. Constantes

Las constantes no cambian de valor, es decir, no varían. En Java, para crear una constante usamos la estructura `final tipo nombre = valor;`. Los nombres, por prolíjidad, se escribirán en mayúscula. Como esto trae problemas para la lectura cuando el nombre tiene varias palabras, separamos con guiones bajos. Ejemplos:

```
39 final int a = 1;
40
41 final int b;
42 b = 3;
43
44 final int c;
45 c = 3;
46 c = 2; // Esto dara error porque no puede cambiar su valor.
```

1.1.7. Orden de las instrucciones

La principal diferencia en computación de muchas otras áreas de las matemáticas, es que el orden importa, y mucho. Veamos el siguiente código:

```
47 int a;
48
49 a = 1;
50 a = 2;
```

Al final del código, a valdrá 2, y el valor 1 se habrá perdido para siempre. El valor que se perdió será irrecuperable dado que el espacio en memoria fue sobreescrito con el nuevo valor.

1.1.8. Lectura de variables

Si colocamos una variable donde se espera un valor, entonces el código será válido y se usará como valor el contenido de la variable. Ejemplo:

```
51 int a = 1;
52 int b = a;
53 int a = 2;
54 // a vale 2
55 // b vale 1
```

1.1.9. Intercambio de variables

El algoritmo para intercambiar los valores de dos variables tiene el siguiente formato:

```
56 // Necesitamos dos variables con valores arbitrarios para
57 // intercambiar su contenido
58 int a = 1;
59 int b = 2;
60
61 // Luego, si hacemos
62 // a = b
63 // b = a
64 // Esto no intercambia los valores porque entonces a vale 2 y b vale 2
65
66 // Usamos variable auxiliar
67 int backup = a;
68 a = b;
69 b = backup;
70
71 // a vale 2
72 // b vale 1
```

1.1.10. Operaciones aritméticas

Operación aritmética	Ejemplo en Java
Suma	int a = 5 + 3;
Resta	int b = 7 - 2;
Multiplicación	int c = 4 * 6;
División	double d = 8.0 / 2.0;
Módulo	int e = 9 % 2;
Incremento	int f = 1; f++;
Decremento	int g = 10; g--;
Negación	int h = -5;

Cuadro 1.4: Operaciones aritméticas en Java

La operación módulo es la función que da el resto positivo de una división, que no necesariamente es entera (usualmente el álgebra modular se aprende primero para números enteros). El incremento aumenta en 1 la variable, y el decremento la decrece en 1. Todas estas operaciones se pueden aplicar a `byte`, `short`, `int`, `long`, `float` y `double`. En particular el decremento y el incremento devuelven también el valor de la variable modificada, y se pueden colocar a izquierda o a derecha. Ejemplo:

```
73 int a = 7;
74 int b = a++;
```

```

75 // a vale 8
76 // b vale 7
77
78 int c = 7;
79 inb d = ++c;
80 // c vale 8
81 // d vale 8
82
83 int e = 7;
84 inb f = e--;
85 // e vale 6
86 // f vale 7
87
88 int g = 7;
89 inb h = --g;
90 // g vale 6
91 // h vale 6
92
93 int i = --7; // 7 es un valor, no una variable. --7 es igual a 7

```

Es válido usar paréntesis, y se respeta que la multiplicación se resuelve antes que la suma. Entonces, la *expresión* $((8 + 1) + 2) + 1 * 5$ es una expresión válida y *reduce* a $((8 + 1) + 2) + 5$, que reduce a $(9 + 2) + 5$, etc. Dependiendo del lenguaje de programación la lectura de una expresión que no tiene paréntesis puede ser de izquierda a derecha o de derecha a izquierda. En Java, si no colocamos paréntesis, la lectura será de izquierda a derecha.

1.1.11. Operaciones booleanas

Recordemos las tablas de verdad de AND, OR y NOT:

p	q	p AND q
true	true	true
true	false	false
false	true	false
false	false	false

Cuadro 1.5: Tabla de verdad del operador AND

p	q	p OR q
true	true	true
true	false	true
false	true	true
false	false	false

Cuadro 1.6: Tabla de verdad del operador OR

p	NOT p
true	false
false	true

Cuadro 1.7: Tabla de verdad del operador NOT

Tanto AND, OR y NOT son funciones booleanas. AND y OR reciben dos parámetros booleanos. En general, en la tercera columna de nuestra tabla de verdad tenemos 2^4 posibilidades. Podríamos inventar una nueva función booleana *F*, como por ejemplo:

p	q	p F q
true	true	true
true	false	false
false	true	true
false	false	false

Cuadro 1.8: Tabla de verdad de ejemplo

Sin embargo, con AND, OR y NOT, combinadas de la forma correcta y aplicadas a dos booleanos, pueden recrear esta compuerta F . Es por eso de que no hay necesidad de utilizar otras *compuertas*. Se dice que, AND, OR y NOT forman un *conjunto funcionalmente completo* (o simplemente *conjunto completo*). Existen otros conjuntos completos como por ejemplo el conjunto de un único elemento NAND, y el conjunto de un único elemento NOR. Sin embargo, si usáramos estos conjuntos sería programáticamente complejo programar, y además, existe un isomorfismo de las compuertas AND, OR y NOT con las operaciones intersección, unión y complemento de la teoría de conjuntos que las hacen muy útiles. De todas formas, dado que NAND y NOR son de importancia para la computación, aquí sus tablas de verdad:

p	q	p NAND q
true	true	false
true	false	true
false	true	true
false	false	true

Cuadro 1.9: Tabla de verdad del operador NAND

p	q	p NOR q
true	true	false
true	false	false
false	true	false
false	false	true

Cuadro 1.10: Tabla de verdad del operador NOR

Ejemplo en código

```

94 boolean p = true;
95 boolean q = false;
96
97 boolean r = p && q; // p AND q -> r = false
98 boolean r2 = p || q; // p OR q -> r2 = true
99 boolean r3 = !p; // NOT p -> r3 = false
100
101 boolean r4 = !(p && q); // p NAND q -> r4 = true
102 boolean r5 = !(p || q); // p NOR q -> r5 = false

```

El orden de ejecución es primero el AND antes que el OR, y esto no es una decisión arbitraria, dado que el AND y el OR forman un *anillo booleano*, así como la suma y la multiplicación pueden formar un anillo. Es válido también agrupar en *términos* (en el sentido matemático, ya que en computación significa una expresión válida).

1.1.12. Operaciones de comparación

Para los tipos de datos numéricos podemos comparar valores y obtener `true` o `false`. Por ejemplo `1 >= 2` es `true`, `1 == 2` es `false` y `10 < 10` es `false` (un número nunca es menor o mayor a si mismo).

```

103 int a = 1;
104 int b = 2;
105
106 boolean p;
107
108 p = a < b; // a menor que b
109 p = a <= b; // a menor o igual que b
110 p = a == b; // a exactamente igual que b
111 p = a > b; // a mayor que b
112 p = a >= b; // a mayor o igual que b
113
114 p = a > 10 && a < 20; // a pertenece al rango (10, 20)
115 p = a > 10 && a <= 20; // a pertenece al rango (10, 20]
116 p = a >= 10 && a < 20; // a pertenece al rango [10, 20)
117 p = a >= 10 && a <= 20; // a pertenece al rango [10, 20]
118
119 p = (a >= 10 && a <= 20) || (a > 100);
120 // a pertenece al rango [10, 20] U (100, +infinito)
121
122 p = true != false;
123 p = 'a' == 'b';
124 p = 'a' < 'b';

```

1.1.13. Relaciones de orden y de equivalencia

Definición 1.1.1: Relación

Una relación entre dos conjuntos A y B es un subconjunto $R \subseteq A \times B$. Si $(a, b) \in R$, se dice que a está relacionado con b mediante la relación R , lo cual se denota como aRb .

Las propiedades de una relación de equivalencia son:

- Reflexividad: Para todo $a \in A$, se tiene que aRa .
- Simetría: Para todo $a, b \in A$, si aRb , entonces bRa .
- Transitividad: Para todo $a, b, c \in A$, si aRb y bRc , entonces aRc .

En java, el operador `==` es una relación de equivalencia. Por ejemplo, si tenemos 4 números y queremos saber si son todos iguales, entonces en el siguiente código podemos reducir el cálculo de `result` a `result2`.

```

126 int a = 1;
127 int b = 2;
128 int c = 3;
129 int d = 4;
130
131 boolean result = a == a && a == b && a == c && a == d &&
132             b == a && b == b && b == c && b == d &&
133             c == a && c == b && c == c && c == d &&
134             d == a && d == b && d == c && d == d;
135
136 boolean result2 = a == b && b == c && b == d;

```

Las propiedades de una relación de orden son:

- Reflexividad: Para todo $a \in A$, se tiene que $a \leq a$.

- Antisimetría: Para todo $a, b \in A$, si $a \leq b$ y $b \leq a$, entonces $a = b$.
- Transitividad: Para todo $a, b, c \in A$, si $a \leq b$ y $b \leq c$, entonces $a \leq c$.

En Java, los operadores `<` y `>` son una relación de orden.

Las propiedades de una relación de orden parcial son:

- Reflexividad: Para todo $a \in \mathbb{R}$, se tiene que $a \leq a$.
- Antisimetría: Para todo $a, b \in \mathbb{R}$, si $a \leq b$ y $b \leq a$, entonces $a = b$.
- Transitividad: Para todo $a, b, c \in \mathbb{R}$, si $a \leq b$ y $b \leq c$, entonces $a \leq c$.

En Java, los operadores `<=` y `>=` son una relación de orden parcial. Las propiedades de una relación de *orden total* no serán mencionadas. La operación de desigualdad no es de orden ni de equivalencia, solo es cumple la propiedad simétrica ya que si $a != b == b != a$. Además la solución es complementaria a la igualdad, ya que si $a != b$ es `true` entonces $a == b$ es `false` y viceversa.

1.1.14. Operación y Acción

Los tipos de datos que vimos, indican un conjunto de valores posibles. Por ejemplo `int = {-231, -231 + 1, ..., -1, 0, 1, 2, ..., 231 - 1}`, o por ejemplo `boolean = {true, false}`. Entonces, pensando en conjuntos:

Definición 1.1.2: Acción

La *acción* de un conjunto A sobre un conjunto B es una función $\varphi : A \times B \rightarrow B$, que asigna a cada elemento de A una transformación de B .

Definición 1.1.3: Operación binaria

La *operación binaria* con un conjunto A , es una función $f : A \times A \rightarrow A$, que toma dos elementos de A y produce un nuevo elemento que también pertenece a A .

Definición 1.1.4: Elemento neutro

Sea S un conjunto y $*$ una operación binaria definida en S . Un elemento $e \in S$ se llama *elemento neutro* de $*$ si para cualquier $a \in S$, se cumple que $a * e = e * a = a$.

Definición 1.1.5: Elemento idempotente

Sea S un conjunto y $*$ una operación binaria definida en S . Un elemento $a \in S$ se llama *elemento idempotente* si se cumple que para otro elemento $b \in S$ siempre se cumple que $a * b = b * a = a$.

Para la suma, si a es un número ¿qué elemento debo colocar en $a + _ = a$ para que se cumpla la igualdad? La respuesta es el número 0. Entonces, 0 es el elemento neutro de la suma a derecha. Luego, como también puede completar la igualdad $_ + a = a$, entonces 0 también es el elemento neutro a izquierda. Cuando se cumplen ambas situaciones decimos que es el elemento neutro. En general, nos interesarán sobre todo los elementos neutros a izquierda. Es importante aclarar que un elemento neutro para un operador quizás no exista. Por ejemplo, si planteo $a / _ = a$, el elemento que busco es el número 1 pero si planteo $_ / a = a$ no hay solución (lo que busco tiene que ser un elemento concreto, porque está claro que con a^2 esto podría volverse verdadero pero como a es variable no nos sirve).

Tipo de dato	Operación	Elemento neutro	Elemento idempotente
Números enteros	Suma (+)	0	No existe
	Multiplicación (\times)	1	0
Números racionales	Suma (+)	0	No existe
	Multiplicación (\times)	1	0
Números reales	Suma (+)	0	No existe
	Multiplicación (\times)	1	0
Booleanos	Conjunción (\wedge , AND)	true	false
	Disyunción (\vee , OR)	false	true

Cuadro 1.11: Tipos de datos primitivos, sus operaciones, elementos neutros y elementos idempotentes

1.1.15. Propiedades del álgebra de Boole

Sean p , q , r , variables booleanas, las propiedades del álgebra de Boole son:

- Leyes de la identidad:

- $p \text{ || false} == p$
- $p \text{ && true} == p$

- Leyes de la inversión:

- $p \text{ || !p} == \text{true}$
- $p \text{ && !p} == \text{false}$

- Leyes de la commutatividad:

- $p \text{ || q} == q \text{ || p}$
- $p \text{ && q} == q \text{ && p}$

- Leyes de la asociatividad:

- $(p \text{ || q}) \text{ || r} == p \text{ || (q || r)}$
- $(p \text{ && q}) \text{ && r} == p \text{ && (q && r)}$

- Leyes de la distributividad:

- $p \text{ && (q || r)} == (p \text{ && q}) \text{ || (p && r)}$
- $p \text{ || (q && r)} == (p \text{ || q}) \text{ && (p || r)}$

- Leyes de la absorción:

- $p \text{ || (p && q)} == p$
- $p \text{ && (p || q)} == p$

- Leyes de De Morgan:

- $!(p \text{ || q}) == !p \text{ && !q}$
- $!(p \text{ && q}) == !p \text{ || !q}$

p	q	p XOR q
true	true	false
true	false	true
false	true	true
false	false	false

Cuadro 1.12: Tabla de verdad del operador XOR

1.1.16. Compuerta XOR

La compuerta XOR (también llamada *OR exclusivo*) cuenta con varias propiedades:

- Inversa: La compuerta XOR es la función booleana inversa de si misma. Entonces $p \text{ XOR } p$ da false.
- Conmutativa: $p \text{ XOR } q$ es equivalente a $q \text{ XOR } p$.
- Asociativa: $p \text{ XOR } (q \text{ XOR } r)$ es equivalente a $(p \text{ XOR } q) \text{ XOR } r$.
- Paridad: Si se aplica la compuerta XOR a una lista de booleanos, donde hay n veces true, el resultado será true solo si n es impar.

La compuerta XOR puede ser expresada mediante una expresión lógica, o bien, podemos usar una compresión. Ejemplo:

```
137 boolean p = false;
138 boolean q = false;
139
140 boolean result = (p || q) && !(p && q);
141 boolean result2 = p != q;
142 // result y result2 valen lo mismo.
```

1.1.17. Operaciones con caracteres

En Java, los caracteres son tratados como números enteros y visualizados según la tabla ASCII (o EASCII). Es decir, vamos a poder aplicar todas las operaciones aritméticas vistas, de igualdad y de comparación.

```
143 char c = 'a';
144 c = c + 1; // c vale 'b'
145 c = c + 3; // c vale 'e'
```

1.1.18. Casteo implícito

El casteo es la forma que tenemos de convertir un tipo de dato a otro. Java lo hará de forma automática siempre y cuando no haya *pérdida de información*. Por ejemplo, todo número de tipo byte se puede escribir como un short. O con el mismo razonamiento podemos convertir un float a un double. La primera excepción a la regla es char, que se puede convertir a un tipo de dato numérico dado que internamente funciona como un entero, pero que se puede convertir a byte. En cambio a tipos esto suena lógico, hasta que recordamos que char ocupa 2 bytes y el tipo de dato byte solo 1. Entonces, en este caso, tiene que existir una pérdida de información a no ser que se esté en el rango válido. La última excepción a la regla es cuando convertimos long a float.

```
146 char c = 'a';
147 c = 19;
148
149 byte a = 10;
150 int b = a;
151 float x = b;
```

```

152 double y = x;
153
154 byte a2;
155 int b2 = a2;
156 // Esto no es posible dado que a2 no tiene un valor asignado.
157
158 double z = 1 + 1.5 + 2L; // Gana double

```

Para poder realizar un casteo implícito es necesario respetar la jerarquía `char` → `byte` → `short` → `int` → `long` → `float` → `double`. Son validas todas las transitividades. Cuando se hacen operaciones entre distintos tipos, gana el tipo de dato que se encuentra más alto en la jerarquía.

1.1.19. Precisión

El tipo de dato `byte` tiene como tope el número 127. ¿Qué pasaría si hacemos `byte a = 127 + 10`? En general, se dice que los números enteros son *cerrados* respecto a la suma, y esto es porque la suma es una operación binaria y nos permite que a partir de dos números enteros obtengamos otro número entero. Pero el conjunto \mathbb{Z} es infinito, mientras que `byte` no lo es. Dependiendo del lenguaje, esto puede dar error o descartar el bit más significativo. Java hace lo segundo, entonces si nos excedemos de 127, apareceremos en el lado negativo de su rango. Es decir, $127 + 1 = -128$. Podemos considerar una solución parcial, por ejemplo:

```

159 byte a = 127;
160 byte b = 100;
161 short c = a + b;

```

En el código anterior, usamos un tipo de datos más grande para contener la solución. Sin embargo, no existe un tipo más grande que `long`, por lo que el problema no desaparece complemente. Para lidiar con esto, se suele usar cadenas de caracteres y algoritmos de suma de dígito a dígito (y aún así para números muy grandes se nos puede llenar la memoria y hay que recurrir a técnicas sobre archivos). Además, en el ejemplo anterior, la suma de $127 + 127$ nos daría 254, y esto solo necesita 1 bit más para guardarlo en la memoria, mientras que `short` nos da 8 bits más, por lo que es ineficiente. Usualmente con los programas que trabajaremos `int` será suficientemente preciso, pero en casos extremos es recomendable usar cadenas de caracteres.

Por otro lado, como se comentó al iniciar este capítulo, Java usa el algoritmo de coma flotante. Este algoritmo, tiene un margen de error pequeño, pero existe. Por ejemplo:

```

162 float x = 1.3f;
163 float y = 1.2f;
164 float z = 0.1f;
165 boolean equals = x - y == z; // Da false, porque tiene un margen de error

```

Entonces, para estos cálculos debemos apoyarnos en considerar un margen de error. Podemos, "pasar" para la izquierda la variable `z` restando, y luego verificar que el valor absoluto del error sea muy pequeño. Existen muchas formas, por lo que dejo aquí solo una:

```

166 float x = 1.3f;
167 float y = 1.2f;
168 float z = 0.1f;
169 float error = (x - y) - (z); // Los parentesis son optativos
170 boolean equals = error > -0.001 && error < 0.001;

```

En este ejemplo, la tolerancia es de módulo 0.001. Si el margen de error es más pequeño que esto, podemos considerar los valores como iguales.

1.2. Condicionales y ciclos

1.2.1. Sentencia If

La implicación lógica y la equivalencia lógica tienen las siguientes tablas:

p	q	p EQUIV q
true	true	true
true	false	false
false	true	false
false	false	true

Cuadro 1.13: Tabla de verdad del operador Equivalencia

p	q	p IMP q
true	true	true
true	false	false
false	true	true
false	false	true

Cuadro 1.14: Tabla de verdad del operador Implicación

La equivalencia, la podemos obtener usando el operador == con booleanos, mientras que la implicación con la expresión lógica !p || q, que es lo mismo que decir que p implica q.

Los condicionales son estructuras de control de flujo que permiten que un *bloque de código* se ejecute solo si la condición que le indiquemos es verdadera. La condición debe ser de tipo booleana, y el bloque de código lo indicaremos con {} . Ejemplo:

```
171 int i = 0;
172 i = i + 1;
173 if (i % 2 == 0) {
174     i = i + 1;
175 }
```

El bloque de código anterior toma un entero i y usa la condición i % 2 == 0, que es lo mismo que preguntar si i es par. Luego, solo si es par, i será incrementado. Es decir, que no importa con qué valor inicializamos la variable i, al final de nuestro programa siempre será un número impar. También podemos notar que el bloque de código está tabulado una sola vez a la derecha. Esto se hace únicamente por cuestiones de prolíjidad.

Cuando una sentencia if tienen una única instrucción dentro de su bloque de código, entonces no es necesario colocar {}. Esto se considera una mala práctica en cuanto a prolíjidad, dado que incrementa las posibilidades de que un programador se equivoque al escribir el código. Los ejemplos siguientes son equivalentes al código anterior:

```
176 int i = 0;
177 i = i + 1;
178 if (i % 2 == 0)
179     i = i + 1;
```

```
180 int i = 0;
181 i = i + 1;
182 if (i % 2 == 0) i = i + 1;
```

En la tabla de verdad de la implicación, vimos que da como resultado false únicamente cuando false implica true. Esta situación es análoga, nunca se ejecutará el bloque de código cuando la condición sea false.

Se dice que hay `if` anidados, cuando una sentencia `if` se encuentra dentro de otra. También contamos con la sentencia `else` que se ejecuta cuando la condición no se cumple. La sentencia `else` es optativa. Por otro lado, contamos con la sentencia `else if` que permite volver a establecer una condición de forma consecutiva al `else`. Ejemplos:

```

183 int i = 0;
184 if(i > 1) {
185     i = 1;
186 } else if(i < -1) {
187     i = -1;
188 }
189 // i siempre vale -1, 0 o 1.
190
191 if(i > 0) {
192
193 } else {
194     i = i * i;
195 }
196 // i siempre vale 0 o 1.

```

En la última parte del código, vemos la estructura `if` vacía pero `else` no. Solo la sentencia `else` es optativa, por lo que hay que recurrir a otra técnica para poder simplificar el código. Podemos invertir el contenido de los bloques de código de una sentencia `else` y una sentencia `if` invirtiendo la condición:

```

197 int i = 0;
198 if(i > 0) {
199
200 } else {
201     i = i * i;
202 }
203
204 // Es equivalente a
205
206 int j = 0;
207 if(!(j > 0)) {
208     j = j * j;
209 } else {
210
211 }
212
213 // Luego, esto es equivalente a
214
215 int k = 0;
216 if(!(k > 0)) {
217     k = k * k;
218 }

```

Un ejemplo más complicado con esta inversión de bloque es con la sentencia `else if`. Para esto, reescribimos el código para obtener un `else` y luego invertimos la condición:

```

219 int i = 0;
220 if(i > 0) {
221
222 } else if (i < 0){
223     i = i * i;
224 }
225
226 // Es equivalente a
227
228 int j = 0;

```

```

229 if(j > 0) {
230 }
231 } else {
232     if(j < 0) {
233         j = j * j;
234     }
235 }
236
237 // Es equivalente a
238
239 int k = 0;
240 if(k > 0) {
241     if(k < 0) {
242         k = k * k;
243     }
244 } else {
245 }
246
247
248 // Luego, esto es equivalente a
249
250 int l = 0;
251 if(l > 0) {
252     if(l < 0) {
253         l = l * l;
254     }
255 }
256 // Esto siempre es false. Este código es innecesario.

```

Por otro lado, las sentencias `if` anidadas pueden simplificarse usando el operador AND:

```

257 int i = 0;
258 if(i >= 0) {
259     if(i == 0) {
260         i = 100;
261     }
262 }
263
264 // Es equivalente a
265
266 int j = 0;
267 if(j >= 0 && j == 0) {
268     j == 100;
269 }

```

Dado que las desigualdades representan intervalos de valores posibles, el AND funciona como la intersección de estos intervalos. Entonces, como la intersección entre $\{0\}$ y $[0, \text{infinito})$ es $\{0\}$, entonces la condición $j \geq 0 \&\& j == 0$ se puede simplificar a $j == 0$. A su vez, una condición $l < 0 \&\& l > 0$ es imposible porque no existen valores en su intersección, por lo que esto siempre será `false`.

Los `if` consecutivos que comparten código se pueden unificar mediante el operador `||`. Ejemplo:

```

270 int i = 0;
271 if(i >= 0) {
272     i = 1;
273 }
274 if(i < 0) {
275     i = 1;
276 }
277

```

```

278 // Es equivalente a
279
280 int j = 0;
281 if(j >= 0 || j < 0) {
282     j = 1;
283 }

```

En este caso, $j \geq 0 \text{ || } j < 0$, se lo puede interpretar como la unión de estos intervalos: $[0, +\infty)$ $\cup (-\infty, 0)$ es igual \mathbb{Z} . Entonces, esto será `true` para cualquier entero. Al saber si la condición es `true` o `false` siempre, podemos reducir el código también:

```

284 int i = 0;
285 if(true) {
286     i = 1;
287 } else {
288     i = 2;
289 }
290
291 // Es equivalente a
292
293 int j = 0;
294 j = 1;
295
296 // A su vez,
297
298 if(false) {
299     i = 1;
300 } else {
301     i = 2;
302 }
303
304 // Es equivalente a
305
306 j = 2;

```

Achicar el código aplicando el operador `||` es posible cuando el bloque de código de las condiciones es el mismo, y esto se puede generalizar a múltiples condicionales. Un poco más difícil, pero válido es algo muy parecido a *sacar factor común* en aritmética. Veamos:

```

307 int i = 0;
308 int i2 = 0;
309 if(i > 0) {
310     i = 1;
311     i = 3;
312 } else {
313     i = 1;
314 }
315
316 // Es equivalente a
317
318 int j = 0;
319 int j2 = 0;
320 j = 1;
321 if(i > 0) {
322     j = 3;
323 } else {
324
325 }
326

```

```

327 // A su vez
328
329 if(i > 0) {
330     i = 3;
331     i = 1;
332 } else {
333     i = 1;
334 }
335
336 // Es equivalente a
337
338 if(i > 0) {
339     i = 3;
340 } else {
341 }
342 i = 1;
343

```

Por otro lado, aplicando lo que aprendimos para condicionales anidados pero que contienen condicionales consecutivos por dentro, su comportamiento es algo similar a la *propiedad distributiva* en aritmética. Veamos:

```

344 int i = 0;
345 if(i > 0) {
346     if(i > 1) {
347         i = 1;
348     }
349     if(i > 10) {
350         i = 10;
351     }
352 }
353
354 // Es equivalente a
355
356 int j = 0;
357 if(j > 0 && j > 1) {
358     j = 1;
359 }
360 if(j > 0 && j > 10) {
361     j = 10;
362 }

```

Por último, cuando queremos aplicar la negación de una condición, y tenemos operadores de comparación, podemos hacer una sustitución por algo equivalente. Se adjunta una tabla con la equivalencia.

Operador negado	Equivalente
$! (a >b)$	$a \leq b$
$! (a \geq b)$	$a <b$
$! (a <b)$	$a \geq b$
$! (a \leq b)$	$a >b$
$! (a == b)$	$a != b$
$! (a != b)$	$a == b$

Cuadro 1.15: Operadores de comparación en Java negados

1.3. Sentencia Short If

La sentencia `if` también es conocida como sentencia `if-then-else`. Esta estructura puede ser usada muchas veces para asignar valores a variables y con una única instrucción de por medio en el bloque de código. Ejemplo:

```
363 int a = -8;
364 int b;
365 if(a >= 0) {
366     b = a;
367 } else {
368     b = -a;
369 }
370 // Calculamos el valor absoluto de a
```

Esto puede simplificarse usando la estructura `short-if` que es de la forma `condicion ? valorVerdadero : valorFalso;` y entonces el código anterior es equivalente al siguiente:

```
371 int a = -8;
372 int b = a >= 0 ? a : -a;
```

Se pueden anidar sentencias de tipo `short-if` de la siguiente manera:

```
373 int n = -8;
374 int collatz = n % 2 == 0 ? n / 2 : 3 * n + 1;
```

1.3.1. Sentencia Switch

La `switch` es fuerte en programación funcional y en pattern matching, aunque, por buenas prácticas, un poco más débil en su estructura básica. Esta es una pequeña presentación de esta estructura, dado que en lo posible evitaremos usarla.

```
375 int n = 5;
376 int result = 1;
377 switch (n) {
378     case 1:
379         result = result + 1;
380         break;
381     case 2:
382         result--;
383         break;
384     case 3:
385         result = 0;
386         break;
387     default:
388         result++;
389         result = result - n;
390         break;
391 }
```

El `switch` permite evaluar varios casos para valores exactos en el contenido de una variable, y usar `default` como el análogo al `else`. Los `break` sirven para frenar la ejecución del código y salir del `switch`. La contraparte de no poner un `break` es que se ejecutaría automáticamente el caso siguiente. En base a esto, se puede deducir que el último `break` es optativo.

1.3.2. Sentencia While

La sentencia `while` es una estructura de control de flujo que se ejecutará mientras una condición dada es verdadera. De esta forma, necesita que en algún momento se llegue a una *condición de corte* o de lo contrario,

se quedará en un ciclo infinito. Es necesario que la condición pueda variar, dado que si siempre es `true`, es equivalente a decir que se encuentra en un ciclo infinito. Para introducirlo, vamos a ver la estructura más básica que podemos escribir con ciclos es la de un *contador*:

```
392 int i = 0;
393 int a = 1;
394 while(i < 10) {
395     a = a * 2; // Instrucciones arbitrarias
396     i++;
397 }
```

En este ejemplo, el contador es un ciclo que se ejecuta 10 veces, obteniendo así el resultado de 2^{10} dentro de la variable `a`. Distinguimos un contador a través de su variable contadora normalmente nombrada *i*, *j*, *k*, o simplemente *count*. El único propósito de esta variable es que el ciclo haga tantas *iteraciones* como queremos (en este caso 10).

Si la condición es falsa desde un inicio, nunca se ejecutará el ciclo. Sin embargo, existen ocasiones como en las entradas de datos (se verá en secciones posteriores) donde se requiere que se ejecute el bloque de código que está dentro del ciclo al menos una vez. Para esto usamos la sentencia `do-while`, de la siguiente forma:

```
398 int i = 1;
399 int a = 1;
400 do {
401     a = a * 2; // Instrucciones arbitrarias
402     i++;
403 } while(i < 10);
```

Notar que luego del `while` tenemos un `;`. Además, de tener una sola instrucción, en `while` las llaves son optativas (aunque esto es considerado una mala práctica).

1.3.3. Sentencia For

La sentencia `for` es una estructura de control de flujo de la forma `for (inicialización; condición; expresiónDeIncremento) bloqueDeCódigo;`. De esta forma, podemos reescribir el código anterior de la forma:

```
404 int i;
405 int a = 1;
406 for(i = 0; i < 10; i++) {
407     a = a * 2;
408 }
```

Es importante notar que:

- La instrucción de inicialización se ejecuta una única vez al comenzar el ciclo.
- La instrucción de condición se ejecuta antes de comenzar cada iteración. Si esta da `false` desde un inicio, entonces no se ejecutará el ciclo.
- La instrucción de incremento se ejecuta luego de cada iteración.

Cabe destacar que la *instrucción vacía* es válida y entonces podríamos cambiar el código anterior a la siguiente forma:

```
409 int i = 0;
410 int a = 1;
411 for(; i < 10; i++) {
412     a = a * 2;
413 }
```

Incluso generar ciclos infinitos con lo siguiente:

```

414 int i = 0;
415 int a = 1;
416 for(; i < 10;) {
417     a = a * 2;
418 }

419 int i = 0; // innecesario
420 int a = 1;
421 for(;;) {
422     a = a * 2;
423 }
```

Esta característica es muy importante dado que una estructura como la del ciclo while tiene la instrucción de condición de forma obligatoria.

Por prolíjidad, usaremos las letras *i*, *j*, *k*, dado que los ciclos nos permitirán recorrer estructuras matriciales en varias dimensiones, y el concepto de índices lo traemos del álgebra lineal.

Al igual que las sentencias if, else y while, de tener una sola instrucción en su bloque de código, entonces las llaves son optativas pero se considera una mala práctica no colocarlas.

1.3.4. Break y Continue

El caso de do-while y while muestran casos donde queremos ejecutar código justo antes o después de *evaluar* una condición. Pero, ¿cuál sería la forma de lograr tener una condición en el medio del código?. Podemos usar el concepto de flag, el cual es usar un booleano como marca de agua y usarlo para cortar un ciclo. Por ejemplo:

```

424 int i = 0;
425 int a = 0;
426 int b = 1;
427 int result = 1;
428 boolean flag = true;
429 while(i < 100 && flag) {
430     a = b;
431     b = result;
432     result = a + b;
433     if(result % 17) {
434         flag = false;
435     }
436 }
```

El ejemplo anterior, calcula el último elemento de la serie de Fibonacci que se encuentre lo más cerca del tope del ciclo, siempre que no sea múltiplo de 17. Una forma de evitar usar flag, es que podemos usar la palabra reservada break, que permite cortar un ciclo cuando la instrucción es ejecutada (análogo a lo que sucedía en la sentencia switch):

```

437 int i = 0;
438 int a = 0;
439 int b = 1;
440 int result = 1;
441 while(i < 100) {
442     a = b;
443     b = result;
444     result = a + b;
445     if(result % 17) {
446         break;
447     }
448 }
```

Es más, si hubiésemos tenido instrucciones por debajo del `if`, necesariamente hubiésemos tenido que usar un `else`, pero con el `break` leyéndose dentro del `if`, el `else` se vuelve optativo dado que de todas formas el código no se hubiese ejecutado. Ejemplo:

```
449 int i = 0;
450 int a = 0;
451 int b = 1;
452 int result = 1;
453 int total = 0;
454 boolean flag = true;
455 while(i < 100 && flag) {
456     a = b;
457     b = result;
458     result = a + b;
459     if(result % 17) {
460         flag = false;
461     } else {
462         total = total + result;
463     }
464 }
```

Se puede escribir como:

```
465 int i = 0;
466 int a = 0;
467 int b = 1;
468 int result = 1;
469 int total = 0;
470 while(i < 100) {
471     a = b;
472     b = result;
473     result = a + b;
474     if(result % 17) {
475         break;
476     }
477     total = total + result;
478 }
```

Ahora, ¿cómo podemos cambiar el código para que si el elemento analizado es múltiplo de 17 no corte el ciclo sino que lo ignore en la suma? Si bien existen mejores formas del siguiente código, nos permite presentar a la palabra reservada `continue`, la cual corta el bloque de código en ejecución y salta a la siguiente iteración:

```
479 int i = 0;
480 int a = 0;
481 int b = 1;
482 int result = 1;
483 int total = 0;
484 while(i < 100) {
485     a = b;
486     b = result;
487     result = a + b;
488     if(res % 17) {
489         continue;
490     }
491     total = total + result;
492 }
```

Tanto `break` como `continue` pueden ser usados en ciclos `while`, `do-while` y `for`.

1.3.5. Scopes

Los scopes (o alcance) son muy importantes en Java. Sin embargo, con lo visto hasta el momento, solo nos va a interesar el *scope local*. Esto se refiere al *ámbito de las variables*. Una variable se considera *viva* mientras su espacio esté reservado en memoria y *muerta* cuando deja de hacerlo. En Java, una variable está viva mientras el bloque de código donde se creó siga ejecutándose. Por ejemplo:

```
493 float x = 3.14f;
494 double z = 0;
495 if(x > 3) {
496     float y = 2.71f;
497     z = y + x;
498 }
499 // Siguen existiendo 'x' y 'z', pero no 'y'
```

Este código nos muestra además que bloques de código que contienen otros bloques de código le transfieren la posibilidad de modificar una variable a un bloque más interno. En otras palabras, existe una jerarquía de bloques. Formalmente, define un *contexto* Γ que contiene las variables de nuestro código. En este contexto, las variables no pueden repetirse, pero como las variables viven y mueren, entonces puedo considerar esta restricción únicamente a las variables vivas. De esta forma, el siguiente es un código válido:

```
500 float x = 3.14f;
501 double z = 0;
502 if(x > 3) {
503     float y = 2.71f;
504     z = y + x;
505 }
506 if(x < 3) {
507     float y = 1.61f;
508     z = y - x;
509 }
```

En este caso, no hay conflicto con la variable *y*.

Un caso especial, es el de crear una variable que solo exista para el bloque de código de un ciclo *for* y luego sea desecharla. Para hacerlo, creamos la variable en la instrucción de inicialización:

```
510 int a = 1;
511 for(int i = 0; i < 10; i++) {
512     a = a * 2;
513 }
514 // La variable 'a' sigue existiendo pero 'i' no
```

Si hubiésemos creado la variable dentro del ciclo, en cada iteración esta variable se crea y muere, y podría costarnos un ciclo infinito. Si hubiésemos creado la variable por fuera, quedaría ocupando espacio en la memoria luego de finalizar el ciclo. Esta opción es muy eficiente cuando queremos desechar la variable.

1.4. Otros conceptos importantes

1.4.1. Operaciones simplificadas

Las operaciones aritméticas básicas pueden simplificarse en el caso de que tengan una estructura de la forma *variable = variable operacion valor*. Veamos ejemplos:

```
515 int a = 10;
516 int b = 5;
517
518 a += b; // a = 15
519 a -= b; // a = 10
520 a *= b; // a = 50
```

```
521 | a /= b; // a = 2
522 | a %= b; // a = 0
```

1.4.2. Función main

La función main, dentro de una clase principal, será la que permita ejecutar nuestro código. En la sección de programación funcional se verá más a detalle como se comportan las funciones en Java. De momento, el código que necesitemos probar, deberá estar dentro de esta función, con el formato siguiente:

```
523 public class App {
524
525     public static void main(String[] args) {
526         // Código
527     }
528
529 }
```

Existen varias formas de escribir el main en Java, pero de momento usaremos esta hasta entender parte por parte que significa cada palabra. Esta *clase* deberá tener el mismo archivo que donde se encuentra, en este caso será App.java. El nombre App suele elegirse por prolijidad pero es común usar también nombres como Application, Main, o que incluya el nombre del *framework* donde se este programando (por ejemplo SpringBootApplication). También es común que se utilice el nombre del proyecto como nombre de esta clase principal.

Nuestra aplicación puede tener varias funciones main, pero en diferentes archivos. Esta función debe encontrarse en el *paquete* raíz del proyecto.

1.4.3. String

String es un tipo de dato no primitivo que permite almacenar cadenas de caracteres. Estas cadenas deben estar escritas entre comillas dobles, a diferencia de los caracteres que utilizan comillas simples.

```
530 String name = "Isaac";
531 String lastName = "Newton";
532
533 String empty = ""; // Conocida como string vacía
```

La operación que permite juntar dos strings en una sola, se llama *concatenación*. Usaremos la estructura variable + variable. Si al menos una de las variables es de tipo String, entonces toda la expresión se volverá una string.

```
534 String name = "Isaac";
535 String lastName = "Newton";
536 String fullName = lastName + ", " + name;
537
538 int n = 100;
539 String number = 100 + "";
```

Como las comillas dobles indican el principio y fin de una string, ¿cómo podríamos usar una comilla doble dentro de una string? Este concepto se conoce como *escapear* un carácter, y usa la contrabarra. Existen varios caracteres que necesitamos escapar:

```
540 String text = "\"Hello\" contains five characters"; // Escapeo de comillas
541 String text2 = "It's raining today"; // No escapeo comillas simples
542 String path = "C:\\Users\\User\\Documents\\file.txt"; // Escapeo barras
543
544 String hello = "Hello\nWorld"; // \n es un salto de linea
545 String hello2 = "Hello\tWorld"; // \t es una tabulacion
546 String hello3 = "Hello\rWorld"; // \r es un retorno de carro
```

```
547 String string = "\u00A9 Copyright"; // Caracteres Unicode
```

En la sección de *Programación orientada a objetos*, se verá que los objetos tienen un operador punto, el cual nos permite acceder al comportamiento de estos objetos. Usando una sintaxis de punto, podemos hacer algunas cosas útiles con las strings:

```
549 String original = "My turtle Flash is angry";
550
551 int length = original.length();
552 String withLowerCase = original.toLowerCase(Locale.ROOT);
553 String withUpperCase = original.toUpperCase(Locale.ROOT);
554 // Locale.ROOT es opcional
555
556 // Quitar espacios de los bordes
557 String trimmed = original.trim();
558
559 // Quitar caracteres blancos de los bordes
560 String trimmed2 = original.strip();
561 // stripLeading lo hace solo a izquierda
562 // stripTrailing lo hace solo a derecha
563
564 String repeated = original.repeat(2);
565
566 String withOtherChar = original.replace(' ', '@');
567 String withOtherSubstring = original.replace("turtle", "tortoise");
568 String withOtherSubstring2 = original.replaceAll("Flash", "Speedy");
569 String withOtherSubstring3 = original.substring(5); // Caracteres de [0, 5)
570 String withOtherSubstring4 = original.substring(2, 5); // y de [2, 5)
571
572 char c = original.charAt(0); // Primer caracter
573
574 boolean p = original.isEmpty(); // true si original es ""
575 boolean q = original.isBlank(); // true si solo tiene espacios en blanco
576 boolean r = original.endsWith("angry");
577 boolean s = original.startsWith("angry");
578
579 boolean equals = original.equals("Test");
580 boolean equals2 = original.equalsIgnoreCase("Test");
581 boolean match = original.matches("Flash");
582 boolean contains = original.contains("is");
583
584 // menor a 0 si es menor, 0 si es igual, mayor a 0 si es mayor
585 int difference = original.compareTo("ABC");
586
587 int index = original.indexOf('a');
588 int index2 = original.lastIndexOf('a');
589 String intern = original.intern();
590 String other = original.concat("Test"); // Lo mismo que el operador +
591
592 // Quita tabulaciones en strings multilineas
593 String other2 = original.translateEscapes();
```

En el código anterior, se hace referencia a caracteres blancos, los cuales corresponden a espacios, tabulaciones, retorno de carro, saltos de línea o simplemente espacios en blanco en Unicode. También se mencionan las Strings multilinea. Acá un ejemplo de como son:

```
594 String original = """
595             My Flash
```

```

596     turtle
597     is angry
598     """;
599
600 // Es equivalente a
601
602 String original2 = "My Flash" + "\n"
603     "turtle " + "\n"
604     "is angry";

```

También se mencionan índices. En general el primer carácter de una string se encuentra en la posición 0, el siguiente en la posición 1, etc. La igualdad entre strings es un poco más complicada. Para esto hay que llegar a objetos, pero de momento, es importante saber que dos strings pueden ser iguales en contenido, pero al ser creadas en instancias distintas pueden considerarse distintas. Entonces, la igualdad carácter a carácter depende del método `equals`. El método `intern` se puede entonces usar de la siguiente forma:

```

605 String str1 = "Hello";
606 String str2 = new String("Hello");
607 // Esta sintaxis permite crear una string de mismo contenido
608 // pero diferente igualdad
609 String str3 = str2.intern();
610
611 System.out.println(str1 == str2); // false
612 System.out.println(str1 == str3); // true

```

Las strings son inmutables, eso quiere decir que la variable que usamos para obtener las nuevas strings (`original`) no ha cambiado de valor luego de usar estos métodos. Es decir, no modifican la variable, sino que devuelven una copia modificada.

Con estos métodos, podemos recorrer una String carácter a carácter y generar una nueva con cambios modificados. Por ejemplo, si queremos eliminar los espacios de una string, podemos hacer lo siguiente:

```

613 String text = "Lorem ipsum dolor sit amet";
614 String result = "";
615 for(int i = 0; i < text.length(); i++) {
616     result += text.charAt(i) == ' ' ? "" : text.charAt(i);
617 }

```

Podemos notar que el método `length()` es de tipo `int`. Con esto podemos deducir que la longitud de una string es de, por lo menos, $2^{31} - 1$ caracteres. Dado que cada carácter pesa 2 bytes, entonces una string puede pesar hasta $2^{31} - 2 \cdot 2^{31} = \frac{2^{32}}{2}$. Por otro lado 2^{32} bytes es aproximadamente 4.29 Gb.

1.4.4. Math

`Math` nos pone a disposición funciones matemáticas que son frecuentes en muchos algoritmos. Por ejemplo, en la tabla siguiente se ven las funciones trigonométricas:

Método	Descripción
double sin(double a)	Devuelve el seno del ángulo especificado en radianes.
double cos(double a)	Devuelve el coseno del ángulo especificado en radianes.
double tan(double a)	Devuelve la tangente del ángulo especificado en radianes.
double asin(double a)	Devuelve el arcoseno del valor especificado, en radianes.
double acos(double a)	Devuelve el arcocoseno del valor especificado, en radianes.
double atan(double a)	Devuelve el arcotangente del valor especificado, en radianes.
double atan2(double y, double x)	Devuelve el arcotangente de las coordenadas (x, y).
double sinh(double a)	Devuelve el seno hiperbólico del ángulo especificado en radianes.
double cosh(double a)	Devuelve el coseno hiperbólico del ángulo especificado en radianes.
double tanh(double a)	Devuelve la tangente hiperbólica del ángulo especificado en radianes.
double asinh(double a)	Devuelve el arcoseno hiperbólico del valor especificado, en radianes.
double acosh(double a)	Devuelve el arcocoseno hiperbólico del valor especificado, en radianes.
double atanh(double a)	Devuelve el arcotangente hiperbólico del valor especificado, en radianes.

Podemos, por ejemplo, calcular el mayor entre dos números con la función `max`, redondear con `floor` o calcular una raíz cuadrada con `sqrt`.

Con el método `pow` podemos elevar una base a una potencia, mientras que no tenemos un método para una raíz n-ésima. Podemos por ejemplo, calcular una raíz cuarta, calculando dos veces seguidas una raíz cuadrada. Pero una forma general, puede ser la siguiente:

```
618 int index = 3;
619 double value = 8;
620
621 // Para calcular la raiz cubica de 8
622 double result = Math.pow(value, 1/index);
```

En este caso usamos la propiedad $\sqrt[n]{a} = a^{\frac{1}{n}}$. De la misma forma, se suelen usar series de Taylor para dar una aproximación a funciones con las que no contamos.

1.4.5. Entrada y salida de datos

Para la entrada de datos, debemos usar `Scanner`. Un scanner estará atento a una instrucción de leer por consola. Este sistema debe *cerrarse* al terminar de usarse porque de no hacerlo, aún sin usar el scanner esto puede estar usando recursos de la computadora. Ejemplo:

```
623 import java.util.Scanner;
624
625 public class Main {
626     public static void main(String[] args) {
627         Scanner sc = new Scanner(System.in);
628         System.out.print("Introduce un numero entero: ");
629         int num = sc.nextInt();
630         System.out.println("El numero introducido es: " + num);
631         sc.close();
632     }
633 }
```

Algunos métodos útiles se ven en la tabla de a continuación.

Podemos tener más precisión en el tipo de dato usando métodos como `nextByte()`, `nextLong()`, `nextFloat()`, entre otros.

Por otro lado, podemos imprimir en pantalla con strings normales o parametrizadas.

```
634 String subject = "Programacion";
635 int count = 2;
636 String description = "1C 2023";
```

Método	Descripción
nextInt ()	Lee un número entero
nextDouble ()	Lee un número decimal
nextBoolean ()	Lee un valor booleano (verdadero o falso)
next ()	Lee una cadena de caracteres (hasta el siguiente espacio en blanco)
nextLine ()	Lee una línea completa de texto (hasta el final de la línea)

Cuadro 1.16: Métodos de la clase Scanner para leer diferentes tipos de datos

```
637 | String text = String.format("%s %d, %s", subject, count, description);
638 | System.out.println(text);
```

Esto imprimirá en pantalla Programacion 2, 1C 2023. Si la cantidad de parámetros no coincide, puede tirar error. Además % se usa para expresar un parámetro. Por ejemplo, %s expresa que el parámetro es de tipo string y %d que es numérico de tipo entero. Si queremos un número decimal, podemos indicar cuantos decimales queremos con %.2f (en este caso 2 decimales). Por último, para indicar que es de tipo booleano podemos usar %b.

String.format(string) permite parametrizar una string y guardar el resultado en una variable. Sin embargo si esta variable tiene como único propósito imprimirse en pantalla, se puede simplificar un System.out.print(String.format(string, param)) por un System.out.printf(string, param).

1.4.6. Casteo explícito

El casteo de un tipo de dato a otro, puede hacerse de forma explícita. Por ejemplo:

```
639 | int n = 2;
640 | double x = (double) n;
```

Con esto, le indicamos del lado derecho de la asignación, que el valor a asignar se calcula como la transformación de la variable n al tipo double. Como vimos anteriormente, el casteo implícito permite esto, entonces hacerlo es optativo. Pero existen casos donde el casteo no se puede omitir. Si hacemos el proceso inverso el casteo se vuelve obligatorio:

```
641 | double x = 2.1;
642 | int n = (int) x;
```

En este caso, double tiene que perder información para poder convertirse en un int, y como esto puede traer errores, Java no lo hará a no ser que se lo indiquemos. De lo contrario esta asignación dará error. Convertir un double a un int significa perder los decimales (da el mismo efecto que redondear hacia abajo, pero además cambiando el tipo de dato). Adicionalmente, si el número era muy grande se perderán sus bits significativos por no ser de tipo long.

Cuando ni siquiera existe la pérdida de información, por ejemplo cuando no es posible una conversión, entonces Java dará error. Por ejemplo no podemos convertir un boolean a un byte.

Es válido intentar castear incluso con datos no primitivos. En POO se verá la herencia. Cuando un a clase hija se intente castear a una clase padre o a una interfaz de la que implemente, entonces el casteo será válido. De lo contrario, Java dará error. La diferencia es que con datos primitivos el error es en *tiempo de compilación*, es decir, antes de ejecutar el programa. Mientras que en los no primitivos el error será en *tiempo de ejecución*, es decir, solo sabremos del error cuando el programa ejecute la instrucción.

1.5. Ejercicios

Ejercicio 1.5.1: Intercambio de valores

Sean $a, b \in \mathbb{Z}$, variables, realizar un intercambio de valores sin utilizar estructuras ni variables auxiliares.

Ejercicio 1.5.2: Mayor

Sean $a, b, c \in \mathbb{R}$, variables, crear un programa que calcule el mayor.

† Sean $a, b, c \in \mathbb{R}$, variables, crear un programa que calcule el mayor estricto, y en caso de no existir devolver -1 . En caso de usar condicional, solo se permite usar la forma corta de la sentencia if.

Ejercicio 1.5.3: Figuras geométricas

Sean $a, b, c, d \in \mathbb{R}$, variables que representan los posibles lados de un cuadrado, crear un programa que guarde en una variable true solo si pueden representar los lados de un cuadrado.

† Sean $a, b, c \in \mathbb{R}$, variables que representan los posibles lados de un triángulo, crear una programa que guarde en una variable true solo si pueden representar los lados de un triángulo.

Ejercicio 1.5.4: Raíz n-ésima

Sea $a \in \mathbb{R}$, variable, crear un programa que calcule su raíz n-ésima de a utilizando la librería Math.

† Sea $a \in \mathbb{R}$, variable, crear un programa que calcule su raíz n-ésima de a utilizando Math.sqrt(a) y recursividad.

Ejercicio 1.5.5: Inverso en dígitos

Sea $a \in \mathbb{N}$, variable, crear un programa que invierta sus dígitos.

† Sea $a \in \mathbb{N}$, variable, crear un programa que invierta sus dígitos utilizando únicamente operaciones matemáticas.

Capítulo 2

Arreglos, matrices y algoritmos

2.1. Arreglos

Un arreglo es una estructura de dato que permite almacenar datos, de forma ordenada, a través de una secuencia de índices (empezando desde el 0). Los arreglos tienen una *longitud* fija, es decir, no se puede alterar una vez inicializada la variable.

2.1.1. Crear un arreglo con los elementos definidos

Formato: tipo[] variable = valor, valor2, valor3, ...;. Ejemplo:

```
643 import java.util.Arrays;  
644  
645 public class Main  
646 {  
647     public static void main(String[] args) {  
648         int[] array = {1, 1, 2, 3, 5, 8, 13, 21, 34};  
649         System.out.println(Arrays.toString(array));  
650     }  
651 }
```

Si imprimimos en pantalla el arreglo anterior nos mostrará su referencia a memoria. Para evitarlo, podemos usar `Arrays.toString(array)`. Luego, la impresión en pantalla mostrará: [1, 1, 2, 3, 5, 8, 13, 21, 34].

Es posible dejar saltos de línea entre los elementos dado que la instrucción únicamente finaliza con el punto y coma.

Como forma optativa, se puede usar el formato: tipo[] variable = new tipo[] valor, valor2, valor3, ...;. En el ejemplo anterior tendremos:

```
652 int[] array = new int[]{1, 1, 2, 3, 5, 8, 13, 21, 34};
```

2.1.2. Crear un arreglo sin los elementos definidos

Formato: tipo[] variable = new tipo[longitud];.

```
653 int[] array = new int[9]; // Defino la longitud  
654  
655 // Asigno valores:  
656 array[0] = 1; // Empiezo desde el indice 0  
657 array[1] = 1;  
658 array[2] = 2;  
659 array[3] = 3;
```

```

660 | array[4] = 5;
661 | array[5] = 8;
662 | array[6] = 13;
663 | array[7] = 21;
664 | array[8] = 34;

```

2.1.3. Arreglo vacío

El arreglo vacío tiene longitud 0. Entonces, para crearlo, tenemos diferentes opciones:

```

665 | int[] array = new int[0];
666 | double[] array2 = {};
667 | String[] array3 = new String[] {};

```

2.1.4. Acceso y manipulación de un arreglo

Cada elemento de un arreglo tiene una posición (o índice) empezando desde el 0. Para leer un valor en particular, usamos la sintaxis `array[índice]`.

```

668 | int[] array = {1, 1, 2, 3, 5, 8, 13};
669 | System.out.println(array[0]);

```

A su vez, podemos usar esto como variable y reemplazar el valor que contiene:

```

670 | array[0] = 0; // array = [0, 1, 2, 3, 5, 8, 13]

```

Podemos obtener el tamaño de un array con la propiedad `length`:

```

671 | array[array.length - 1] = 0; // array = [0, 1, 2, 3, 5, 8, 0]

```

En el ejemplo anterior, si el array tiene tamaño 7, entonces sus índices van de 0 a 6, y entonces, la posición 7 (su longitud) no existe. Corregimos esto con un `-1`. En el caso que el índice que usemos esté fuera del rango, Java dará el error `IndexOutOfBoundsException`.

2.1.5. Iteración sobre arreglos

El siguiente ejemplo muestra como iterar sobre un arreglo e imprimirla en pantalla:

```

672 | int[] array = {1, 1, 2, 3, 5, 8, 13};
673 | System.out.print("[");
674 | if(array.length == 0) {
675 |     System.out.print("]");
676 |     return;
677 | }
678 |
679 | System.out.print(array[0]);
680 | if(array.length == 1) {
681 |     System.out.print("]");
682 |     return;
683 | }
684 |
685 | for(int i = 1; i < array.length; i++) {
686 |     System.out.print(", " + array[i]);
687 | }
688 |
689 | System.out.print("]");

```

2.1.6. Contador

Un contador es una variable que se incrementa en una cantidad fija de unidades cuando se cumple una condición, o simplemente contar todos los casos de una estructura sin excepción.

Ejemplo de contador de números pares en un arreglo:

```
690 int[] array = {1, 1, 2, 3, 5, 8, 13};
691
692 int count = 0;
693 for(int i = 0; i < array.length; i++) {
694     if(array[i] % 2 == 0) {
695         count++;
696     }
697 }
698
699 System.out.println("Total: " + count);
```

2.1.7. Acumulador

En lugar de contar los elementos, un acumulador realiza una operación sobre todos los elementos que queremos acumular. Por ejemplo, si tenemos un arreglo de enteros, y tomamos la operación suma, tenemos un *sumador* o *totalizador*:

```
700 int[] array = {1, 1, 2, 3, 5, 8, 13};
701
702 int total = 0;
703 for(int i = 0; i < array.length; i++) {
704     total += array[i];
705 }
706
707 System.out.println("Total: " + total);
```

Sin embargo, esto no es necesario que se limite a solo este caso. Podemos tener de elementos a los conjuntos y tomar la operación unión, o incluso un arreglo de arreglos y crear un arreglo que unifique todos los datos mediante una operación *unificar*.

2.1.8. Algoritmos de búsqueda

A continuación, la implementación en Java de algunos algoritmos de búsqueda.

Encontrar el mínimo o máximo

La estrategia de este algoritmo es tomar *candidatos*. Un candidato es un valor que tiene más chances hasta un momento dado de ser el resultado final de un algoritmo. En el siguiente algoritmo, nuestra *variable candidata* es min.

```
708 public static int findMin(int[] array) {
709     int min = array[0];
710     for (int i = 1; i < array.length; i++) {
711         if (array[i] < min) {
712             min = array[i];
713         }
714     }
715     return min;
716 }
```

Al final del proceso, el algoritmo nos devuelve el mínimo del arreglo. Podemos modificarlo para obtener el máximo:

```

717 public static int findMax(int[] array) {
718     int max = array[0];
719     for (int i = 1; i < array.length; i++) {
720         if (array[i] > max) {
721             max = array[i];
722         }
723     }
724     return max;
725 }
```

Búsqueda secuencial

La búsqueda secuencial consiste en comparar elemento a elemento de un arreglo hasta encontrar el elemento buscado.

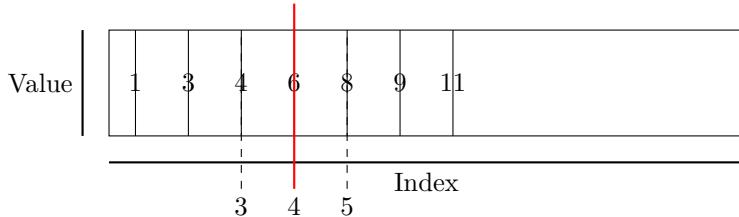
```

726 public class SequentialSearch {
727     public static int search(int[] array, int x) {
728         int n = array.length;
729         for (int i = 0; i < n; i++) {
730             if (array[i] == x) {
731                 return i; // Devuelve la posición del elemento encontrado
732             }
733         }
734         return -1; // Si no se encuentra el elemento, devuelve -1
735     }
736
737     public static void main(String[] args) {
738         int[] array = {2, 3, 1, 7, 5, 6, 4};
739         int n = 5;
740         int index = search(array, n);
741         if (index == -1) {
742             System.out.println("El elemento no se encuentra en el arreglo");
743         } else {
744             System.out.println("El elemento " + n + " se encuentra en la posición " +
745                               index);
746         }
747     }
}
```

Devolvemos `-1` cuando queremos marcar un error (lo hacemos así por razones históricas).

Búsqueda binaria

Supongamos el arreglo `[1, 1, 2, 3, 5, 8, 13]`, nos posicionamos sobre el elemento medio. Luego, si el arreglo está ordenado, podemos considerar comparar este elemento con el elemento buscado. Si el elemento es el que buscamos, termina nuestro algoritmo. En cambio, si es mayor, solo hay que buscar en el lado derecho o de lo contrario en el izquierdo. En estos casos, estamos quitando de nuestra búsqueda la mitad de los elementos en un solo paso, lo cual es muy buena ventaja sobre un algoritmo secuencial.



Se requiere que el array esté ordenado.

```

748 public class BinarySearch {
749
750     public static int binarySearch(int[] array, int key) {
751         int left = 0;
752         int right = array.length - 1;
753         while (left <= right) {
754             int mid = left + (right - left) / 2;
755             if (array[mid] == key) {
756                 return mid;
757             } else if (array[mid] < key) {
758                 left = mid + 1;
759             } else {
760                 right = mid - 1;
761             }
762         }
763         return -1;
764     }
765
766     public static void main(String[] args) {
767         int[] array = {1, 3, 4, 6, 7, 8, 9, 11};
768         int key = 7;
769         int result = binarySearch(array, key);
770         if (result == -1) {
771             System.out.println("El elemento no esta presente");
772         } else {
773             System.out.println("Posicion del elemento: " + result);
774         }
775     }
776 }
```

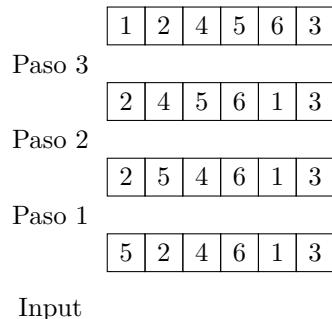
El algoritmo de búsqueda binaria parte en 2 el arreglo. Una modificación posible es partirla en más unidades para obtener una mayor eficiencia para grandes cantidades de datos. Repasar este razonamiento cuando se estudie el *cálculo de complejidad computacional*.

2.1.9. Algoritmos de ordenamiento

Para aplicar la búsqueda binaria es necesario tener el arreglo ordenado. A continuación, la implementación en Java de algunos algoritmos de ordenamiento.

Ordenamiento por inserción

De abajo hacia arriba, el siguiente gráfico muestra un paso a paso de este algoritmo. Tomamos un elemento, consideramos que a la izquierda está ordenado, y lo colocamos en la posición que corresponda pero dentro de ese rango. A la derecha, lo que nos falta por ordenar. Entonces, recorremos el arreglo de izquierda a derecha.



```

777 public class InsertionSort {
778
779     public static void sort(int[] array) {
780         for (int i = 1; i < array.length; i++) {
781             int key = array[i];
782             int j = i - 1;
783             while (j >= 0 && array[j] > key) {
784                 array[j + 1] = array[j];
785                 j--;
786             }
787             array[j + 1] = key;
788         }
789     }
790
791     public static void main(String[] args) {
792         int[] array = {9, 3, 1, 4, 8, 7, 5};
793         System.out.println("Arreglo sin ordenar: " + Arrays.toString(array));
794         sort(array);
795         System.out.println("Arreglo ordenado: " + Arrays.toString(array));
796     }
797 }
```

Ordenamiento de burbuja

El ordenamiento de burbuja consiste en tomar elementos de a pares. Si estos elementos estan ordenados, entonces no se hace nada. Si estan desordenados, se intercambian. La estrategia consiste en tomar el elemento de la primera posición, y buscar si hay algún elemento menor a este. Si lo hay se intercambia. Una vez que en esta posición queda el menor de los elementos, entonces se avanza con el siguiente.

En la figura siguiente, en gris el elemento ya ordenado y un ejemplo de pasos de intercambio para ordenar:



Arreglo inicial

Arreglo final

```

798 public class BubbleSort {
799
800     public static void sort(int[] array) {
801         int n = array.length;
802         for (int i = 0; i < n - 1; i++) {
803             for (int j = 0; j < n - i - 1; j++) {
804                 if (array[j] > array[j + 1]) {
805                     int temp = array[j];
806                     array[j] = array[j + 1];
807                     array[j + 1] = temp;
808                 }
809             }
810 }
```

```

810         }
811     }
812
813     public static void main(String[] args) {
814         int[] array = {9, 3, 1, 4, 8, 7, 5};
815         System.out.println("Arreglo sin ordenar: " + Arrays.toString(array));
816         sort(array);
817         System.out.println("Arreglo ordenado: " + Arrays.toString(array));
818     }
819 }
```

Ordenamiento de burbuja bidireccional

Es similar al unidireccional, solo que se procede a dejar los extremos ordenados en primer lugar.

```

820 public class BidirectionalBubbleSort {
821
822     public static void sort(int[] array) {
823         int n = array.length;
824         int left = 0;
825         int right = n - 1;
826         while (left < right) {
827             for (int i = left; i < right; i++) {
828                 if (array[i] > array[i + 1]) {
829                     int temp = array[i];
830                     array[i] = array[i + 1];
831                     array[i + 1] = temp;
832                 }
833             }
834             right--;
835             for (int i = right; i > left; i--) {
836                 if (array[i] < array[i - 1]) {
837                     int temp = array[i];
838                     array[i] = array[i - 1];
839                     array[i - 1] = temp;
840                 }
841             }
842             left++;
843         }
844     }
845
846     public static void main(String[] args) {
847         int[] array = {9, 3, 1, 4, 8, 7, 5};
848         System.out.println("Arreglo sin ordenar: " + Arrays.toString(array));
849         sort(array);
850         System.out.println("Arreglo ordenado: " + Arrays.toString(array));
851     }
852 }
```

Ordenamiento por casilleros

Solo se puede aplicar a un arreglo de enteros. La idea es dividir números del arreglo en *buckets* que representan rangos, y luego solo hay que ordenar esos rangos. Aunque la idea es sencilla, en Java no es tan eficiente llevarla adelante tal cual, vamos a quedarnos con el concepto de rango y considerar a cada casilla como un bucket (es decir, un solo elemento por bucket). Como no se respeta el concepto de bucket como tal, vamos a llamarlos *box*.

```

853 public class BucketSort {
854
855     public static void sort(int[] array) {
856
857         // Determinar el valor maximo y minimo del arreglo
858         int max = array[0];
859         int min = array[0];
860         for (int i = 1; i < array.length; i++) {
861             if (array[i] > max) {
862                 max = array[i];
863             }
864             if (array[i] < min) {
865                 min = array[i];
866             }
867         }
868
869         // Crear los casilleros
870         int[] boxes = new int[max - min + 1];
871
872         // Contar la frecuencia de cada valor en el arreglo y asignarlo al casillero
873         // correspondiente
874         for (int i = 0; i < array.length; i++) {
875             boxes[array[i] - min]++;
876         }
877
878         // Reordenar el arreglo utilizando los valores en los casilleros
879         int index = 0;
880         for (int i = 0; i < boxes.length; i++) {
881             for (int j = 0; j < boxes[i]; j++) {
882                 array[index] = i + min;
883                 index++;
884             }
885         }
886
887     public static void main(String[] args) {
888         int[] array = {9, 3, 1, 4, 8, 7, 5};
889         System.out.println("Arreglo sin ordenar: " + Arrays.toString(array));
890         sort(array);
891         System.out.println("Arreglo ordenado: " + Arrays.toString(array));
892     }
893 }
```

Ordenamiento por cuentas

Es una modificación del algoritmo de ordenamiento por casilleros. Se debe prestar atención a la parte final del algoritmo.

```

894 public class CountingSort {
895
896     public static void sort(int[] array) {
897         // Encontrar el valor maximo y minimo del arreglo
898         int max = array[0];
899         int min = array[0];
900         for (int i = 1; i < array.length; i++) {
901             if (array[i] > max) {
902                 max = array[i];
903             }
904         }
905 }
```

```

904         if (array[i] < min) {
905             min = array[i];
906         }
907     }
908
909     // Crear el arreglo de cuentas
910     int[] countArray = new int[max - min + 1];
911
912     // Contar la frecuencia de cada valor en el arreglo
913     for (int i = 0; i < array.length; i++) {
914         countArray[array[i] - min]++;
915     }
916
917     // Actualizar el arreglo original con los valores ordenados
918     int j = 0;
919     for (int i = min; i <= max; i++) {
920         while (countArray[i - min] > 0) {
921             array[j++] = i;
922             countArray[i - min]--;
923         }
924     }
925 }
926
927 public static void main(String[] args) {
928     int[] array = {9, 3, 1, 4, 8, 7, 5};
929     System.out.println("Arreglo sin ordenar: " + Arrays.toString(array));
930     sort(array);
931     System.out.println("Arreglo ordenado: " + Arrays.toString(array));
932 }
933 }
```

2.1.10. Sub-arreglo

Obtener una parte de un arreglo, dentro de otro arreglo, implica tener dos arreglos de distinto tamaño en la mayoría de los casos. Podemos colocar como tamaño, una expresión que indique lo que necesitamos.

```

934 public static int[] getSubarray(int[] array, int startIndex, int endIndex) {
935     int[] subarray = new int[endIndex - startIndex + 1];
936
937     for (int i = startIndex; i <= endIndex; i++) {
938         subarray[i - startIndex] = array[i];
939     }
940
941     return subarray;
942 }
```

2.2. Matrices

Si el formato de un arreglo es `tipo[]`, entonces, dado que el arreglo también es un tipo de dato, es posible escribir `double[][]` para un arreglo de arreglos. Esto se conoce como arreglo bidimensional, y el concepto es extensible a n -dimensiones.

Cuando la estructura que almacenamos en un arreglo de arreglos tiene una dimensión equivalente para el elemento que almacenamos, entonces es una matriz.

```

943 int[][] arrayOfArrays = {{1, 2, 3}, {4, 6}, {7, 8, 9}};
944 }
```

```
945 | String[][] matrix = {{ "Hola", "Mundo"}, {"Hola", "Java"}, {"Hola", "Programacion 2"}};
```

En el primer caso, `{4, 6}` tiene solo 2 elementos, y entonces se pierde el formato de que los arreglos del arreglo general tengan la misma cantidad de elementos. Entonces, ya no es una matriz. Sin embargo en la segunda estructura, esto se respeta y entonces, cumple.

2.2.1. Recorrer una matriz

Una matriz A tiene la siguiente forma:

$$A = \begin{matrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & a_{m2} & \cdots & a_{mn} \end{matrix}$$

Vamos a obtener un elemento general a_{ij} de la siguiente forma:

```
946 | for (int i = 0; i < matrix.length; i++) {  
947 |     for (int j = 0; j < matrix[i].length; j++) {  
948 |         System.out.print(matrix[i][j] + " ");  
949 |     }  
950 |     System.out.println();  
951 | }
```

2.2.2. Matriz de dimensión cero

Al igual que un arreglo puede crearse vacío, y tener 0 elementos, es posible tener una matriz de 0×0 elementos, y entonces, tener dimensión 0.

```
952 | int [][] matrix;  
953 |  
954 | matrix = new int[0][0];  
955 | matrix = new int[0][];  
956 | matrix = new int[0][1];
```

2.2.3. Matriz nula y matriz identidad

Se deja como ejercicio crear una matriz identidad (ver sección de ejercicios). Una matriz nula, es una matriz donde todos sus elementos son 0. En cambio, una matriz identidad es una matriz *cuadrada* (es decir, tiene dimensión $n \times n$) tal que su diagonal siempre tiene 1 como elemento y el resto de los elementos es 0.

$$I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.2.4. Traza de una matriz

La *traza* de una matriz es igual a la suma de los elementos de su diagonal principal (que va desde arriba a la izquierda, hasta abajo a la derecha).

```
957 | public static int trace(int [][] matrix) {  
958 |     int trace = 0;  
959 |     int n = matrix.length; // para no acceder al valor en cada iteracion  
960 | }
```

```

961     for (int i = 0; i < n; i++) {
962         trace += matrix[i][i];
963     }
964
965     return trace;
966 }
```

2.2.5. Traspuesta de una matriz

$$\text{Si } A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix} \text{ entonces } A^T = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{n,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,m} & a_{2,m} & \cdots & a_{n,m} \end{pmatrix}$$

```

967 public static int[][] transpose(int[][] matrix) {
968     int rows = matrix.length;
969     int columns = matrix[0].length;
970     int[][] transpose = new int[columns][rows];
971
972     for (int i = 0; i < rows; i++) {
973         for (int j = 0; j < columns; j++) {
974             transpose[j][i] = matrix[i][j];
975         }
976     }
977
978     return transpose;
979 }
```

2.3. Ejercicios

Ejercicio 2.3.1: Acumulador

Sea A un array de $a \in \mathbb{Z}$, calcular la suma de sus elementos.

† Sea A un array de cadenas de caracteres, calcular la concatenación de todos sus elementos.

Ejercicio 2.3.2: Rango

Sea A un array de $a \in \mathbb{Z}$, dado un rango $[n, m]$. Imprimir en pantalla los valores del rango válidos. Crear una versión alternativa donde si el rango no es válido se muestre un error en pantalla.

† Sea A un array de $a \in \mathbb{Z}$, dado un rango $[n, m]$. Si el rango excede los índices del array, suponga que el array se extiende infinitamente con copias de el. Por ejemplo, en $[1, 2, 3, 4, 5]$, el elemento en la posición -2 es 4 .

Ejercicio 2.3.3: Reductor

Cree una función que reciba un array de caracteres, y devuelva el carácter mas a la izquierda en el alfabeto.

† Cree una función que reciba un array de caracteres, y devuelva el carácter con menos ocurrencias.

Ejercicio 2.3.4: Filtro

Cree una función que reciba un array de caracteres, y se quede solo con los que estan en mayúscula.
 † Cree una función que reciba un array de cadenas de caracteres, y se quede solo con las que estan escritas en mayúscula.

Ejercicio 2.3.5: Mapa

Cree una función que reciba un array de cadenas de caracteres, y devuelva un array con las longitudes de cada una.
 † Cree una función que reciba un array de cadenas de caracteres, y devuelva un array de booleanos de misma dimensión, donde el booleano que corresponde a la misma posición del array de entrada es true únicamente si la cadena contiene una vocal.

Ejercicio 2.3.6: Matriz identidad

Desarrolle una función que tome como parámetro un $n \in \mathbb{N}$ e imprima en pantalla una matriz de tamaño $n \times n$ de la siguiente forma:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

† Cree un programa que tome dos matrices cuadradas y realice la multiplicación de matrices. Muestre que el elemento neutro de la multiplicación de matrices es la identidad, al menos en dos dimensiones distintas.

Ejercicio 2.3.7: Matriz traspuesta

Desarrolle una función que tome como parámetro una matriz cuadrada e imprima en pantalla su matriz traspuesta.

† Desarrolle una función que calcule la inversa de una matriz de tamaño 2×2 .

Ejercicio 2.3.8: Matriz caracol

Desarrolle una función que tome como parámetro un $n \in \mathbb{N}$ e imprima en pantalla una matriz de tamaño $n \times n$ de la siguiente forma:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 12 & 13 & 14 & 5 \\ 11 & 16 & 15 & 6 \\ 10 & 9 & 8 & 7 \end{pmatrix}$$

† Modifique la función anterior para que la matriz pueda ser rectangular.

Capítulo 3

Programación funcional

3.1. Funciones

Una función, en el sentido matemático, es una *relación* entre dos conjuntos. Por ejemplo, podemos crear una función tal que:

$$f(1) = 2, f(2) = 4, f(3) = 6$$

Esta función solo admite como entrada los valores del conjunto $\{1, 2, 3\}$ y como salida los valores del conjunto $\{2, 4, 6\}$. En este contexto, $f(5)$ no existe, porque recibe un valor que no se encuentra presente en su conjunto de entrada. Por otro lado, tampoco es la única función que podemos crear con estos conjuntos. Por ejemplo,

$$f_2(1) = 2, f_2(2) = 4, f_3(3) = 6$$

también es válido.

Al conjunto de entrada, lo llamamos *dominio* y al conjunto de salida lo llamamos *imagen*. Entonces, en el caso anterior $f : \{1, 2, 3\} \rightarrow \{2, 4, 6\}$, y $f_2 : \{1, 2, 3\} \rightarrow \{2, 4, 6\}$. Podemos generalizar la función y mantener su dominio, tal que $f(x) = 2x, x \in \{1, 2, 3\}$. En este contexto podemos calcular la imagen, pero es usual colocar un conjunto más amplio. Por ejemplo, la función $f_3(x) = x^2$, se suele escribir como $f_2 : \mathbb{R} \rightarrow \mathbb{R}$, siendo que como imagen nunca va a tener un número negativo.

En el aspecto computacional, nos va a suceder lo mismo. Nuestro tipo de dato representa un conjunto de valores posibles. Entonces, podemos tener una función suma que tome dos valores a y b de tipo `int` y devuelva un `int`, siendo que, esta función puede dar error en lugar de dar resultados negativos, acotando así la imagen de la función.

En Java, la función suma se puede definir como

```
980 public static int plus(int a, int b) {  
981     return a + b;  
982 }
```

Aquí, vamos a omitir de momento las palabras reservadas `public` y `static` porque se verán en POO. Seguido a esto, se coloca el tipo de dato que devuelve nuestra función, seguido del nombre y los parámetros. Formalmente, antes de llegar a la llave de apertura, se llama *firma de método*. La sentencia `return` indica el resultado de la función.

Las funciones en programación pueden dar resultados de una forma distinta al concepto matemático. Podemos considerar una salida de datos por pantallas, por ejemplo. En este caso, estas salidas de datos por pantalla se diferencian en que pueden ser múltiples. Una función puede tener varios `return` escritos pero solo uno se ejecutará y dará fin a la ejecución de la función. Pero la existencia de estas permite que en ocasiones, solo se quiera un resultado de este tipo y no se necesite devolver un dato mediante un `return`. Para esto, existen las funciones *vacías*. Para crearlas, debemos indicar como tipo de dato de salida `void`. Si bien esta palabra reservada va en el lugar de un tipo de dato, este no puede ser usado para inicializar variables (aunque conceptualmente podemos crearlo con POO).

```

983 public static void plus(int a, int b) {
984     System.out.println(a + b);
985 }
```

return puede ser usado para salir de una función en cualquier momento, por ejemplo:

```

986 public static void plus(int a, int b) {
987     if(a == 0) {
988         return b;
989     } else if (b == 0) {
990         return a;
991     } else {
992         return a + b;
993     }
994 }
```

En el caso anterior, dado que las instrucciones por debajo de un return no serán ejecutadas, podemos obviarlo:

```

995 public static int plus(int a, int b) {
996     if(a == 0) {
997         return b;
998     }
999     if (b == 0) {
1000         return a;
1001     }
1002     return a + b;
1003 }
```

En funciones de tipo void no es necesario devolver un valor, podemos usar return para cortar una función así como break permite cortar un ciclo:

```

1004 public static void plus(int a, int b) {
1005     if(a == 0) {
1006         System.out.println(b);
1007         return;
1008     }
1009     if (b == 0) {
1010         System.out.println(a);
1011         return;
1012     }
1013     System.out.println(a + b);
1014     return; // No es necesario, no hay nada por debajo
1015 }
```

3.1.1. Reducciones

Podemos decir que cada paso que realiza nuestro código para poder calcular un resultado, es un término. En este aspecto, la función plus anterior, la podemos ver de la siguiente forma: plus(15 + 3, 12 - 2) reduce a plus(18, 12 - 2), que reduce a plus(18, 10) que reduce a 18 + 10 que reduce a 28. Esto último ya no puede reducir, dado que es un *valor*. Este reducción se llama *reducción β*. En general, un programa no puede reducir valores o *errores*. La forma en que leemos los parámetros es importante. Por ejemplo:

```

1016 public static int plus(int a, int b) {
1017     return a + b;
1018 }
1019
```

```

1020 public static void f() {
1021     int a = 8;
1022     System.out.println(plus(a, ++a));
1023 }

```

En este caso, si leemos primero el parámetro de la derecha, la función dará 17, mientras que si leemos el de la izquierda será 16.

También es importante saber si el lenguaje primero reemplaza los parámetros en una expresión que va a devolver o si primero los calcula y luego los calcula como parámetro.

```

1024 public static int times(int a, int b) {
1025     return a * b;
1026 }
1027
1028 public static int g() {
1029     while(true) {
1030         }
1031     return -1;
1032 }
1033
1034
1035 public static void f() {
1036     System.out.println(times(0, g()));
1037 }

```

En el caso anterior, si primero reemplaza, entonces se reduce a la expresión $0 * g()$, y esto se puede deducir como cero sin necesidad de ejecutar g . Pero si primero se calculan los parámetros, entonces esto se quedará en un loop infinito. Java lee de izquierda a derecha los parámetros y los calcula primero.

La *reducción α* en cambio, trata de los nombres de las variables. Los parámetros de una función son en el fondo variables. Esto puede traer un problema:

```

1038 public static void print() {
1039     System.out.println(f(1) + g(2));
1040 }
1041
1042 public static int f(int a) {
1043     return a + 1;
1044 }
1045
1046 public static int g(int a) {
1047     return a + 2;
1048 }

```

En este caso, tanto f como g tienen un parámetro con el mismo nombre. Java soluciona esto mediante la JVM, y su garbage collector, eliminando las variables que ya no son necesarias. Luego de ejecutar f , a ya no es necesario y luego g lo puede solicitar sin problemas. Pero veamos el siguiente código:

```

1050 public static void print() {
1051     System.out.println(f(g(2)));
1052 }
1053
1054 public static int f(int a) {
1055     return a + 1;
1056 }
1057
1058 public static int g(int a) {
1059     return a + 2;
1060 }

```

En este caso, f y g tienen que convivir al mismo tiempo y entonces, tenemos un problema de variables repetidas. Para evitar problemas, un primer paso es usar el concepto de *contextos*. Si tenemos un contexto Γ , que contiene las variables de un bloque de código, entonces está claro que el contexto de f no es el mismo que el de g y entonces se evita este problema. Sin embargo, la función `print` tiene como contexto sus propias variables (en este caso ninguna) y la unión de los contextos de las funciones que ejecuta. Como la unión de conjuntos no admite elementos repetidos, estos variables repetidas se pueden unir, funcionando como *variables globales* y esto se vuelve un problema. La reducción α permite que el lenguaje renombre las variables para evitar estos problemas, y lo hace para la compilación, por lo que para nosotros será indistinto.

3.1.2. Composición de funciones

Sean A, B, C, D conjuntos, podemos componer dos funciones $f : A \rightarrow B$, $g : C \rightarrow D$ siempre que $B \subseteq C$, y lo escribimos como $f \circ g$ (en computación solemos escribir fg). Si $f \circ g = h$, en Java, tenemos:

```

1062 public static int f(int a) {
1063     return a + 1;
1064 }
1065
1066 public static int g(int a) {
1067     return a + 2;
1068 }
1069
1070 public static int h(int a) {
1071     return f(g(a));
1072 }
```

En este ejemplo A, B, C, D son `int`, y se cumple que $B = C$, que es un caso particular de $B \subseteq C$. Esto no es posible si $C \subset B$, por ejemplo, si g retornaba un `long`.

La composición de funciones toman dos funciones y devuelven una función. Entonces, esto es una operación. ¿Tiene elemento neutro? ¿Es de utilidad? A este punto, los elementos neutros nos eran útiles para acumular variables, pero las funciones no se comportan como variables, de momento. Con la introducción de los *lambdas* en Java 8, las funciones pueden almacenarse dentro de variables y esto se vuelve de utilidad. Entonces, nuevamente, ¿cuál es el elemento neutro de la composición de funciones? Sea a izquierda o a derecha, la respuesta es la *función identidad*. Cuando pasamos por parámetro un valor a una función, se dice que estamos *aplicando* un valor a la función. Entonces, aplicar la función identidad, es lo mismo que no hacer nada. Ejemplo:

```

1074 public static int f(int a) {
1075     return a + 1;
1076 }
1077
1078 // La función identidad devuelve lo mismo que recibe y no hace nada más
1079 public static int identity(int a) {
1080     return a;
1081 }
1082
1083 public static int h(int a) {
1084     return f(identity(a));
1085 }
1086
1087 public static int h2(int a) {
1088     return identity(f(a));
1089 }
```

En este ejemplo, para un número arbitrario n , será equivalente $f(n)$ que $h(n)$ que $h2(n)$.

En el fondo, un lenguaje de programación esta formado por únicamente funciones. Lo que conocemos como instrucciones, en la teoría también lo son. Entonces, si tenemos dos instrucciones `a;` `b;`, estas se pueden visualizar como una única instrucción que realiza ambas. Esto, es una nueva operación al estilo de

concatenar instrucciones. Su elemento neutro, es la instrucción vacía. En Java, es una instrucción del tipo ; solamente. Por ejemplo:

```
1091 System.out.println("Test");
```

Esta instrucción la vimos anteriormente en las variaciones de la sentencia `for`. La *función vacía* es la función análoga a la instrucción vacía. Sin importar el nombre que le pongamos será la siguiente:

```
1092 // void es una palabra reservada, no puedo ponerla de nombre
1093 public static void fVoid() {
1094 }
```

Las funciones de tipo `void` no se pueden componer. Es intuitivo pensar que una función de tipo `void` tiene como imagen el conjunto vacío. Sin embargo, una función sin parámetros debería ser entonces una función con dominio igual al conjunto vacío. Bajo esta suposición, deberíamos poder componer, pero esto no es posible, dado que la función sin parámetros no admite nada, incluso una función que no devuelve nada.

La composición de funciones no es commutativa, es decir, no se cumple siempre que $f \circ g = g \circ f$.

3.1.3. Bitwise

Las operaciones bitwise en Java son operaciones que se realizan a nivel de bit en números enteros. Las operaciones más comunes son:

- **AND bitwise (&):** Realiza una operación AND a nivel de bit entre dos números enteros. El resultado es 1 si ambos bits son 1, y 0 en cualquier otro caso.
- **OR bitwise (|):** Realiza una operación OR a nivel de bit entre dos números enteros. El resultado es 1 si al menos uno de los bits es 1, y 0 en cualquier otro caso.
- **XOR bitwise (^):** Realiza una operación XOR a nivel de bit entre dos números enteros. El resultado es 1 si los bits son diferentes, y 0 si son iguales.
- **Desplazamiento a la izquierda («):** Desplaza los bits a la izquierda en una cantidad especificada. Los bits que se desplazan más allá del tamaño del número son descartados, y los bits vacíos a la derecha se llenan con ceros.
- **Desplazamiento a la derecha (»):** Desplaza los bits a la derecha en una cantidad especificada. Los bits que se desplazan más allá del tamaño del número son descartados, y los bits vacíos a la izquierda se llenan con el bit de signo (para números con signo) o con ceros (para números sin signo).
- **Desplazamiento a la derecha sin signo (»>):** Desplaza los bits a la derecha en una cantidad especificada. Los bits que se desplazan más allá del tamaño del número son descartados, y los bits vacíos a la izquierda se llenan con ceros.

No existe el desplazamiento a la izquierda sin signo. A continuación se muestra un ejemplo de código en Java que utiliza estas operaciones:

```
1096 int a = 5;
1097 int b = 3;
1098
1099 // AND bitwise
1100 int andResult = a & b;
1101 System.out.println("AND: " + andResult);
1102
1103 // OR bitwise
1104 int orResult = a | b;
1105 System.out.println("OR: " + orResult);
1106
```

```

1107 // XOR bitwise
1108 int xorResult = a ^ b;
1109 System.out.println("XOR: " + xorResult);
1110
1111 // Desplazamiento a la izquierda
1112 int leftShiftResult = a << 2;
1113 System.out.println("Desplazamiento a la izquierda: " + leftShiftResult);
1114
1115 // Desplazamiento a la derecha
1116 int rightShiftResult = a >> 1;
1117 System.out.println("Desplazamiento a la derecha: " + rightShiftResult);
1118
1119 // Desplazamiento a la derecha sin signo
1120 int unsignedRightShiftResult = a >>> 1
1121 System.out.println("Desplazamiento sin signo a la derecha: " + rightShiftResult);

```

Por ejemplo, en binario, si desplazamos a la derecha sin signo 100101101 obtenemos 010010110. Esto es, en base 2, obtener el cociente de la división entera por dos. De forma similar, desplazar a la izquierda 100101101 obtenemos 1001011010. Esto es, en base 2, equivalente a multiplicar por 2 (de la misma forma que agregar un 0 en base 10 es equivalente a multiplicar por 10).

El comportamiento con o sin signo, varía. El desplazamiento a la derecha con signo ($>>$): 11111111 \rightarrow 11111111, mientras que, el desplazamiento a la derecha sin signo ($>>>$): 11111111 \rightarrow 01111111

3.1.4. Funciones inversas

Una función inversa es aquella que anula la el comportamiento de una función dada. Bajo este concepto, la función $f(x) = x + 1$ se anula con $g(x) = x - 1$, ya que $f(g(x)) = x$. Esto indica que g es *función inversa* de f , y viceversa. En clases de análisis matemático se hace un análisis más profundo que este capítulo no abordará, pero es importante saber que sucede con funciones isomorfas.

En lo que nos corresponde en computación, si puedo revertir una función, es porque existe su inversa. Por ejemplo, si tengo una función que escribe en pantalla la palabra Hola, y luego tengo una función que la borra, entonces la función que borra es función inversa pero no al revés, dado que borrar y luego escribir Hola lo mantuvo en pantalla. En cambio, una función que suma 10 a una variable entera, y luego otra función que reste 10, ejecutar una consecutivamente a la siguiente la anula. Entonces ambas serían funciones inversas entre sí.

Puede que estas funciones inversas no existan, y suele suceder sobre todo en *funciones de ida* o *funciones de pérdida de información*. Una función de ida, es la función módulo. Por ejemplo, $f(n) = n \% 2$ me lleva fácilmente a 1 o a 0 como resultado ($f(7) == 1$ por ejemplo). Sin embargo, la *vuelta* de esta función no me permite encontrar quien fue el parámetro de entrada (Dado el 1, el n podría haber sido 1, 3, 5, 7, etc.). Una función de pérdida de información, puede ser la del MP3 para sonido, o JPG para imágenes. Cuando estos archivo se comprimen, no es posible volver al formato original, aunque su calidad sea muy alta. Estas técnicas son diferentes a la del RAR o el ZIP que permiten comprimir sin pérdida de información. Otro ejemplo es tomar una imagen, y achatarla. Acharar una imagen será eliminar filas de píxeles, y es imposible deducir cuales son los píxeles borrados a partir de los que queden (se puede tratar de deducir, pero imaginar un caso de píxeles aleatorios).

También existen funciones que tienen de función inversa a sí mismas. Por ejemplo, invertir los colores de una imagen dos veces, recupera la original, entonces es inversa de sí misma. O bien, la compuerta NOT aplicada dos veces anula el comportamiento de esta compuerta, entonces también es su propia inversa. La compuerta XOR también es inversa a sí misma (incluso en bitwise).

En el caso de que exista una función inversa, entonces, se tiene que $f \circ f^{-1} = i$, siendo f^{-1} la función inversa de f e i la función identidad.

3.1.5. Función tirar

La función tirar es una función que recibe n parámetros, y retorna 1 solo de estos. Ejemplo: $f(n_1, n_2, \dots, n_i, \dots, n_k) = n_i$.



(a) imagen



(b) invertir(imagen)



(c) invertir(invertir(imagen))



(a) Acharar no tiene inversa

```

1122 // throw es una palabra reservada, no puedo ponerla de nombre
1123 public static void fThrow(int a, int b) {
1124     return a;
1125 }
```

Esta función, demuestra lo importante de que un lenguaje sea *lazy*, es decir, que sus parámetros se lean parcialmente hasta ser necesarios. Si en la función tirar primero calculamos los parámetros, y justo uno que no es necesario porque luego se tira, tiene un tiempo de ejecución alto, entonces sería problemático.

La función tirar también se conoce como *proyección* y está presente con diversos nombres (más adelante se verá de tomar el tope de una pila o el primero de una cola), por ejemplo, tomer el *head* o el *tail* de una lista (el *head* toma el primer elemneto y el *tail* la sublista con los elementos restantes). También es importante en computación cuántica, pero con valor de probabilidad asignado.

3.1.6. Función constante

Una función constante es la que tiene como imagen un conjunto con un solo elemento. Por ejemplo:

```

1126 public static double pi() {
1127     final int n = 1000;
1128     double total = 0.0;
1129     for (int i = 0; i < n; i++) {
1130         total += Math.pow(-1, i) / (2 * i + 1);
1131     }
1132     double pi = 4 * total; // Aproximacion de pi usando la funcion seno
1133     return pi;
1134 }
1135
1136 public static double pi() {
1137     return 3.141592;
1138 }
```

Ambas funciones son funciones constantes. Si les agregamos parámetros y los ignoran, seguirán siendo funciones constantes. Dado que no necesitan sus parámetros, no necesitan de inversa. La composición de funciones, que involucran funciones constantes, usualmente terminan siendo constantes también.

3.1.7. Consumidores y productores

El concepto es sencillo. Una función consumidora (o *consumer*) es aquella que recibe un parámetro y no retorna nada, y una función productora (o *supplier*) es una función que no recibe parámetros y retorna algo.

Por ejemplo:

```
1139 public static void consumer(int n) {
1140     System.out.println(n);
1141 }
1142
1143 public static double supplier() {
1144     return 3.141592;
1145 }
```

3.1.8. Funciones puras e impuras

Una función es *pura* cuando siempre devuelve lo mismo para una cierta combinación de parámetros (o es constante sin parámetros), o *impura* de caso contrario. Ejemplos:

```
1146 public static int plus(int a, int b) { // Pura
1147     return a + b;
1148 }
1149
1150 public static int now() { // Impura
1151     return java.time.LocalTime.now().getHour();
1152 }
```

3.1.9. Funciones partidas

Una función partida es aquella que toma el aspecto de diferentes funciones según el parámetro de entrada. Nos valemos de los condicionales para representarlo. El siguiente ejemplo representa la función de Collatz (una función partida muy famosa, y actualmente su conjectura no está resuelta):

```
1153 public static void collatz(int n) {
1154     System.out.println("Número inicial: " + n);
1155     System.out.print("Secuencia de Collatz: " + n + " ");
1156
1157     while (n != 1) {
1158         if (n % 2 == 0) {
1159             n = n / 2;
1160         } else {
1161             n = 3 * n + 1;
1162         }
1163         System.out.print(n + " ");
1164     }
1165     System.out.println();
1166 }
```

3.2. Recursividad

Una función recursiva es aquella que dentro de su bloque de código se invoca a sí misma. Por ejemplo, la siguiente función representa un ciclo infinito:

```
1167 public static void inf(int n) {
1168     return inf(n);
1169 }
```

Java tiene la capacidad de cortar estos ciclos infinitos y tirar un error gracias al *stack overflow*, una técnica que implica que las funciones *viven o mueren* y las funciones vivas se colocan dentro de una pila. Haciendo que, si la pila se llena entonces el programa se detiene con el error de desbordamiento de pila.

Veamos cuatro ejemplos sencillos de recursividad para la suma, multiplicación, potencia y factorial de un número:

```

1170 public static int plus(int a, int b) {
1171     if (b == 0) {
1172         return a;
1173     }
1174     return plus(a, b - 1) + 1;
1175 }
1176
1177 public static int times(int a, int b) {
1178     if (b == 0) {
1179         return 0;
1180     }
1181     return a + times(a, b - 1);
1182 }
1183
1184 public static int pow(int a, int b) {
1185     if (b == 0) {
1186         return 1;
1187     }
1188     return a * pow(a, b - 1);
1189 }
1190
1191 public static int factorial(int n) {
1192     if (n == 0) {
1193         return 1;
1194     }
1195     return n * factorial(n - 1);
1196 }
```

En general, las funciones recursivas tienen un *caso base* o *condición de corte*, seguido de un *caso general*. Si la recursión nunca llega al caso base, entonces se queda en un ciclo infinito. Por ejemplo, en los ejemplos anteriores, es necesario definir como precondición que calculamos el factorial para cualquier $n \geq 0$, de lo contrario, decrementar n no es suficiente para alcanzar el caso base y se quedará en ciclo infinito.

El *teorema de la recursión* establece que cualquier función puede ser definida en forma recursiva. Entonces, si podemos resolver un problema con iteraciones, también podemos resolverlo con *recursión*.

La serie de Fibonacci es una serie famosa, donde $F_0 = 0$, $F_1 = 1$ y en general $F_n = F_{n-1} + F_{n-2}$. Entonces, una de las formas iterativas de resolver este problema es:

```

1197 public static int fibonacci(int n) {
1198     if (n <= 1) {
1199         return n; // Casos base: Fibonacci(0) = 0, Fibonacci(1) = 1
1200     }
1201
1202     int prev1 = 0;
1203     int prev2 = 1;
1204     int current = 0;
1205
1206     for (int i = 2; i <= n; i++) {
1207         current = prev1 + prev2;
1208         prev1 = prev2;
1209         prev2 = current;
1210     }
1211
1212     return current;
```

1213 }

En cambio, la forma recursiva es:

```
1214 public static int fibonacci(int n) {
1215     if (n <= 1) {
1216         return n;
1217     }
1218     return fibonacci(n - 1) + fibonacci(n - 2);
1219 }
```

La versión recursiva de Fibonacci nos muestra un caso de recursión doble. Por otro lado, el algoritmo del MCD (Mayor común divisor, GCD en inglés) es un ejemplo de una función recursiva con más de un parámetro:

```
1220 public static int findGCD(int a, int b) {
1221     if (b == 0) {
1222         return a;
1223     }
1224     return findGCD(b, a % b);
1225 }
```

3.3. Ejercicios

Ejercicio 3.3.1: Factorial

ea $a \in \mathbb{N}$, variable, calcular su factorial. En el caso de no poder calcularlo debido al tamaño del a , entonces indicarlo con un mensaje en pantalla.

† Sea $a \in \mathbb{N}$, variable, calcular su doble factorial. En el caso de no poder calcularlo debido al tamaño del a , entonces indicarlo con un mensaje en pantalla.

Ejercicio 3.3.2: Sucesión de Lucas

ea $i \in \mathbb{N}$, variable que representa un índice, calcular el i -ésimo número de la serie de Fibonacci modificada de tal manera que $F_1 = 15$ y $F_2 = -1$. El caso general de este tipo de sucesiones se conocen como sucesiones de Lucas.

† Cree una función que reciba un $n \in \mathbb{N}$, que mientras más grande sea, mejor aproxime el valor de $\sqrt{5}$.

Ejercicio 3.3.3: Palíndromo

ea s una cadena de texto, desarrolle una función que devuelva `true` si es palíndromo.

† Modifique la función anterior para que frases como `Madam`, `in Eden`, `I'm Adam`. sean válidas independientemente de sus minúsculas, mayúsculas, espacios y caracteres especiales.

Ejercicio 3.3.4: Múltiplo de 3

ea $n \in \mathbb{N}$ con $n > 15000000$, variable, calcular si es múltiplo de 3 sin usar el operador `%`.

† Modifique la función anterior para verifique la multiplicidad usando la suma de los dígitos de n . Entonces, n es múltiplo de 3 si la suma de sus dígitos es 3, 6 o 9.

Ejercicio 3.3.5: Sistema binario vs sistema decimal

ea $a \in \mathbb{N}$, variable que representa un número en base 10, imprimir en pantalla su forma en base 2.

† Sea $a \in \mathbb{N}$, variable que representa un número en base 10, imprimir en pantalla su forma en base 16.

Ejercicio 3.3.6: Único número par

ea A un array de $a \in \mathbb{N}$, compuesto por números repetidos y un único número sin repetir, crear una función que encuentre el número que no se repite. Puede utilizar una función auxiliar que tome un número e indique si pertenece o no al array.

† Desarrolle la función anterior sin usar una función auxiliar y usando el operador bitwise XOR.

Ejercicio 3.3.7: Ordenamiento

ea A un array de $a \in \mathbb{Z}$, realizar la búsqueda del elemento más grande con un algoritmo de maximización.

† Sea A un array de $a \in \mathbb{Z}$, ordenar el array con Merge Sort y luego tomar el elemento más grande. Sin implementar, describa como podría mejorar este ordenamiento utilizando Merge Sort híbrido.

Ejercicio 3.3.8: Collatz

implementar la conjetura de Collatz y demostrar que es válida para $n < 10^3$. Se define de la siguiente forma:

$$f(n) = \begin{cases} \frac{n}{2}, & \text{si } n \equiv 0 \pmod{2}, \\ 3n + 1, & \text{si } n \equiv 1 \pmod{2}. \end{cases}$$

En la conjetura de Collatz, se toma un número natural y se lo aplica en la función anterior. Si el resultado es distinto de 1, entonces el resultado se vuelve a aplicar en la función. La conjetura establece que evaluando el resultado en la función, en algún paso se llegará a 1. Luego, se queda en el ciclo $1 \rightarrow 4 \rightarrow 2 \rightarrow 1$.

† De los valores anteriores, encontrar el número que llevó más pasos en llegar al 1.

Ejercicio 3.3.9: Función de Ackermann [Wilhelm Ackermann, 1926]

implemente la función

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

La función de Ackermann es una función recursiva pura (no depende de variables externas), de interés para la computación porque tiene un crecimiento extremadamente rápido. Será analizado en la sección de complejidad computacional.

† Encuentre los valores para los cuales el tipo `long` funciona. Luego, calcule los tiempos de ejecución y comparelos con la función factorial.

Ejercicio 3.3.10: Identidad de Vandermonde

esarrolle una función que calcule $\binom{n}{k}$ usando la identidad de Vandermonde:

$$\binom{m+n}{r} = \sum_{k=0}^r \binom{m}{k} \binom{n}{r-k}$$

† Verifique para $1 \leq n \leq 15$, que la sumatoria de las combinatorias de la forma $\binom{n}{k}$ con $0 \leq k \leq n$, da como resultado una potencia de 2.

Capítulo 4

Fundamentos de POO

4.1. Introducción a la POO

4.1.1. Clases y objetos

La *Programación Orientada a Objetos* (POO), permite definir *clases* que funcionan como esquema de un nuevo tipo de dato. Los valores que este nuevo tipo de dato puede tomar se llaman *objetos* o *instancias* de la clase.

Por ejemplo, puedo crear una clase *Animal* que me permita crear diferentes instancias que representen animales. Estas instancias funcionan como valores, por lo que es posible guardarlos en variables.

El siguiente diagrama es una representación UML de una clase:

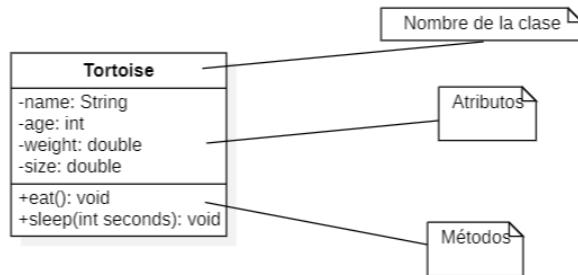


Figura 4.1: Clase

Una clase puede representar conceptos de nuestro cotidiano sean abstractos o no. La base es que una clase posee *atributos* y *comportamiento* (dado por *métodos*, que son funciones dentro del concepto de una clase). Si una clase tiene únicamente atributos, entonces probablemente sea un DTO (Data Transfer Object). Ejemplos de DTO son clases que representan entidades de una base datos, request en una conexión HTTP, responses, o simplemente clases para transportar datos de una capa a otra en distintos frameworks. Por otro lado, si la clase tiene solo comportamiento, probablemente funcione como un DAO (Data Access Object).

Es común usar el concepto de clases para agrupar métodos útiles para todo el proyecto mediante *clases utilitarias*. Sin embargo, esto es considerado una mala práctica, dado que no representa ningún concepto específico y solo se aprovecha la sintaxis para que el código funcione.

En lo que va de esta materia, analizaremos sobre todo, el caso donde las clases tienen atributos y métodos. Los atributos funcionan como *variables globales* dentro de una instancia, donde todos sus métodos pueden acceder a estos valores en cualquier momento.

Además, existe un método especial llamado *constructor*, que indica como crear una nueva instancia en memoria (y en muchos otros lenguajes tenemos su análogo conocido como *destructor*).

La *abstracción* es un concepto clave en la POO que se refiere a la capacidad de representar conceptos o entidades del mundo real de manera simplificada y conceptual en el código. En otras palabras, la abstracción permite identificar las características esenciales de un objeto o entidad y representarlas en un modelo de programación.

4.1.2. Interfaces y clases abstractas

Una *interfaz* es un conjunto de métodos (funciones) *abstractos* (luego veremos más a detalle esto) que define un *contrato* o una *especificación* para el comportamiento que debe tener una clase en particular. Una interfaz define qué métodos debe implementar una clase que la utilice, pero no proporciona una implementación concreta de esos métodos. Es decir, una interfaz solo establece la *firma* de los métodos (nombre, parámetros y tipo de retorno), pero no contiene la lógica o el código real de cómo se lleva a cabo la implementación de esos métodos.

Las interfaces se utilizan para establecer un contrato común o una especificación que varias clases pueden cumplir, lo que permite la interoperabilidad y la reutilización del código. Una clase que implementa una interfaz debe proporcionar una implementación concreta de todos los métodos definidos en la interfaz. Esto asegura que las clases que implementan la misma interfaz tengan un conjunto consistente de métodos que puedan ser invocados de manera uniforme, independientemente de la clase concreta que se esté utilizando.

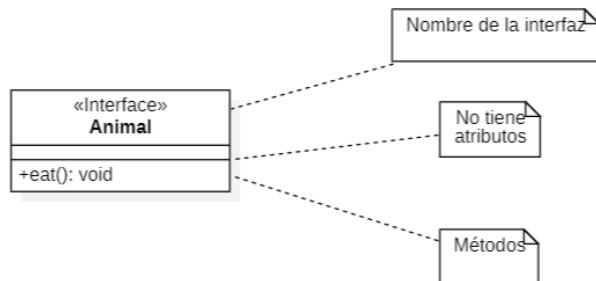


Figura 4.2: Interfaz

Las *clases abstractas* son similares a las interfaces. La diferencia principal es que sus métodos pueden o no tener una implementación ya definida. De esta forma, la clase que use una clase abstracta como contrato, tendrá algunas implementaciones ya creadas. En el caso de querer reemplazar este comportamiento por defecto, la clase debe hacer una *sobreescritura* de método.

Tanto las clases abstractas como las interfaces no pueden tener instancias. Su propósito es obligar a otras clases a tener un comportamiento que se indique, y a la reutilización de código.

Cuando una clase usa una interfaz, se dice que la clase *implementa* la interfaz. Cuando una clase usa una clase abstracta, se dice que *hereda* de la clase abstracta (en UML implementar y heredar son representadas con flechas distintas). Sumado a esto, una clase puede heredar de otra clase aunque no sea abstracta.

En UML se utiliza el nombre de la clase en cursiva para indicar que nos referimos a una clase abstracta.

4.1.3. Records y annotations

Cuando los atributos de una clase no se modifican en el tiempo, entonces sus posiciones en memoria pueden ser constantes una vez se crea la instancia. En este caso particular, nuestra clase puede ser convertida en un *record*, el cuál se considera mucho más prolífico en cuanto a buenas prácticas y es más eficiente. Usualmente, los DTOs se convierten a records a largo plazo. Dado que los DTOs no son de mucho interés a lo largo de la materia, es probable que los records se requieran poco. En UML:

Por otro lado, las *anotaciones* funcionan como marcas de agua para nuestros métodos, clases, variables, etc. Es decir, podemos poner marcas a nuestro código, que luego pueden ser usadas por técnicas de *reflexión* (no entra en esta materia) para que el código modifique su ejecución o compilación. Algunas anotaciones

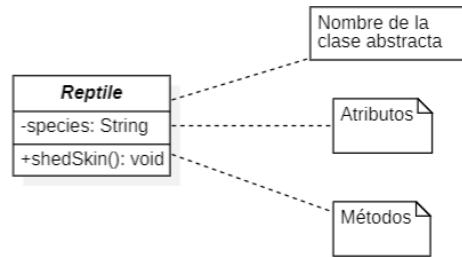


Figura 4.3: Clase abstracta

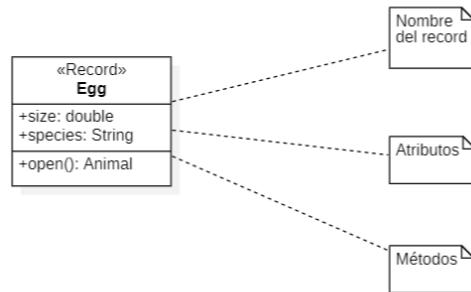


Figura 4.4: Record

como `@Override`, son solo a modo de documentación y suelen ser totalmente optativos (sin embargo no podemos poner esta anotación sobre métodos que no sobrescriben).

Por último, un *enumerado* es un tipo especial de clase donde conocemos todas las posibles instancias. Estas instancias deberán ser indicadas, no se podrán alterar, agregar o eliminar.



Figura 4.5: Enum y Annotation

4.2. Sintaxis en Java

La sintaxis de una clase en java es la siguiente:

```
1226 public class Example {  
1227  
1228 }
```

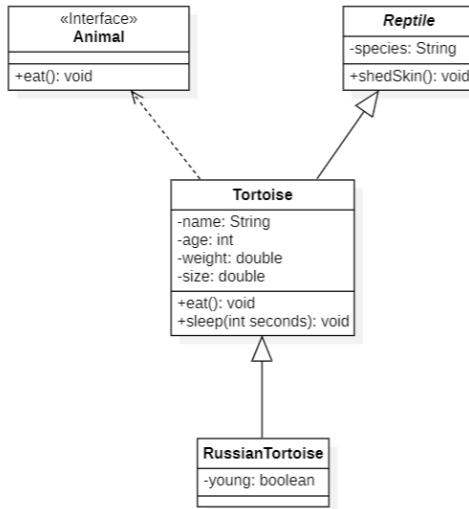


Figura 4.6: Herencia e implementación

Aquí, `public` representa el alcance la clase (se verá luego), y se indica que creamos una clase mediante `class`. Las clases deben tener atributos (lo veremos como variables) y métodos (lo veremos como funciones). La sintaxis es la siguiente:

```

1229 public class Tortoise {
1230
1231     public String name;
1232     public int age;
1233     public double weight;
1234     public double size;
1235
1236     public void eat() {
1237         System.out.println("Looking for lettuce");
1238     }
1239
1240     public void sleep(int seconds) {
1241         System.out.println("Waking up after " + seconds + " seconds");
1242     }
1243
1244 }
```

El orden no es importante, pero por prolíjidad seguiremos la regla de colocar los atributos antes que los métodos. Dado que los atributos se comportan como variables podemos asignar o leer sus valores. Para crear una nueva instancia de nuestra clase usaremos la siguiente sintaxis:

```

1245 public static void main(String[] args) {
1246     Tortoise tortoise = new Tortoise();
1247     tortoise.age = 10;
1248     System.out.println(tortoise.age);
1249
1250     tortoise.eat();
1251     tortoise.sleep(100);
1252 }
```

La palabra reservada `new` reserva un nuevo espacio en memoria para almacenar la referencia al objeto que se creará usando el formato de la clase `Tortoise`. Luego mediante el *operador punto* podemos acceder a sus atributos y métodos. Usamos `new` para llamar a un método especial llamado *constructor*, el cual en

principio no está escrito porque en su forma base es optativo. La clase anterior con su constructor vacío expuesto queda:

```

1253 public class Tortoise {
1254
1255     public String name;
1256     public int age;
1257     public double weight;
1258     public double size;
1259
1260     public Tortoise() {
1261
1262     }
1263
1264     public void eat() {
1265         System.out.println("Looking for lettuce");
1266     }
1267
1268     public void sleep(int seconds) {
1269         System.out.println("Waking up after " + seconds + " seconds");
1270     }
1271
1272 }
```

La utilidad del constructor es que nos permite fijar valores por defecto al crear una nueva instancia solicitandolos como parámetros. La sintaxis de este constructor, es que se comporta como método pero sin indicar un tipo de dato de retorno y teniendo el mismo nombre que la clase. Lo podemos modificar para que fije por defecto un nombre:

```

1273 public class Tortoise {
1274
1275     public String name;
1276     public int age;
1277     public double weight;
1278     public double size;
1279
1280     public Tortoise() {
1281         name = "Flash";
1282     }
1283
1284     public void eat() {
1285         System.out.println("Looking for lettuce");
1286     }
1287
1288     public void sleep(int seconds) {
1289         System.out.println("Waking up after " + seconds + " seconds");
1290     }
1291
1292 }
```

Sin embargo, sería de interés solicitar el nombre de que se fijará mediante un parámetro:

```

1293 public class Tortoise {
1294
1295     public String name;
1296     public int age;
1297     public double weight;
1298     public double size;
1299
1300     public Tortoise(String name) {
```

```

1301     this.name = name;
1302 }
1303
1304 public void eat() {
1305     System.out.println("Looking for lettuce");
1306 }
1307
1308 public void sleep(int seconds) {
1309     System.out.println("Waking up after " + seconds + " seconds");
1310 }
1311
1312 }
```

La razón por la que escribimos `this.name = name;` en lugar de `name = name;` es que tanto el atributo como el parámetro que representa el nombre provocan una ambigüedad. Para solucionarlo, siempre que queremos hacer referencia a los atributos o métodos de la clase en cuestión, usamos la palabra reservada `this`. Ahora, con el constructor modificado, es obligatorio que desde el `main` se cree una instancia nueva pasando como parámetro el nombre.

```

1313 public static void main(String[] args) {
1314     Tortoise tortoise = new Tortoise("Speed");
1315     tortoise.age = 10;
1316     System.out.println(tortoise.age);
1317
1318     tortoise.eat();
1319     tortoise.sleep(100);
1320 }
```

Es importante notar que el constructor por defecto no fue necesario escribirlo en primera instancia para poder tenerlo, pero una vez que agregamos un constructor personalizado, este ya será generado. Podemos, de todas formas, crear múltiples constructores para nuestra clase:

```

1321 public class Tortoise {
1322
1323     public String name;
1324     public int age;
1325     public double weight;
1326     public double size;
1327
1328     public Tortoise() {
1329
1330     }
1331
1332     public Tortoise(String name) {
1333         this.name = name;
1334     }
1335
1336     public Tortoise(double weight) {
1337         this.weight = weight;
1338     }
1339
1340     public void eat() {
1341         System.out.println("Looking for lettuce");
1342     }
1343
1344     public void sleep(int seconds) {
1345         System.out.println("Waking up after " + seconds + " seconds");
1346     }
1347 }
```

1348 }

De esta forma, podemos hacer explícito que queremos el constructor por defecto a pesar de tener otros constructores. No podemos crear dos constructores que reciban argumentos con la misma secuencia de tipos de datos. Por ejemplo:

```
1349 public Tortoise(double weight) {
1350     this.weight = weight;
1351 }
1352
1353 public Tortoise(double size) {
1354     this.size = size;
1355 }
```

Esto no está permitido dado que desde el main, ejecutar new Tortoise(2) sería ambiguo, dado que no se podría deducir a cual se hace referencia.

Cuando usamos un constructor vacío, ninguno de los atributos tendrá fijado un valor. ¿Qué sucede entonces si leemos un atributo no inicializado? Tendrá un valor por defecto. Al principio de este apunte contamos con una lista de valores por defecto para tipos de datos primitivos. Mientras que los tipos de datos no primitivos (como **String**, **Scanner** o **Tortoise**) será **null**.

El valor **null** representa la nada. Es el valor que encontramos en una variable de tipo de dato no nativo cuando no está inicializada. Sin embargo, podemos forzar esto fijando el valor **null** por defecto:

```
1356 public static void main(String[] args) {
1357     Tortoise tortoise = null;
1358     tortoise.age = 10;
1359     System.out.println(tortoise.age);
1360
1361     tortoise.eat();
1362     tortoise.sleep(100);
1363 }
```

El valor **null** no es una instancia, simplemente es una referencia a la nada. Entonces, no es posible cambiar sus atributos o llamar a uno de sus métodos. Es decir, no podemos usar el operador punto. Si ejecutamos el código anterior nos dará un NPE (**NullPointerException**), un error para indicar que no podemos usar el operador punto sobre un valor nulo.

4.3. Encapsulamiento

La palabra **public** en los atributos de nuestra clase nos indica el nivel de acceso a estos desde el exterior de la clase. En particular **public** permite usar el operador punto para modificar estos atributos desde cualquier lado. Contamos con cuatro niveles de acceso distintos:

- **public**: Accesible desde cualquier clase en cualquier paquete
- **default**: Dentro de la misma clase y del mismo paquete
- **protected**: Dentro de la misma clase, del mismo paquete y desde las clases hijas (subclases) en paquetes diferentes
- **private**: Solo dentro de la misma clase

El caso de **default** es especial dado que no es una palabra reservada, sino que este alcance se obtiene al no aclarar el alcance del atributo.

El concepto de sub-clases no será visto en este apunte, ni de como funcionan los paquetes.

El *encapsulamiento* de nuestro código lo obtendremos al colocar nuestros atributos como privados, y nos dará algunos beneficios sobre nuestro código:

- Seguridad y control: Al declarar los atributos como `private`, se evita que sean modificados o accedidos directamente desde fuera de la clase. Esto permite un mayor control sobre cómo se accede y modifica el estado de los objetos de la clase, lo que puede ayudar a prevenir errores y comportamientos inesperados.
- Ocultamiento de la implementación: El encapsulamiento permite ocultar los detalles de implementación de una clase, lo que brinda flexibilidad para cambiar la implementación interna de la clase sin afectar a los usuarios externos de la misma. Si se cambia la implementación interna de una clase, como el tipo de dato de un atributo, los usuarios externos no se ven afectados si los atributos son privados.
- Mantenimiento y refactorización: Al utilizar atributos privados, se facilita el mantenimiento y refactorización del código. Si los atributos fueran públicos, cualquier cambio en la forma en que se acceden o modifican podría tener un impacto en todas las partes del código que utilizan esa clase. Con atributos privados, los cambios internos a la clase se pueden realizar sin afectar a otras partes del código.
- Encapsulamiento de la lógica de negocio: Los atributos privados permiten encapsular la lógica de negocio de una clase, lo que significa que la manipulación de los datos y su comportamiento asociado se mantiene dentro de la clase. Esto ayuda a garantizar la integridad de los datos y asegura que la lógica de negocio se aplique de manera coherente en toda la clase.

Una clase sin acceso externo a sus atributos necesariamente necesita usarlos en la lógica de sus métodos. De lo contrario ¿para qué tendríamos datos almacenados en los atributos? Esto nos permite darnos una idea de que tan bien está implementada una clase, deberíamos tener un fuerte uso de nuestros atributos. Pero, de todas formas, hay ocasiones en las que necesitamos leer estos atributos. Para mantener el modo privado de los atributos, pero poder accederlos o modificarlos, usaremos *getters* y *setters*:

```
1364 public class Tortoise {  
1365  
1366     private String name;  
1367     private int age;  
1368     private double weight;  
1369     private double size;  
1370  
1371     public void eat() {  
1372         System.out.println("Looking for lettuce");  
1373     }  
1374  
1375     public void sleep(int seconds) {  
1376         System.out.println("Waking up after " + seconds + " seconds");  
1377     }  
1378  
1379     public String getName() {  
1380         return name;  
1381     }  
1382  
1383     public void setName(String name) {  
1384         this.name = name;  
1385     }  
1386  
1387     public int getAge() {  
1388         return age;  
1389     }  
1390  
1391     public void setAge(int age) {  
1392         this.age = age;  
1393     }  
1394  
1395     public double getWeight() {  
1396         return weight;  
1397     }
```

```

1398
1399     public void setWeight(double weight) {
1400         this.weight = weight;
1401     }
1402
1403     public double getSize() {
1404         return size;
1405     }
1406
1407     public void setSize(double size) {
1408         this.size = size;
1409     }
1410 }
```

El formato de un *getter* siempre es el mismo: acceder a un dato. Vemos que no necesita ningún parámetro y siempre retorna el dato que quiere leer. De la misma forma, un *setter* modifica un atributo, y para modificarlo necesita un valor a fijar y no necesita devolver nada. La sintaxis para los getters y setters es usar el prefijo *get* o el prefijo *set* delante del nombre del atributo en cuestión. La única ocasión donde esta regla cambia, es con los booleanos nativos (*boolean*), donde el getter tendrá como prefijo *is* en lugar de *get*. Por prolividad, los getters y setters se escriben luego de los métodos, y no se consideran comportamiento de nuestra clase (aunque un DTO tenga getters y setters seguirá siendo un DTO).

Es importante tener los getters y setters bien definidos para que plugins, operaciones de reflection, y algunas librerías (o incluso el IDE) se comporten correctamente respecto a nuestra clase.

Existe un debate actualmente si los getters y setters rompen o no el encapsulamiento. De todas formas, la utilidad principal, es que la forma de acceder a los atributos de forma externa es distinta a la interna (desde afuera uso métodos y desde adentro atributos), dando escalabilidad al código.

Por último, si una vez fijado por los constructores, nuestros atributos no cambian a lo largo del código (no tienen setters y ningún método los modifica) entonces este atributo puede ser constante. En el caso de ser de tipo no primitivo, será constante su referencia a memoria. Para indicar esto podemos usar la palabra *final*, como por ejemplo, `private final String name = "Flash";`. Sin embargo, fijar un valor directamente en el atributo y no en el constructor es considerado una mala práctica porque puede ocasionar una mala flexibilidad o un mal *acoplamiento* de los datos. Al poner un atributo como *final*, es necesario fijarle de una vez el valor que tendrá de forma definitiva, por lo que, para ser prolijos, lo deberá hacer un constructor.

4.3.1. GRASP

Los patrones de diseño (concepto de origen en la arquitectura) nos permiten diseñar nuestro código mediante reglas que le dan mayor calidad. No se trata de algoritmos, sino de diseño de la aplicación. GRASP (General Responsibility Assignment Software Patterns, Patrones Generales de Asignación de Responsabilidades en Software) es un conjunto de patrones de diseño de software que se utilizan para asignar responsabilidades claras y cohesivas a las clases y objetos en un *diseño orientado a objetos*.

La siguiente es la lista de patrones de diseño envueltos bajo el concepto de GRASP:

- **Information Expert (Experto en Información):** Según el patrón Information Expert, una clase debe ser responsable de realizar una tarea si posee la información necesaria para hacerlo. Por ejemplo, si tenemos un función que permite darle de comer a una tortuga y una clase *Tortoise* tiene un atributo que indica su alimentación, entonces *Tortoise* es quien es el experto en como se debe comportar este método y por ende es quien debe tener este atributo.
- **Creator (Creador):** Según el patrón Creator, una clase debe ser responsable de crear instancias de otras clases si tiene la información necesaria para hacerlo. Por ejemplo, podemos crear una clase *Shell* que tenga información del caparazón de una tortuga y tener un atributo en *Tortoise* que represente el caparazón y tenga este tipo de dato. Quien debe inicializar este atributo es *Tortoise* y no por ejemplo el *main* de nuestra aplicación. Remarco que esto es referente a la inicialización del atributo, luego se puede modificar donde sea necesario.

- Controller (Controlador): Según el patrón Controller, una clase debe ser responsable de coordinar y controlar las interacciones entre objetos si tiene la información necesaria para hacerlo.
- High Cohesion (Alta Cohesión): La cohesión es un principio de diseño que indica que una clase debe tener una sola responsabilidad clara y bien definida. En Java, esto implica que una clase debe tener métodos y atributos relacionados entre sí y que estén cohesionados en términos de su funcionalidad y propósito. La clase Tortoise no podría tener un método fly porque no tendría sentido para una tortuga, aunque el código desde allí funcione perfectamente para representar el comportamiento de otra clase que si tiene la información correspondiente.
- Low Coupling (Bajo Acoplamiento): El acoplamiento es un principio de diseño que indica que las clases y objetos deben tener una baja dependencia entre sí. El uso de clases abstractas o interfaces dan más abstracción (lo veremos en la siguiente sección).

4.4. Herencia y polimorfismo

Empezamos con una clase Tortoise sin constructor:

```
1411 public class Tortoise {  
1412  
1413     private String name;  
1414     private int age;  
1415     private double weight;  
1416     private double size;  
1417  
1418     public void eat() {  
1419         System.out.println("Looking for lettuce");  
1420     }  
1421  
1422     public void sleep(int seconds) {  
1423         System.out.println("Waking up after " + seconds + " seconds");  
1424     }  
1425  
1426     public String getName() {  
1427         return name;  
1428     }  
1429  
1430     public void setName(String name) {  
1431         this.name = name;  
1432     }  
1433  
1434     public int getAge() {  
1435         return age;  
1436     }  
1437  
1438     public void setAge(int age) {  
1439         this.age = age;  
1440     }  
1441  
1442     public double getWeight() {  
1443         return weight;  
1444     }  
1445  
1446     public void setWeight(double weight) {  
1447         this.weight = weight;  
1448     }  
1449  
1450     public double getSize() {
```

```
1451     return size;
1452 }
1453
1454     public void setSize(double size) {
1455         this.size = size;
1456     }
1457 }
```

Podemos crear una segunda clase RussianTortoise que claramente es una tortuga, a la cual consideraremos como una variación de la anterior. Más allá de su que Tortoise es más general, esta generalidad marca una diferencia y se consideran abstracciones distintas, por lo que crearemos una clase nueva para representar este concepto:

```
1458 public class RussianTortoise {
1459
1460     private String name;
1461     private int age;
1462     private double weight;
1463     private double size;
1464     private boolean young;
1465
1466     public void eat() {
1467         System.out.println("Looking for lettuce");
1468     }
1469
1470     public void sleep(int seconds) {
1471         System.out.println("Waking up after " + seconds + " seconds");
1472     }
1473
1474     public String getName() {
1475         return name;
1476     }
1477
1478     public void setName(String name) {
1479         this.name = name;
1480     }
1481
1482     public int getAge() {
1483         return age;
1484     }
1485
1486     public void setAge(int age) {
1487         this.age = age;
1488     }
1489
1490     public double getWeight() {
1491         return weight;
1492     }
1493
1494     public void setWeight(double weight) {
1495         this.weight = weight;
1496     }
1497
1498     public double getSize() {
1499         return size;
1500     }
1501
1502     public void setSize(double size) {
1503         this.size = size;
```

```

1504     }
1505
1506     public boolean isYoung() {
1507         return young;
1508     }
1509
1510     public void setYoung(boolean young) {
1511         this.young = young;
1512     }
1513 }
```

Claramente, tenemos código repetido. La *herencia* entre clases nos permite aclarar una *jerarquía* entre las clases, algo así como *subtipos*. Lo podemos indicar de la forma `public class RussianTortoise extends Tortoise`, lo cual se lee como "RussianTortoise es un subtipo de Tortoise". Esto nos permitirá que RussianTortoise *herede* todo lo que tiene Tortoise sin necesidad de escribirlo repetido:

```

1514 public class RussianTortoise extends Tortoise {
1515
1516     private boolean young;
1517
1518     public boolean isYoung() {
1519         return young;
1520     }
1521
1522     public void setYoung(boolean young) {
1523         this.young = young;
1524     }
1525 }
```

Y luego desde el main tenemos:

```

1526 public static void main(String[] args) {
1527     RussianTortoise russianTortoise = new RussianTortoise();
1528     russianTortoise.setName("Manuelita");
1529     russianTortoise.setYoung(true);
1530 }
```

Supongamos que ahora agregamos el siguiente constructor a Tortoise:

```

1531 public Tortoise(String name, int age, double weight, double size) {
1532     this.name = name;
1533     this.age = age;
1534     this.weight = weight;
1535     this.size = size;
1536 }
```

Estamos obligados a agregar el siguiente constructor a RussianTortoise usando la palabra reservada `super`:

```

1537 public RussianTortoise(String name, int age, double weight, double size) {
1538     super(name, age, weight, size);
1539 }
```

La palabra `super` se puede usar también de forma análoga al `this` pero para hacer referencia a la *clase padre* (y llamaremos *clase hija* a quien hereda de otra clase). Por ejemplo:

```

1540 public RussianTortoise(String name, int age, double weight, double size) {
1541     super(name, age, weight, size);
1542     super.sleep(100);
1543 }
```

Esta palabra `super` puede ser usada en cualquier método, no solamente en el constructor. Es importante que el `super` que funciona como pasamanos a la clase padre siempre sea la primera línea de nuestro constructor o tendremos un error de compilación. Tampoco es necesario solicitar por parámetros lo que pasaremos como pasamanos, sino que podemos pasar un valor manualmente:

```
1544 public RussianTortoise(int age, double weight, double size) {
1545     super("Bartolito", age, weight, size);
1546     super.sleep(100);
1547 }
```

4.5. La superclase `Object`

Cuando no usamos `extends`, por defecto Java hace que nuestra clase herede una la clase `Object`. Esta clase se comporta como la *superclase Object*. Todas nuestras instancias podrán acceder a los métodos que proporciona mediante el operador punto. Esta clase además no tiene atributos. La lista de algunos métodos que podemos usar es la siguiente:

Método	Descripción
<code>clone()</code>	Crea y devuelve una copia superficial del objeto actual.
<code>equals(Object obj)</code>	Compara el objeto actual con otro objeto para determinar si son iguales.
<code>finalize()</code>	Limpia el objeto antes de que sea reclamado por el recolector de basura.
<code>getClass()</code>	Devuelve la clase del objeto actual.
<code>hashCode()</code>	Devuelve el código hash del objeto actual.
<code>notify()</code>	Despierta un hilo en espera en el objeto actual.
<code>notifyAll()</code>	Despierta todos los hilos en espera en el objeto actual.
<code>toString()</code>	Devuelve una representación en forma de cadena de caracteres del objeto actual.
<code>wait()</code>	Hace que el hilo actual espere en el objeto actual.
<code>wait(long timeout)</code>	Hace que el hilo actual espere en el objeto actual durante un tiempo máximo especificado en milisegundos.
<code>wait(long timeout, int nanos)</code>	Hace que el hilo actual espere en el objeto actual durante un tiempo máximo especificado en milisegundos y nanosegundos.

El *código hash* es un código único que representa un objeto. Para calcularlo se utiliza como input la referencia a memoria del objeto, y gracias a esto, el código es único. Si el objeto se mueve en memoria debido a la recolección de basura o si se utiliza una técnica de optimización como la compresión de punteros, puede que este código cambie durante la ejecución del programa. El método `hashCode` nos proporciona este código, y junto al método `equals` nos permite identificar la igualdad entre instancias.

El método `equals` compara la referencia de memoria de dos objetos (el que utiliza el método y el que llega como parámetro) y devuelve `true` si son iguales. Sin embargo, hay ocasiones donde queremos considerar que dos instancias son iguales a pesar de que difieren en algún atributo. Por ejemplo, una instancia que tiene el estado actual de una tortuga, y luego, una instancia que tiene la misma tortuga pero con sus atributos del pasado, será diferente. Pero queda en responsabilidad del programador aclarar qué hace a dos tortugas iguales: ¿el nombre? ¿agregarle un atributo id? ¿que todos sus atributos sean iguales?. Podemos programar la forma en que funciona el `equals` haciendo una *sobreescritura de método*. Primero definimos los atributos en los cuales queremos verificar la igualdad (para el ejemplo usaremos todos) y luego usamos la anotación `Override` seguida de nuestra implementación:

```
1548 @Override
1549 public boolean equals(Object o) {
1550     if (this == o) return true;
1551     if (o == null || getClass() != o.getClass()) return false;
1552     Tortoise tortoise = (Tortoise) o;
1553     return getAge() == tortoise.getAge() && Double.compare(tortoise.getWeight(),
1554         getWeight()) == 0 && Double.compare(tortoise.getSize(), getSize()) == 0 &&
1555         getName().equals(tortoise.getName());
```

1554 }

En este caso, hemos considerado que `name` es un atributo obligatorio. Además la anotación `Override` es optativa, es una buena práctica usarla. El método `hashCode` también puede ser sobreescrito:

```
1555 @Override
1556 public int hashCode() {
1557     return Objects.hash(getName(), getAge(), getWeight(), getSize());
1558 }
```

4.6. Clases abstractas e interfaces

Supongamos que queremos usar el concepto de clases para reutilizar código. Por ejemplo, si ahora creamos otra clase llamada `Snake` para representar una serpiente, seguro comparta con `Tortoise` algunos atributos y métodos (y con estos, getters y setters, equals y hashCode para estos atributos). En POO contamos con el concepto de clase abstracta para esto. Vamos a crear una clase `Reptile` para esto, pero vamos a hacerlo como *clase abstracta*. El motivo es que la creamos para abstraer código común y eliminar esta repetición de código, pero a su vez no nos interesa tener instancias de esta clase. En POO, no podemos crear instancias de una clase abstracta.

```
1559 public abstract class Reptile {
1560
1561     private String species;
1562
1563     public Reptile(String species) {
1564         this.species = species;
1565     }
1566
1567     public void shedSkin() {
1568         System.out.println("New skin...");
1569     }
1570
1571     public String getSpecies() {
1572         return species;
1573     }
1574
1575     public void setSpecies(String species) {
1576         this.species = species;
1577     }
1578 }
```

Luego, debemos modificar `Tortoise` para que herede de esta clase:

```
1579 public class Tortoise extends Reptile {
1580
1581     private String name;
1582     private int age;
1583     private double weight;
1584     private double size;
1585
1586     public Tortoise(String species, String name, int age, double weight, double size)
1587     {
1588         super(species);
1589         this.name = name;
1590         this.age = age;
1591         this.weight = weight;
1592         this.size = size;
1593     }
1594 }
```

```
1593
1594     public void eat() {
1595         System.out.println("Looking for lettuce");
1596     }
1597
1598     public void sleep(int seconds) {
1599         System.out.println("Waking up after " + seconds + " seconds");
1600     }
1601
1602     public String getName() {
1603         return name;
1604     }
1605
1606     public void setName(String name) {
1607         this.name = name;
1608     }
1609
1610     public int getAge() {
1611         return age;
1612     }
1613
1614     public void setAge(int age) {
1615         this.age = age;
1616     }
1617
1618     public double getWeight() {
1619         return weight;
1620     }
1621
1622     public void setWeight(double weight) {
1623         this.weight = weight;
1624     }
1625
1626     public double getSize() {
1627         return size;
1628     }
1629
1630     public void setSize(double size) {
1631         this.size = size;
1632     }
1633
1634     @Override
1635     public boolean equals(Object o) {
1636         if (this == o) return true;
1637         if (o == null || getClass() != o.getClass()) return false;
1638         Tortoise tortoise = (Tortoise) o;
1639         return getAge() == tortoise.getAge() && Double.compare(tortoise.getWeight(),
1640             getWeight()) == 0 && Double.compare(tortoise.getSize(), getSize()) == 0 &&
1641             getName().equals(tortoise.getName());
1642     }
1643
1644     @Override
1645     public int hashCode() {
1646         return Objects.hash(getName(), getAge(), getWeight(), getSize());
1647     }
1648 }
```

Los métodos `equals` y `hashCode` pueden o no usar el nuevo atributo `species`.

Puede que no todos los reptiles cambien de piel de la misma forma, y para aclararlo podemos usar en

las clases hijas una sobreescritura de método sobre el método `shedSkin`. Sin embargo, podemos suponer que todas las especies lo hacen distinto y que cada una lo deba aclarar. Para esto, podemos usar un *método abstracto* el cual obligue a que las clases hijas sobreescriban el método siempre, dado que en la clase padre no definiremos una implementación. El código de `Reptile` quedaría:

```

1647 public abstract class Reptile {
1648
1649     private String species;
1650
1651     public Reptile(String species) {
1652         this.species = species;
1653     }
1654
1655     public abstract void shedSkin();
1656
1657     public String getSpecies() {
1658         return species;
1659     }
1660
1661     public void setSpecies(String species) {
1662         this.species = species;
1663     }
1664 }
```

Nuestra clase `Tortoise`, luego de este cambio, debe implementar el método de forma obligatoria. Los métodos abstractos tienen una sintaxis que permite indicar solo la firma del método. Además, las clases abstractas permiten la mezcla de métodos que son abstractos con algunos que no lo sean.

Si en algún momento todos los métodos se vuelven abstractos, y no tenemos herencia con otra clase, entonces lo que necesitamos utilizar es una *interfaz*. Las interfaces tienen todos sus métodos abstractos y públicos:

```

1665 public interface Animal {
1666
1667     public abstract void eat();
1668
1669 }
```

En este caso tenemos un solo método pero admite múltiples. Dado que todos sus métodos son abstractos y públicos, entonces `public` y `abstract` son optativos. Podemos reescribir la interfaz de la siguiente forma:

```

1670 public interface Animal {
1671
1672     void eat();
1673
1674 }
```

En cuanto a jerarquía tenemos las siguientes reglas:

- Una interfaz no puede heredar de una clase, pero una clase puede heredar de una interfaz.
- Una clase abstracta no puede heredar de una clase normal, pero una clase normal puede heredar de una clase abstracta.
- Java admite la herencia múltiple de interfaces, pero solo la herencia simple de clases (si tenemos dos métodos con misma firma en distintas clases padre pero con distinta implementación tendríamos un problema).
- Una clase puede heredar de una clase e implementar múltiples interfaces a la vez.

Vamos a modificar `Tortoise` para que implemente la interfaz `Animal`:

```
1675 public class Tortoise extends Reptile implements Animal {  
1676  
1677     private String name;  
1678     private int age;  
1679     private double weight;  
1680     private double size;  
1681  
1682     public Tortoise(String species, String name, int age, double weight, double size)  
1683     {  
1684         super(species);  
1685         this.name = name;  
1686         this.age = age;  
1687         this.weight = weight;  
1688         this.size = size;  
1689     }  
1690  
1691     @Override  
1692     public void eat() {  
1693         System.out.println("Looking for lettuce");  
1694     }  
1695  
1696     @Override  
1697     public void shedSkin() {  
1698         System.out.println("New tortoise skin...");  
1699     }  
1700  
1701     public void sleep(int seconds) {  
1702         System.out.println("Waking up after " + seconds + " seconds");  
1703     }  
1704  
1705     public String getName() {  
1706         return name;  
1707     }  
1708  
1709     public void setName(String name) {  
1710         this.name = name;  
1711     }  
1712  
1713     public int getAge() {  
1714         return age;  
1715     }  
1716  
1717     public void setAge(int age) {  
1718         this.age = age;  
1719     }  
1720  
1721     public double getWeight() {  
1722         return weight;  
1723     }  
1724  
1725     public void setWeight(double weight) {  
1726         this.weight = weight;  
1727     }  
1728  
1729     public double getSize() {  
1730         return size;  
1731     }
```

```

1732     public void setSize(double size) {
1733         this.size = size;
1734     }
1735
1736     @Override
1737     public boolean equals(Object o) {
1738         if (this == o) return true;
1739         if (o == null || getClass() != o.getClass()) return false;
1740         Tortoise tortoise = (Tortoise) o;
1741         return getAge() == tortoise.getAge() && Double.compare(tortoise.getWeight(),
1742             getWeight()) == 0 && Double.compare(tortoise.getSize(), getSize()) == 0 &&
1743             getName().equals(tortoise.getName());
1744     }
1745
1746     @Override
1747     public int hashCode() {
1748         return Objects.hash(getName(), getAge(), getWeight(), getSize());
1749     }
1750 }
```

En este ejemplo, agregamos `Override` sobre el método `eat`, implementamos `shedSkin` y dejamos los métodos `equals` y `hashCode` al final de la clase por prolíjidad.

El concepto de interfaz se extiende a partir de Java 8 con la introducción de las interfaces funcionales y los métodos por defecto, pero queda fuera del alcance de este apunte.

Conociendo ahora la jerarquía entre clases e interfaces, podemos aplicar *polimorfismo*, una característica muy importante de POO. Este concepto se define como la capacidad de tratar a un objeto mediante un tipo de dato más general. Veamos este ejemplo:

```

1749 public static void main(String[] args) {
1750     RussianTortoise russianTortoise = new RussianTortoise(1, 2, 3);
1751     Tortoise tortoise = russianTortoise;
1752     Animal animal = tortoise;
1753     Reptile reptile = tortoise;
1754     Object object = russianTortoise;
1755 }
```

Al hacer esto, estamos haciendo un casteo implícito (podemos usar el explícito de forma optativa, cuando sea posible, porque de no serlo dará un error en tiempo de ejecución). Si bien el contenido de cada variable es el mismo, solo podremos hacer uso de los métodos y atributos definidos en el tipo de dato, por lo que, mientras más general, menos cosas podemos hacer. Solo para aclarar, los nombres de las variables pueden ser cualquiera, pero por buenas prácticas se suele utilizar el mismo nombre que el tipo de dato empezando por una minúscula, logrando así un código más descriptivo.

Si nuestras subclases no sobreescriben comportamientos definidos en una clase padre, entonces nuestras subclases son clases *compatibles* con esas clases padres. Si esto se cumple, se mantiene la *correctitud* del programa y estaremos aplicando el *Principio de Liskov*.

Por último, en el ejemplo anterior, usamos el operador de asignación para guardar en una variable la *referencia a memoria* de un objeto, nunca el objeto en sí. Por lo que, si queremos crear una copia de un objeto, es necesario crear una instancia nueva de ese objeto y copiar los valores de sus atributos. El método `clone` no es de utilidad en este caso porque su copia es *superficial*, es decir, crea una nueva referencia en memoria pero al mismo objeto.

4.7. Manejo de errores

Un *error de sintaxis* es un error que se puede detectar cuando tipeamos en Java (por ejemplo, cuando nos olvidamos un punto y coma). Sin embargo, hay otros dos tipos de errores que son interesantes. Un *error en tiempo de compilación* es un error donde si bien la sintaxis es incorrecta, carece de la información necesaria

(por ejemplo, usar una variable que no existe). Por último, un *error en tiempo de ejecución* es un error que se detecta solo cuando el programa ya está funcionando.

Es en este último caso donde nos vamos a centrar. Existen errores, como por ejemplo, que el disco de la computadora se llene o que se intente crear un archivo en una carpeta que no tiene los permisos suficientes. Otros errores, como el tratar de leer un valor de un índice inválido de un arreglo, o un NPE cuando intentamos usar el operador punto de una variable que contiene `null`. Vamos a llamar a estos errores como *excepciones* y las más comunes que nos toparemos son:

1. **NullPointerException**: Se produce cuando se intenta acceder a un objeto que no ha sido inicializado o que se ha establecido como nulo.
2. **ArrayIndexOutOfBoundsException**: Se produce cuando se intenta acceder a un índice fuera de los límites de un array.
3. **ClassCastException**: Se produce cuando se intenta convertir un objeto a un tipo incompatible.
4. **NumberFormatException**: Se produce cuando se intenta convertir una cadena a un número, pero la cadena no tiene un formato numérico válido.
5. **ArithmaticException**: Se produce cuando se intenta realizar una operación matemática que no se puede realizar, como una división por cero.
6. **IllegalArgumentException**: Se produce cuando se pasa un argumento no válido a un método o constructor.
7. **IllegalStateException**: Se produce cuando el estado de un objeto no es compatible con la operación solicitada.
8. **IOException**: Se produce cuando ocurre un error al leer o escribir un archivo o un flujo de entrada/salida.

La idea es crear nuestra propia excepción. Para hacerlo debemos heredar de la clase `Exception`:

```

1756 public class TurtleWithoutShellException extends Exception {
1757
1758     private String message;
1759
1760     public TurtleWithoutShellException(String message) {
1761         super(message);
1762         this.message = message;
1763     }
1764
1765     public String getMessage() {
1766         return message;
1767     }
1768 }
```

Luego, esta clase puede ser tratada de forma normal, creando una instancia y usando la instancia cuando deseemos. Sin embargo, eso no será algo que ocasione un error real en nuestro programa. Para hacerlo, debemos apoyarnos en las palabras reservadas `throw` y `throws`.

```

1769 public static void main(String[] args) throws TurtleWithoutShellException {
1770     throw new TurtleWithoutShellException("Warning: The turtle needs a shell");
1771 }
```

La única línea de este método provoca un error, y se nos obliga a agregar en la firma del método una aclaración de que este error está contenido dentro de nuestro bloque de código. Podríamos sin embargo, hacer que esto suceda solo bajo ciertas condiciones usando condicionales. Por otro lado, si la función en la que estamos no es la función `main`, puede que necesitemos usarla desde alguna otra función. En ese caso, debemos colocar en cada función que se llame entre sí, esto a su firma. Por ejemplo:

```

1772 public static void main(String[] args) throws TurtleWithoutShellException {
1773     f();
1774 }
1775
1776 public static void f() throws TurtleWithoutShellException {
1777     throw new TurtleWithoutShellException("Warning: The turtle needs a shell");
1778 }
```

Sin embargo, podemos decidir *atrapar* este error en algún lado y darle un *tratamiento*. La sentencia `try` y `catch`, nos permite manejar excepciones. En el bloque de código de `try` tendremos código y que se intentará ejecutar sin problemas. El bloque de código del `catch` solo se activará si se *lanza* una excepción con `throw`.

```

1779 public static void main(String[] args) {
1780     try {
1781         f();
1782     } catch (TurtleWithoutShellException e) {
1783         System.out.println(e.getMessage());
1784     }
1785 }
1786
1787 public static void f() throws TurtleWithoutShellException {
1788     throw new TurtleWithoutShellException("Warning: The turtle needs a shell");
1789 }
```

En el `catch` usamos `TurtleWithoutShellException e` para que la instancia de la excepción que representa un error quede dentro de la variable `e` (Suele usarse simplemente `e` para el nombre de las excepciones en un `catch`), y esta instancia puede ser usada dentro del bloque `catch`. Sin embargo, podemos usar herencia y en lugar de `TurtleWithoutShellException e` podríamos escribir `Exception e`. Esto es mucho más general, se recomienda ser lo más específico que se pueda.

Podemos además, agregar la palabra reservada `finally` con su propio bloque de código, que se ejecuta de todas formas aunque se llegue al `catch` o no:

```

1790 public static void main(String[] args) {
1791     try {
1792         f();
1793     } catch (TurtleWithoutShellException e) {
1794         System.out.println(e.getMessage());
1795     } finally {
1796         System.out.println("Finally block executed");
1797     }
1798 }
```

Podemos además usar un OR entre varias excepciones o, colocar varios `catch` para tratar cada error de forma separada:

```

1799 public static void main(String[] args) {
1800     try {
1801         f();
1802     } catch (TurtleWithoutShellException | ArrayIndexOutOfBoundsException e) {
1803         System.out.println("Caught exception: " + e.getMessage());
1804     } catch (Exception e) {
1805         System.out.println("Caught exception: " + e.getMessage());
1806     } finally {
1807         System.out.println("Finally block executed");
1808     }
1809 }
```

Se recomienda colocar más abajo los casos generales, dado que los `catch` tienen orden de prioridad del superior al inferior, y si se coloca una excepción general en la parte superior, nunca ejecutará las de la parte inferior.

Existe un caso especial en Java, que es cuando intentamos consumir un recurso (como lo hicimos con `Scanner` en secciones anteriores). En estos casos, podemos usar lo que se conoce como `try-with-resources`:

```
1810 public static void main(String[] args) {
1811     try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
1812         String line;
1813         while ((line = br.readLine()) != null) {
1814             System.out.println(line);
1815         }
1816     } catch (IOException e) {
1817         System.err.println("Error reading file: " + e.getMessage());
1818     }
1819 }
```

El ejemplo anterior lee un archivo en la ruta del proyecto e intenta leerlo. En este caso, el recurso solicitado se cierra independientemente de si hubo o no una excepción, luego de ejecutar estos bloques de código.

Por último, contamos con excepciones especiales, que se crean heredando de `RuntimeException`. Estas excepciones tienen la particularidad de que no necesitan usar `throws` en la firma del método o atraparse. Entonces, las podemos lanzar en cualquier parte del código y esperar a que en alguna otra parte alguien las atrape o no (esto evita llenar las firmas con excepciones). Cuando una firma se llena de múltiples excepciones, estas se separan con coma.

Se pueden crear instancias de `Exception` y de `RuntimeException` sin necesidad de crear una clase que herede de ellas. Por ejemplo:

```
1820 public static void main(String[] args) {
1821     f();
1822 }
1823
1824 public static f() {
1825     throw new RuntimeException("Example");
1826 }
```

4.8. Operador For-each

Este tipo de ciclo es utilizado para recorrer estructuras. Para hacerlo, esta estructura debe ser un arreglo nativo o bien, ser una clase que implemente la interfaz `Iterable`. En general, son las *colecciones* las que pueden ser iteradas con este ciclo.

Veamos un ejemplo con arrays nativos:

```
1827 public static void print(int[] array) {
1828     for (int num : array) {
1829         System.out.println(num);
1830     }
1831 }
```

Perdemos la capacidad de contar por qué iteración vamos, lo que es útil para dejar el código más limpio cuando no necesitamos conocer el número de iteraciones.

4.9. Wrappers

Un *wrapper* es un tipo de dato representado por una clase que `.envuelve.`^a un tipo de dato primitivo para darle la capacidad de un objeto, como por ejemplo aceptar `null` o contar con métodos.

Cuadro 4.1: Tipos de datos nativos y sus tipos de datos wrapper en Java

Tipo de dato nativo	Tipo de dato wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character
void	Void

Contamos con algunos métodos útiles como `Integer.parseInt(String s)` que intenta convertir una String a un número y de no poderse lanza un error. O como la abstracción `Number` que es el padre de los tipos de datos numéricos.

4.10. Ejercicios

Ejercicio 4.10.1: Complejos

Desarolla una interfaz `Complex` y escriba al menos 5 firmas de métodos posibles. No es necesaria su implementación.

Ejercicio 4.10.2: Irracionales

Desarolla una interfaz `Irrational` y escriba al menos 5 firmas de métodos posibles. No es necesaria su implementación.

Ejercicio 4.10.3: Usuario

Desarolla una interfaz `User` y escriba al menos 5 firmas de métodos posibles. No es necesaria su implementación.

Ejercicio 4.10.4: Polinomio

Sean $a, b, c \in \mathbb{R}$, variables, crear una clase `Polynomial` que represente un polinomio de grado 2 usando estas variables como coeficientes. Agregue un método para indicar las raíces si es que se puede.

† Agregue otro método que muestre en pantalla las coordenadas del vértice.

Ejercicio 4.10.5: Círculo

Crear una clase `Circle` que represente un círculo. Agregue los atributos que considere necesario. Cree un método para obtener su área y otro para obtener su perímetro.

† Agregue un método que reciba como parámetro otra instancia de `Circle` y devuelva `true` en el caso de que ambos tengan intersección.

Ejercicio 4.10.6: Dupla

Cree el TDA `Coordinate`, el cual almacena los valores x e y de un punto en el plano. Agréguele un método que reciba como parámetro otro punto del plano e imprima en pantalla su distancia.

† Cree un TDA `Couple` que recibe dos datos del mismo tipo como parámetro y los almacena. Agregue los métodos `head` y `tail` que devuelven el primer y último elemento respectivamente.

Ejercicio 4.10.7: Vector (físico)

Cree el TDA `Vector`, el cual almacena los valores x e y de un punto en el plano. Agréguele un método que reciba como parámetro otro vector y calcule su producto punto.

† Dada una lista de `Vector` que corresponden con los lados de un polígono, calcular su perímetro. Sin desarrollar, ¿qué estrategia usaría para calcular su área?

Ejercicio 4.10.8: Áreas

Cree el TDA `Square`, el cual almacena los valores (x_1, y_1) e (x_2, y_2) que son sus extremos diagonales, de izquierda a derecha. Agréguele un método que reciba como parámetro un punto y devuelva `true` solamente si el punto está dentro de su región.

† Suponga un círculo de radio $r = \frac{l}{2}$, donde l es el tamaño de uno de los lados del cuadrado, use el método Monte Carlo para calcular el área del círculo.

Capítulo 5

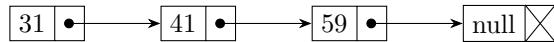
Estructuras de datos lineales

5.1. Tipo de Dato Abstracto

Un *Tipo de Dato Abstracto* (o *TDA*) es una estructura que encapsula un tipo de dato y le otorga un comportamiento específico. Esto coincide con el funcionamiento de las interfaces en POO, por lo que *definiremos* un TDA y podremos definir múltiples implementaciones.

5.2. Lista enlazada

Los arreglos son una secuencia de elementos que poseen un índice que representa su posición. Vamos a usar entonces el concepto de *cursor*, el cuál permite mediante referencias indicar que elemento es el siguiente de un elemento dado. La unidad mínima es el *Nodo*. Cada Nodo tiene un valor contenido y una referencia al siguiente nodo. Debe existir además el concepto de *referencia vacía* dado que de lo contrario, solo podríamos tener secuencias infinitas de nodos. Con una referencia vacía, que no apunta a ningún nodo, podemos establecer un límite en la cantidad de elementos guardados.



Un Nodo en Java lo podemos definir de la siguiente forma:

```
1832 public class Node {  
1833  
1834     private int value;  
1835     private Node next;  
1836  
1837     public Node(int value, Node next) {  
1838         this.value = value;  
1839         this.next = next;  
1840     }  
1841  
1842     public int getValue() {  
1843         return value;  
1844     }  
1845  
1846     public void setValue(int value) {  
1847         this.value = value;  
1848     }  
1849  
1850     public Node getNext() {  
1851         return next;
```

```

1852     }
1853
1854     public void setNext(Node next) {
1855         this.next = next;
1856     }
1857 }
```

Al tener setters podemos decir que esta es una clase mutable, y que además se tiene a sí misma como atributo. A esto último se lo conoce como *atributo autorreferencial*.

Una lista puede tener métodos para verificar si tiene elementos repetidos, si es palíndromo, etc. La idea es identificar el conjunto mínimo de operaciones que nos permitan generar al resto. Una lista se caracteriza por tener un orden, admitir elementos repetidos, tener una longitud, etc. Entonces, para definir el TDA Lista Enlazada, debemos crear una interfaz que contenga estas operaciones mínimas:

```

1858 /**
1859 * Precondicion: para usar cualquier de estos metodos la estructura debe estar
1860     inicializada.
1861 */
1862 public interface ILinkedList {
1863
1864     /**
1865      * Agregar un elemento al en la ultima posicion.
1866      * @param value contiene el valor a guardar en la ultima posicion.
1867      */
1868     void add(int value);
1869
1870     /**
1871      * Postcondicion: Inserta un elemento solo si el indice es valido.
1872      * @param index valido si esta en el rango [0, length], con length la longitud de
1873          la lista.
1874      * @param value contiene el valor a guardar en la posicion indicada.
1875      */
1876     void insert(int index, int value);
1877
1878     /**
1879      * Postcondicion: Elimina el elemento si existe. De lo contrario no hace nada.
1880      * @param index positivo o cero, menor que la longitud de la lista.
1881      */
1882     void remove(int index);
1883
1884     /**
1885      * @param index valido si esta en el rango [0, length], con length la longitud de
1886          la lista.
1887      * @return valor que se encuentra en la posicion indicada.
1888      */
1889     int get(int index);
1890
1891     /**
1892      * @return longitud de la lista.
1893      */
1894     int size();
1895
1896     /**
1897      * @return <code>true</code> solo cuando la lista esta vacia, <code>false</code>
1898          en otro caso.
1899      */
1900     boolean isEmpty();
1901 }
```

La interfaz es nombrada `ILinkedList` donde la *I*- es para indicar que es una interfaz. Esta práctica es solo para evitar ambigüedad, dado que en un momento veremos la implementación y los nombres de clases e interfaces no pueden repetirse entre sí. Además se han colocado comentarios de documentación siguiendo la sintaxis de Javadoc, donde `@return` es otra forma de aclarar postcondiciones. Nuestras interfaces, en el mejor caso, deberían contener aclaraciones del tipo *precondición* y *postcondición*.

La operación que inserta un elemento en cualquier lado de la lista puede ser reemplazada por una operación que solo coloque el elemento al final. Esto también sería un conjunto mínimo de operaciones que permite generar todas las demás operaciones. Por ejemplo, con el método actual para colocar al final podemos primero contar el total de elementos y luego colocarlo en la última posición. En cambio, si tenemos la función que coloca el elemento al final, podemos partir la lista en dos listas donde al colocar el elemento al final de la primera parte y luego unirlas, de el efecto de que lo colocamos en la posición indicada.

Las listas enlazadas además tienen una particularidad: no podemos acceder a un elemento a una posición dada sin haber pasado por los anteriores. Esto no es lo que sucede con los arreglos. En un arreglo, leer un elemento en una posición, por ejemplo, 90, implica acceder a esa posición de forma directa. Mientras tanto, en una lista enlazada, leer un elemento en la posición 90 implica haber recorrido los 89 elementos anteriores antes de llegar a este.

Ahora que tenemos la interfaz, podemos definir diferentes implementaciones. En este caso se dará una posible implementación:

```

1898 public class LinkedList implements ILinkedList {
1899
1900     private Node first;
1901     private int size;
1902
1903     public LinkedList() {
1904         this.size = 0;
1905     }
1906
1907     @Override
1908     public void add(int value) {
1909         Node newNode = new Node(value, null);
1910         if(this.first == null) {
1911             this.first = newNode;
1912             this.size++;
1913             return;
1914         }
1915         Node candidate = this.first;
1916         while(candidate.getNext() != null) {
1917             candidate = candidate.getNext();
1918         }
1919         candidate.setNext(newNode);
1920         this.size++;
1921     }
1922
1923     @Override
1924     public void insert(int index, int value) {
1925         if(index < 0 || index > size) {
1926             System.out.println("Error, rango excedido");
1927             return;
1928         }
1929         if(index == 0) {
1930             this.first = new Node(value, this.first);
1931             return;
1932         }
1933         int count = 1;
1934         Node candidate = this.first;
1935         while(count != index) {
1936             candidate = candidate.getNext();

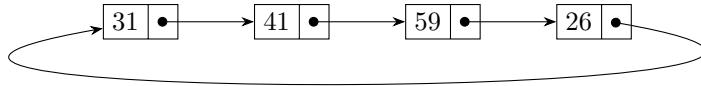
```

```
1937         count++;
1938     }
1939     candidate.setNext(new Node(value, candidate.getNext()));
1940 }
1941
1942 @Override
1943 public void remove(int index) {
1944     if(this.first == null || index < 0 || index > size) {
1945         System.out.println("Error, rango excedido");
1946         return;
1947     }
1948     if(index == 0) {
1949         this.first = this.first.getNext();
1950         return;
1951     }
1952     int count = 1;
1953     Node candidate = this.first;
1954     while(count != index) {
1955         candidate = candidate.getNext();
1956         count++;
1957     }
1958     candidate.setNext(candidate.getNext().getNext());
1959 }
1960
1961 @Override
1962 public int get(int index) {
1963     if(this.first == null) {
1964         System.out.println("Lista vacía");
1965         return -1;
1966     }
1967     if(index == 0) {
1968         return this.first.getValue();
1969     }
1970     int count = 0;
1971     Node candidate = this.first;
1972     while(count != index) {
1973         candidate = candidate.getNext();
1974         count++;
1975     }
1976     return candidate.getValue();
1977 }
1978
1979 @Override
1980 public int size() {
1981     return this.size;
1982 }
1983
1984 @Override
1985 public boolean isEmpty() {
1986     return this.size == 0;
1987 }
1988 }
```

Es importante aclarar que suelen nombrarse como *head* y *tail* como el valor de la primera posición y la sublista que contiene desde el segundo elemento en adelante.

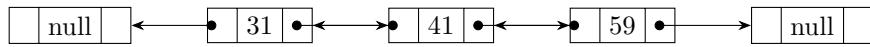
5.3. Lista enlazada cíclica

El propósito de esta lista es reducir la complejidad de algunas operaciones como la del borrado de un nodo, dado que se puede generalizar el método de guardar el anterior. La idea es sencilla: agregamos en el nodo final una referencia al nodo inicial. Es necesario almacenar el total de elementos, dado que de lo contrario, si solo recorremos no podríamos saber si volvimos al principio de la lista.



5.4. Lista doblemente enlazada

En varias secciones del código necesitamos recordar el Nodo anterior al Nodo en el que estamos parados. Esto lo podríamos simplificar si el Nodo mismo fuese el que almacena tanto el Nodo siguiente como el Nodo anterior.



5.5. Lista cíclica doblemente enlazada

Si tenemos una lista de n elementos, para llegar al elemento en la posición $n - 1$ debemos recorrer al menos $n - 2$ elementos si estamos en una lista enlazada. Si pudiésemos recorrer la lista hacia atrás y aparecer en el otro extremo, solo deberíamos recorrer 1 elemento. Entonces, en una lista enlazada el peor escenario es buscar el elemento en la última posición, mientras que en una lista cíclica doblemente enlazada solo deberíamos recorrer la mitad de los elementos en el peor escenario.

5.6. Pilas

Una Pila es una estructura de datos que tiene un *tope* y un *cuerpo*. Para acceder a este cuerpo si o si debemos pasar primero por el tope. Podemos imaginar una pila de libros, o una pila de objetos.

$$p = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}$$

Cuando *apilamos* un elemento a a la pila p este queda en la parte superior de la Pila:

$$p = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ a \end{bmatrix}$$

Si luego queremos apilar un elemento b este será el nuevo tope:

$$p = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ b \\ a \end{bmatrix}$$

Por otro lado, una pila solo nos permite ver el elemento superior. En este caso, solo podríamos acceder al elemento *b*. Para acceder al elemento *a*, debería *desapilar* la pila. Si desapilamos nos queda:

$$p = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ a \end{bmatrix}$$

Y ahora, dado que *a* es el nuevo tope, podemos ver este elemento. Tratar de leer el tope de una *Pila vacía* trae problemas, y lo normal es esperar un error. Entonces, para evitarlo, debemos poder saber si una pila es vacía o no.

Además vamos a contemplar un método que *inicialice* la pila. Inicializar es la acción de preparar una pila nueva, es similar al concepto de *crear* pero no necesariamente son lo mismo. El inicializar puede crear la Pila pero además prepara las variables internas que va a utilizar, funcionando como un reseteo interno para la Pila.

La estructura Pila entonces se puede definir de la siguiente forma:

```

1989 /**
1990 * Precondicion: para usar cualquier de estos metodos la estructura debe estar
1991 *               inicializada.
1992 */
1993 public interface IStack {
1994
1995     /**
1996      * Postcondicion: Apila (coloca en el tope) un valor.
1997      * @param a valor a apilar.
1998     */
1999     void add(int a);
2000
2001     /**
2002      * Precondicion: La pila no esta vacia.
2003      * Postcondicion: Desapila (quita el tope).
2004     */
2005     void remove();
2006
2007     /**
2008      * @return <code>true</code> si la pila esta vacia, <code>false</code> en otro
2009      *       caso.
2010     */
2011     boolean isEmpty();
2012
2013     /**
2014      * @return Devuelve el tope.
2015     */
2016     int getTop();
}

```

5.7. Pila - Implementación estática

Una implementación estática de una Pila, para definirse como tal debe almacenar los datos internamente en una estructura de tamaño fijo. Por ejemplo, podemos usar un arreglo o incluso una implementación estática de Pila diferente a esta, o cualquier estructura que seguiremos viendo siempre que su longitud sea fija.

Una posible implementación es:

```

2017 public class Stack implements IStack {
2018
2019     private final int[] array;
2020     private int count;
2021
2022     public Stack() {
2023         this.array = new int[10000];
2024         this.count = 0;
2025     }
2026
2027     @Override
2028     public void add(int a) {
2029         this.array[this.count] = a;
2030         this.count++;
2031     }
2032
2033     @Override
2034     public void remove() {
2035         if(count == 0) {
2036             System.out.println("Error, no se puede desapilar una pila vacia");
2037             return;
2038         }
2039         this.count--;
2040     }
2041
2042     @Override
2043     public boolean isEmpty() {
2044         return this.count == 0;
2045     }
2046
2047     @Override
2048     public int getTop() {
2049         if(count == 0) {
2050             System.out.println("Error, no se puede obtener el tope de una pila vacia")
2051             ;
2052             return -1;
2053         }
2054         return this.array[this.count - 1];
2055     }
2056 }
```

Aquí el número 10000 es anodino, es solo para representar un tamaño suficientemente grande como para no controlar el error. Aunque un código más complejo pero que contemple los posibles errores (por ejemplo, no permitir insertar un elemento en una Pila llena), sería más conveniente.

Por otro lado, se devuelve -1 para representar un error por razones históricas. También sería más apropiado lanzar una excepción.

5.8. Pila - Implementación dinámica

Una implementación dinámica implica el uso de una lista enlazada. Más apropiadamente, el uso de Nodos. En este caso, un arreglo es eficiente para acceder a los elementos, pero su tamaño fijo hace que ocupe mucha memoria. En cambio, una lista interna formada por Nodos es más lento al leer una Nodo que está alejado del Nodo inicial, pero ocupa en memoria solo los Nodos que fueron necesarios utilizar.

Por otro lado, crear un Nodo y luego ignorarlo en el código nos da un poco de eficiencia, dado que si estos Nodos no se utilizan más, la JVM de Java invocará al *garbage collector* para que elimine de la memoria

este Nodo.

Una posible implementación dinámica es:

```

2057 public class Stack implements IStack {
2058
2059     private Node first;
2060
2061     @Override
2062     public void add(int a) {
2063         this.first = new Node(a, this.first);
2064     }
2065
2066     @Override
2067     public void remove() {
2068         if(this.first == null) {
2069             System.out.println("No se puede desapilar una pila vacía");
2070             return;
2071         }
2072         this.first = this.first.getNext();
2073     }
2074
2075     @Override
2076     public boolean isEmpty() {
2077         return this.first == null;
2078     }
2079
2080     @Override
2081     public int getTop() {
2082         if(this.first == null) {
2083             System.out.println("No se puede obtener el tope una pila vacía");
2084             return -1;
2085         }
2086         return this.first.getValue();
2087     }
2088 }
```

En esta estructura no necesitamos inicializar el Nodo con un valor, podemos simplemente dejarlo en `null` que es el valor por defecto al no asignarle un valor. Entonces, el constructor funciona como el método *inicializar* tanto aquí como en la implementación estática. En este caso no está escrito porque el constructor por defecto es más que suficiente.

5.9. Pila - Operaciones

Recorrer una Pila implica destruirla. Veamos:

```

2089 public static void main(String[] args) {
2090     Stack stack = new Stack();
2091     stack.add(1);
2092     stack.add(2);
2093     stack.add(3);
2094
2095     while(!stack.isEmpty()) {
2096         System.out.println(stack.getTop());
2097         stack.remove();
2098     }
2099 }
```

Aquí llenamos la Pila con números enteros dado que es el tipo de dato que indicamos en su estructura que va a almacenar (podríamos modificar este tipo de dato, por ejemplo, para tener una Pila de strings). Además, luego de imprimir los elementos en pantalla, la Pila quedará vacía (es la condición de corte del ciclo). Esto no es lo que sucedía con las listas enlazadas y es entonces necesario introducir el concepto de *estructuras destructivas*.

Una estrategia para resolver este problema es trabajar con copias. Creamos una copia sin modificar (en realidad aparentamos no modificar) la original.

```

2100 public static IStack copy(IStack stack) {
2101     IStack copy = new Stack();
2102     IStack aux = new Stack();
2103     while (!stack.isEmpty()) {
2104         aux.add(stack.getTop());
2105         stack.remove();
2106     }
2107     while (!aux.isEmpty()) {
2108         stack.add(aux.getTop());
2109         copy.add(aux.getTop());
2110         aux.remove();
2111     }
2112     return copy;
2113 }
```

En esta función volcamos los elementos de una Pila `stack` a las pilas `aux`. Cuando lo hacemos, la pila quedará invertida, dado que el tope de la pila pasa a ser el primero en apilarse y quedar en el fondo de `aux`. Esto es lo que se conoce como *LIFO* (Last In First Out). Invertir una pila invertida la deja nuevamente como se encontraba, por lo que reponemos la pila original que se vacío, pero con el mismo elemento que leímos vamos llenando la copia que vamos a retornar. Por lo tanto, pasar elementos de una pila a otra implica que la pila resultante queda invertida. Pero, podemos aplicarlo para invertir los elementos de la Pila misma:

```

2114 public static void revert(IStack stack) {
2115     IStack aux = new Stack();
2116     IStack aux2 = new Stack();
2117     while (!stack.isEmpty()) {
2118         aux.add(stack.getTop());
2119         stack.remove();
2120     }
2121     while (!aux.isEmpty()) {
2122         aux2.add(aux.getTop());
2123         aux.remove();
2124     }
2125     while (!aux2.isEmpty()) {
2126         stack.add(aux2.getTop());
2127         aux2.remove();
2128     }
2129 }
```

5.10. Cola

Una Cola es una estructura de datos que tiene un primer elemento y sus siguientes. Para acceder debemos acceder por su primer elemento. Para poder ver el siguiente elemento debemos *desacolar* este elemento. De la misma forma que en una Pila podemos apilar, en una Cola podemos *acolar* pero considerando que en las colas el elemento entrante se coloca en la última posición.

$$c = \xrightarrow{\quad} [\emptyset] [\emptyset] [\emptyset] [\emptyset]$$

Podemos acolar un primer elemento a para que este quede como el *primero* de la cola:

$$c = \overrightarrow{[\emptyset] [\emptyset] [\emptyset] [a]}$$

Luego, acolar un segundo elemento b coloca este elemento al final:

$$c = \overrightarrow{[\emptyset] [\emptyset] [b] [a]}$$

Por último, desacolar quita el primer elemento:

$$c = \overrightarrow{[\emptyset] [\emptyset] [\emptyset] [b]}$$

Además vamos a considerar como *cola vacía* aquella que no tiene elementos. Por otro lado, necesitamos una forma de *inicializar* la Cola para que esta exista. Para esto vamos a usar su constructor.

El TDA será:

```

2130 /**
2131 * Precondicion: para usar cualquier de estos metodos la estructura debe estar
2132 * inicializada.
2133 */
2134 public interface IQueue {
2135
2136     /**
2137     * Postcondicion: Acola (coloca en el final de la cola) un valor.
2138     *
2139     * @param a valor a acolar.
2140     */
2141     void add(int a);
2142
2143     /**
2144     * Precondicion: La cola no esta vacia.
2145     * Postcondicion: Desacola (quita el primer elemento).
2146     */
2147     void remove();
2148
2149     /**
2150     * @return <code>true</code> si la cola esta vacia, <code>false</code> en otro
2151     * caso.
2152     */
2153     boolean isEmpty();
2154
2155     /**
2156     * @return Devuelve el primero.
2157     */
2158     int getFirst();
2159 }
```

5.11. Cola - Implementación estática

De la misma forma que sucede con Pila, necesitamos un arreglo para mantener esta estructura con tamaño fijo.

```

2159 public class Queue implements IQueue {
2160
2161     private final int[] array;
```

```

2162     private int count;
2163
2164     public Queue() {
2165         this.array = new int[10000];
2166         this.count = 0;
2167     }
2168
2169     @Override
2170     public void add(int a) {
2171         this.array[this.count] = a;
2172         this.count++;
2173     }
2174
2175     @Override
2176     public void remove() {
2177         if (count == 0) {
2178             System.out.println("Error, no se puede desacolar una cola vacia");
2179             return;
2180         }
2181         for (int i = 0; i < this.array.length - 1; i++) {
2182             this.array[i] = this.array[i + 1];
2183         }
2184         this.count--;
2185     }
2186
2187     @Override
2188     public boolean isEmpty() {
2189         return this.count == 0;
2190     }
2191
2192     @Override
2193     public int getFirst() {
2194         if (count == 0) {
2195             System.out.println("Error, no se puede obtener el primero de una cola
2196                         vacia");
2197             return -1;
2198         }
2199         return this.array[0];
2200     }
}

```

Existen múltiples formas de implementarlo. En este caso, elegimos que el elemento a tomar siempre se encuentre en la primera posición. Esto hace que el método para eliminar sea más difícil de implementar que método para agregar. Pero, podríamos haber elegido una implementación donde el elemento siempre este en su última posición. En este caso, el agregar sería más difícil de implementar que el eliminar. Dependiendo cual será el uso más común para esta estructura, podemos elegir entre una y otra.

5.12. Cola - Implementación dinámica

Utilizaremos el mismo Nodo que utilizamos para Pila. Sin embargo, la implementación debe cumplir con la definición de Cola:

```

2201     public class Queue implements IQueue {
2202
2203         private Node first;
2204
2205         @Override
2206         public void add(int a) {

```

```

2207     Node node = new Node(a, null);
2208     if (this.first == null) {
2209         this.first = node;
2210         return;
2211     }
2212     Node candidate = this.first;
2213     while (candidate.getNext() != null) {
2214         candidate = candidate.getNext();
2215     }
2216     candidate.setNext(node);
2217 }
2218
2219 @Override
2220 public void remove() {
2221     if (this.first == null) {
2222         System.out.println("No se puede desacolar una cola vacía");
2223         return;
2224     }
2225     this.first = this.first.getNext();
2226 }
2227
2228 @Override
2229 public boolean isEmpty() {
2230     return this.first == null;
2231 }
2232
2233 @Override
2234 public int getFirst() {
2235     if (this.first == null) {
2236         System.out.println("No se puede obtener el primero una cola vacía");
2237         return -1;
2238     }
2239     return this.first.getValue();
2240 }
2241 }
```

5.13. Cola - Operaciones

La forma de recorrer una Cola es destructiva, al igual que la Pila. La forma de recorrerla es la siguiente:

```

2242 public static void print(IQueue queue) {
2243     while (!queue.isEmpty()) {
2244         System.out.println(queue.getFirst());
2245         queue.remove();
2246     }
2247 }
```

Las colas son estructuras de tipo *FIFO* (First In First Out), por lo que al pasar elementos de una Cola a otra estos elementos quedan en el mismo orden.

La copia, a diferencia de la Pila, no necesita dos ciclos, porque tal como dijimos las colas dejan los elementos en su misma posición. Entonces, la diferencia estará en el primer ciclo:

```

2248 public static IQueue copy(IQueue queue) {
2249     IQueue copy = new Queue();
2250     IQueue aux = new Queue();
2251     while (!queue.isEmpty()) {
2252         aux.add(queue.getFirst());
2253         copy.add(queue.getFirst());
```

```

2254         queue.remove();
2255     }
2256     while (!aux.isEmpty()) {
2257         queue.add(aux.getFirst());
2258         aux.remove();
2259     }
2260     return copy;
2261 }
```

Revertir una Cola ahora es más complicado, dado que no se revierten al volcarse entre sí. Una forma sencilla de hacerlo, es pasar sus elementos a una Pila y luego de la Pila volverlos a pasar a la Cola:

```

2262 public static void revert(IQueue queue) {
2263     IStack aux = new Stack();
2264     while (!queue.isEmpty()) {
2265         aux.add(queue.getFirst());
2266         queue.remove();
2267     }
2268     while (!aux.isEmpty()) {
2269         queue.add(aux.getTop());
2270         aux.remove();
2271     }
2272 }
```

5.14. Cola con prioridad

Las colas con prioridad almacenan *dúplas* de elementos. Estas duplas son de la forma `(valor, prioridad)`. En el siguiente gráfico se visualiza la prioridad por debajo del valor.

La particularidad de esta estructura es que al agregarse un elemento, sus prioridades siguen el algoritmo de *ordenamiento por inserción*. Cuando varias prioridades iguales quedan de forma consecutiva, lo que hacemos es mantener el comportamiento normal de una Cola.

El TDA queda:

```

2273 /**
2274  * Precondicion: para usar cualquier de estos métodos la estructura debe estar
2275  * inicializada.
2276 */
2277 public interface IPriorityQueue {
2278
2279     /**
2280      * Postcondicion: Acola (coloca en el final de la cola) un valor, respecto a su
2281      * prioridad.
2282      *
2283      * @param a      valor a acolar.
2284      * @param priority prioridad del valor a agregar.
2285      */
2286     void add(int a, int priority);
2287
2288     /**
2289      * Precondicion: La cola no está vacía.
2290      * Postcondicion: Desacola (quita el primer elemento).
2291      */
2292     void remove();
2293
2294     /**
2295      * @return <code>true</code> si la cola está vacía, <code>false</code> en otro
2296      * caso.
2297 }
```

```

2294     */
2295     boolean isEmpty();
2296
2297     /**
2298      * @return Devuelve el primero.
2299      */
2300     int getFirst();
2301
2302     /**
2303      * @return Devuelve la prioridad del primero.
2304      */
2305     int getPriority();
2306
2307 }

```

5.15. Cola con prioridad - Implementación estática

Para la implementación estática solo necesitamos un tamaño fijo de almacenamiento de datos. Podemos optar por usar un arreglo de duplas, una matriz de dos filas, o como en este caso, dos arreglos independientes entre sí:

```

2308 /**
2309 * Esta implementacion no acepta valores repetidos en las prioridades,
2310 * de lo contrario la busqueda binaria no seria valida
2311 */
2312 public class PriorityQueue implements IPriorityQueue {
2313
2314     private static final int LENGTH = 10000;
2315
2316     private final int[] array;
2317     private final int[] priorities;
2318     private int count;
2319
2320     public PriorityQueue() {
2321         this.array = new int[LENGTH];
2322         this.priorities = new int[LENGTH];
2323         this.count = 0;
2324     }
2325
2326     @Override
2327     public void add(int a, int priority) {
2328         int index = binarySearch(priority);
2329
2330         for (int i = this.count; i > index; i--) {
2331             this.array[i] = this.array[i - 1];
2332             this.priorities[i] = this.priorities[i - 1];
2333         }
2334
2335         this.array[index] = a;
2336         this.priorities[index] = priority;
2337
2338         this.count++;
2339     }
2340
2341     // Solo para prioridades no repetidas, sino debe utilizarse otra tecnica
2342     private int binarySearch(int priority) {
2343         int low = 0;

```

```
2344     int high = this.count - 1;
2345
2346     while (low <= high) {
2347         int mid = (low + high) / 2;
2348         if (this.priorities[mid] == priority) {
2349             return mid;
2350         } else if (this.priorities[mid] < priority) {
2351             low = mid + 1;
2352         } else {
2353             high = mid - 1;
2354         }
2355     }
2356
2357     return low;
2358 }
2359
2360 @Override
2361 public void remove() {
2362     if (count == 0) {
2363         System.out.println("Error, no se puede desacolar una cola vacia");
2364         return;
2365     }
2366     for (int i = 0; i < this.array.length - 1; i++) {
2367         this.array[i] = this.array[i + 1];
2368         this.priorities[i] = this.priorities[i + 1];
2369     }
2370     this.count--;
2371 }
2372
2373 @Override
2374 public boolean isEmpty() {
2375     return this.count == 0;
2376 }
2377
2378 @Override
2379 public int getFirst() {
2380     if (count == 0) {
2381         System.out.println("Error, no se puede obtener el primero de una cola
2382                         vacia");
2383         return -1;
2384     }
2385     return this.array[0];
2386 }
2387
2388 @Override
2389 public int getPriority() {
2390     if (count == 0) {
2391         System.out.println("Error, no se puede obtener la prioridad del primero de
2392                         una cola vacia");
2393         return -1;
2394     }
2395     return this.priorities[0];
2396 }
```

5.16. Cola con prioridad - Implementación dinámica

```
2396 public class PriorityQueue implements IPriorityQueue {  
2397  
2398     private PriorityNode first;  
2399  
2400     @Override  
2401     public void add(int a, int priority) {  
2402         if (this.first == null) { // 0 elements  
2403             this.first = new PriorityNode(a, priority, null);  
2404             return;  
2405         }  
2406  
2407         if (this.first.getNext() == null) { // 1 element  
2408             if (this.first.getPriority() > priority) {  
2409                 this.first = new PriorityNode(a, priority, this.first);  
2410             } else {  
2411                 this.first.setNext(new PriorityNode(a, priority, null));  
2412             }  
2413             return;  
2414         }  
2415  
2416         // n elements  
2417  
2418         // First, check first element:  
2419         if (this.first.getPriority() > priority) {  
2420             this.first = new PriorityNode(a, priority, this.first);  
2421             return;  
2422         }  
2423  
2424         PriorityNode backup = this.first;  
2425         PriorityNode candidate = this.first.getNext();  
2426         while (candidate.getNext() != null) {  
2427             if (candidate.getPriority() > priority) {  
2428                 backup.setNext(new PriorityNode(a, priority, candidate));  
2429                 return;  
2430             }  
2431             candidate = candidate.getNext();  
2432         }  
2433         if (candidate.getPriority() > priority) {  
2434             backup.setNext(new PriorityNode(a, priority, candidate));  
2435         } else {  
2436             candidate.setNext(new PriorityNode(a, priority, null));  
2437         }  
2438     }  
2439  
2440     @Override  
2441     public void remove() {  
2442         if (this.first == null) {  
2443             System.out.println("No se puede desacolar una cola vacía");  
2444             return;  
2445         }  
2446         this.first = this.first.getNext();  
2447     }  
2448  
2449     @Override  
2450     public boolean isEmpty() {  
2451         return this.first == null;  
2452     }  
2453 }
```

```

2454     @Override
2455     public int getFirst() {
2456         if (this.first == null) {
2457             System.out.println("No se puede obtener el primero una cola vacia");
2458             return -1;
2459         }
2460         return this.first.getValue();
2461     }
2462
2463     @Override
2464     public int getPriority() {
2465         if (this.first == null) {
2466             System.out.println("No se puede obtener la prioridad del primero una cola
2467                         vacia");
2468             return -1;
2469         }
2470         return this.first.getPriority();
2471     }

```

5.17. Cola con prioridad - Operaciones

Una cola con prioridad cuenta con un algoritmo de ordenamiento por inserción por defecto para sus prioridades. Entonces, si colocamos como prioridad elementos de otra estructura, esos elementos quedarán ordenados.

Entonces, por ejemplo, para ordenar los elementos de una pila podemos volcar sus elementos a una cola con prioridad y luego nuevamente a la pila:

```

2472     public static void sort(IStack stack) {
2473         IPriorityQueue priorityQueue = new PriorityQueue();
2474         while (!stack.isEmpty()) {
2475             priorityQueue.add(stack.getTop(), stack.getTop());
2476             stack.remove();
2477         }
2478         while (!priorityQueue.isEmpty()) {
2479             stack.add(priorityQueue.getPriority());
2480             priorityQueue.remove();
2481         }
2482     }

```

Es importante notar que lo que tomamos como importante siempre es la prioridad, y el valor en este caso lo ignoramos.

5.18. Comentarios

Realmente no es importante la forma en la que se ordena una cola con prioridad. La definición de ordenamiento queda en manos del programador. En el caso de usar un tipo de dato que no sea primitivo (una clase) entonces es importante que implemente la interfaz Comparable que viene en el JRE de Java. Esta estructura queda por fuera del contenido de la materia, y como investigación útil para el TPO.

Las estructuras vistas en este capítulo se consideran lineales porque permiten elementos repetidos y cada elemento tiene un orden. Esto es sencillo de ver para una lista, pero para casos como el de una Pila no lo es tanto. El tener un tope, y luego poder recuperar su anterior, nos indica un orden porque de alguna forma recuerda los elementos en cuanto a su forma de ser ingresado a la estructura.

5.19. Ejercicios de TDA Pila

5.19.1. Métodos

Ejercicio 5.19.1: Contador

Defina un método que permita contar los elementos que tiene una Pila.

Ejercicio 5.19.2: Acumulador

Defina un método que permita sumar los elementos que tiene una Pila de números.

Ejercicio 5.19.3: Promedio

Defina un método que permita calcular el promedio los elementos que tiene una Pila de números.

Ejercicio 5.19.4: Capicúa

Defina una función que indique si una pila es capicúa.

Ejercicio 5.19.5: Repetidos

Eliminar de una Pila P las repeticiones de elementos, dejando un representante de cada uno de los elementos presentes originalmente. Se deberá respetar el orden original de los elementos, y en el caso de los repetidos se conservará el primero que haya ingresado en P .

Ejercicio 5.19.6: Repetidos 2

Generar el conjunto de elementos que se repiten en una Pila. Puede intentar este mismo ejercicio usando la estructura Conjunto cuando se llegue al capítulo correspondiente.

Ejercicio 5.19.7: Mitades

Repartir una Pila P en dos mitades M_1 y M_2 de elementos consecutivos, respetando el orden. Asumir que la Pila P contiene un número par de elementos.

5.19.2. Implementación y complejidad

Ejercicio 5.19.8: Implementación

scribir al menos dos implementaciones distintas (basadas en arreglos) del TDA Pila. Comparar los costos de cada una de las operaciones.

5.19.3. Algoritmos

Ejercicio 5.19.1: Cantidad par

Desarrolla una función que indique si una pila tiene una cantidad par de elementos sin usar variables auxiliares.

† Generalizar la función anterior para saber si es divisible por un n arbitrario.

Ejercicio 5.19.2: Partes

Eliminar el o los elementos del medio de una Pila arbitraria.

† Dada una Pila arbitraria, invertir solo su tercio del medio.

Ejercicio 5.19.3: Paréntesis balanceados

Dada una String arbitraria que contiene paréntesis, utilizar una Pila para saber si los paréntesis se abren y cierran correctamente. Comparelo con no usar Pilas.

† Modifique la función anterior para que también valide corchetes y llaves.

5.20. Ejercicios de TDA Cola

5.20.1. Definición

Ejercicio 5.20.1: Operaciones y precondiciones

Definir el TDA Cola, listando las operaciones asociadas y establecer sus precondiciones.

5.20.2. Métodos

Ejercicio 5.20.2: Pasar elementos

Definir un método que permita pasar los elementos de una Cola a otra.

Ejercicio 5.20.3: Inverso 1

Invertir el contenido de una Cola usando una Pila auxiliar.

Ejercicio 5.20.4: Inverso 2

Invertir el contenido de una Cola sin usar una Pila auxiliar.

Ejercicio 5.20.5: Subcola

Determinar si el final de una Cola $C1$ coincide con una Cola $C2$.

Ejercicio 5.20.6: Capicúa

eterminar si una Cola es capicúa o no.

Ejercicio 5.20.7: Inversas

eterminar si la Cola C1 es la inversa de la Cola C2. Dos Colas serán inversas, si tienen los mismos elementos pero en orden inverso

Ejercicio 5.20.8: Repetidos

liminar de una Cola C las repeticiones de elementos, dejando un representante de cada uno de los elementos presentes originalmente. Se deberá respetar el orden original de los elementos, y en el caso de los repetidos se conservará el primero que haya ingresado en C.

Ejercicio 5.20.9: Repetidos 2

enerar el conjunto de elementos que se repiten en una Cola.

Ejercicio 5.20.10: Mitades

epartir una Cola C en dos mitades M1 y M2 de elementos consecutivos, respetando el orden. Asumir que la cantidad de elementos de C es par.

5.20.3. Implementación y complejidad

Ejercicio 5.20.11: Implementación

scribir al menos dos implementaciones distintas (basadas en arreglos) del TDA Cola. Comparar los costos de cada una de las operaciones..

5.21. Ejercicios de TDA Cola con prioridad

5.21.1. Definición

Ejercicio 5.21.1: Operaciones y precondiciones

efinir el TDA Cola con prioridades, listando las operaciones asociadas y establecer sus precondiciones.

5.21.2. Métodos

Ejercicio 5.21.2: Merge

efina un método que permita combinar dos colas con prioridades CP1 y CP2, generando una nueva cola con prioridades. Considerar que a igual prioridad, los elementos de la CP1 son más prioritarios que los de la CP2

Ejercicio 5.21.3: Igualdad

eterminar si dos Colas con prioridad son idénticas.

5.21.3. Implementación y complejidad

Ejercicio 5.21.4: Implementación

scribir al menos dos implementaciones distintas (basadas en arreglos) del TDA Cola con prioridad. Comparar los costos de cada una de las operaciones.

Capítulo 6

Complejidad

Supongamos que tenemos dos versiones de un mismo algoritmo. Por ejemplo, para saber si un número es par podemos crear las siguientes funciones:

```
2483 public static boolean multipleOf2v1(int number) {
2484     if(number < 0) {
2485         return multipleOf2v1(-number);
2486     }
2487     if(number == 0) {
2488         return true;
2489     }
2490     if(number == 1) {
2491         return false;
2492     }
2493     return multipleOf2v1(number - 2);
2494 }
2495
2496 public static boolean multipleOf2v2(int number) {
2497     return number % 2 == 0;
2498 }
```

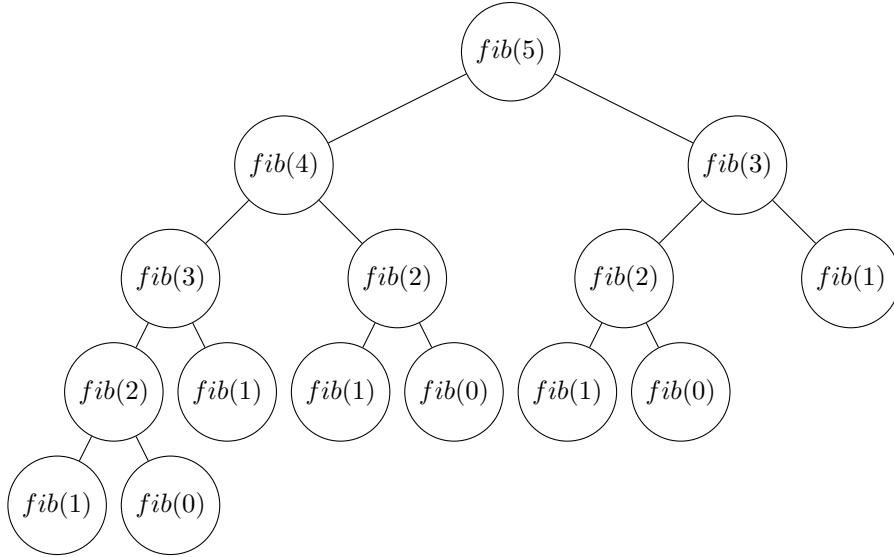
El primer caso tiene un código más largo pero más natural para quien no está familiarizado con el álgebra modular. El segundo caso tiene un código mucho más sencillo. Aún así, hay algunos otros aspectos además de la prolíjidad que pueden ser de interés, como por ejemplo el *costo espacial* y el *costo temporal*. El *análisis de costos* permite este análisis, donde comparamos para el costo espacial la relación entre la entrada del algoritmo y el uso de la memoria, y el costo temporal que indica cuántos pasos debe ejecutar nuestro algoritmo en base a la entrada (podemos pensarla también como cuánto vamos a usar nuestro procesador).

6.0.1. Costo espacial

El costo espacial hace referencia al uso de la memoria. Veamos lo que sucede con la sucesión Fibonacci:

```
2499 public static int fibonacci(int n) {
2500     if (n <= 1) {
2501         return n;
2502     }
2503     return fibonacci(n-1) + fibonacci(n-2);
2504 }
```

Cuando ejecutamos `fibonacci (n-1)`, esa iteración ejecutará `fibonacci (n-2)` y `fibonacci (n-3)`, aún ya habiendo calculado `fibonacci (n-2)`. Veamos un gráfico:



Es decir, hay cálculo repetido. La complejidad espacial aumenta ya que cada iteración instancia nuevas llamadas a la función en el stack de llamadas. La cantidad de llamadas generadas crece de forma exponencial a medida que aumenta el n al querer averiguar el n -ésimo término de la sucesión.

Entonces, un algoritmo como el siguiente, permitiría reducir este coste ya que no vamos a recalcular:

```

2505 public static int fibonacci(int n) {
2506     if (n <= 1) {
2507         return n;
2508     }
2509     int prev1 = 0;
2510     int prev2 = 1;
2511     for (int i = 2; i <= n; i++) {
2512         int current = prev1 + prev2;
2513         prev1 = prev2;
2514         prev2 = current;
2515     }
2516     return prev2;
2517 }
```

6.0.2. Costo temporal

Veamos la siguiente porción de código:

```

2518 public static void example() {
2519     System.out.println("Hello world");
2520 }
```

Esta función no tiene un parámetro que haga que su paso a paso varíe, por lo que siempre ejecutará las mismas instrucciones. Podemos, entonces, calcular cuantos milisegundos tarda en ejecutarse:

```

2521 public static void main(String[] args) {
2522     long init = System.nanoTime();
2523     example();
2524     long end = System.nanoTime();
2525     System.out.println("Difference: " + (end - init));
2526 }
```

La función `nanoTime` devuelve la cantidad de nanosegundos que pasaron de una fecha arbitraria (podría estar en el futuro o incluso moverse en distintas versiones) con una capacidad de 292 años de correcto

funcionamiento (dado que es de tipo `long` su resultado y en nanosegundos solo nos alcanza para este total de años). Entonces, la diferencia entre `end` e `init` nos dice cuánto tiempo en nanosegundos transcurrió entre ambas mediciones. Podemos usar también `currentTimeMillis`, pero esta utiliza un redondeo por lo que si la computadora es lo suficientemente rápido nos dirá que transcurrió 0 milisegundos. Como dato de color, esta fecha arbitraria mencionada suele ser en general, independientemente del lenguaje de programación, el 1 de enero de 1970. Esta fecha se conoce como *The Epoch*.

En general, esta función tardará en ejecutarse siguiendo una distribución normal, con *promedio* y *desvío estándar*. Para calcular ese promedio, ejecutamos n veces el algoritmo y sumamos los tiempos que tardó en cada ejecución, y luego, dividir ese total por n . Esta es la forma común de calcular un promedio. El código queda:

```

2527 public static void main(String[] args) {
2528     final int N = 10000;
2529     long total = 0;
2530     for(int i = 0; i < N; i++) {
2531         long init = System.nanoTime();
2532         example();
2533         long end = System.nanoTime();
2534         total += end - init;
2535     }
2536
2537     System.out.println("Average: " + (total / N));
2538 }
```

Podemos hacer esto más preciso haciendo que N tienda a ∞ . Ese promedio variará según la computadora donde se ejecute, por lo que definiremos como C (de constante, sin valor porque no lo conocemos).

El *coste constante* está presente en cualquier instrucción donde no se dependa del input de entrada, o donde el input de entrada siempre sea el mismo. Por ejemplo, el mensaje anterior Hello World nunca varió, dado que no estaba parametrizado.

Algunos casos donde podemos establecer un promedio son:

- Creación de una variable.
- Instanciación de un nuevo valor.
- Asignación de un valor a una variable.
- Comparaciones, operaciones matemáticas, lógicas o de bitwise.
- Imprimir un mensaje en pantalla no parametrizado.
- Lectura del valor de una variable.
- Llamada a un método (solo la llamada, la ejecución puede tener otro costo).

Sin embargo, nuestros programas no suelen tener una sola instrucción, sino que puede tener varias:

```

2539 public static void example2() {
2540     int a; // C_1
2541     a = 80; // C_2
2542     System.out.println(a + 20); // C_3
2543 }
```

En este caso, tenemos 3 constantes distintas para cada instrucción. Para conocer el costo del algoritmo total, tenemos que pensar en la definición: cuánto cuesta en tiempo. Entonces, sale a la vista que lo que tarda es la suma de lo que tarda cada una de sus instrucciones. Entonces, el costo de este algoritmo es $C_1 + C_2 + C_3$. Como esto es engorroso para escribir, usamos la propiedad matemática de que una constante sumada otra constante (multiplicada, dividida o restada también sucede, así como elevar una constante a una potencia o calcular su raíz, etc.) da otra constante. Entonces podemos definir $C_1 + C_2 + C_3 = C_4$ y

decir que nuestro algoritmo tarda C_4 . Yendo más profundo, vamos a quedarnos con que nuestro algoritmo tarda una constante que no conocemos, y dado que decir que tarda C puede confundir con otras constantes, necesitamos establecer una escritura que represente una familia de valores posibles. Vamos a representar cualquier costo constante con la notación Big-O.

6.0.3. Big- \mathcal{O}

La notación $\mathcal{O}(C)$ representa a cualquier costo que sea de tipo constante. No vamos a comparar algoritmos según lo que tardan en cada computadora, sino que vamos a comparar las familias de costos en las que estos algoritmos se clasifican. Por ejemplo, un algoritmo que imprime en pantalla Hello World comparado con otro que imprime en pantalla dos veces Hello World se clasifican como constantes, es decir, como $\mathcal{O}(C)$. Entonces, estos dos algoritmos se considerarán equivalentes.

Lo que colocamos dentro del símbolo \mathcal{O} es un elemento que represente a toda la familia de valores. Entonces, $\mathcal{O}(C)$ es equivalente a $\mathcal{O}(2C)$, dado que el número 2 es una constante, y multiplicar dos constantes da una nueva constante, por lo que puede representar a la familia de estos valores. Es por eso, que la complejidad $\mathcal{O}(C)$ puede aparecer escrita como $\mathcal{O}(1)$.

Podemos escribir dentro de \mathcal{O} cualquier función que se nos ocurra, siempre que su variable sea n , es decir, siempre escribiremos algo de la forma $\mathcal{O}(f(n))$. Este n es el tamaño del parámetro que hace variar el tiempo de ejecución de nuestra función. Por ejemplo, si calculamos el factorial de n , requeriremos multiplicar todos los números de 1 a n , por lo que el número en sí hace que se incremente la cantidad de pasos que se ejecutan del algoritmo. Sin embargo, si quiero calcular la suma de los elementos de un arreglo, el tamaño del arreglo será el n que hace variar la cantidad de iteraciones. Las complejidades más comunes que analizaremos serán:

- Constante: $\mathcal{O}(C)$.
- Logarítmico: $\mathcal{O}(\log(n))$.
- Lineal: $\mathcal{O}(n)$.
- Lineal-logarítmico: $\mathcal{O}(n\log(n))$.
- Cuadrático: $\mathcal{O}(n^2)$.
- Polinómico: $\mathcal{O}(n^a)$.
- Exponencial: $\mathcal{O}(a^n)$.
- Factorial: $\mathcal{O}(n!)$.

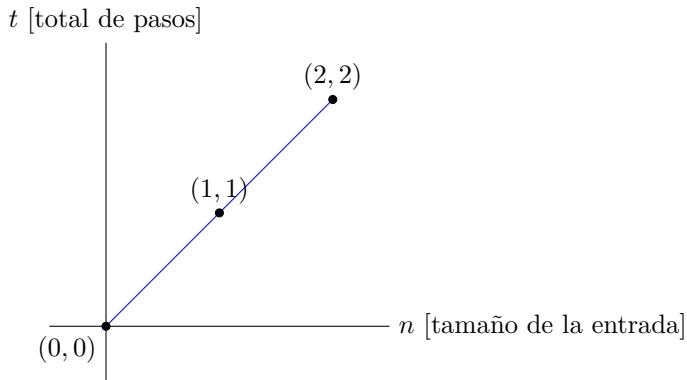
6.0.4. Costo lineal

La forma que tenemos de variar la cantidad de pasos de un algoritmo es mediante ciclos. Si queremos que el parámetro de nuestro algoritmo haga variar linealmente nuestro algoritmo, debemos al menos tener un ciclo que dependa de este valor. Este ciclo puede ser iterativo o por recursividad. Por el alcance de la materia, el análisis de costos en recursividades queda pendiente para verlo en Programación III.

Por ejemplo, vamos a calcular el costo de una función que recorre un arreglo e imprime sus valores en pantalla:

```
2544 public static void printArray(int[] array) {
2545     for(int i = 0; i < array.length; i++) {
2546         System.out.println(array[i]);
2547     }
2548 }
```

La impresión en pantalla se imprimirá una cantidad distinta de veces dependiendo del tamaño del arreglo. Para un arreglo de tamaño 0 se ejecutará 0, para un arreglo de tamaño 1 se ejecutará 1, etc. Es decir, si dibujamos en el plano estos puntos siendo el eje y la cantidad de instrucciones, y siendo x el tamaño del array, tenemos puntos del tipo (n, n) , es decir, tiene un comportamiento lineal:



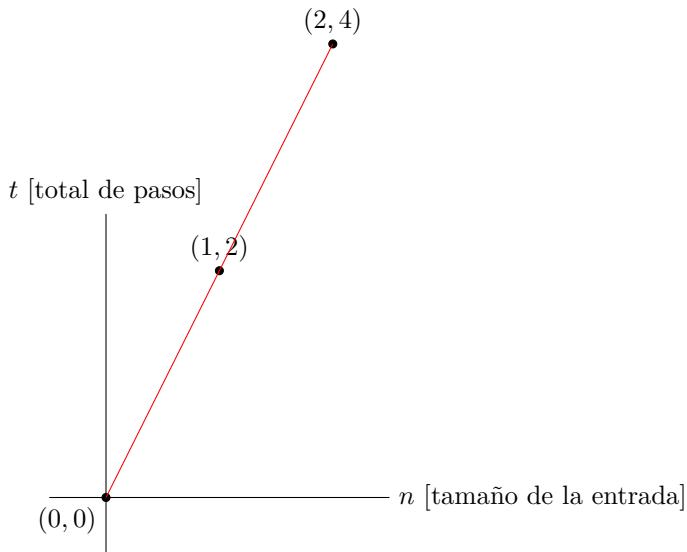
Es importante notar que los puntos sueltos en el plano son los que realmente interesan, dado que no podemos ejecutar un número decimal de veces una cantidad de pasos. La recta que se forma de unir los puntos es una aproximación. Si bien en este caso el gráfico se hace en base a la cantidad de pasos, algunas veces se hace en base al tiempo de ejecución y esto puede provocar que los puntos no estén totalmente alineados. Para corregir el gráfico en esos casos se usa regresión lineal.

El código anterior ejecuta una línea de código, o un paso, dentro del ciclo. ¿Qué sucede si tenemos 2 o más?

```

2549 public static void printArray(int[] array) {
2550     for(int i = 0; i < array.length; i++) {
2551         System.out.println(array[i]);
2552         System.out.println(array[i] + 1);
2553     }
2554 }
```

En este caso, para un arreglo de 0 ejecutará 0 instrucciones, pero para un arreglo de tamaño 1 ejecutará 2 instrucciones, y para un arreglo de tamaño 2 ejecutará 4 instrucciones, etc. Entonces, el gráfico quedará:



Es decir, sigue siendo lineal. Estamos frente a la costo lineal de un algoritmo. Ambos algoritmos se consideran dentro de la familia $\mathcal{O}(n)$.

Es importante hacer una observación: las líneas dentro del bloque de código que corresponde a nuestro ciclo, se ejecutarán tantas veces como el ciclo se ejecute. Entonces, la complejidad va a resultar de la complejidad del bloque multiplicado por la cantidad de veces que se ejecuta el ciclo. En el código anterior tenemos:

```

2555 public static void printArray(int[] array) {
2556     for(int i = 0; i < array.length; i++) { // N
2557         System.out.println(array[i]); // C_1
2558         System.out.println(array[i] + 1); // C_2
2559     }
2560 }
```

Entonces, la complejidad de este algoritmo es $\mathcal{O}(N * (C_1 + C_2))$, que cae dentro de la familia $\mathcal{O}(n)$.

¿Qué sucederá si tenemos un ciclo dentro de otro ciclo? Esta pregunta es razonable, dado que en estructuras de tipo matricial o estructuras en más dimensiones (por ejemplo las que se estudian para IA's o para manejo de grandes cantidades de datos) solemos tener muchos ciclos dentro de ciclos. Lo que sucede es que si los ciclos varían, hay que multiplicarlos:

```

2561 public static void printArray(int[][] array) {
2562     for(int i = 0; i < array.length; i++) { // N_1
2563         for(int j = 0; j < array[i].length; j++) { // N_2
2564             System.out.println(array[i][j]); // C
2565         }
2566     }
2567 }
```

La complejidad en este caso es $\mathcal{O}(N_1 * N_2 * C)$ que cae dentro de la familia $\mathcal{O}(n^2)$. Este es un caso de costo cuadrático, mientras que si tenemos tres ciclos será cúbico, y podemos generalizarlo como polinómico.

6.0.5. Jerarquía de los costos

Cuando tenemos varios tipos distintos de complejidades y necesitamos sumarlas, estas caen dentro de las complejidades de mayor jerarquía. Por ejemplo, $\mathcal{O}(n^2 + n^3 + c)$ es un caso particular de $\mathcal{O}(n^3)$. Esto se debe a que podemos pensarla como que el polinomio n^3 representa a cualquier polinomio de grado 3. Sin embargo, puede que haya complejidades que no podemos agrupar, por ejemplo, $\mathcal{O}(n + n * \log(n))$. Esto se dejaría así y no tiene un nombre en particular.

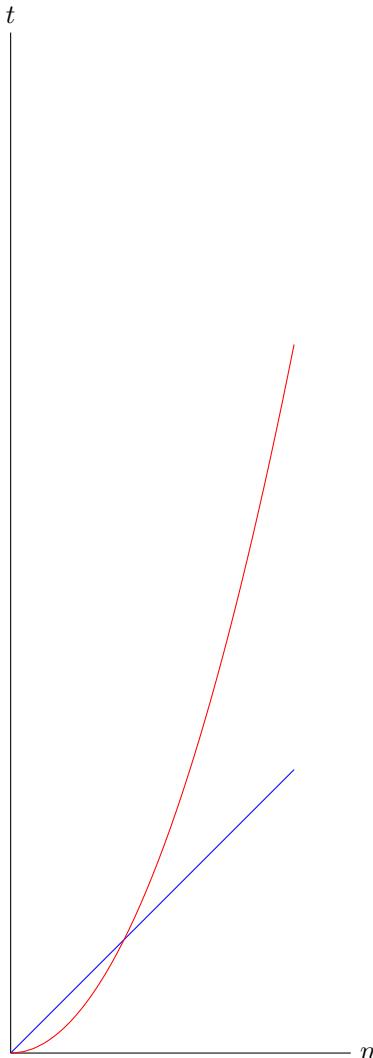
La jerarquía de complejidades se arma considerando el *peor caso*. Veamos el siguiente algoritmo:

```

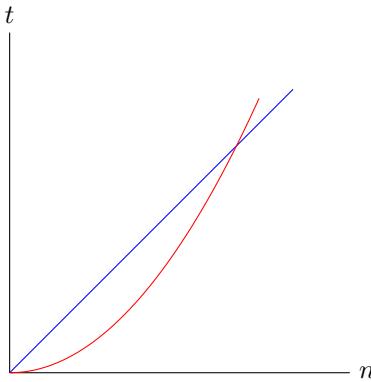
2568 public static int filter(int[] array) {
2569     for(int i = 0; i < array.length; i++) {
2570         if(array[i] == 5) {
2571             return i;
2572         }
2573     }
2574     return -1;
2575 }
```

Este algoritmo busca el número 5 dentro del arreglo y si lo encuentra devuelve su índice. Si el 5 se encuentra en la primera posición, su coste fue muy bajo, a lo sumo se ejecutó la creación de la variable `i`, su asignación, el acceso en el arreglo y la comparación para el primer elemento. Esto se conoce como el *mejor caso*. Luego, si hacemos un promedio de muchísimos arreglos, entonces seguramente si son lo suficientemente variados sea difícil que siempre se encuentre en la misma posición. En este caso estamos frente al *caso promedio*. En Programación II nuestro foco está en el peor caso. En este caso, el peor caso es que no se encuentre en el arreglo y que haya iterado por cada valor del arreglo.

Si graficamos una función cuadrática y una función lineal en el mismo gráfico, notaremos que la función cuadrática siempre superará a la lineal:



Es imposible que esta función cuadrática este invertida, o que la función lineal tenga pendiente negativa. El total de pasos que ejecutamos en un algoritmo siempre es un número positivo y siempre se está incrementando o manteniéndose. Entonces, una función cuadrática de la forma $f(x) = ax^2 + bx + c$ siempre superará a una lineal de la forma $g(x) = dx + f$. Podemos tratar de alejar la intersección estirando la parábola: solo tenemos que achicar el valor de a . Pero esto no evita la intersección y que la parábola en algún punto supere a la lineal. La única forma es que a tienda a cero. Entonces, vamos a considerar que tener un costo cuadrático es peor que tener un costo lineal. Podemos pensar el caso análogo de la comparación entre un coste constante contra un coste lineal, no importa que función constante elijamos, y que función lineal con pendiente positiva elijamos, siempre la función lineal superará a la función constante en algún punto.



La jerarquía entonces queda de la siguiente forma: $\mathcal{O}(C) < \mathcal{O}(\log(n)) < \mathcal{O}(n) < \mathcal{O}(n^a) < \mathcal{O}(a^n) < \mathcal{O}(n!)$. Veamos un ejemplo donde tenemos que sumar distintas complejidades:

```

2576 public static int filter(int[] array) {
2577     for(int i = 0; i < array.length; i++) { // N_1
2578         if(array[i] == 5) { // C_1
2579             return i; // C_2
2580         } else {
2581             for(int j = 0; j < i; j++) { // N_2
2582                 if(array[j] == 6) { // C_3
2583                     return i; // C_4
2584                 }
2585             }
2586         }
2587     }
2588     return -1; // C_5
2589 }
```

La complejidad queda como $\mathcal{O}(N_1 * (C_1 + N_2 * (C_3 + C_4)))$. En este caso, solo analizamos los costes de las líneas que se ejecutan en el peor caso. Terminamos con un costo de la familia $\mathcal{O}(n^2)$.

6.0.6. Coste logarítmico

Si tenemos una tanda de datos o un valor n que nos hace variar la cantidad de pasos, si la cantidad de potenciales pasos n se divide en un total de posibles m tramos, entonces la complejidad es $\log_m(n)$. Para bajarlo a tierra, podemos analizar el caso de búsqueda binaria. En el algoritmo de búsqueda binaria descartamos la mitad de los elementos luego de cada paso: para un potencial de pasos n , en la siguiente iteración de búsqueda binaria analizamos solo $\frac{n}{2}$. Luego, en el siguiente solo analizamos la mitad de esa mitad, es decir, $\frac{n}{4}$. Entonces, ¿cuántas veces podemos seguir dividiendo esta tanda de datos? Bueno, frenamos cuando el arreglo tiene tamaño 1. Entonces, ¿luego de cuantas divisiones logramos obtener esto? Tenemos que analizar esto: $\frac{n}{2^m} = 1$, y despejar m , que es el número de veces que dividimos por 2. Despejando, obtenemos la complejidad $\log_2(n)$, que cae dentro de la familia $\mathcal{O}(\log(n))$.

Para grandes tandas de datos, quizás sea más eficiente una variación de este algoritmo donde en lugar de dividir en 2 partes el arreglo, dividamos en p partes. En dicho caso, una vez escrito el algoritmo, podemos comparar para ver a partir de qué momento es más eficiente hacer esto. Para encontrar el punto de quiebre entre dos complejidades, solo tenemos que igualar las curvas que formamos a partir de sus complejidades.

6.0.7. Costo indeterminado

Existen problemas donde no podemos determinar el costo. Por ejemplo, la conjetura de Collatz es un caso donde no se conocen la cantidad de pasos para un número n para el cual se analice esta conjetura. Entonces, no tenemos una forma de calcular su complejidad computacional.

Otro ejemplo, es el de un ciclo infinito. No podemos definir un ciclo infinito como constante por tardar siempre indefinidamente, porque también podemos pensar lo como que esto es un peor caso que el exponencial.

6.0.8. Comentarios sobre costos temporales

A lo largo de esta materia solo nos enfocaremos en complejidades lineales, polinómicas, logarítmicas y constantes. Es importante ser sutil con estos conceptos. Por ejemplo:

```
2590 public static void printArray(int[] array) {
2591     for(int i = 0; i < 5; i++) {
2592         System.out.println(array[i]);
2593     }
2594 }
```

Este código no tiene complejidad lineal, porque no depende del tamaño del arreglo, sino que siempre ejecuta en el peor caso 5 veces la instrucción. ¿Podrían ser menos? Si, si se lanza un error por exceder el tamaño del arreglo, pero en nuestro caso siempre analizamos el peor caso.

6.0.9. Costos de las estructuras lineales

A continuación, un cuadro con los métodos y complejidades de los TDAs que vimos a partir de las implementaciones dadas. Es importante destacar que cada implementación requiere su análisis propio. Puede que existan mejores formas de desarrollarlo.

	Método	Dinámico
Lista enlazada	void add(int value)	$\mathcal{O}(n)$
	void insert(int index, int value)	$\mathcal{O}(n)$
	void remove(int index)	$\mathcal{O}(n)$
	int get(int index)	$\mathcal{O}(n)$
	int size()	$\mathcal{O}(C)$
	boolean isEmpty()	$\mathcal{O}(C)$

La lista enlazada es dinámica. Su análogo estático es el arreglo nativo. En ese caso, el acceso a cualquier posición es constante. Eliminar un elemento queda a la imaginación del programador, dado que al tener un tamaño fijo, eliminar un elemento puede significar reemplazarlo por un elemento arbitrario, entre otras opciones. No existe un borrado real en el caso de un arreglo nativo.

Pila	Método	Estático	Dinámico
	void add(int a)	$\mathcal{O}(C)$	$\mathcal{O}(C)$
	void remove()	$\mathcal{O}(C)$	$\mathcal{O}(C)$
	boolean isEmpty()	$\mathcal{O}(C)$	$\mathcal{O}(C)$
	int getTop()	$\mathcal{O}(C)$	$\mathcal{O}(C)$

Cola	Método	Estático	Dinámico
	void add(int a)	$\mathcal{O}(C)$	$\mathcal{O}(n)$
	void remove()	$\mathcal{O}(n)$	$\mathcal{O}(C)$
	boolean isEmpty()	$\mathcal{O}(C)$	$\mathcal{O}(C)$
	int getFirst()	$\mathcal{O}(C)$	$\mathcal{O}(C)$

Cola con prioridad	Método	Estático	Dinámico
	void add(int a, int priority)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	void remove()	$\mathcal{O}(n)$	$\mathcal{O}(C)$
	boolean isEmpty()	$\mathcal{O}(C)$	$\mathcal{O}(C)$
	int getFirst()	$\mathcal{O}(C)$	$\mathcal{O}(C)$
	int getPriority()	$\mathcal{O}(C)$	$\mathcal{O}(C)$

Tener en cuenta que en la implementación de cola con prioridad estática dada en este apunte utiliza búsqueda binaria, pero esto limita a que no acepte elementos repetidos. Su complejidad en ese caso es $\mathcal{O}(n + \log(n))$, que es *absorbida* en jerarquía por $\mathcal{O}(n)$.

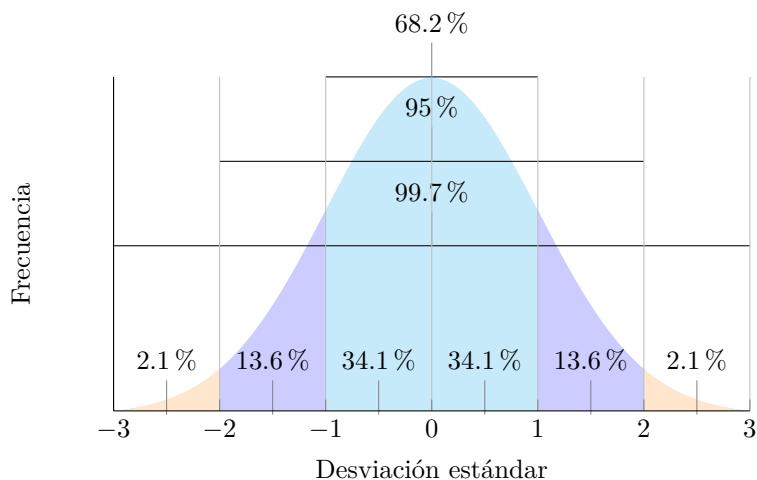
Capítulo 7

Conjuntos

7.1. Números aleatorios

Generar números aleatorios para una computadora es relativamente difícil, porque no existe un concepto bien definido para *aleatorio*. Entonces, existen distintos enfoques respecto a esto. Las *funciones pseudoaleatorias* son funciones que generan números lo suficientemente distintos como para no poder predecir el siguiente y dar ese aspecto de aleatoriedad a los números. Sin embargo, a largo plazo estas funciones pueden ser reconstruidas. Supongamos además lo siguiente: llamemos f a esta función y tomemos $f(1)$. Luego, tomemos $f(2)$, luego $f(3)$, etc. En algún punto, el número que recibe como parámetro tendrá una cota. Esta cota puede ser porque tipos de datos como `int` o `long` tienen un máximo, o bien por límites técnicos como el tamaño de la memoria de la computadora. Esto quiere decir que podemos quedarnos sin números al llegar al máximo. Pero este es el menor de los problemas, porque luego de sobrepasar el máximo volvemos al mínimo (es una propiedad de algunos lenguajes, como en Java), y entonces, si seguimos incrementando desde el mínimo en adelante, llegaremos nuevamente a $f(1)$. Es decir, recorrimos `1, 2, 3, ..., max long, min long, ..., -1, 0, 1, ...`. Solo hay que imaginar lo malo que sería para una simulación aleatoria, como la de un casino, tuviese este patrón tan a la vista. Por más que usemos distintas funciones pseudoaleatorias tenemos este problema.

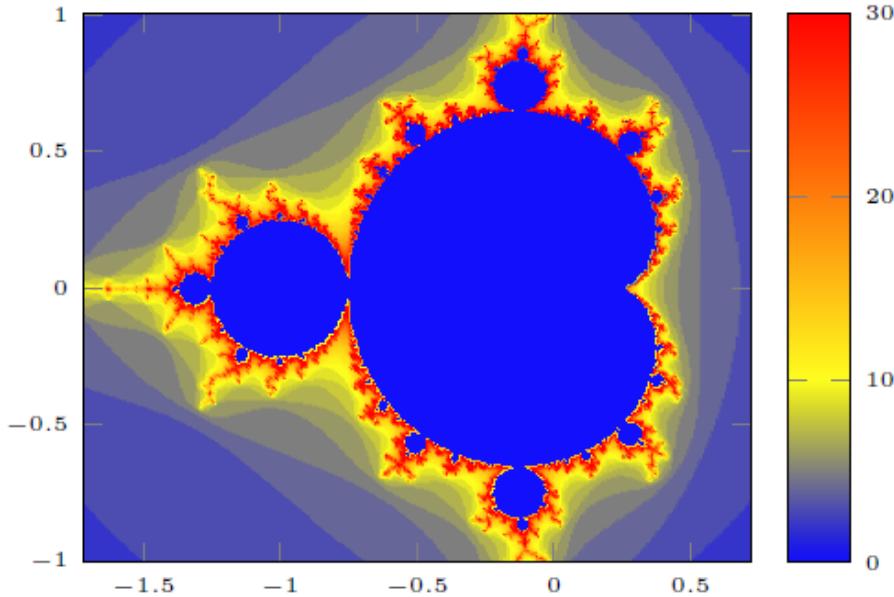
Por otro lado, tenemos datos de la naturaleza. Por ejemplo, podemos tomar una distribución normal de algún fenómeno fijo:



Al ser una distribución normal, mientras más al centro de la campana de Gauss tengamos valores, más probable serán de aparecer. Esto trae otro problema: podemos tomar datos aleatorios de fenómenos físicos, pero teniendo en contra que los valores pueden tomar distintas probabilidades entre sí. Esto no nos serviría por ejemplo para simular el lanzamiento de un dado, donde cada cara debe tener la misma probabilidad. Una

forma de solucionar esto es enfocarnos en valores muy cercanos a cero en el eje x de la distribución normal. Esto da el efecto del que tope de la gráfica se achata (algo así como nos sucede de que el planeta tierra visto tan de cerca como en nuestro día a día da el efecto de parecer plano). Sin embargo, esta opción no se puede considerar tan sencillamente porque para tomar datos de forma controlada necesitamos que cualquier computadora tenga acceso, y llegaremos a la conclusión de que necesitamos internet siempre para mantener estos datos comunicados.

No nos queda mucho más que usar funciones pseudoaleatorias. Existen muchas formas de hacerlo. Por ejemplo, usando la rama de las matemáticas conocida como *teoría del caos*, se puede usar funciones matemáticas relativamente sencillas para generar datos suficientemente complicados de predecir. Por ejemplo, usando la función $z_i = z^2 - 1$, podemos conseguir el fractal de Mandelbrot:



Podemos trazar alguna función que tome puntos de esta gráfica que no es tan sencilla de predecir. Existen entonces muchas técnicas, pero en general, necesitamos funciones que cumplan con la definición de *hash*.

7.1.1. Hash

Las funciones hash deben cumplir las siguientes propiedades:

- Distribución uniforme: La función debería dispersar los elementos de forma uniforme por todo el rango de posibles valores de hash. De esta manera, se minimiza la probabilidad de colisiones y se aprovecha al máximo el espacio de almacenamiento disponible. Con colisiones nos referimos a que los valores de salida de la función hash están en un rango, pero sus entradas pueden ser cualquiera, entonces pueden existir múltiples entradas que den el mismo resultado, y a esto lo llamamos colisión.
- Baja tasa de colisiones: Aunque es prácticamente imposible evitar completamente las colisiones, una buena función hash debería minimizar la probabilidad de que dos elementos distintos tengan el mismo valor de hash.
- Baja sensibilidad a patrones de entrada: Una función hash ideal debería producir valores de hash aleatorios y sin patrones predecibles a partir de los datos de entrada. De lo contrario, un atacante podría aprovechar patrones de entrada predecibles para aumentar la probabilidad de colisiones. En este caso, es normal hablar de atacantes porque hoy en día las funciones hash se usan mucho en seguridad informática.
- Eficiencia: La función hash debe ser rápida de calcular y no introducir demasiado overhead en las operaciones de inserción y búsqueda.

- Invertibilidad: Si se necesita recuperar el valor original a partir del hash (por ejemplo, para hacer una búsqueda exacta), la función hash debería permitir esta operación de manera poco eficiente. Esto es lo que se conoce como *función de ida*, dado que es fácil calcular y en $f(x) = y$ a partir de x , pero conociendo y es difícil saber quién fue el x que lo generó.
- Tamaño de salida adecuado (no es de tanta importancia para nuestro caso): La función hash debe producir un número de bits adecuado para el tamaño del espacio de almacenamiento y el número de elementos que se espera almacenar. Si el espacio de almacenamiento es demasiado pequeño o el número de elementos demasiado grande, puede aumentar la probabilidad de colisiones. Si el espacio de almacenamiento es demasiado grande o el número de elementos demasiado pequeño, se puede desperdiciar espacio.

Un caso sencillo de función hash es la que nos da el álgebra modular. Aplicar una fórmula con la estructura $(n + a) \% m$, nos permite clasificar en m valores posibles de resultado.

7.1.2. Semilla

Ya teniendo funciones hash, y técnicas para generar funciones pseudoaleatorias, planteamos un nuevo problema. Si corremos nuestro programa para una función pseudoaleatoria f , supongamos que tenemos una variable c que funciona como contador para saber el último valor aleatorio generado con $f(c)$. Entonces, al ir incrementando este valor a partir de un valor inicial (por ejemplo $c = 0$), iremos simulando valores de forma pseudoaleatoria. Supongamos que genera la sucesión 1, 1, 2, 3, 5, 8, si nuestro programa frena y vuelve a arrancar, nos topamos con que generará la misma sucesión de números. Esto rompe con la noción de aleatorio, porque conociendo el resultado de ejecutar nuestro programa una primera vez, las siguientes veces podremos predecirlo.

Una primera idea, es el concepto de semilla. La semilla parte de lo siguiente, en lugar de tener una función $f(x)$, tomamos una función $f(x, y)$, solo que este segundo parámetro será considerado nuestra semilla que va modificándose con el tiempo. Podemos, por ejemplo, usar un archivo para ir almacenando un contador especial que funcione como semilla, persistiendo en el archivo el último valor utilizado para no repetir la semilla la próxima vez.

Esto no es una solución definitiva. No siempre tenemos acceso a creación de archivos, también tenemos un límite, y si tenemos dos computadoras diferentes que corren el mismo programa irán generando las mismas sucesiones de números. La función `System.currentTimeMillis` vista en la sección anterior, funciona como una semilla perfecta, dado que no depende de la computadora, sino del tiempo. Esta es la solución más acorde usada hoy en día.

7.1.3. Función random

La función `Math.random()` nos da un número aleatorio entre 0 y 1 de tipo `double`. Más formalmente nos da un valor aleatorio en el rango $[0, 1]$, dado que el 1 no se incluye. Podemos distorsionar este rango para obtener un rango $[n, m]$ arbitrario.

Si multiplicamos por un número a lo anterior, tenemos un rango nuevo que es $[0, a]$. Es decir, multiplicar por un número el rango lo estira, alargándolo o achántolo dependiendo de que elegimos. De hecho, si $a = 0$ el rango es $[0, 0]$ que no existe, pero sin embargo multiplicar cualquier número por cero nos da cero. Obviando este caso, siempre se cumplirá que el rango tiene la forma $[0, a]$.

Ahora bien, si al número que obtuvimos le sumamos un valor b , entonces el rango se desplaza. Por ejemplo, si tenemos números en el rango del 1 al 10 y les sumamos 20, entonces tenemos como resultado números en el rango de 21 a 30. Es el rango original *desplazado b unidades*.

Con todo esto, estamos preparados para generar una función en java que genera números en un rango arbitrario:

```
2595 public static double random(int min, int max) {
2596     return Math.random() * (max - min) + min;
2597 }
```

Esto nos da valores para el rango $[min, max]$.

7.1.4. Función next

Otra forma de generar números aleatorios en Java es usando la función `random.nextDouble()` de la clase `Random`. A diferencia de lo anterior, esto nos da un `double` cualquiera de todos los valores posibles para números flotantes en Java. Entonces, podemos hacer lo siguiente:

```
2598 public static double random(int min, int max) {
2599     Random random = new Random();
2600     double number = random.nextDouble() / Double.MAX_VALUE;
2601     if(number < 0) {
2602         number = -number;
2603     }
2604     return number*(max-min) + min;
2605 }
```

En este caso, al dividir por el número más grande posible (aunque negado), nuestro rango se achica a $(-1, 1)$. El motivo de esto es que esta función nunca da como resultado el máximo posible, entonces nunca puede dar 1 la división. De la misma forma sucede con el mínimo. El único problema con esto es que al admitir negativos, todos los números tienen dos veces más de chances de salir que el 0.

Una forma de corregir esto es usar:

```
2606 public static double random(int min, int max) {
2607     return (new Random()).nextDouble(max - min) + min;
2608 }
```

Es decir, contamos con una forma de indicar un máximo a la hora de generarlos.

La clase `Random` es importante, dado que también podemos generar números enteros aleatorios.

7.1.5. Comparación de generaciones aleatorias

`Math.random()` parece menos costoso dado que no debemos crear instancias de ningún tipo. Además utiliza apropiadamente el concepto de semilla. El problema es que genera números aleatorios usando como semilla el valor de `System.currentTimeMillis` (o la de nanosegundos en versiones nuevas de Java), pero no recuerda la serie anterior, es decir, siempre toma el primer número de la serie únicamente. Al hacer esto, si generamos números aleatorios con esta función en el mismo milisegundo (o nanosegundo) entonces nos dará el mismo resultado. Esto quiere decir que tenemos una limitación técnica, donde no podemos generar más de 1000 números aleatorios por segundo. En cambio, esto no sucede con `Random`.

Con esto sobre la mesa, un criterio posible de cuando usar cada uno, es la necesidad de generar un total de números aleatorios. Si el total es muy grande, conviene usar `Random`, de lo contrario `Math`.

7.1.6. TDA Conjunto

Un conjunto es un tipo de *colección* de datos, no lineal, sin elementos repetidos, y que dependiendo de la implementación puede o no tener un orden. Dado que vamos a estudiar conjuntos de forma general, vamos a forzar la no dependencia del orden de los elementos, haciendo que la lectura de datos de un conjunto sea totalmente aleatoria.

Los conjuntos se suelen representar con letras mayúsculas A, B, C, \dots etc. Se representan como una lista de valores envueltas por llaves y cada elemento separado por coma, como por ejemplo el conjunto de los números enteros del 1 al 5 se escribe $1, 2, 3, 4, 5$. Se pueden representar por *extensión*, lo cual implica listar cada uno de sus elementos, o bien, por *comprensión* describiéndolo, por ejemplo: $n \in \mathbf{N} : 1 \leq n \leq 5$ (o simplemente con nuestras palabras).

Algunos conjuntos famosos son los siguientes:

Nombre	Símbolo
Conjunto de números naturales	\mathbb{N}
Conjunto de números enteros	\mathbb{Z}
Conjunto de números racionales	\mathbb{Q}
Conjunto de números reales	\mathbb{R}
Conjunto de números reales mayores a 0	\mathbb{R}^+
Conjunto de números pares	\mathbb{P}
Conjunto de números impares	\mathbb{I}

A un conjunto vamos a poder agregar un elemento o eliminarlo. Además, dado un conjunto, vamos a poder leer un elemento que tenga almacenado, aunque esta lectura no la elegimos, sino que queda en manos de la implementación. En este caso, el elemento obtenido será aleatorio, por lo que podemos imaginar a un conjunto como un bolillero donde no sabemos qué es lo que vamos a poder tomar.

Por otro lado, no podemos obtener un elemento de un conjunto sin elementos, por lo que es importante definir al *conjunto vacío*, el cual se escribe con el símbolo \emptyset , o simplemente como . Además, al igual que sucedía con pilas y colas, necesitamos saber si un conjunto se encuentra vacío para poder iterarlo, por lo que una condición de corte dependería de saber si el conjunto es vacío o no. Esto hace que el conjunto sea una estructura destructiva.

El TDA conjunto queda definido de la siguiente forma:

```

2609 /**
2610 * Precondicion: para usar cualquier de estos metodos la estructura debe estar
2611 * inicializada.
2612 */
2613 public interface ISet {
2614
2615     /**
2616     * Postcondicion: Agrega un valor al conjunto.
2617     *
2618     * @param a valor a agregar.
2619     */
2620     void add(int a);
2621
2622     /**
2623     * Postcondicion: Quita el elemento indicado si existe, de lo contrario no hace
2624     * nada.
2625     *
2626     * @param a valor a quitar.
2627     */
2628     void remove(int a);
2629
2630     /**
2631     * @return <code>true</code> si es el conjunto vacio, <code>false</code> en otro
2632     * caso.
2633     */
2634     boolean isEmpty();
2635
2636     /**
2637     * @return Devuelve el un elemento del conjunto.
2638     */
2639     int choose();
2640 }
```

7.1.7. Implementación estática

Para la implementación estática usamos arreglos, de forma análoga a lo que sucede con Pilas y Colas.

```
2639 public class Set implements ISet {  
2640  
2641     private final int[] array;  
2642     private int count;  
2643  
2644     public Set() {  
2645         this.array = new int[10000];  
2646         this.count = 0;  
2647     }  
2648  
2649     @Override  
2650     public void add(int a) {  
2651         for (int i = 0; i < this.count; i++) {  
2652             if (this.array[i] == a) {  
2653                 return;  
2654             }  
2655         }  
2656  
2657         this.array[this.count] = a;  
2658         this.count++;  
2659     }  
2660  
2661     @Override  
2662     public void remove(int a) {  
2663         for (int i = 0; i < this.count; i++) {  
2664             if (this.array[i] == a) {  
2665                 if (i == this.count - 1) {  
2666                     this.count--;  
2667                     return;  
2668                 }  
2669                 this.array[i] = this.array[this.count - 1];  
2670                 this.count--;  
2671             }  
2672         }  
2673     }  
2674  
2675     @Override  
2676     public boolean isEmpty() {  
2677         return this.count == 0;  
2678     }  
2679  
2680     @Override  
2681     public int choose() {  
2682         if (this.count == 0) {  
2683             System.out.println("No se puede elegir un elemento del conjunto vacío");  
2684             return -1;  
2685         }  
2686         int randomIndex = (new Random()).nextInt(this.count);  
2687         return this.array[randomIndex];  
2688     }  
2689 }
```

7.1.8. Implementación dinámica

De la misma forma procedemos para la implementación dinámica. El Nodo que usaremos será idéntico al de Pila.

```
2690 public class Set implements ISet {  
2691  
2692     private Node first;  
2693     private int count;  
2694  
2695     @Override  
2696     public void add(int a) {  
2697         if (this.first == null) {  
2698             this.first = new Node(a, null);  
2699             this.count++;  
2700             return;  
2701         }  
2702  
2703         if (this.first.getValue() == a) {  
2704             return;  
2705         }  
2706  
2707         Node candidate = this.first;  
2708         while (candidate.getNext() != null) {  
2709             candidate = candidate.getNext();  
2710             if (candidate.getValue() == a) {  
2711                 return;  
2712             }  
2713         }  
2714         candidate.setNext(new Node(a, null));  
2715         this.count++;  
2716     }  
2717  
2718     @Override  
2719     public void remove(int a) {  
2720         if (this.first == null || (this.first.getNext() == null && this.first.getValue()  
2721             () != a)) {  
2722             return;  
2723         }  
2724  
2725         if (this.first != null && this.first.getNext() == null) {  
2726             if (this.first.getValue() == a) {  
2727                 this.first = null;  
2728                 this.count--;  
2729             }  
2730             return;  
2731         }  
2732  
2733         if (this.first.getValue() == a) {  
2734             this.first = this.first.getNext();  
2735             this.count--;  
2736             return;  
2737         }  
2738  
2739         Node backup = this.first;  
2740         Node candidate = this.first.getNext();  
2741  
2742         while (candidate != null) {  
2743             if (candidate.getValue() == a) {  
2744                 backup.setNext(candidate.getNext());  
2745                 this.count--;  
2746                 return;  
2747             }  
2748         }  
2749         backup = candidate;
```

```

2748         candidate = candidate.getNext();
2749     }
2750 }
2751
2752 @Override
2753 public boolean isEmpty() {
2754     return this.count == 0;
2755 }
2756
2757 @Override
2758 public int choose() {
2759     if (this.count == 0) {
2760         System.out.println("No se puede elegir un elemento del conjunto vacío");
2761         return -1;
2762     }
2763     int randomIndex = (new Random()).nextInt(this.count);
2764     Node candidate = this.first;
2765     for (int i = 1; i <= randomIndex; i++) {
2766         candidate = candidate.getNext();
2767     }
2768     return candidate.getValue();
2769 }
2770
2771 }
```

7.1.9. Algoritmos útiles

Iterar un conjunto

```

2772 public static void print(ISet set) {
2773     while (!set.isEmpty()) {
2774         int element = set.choose();
2775         System.out.println(element);
2776         set.remove(element);
2777     }
2778 }
```

Operador In

El operador \in se usa para saber si un elemento está dentro de un conjunto. Más precisamente, es una función que recibe un conjunto, un elemento a verificar si está en el conjunto, y devuelve `true` si el elemento existe en el conjunto. Ejemplo $1 \in \{1, 2, 3\}$ pero $4 \notin \{1, 2, 3\}$. No hay que confundir con la operación que chequea entre conjuntos: $1 \notin \{1, 2, 3\}$, dado que este conjunto tiene los números 1, 2 y 3, pero no tiene conjuntos como elementos.

```

2779 public static boolean in(ISet a, int element) {
2780     ISet aux = new Set();
2781     boolean exists = false;
2782     while(!a.isEmpty()) {
2783         int value = a.choose();
2784         if(element == value) {
2785             exists = true;
2786             break;
2787         }
2788         aux.add(value);
2789         a.remove(value);
2790     }
```

```

2791     while(!aux.isEmpty()) {
2792         int value = aux.choose();
2793         a.add(value);
2794         aux.remove(value);
2795     }
2796     return exists;
2797 }
```

Igualdad entre conjuntos

Dos conjuntos A y B cumplen $A = B$, es decir, se consideran iguales, si tienen los mismos elementos. Dado que los conjuntos no poseen orden, no importa como aparecen los elementos al leer los conjuntos, solo que existan.

```

2798 public static boolean equals(ISet a, ISet b) {
2799     ISet aux = new Set();
2800     boolean equals = true;
2801     while(!a.isEmpty()) {
2802         int element = a.choose();
2803         if(!in(b, element)) {
2804             equals = false;
2805             break;
2806         }
2807         aux.add(element);
2808         a.remove(element);
2809         b.remove(element);
2810     }
2811     if(!b.isEmpty()) {
2812         equals = false;
2813     }
2814     while(!aux.isEmpty()) {
2815         int element = aux.choose();
2816         a.add(element);
2817         b.add(element);
2818         aux.remove(element);
2819     }
2820     return equals;
2821 }
```

Operador SubSet

El operador \subset define que si $A \subset B$ entonces A es un subconjunto propio de B . Es decir, cada elemento de A existe en B pero $A \neq B$.

```

2822 public static boolean subset(ISet a, ISet b) {
2823     ISet aux = new Set();
2824     boolean subset = true;
2825     while(!a.isEmpty()) {
2826         int element = a.choose();
2827         if(!in(b, element)) {
2828             subset = false;
2829             break;
2830         }
2831         aux.add(element);
2832         a.remove(element);
2833         b.remove(element);
2834     }
```

```

2835     if(b.isEmpty()) {
2836         subset = false;
2837     }
2838     while(!aux.isEmpty()) {
2839         int element = aux.choose();
2840         a.add(element);
2841         b.add(element);
2842         aux.remove(element);
2843     }
2844     return subset;
2845 }
```

Operador SubSetEq

El operador \subseteq define que si $A \subseteq B$ entonces A es un subconjunto de B . Es decir, cada elemento de A existe en B .

```

2846     public static boolean subseTEq(ISet a, ISet b) {
2847         return subset(a, b) || equals(a, b);
2848     }
```

Operador Union

La unión entre dos conjuntos A y B crea un conjunto C con todos los elementos de A y B . La representación de esto es $A \cup B = C$.

```

2849     public static ISet union(ISet a, ISet b) {
2850         ISet aux = new Set();
2851         ISet result = new Set();
2852         while(!a.isEmpty()) {
2853             int element = a.choose();
2854             aux.add(element);
2855             result.add(element);
2856             a.remove(element);
2857         }
2858         while(!aux.isEmpty()) {
2859             int element = aux.choose();
2860             a.add(element);
2861             aux.remove(element);
2862         }
2863         while(!b.isEmpty()) {
2864             int element = b.choose();
2865             aux.add(element);
2866             result.add(element);
2867             b.remove(element);
2868         }
2869         while(!aux.isEmpty()) {
2870             int element = aux.choose();
2871             b.add(element);
2872             aux.remove(element);
2873         }
2874         return result;
2875     }
```

Es importante notar que el elemento neutro de la unión es el conjunto E que cumple $A \cup E = A$. Esto implica que necesariamente $E = \emptyset$. Entonces, podemos usar esto para unir muchos conjuntos mediante un acumulador. Se recomienda releer el primer capítulo de este apunte para repasar este concepto.

```

2876 public static ISet unionAll(ISet[] sets) {
2877     ISet result = new Set(); // empty set
2878     for (ISet set : sets) {
2879         result = union(set, result);
2880     }
2881     return result;
2882 }
```

Operador Intersection

La unión entre dos conjuntos A y B crea un conjunto C con todos los elementos comunes de A y B . La representación de esto es $A \cap B = C$.

```

2883 public static ISet intersection(ISet a, ISet b) {
2884     ISet aux = new Set();
2885     ISet result = new Set();
2886     while(!a.isEmpty()) {
2887         int element = a.choose();
2888         aux.add(element);
2889         if(in(b, element)) {
2890             result.add(element);
2891         }
2892         a.remove(element);
2893     }
2894     while(!aux.isEmpty()) {
2895         int element = aux.choose();
2896         a.add(element);
2897         aux.remove(element);
2898     }
2899     return result;
2900 }
```

En este caso el elemento neutro de la intersección es el llamado *conjunto universal* representado como \mathbb{U} . Dado que no tenemos una representación, podemos tomar como valor inicial el primer conjunto del arreglo. Esto dará el mismo efecto, ya que si o si el conjunto universal intersección con cualquier conjunto, siempre será este otro conjunto.

```

2901 public static ISet intersectionAll(ISet[] sets) {
2902     if(sets == null || sets.length == 0) {
2903         return new Set();
2904     }
2905     ISet result = sets[0];
2906     for (int i = 1; i < sets.length; i++) {
2907         result = intersection(sets[i], result);
2908     }
2909     return result;
2910 }
```

Operador Difference

La diferencia $A - B$ crea un conjunto C tal que C tiene cada elemento de A que no está presente en B .

```

2911 public static ISet difference(ISet a, ISet b) {
2912     ISet aux = new Set();
2913     ISet result = new Set();
2914     while(!a.isEmpty()) {
2915         int element = a.choose();
```

```

2916     aux.add(element);
2917     if(!in(b, element)) {
2918         result.add(element);
2919     }
2920     a.remove(element);
2921 }
2922 while(!aux.isEmpty()) {
2923     int element = aux.choose();
2924     a.add(element);
2925     aux.remove(element);
2926 }
2927 return result;
2928 }
```

Operador Complement

Dado un contexto, por ejemplo, los números enteros, el complemento de un conjunto A de enteros es el conjunto \bar{A} que tiene todos los elementos del contexto que no están en A . Por ineficiencia, esto no es tratado, aunque si el conjunto que da el contexto (el conjunto universal) es pequeño, entonces podemos calcular \bar{A} como $\mathbb{U} - A$.

Operador Symmetric Difference

La diferencia simétrica $A \Delta B = C$, es tal que C contiene todos los elementos de A y B que no son comunes a ambos.

```

2929 public static ISet symmetricDifferenceV1(ISet a, ISet b) {
2930     return difference(union(a, b), intersection(a, b));
2931 }
2932
2933 public static ISet symmetricDifferenceV2(ISet a, ISet b) {
2934     return union(difference(a, b), difference(b, a));
2935 }
```

7.1.10. Cardinalidad

La cardinalidad $\#A$ devuelve el total de elementos del conjunto A .

```

2936 public static int cardinality(ISet a) {
2937     ISet aux = new Set();
2938     int count = 0;
2939     while(!a.isEmpty()) {
2940         int element = a.choose();
2941         aux.add(element);
2942         a.remove(element);
2943         count++;
2944     }
2945     return count;
2946 }
```

7.1.11. Conjunto de partes

Dado un conjunto A , el conjunto de partes de A se representa como $\mathcal{P}(A)$, el cual es el conjunto que contiene todos los posibles subconjuntos de A , incluido el conjunto vacío \emptyset y el conjunto A en sí mismo.

Consideremos el conjunto $A = \{1, 2, 3\}$. El conjunto de partes de A , $\mathcal{P}(A)$, se define como sigue:

$$\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

La cardinalidad del conjunto de partes $\mathcal{P}(A)$ está relacionada con la cardinalidad del conjunto A de la siguiente manera: si el conjunto A tiene n elementos, entonces la cardinalidad de $\mathcal{P}(A)$ es 2^n . Esto se debe a que cada elemento de A tiene dos opciones: estar presente o no estar (pensemos en bits y la cantidad de números en binario posibles de formar con n dígitos) presente en un subconjunto de $\mathcal{P}(A)$.

La definición formal es:

$$\#\mathcal{P}(A) = 2^{\#A}$$

En nuestro ejemplo, $A = \{1, 2, 3\}$ tiene 3 elementos. Por lo tanto, la cardinalidad de $\mathcal{P}(A)$ es $2^3 = 8$.

Vamos a crear un arreglo con todos los conjuntos de partes de un conjunto. Si intentamos hacerlo primero para un conjunto de cardinalidad 2, luego para uno de cardinalidad 3, etc, veremos que no podemos generalizar el código porque tenemos una cantidad de ciclos anidados variables, por lo que es un poco más difícil de programar que de costumbre. Esto viene de que la complejidad de este algoritmo es exponencial. Entonces, tenemos que recurrir a un concepto más simple. Las posibilidades de un elemento sobre un conjunto son 2: estar o no estar. Es decir, necesitamos 2 valores para representar esto, y eso es justamente lo que nos brinda el álgebra de Boole. Entonces, para un conjunto 1, 2, 3, podemos corresponder el array `[true, true, true]`, donde `true` es que el elemento está presente y `false` que no lo está. De esta forma, podemos representar el subconjunto 1, 2 como `[true, true, false]`. Esto es gracias a un isomorfismo entre el binario y las permutaciones, pero no es parte de esta materia su demostración. Entonces, podemos crear todas las permutaciones de booleanos de un arreglo de tamaño igual que la cardinalidad del conjunto, y en base a ese arreglo crear un subconjunto posible. Vamos a estar recorriendo todas las posibilidades de esta forma.

Entonces, como primer paso debemos crear todas las permutaciones posibles de un arreglo de booleanos de tamaño n . Si imaginamos que el arreglo `[true, true, false]` es el valor binario 110, podemos entonces pensar que estamos contando. Así como en el sistema decimal contamos 0, 1, 2, 3, 4, etc; en binario contamos como 0, 1, 10, 11, 100, etc. Debemos entonces pasar cada valor decimal a un valor en binario. Para esto podemos aplicar el algoritmo de conversión entre la base decimal a la base binaria:

```

2947 private static String decimalToBinary(int n, int digits) {
2948     String candidate = "";
2949     while(n != 0) {
2950         candidate += n % 2;
2951         n = n / 2;
2952     }
2953     while(candidate.length() < digits) {
2954         candidate += "0";
2955     }
2956     return (new StringBuilder(candidate)).reverse().toString();
2957 }
```

Luego, que en nuestra String haya un 1 es equivalente a `true` y que haya un 0 es equivalente a `false`. Ahora bien, ¿hasta cuánto debemos contar? Bueno, la cantidad de combinaciones para un arreglo de tamaño n es todas las combinaciones del subarray de 0 a $n - 1$ concatenado con un `true` y lo mismo concatenado con un `false`. Es decir, cada booleano duplica la cantidad de combinaciones. Entonces sin con 1 booleano tenemos 2 posibilidades, con n tenemos 2^n . Pero además, como empezamos a contar desde 0, entonces nuestro rango real de número a pasar a binario es de 0 a $2^n - 1$.

Con todo esto, podemos generar un arreglo con todos los conjuntos dentro del conjunto de partes de la siguiente forma:

```

2958 public static ISet[] parts(ISet set) {
2959     int cardinality = cardinality(set);
2960     int limit = (int) Math.pow(2, cardinality);
2961
2962     int[] values = new int[cardinality];
2963     int count = 0;
2964     ISet aux = new Set();
```

```

2965     while(!set.isEmpty()) {
2966         int element = set.choose();
2967         values[count] = element;
2968         aux.add(element);
2969         set.remove(element);
2970         count++;
2971     }
2972     while(!aux.isEmpty()) {
2973         int element = aux.choose();
2974         set.add(element);
2975         aux.remove(element);
2976     }
2977
2978     ISet[] result = new Set[limit];
2979     for(int i = 0; i < limit; i++) {
2980         String binary = decimalToBinary(i, cardinality);
2981         ISet part = new Set();
2982         for(int j = 0; j < binary.length(); j++) {
2983             if(binary.charAt(j) == '1') {
2984                 part.add(values[j]);
2985             }
2986         }
2987         result[i] = part;
2988     }
2989
2990     return result;
2991 }
```

7.1.12. Comparación con el álgebra de Boole

Si nos regresamos a los operadores AND (\wedge), OR (\vee) y NOT (\neg), notaremos ciertas similitudes con las propiedades de conjuntos. Se adjunta una tabla para poder compararlo.

Existe una correlación directa entre ambas áreas, y es que ambas, aún con diferentes contextos (el álgebra de Boole tiene solo 2 elementos y los conjuntos tienen una variedad de elementos, o por ejemplo la intersección de conjuntos nos da un valor que puede ser distinto a la entrada) las reglas parecerían ser las mismas. De hecho, esto es un *isomorfismo* entre ambas áreas de la matemática, por lo que podemos pasar sus propiedades de una a otra.

7.1.13. Comentarios

Los conjuntos son una estructura particular dado que existen muchas formas diferentes de implementar una misma definición, y no suele ser sencillo encontrar la alternativa de menor complejidad.

Para poner un ejemplo, la siguiente es una forma de definir si $A \subset B$:

- Si $\forall n \in A, n \in B$ y además $\#A \neq \#B$ entonces cumple.
- Si $A \cap B = A$ y además $\#A \neq \#B$ entonces cumple.
- Si $A \cup B = B$ y además $A \neq B$ entonces cumple.
- Si $A - B = \emptyset$ y además $B \not\subset A$ entonces cumple.
- Si $A \cap B = A$ y además $B - A \neq \emptyset$ entonces cumple.

Estos son solo algunos casos, pero existen muchas combinaciones más.

Propiedad	Álgebra de Boole	Conjuntos
Commutatividad	$p \vee q = q \vee p$ $p \wedge q = q \wedge p$	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Asociatividad	$(p \vee q) \vee r = p \vee (q \vee r)$ $(p \wedge q) \wedge r = p \wedge (q \wedge r)$	$(A \cup B) \cup C = A \cup (B \cup C)$ $(A \cap B) \cap C = A \cap (B \cap C)$
Distributividad	$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
Neutro	$p \vee \text{false} = p$ $p \wedge \text{true} = p$	$A \cup \emptyset = A$ $A \cap U = A$
Complemento	$p \vee \neg p = \text{true}$ $p \wedge \neg p = \text{false}$	$A \cup \overline{A} = U$ $A \cap \overline{A} = \emptyset$
Identidad	$p \vee p = p$ $p \wedge p = p$	$A \cup A = A$ $A \cap A = A$
Idempotencia	$p \vee \text{true} = \text{true}$ $p \wedge \text{false} = \text{false}$	$A \cup U = U$ $A \cap \emptyset = \emptyset$
Ley de De Morgan	$\neg(p \vee q) = \neg p \wedge \neg q$ $\neg(p \wedge q) = \neg p \vee \neg q$	$\overline{A \cup B} = \overline{A} \cap \overline{B}$ $\overline{A \cap B} = \overline{A} \cup \overline{B}$

Cuadro 7.1: Comparación de propiedades del álgebra de Boole y conjuntos

7.1.14. Análisis de costos

A continuación se adjuntan los costos para el TDA Conjunto:

Conjunto	Método	Estático	Dinámico
	void add(int a)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	void remove(int a)	$\mathcal{O}(n)$	$\mathcal{O}(C)$
	boolean isEmpty()	$\mathcal{O}(C)$	$\mathcal{O}(C)$
	int choose()	$\mathcal{O}(C)$	$\mathcal{O}(n)$

7.2. Ejercicios

Ejercicio 7.2.1: Conjunto modificado

Implementar el TDA Conjunto con las siguientes restricciones:

- Tamaño máximo acotado
- Tamaño máximo no acotado
- Universo acotado. Considerar por ejemplo el Universo de los números enteros entre 0 y N.

En todos los casos, dar al menos dos implementaciones utilizando arreglos y con listas dinámicas. Comparar los costos de las operaciones definidas en el TDA Conjunto según las implementaciones anteriores.

Ejercicio 7.2.2: Operaciones de conjuntos

Implementar métodos para calcular la unión, intersección y diferencia. Luego implementar funciones externas que permitan:

- Implementar la diferencia simétrica utilizando estos métodos.
- Implementar la diferencia simétrica sin utilizarlos.
- Implementar la igualdad entre conjuntos.
- Calcular la cardinalidad del conjunto.
- Generar el conjunto de elementos que están tanto en la Pila P y en la Cola C .
- Determinar si los elementos de una Pila P son los mismos que los de una Cola C . No interesa el orden ni si están repetidos o no.

Capítulo 8

Diccionarios

Un diccionario es una estructura de datos que funciona similar a un arreglo, pero sus índices no necesariamente son consecutivos o siquiera son números. Podemos imaginar un diccionario exactamente como un diccionario de la vida real, y lo representaremos mediante una tabla de 2 columnas, donde la primera es la *clave* y la segunda es el *valor* asociado a esa clave:

Clave	Valor
Nombre	Juan
Edad	25
País	España
Profesión	Ingeniero

Para comenzar con algo sencillo, vamos a considerar una tabla que tiene un tipo de dato `int` para la clave y un tipo de dato `int` para el valor. Vamos a decir que la clave y el valor forman un *par* o una *dúpla*.

Tenemos que poder agregar o eliminar un par, verificar si ya existe una clave y dada una clave poder conocer el valor asoaciado a esta clave. Operaciones como modificar un un par, siempre toman al valor como lo que se va a modificar, por lo que podemos considerar verificar si la clave existe, y si existe eliminarla y agregar el nuevo par (esto da como efecto el modificar, por lo que entonces no es una operación básica para este conjunto de operaciones). Además vamos a hacer un cambio útil, el cual es que, en lugar de considerar la operación que nos indica si una clave existe, vamos a crear una función que nos devuelve todas las claves existentes en un diccionario. De esta forma, podemos usar el `in` que vimos en el capítulo de Conjuntos, para poder chequear si el valor existe o no en el diccionario.

El TDA Diccionario nos queda de la siguiente forma:

```
2992 /**
2993 * Precondicion: para usar cualquier de estos metodos la estructura debe estar
2994 * inicializada.
2995 */
2996 public interface IDictionary {
2997     /**
2998     * Agrega un valor a una key, y de existir la reemplaza.
2999     *
3000     * @param key -
3001     * @param value -
3002     */
3003     void add(int key, int value);
3004
3005     /**
3006     * Para diccionarios simples se puede obviar el value.
3007     * Si una key que no existe, o un value que no existe esta asociado
3008     * a una key que si existe, entonces no se hace nada.
3009     *
3010     * @param key -
```

```

3011     * @param value -
3012     */
3013     void remove(int key, int value);
3014
3015     /**
3016     * @return conjunto con todas las claves del diccionario
3017     */
3018     ISet getKeys();
3019
3020     /**
3021     * Devuelve el valor asociado a una key.
3022     * Precondicion: No se puede obtener un valor de una key que no existe.
3023     *
3024     * @param key -
3025     * @return value asociado al key
3026     */
3027     int getValue(int key);
3028
3029     /**
3030     * @return <code>true</code> si el diccionario esta vacio, <code>false</code> en
3031     * otro caso.
3032     */
3033     boolean isEmpty();
3034 }
```

El método para saber si un diccionario es vacío no es realmente necesario, dado que podemos saberlo si al obtener sus claves obtenemos un conjunto vacío. Sin embargo, es de utilidad cuando queremos remover un elemento de un diccionario vacío, podemos chequear esto sin necesidad de pedir sus claves.

8.0.1. Diccionario Simple - Implementación estática

En la implementación estática no nos interesa si el diccionario utiliza un conjunto dinámico para obtener las claves. Lo importante es que tenga un tamaño fijo.

```

3034 public class Dictionary implements IDictionary {
3035
3036     private int[] keys;
3037     private int[] values;
3038     private int size;
3039
3040     public Dictionary() {
3041         this.keys = new int[10000];
3042         this.values = new int[10000];
3043         this.size = 0;
3044     }
3045
3046     @Override
3047     public void add(int key, int value) {
3048         int index = indexOfKey(key);
3049         if (index != -1) {
3050             this.values[index] = value; // Si la key ya existe, se reemplaza el value
3051             return;
3052         }
3053         if (this.size == this.keys.length) {
3054             // Si el array de keys esta lleno, se duplica su longitud
3055             this.keys = Arrays.copyOf(this.keys, this.keys.length * 2);
3056             this.values = Arrays.copyOf(this.values, this.values.length * 2);
3057         }
3058         this.keys[this.size] = key;
```

```

3059     this.values[this.size] = value;
3060     this.size++;
3061 }
3062
3063 @Override
3064 public void remove(int key, int value) {
3065     int index = indexOfKey(key);
3066     if (index != -1 && this.values[index] == value) {
3067         for (int i = index; i < this.size - 1; i++) {
3068             this.keys[i] = this.keys[i + 1];
3069             this.values[i] = this.values[i + 1];
3070         }
3071         this.size--;
3072     }
3073 }
3074
3075 @Override
3076 public ISet getKeys() {
3077     ISet keySet = new Set();
3078     for (int i = 0; i < this.size; i++) {
3079         keySet.add(this.keys[i]);
3080     }
3081     return keySet;
3082 }
3083
3084 @Override
3085 public int getValue(int key) {
3086     int index = indexOfKey(key);
3087     if (index != -1) {
3088         return this.values[index];
3089     }
3090     return -1; // Error
3091 }
3092
3093 @Override
3094 public boolean isEmpty() {
3095     return this.size == 0;
3096 }
3097
3098 private int indexOfKey(int key) {
3099     for (int i = 0; i < this.size; i++) {
3100         if (this.keys[i] == key) {
3101             return i;
3102         }
3103     }
3104     return -1;
3105 }
3106 }

```

8.0.2. Diccionario Simple - Implementación dinámica

Para poder realizar la implementación dinámica vamos a necesitar un nuevo tipo de nodo que nos permita almacenar nuestro par de datos:

```

3107 @Getter
3108 @Setter
3109 @AllArgsConstructor
3110 public class DictionaryNode {

```

```
3111
3112     private int key;
3113     private int value;
3114     private DictionaryNode next;
3115
3116 }
```

En este caso, solo para dar un aspecto más cercano al ambiente profesional, utilicé Lombok. Las annotations encima de la clase agregan todos los métodos necesarios en tiempo de compilación, por lo que los archivos java al ser convertidos en class, agregan todo lo que indiquen las annotations aunque en nuestro código no esté escrito.

Con nuestro nuevo Nodo ya podemos crear nuestra implementación.

```
3117 public class Dictionary implements IDictionary {
3118
3119     private DictionaryNode first;
3120     private int size;
3121
3122     public Dictionary() {
3123         size = 0;
3124     }
3125
3126     @Override
3127     public void add(int key, int value) {
3128         if (this.first == null) {
3129             this.first = new DictionaryNode(key, value, null);
3130             this.size++;
3131             return;
3132         }
3133         DictionaryNode index = indexOfKey(key);
3134         if (index != null) {
3135             index.setValue(value); // Si la key ya existe, se reemplaza el value
3136             return;
3137         }
3138         DictionaryNode lastNode = this.first;
3139         while (lastNode.getNext() != null) {
3140             lastNode = lastNode.getNext();
3141         }
3142         lastNode.setNext(new DictionaryNode(key, value, null));
3143         this.size++;
3144     }
3145
3146     @Override
3147     public void remove(int key, int value) {
3148         if (this.first == null) {
3149             return;
3150         }
3151         if (this.first.getKey() == key && this.first.getValue() == value) {
3152             this.first = this.first.getNext();
3153             this.size--;
3154             return;
3155         }
3156         DictionaryNode backup = null;
3157         DictionaryNode candidate = this.first;
3158         while (candidate.getNext() != null) {
3159             if (candidate.getKey() == key && candidate.getValue() == value) {
3160                 backup.setNext(candidate.getNext());
3161                 this.size--;
3162                 return;
3163             }
3164         }
3165     }
3166 }
```

```
3163     }
3164     backup = candidate;
3165     candidate = candidate.getNext();
3166 }
3167 if (candidate.getKey() == key && candidate.getValue() == value) {
3168     backup.setNext(null);
3169     this.size--;
3170 }
3171 }
3172
3173 @Override
3174 public ISet getKeys() {
3175     ISet keySet = new Set();
3176     DictionaryNode candidate = this.first;
3177     while (candidate != null) {
3178         keySet.add(candidate.getKey());
3179         candidate = candidate.getNext();
3180     }
3181     return keySet;
3182 }
3183
3184 @Override
3185 public int getValue(int key) {
3186     DictionaryNode candidate = this.first;
3187     while (candidate != null) {
3188         if (candidate.getKey() == key) {
3189             return candidate.getValue();
3190         }
3191         candidate = candidate.getNext();
3192     }
3193     return -1; // Error
3194 }
3195
3196
3197 @Override
3198 public boolean isEmpty() {
3199     return size == 0;
3200 }
3201
3202 private DictionaryNode indexOfKey(int key) {
3203     if (this.first == null) {
3204         return null;
3205     }
3206     DictionaryNode candidate = this.first;
3207     while (candidate != null) {
3208         if (candidate.getKey() == key) {
3209             return candidate;
3210         }
3211         candidate = candidate.getNext();
3212     }
3213     return null;
3214 }
3215 }
```

8.0.3. Diccionario Simple - Operaciones

Dado que podemos obtener un conjunto de claves de forma inmutable, podemos destruirlo al recorrerlo sin preocuparnos por hacer una copia. Entonces, podemos recorrer un diccionario de la siguiente forma:

```

3216 public static void print(IDictionary dictionary) {
3217     ISet keys = dictionary.getKeys();
3218     while(!keys.isEmpty()) {
3219         int key = keys.choose();
3220         System.out.printf("key: %d, value: %d", key, dictionary.getValue(key));
3221         keys.remove(key);
3222     }
3223 }
```

Otra utilidad de los diccionarios es poder contar la cantidad de apariciones de un elemento. En el siguiente ejemplo contamos la cantidad de veces que aparece cada elemento de un arreglo. El elemento será la clave y el total de veces que aparece será el valor:

```

3224 public static IDictionary counter(int[] array) {
3225     IDictionary dictionary = new Dictionary();
3226
3227     for(int i = 0; i < array.length; i++) {
3228         if(!in(dictionary.getKeys(), array[i])) {
3229             dictionary.add(array[i], 1);
3230         } else {
3231             int total = dictionary.getValue(array[i]);
3232             dictionary.remove(array[i], total);
3233             total++;
3234             dictionary.add(array[i], total);
3235         }
3236     }
3237
3238     return dictionary;
3239 }
```

El método `in` es el que creamos previamente en el capítulo de Conjuntos.

8.0.4. Diccionario Múltiple - Implementación estática

El Diccionario Múltiple es una variación del Diccionario Simple, donde cada valor tiene como tipo de dato una Lista. Esto se parece más a un diccionario de la vida real, donde para cada palabra tenemos varias definiciones. Para esto, tenemos que readaptar el TDA:

```

3240 /**
3241 * Precondicion: para usar cualquier de estos metodos la estructura debe estar
3242 * inicializada.
3243 */
3244 public interface IMultipleDictionary {
3245
3246     /**
3247     * Agrega un valor a una key, y de existir no hace nada.
3248     *
3249     * @param key -
3250     * @param value -
3251     */
3252     void add(int key, int value);
3253
3254     /**
3255     * Si una key que no existe, o un value que no existe esta asociado
3256     * a una key que si existe, entonces no se hace nada.
3257     * En el caso de que la key conserve valores asociados, seguira presente.
3258     * Si la key se queda sin valores asociados, entonces se elimina.
3259     */
3260 }
```

```

3259     * @param key -
3260     * @param value -
3261     */
3262     void remove(int key, int value);
3263
3264     /**
3265     * @return conjunto con todas las claves del diccionario
3266     */
3267     ISet getKeys();
3268
3269     /**
3270     * Devuelve los valores asociados a una key.
3271     * Precondicion: No se puede obtener un valor de una key que no existe.
3272     *
3273     * @param key -
3274     * @return values asociados al key
3275     */
3276     ISet getValues(int key);
3277
3278     /**
3279     * @return <code>true</code> si el diccionario esta vacio, <code>false</code> en
3280     * otro caso.
3281     */
3282     boolean isEmpty();
3283 }
```

Una posible implementación será:

```

3283 public class MultipleDictionary implements IMultipleDictionary {
3284     private int[] keys;
3285     private ISet[] values;
3286     private int size;
3287
3288     public MultipleDictionary() {
3289         this.keys = new int[10000];
3290         this.values = new Set[10000];
3291         this.size = 0;
3292     }
3293
3294     @Override
3295     public void add(int key, int value) {
3296         int index = indexOfKey(key);
3297         if (index != -1) {
3298             this.values[index].add(value);
3299             return;
3300         }
3301         if (this.size == this.keys.length) {
3302             // Si el array de keys esta lleno, se duplica su longitud
3303             this.keys = Arrays.copyOf(this.keys, this.keys.length * 2);
3304             this.values = Arrays.copyOf(this.values, this.values.length * 2);
3305         }
3306         this.keys[this.size] = key;
3307         this.values[this.size] = new Set();
3308         this.values[this.size].add(value);
3309         this.size++;
3310     }
3311
3312     @Override
3313     public void remove(int key, int value) {
```

```
3314     int index = indexOfKey(key);
3315     if (index != -1 && this.existsValue(this.values[index], value)) {
3316         this.values[index].remove(value);
3317         if (this.values[index].isEmpty()) {
3318             for (int i = index; i < this.size - 1; i++) {
3319                 this.keys[i] = this.keys[i + 1];
3320                 this.values[i] = this.values[i + 1];
3321             }
3322             this.size--;
3323         }
3324     }
3325 }
3326
3327 private boolean existsValue(ISet set, int value) {
3328     ISet copy = new Set();
3329     boolean exists = false;
3330     while (!set.isEmpty()) {
3331         int element = set.choose();
3332         if (element == value) {
3333             exists = true;
3334             break;
3335         }
3336         copy.add(element);
3337         set.remove(element);
3338     }
3339     while (!copy.isEmpty()) {
3340         int element = copy.choose();
3341         set.add(element);
3342         copy.remove(element);
3343     }
3344     return exists;
3345 }
3346
3347 @Override
3348 public ISet getKeys() {
3349     ISet keySet = new Set();
3350     for (int i = 0; i < this.size; i++) {
3351         keySet.add(this.keys[i]);
3352     }
3353     return keySet;
3354 }
3355
3356 @Override
3357 public ISet getValues(int key) {
3358     int index = indexOfKey(key);
3359     if (index != -1) {
3360         return this.values[index];
3361     }
3362     return null; // Error
3363 }
3364
3365 @Override
3366 public boolean isEmpty() {
3367     return this.size == 0;
3368 }
3369
3370 private int indexOfKey(int key) {
3371     for (int i = 0; i < this.size; i++) {
3372         if (this.keys[i] == key) {
```

```

3373         return i;
3374     }
3375   }
3376   return -1;
3377 }
3378 }
```

8.0.5. Diccionario Múltiple - Implementación dinámica

Para que una implementación dinámica sea posible, debemos modificar el tipo de dato de nuestro Nodo en el Diccionario Simple, por lo que tenemos:

```

3379 public class MultipleDictionaryNode {
3380
3381     private int key;
3382     private ISet value;
3383     private MultipleDictionaryNode next;
3384
3385     public MultipleDictionaryNode(int key, ISet value, MultipleDictionaryNode next) {
3386         this.key = key;
3387         this.value = value;
3388         this.next = next;
3389     }
3390
3391     public int getKey() {
3392         return key;
3393     }
3394
3395     public void setKey(int key) {
3396         this.key = key;
3397     }
3398
3399     public ISet getValue() {
3400         return value;
3401     }
3402
3403     public void setValue(ISet value) {
3404         this.value = value;
3405     }
3406
3407     public MultipleDictionaryNode getNext() {
3408         return next;
3409     }
3410
3411     public void setNext(MultipleDictionaryNode next) {
3412         this.next = next;
3413     }
3414 }
```

Una posible implementación será:

```

3415 public class MultipleDictionary implements IMultipleDictionary {
3416
3417     private MultipleDictionaryNode first;
3418
3419     @Override
3420     public void add(int key, int value) {
3421         ISet set = new Set();
```

```
3422     set.add(value);
3423     if (this.first == null) {
3424         this.first = new MultipleDictionaryNode(key, set, null);
3425         return;
3426     }
3427     MultipleDictionaryNode candidate = this.first;
3428     while (candidate.getNext() != null) {
3429         if (candidate.getKey() == key) {
3430             candidate.getValue().add(value);
3431             return;
3432         }
3433         candidate = candidate.getNext();
3434     }
3435     if (candidate.getKey() == key) {
3436         candidate.getValue().add(value);
3437         return;
3438     }
3439     candidate.setNext(new MultipleDictionaryNode(key, set, null));
3440 }
3441
3442 @Override
3443 public void remove(int key, int value) {
3444     MultipleDictionaryNode backup = null;
3445     MultipleDictionaryNode candidate = this.first;
3446     while (candidate != null) {
3447         if (candidate.getKey() == key) {
3448             candidate.getValue().remove(value);
3449             if (candidate.getValue().isEmpty()) {
3450                 if (backup == null) {
3451                     if (candidate.getNext() == null) {
3452                         this.first = null;
3453                         return;
3454                     }
3455                     this.first = this.first.getNext();
3456                     return;
3457                 }
3458                 if (candidate.getNext() == null) {
3459                     backup.setNext(null);
3460                     return;
3461                 }
3462                 candidate.setNext(candidate.getNext().getNext());
3463             }
3464             return;
3465         }
3466         backup = candidate;
3467         candidate = candidate.getNext();
3468     }
3469 }
3470
3471 @Override
3472 public ISet getKeys() { // N^2
3473     ISet keySet = new Set(); // C
3474     MultipleDictionaryNode candidate = this.first; // C
3475     while (candidate != null) { // N * (N * C) = N^2 C = N^2
3476         keySet.add(candidate.getKey()); // N*C
3477         candidate = candidate.getNext();
3478     }
3479     return keySet; // C
3480 }
```

```

3481
3482     @Override
3483     public ISet getValues(int key) {
3484         MultipleDictionaryNode candidate = this.first;
3485         while (candidate != null) {
3486             if (candidate.getKey() == key) {
3487                 return candidate.getValue();
3488             }
3489             candidate = candidate.getNext();
3490         }
3491         return null; // Error
3492     }
3493
3494     @Override
3495     public boolean isEmpty() {
3496         return this.first == null;
3497     }
3498 }
```

Es importante indicar, que esta implementación utiliza una técnica de *buckets*, la cual permite ahorrar espacio en nuestros arreglos, a costa de sacrificar el acceso constante a los datos, el cual ahora se vuelve lineal.

En general, en Java ya existe un tipo de Tabla Hash, conocida como `HashMap<?>`. Lo más importante de utilizarlas, es que reducimos el costo del acceso a un valor, de lineal a constante.

Análisis de costos

	Método	Estático	Dinámico
Diccionario Simple	void add(int key, int value)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	void remove(int key, int value)	$\mathcal{O}(n)$	$\mathcal{O}(C)$
	ISet getKeys()	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	int getValue(int key)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	boolean isEmpty()	$\mathcal{O}(C)$	$\mathcal{O}(C)$
Diccionario Múltiple	void add(int key, int value)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	void remove(int key, int value)	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	ISet getKeys()	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	ISet getValues(int key)	$\mathcal{O}(C)$	$\mathcal{O}(n)$
	boolean isEmpty()	$\mathcal{O}(C)$	$\mathcal{O}(n)$
Tabla Hash	Método	Estático	
	int size()	$\mathcal{O}(C)$	
	put(Integer key, Integer value)	$\mathcal{O}(n)$	
	Integer get(Integer key)	$\mathcal{O}(n)$	
	void remove(Integer key)	$\mathcal{O}(n)$	

Como comentario, la complejidad de leer un elemento de la tabla hash es lineal debido a la implementación que dimos, aunque la mayoría de las veces será constante.

8.1. Ejercicios

Ejercicio 8.1.1: Diccionario modificado

ar al menos dos implementaciones utilizando arreglos y con listas dinámicas, primero para diccionario simple, y luego para diccionario múltiple, dando las complejidades asociadas.

Ejercicio 8.1.2: Cola con prioridad

scribir un método externo que permita generar un Diccionario Múltiple que permita, para cada valor presente en la Cola con Prioridad C recuperar todas las prioridades que tiene asociadas en C.

Ejercicio 8.1.3: Mezcla de diccionarios

ados dos Diccionarios múltiples D1 y D2, generar un Diccionario múltiple que contenga:

- las claves presentes en D1 y D2, con todos los elementos asociados a cada clave.
- las claves presentes en D1 y D2, con todos los elementos comunes a las claves coincidentes en ambos.
- las claves comunes de D1 y D2, con todos los elementos asociados a cada clave.
- las claves comunes de D1 y D2, con todos los elementos comunes a las claves coincidentes en ambos.

Ejercicio 8.1.4: Sinónimos

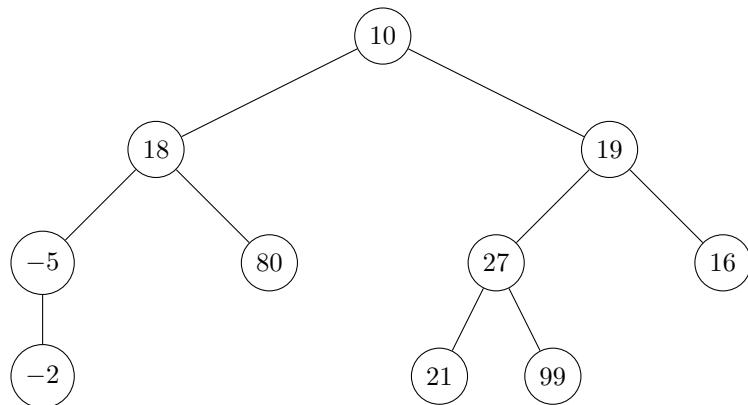
ado un Diccionario Simple D, que representa el concepto clásico de diccionario: la clave representa una palabra y el valor su significado. Generar un Diccionario Múltiple DS que a partir de un significado s, vincule todas las palabras que tienen dicho significado, es decir que son sinónimos. Cada clave s será un significado y los valores asociados (sinónimos) aquellas claves de D que tenían asociado el valor s.

Capítulo 9

Árbol binario

Los árboles son un tipo especial de grafos. En particular, no poseen ciclos y son conexos (esto se verá más a detalles en el capítulo de grafos).

En este capítulo vamos a comenzar con un tipo especial de árbol conocido como árbol binario. Un ejemplo de estos árboles podrían ser:



En el gráfico anterior, los círculos que tienen un númerito dentro, son conocidos como *nodos* o *vértices*. Un nodo cualquiera puede tener una *arista* que lo une con otro nodo. La forma de leerlo es de arriba hacia abajo, donde un nodo tiene como máximo dos aristas. El nodo de la parte superior es un *nodo padre*, y los nodos que se desprenden en sus aristas son *nodos hijo*. Si bien nosotros vamos a trabajar con este gráfico de arriba hacia abajo, no siempre vamos a encontrar la representación del árbol de esta forma, pero puede que encontremos a las aristas como una flecha dirigida para que sepamos en orden leer las conexiones.

Cuando un nodo tiene hijos, esos hijos pueden ser 2 como máximo cuando hablamos de un árbol binario. En particular, vamos a clasificar los hijos como *hijo izquierdo* e *hijo derecho*. Cuando un nodo tiene hijos, se dice que es un *nodo interno*, mientras que si no los tiene se dice que es una *hoja*.

- 9.1. BT - Implementación estática
- 9.2. BT - Implementación dinámica
- 9.3. Recorridos recursivos
- 9.4. Recorridos iterativos
- 9.5. Recorrido por nivel
- 9.6. Árbol sesgado, degenerado, completo y perfecto
- 9.7. Caminos
- 9.8. Árbol Binario de Búsqueda (SBT)
- 9.9. SBT - Implementación
- 9.10. Árbol AVL
- 9.11. AVL - Implementación
- 9.12. Análisis de Costos

Capítulo 10

Resumen TDAs

10.1. Lista enlazada

Capítulo 11

Ejemplo: Juego de Cartas

Vamos a crear el siguiente juego: vamos a tomar un *mazo* de cartas, mezclarlo, y repartir las cartas cada 4 personas. No es importante que la *mano* de cada jugador tenga un orden. Luego, un jugador *lanzar* una carta y cada jugador, en orden, deben lanzar una carta mayor a la última lanzada. El primer jugador que no pueda lanzar una carta pierde.

11.1. Creación del mazo de cartas

Un mazo de cartas se lo puedo abstraer a una Pila de cartas. Es decir, una Pila que almacena un tipo de dato que representa una carta. Vamos a crear una clase que represente una carta:

```
3499 public class Card {  
3500  
3501     private final String suit;  
3502     private final String rank;  
3503  
3504     public Card(String suit, String rank) {  
3505         this.suit = suit;  
3506         this.rank = rank;  
3507     }  
3508  
3509     public String getSuit() {  
3510         return suit;  
3511     }  
3512  
3513     public String getRank() {  
3514         return rank;  
3515     }  
3516  
3517     public String toString() {  
3518         return rank + " of " + suit;  
3519     }  
3520 }
```

Una carta tiene un *rank*, los cuales son números del 2 al 10, *As* (Ace), *Jota* (Jack), *Reina* (Queen) o *Rey* (King). También tienen un *suit*: *Picas* (Spade), *Corazones* (Hearts), *Rombos* (Diamonds) o *Treboles* (Clubs).

Una carta es inalterable por lo que no necesitamos setters en nuestra clase. Esto nos permite decir, que como no se van a modificar los atributos, estos pueden ser constantes en cuanto a referencia a memoria (usamos *final*).

Pero además, esto sigue el concepto de inmutabilidad, dado que los atributos no se modifican, por lo que puede escribirse como un record:

```

3521 public record Card(String suit, String rank) {
3522
3523     public String toString() {
3524         return rank + " of " + suit;
3525     }
3526 }
```

Dado que las cartas son finitas en cuanto a valores posibles para `rank` y `suit`, entonces, podemos crear un enumerado (será extenso, pero cumple con las características de un enumerado):

```

3527 public enum Card {
3528     ACE_OF_CLUBS("Ace", "Clubs"),
3529     TWO_OF_CLUBS("2", "Clubs"),
3530     THREE_OF_CLUBS("3", "Clubs"),
3531     FOUR_OF_CLUBS("4", "Clubs"),
3532     FIVE_OF_CLUBS("5", "Clubs"),
3533     SIX_OF_CLUBS("6", "Clubs"),
3534     SEVEN_OF_CLUBS("7", "Clubs"),
3535     EIGHT_OF_CLUBS("8", "Clubs"),
3536     NINE_OF_CLUBS("9", "Clubs"),
3537     TEN_OF_CLUBS("10", "Clubs"),
3538     JACK_OF_CLUBS("Jack", "Clubs"),
3539     QUEEN_OF_CLUBS("Queen", "Clubs"),
3540     KING_OF_CLUBS("King", "Clubs"),
3541     ACE_OF_SPADES("Ace", "Spades"),
3542     TWO_OF_SPADES("2", "Spades"),
3543     THREE_OF_SPADES("3", "Spades"),
3544     FOUR_OF_SPADES("4", "Spades"),
3545     FIVE_OF_SPADES("5", "Spades"),
3546     SIX_OF_SPADES("6", "Spades"),
3547     SEVEN_OF_SPADES("7", "Spades"),
3548     EIGHT_OF_SPADES("8", "Spades"),
3549     NINE_OF_SPADES("9", "Spades"),
3550     TEN_OF_SPADES("10", "Spades"),
3551     JACK_OF_SPADES("Jack", "Spades"),
3552     QUEEN_OF_SPADES("Queen", "Spades"),
3553     KING_OF_SPADES("King", "Spades"),
3554     ACE_OF_DIAMONDS("Ace", "Diamonds"),
3555     TWO_OF_DIAMONDS("2", "Diamonds"),
3556     THREE_OF_DIAMONDS("3", "Diamonds"),
3557     FOUR_OF_DIAMONDS("4", "Diamonds"),
3558     FIVE_OF_DIAMONDS("5", "Diamonds"),
3559     SIX_OF_DIAMONDS("6", "Diamonds"),
3560     SEVEN_OF_DIAMONDS("7", "Diamonds"),
3561     EIGHT_OF_DIAMONDS("8", "Diamonds"),
3562     NINE_OF_DIAMONDS("9", "Diamonds"),
3563     TEN_OF_DIAMONDS("10", "Diamonds"),
3564     JACK_OF_DIAMONDS("Jack", "Diamonds"),
3565     QUEEN_OF_DIAMONDS("Queen", "Diamonds"),
3566     KING_OF_DIAMONDS("King", "Diamonds"),
3567     ACE_OF_HEARTS("Ace", "Hearts"),
3568     TWO_OF_HEARTS("2", "Hearts"),
3569     THREE_OF_HEARTS("3", "Hearts"),
3570     FOUR_OF_HEARTS("4", "Hearts"),
3571     FIVE_OF_HEARTS("5", "Hearts"),
3572     SIX_OF_HEARTS("6", "Hearts"),
3573     SEVEN_OF_HEARTS("7", "Hearts"),
3574     EIGHT_OF_HEARTS("8", "Hearts"),
3575     NINE_OF_HEARTS("9", "Hearts"),
```

```

3576     TEN_OF_HEARTS("10", "Hearts"),
3577     JACK_OF_HEARTS("Jack", "Hearts"),
3578     QUEEN_OF_HEARTS("Queen", "Hearts"),
3579     KING_OF_HEARTS("King", "Hearts");
3580
3581     private final String rank;
3582     private final String suit;
3583
3584     Card(String rank, String suit) {
3585         this.rank = rank;
3586         this.suit = suit;
3587     }
3588
3589     public String getRank() {
3590         return rank;
3591     }
3592
3593     public String getSuit() {
3594         return suit;
3595     }
3596 }
```

Agregar un comodín puede considerarse un caso especial, y deberíamos agregar una línea de código a este enum similar a JOKER("Joker", "Joker"). No lo haremos en este ejemplo.

Vamos a crear un TDA para representar el mazo, y una implementación posible, que iremos modificando a medida que lo necesitemos. Inicializar el mazo debería funcionar similar a la *creación* del mazo, que nos podría dar un *mazo vacío* o un *mazo completo*. Vamos a considerar acá el mazo completo. Entonces, el constructor representará el método que inicializa el mazo y le agrega todas las cartas del enum. Dado que un constructor no va en la interfaz, podemos colocar la lógica que agrega las cartas propiamente mediante un método inicializar:

```

3597 public interface IDeck {
3598
3599     /**
3600      * Quitar las cartas actuales del mazo de cartas y lo recrea con una tanda nueva de
3601      * cartas
3602     */
3603     void initialize();
3604 }
```

Y la implementación:

```

3605 public class Deck implements IDeck {
3606
3607     private IStack<Card> cards;
3608
3609     public Deck() {
3610         this.initialize();
3611     }
3612
3613     @Override
3614     public void initialize() {
3615         this.cards = new Stack<>();
3616         for(Card card : Card.values()) {
3617             this.cards.add(card);
3618         }
3619     }
3620 }
```

Aquí usamos un `foreach` para iterar un arreglo de cartas que obtenemos mediante el método `values()` que tiene cualquier enumerado para devolvernos todos sus valores definidos. Cada carta que obtuvimos la agregamos a nuestra Pila interna. Podríamos haber usado una Pila directamente, pero definir la estructura mazo nos deja exponer solo lo que necesitamos para considerarlo un mazo.

11.2. Mezcla de cartas

Una primera estrategia es la de replicar la baraja de cartas, pero esto implica mezclar varias cartas a la vez y en un orden, siendo que una instrucción no puede mezclar en paralelo a no ser que usemos hilos (no es parte del alcance de la materia). Vamos a implementar una versión alternativa, tal que tomamos una carta de la pila y la colocamos en otra posición de la pila de forma aleatoria.

Por ejemplo, supongamos un mazo que tiene solo 5 cartas:

$$deck = \begin{bmatrix} \text{King of spades} \\ 8 \text{ of hearts} \\ \text{Jack of diamonds} \\ 2 \text{ of diamonds} \\ 3 \text{ of clubs} \end{bmatrix}$$

Tomamos un elemento de forma aleatoria, volcando en una nueva pila los elementos de nuestro mazo, hasta que quede como tope la carta que queremos tomar. Para esto, podemos tomar un numero random entre 1 y 5 (nuestro mazo real tiene 52 naipes) que indique cuantas cartas hay que volcar. Supongamos que tenemos una variable `value` que almacena el numero aleatorio y en nuestro caso es 3, entonces, desapilamos 3 elementos y los apilamos en una pila auxiliar:

$$deck = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ 2 \text{ of diamonds} \\ 3 \text{ of clubs} \end{bmatrix}, aux = \begin{bmatrix} \emptyset \\ \emptyset \\ \text{Jack of diamonds} \\ 8 \text{ of hearts} \\ \text{King of spades} \end{bmatrix}$$

Luego, tomamos de nuestro mazo la carta en el tope y la guardamos en una variable, y volvemos a pasar los elementos de la pila auxiliar a nuestro mazo:

$$deck = \begin{bmatrix} \emptyset \\ \text{King of spades} \\ 8 \text{ of hearts} \\ \text{Jack of diamonds} \\ 3 \text{ of clubs} \end{bmatrix}$$

Ahora, tomamos una posición random entre 1 y 4 (el total de cartas decrementado en una unidad debido a la carta que sacamos), y volvemos a colocar la carta en esa posición. Entonces, supongamos que tocó la posición 1. Volcamos todos los elementos en aux hasta que quede la pila con la posición 1 vacía:

$$deck = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}, aux = \begin{bmatrix} \emptyset \\ 3 \text{ of clubs} \\ \text{Jack of diamonds} \\ 8 \text{ of hearts} \\ \text{King of spades} \end{bmatrix}$$

Colocamos 2 of diamonds que fue la carta que sacamos en la posición que queremos:

$$deck = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ 2 \text{ of diamonds} \end{bmatrix}, aux = \begin{bmatrix} \emptyset \\ 3 \text{ of clubs} \\ \text{Jack of diamonds} \\ 8 \text{ of hearts} \\ \text{King of spades} \end{bmatrix}$$

Y ahora volcamos de las cartas de la pila auxiliar al mazo nuevamente:

$$deck = \begin{bmatrix} \text{King of spades} \\ \text{8 of hearts} \\ \text{Jack of diamonds} \\ \text{3 of clubs} \\ \text{2 of diamonds} \end{bmatrix}$$

Nos ha quedado el mazo un poco más distorsionado que el mazo original. ¿Podría haber caído la carta en la misma posición en la que se encontraba? Si, pero con la suficiente cantidad de veces que repetimos esto, el mazo nos quedará desordenado. ¿Qué es suficiente? Hay que definirlo, mientras más grande sea la cantidad de iteraciones, más *entropía* tenemos.

Veamos su definición e implementación:

```

3621 public interface IDeck {
3622
3623     /**
3624      * QUITA LAS CARTAS ACTUALES DEL MAZO DE CARTAS Y LO RECREA CON UNA TANDA NUEVA DE
3625      * CARTAS
3626     */
3627     void initialize();
3628
3629     /**
3630      * SIMULA BARAJAR EL MAZO.
3631     */
3632     void shuffle();
3633 }
```



```

3634 import java.util.Random;
3635
3636 public class Deck implements IDeck {
3637
3638     private IStack<Card> cards;
3639     private int total;
3640
3641     public Deck() {
3642         this.initialize();
3643     }
3644
3645     @Override
3646     public void initialize() {
3647         this.cards = new Stack<>();
3648         for(Card card : Card.values()) {
3649             this.cards.add(card);
3650         }
3651         this.total = Card.values().length;
3652     }
3653
3654     @Override
3655     public void shuffle() {
3656         int value = this.random(0, this.total - 1);
3657         IStack<Card> aux = new Stack<>();
3658         for(int i = 0; i < value; i++) {
3659             aux.add(this.cards.getTop());
3660             this.cards.remove();
3661         }
3662
3663         Card card = this.cards.getTop();
```

```

3664     this.cards.remove();
3665     while(!aux.isEmpty()) {
3666         this.cards.add(aux.getTop());
3667         aux.remove();
3668     }
3669
3670     value = this.random(0, this.total - 2);
3671     for(int i = 0; i < value; i++) {
3672         aux.add(this.cards.getTop());
3673         this.cards.remove();
3674     }
3675
3676     this.cards.add(card);
3677     while(!aux.isEmpty()) {
3678         this.cards.add(aux.getTop());
3679         aux.remove();
3680     }
3681 }
3682
3683 /**
3684 * Genera un valor aleatorio.
3685 * @param min valor mínimo del rango, incluyendolo
3686 * @param max valor máximo del rango, incluyendolo
3687 * @return valor aleatorio entero entre min y max.
3688 */
3689 private int random(int min, int max) {
3690     return (new Random()).nextInt((max - min) + 1) + min;
3691 }
3692 }
```

Si calculamos la complejidad computacional del método que agregamos tenemos:

```

3693 public void shuffle() {
3694     int value = this.random(0, this.total - 1); // C1
3695     IStack<Card> aux = new Stack<>(); // C2
3696     for(int i = 0; i < value; i++) { // N1
3697         aux.add(this.cards.getTop()); // C3
3698         this.cards.remove(); // C4
3699     }
3700
3701     Card card = this.cards.getTop(); // C5
3702     this.cards.remove(); // C6
3703     while(!aux.isEmpty()) { // N2
3704         this.cards.add(aux.getTop()); // C7
3705         aux.remove(); // C8
3706     }
3707
3708     value = this.random(0, this.total - 2); // C9
3709     for(int i = 0; i < value; i++) { // N3
3710         aux.add(this.cards.getTop()); // C10
3711         this.cards.remove(); // C11
3712     }
3713
3714     this.cards.add(card); // C12
3715     while(!aux.isEmpty()) { // N4
3716         this.cards.add(aux.getTop()); // C13
3717         aux.remove(); // C14
3718     }
3719 }
```

Las constantes las podemos agrupar y los ciclos tienen complejidad lineal multiplicada por la complejidad de lo que contienen. Entonces su complejidad es

$$\mathcal{O}(C_1 + C_2 + N_1 * (C_3 + C_4) + C_5 + C_6 + \dots) \approx \mathcal{O}(C + 4 * N * C) \approx \mathcal{O}(4n)$$

($n \neq N$ porque, $n = 4 * N * C$). Esta es la complejidad de mezclar 1 carta, mezclar todas las cartas es multiplicar esta complejidad por la cantidad de veces que querremos aplicar.

Analicemos una opción mejor. Podemos tomar una carta de forma aleatoria y colocarla en una nueva pila de forma aleatoria. Pero dando un paso más, ya es suficientemente aleatorio tomar una carta de forma aleatoria y apilarla en una nueva pila. Podemos repetir este proceso hasta quedarnos sin cartas y luego volcar la pila auxiliar en la pila original (elementos aleatorios que se invierten siguen siendo aleatorios).

Ejemplo:

$$deck = \begin{bmatrix} \text{King of spades} \\ 8 \text{ of hearts} \\ \text{Jack of diamonds} \\ 2 \text{ of diamonds} \\ 3 \text{ of clubs} \end{bmatrix}$$

Supongamos que tomamos aleatoriamente Jack of diamonds, entonces al colocarlo en una pila nueva:

$$deck = \begin{bmatrix} \emptyset \\ \text{King of spades} \\ 8 \text{ of hearts} \\ 2 \text{ of diamonds} \\ 3 \text{ of clubs} \end{bmatrix}, aux = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \text{Jack of diamonds} \end{bmatrix}$$

Supongamos que ahora tomamos de forma aleatoria 3 of clubs:

$$deck = \begin{bmatrix} \emptyset \\ \emptyset \\ \text{King of spades} \\ 8 \text{ of hearts} \\ 2 \text{ of diamonds} \end{bmatrix}, aux = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ 3 \text{ of clubs} \\ \text{Jack of diamonds} \end{bmatrix}$$

Continuamos, obtendremos en una pila auxiliar las cartas mezcladas:

$$deck = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}, aux = \begin{bmatrix} \text{King of spades} \\ 2 \text{ of diamonds} \\ 8 \text{ of hearts} \\ 3 \text{ of clubs} \\ \text{Jack of diamonds} \end{bmatrix}$$

Luego, volcamos los elementos de la pila auxiliar a nuestro mazo:

$$deck = \begin{bmatrix} \text{Jack of diamonds} \\ 3 \text{ of clubs} \\ 8 \text{ of hearts} \\ 2 \text{ of diamonds} \\ \text{King of spades} \end{bmatrix}, aux = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}$$

Veamos la implementación:

```

3720  @Override
3721  public void shuffle() {
3722      IStack<Card> aux = new Stack<>();
3723      IStack<Card> aux2 = new Stack<>();
3724      while(!this.cards.isEmpty()) {
3725          int value = this.random(0, this.total - 1);

```

```

3726     for(int i = 0; i < value; i++) {
3727         aux.add(this.cards.getTop());
3728         this.cards.remove();
3729         this.total--;
3730     }
3731
3732     aux2.add(this.cards.getTop());
3733     this.cards.remove();
3734     this.total--;
3735     while(!aux.isEmpty()) {
3736         this.cards.add(aux.getTop());
3737         aux.remove();
3738     }
3739 }
3740
3741 while(!aux2.isEmpty()) {
3742     this.cards.add(aux2.getTop());
3743     aux2.remove();
3744 }
3745 }
```

Ahora la complejidad total es n por la complejidad de mezclar una carta. Es decir: $\mathcal{O}(2n^2)$. Podemos tratar de encontrar algoritmos más eficientes. Por ejemplo, podemos considerar que nuestra pila, luego de cierta posición almacena las cartas ya barajadas, ahorrando así una pila extra y entonces el ciclo que requiere volcar las cartas de una pila al mazo.

Vamos a ver una solución más expresiva: dado que los conjuntos no tienen orden, podemos volcar los elementos del mazo a un conjunto, y luego del conjunto al mazo. La complejidad será cuadrática sin duda porque un ciclo que vuelca los elementos al conjunto tiene que usar el método que agrega un elemento al conjunto, que a su vez tiene un ciclo para chequear elementos no repetidos. Sin embargo, esto es más expresivo y sencillo de entender. La implementación será (sin ser necesario un método que genera números aleatorios ya que de eso se encarga el TDA Conjunto):

```

3746 @Override
3747 public void shuffle() {
3748     ISet<Card> set = new Set<>();
3749     while (!this.cards.isEmpty()) {
3750         set.add(this.cards.getTop());
3751         this.cards.remove();
3752     }
3753     while (!set.isEmpty()) {
3754         Card card = set.choose();
3755         this.cards.add(card);
3756         set.remove(card);
3757     }
3758 }
```

Esto tiene sentido porque no hay elementos repetidos. Si hubiésemos tenido 2 comodines, el conjunto hubiese eliminado una copia, y la estrategia debería haber sido diferente.

11.3. Repartir las cartas

Vamos a suponer que cada participante le corresponde un número: Jugador 1, Jugador 2, Jugador 3 y Jugador 4. Podemos hacer que una estructura que tenga posiciones, tenga en la posición 0 la mano del Jugador 1, en la posición 1 la mano del Jugador 2, etc. Además, dado que una mano no tiene orden, y tampoco tenemos cartas repetidas, el TDA Conjunto es ideal para representar una mano. Estamos necesitando entonces un arreglo de Conjuntos (o su versión dinámica, la lista enlazada). Como conocemos la longitud,

y es fija, porque siempre son 4 jugadores, entonces no hace falta usar una lista enlazada, dado que es para tamaños variables, mientras que un arreglo nos da el beneficio del acceso directo a una posición.

Vamos entonces a crear un método que reparte las cartas entre 4 jugadores:

```

3759 public interface IDeck {
3760
3761     /**
3762      * Quita las cartas actuales del mazo de cartas y lo recrea con una tanda nueva de
3763      * cartas
3764      */
3765     void initialize();
3766
3767     /**
3768      * Simula barajar el mazo.
3769      */
3770     void shuffle();
3771
3772     /**
3773      * Reparte las cartas en 4 manos, representadas por conjuntos.
3774      * @return arreglo con cada posición representando a un jugador y su respectiva
3775      * mano.
3776      */
3777     ISet<Card>[] deal();
3778
3779 }
```

Ahora, una posible implementación es:

```

3778 @Override
3779 public ISet<Card>[] deal() {
3780     int count = 0;
3781     ISet<Card>[] array = (ISet<Card>[]) (new Set<?>[] {
3782         new Set<Card>(),
3783         new Set<Card>(),
3784         new Set<Card>(),
3785         new Set<Card>()
3786     });
3787
3788     while(!this.cards.isEmpty()) {
3789         array[count % 4].add(this.cards.getTop());
3790         this.cards.remove();
3791         count++;
3792         if(count == 4) {
3793             count = 0;
3794         }
3795     }
3796
3797     return array;
3798 }
```

Vamos de a poco. `ISetCard [] array` define un arreglo de conjuntos (usamos `ISet` en lugar de `Set` por polimorfismo), y le seteamos un valor con tipo `?` y que entre llaves le seteamos los valores iniciales (todos conjuntos vacíos, distintos, dado que usamos `new` para crear nuevas referencias a memoria para cada uno). En este caso tenemos un signo de interrogación para indicar que el tipo no lo conocemos. Esto sucede porque en tiempo de compilación, Java pierde los tipos genéricos y no sabe que el tipo del que esta creando coincide con el tipo de la variable, entonces colocar `Card` en lugar del signo de interrogación no sería interpretable por el lenguaje. Además, como los tipos de datos ahora son distintos, pero sabemos que estamos creandolo de forma correcta, le decimos a Java que no se preocupe por los tipos de datos usando un casteo explícito con `(ISetCard [])`.

Por otro lado, aplicar módulo 4 siempre da como resultado 0, 1, 2 o 3, que son las posiciones de los jugadores. Es opcional, aunque para mostrarlo decidí agregarlo, el condicional del final. Parecería poco óptimo, y de hecho lo es para una cantidad tan baja de elementos, pero para una cantidad de elementos suficientemente grande no lo es. Esto es porque `count` puede crecer indefinidamente, y al llegar al entero más grande, sumar 1 puede ocasionar error en muchos lenguajes. Para que el concepto sea escalable, podemos resetear la variable para que nunca se nos vaya de las manos.

11.4. Comentarios

La implementación del juego en sí no abarca temas de la materia. Se podría continuar esto planteando una estrategia. Por ejemplo ¿qué sucede si un jugador siempre tira su carta más alta en la primera ronda? Podría suceder que gane, o bien que luego si llega su turno no pueda tirar nada y entonces pierda. Esta estrategia es al estilo *greedy* que se introducirá en Programación III. Podemos por otro lado hacer una escala gradual, lanzando nuestra carta media, y luego la media de la media, etc, hasta tirar la última. O incluso, aplicar teoría de juegos, dado que hay que tratar de considerar las estrategias de otros jugadores.

Capítulo 12

Ejercicios resueltos

12.1. Series

12.1.1. Producto de Wallis (1655)

Motivación

En general, muchas funciones no tienen soluciones analíticas debido a la complejidad de despejar valores. Por ejemplo, ¿cómo despejamos x en la ecuación $\sin(x) + x^2 = 3$? En computación, nos resulta de interés avanzar en soluciones numéricas. Las aproximaciones por series es una estrategia sencilla por muy útil.

Un caso particular, es que calcular sumas es relativamente sencillo para una computadora. Entonces, una multiplicación también lo será por ser una continua aplicación de sumas. Análogamente, lo serán las potencias naturales. Por eso, evaluar polinomios es relativamente sencillo. La aproximación por series de Taylor, nos permite, por ejemplo, aproximar valores cerca de un punto

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (12.1)$$

La letra *sigma* mayúscula o la letra *pi* mayúscula representan la sumatoria y la productoria respectivamente. En grande, simplemente se representa la acción que se hará en cada iteración. Entonces, desde el punto de vista de la computación, estos son ciclos.

Por ejemplo, si consideramos

$$\sum_{i=0}^{n-1} f(i)$$

En código nosotros obtenemos

```
3799 double total = 0;
3800 for(int i = 0; i < n; i++) {
3801     total += f(i);
3802 }
```

Aquí *f* representa una función arbitraria que para nosotros podría ser una serie de pasos. La variable *total* comienza en 0 porque 0 es el *elemento neutro* de la suma. De forma similar,

$$\prod_{i=0}^{n-1} f(i)$$

se puede escribir en código como

```

3803 double total = 1;
3804 for(int i = 0; i < n; i++) {
3805     total *= f(i);
3806 }

```

Objetivo

Vamos a tratar de aproximar π . El producto de Wallis se obtiene a través de una factorización infinita de la función seno (la forma factorizada de un polinomio la obtenemos a partir de sus raíces, y de una función seno conocemos el comportamiento de sus infinitas raíces) y luego evaluando la función en un valor que nos sea útil. De esta forma, dado que la factorización es una multiplicación continua, John Wallis encuentra la siguiente expresión:

$$\frac{\pi}{2} = \prod_{n=1}^{\infty} \frac{2n}{2n-1} \cdot \frac{2n}{2n+1}$$

Vamos a usar esta expresión para calcular π .

Solución

Vamos a utilizar un ciclo para expresar la productoria, y el resultado obtenido será $\frac{\pi}{2}$. Luego, para obtener π debemos multiplicar $\frac{\pi}{2}$ por 2. La idea es simple, pero tenemos un problema. La igualdad es en el infinito, y en computación siempre esperamos que un programa termine. Entonces, hay que establecer un tope. Llamaremos *max* a ese tope, y en el límite $\max \rightarrow \infty$ conseguiremos *pi*. Nunca alcancemos el límite, pero este será más preciso mientras más grande sea *max*. Usaremos el tipo de dato *long* para conseguir un tope lo suficientemente alto. En general, a la *variable acumuladora* la llamaremos *variable candidata* porque en cada iteración se consigue un mejor candidato a ser el resultado que buscamos.

```

3807 public static double getPi(long max) {
3808     double candidate = 1;
3809     for(double n = 0; n < max; n++) { // n <= max genera una iteracion extra
3810         candidate *= (2*n / (2*n - 1)) * (2*n / (2*n + 1));
3811     }
3812     return candidate*2;
3813 }

```

Declaramos *n* de tipo *double* para evitar la división entera de Java, y no elegimos *float* para que el rango tenga sentido con el tipo *long* de *max*.

Optimización

Esta serie no es la que mejor aproxima a pi. Existen series como la de Ramanujan

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{n=0}^{\infty} \frac{(4n)!(1103 + 26390n)}{(n!)^4 396^{4n}} \quad (12.2)$$

que aproximan a pi a una velocidad mucho más alta. Hablamos de velocidad, no en el sentido de hardware sino de cálculo, donde con pocas iteraciones logramos una aproximación *bueno* mucho más rápido.

12.1.2. Problema de Basilea (1735)

En 1735, Leonhard Euler, resolvió una de los problemas más difíciles de Basilea. Lo que se buscaba, es un caso particular de la hoy conocida función *zeta* de Riemann:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} \quad (12.3)$$

donde s es un número complejo con parte real mayor que 1. En ese momento, se buscaba el resultado de $\zeta(s)$. Euler calculó el resultado como

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

Aplicando lo que vimos en el ejercicio anterior, obtenemos:

```
3814 public static double getPi(long max) {
3815     double candidate = 0;
3816     for(long n = 1; n < max; n++) { // n <= max genera una iteracion extra
3817         candidate += 1 / Math.pow(n, 2);
3818     }
3819     return Math.sqrt(candidate*6);
3820 }
```

Dado que n es de tipo `long`, podríamos usar $n*n$ en lugar de `Math.pow(n, 2)`.

12.1.3. Números primos

Motivación

A lo largo de la materia, vamos a estudiar la *complejidad computacional* de diferentes algoritmos. El cálculo de los números primos es suficientemente diverso en implementaciones como para comparar y practicar este análisis.

Versión 1

Un número primo se puede definir, en el contexto más típico (puede extenderse a negativos) como un número $n \in \mathbb{N}$ que es divisible por 1 y por si mismo. Entonces, el algoritmo más básico que podemos plantear es el siguiente:

```
3821 public static void printPrimes(int max) {
3822     for(int prime = 2; prime < max; prime++) {
3823         boolean isPrime = true;
3824         for(int i = 2; i < prime; i++) {
3825             if(prime % i == 0) {
3826                 isPrime = false;
3827                 break;
3828             }
3829         }
3830         if(isPrime) {
3831             System.out.println(prime);
3832         }
3833     }
3834 }
```

Versión 2

El ciclo anterior, para un n se están realizando iteraciones desde 2 hasta $n - 1$. Entonces, para un $n > 3$ tenemos $n - 3$ iteraciones posibles. El único primo par es el 2. Entonces, podemos modificar el ciclo para que analice solo los impares. Vamos a usar la propiedad de que los números impares sumados a un número par nos vuelven a dar un número impar. Particularmente, todos los números impares positivos se puede recorrer de dos en dos:

```
3835 public static void printPrimes(int max) {
3836     System.out.println(2);
3837     for(int prime = 3; prime < max; prime += 2) {
```

```

3838     boolean isPrime = true;
3839     for(int i = 2; i < prime; i++) {
3840         if (prime % i == 0) {
3841             isPrime = false;
3842             break;
3843         }
3844     }
3845     if(isPrime) {
3846         System.out.println(prime);
3847     }
3848 }
3849 }
```

Con esto, aproximadamente logramos reducir la cantidad de iteraciones a $\frac{n-3}{2}$.

Versión 3

Los divisores de un número tienen una propiedad de simetría. Por ejemplo, los divisores de 100 son 1, 2, 4, 5, 10, 20, 25, 50, 100. Estos valores pueden ser puestos de a dos, de la forma (1, 100), (2, 50), (4, 20), (10, 10), donde si divido a 100 por x en (x, y) nos da y , y si divido a 100 por y nos da x . Esta simetría nos dice que si hay algún divisor, es suficiente con mirar x o mirar y . Entonces, no hace falta recorrer hasta $prime - 1$, sino que podemos recorrer hasta su raíz cuadrada. Pero cuidado, porque ya no es una desigualdad del tipo $<$, dado que por ejemplo, para 100, 10 es una raíz. Debemos usar \leq :

```

3850 public static void printPrimes(int max) {
3851     System.out.println(2);
3852     for(int prime = 3; prime < max; prime += 2) {
3853         boolean isPrime = true;
3854         int end = ((int) Math.sqrt(prime)) + 1;
3855         for(int i = 2; i <= end; i++) {
3856             if (prime % i == 0) {
3857                 isPrime = false;
3858                 break;
3859             }
3860         }
3861         if(isPrime) {
3862             System.out.println(prime);
3863         }
3864     }
3865 }
```

Se realiza un casteo a `int` para que nos quede un entero exacto, sin importar los posibles errores del algoritmo de coma flotante. Como esto puede redondear hacia abajo, podemos reemplazar el casteo por un `Math.floor(Math.sqrt(prime))` o bien sumar 1 para evitar perder el análisis del caso de raíz cuadrada si justo se redondea para abajo. En este caso, opté por `int` dado que ocupa menos espacio en memoria.

Optimización

Existen muchas posibles optimizaciones. Pero para el curso, considero que es útil considerar almacenar los primos ya calculados en un array. Notemos que por ejemplo, el divisor 10 de 100, nos indica que como $10 = 2 * 5$, entonces 100 es divisible por 2 y 5. El *teorema fundamental de la aritmética* establece que cualquier número natural podrá ser escrito como un número primo o una multiplicación de estos. Entonces, es suficiente analizar para un n arbitrario, solo si este es divisible por los primos menores que él, que justamente son los ya calculados. El teorema de los números primos establece que

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n / \ln n} = 1 \quad (12.4)$$

donde $\pi(n)$ es la función de conteo de números primos que indica el número de primos menores o iguales a n , y $\ln n$ es la función logarítmica natural. Entonces, la cantidad de iteraciones que usaremos será parecida a esta función.

12.2. Matrices

12.2.1. Imprimir en pantalla una matriz cuadrada

Solución

Dada una matriz M , para imprimir cada elemento M_{ij} de M en la posición (i, j) , podemos escribir lo siguiente:

```
3866 public static void printMatrix(double[][] matrix) {
3867     for(int i = 0; i < matrix.length; i++) {
3868         for(int j = 0; j < matrix[i].length; j++) {
3869             System.out.println(matrix[i][j]);
3870         }
3871     }
3872 }
```

Sin embargo, esto imprimirá en pantalla una tira de números con un formato no muy lindo. La idea es que al imprimirse tenga un aspecto matricial. Para lograr esto, vamos a generar una fila en forma de string e imprimirla línea a línea. De esta forma, si imprimimos cada fila y seguido un salto de línea, tendrá un aspecto matricial. Podemos, para empezar, separar los números mediante espacios:

```
3873 public static void printMatrix(double[][] matrix) {
3874     for(int i = 0; i < matrix.length; i++) {
3875         String row = "";
3876         for (int j = 0; j < matrix[i].length; j++) {
3877             row += matrix[i][j] + " ";
3878         }
3879         System.out.println(row);
3880     }
3881 }
```

El problema, es que una matriz con números cantidad de dígitos distinto, podría hacer que la matriz no se viera del todo bien.

Por ejemplo:

```
3882 1 48 -31
3883 145 -2 39
3884 0 23 0
```

Entonces, una primera mejora, es usar tabulaciones:

```
3885 public static void printMatrix(double[][] matrix) {
3886     for(int i = 0; i < matrix.length; i++) {
3887         String row = "";
3888         for (int j = 0; j < matrix[i].length; j++) {
3889             row += matrix[i][j] + "\t";
3890         }
3891         System.out.println(row);
3892     }
3893 }
```

Lo cual nos mostraría:

```

3894 1   48  -31
3895 145 -2   9
3896 0   23  0

```

Por otro, esto nos está dejando un espacio al final. Podemos solucionar esto haciendo un trim de la string:

```

3897 public static void printMatrix(double[][] matrix) {
3898     for(int i = 0; i < matrix.length; i++) {
3899         String row = "";
3900         for (int j = 0; j < matrix[i].length; j++) {
3901             row += matrix[i][j] + "\t";
3902         }
3903         System.out.println(row.trim());
3904     }
3905 }

```

Optimización

Esto funciona bien para números de no más de tres dígitos, y en los ejemplos se omitió el .0 que se agregan al final por ser de tipo double. Esto hace que la matriz no se vea bien. Agregar más de una tabulación no es una solución válida, dado que sigue siendo una cantidad constante de tabulaciones y lo que se necesita en este caso es una cantidad variable. Para hacer esto, lo que se necesitaría es un algoritmo que busque el número más largo a imprimir, y a cada uno de los demás números se les concatene una cantidad suficiente de tabulaciones para que todos se vean igual en pantalla.

Por otro lado, se puede optar por no usar el `trim`, y usar en su lugar un condicional donde si estamos frente a la última iteración de cada fila, entonces no concatene un espacio. En este caso, no opté por este camino dado que para un número muy alto de iteraciones esta instrucción se estaría ejecutando, comparado con la única vez que se llama al método `trim`.

12.2.2. Determinante matrix 2x2

Motivación

Las transformaciones lineales aplicadas a una matriz permiten estirarlas, rotarlas o desplazarlas. Entonces, esto es extremadamente útil cuando queremos trabajar con imágenes y con colores. Quedemos con ejemplos de imágenes para hacerlo más visual. Si tenemos una matriz M y la aplicamos a una imagen para invertir sus colores:

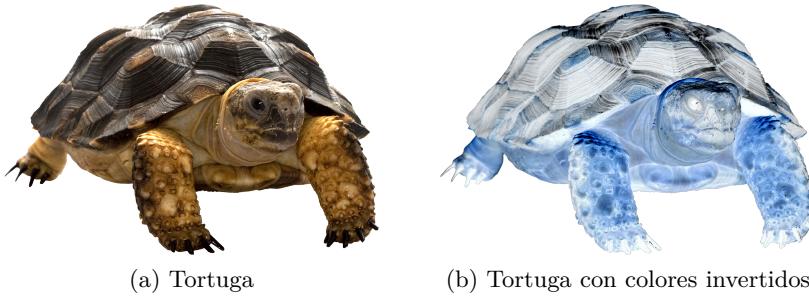


Figura 12.1: Colores invertidos

Esta operación me permite pasar de la Figura A, a la Figura B, pero también me permite pasar de la Figura B a la Figura A. Mientras que si achatamos la figura:

Aquí no podremos pasar de la Figura B a la Figura A nuevamente, dado que los píxeles perdidos podrían haber sido de cualquier valor. Cuando una operación matricial es irreversible, su determinante es 0 y lo podremos saber con anticipación calculandolo.

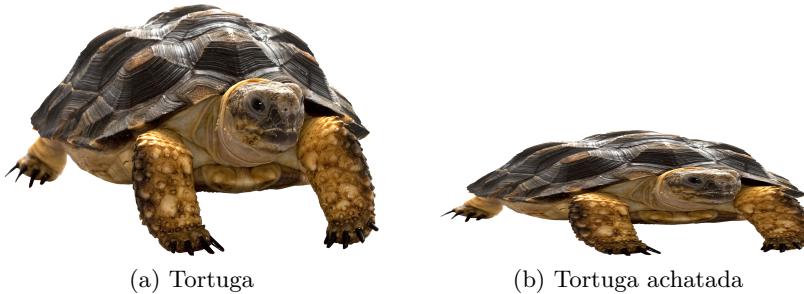


Figura 12.2: Reducción de tamaño de imagen

Solución

Sea una matriz con $a, b, c, d \in \mathbb{R}$, entonces

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

Escribiendo esto en código:

```
3906 public static double determinant(double[][] matrix) {
3907     return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0];
3908 }
```

Optimización

El código anterior es válido, pero no es escalable. Para poder generalizarlo, debemos usar algún método para matrices de tamaño $n \times n$, como por ejemplo la descomposición de Cholesky, usar Sarrus o la regla de Cramer.

12.2.3. Traza de una matriz

La traza de una matriz es especialmente útil cuando trabajamos con diferentes bases, porque realizando una transformación de coordenadas, la traza de la matriz modificada tiene relaciones con la traza original. Para calcular la traza de una matriz cuadrada, tenemos que sumar los elementos de su diagonal principal (desde la posición $(0, 0)$ a la posición (n, n)). Para calcularlo en Java:

```
3909 public static double determinant(double[][] matrix) {
3910     double trace = 0;
3911     for(int i = 0; i < matrix.length; i++) {
3912         trace += matrix[i][i];
3913     }
3914     return trace;
3915 }
```

12.2.4. Multiplicación escalar

Las transformaciones lineales de una matriz son la traslación, la rotación, la escalabilidad o una mezcla de estas propiedades. El producto escalar es el que nos permite escalar toda la matriz. Respecto a vectores por ejemplo, nos permitiría estirarlos sin cambiar su dirección (pero quizás si en sentido). Para hacerlo, debemos generar una nueva matriz, o modificar la actual, para que cada uno de sus elementos se multiplique por un escalar.

Un *escalar* se define necesariamente sobre la propia definición de espacio vectorial. Un elemento de un espacio vectorial se denomina vector. Esto es más general que los vectores que vemos en ingeniería dado

que existen espacios vectoriales de polinomios, o por ejemplo, espacios vectoriales de espacios vectoriales. Mientras que, tomar un elemento de ese vector y aplicarlo a cada elemento del vector se considera un escalar. Es por eso, que como cada elemento de una matriz en nuestro espacio es de tipo `double`, un escalar será para nosotros de tipo `double`.

Un escalar, por estándar, es nombrado como α , β , γ , etc. En Java, nuestro código será:

```
3916 public static double[][][] scalarProduct(double[][][] matrix, double alpha) {
3917     double[][] result = new double[matrix.length][];
3918     for(int i = 0; i < matrix.length; i++) {
3919         result[i] = new double[matrix[i].length];
3920         for (int j = 0; j < matrix[i].length; j++) {
3921             result[i][j] = matrix[i][j] * alpha;
3922         }
3923     }
3924     return result;
3925 }
```

Tener cuidado con las instrucciones, dado que si la inicialización de la nueva matriz es `new double[matrix.length] [m]` fallará para la matriz de dimensión nula.

12.3. Recursividad

12.3.1. Suma de dígitos

Usando strings

Suponiendo que el número dentro de la string es positivo (sino, podemos hacer un replace del signo de negación por una string vacía), entonces una posible solución sería:

```
3926 public static long sumOfDigits(String number) {
3927     if(number.length() == 0) {
3928         return 0;
3929     }
3930     byte n = (byte) (number.charAt(0) - '0');
3931     return n + sumOfDigits(number.substring(1));
3932 }
```

Pero dado que las strings son inmutables, el método `substring` en cada iteración puede ser muy costoso. Si el número dentro de la string se puede convertir a `long` sin perder información, una mejor solución es usar recursividad con operaciones matemáticas.

Usando módulo

```
3933 public static long sumOfDigits(long n) {
3934     if(n < 10) {
3935         return n;
3936     }
3937     return (n % 10) + sumOfDigits(n / 10);
3938 }
```

La recursividad nos permite una legibilidad de código superior pero no de eficiencia. En este caso, quizás la comparación se podría optimizar por una comparación `n == 0` seguido de un `return 0;`. Quizás reescribirse con la sentencia `short-if`. Aún así, la diferencia no será mucha.

12.3.2. Divisores de un número

Este ejercicio es un derivado del cálculo de los números primos. Revisar esa sección puede resultar útil.

Para agregar otra posible solución, se adjunta una versión básica de este algoritmo usando el operador módulo:

```

3939 public static void printDividers(int n) {
3940     System.out.println(1);
3941     for(int i = 2; i < n; i++) {
3942         if(n % i == 0) {
3943             System.out.println(i);
3944         }
3945     }
3946     if(n != 1) {
3947         System.out.println(n);
3948     }
3949 }
```

12.3.3. Números perfectos

Sea $n \in \mathbb{N}$, entonces n es *perfecto* si la suma de sus divisores es igual a n . Si el resultado es mayor a n se dice que es *abundante* y si es menor se dice que es *deficiente*. Vamos a modificar el ejemplo anterior para saber si un número arbitrario es perfecto:

```

3950 public static boolean isPerfect(int n) {
3951     int total = n == 1 ? 0 : 1;
3952     for(int i = 2; i < n; i++) {
3953         if(n % i == 0) {
3954             total += i;
3955         }
3956     }
3957     return n == total;
3958 }
```

12.4. Tipos de Datos Abstractos

12.4.1. Pair

Motivación

La TDA Pair es un tipo de dato que agrupa dos datos cualesquiera. Es la parte fundamental de una base de datos de tipo key-value, pero también nos permite evitar usar arrays de tipo Object cuando queremos devolver 2 valores de distinto tipo en una función. Dependiendo el contexto, esto es equivalente a TDAs Node, Couple, Coordinate, etc.

Definición

Dado el uso general de este TDA, tenemos 2 métodos dentro de su comportamiento. El primero es obtener su primer valor, y el segundo es obtener su segundo valor. Esto en POO, se considera que es una clase sin comportamiento real, dado que estamos teniendo solamente getters y necesitamos de alguna otra acción. Debido a su generalidad no podemos. Además, sus dos valores pueden tener cualquier nombre, por lo que vamos a considerar un caso particular, el de las bases de datos key-value. Precisamente, al primer elemento lo llamamos *key* y al segundo *value*. Sin utilizar programación genérica, nuestra interfaz será:

```

3959 public interface Pair {
3960
3961     int getKey();
3962
3963     int getValue();
```

3964
3965 }

Esta solución nos permite solo un tipo de dato. En este caso `int`, pero podrían ser `String`, `long`, o incluso tipos de datos distintos.

Implementación

Por patrón GRASP (específicamente *experto en información*) dejaremos los atributos con scope `private`. Además, no consideramos instancias de `Pair` vacías, dado que su propósito es agrupar dos elementos, no dejarlos sueltos. Debido a esto, vamos a colocar un *constructor con todos los argumentos* para considerar una asignación obligatoria.

La implementación será:

```
3966 public class Pair implements IPair {
3967
3968     private int key;
3969     private int value;
3970
3971     public Pair(int key, int value) {
3972         this.key = key;
3973         this.value = value;
3974     }
3975
3976     @Override
3977     public int getKey() {
3978         return 0;
3979     }
3980
3981     @Override
3982     public int getValue() {
3983         return 0;
3984     }
3985 }
```

El `@Override` lo dejaremos por prolíjidad. Si prestamos atención, no hay manera de que una instancia cambie los valores de sus atributos asignados. Una mejora a este TDA será hacer sus atributos `final`. En este contexto, `final` indica que sus valores no van cambiar de posición en memoria, es decir, el comportamiento de las constantes. Con esto, nuestro TDA es *immutable*:

```
3986 public class Pair implements IPair {
3987
3988     private final int key;
3989     private final int value;
3990
3991     public Pair(int key, int value) {
3992         this.key = key;
3993         this.value = value;
3994     }
3995
3996     @Override
3997     public int getKey() {
3998         return 0;
3999     }
4000
4001     @Override
4002     public int getValue() {
4003         return 0;
4004     }
4005 }
```

4005 }

Records

Java nos permite que en casos excepcionales, donde tenemos estructuras inmutables luego de su inicialización, usemos un `record` en lugar de un `class`. Los records tienen sus atributos públicos, por lo que los métodos se vuelven innecesarios, y entonces, también se vuelve innecesaria la interfaz. Entonces, nuestro `Pair` será:

4006 public record Pair (int key, int value) { }

La forma de acceder a los atributos de un `record` es:

```
4007 Pair pair = new Pair(1, 2);
4008 System.out.println("key: " + pair.key());
4009 System.out.println("value: " + pair.value());
```

Programación genérica

Implementación:

4010 public record Pair<K, V> (K key, V value) { }

Uso:

```
4011 Pair<Integer, Double> pair = new Pair<>(1, 2d);
4012 System.out.println("key: " + pair.key());
4013 System.out.println("value: " + pair.value());
```

Usamos `K` para indicar que el tipo de dato va a ser usado por un atributo de tipo `key`, y `V` para un atributo de tipo `value`. En la expresión `new Pair<>(1, 2d)`, el operador diamante está vacío debido a que Java puede deducir que los tipos de datos son los mismo que pusimos a la izquierda de la asignación.s

12.4.2. Vector

Motivación

Un vector es el elemento básico de un espacio vectorial y su significado cambia dependiendo del contexto. Vamos a crear un *vector columna* propio de cuando se empieza a estudiar álgebra lineal. Los vectores son fundamentales para simular física y para calcular máximos y mínimos con gradientes.

2 dimensiones

Interfaz:

```
4014 public interface IVector {
4015
4016     double scalarProduct(IVector other);
4017     IVector generateOrthogonal();
4018     double getX();
4019     double getY();
4020 }
4021 }
```

Implementación:

```

4022 public class Vector implements IVector {
4023
4024     private final double x;
4025     private final double y;
4026
4027     public Vector(double x, double y) {
4028         this.x = x;
4029         this.y = y;
4030     }
4031
4032     @Override
4033     public double scalarProduct(IVector other) {
4034         return this.x * other.getX() + this.y * other.getY();
4035     }
4036
4037     @Override
4038     public IVector generateOrthogonal() {
4039         return new Vector(-this.y, -this.x);
4040     }
4041
4042     @Override
4043     public double getX() {
4044         return this.x;
4045     }
4046
4047     @Override
4048     public double getY() {
4049         return this.y;
4050     }
4051 }
```

Cuando generamos un vector ortogonal, es válido tomar cualquier vector en su recta ortogonal (porque estamos en dos dimensiones, si fueran 3, sería un plano ortogonal). La implementación puesta puede variar.

N dimensiones

Interfaz:

```

4052 public interface IVector {
4053
4054     /**
4055      * Precondicion: los vectores deben tener la misma
4056      * dimension.
4057      * @param other con igual dimension
4058      * @return producto escalar
4059      */
4060     double scalarProduct(IVector other);
4061
4062     IVector generateOrthogonal();
4063
4064     double[] getCoordinates();
4065
4066 }
```

Implementación:

```

4067 public class Vector implements IVector {
4068
4069     private final double[] coordinates;
```

```

4070
4071     public Vector(double[] coordinates) {
4072         this.coordinates = coordinates;
4073     }
4074
4075     @Override
4076     public double scalarProduct(IVector other) {
4077         double result = 0;
4078         for(int i = 0; i < coordinates.length; i++) {
4079             result += coordinates[i] * other.getCoordinates()[i];
4080         }
4081         return result;
4082     }
4083
4084     @Override
4085     public IVector generateOrthogonal() {
4086         double[] orthogonalCoordinates = new double[this.coordinates.length];
4087         for(int i = 0; i < coordinates.length; i++) {
4088             orthogonalCoordinates[i] = -this.coordinates[i];
4089         }
4090         return new Vector(orthogonalCoordinates);
4091     }
4092
4093     @Override
4094     public double[] getCoordinates() {
4095         return this.coordinates;
4096     }
4097 }
```

Nuevamente, existen muchas formas de generar un vector ortogonal. De hecho, cualquier múltiplo del vector elegido para esta implementación lo será. Por otro lado, mantenemos el tipo `double` para que sea válido en \mathbb{R} . En el caso de querer usar \mathbb{C} , debemos crear nuestro propio TDA para números complejos.

12.4.3. Qubit

Un *qubit* un bit cuántico, el cual internamente puede ser representado por un vector complejo. Vamos a crear nuestro TDA para números complejos y modificar el vector anterior para que lo admita. Como ventaja, representar un qubit único es suficiente con un vector de dimensión 2. Vamos a omitir la interfaz `IComplex` e ir directo a su implementación, y a la modificación de `IVector` y `Vector`:

TDA Complex:

```

4098 public record Complex(double a, double b) {
4099
4100     public Complex getConjugate() {
4101         return new Complex(a, -b);
4102     }
4103
4104     public Complex timesBy(Complex other) {
4105         return new Complex(
4106             a * other.a() - b * other.b(),
4107             a * other.b() + b * other.a()
4108         );
4109     }
4110
4111     public Complex plusBy(Complex other) {
4112         return new Complex(a + other.a(), b + other.b());
4113     }
4114 }
```

```

4115     public double getModule() {
4116         return Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));
4117     }
4118 }
4119 }
```

Interfaz:

```

4120 interface IVector {
4121
4122     Complex scalarProduct(IVector other);
4123     IVector generateOrthogonal();
4124     Complex getX();
4125     Complex getY();
4126
4127 }
```

Implementación:

```

4128 public class Vector implements IVector {
4129
4130     private final Complex x;
4131     private final Complex y;
4132
4133     public Vector(Complex x, Complex y) {
4134         this.x = x;
4135         this.y = y;
4136     }
4137
4138     @Override
4139     public Complex scalarProduct(IVector other) {
4140         return this.x.getConjugate().timesBy(other.getX()).plusBy(this.y.getConjugate()
4141             .timesBy(other.getY()));
4142     }
4143
4144     @Override
4145     public IVector generateOrthogonal() {
4146         return new Vector(this.y.timesBy(new Complex(-1, 0)), this.x.timesBy(new
4147             Complex(-1, 0)));
4148     }
4149
4150     @Override
4151     public Complex getX() {
4152         return this.x;
4153     }
4154
4155     @Override
4156     public Complex getY() {
4157         return this.y;
4158     }
4159 }
```

Con esto, podemos crear nuestro TDA:

Interfaz:

```

4158 public interface IQubit {
4159
4160     IQubit applyNotGateVector();
4161     IQubit projection();
4162 }
```

4163 }

La compuerta NOT cuántica, consiste en intercambiar las componentes del vector. Sea $|\psi\rangle$ un qubit arbitrario, entonces

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

Que es equivalente a

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

donde

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Entonces $NOT|\psi\rangle = \beta|0\rangle + \alpha|1\rangle$.

Por otro lado, un qubit tiene que tener la particularidad de que $|\alpha|^2 + |\beta|^2 = 1$. Un qubit puede ser proyectado a $|0\rangle$ con probabilidad $|\alpha|^2$ o a $|1\rangle$ con probabilidad $|\beta|^2$. Podemos valernos de que siempre suman una para prestarle únicamente atención a α .

```
4164 public class Qubit implements IQubit {
4165
4166     private IVector vector;
4167
4168     public Qubit(Complex alpha, Complex beta) {
4169         this.vector = new Vector(alpha, beta);
4170     }
4171
4172     @Override
4173     public IQubit applyNotGateVector() {
4174         return new Qubit(this.vector.getY(), this.vector.getX());
4175     }
4176
4177     @Override
4178     public IQubit projection() {
4179         double probability = this.vector.getX().getModule();
4180         double random = Math.random();
4181         if(random < probability) {
4182             return new Qubit(new Complex(1, 0), new Complex(0, 0));
4183         }
4184         return new Qubit(new Complex(0, 0), new Complex(1, 0));
4185     }
4186 }
```

12.5. Pila

12.5.1. Cantidad de elementos par

Estrategia

Supongamos que tenemos la siguiente pila:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix}$$

Nuestro objetivo es conocer si tiene una cantidad de elementos par.

Estrategia:

1. Pasamos los elementos a dos pilas auxiliares de forma intercalada.
2. Desapilamos las pilas auxiliares a la vez hasta que por lo menos una este vacía.
3. La cantidad de elementos es par solo si al final de este proceso todas las pilas quedaron vacías.

En nuestro ejemplo:

$$p = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix}, aux = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}, aux2 = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}$$

Con el paso (1) obtenemos:

$$p = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}, aux = \begin{bmatrix} a_5 \\ a_3 \\ a_1 \end{bmatrix}, aux2 = \begin{bmatrix} \emptyset \\ \emptyset \\ a_4 \end{bmatrix}$$

Con el paso (2) obtenemos:

$$p = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}, aux = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ a_5 \end{bmatrix}, aux2 = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{bmatrix}$$

Con el paso (3), sabemos que al no haber quedado cada pila vacía, entonces la cantidad de elementos es impar (efectivamente tenía 5 elementos).

Código

```

4187 public static boolean totalOfEvenElements(IStack<Integer> stack) {
4188     IStack<Integer> aux = new Stack<>();
4189     IStack<Integer> aux2 = new Stack<>();
4190     while(!stack.isEmpty()) {
4191         aux.add(stack.getTop());
4192         stack.remove();
4193         if(!stack.isEmpty()) {
4194             aux2.add(stack.getTop());
4195             stack.remove();
4196         }
4197     }
4198     while(!aux.isEmpty() && !aux2.isEmpty()) {
4199         aux.remove();
4200         aux2.remove();

```

```

4201     }
4202     return aux.isEmpty() && aux2.isEmpty();
4203 }
```

Generalización

La generalización de este algoritmo es el planteo de cómo saber si la cantidad de elementos de una pila es múltiplo de un n arbitrario. La modificación necesaria en el algoritmo anterior es la cantidad de pilas auxiliares. Si la cantidad aumenta a n , entonces:

1. Pasamos los elementos a las n pilas auxiliares de forma intercalada.
2. Desapilamos las pilas auxiliares a la vez hasta que por lo menos una esté vacía.
3. La cantidad de elementos es múltiplo de n solo si al final de este proceso todas las pilas quedaron vacías.

Implementación en código:

```

4204 public static boolean totalOfEvenElements(IStack<Integer> stack, int divisor) {
4205     IStack<Integer>[] aux = (IStack<Integer>[]) (new Stack<?>[divisor]);
4206     for(int i = 0; i < divisor; i++) {
4207         aux[i] = new Stack<>();
4208     }
4209     int count = 0;
4210     while(!stack.isEmpty()) {
4211         count = count % divisor;
4212         aux[count].add(stack.getTop());
4213         stack.remove();
4214         count++;
4215     }
4216
4217     while(!aux[0].isEmpty()) {
4218         for(int i = 0; i < divisor; i++) {
4219             if(aux[i].isEmpty())
4220                 return false;
4221             }
4222             aux[i].remove();
4223         }
4224     }
4225
4226     return true;
4227 }
```

12.5.2. Eliminar repetidos sin auxiliares

La idea es lograr eliminar los elementos repetidos de una pila sin tener usar otro tipo de estructura distinta de pila, pero tampoco usando variables auxiliares.

Estrategia

Planteamos primero una estrategia con variables auxiliares.

- Creamos una variable auxiliar y guardamos el tope de la pila.
- Creamos una pila que tenga los elementos no repetidos y agregamos este tope.

- Creamos una pila auxiliar para recorrer la pila original sin perder sus elementos. Vamos a pasar uno a uno los elementos de la pila original a la auxiliar.
- Cuando pasamos elemento a elemento, solo lo hacemos si este elemento es diferente al de nuestra variable.

Ahora bien, con estos pasos, nos quedan los elementos de la pila original en la pila auxiliar. Aquí es importante plantear algo: ¿Queremos conservar el orden de los elementos o no? Si no nos interesa, podemos repetir el algoritmo, pero intercambiando los roles de pila auxiliar y la original. En cambio, si nos interesa el orden, pasamos los elementos de la pila auxiliar a la original y volvemos a repetir el proceso hasta que la pila original quede vacía.

Además, en caso de que nos importe el orden, es importante entender que la pila con los elementos no repetidos queda invertida respecto a la original, por lo que hay que invertirla para conservar el orden.

Por último, estaba prohibido en este caso usar variables auxiliares. Por lo que, vamos a crear una pila que manearemos siempre de a 1 elemento, por lo que dará el aspecto de ser una variable.

Implementación

Vamos a implementar el código de eliminar los repetidos manteniendo el orden:

```

4228 public static void deleteRepeatedElements(IStack stack) {
4229     if(stack == null || stack.isEmpty()) {
4230         return;
4231     }
4232
4233     IStack variable = new Stack();
4234     variable.add(stack.getTop());
4235     stack.remove();
4236     if(stack.isEmpty()) {
4237         stack.add(variable.getTop());
4238         return; // No hace falta vaciar variable
4239     }
4240
4241     IStack result = new Stack();
4242     IStack aux = new Stack();
4243
4244     while(!stack.isEmpty()) {
4245         while(!stack.isEmpty()) {
4246             if(stack.getTop() != variable.getTop()) {
4247                 aux.add(stack.getTop());
4248             }
4249             stack.remove();
4250         }
4251
4252         while(!aux.isEmpty()) {
4253             stack.add(aux.getTop());
4254             aux.remove();
4255         }
4256         result.add(variable.getTop());
4257         variable.remove();
4258         if(!stack.isEmpty()) {
4259             variable.add(stack.getTop());
4260             stack.remove();
4261         }
4262     }
4263
4264     if(!variable.isEmpty()) {
4265         result.add(variable.getTop());
4266         variable.remove();

```

```

4267    }
4268
4269    while(!result.isEmpty()) {
4270        aux.add(result.getTop());
4271        result.remove();
4272    }
4273    IStack aux2 = new Stack();
4274    while(!aux.isEmpty()) {
4275        aux2.add(aux.getTop());
4276        aux.remove();
4277    }
4278    while(!aux2.isEmpty()) {
4279        stack.add(aux2.getTop());
4280        aux2.remove();
4281    }
4282
4283 }
```

12.5.3. Invertir una pila sin auxiliares

Vamos a intentar invertir una Pila sin utilizar otras estructuras (incluidas otras pilas). Además, no vamos a devolver una copia, sino que vamos a modificar la pila original.

12.5.4. Estrategia e implementación

Invertir una pila de forma mutable, como se nos plantea en este problema, significa quitar todos los elementos y volverlos a colocar de tal forma que queden invertidos. Pero esto requiere iterar. Si iteramos los elementos para quitarlos, no hay ningún problema, podemos hacer algo como lo siguiente:

```

4284 public static void empty(IStack stack) {
4285     while(!stack.isEmpty()) {
4286         stack.remove();
4287     }
4288 }
```

El problema que nos genera esto, es que no tenemos ningún backup de los elementos para agregarlos nuevamente en orden inverso. Si intentamos crear n variables, bastará con una pila de $n + 1$ elementos para que ya no nos alcancen. Entonces, necesitamos una cantidad variable de variables. Esto, lo conseguimos con otra estructura, o bien, mediante *recursividad*. Aquí estará la clave.

Veamos que pasa si en cada iteración sacamos el tope y luego lo volvemos a agregar:

```

4289 public static void test(IStack stack) {
4290     if(stack.isEmpty())
4291         return;
4292     int top = stack.getTop();
4293     test(stack);
4294     stack.add(top);
4295 }
4296 }
```

Al final de la ejecución de esta función no tenemos ningún cambio. La pila vuelve a estar como estaba originalmente. Como comentario, para Cola esta técnica puede ser buena para revertirla.

Sin embargo podemos empezar a aprovechar características de las funciones a nuestro favor. Podemos generar una comunicación entre las diferentes funciones utilizando sus parámetros y los valores que se devuelven, o incluso, otras funciones (generando recursiones anidadas o consecutivas).

Si hacemos una cadena de valores que se devuelven, podemos conseguir en la primera llamada a la función el último elemento, y colocarlo en el tope:

```

4297 public static int test2(IStack stack) {
4298     if(stack.isEmpty()) {
4299         throw new RuntimeException("Empty stack");
4300     }
4301     int top = stack.getTop();
4302     stack.remove();
4303     if(stack.isEmpty()) {
4304         return top;
4305     }
4306
4307     int result = test2(stack);
4308     stack.add(top);
4309     return result;
4310 }
```

Aquí incluso estamos aprovechando algo más, la referencia a memoria de la Pila nos permite que todas las llamadas a la función la compartan. Sin embargo, solo logramos mover un elemento hasta el momento.

Si ahora hacemos una cadena de valores que se pasan por parámetro, podemos lograr algo similar: que el tope de la pila pase a estar en el fondo de la pila.

```

4311 public static void test3(IStack stack, Integer top) {
4312     if(top == null && stack.isEmpty()) {
4313         throw new RuntimeException("Empty stack");
4314     }
4315     if(stack.isEmpty()) {
4316         stack.add(top);
4317         return;
4318     }
4319
4320     int aux = stack.getTop();
4321     stack.remove();
4322
4323     test3(stack, (top == null) ? aux : top);
4324     stack.add(aux);
4325 }
```

En este caso, se usa `Integer` para aceptar `null` dado que la primera vez que se llame a esta función se ha tomado ningún tope, y luego de haberlo tomado, el valor se vuelve distinto de `null` y podemos usar esto para no tomar otro tope y arrastrar este parámetro hasta agregarlo al fondo de la pila.

Con toda esta información, podemos plantear que esto último, no es más que un caso particular de poder llevar un elemento a la posición que queremos, no solo al fondo de la pila. Veamos el caso donde podemos llevarlo a una posición en particular:

```

4326 public static void move(IStack stack, Integer top, int index) {
4327     if(top == null && stack.isEmpty()) {
4328         throw new RuntimeException("Empty stack");
4329     }
4330
4331     if(index == -1) {
4332         stack.add(top);
4333         return;
4334     }
4335
4336     int aux = stack.getTop();
4337     stack.remove();
4338
4339     move(stack, (top == null) ? aux : top, index - 1);
4340 }
```

```

4341     if(top != null) {
4342         stack.add(aux);
4343     }
4344 }
```

Ahora, podemos hacer un contador que nos permita ir poniendo el tope actual siempre en una posición distinta, pero en el orden de la posición del fondo hacia el mismo tope. De esta forma, logramos que la pila se invierta. Para esto, debemos usar un ciclo que nos permita aplicar esta función para todas sus posiciones posibles. Entonces, vamos a necesitar conocer el total de elementos de una Pila. Esto lo podemos hacer modificando el primer ejemplo que vimos en este análisis.

```

4345 public static void revert(IStack stack) {
4346     int size = size(stack);
4347     for(int i = 0; i < size; i++) {
4348         move(stack, null, size - 1 - i);
4349     }
4350 }
4351
4352 private static int size(IStack stack) {
4353     if(stack.isEmpty()) {
4354         return 0;
4355     }
4356     int top = stack.getTop();
4357     stack.remove();
4358     int size = 1 + size(stack);
4359     stack.add(top);
4360     return size;
4361 }
```

Cabe destacar que pudimos invertir una pila sin usar otras pilas explícitamente, pero implícitamente estamos usando la Pila de llamadas que usa internamente Java para las llamadas de funciones.

12.6. Cola

12.6.1. Invertir elementos sin una cola auxiliar ni estructuras auxiliares

La técnica que vamos a usar para resolverlo es aplicar un algoritmo de ordenamiento con recursividad. La idea es simple: el caso base es que una cola con dos elementos desordenados se invierte, y una cola con un elemento o 0 no hace falta ordenarla; por otro lado el caso general es partir la cola en dos, ordenarlas, y luego ir tomando de menor a mayor los primeros de cada cola de forma intercalada.

Veamos un paso a paso. Vamos a ordenar la siguiente cola:

$$\text{queue} = \overleftarrow{[1] [3] [-1] [9] [8]}$$

Como posee más de dos elementos, dividimos en dos colas y las ordenamos por separado. La primera cola podría ser:

$$\text{queue}_1 = \overleftarrow{[1] [3] [-1]}$$

Nuevamente, partimos la cola en dos dado que tiene más de dos elementos:

$$\text{queue}_{11} = \overleftarrow{[1] [3]}$$

En este caso no hace falta invertir los elementos porque ya está ordenado. Veamos la segunda parte:

$$\text{queue}_{12} = \overleftarrow{[-1]}$$

Como tiene un único elemento, terminamos. Ahora, para recuperar la cola ordenada, tomamos estas dos colas y tomamos el primero de cada una. En este caso 1 y -1. Tomamos el menor y lo colocamos en una cola nueva y desacolamos de donde lo tomamos. Luego volvemos a comparar los priemros de cada cola. Si una queda vacía, se acolan todos los elementos sobrantes. Entonces:

$$queue_1 = \xrightarrow{[-1] [1] [3]}$$

Ahora vamos con la otra primera mitad:

$$queue_2 = \xrightarrow{[9] [8]}$$

Como tiene dos elementos no ordenados, los invertimos:

$$queue_2 = \xrightarrow{[8] [9]}$$

Ahora, tomamos $queue_1$ y $queue_2$ y vamos tomando de forma intercalada los elementos para que la nueva cola quede ordenada:

$$queue_1 = \xrightarrow{[-1] [1] [3] [8] [9]}$$

12.6.2. Código

```

4362 public static IQueue revert(IQueue queue) {
4363     if(queue.isEmpty()) { // 0 elements
4364         return queue;
4365     }
4366
4367     int first = queue.getFirst();
4368     queue.remove();
4369     if(queue.isEmpty()) { // 1 element
4370         queue.add(first);
4371         return queue;
4372     }
4373
4374     int second = queue.getFirst();
4375     queue.remove();
4376     if(queue.isEmpty()) { // 2 elements
4377         queue.add(Math.min(first, second));
4378         queue.add(Math.max(first, second));
4379         return queue;
4380     }
4381
4382     int count = 0;
4383     IQueue aux = new Queue();
4384     while(!queue.isEmpty()) {
4385         aux.add(queue.getFirst());
4386         queue.remove();
4387         count++;
4388     }
4389
4390     int left = count / 2;
4391     IQueue queueLeft = new Queue();
4392     for(int i = 0; i < left; i++) {
4393         queueLeft.add(aux.getFirst());
4394         aux.remove();
4395     }

```

```

4396     IQueue queueRight = new Queue();
4397     while(!aux.isEmpty()) {
4398         queueRight.add(aux.getFirst());
4399         aux.remove();
4400     }
4401
4402     queueLeft = revert(queueLeft);
4403     queueRight = revert(queueRight);
4404     while(!queueLeft.isEmpty() && !queueRight.isEmpty()) {
4405         if(queueLeft.getFirst() < queueRight.getFirst()) {
4406             queue.add(queueLeft.getFirst());
4407             queueLeft.remove();
4408         } else {
4409             queue.add(queueRight.getFirst());
4410             queueRight.remove();
4411         }
4412     }
4413
4414     while(!queueLeft.isEmpty()) {
4415         queue.add(queueLeft.getFirst());
4416         queueLeft.remove();
4417     }
4418
4419     while(!queueRight.isEmpty()) {
4420         queue.add(queueRight.getFirst());
4421         queueRight.remove();
4422     }
4423
4424     return queue;
4425 }
```

12.7. Cola con prioridad

12.7.1. Ordenar elementos de una Cola

Las colas con prioridad se ordenan por prioridad. Es decir, independientemente de lo que coloquemos como valor, las prioridades se ordenarán. Podemos usar esto a nuestro favor, colocando como clave algo que queremos ordenar y como valor cualquier cosa.

Vamos a ordenar una cola siguiendo la estrategia: Tomamos cada elemento n y lo mapeamos una dupla de prioridad valor de la forma n, n . Colocamos estos valores uno a uno en la cola con prioridad, y luego, volvamos solo las prioridades (o solo los valores) en la cola original nuevamente. De esta forma, la cola original queda ordenada.

12.7.2. Código

```

4426 public static void sort(IQueue queue) {
4427     IPriorityQueue priorityQueue = new PriorityQueue();
4428     while(!queue.isEmpty()) {
4429         priorityQueue.add(queue.getFirst(), queue.getFirst());
4430         queue.remove();
4431     }
4432
4433     while(!priorityQueue.isEmpty()) {
4434         queue.add(priorityQueue.getPriority());
4435         priorityQueue.remove();
4436     }
}
```

4437 }

12.8. Conjuntos

12.8.1. Agenda

Supongamos que tenemos que armar un evento donde tenemos un conjunto A de invitados. Luego, de un momento a otro, decidimos que la lista de invitados se actualizará al conjunto B . Consideraremos usar conjuntos porque los invitados no tienen prioridad sobre otros, no es necesario un orden, y no se invita más de una vez a un invitado (no hay repetidos).

Entonces enviaremos actualizaciones por email a cada usuario. Es importante notar que, los usuarios serán clasificados como nuevos, cancelados, y neutros. A los neutros podemos informarles de la actualización o no, mientras que a los nuevos y a los cancelados debería ser obligatorio.

Si realizamos la diferencia de conjuntos $A - B$, tendremos elementos de A que no están en B . Esto quiere decir que los invitados que están en A pero no en B son los cancelados.

Si realizamos la diferencia de conjuntos $B - A$, ahora sucederá con B y estos invitados son los nuevos.

Ahora, los invitados comunes serán los que se encuentren en la intersección de ambos conjuntos.

Esto hace el código más expresivo y sencillo de leer, pero no significa que sea lo más óptimo.

12.8.2. Booleanos

Vamos a modificar la implementación de Conjunto para que internamente utilice una pila de booleanos. Esto podría ser un arreglo, una pila o cualquier estructura que queremos donde se pueda asociar un valor a un índice (otro conjunto no podríamos usar porque los conjuntos no tienen orden).

Para este problema, vamos a agregarle la complejidad de que además el conjunto solo admite números enteros entre 100 y 160.

Estrategia

Dado que no podemos almacenar los valores en la estructura interna, pero sí podemos almacenar booleanos. Entonces, una primera solución que tenemos es que consideremos a la estructura interna como variable, y fijar que cada booleano representa un bit del elemento a guardar. Entonces, si un entero ocupa 32 bits, necesitamos un total de 32 booleanos para cada entero que vayamos a guardar.

Otra posible solución, es fijar que cada índice representa un elemento y el booleano asignado a ese índice indica la presencia de ese elemento. Por ejemplo, para el arreglo [0, 1, 2], el arreglo [false, true, false] nos indica que el 1 está presente dentro de la estructura. De la misma forma [true, false, true] nos indica que el 0 y el 2 están presentes. Si planteamos nuestro caso, con un arreglo entre el 100 y el 160 tendríamos 61 booleanos. Es un tamaño fijo, mayor que los 32 de la estrategia anterior. Sin embargo, si utilizamos la estrategia anterior, con tener 2 valores o más, ya ocupamos más espacio que en este caso. No solo eso, sino que recuperar los booleanos y transformarlos a bits, y luego a decimal, es mucho más complejo que asumir que los índices tienen una correlación con los valores a guardar.

Vamos a avanzar con este enfoque, siendo una pila en lugar de un arreglo, y tomaremos que el índice 0 se corresponde con el número 100, el índice 1 con el 101, y así hasta el 160. Un conjunto vacío se corresponde con un arreglo lleno de false, por lo que vamos a inicializar el arreglo originalmente así.

Implementación

Vamos a crear una Pila de Booleanos.

```
4438 public class BooleanStack {
4439
4440     private boolean[] array;
4441     private int count;
4442
4443     public BooleanStack() {
```

```

4444     this.array = new boolean[10000];
4445     this.count = 0;
4446 }
4447
4448 public void add(boolean p) {
4449     this.array[this.count] = p;
4450     this.count++;
4451 }
4452
4453 public void remove() {
4454     if(this.count == 0) {
4455         throw new RuntimeException("Empty stack");
4456     }
4457     this.count--;
4458 }
4459
4460 public boolean isEmpty() {
4461     return this.count == 0;
4462 }
4463
4464 public boolean getTop() {
4465     if(this.count == 0) {
4466         throw new RuntimeException("Empty stack");
4467     }
4468     return this.array[this.count - 1];
4469 }
4470 }
```

Si bien el problema es para números entre 100 y 160 podemos generalizarlo para números entre un mínimo y un máximo. Ahora, vamos a crear una estructura de Conjunto basada en booleanos que utilice este rango y la pila.

```

4471 public class BooleanSet implements ISet {
4472
4473     private final BooleanStack stack;
4474     int min;
4475     int max;
4476     private int count;
4477
4478     public BooleanSet(int min, int max) {
4479         this.stack = new BooleanStack();
4480
4481         // Considero el tope como la posición 0
4482         for(int i = 0; i < max - min + 1; i++) {
4483             this.stack.add(false);
4484         }
4485
4486         this.min = min;
4487         this.max = max;
4488         this.count = 0;
4489     }
4490
4491     @Override
4492     public void add(int a) {
4493         int index = a - min;
4494         this.set(index, true);
4495         this.count++;
4496     }
4497 }
```

```
4498     private void set(int index, boolean p) {
4499         BooleanStack aux = new BooleanStack();
4500         for(int i = 0; i < index; i++) {
4501             aux.add(this.stack.getTop());
4502             this.stack.remove();
4503         }
4504
4505         // El tope es el valor asociado a index
4506         this.stack.remove();
4507         this.stack.add(p);
4508
4509         // Recompongo la pila
4510         while(!aux.isEmpty()) {
4511             this.stack.add(aux.getTop());
4512             aux.remove();
4513         }
4514     }
4515
4516     @Override
4517     public void remove(int a) {
4518         int index = a - min;
4519         this.set(index, false);
4520         this.count++;
4521     }
4522
4523     @Override
4524     public boolean isEmpty() {
4525         return this.count == 0;
4526     }
4527
4528     @Override
4529     public int choose() {
4530         if (this.count == 0) {
4531             System.out.println("No se puede elegir un elemento del conjunto vacío");
4532             return -1;
4533         }
4534         int randomIndex = (new Random()).nextInt(this.max - this.min + 1);
4535         if(this.get(randomIndex)) {
4536             return randomIndex + this.min;
4537         }
4538         return this.choose();
4539     }
4540
4541     private boolean get(int index) {
4542         BooleanStack aux = new BooleanStack();
4543         for(int i = 0; i < index; i++) {
4544             aux.add(this.stack.getTop());
4545             this.stack.remove();
4546         }
4547
4548         // El tope es el valor asociado a index
4549         boolean result = this.stack.getTop();
4550
4551         // Recompongo la pila
4552         while(!aux.isEmpty()) {
4553             this.stack.add(aux.getTop());
4554             aux.remove();
4555         }
4556 }
```

```

4557         return result;
4558     }
4559 }
```

Mientras tengamos una serie de elementos (en este caso la serie empieza en 100 y sigue paso a paso incrementándose en 1) finita, entonces podremos usar esto de forma similar. Por ejemplo, para números pares entre 100 y 160 podríamos considerar muchos menos elementos, y desarrollar una correlación entre el índice y el número tal que $index * 2 + min$ nos da el valor original.

12.9. Diccionario Simple

12.9.1. Calcular la intersección de dos diccionarios

Debemos tomar aquellos elementos que posean la misma clave y valor de ambos diccionarios. Para eso, podemos utilizar en un primer paso ambos conjuntos de claves y calcular la intersección. Pero esto no es suficiente, debemos además sobre cada clave verificar si en ambos diccionarios el valor es el mismo. Para poder ejecutar este algoritmo, necesitamos primero tener un método que calcule las intersecciones de dos conjuntos

12.9.2. Código

```

4560 public static ISet intersection(ISet set, ISet set2) {
4561     ISet intersection = new Set();
4562     while(!set.isEmpty()) {
4563         int element = set.choose();
4564         if(in(element, set2)) {
4565             intersection.add(element);
4566         }
4567         set.remove(element);
4568     }
4569     return intersection;
4570 }
4571
4572 private static boolean in(int a, ISet set) {
4573     boolean exists = false;
4574     ISet aux = new Set();
4575     while(!set.isEmpty()) {
4576         int element = set.choose();
4577         if(element == a) {
4578             exists = true;
4579             break;
4580         }
4581         aux.add(element);
4582         set.remove(element);
4583     }
4584     while(!aux.isEmpty()) {
4585         int element = aux.choose();
4586         set.add(element);
4587         aux.remove(element);
4588     }
4589     return exists;
4590 }
4591
4592 public static IDictionary intersection(IDictionary dictionary, IDictionary dictionary2)
4593 ) {
4594     ISet keys = dictionary.getKeys();
4595     ISet keys2 = dictionary2.getKeys();
```

```

4595     ISet commonKeys = intersection(keys, keys2);
4596
4597     IDictionary intersection = new Dictionary();
4598     while(!commonKeys.isEmpty()) {
4599         int key = commonKeys.choose();
4600         if(dictionary.getValue(key) == dictionary2.getValue(key)) {
4601             intersection.add(key, dictionary.getValue(key));
4602         }
4603         commonKeys.remove(key);
4604     }
4605     return intersection;
4606 }
```

Es importante destacar que podemos destruir los conjuntos sin utilizar copias porque el método que devuelve las claves de un diccionario siempre instancia un nuevo conjunto.

12.10. Diccionario Múltiple

12.10.1. Búsqueda sobre valores

Dado un diccionario multiple vamos a encontrar el elemento para que mas se repite entre sus valores.

Estrategia

Vamos usar un diccionario simple que tenga como clave cada valor del diccionario múltiple, y que tenga como valor la cantidad de apariciones de este valor. Además, como solo nos interesan los valores pares, vamos a agregarlo al diccionario simple solo bajo esa condición. Hacer este filtrado antes o después da igual en términos computacionales pero no en cuestiones de memoria, dado que el diccionario simple que estamos creando en memoria es igual o menor en peso que el que obtendríamos sin previamente filtrar.

Implementación

```

4607 public static int evenValueThatIsRepeatedTheMost (IMultipleDictionary
4608     multipleDictionary) {
4609     ISet keys = multipleDictionary.getKeys();
4610     IDictionary dictionary = new Dictionary();
4611     while(!keys.isEmpty()) {
4612         int key = keys.choose();
4613         ISet values = multipleDictionary.getValues(key);
4614         while(!values.isEmpty()) {
4615             int value = values.choose();
4616             if(value % 2 == 0) {
4617                 if(dictionary.getValue(value) == -1) { // No existe
4618                     dictionary.add(value, 1);
4619                 } else {
4620                     dictionary.add(value, dictionary.getValue(value) + 1);
4621                 }
4622             }
4623             values.remove(value);
4624         }
4625         keys.remove(key);
4626     }
4627
4628     ISet numbers = dictionary.getKeys();
4629     int candidate = -1;
4630     int total = Integer.MIN_VALUE;
4631     while(!numbers.isEmpty()) {
```

```

4631     int number = numbers.choose();
4632     if(dictionary.getValue(number) > total) {
4633         candidate = number;
4634         total = dictionary.getValue(number);
4635     }
4636     numbers.remove(number);
4637 }
4638
4639     return candidate;
4640 }
```

12.10.2. Valor común a cada clave

Tomando un diccionario múltiple, vamos a buscar un valor repetido asociado a cada clave. Analizaremos si es posible construir un algoritmo que lo encuentre con complejidad cuadrática.

Análisis

Una posible estrategia es buscar el conjunto intersección entre todos los conjuntos de valores. Esto nos permite obtener un conjunto que tendrá todos los valores en común asociados a cada clave. Si el conjunto está vacío, entonces este valor no existe. Esta es una primera solución, que en el fondo, es lo que estamos necesitando. Buscar la intersección entre todos implica una complejidad lineal, y formar el conjunto de cada diccionario también es complejidad lineal ejecutándose en cada iteración. De esta forma, este algoritmo tendrá una complejidad cuadrática.

Una posible mejora, es iterar sobre los elementos del primer conjunto de valores que obtengamos, dado que si existe una solución, tiene que existir dentro de cualquier conjunto que tomemos. Cada valor será un candidato, y debemos revisar cada conjunto para saber si este valor está presente o no. También podemos acortar este ciclo asumiendo que si en un conjunto no existe este valor, podemos devolver `false` en nuestra función y será suficiente.

Mientras tengamos que recorrer una cantidad variable de estructuras no podremos reducir la complejidad. Entonces, podrá tener una complejidad lineal si podemos hacer con una complejidad lineal o menor, un traspaso de todos los valores a una única estructura. Y esto no es posible, porque para hacerlo tenemos que iterar sobre las estructuras para leer sus elementos. No es posible desde este enfoque cambiar la complejidad computacional.

Sin embargo, una forma en que quizás podamos lograrlo es hacerlo desde dentro de la implementación del TDA Diccionario Múltiple, y para esto debemos modificar la estructura o agregar un método que nos permita esto.

12.11. Árbol binario

12.11.1. Mostrar elementos no repetidos

La estrategia a implementar es la siguiente: Tomamos los elementos del conjunto por única vez en un conjunto, luego, recorremos el árbol y eliminamos del conjunto los elementos que están en el árbol; en el caso de intentar eliminar un elemento y no se encuentre en el conjunto, quiere decir que ese elemento está repetido y lo guardamos en una estructura. Si por ejemplo, tenemos un árbol que tiene al 5 unas tres veces, la primera se elimina del conjunto, la segunda se agrega a la estructura y la tercera también. Es decir, agregariamos dos veces a la estructura final este elemento. Para solucionar esto, utilizamos para el resultado una estructura que no tenga repetidos (en este caso conjuntos porque es lo que manejamos en la cursada).

```

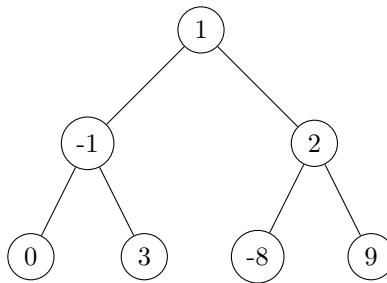
4641 public static ISet getNoRepeatedElements(IBinaryTree binaryTree) {
4642     ISet elements = new Set();
4643     fillElements(binaryTree, elements);
4644     ISet result = new Set();
4645     fillNoRepeatedElements(binaryTree, elements, result);
```

```

4646     return result;
4647 }
4648
4649 private static void fillElements(IBinaryTree binaryTree, ISet set) {
4650     if(binaryTree == null || binaryTree.isEmpty()) {
4651         return;
4652     }
4653     fillElements(binaryTree.getLeft(), set);
4654     fillElements(binaryTree.getRight(), set);
4655     set.add(binaryTree.getValue());
4656 }
4657
4658 private static void fillNoRepeatedElements(IBinaryTree binaryTree, ISet candidates,
4659     ISet result) {
4660     if (binaryTree == null || binaryTree.isEmpty()) {
4661         return;
4662     }
4663     fillNoRepeatedElements(binaryTree.getLeft(), candidates, result);
4664     fillNoRepeatedElements(binaryTree.getRight(), candidates, result);
4665
4666     if (in(binaryTree.getValue(), candidates)) {
4667         candidates.remove(binaryTree.getValue());
4668     }
4669
4670     result.add(binaryTree.getValue());
4671 }
```

12.11.2. Decidir si pudo haber sido generado con ABB

El siguiente es un árbol binario, donde, si tomamos cada nodo, su hijo izquierdo es tiene raíz menor a la suya y su hijo derecho tiene raíz mayor a la suya.



Sin embargo, esto no pudo haber sido generado por un árbol binario de búsqueda, dado que, por ejemplo, el 3 debería haber caído del lado derecho.

Entonces, la regla general para saber si un árbol cumple con la estructura de haber sido formado con un ABB es: ningún nodo del lado izquierdo puede ser mayor a la raíz y ningún nodo del lado derecho puede ser menor a la raíz. A su vez, sus hijos deben también cumplir con esto de forma recursiva.

El código de la solución es el siguiente:

```

4672 private static boolean existsLT(BinaryTree binaryTree, int value) {
4673     if (binaryTree == null || binaryTree.isEmpty()) {
4674         return false;
4675     }
4676     if (binaryTree.getValue() < value) {
4677         return true;
4678     }
```

```

4679     return existsLT(binaryTree.getLeft(), value) || existsLT(binaryTree.getRight(),
4680                     value);
4681 }
4682 private static boolean existsGT(BinaryTree binaryTree, int value) {
4683     if (binaryTree == null || binaryTree.isEmpty()) {
4684         return false;
4685     }
4686     if (binaryTree.getValue() > value) {
4687         return true;
4688     }
4689     return existsGT(binaryTree.getLeft(), value) || existsGT(binaryTree.getRight(),
4690                     value);
4691 }
4692 public static boolean isSBT(BinaryTree binaryTree) {
4693     if (binaryTree == null || binaryTree.isEmpty()) {
4694         return true;
4695     }
4696     return !existsGT(binaryTree.getLeft(), binaryTree.getValue()) &&
4697             !existsLT(binaryTree.getRight(), binaryTree.getValue()) &&
4698             isSBT(binaryTree.getLeft()) &&
4699             isSBT(binaryTree.getRight());
4700 }

```

Acá usamos SBT como las siglas de *Search Binary Tree*.

Este algoritmo no es del todo eficiente. Podemos, para cada nodo, establecer un rango que no debe exceder. De esta forma, existe una forma para hacer una recursividad donde se recibe un árbol y un rango, y define si es ABB. Esto recorrería una vez cada nodo teniendo una complejidad lineal (Ojo: ejercicio tipo Final).

12.11.3. Elemento no repetido

Vamos a resolver el siguiente problema: Dado un árbol donde todos sus elementos aparecen 2 veces excepto 1, encontrar el elemento que aparece 1 sola vez.

Una posible solución es utilizar conjuntos como vimos en el ejemplo de elementos no repetidos de un AB. También podríamos almacenar los datos en una estructura y si aparece nuevamente quitarlo, al final del proceso tendríamos un solo elemento en la estructura y es el que buscamos. El problema es que esto implica un algoritmo de búsqueda, por cada dato a agregar. Como las complejidades se multiplican tendríamos una complejidad lineal-logarítmica. Además esto lo logramos aplicando búsqueda binaria, de lo contrario no podríamos, y eso implica guardar los métodos ordenadas: por ejemplo usando un algoritmo de ordenamiento por inserción.

Pero, dado que manejamos números enteros en estos ejemplos, podemos aplicar operadores bitwise. Utilizando la compuerta XOR sobre todos los elementos del árbol nos dará al final el elemento sin repetir. Esto se puede utilizar siempre que busquemos un número que se encuentre n veces con n impar, y el resto de los elementos se encuentren repetidos cualquier cantidad par de veces. Es decir, este ejercicio es un caso particular.

12.11.4. Encontrar el elemento más repetido

La estrategia es volcar los elementos del árbol en un diccionario simple. La clave será el elemento que encontramos y el valor la cantidad de veces. Luego iteraremos sobre el diccionario buscando el elemento que más se repite. Vamos a considerar que el que más se repite es único, así devolvemos un valor, pero podemos modificar el algoritmo para devolver un conjunto con los más repetidos. El código queda:

```

4701 public static int getTheMostRepeatedElement(IBinaryTree binaryTree) {
4702     IDictionary counter = new Dictionary();

```

```

4703     int candidateKey = -1; // Error
4704     int candidateValue = -1; // Error
4705     if(binaryTree.isEmpty()) {
4706         return candidateKey; // Error
4707     }
4708     fill(binaryTree, counter);
4709     ISet keys = counter.getKeys();
4710     while(!keys.isEmpty()) {
4711         int key = keys.choose();
4712         int value = counter.getValue(key);
4713         if(candidateValue < value) {
4714             candidateKey = key;
4715             candidateValue = value;
4716         }
4717         keys.remove(key);
4718     }
4719     return candidateKey;
4720 }
4721
4722 public static void fill(IBinaryTree binaryTree, IDictionary dictionary) {
4723     if(binaryTree == null || binaryTree.isEmpty()) {
4724         return;
4725     }
4726     fill(binaryTree.getLeft(), dictionary);
4727     fill(binaryTree.getRight(), dictionary);
4728     if(in(binaryTree.getValue(), dictionary.getKeys())) {
4729         dictionary.add(binaryTree.getValue(), dictionary.getValue(binaryTree.getValue()
4730             ()) + 1);
4731     } else {
4732         dictionary.add(binaryTree.getValue(), 1);
4733     }
4734 }
```

Si bien esto no es muy eficiente, algo que podemos considerar son las *Hash tables*. En Java tenemos la clase `HashMap<T>` que funciona como tal. Esto permite que el leer o el guardar un elemento en esta tabla tenga un acceso constante, reduciendo así muchísimo la complejidad computacional.

12.11.5. Recorrer un árbol de forma iterativa

Para recorrer un árbol de forma iterativa vamos a usar una cola como *cola de espera*, donde vamos recorriendo la raíz actual y dejamos en espera a los hijos de esa raíz.

```

4734 public static void print(IBinaryTree binaryTree) {
4735     if (binaryTree.isEmpty()) {
4736         return;
4737     }
4738
4739     IQueue<IBinaryTree> queue = new Queue<>();
4740     queue.add(binaryTree);
4741
4742     while (!queue.isEmpty()) {
4743         int size = size(queue);
4744
4745         for (int i = 0; i < size; i++) {
4746             IBinaryTree node = queue.getFirst();
4747             queue.remove();
4748             System.out.print(node.getValue() + " ");
4749
4750             IBinaryTree left = node.getLeft();
```

```

4751     if (left != null) {
4752         queue.add(left);
4753     }
4754
4755     IBinaryTree right = node.getRight();
4756     if (right != null) {
4757         queue.add(right);
4758     }
4759 }
4760
4761 System.out.println();
4762 }
4763 }
4764
4765 public static int size(IQueue<IBinaryTree> queue) {
4766     IQueue<IBinaryTree> aux2 = new Queue<>();
4767     int total = 0;
4768     while(!queue.isEmpty()) {
4769         aux2.add(queue.getFirst());
4770         total++;
4771         queue.remove();
4772     }
4773     while(!aux2.isEmpty()) {
4774         queue.add(aux2.getFirst());
4775         aux2.remove();
4776     }
4777     return total;
4778 }
```

La versión iterativa se usa en BigData para evitar un stack overflow sobre el stack interno que usa el lenguaje para guardar el historial de funciones.

12.11.6. Recorrido por niveles

Se puede modificar el algoritmo anterior para recorrer por niveles, pero la forma recursiva suele ser más expresiva. Acá el código:

```

4779 public static void printByLevel(IBinaryTree binaryTree) {
4780     int height = getHeight(binaryTree);
4781     for (int level = 1; level <= height; level++) {
4782         printLevel(binaryTree, level);
4783     }
4784 }
4785
4786 private static void printLevel(IBinaryTree node, int level) {
4787     if (node == null) {
4788         return;
4789     }
4790     if (level == 1) {
4791         System.out.print(node.getValue() + " ");
4792     } else if (level > 1) {
4793         printLevel(node.getLeft(), level - 1);
4794         printLevel(node.getRight(), level - 1);
4795     }
4796 }
4797
4798 private static int getHeight(IBinaryTree node) {
4799     if (node == null) {
5000         return 0;
```

```

4801     }
4802     int leftHeight = getHeight(node.getLeft());
4803     int rightHeight = getHeight(node.getRight());
4804     return Math.max(leftHeight, rightHeight) + 1;
4805 }
```

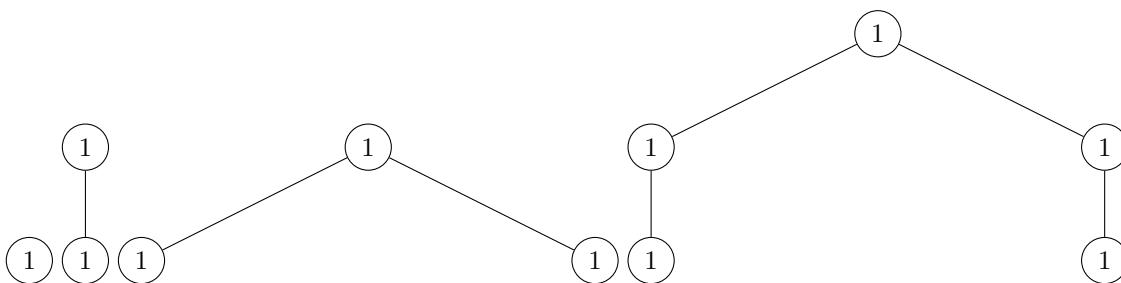
12.11.7. Decidir si todas las hojas están a la misma altura

Un árbol perfecto o un árbol completo fácilmente cumplen con esta condición. También lo cumplen un árbol degenerado o un árbol sesgado. Lo que trataremos de hacer es buscar una forma genérica de saberlo, más allá de los casos particulares.

```

4806 private static boolean validLeafDepth(IBinaryTree binaryTree, int depth) {
4807     if(depth == 0) {
4808         return binaryTree == null || binaryTree.isEmpty();
4809     }
4810
4811     if(depth == 1) {
4812         return binaryTree != null &&
4813             !binaryTree.isEmpty() &&
4814             binaryTree.getLeft() == null &&
4815             binaryTree.getRight() == null;
4816     }
4817
4818     if(binaryTree == null ||
4819         binaryTree.isEmpty() ||
4820         (binaryTree.getLeft() == null && binaryTree.getRight() == null)) {
4821         return false;
4822     }
4823
4824     if(binaryTree.getLeft() != null && binaryTree.getRight() == null) {
4825         return validLeafDepth(binaryTree.getLeft(), depth - 1);
4826     }
4827
4828     if(binaryTree.getRight() != null && binaryTree.getLeft() == null) {
4829         return validLeafDepth(binaryTree.getRight(), depth - 1);
4830     }
4831
4832     return validLeafDepth(binaryTree.getLeft(), depth - 1) && validLeafDepth(
4833         binaryTree.getRight(), depth - 1);
    }
```

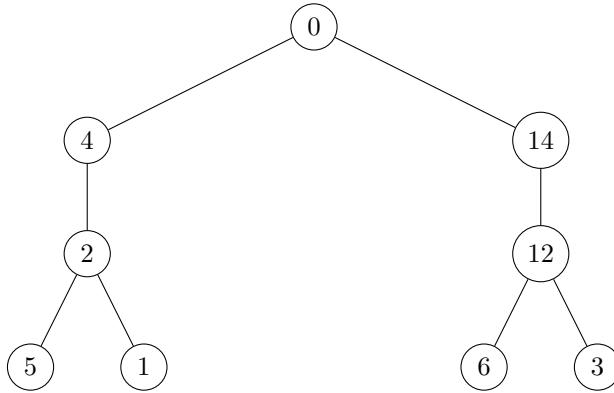
Los siguientes son árboles que cumplen la condición de que sus hojas tienen todas la misma altura:



12.11.8. Imprimir un camino del árbol

Un camino directo entre dos nodos se da mediante la relación padre e hijo. Por ejemplo, en el siguiente gráfico un camino entre el 0 y el 14 es justamente la arista que une a ambos, y lo podemos representar

mediante el arreglo [0, 14].



Por otro lado, un camino entre un nodo consigo mismo es lo mismo que no desplazarse, por lo que por ejemplo, un camino del 2 al 2 se puede representar mediante el arreglo 2.

Sin embargo, existen caminos más complejos, como por ejemplo el camino entre el nodo que tiene un 2 y el 12, que puede ser representado como [2, 4, 0, 14, 12]. Sin embargo este no es el único camino, dado que en uno de los pasos mantenerse en sí mismo podría considerarse válido según el contexto, o bien, y del nodo 2 al 1 y luego volver al 2, podría repetirse varias veces haciendo que arreglos como [2, 1, 2, 1, 2, 4, 0, 14, 12] sea válido también. Es por esto, que vamos a considerar como camino únicamente al más corto, o lo que es equivalente, consideraremos aquel camino sin nodos repetidos.

Por último, tenemos que tener en cuenta el orden en que aparecen los nodos. Por ejemplo, si queremos ir del 12 al 2, el arreglo es el inverso de ir del 2 al 12: [12, 14, 0, 4, 2].

Para llevar este algoritmo adelante vamos a considerar la precondition de que el árbol no tiene elementos repetidos. También vamos a tomar la precondition que tanto el nodo de inicio como de destino existen.

Estrategia 1

Si los nodos no están conectados directamente o por un camino de jerarquía (por ejemplo [2, 5] o 5, 2, 4, 0]), vamos a buscar el nodo padre que contiene a ambos extremos. Por ejemplo, en el camino [5, 2, 1] el 2 es el padre, dado que los elementos que inician y finalizan el camino están en hijos diferentes: el 5 está en hijo izquierdo y el 1 en su hijo derecho. Análogamente el camino inverso también lo cumple. Entonces, podemos proceder a hacer lo siguiente:

- Buscamos el camino entre el nodo padre y el nodo hijo del lado izquierdo. Vamos a llamarlo *A*.
- Buscamos el camino entre el nodo padre y el nodo hijo del lado derecho. Vamos a llamarlo *B*.
- Si queremos ir del lado izquierdo al derecho, vamos a invertir *A*, concatenarlo con el padre y concatenarlo con *B*. En cambio, si queremos ir del lado derecho al lado izquierdo, vamos a invertir *B*, concatenarlo con el padre y luego con *A*.

Por ejemplo, si queremos ir del 5 al 12, entonces *A* es [4, 2, 5] y *B* es [14, 12]. Si invertimos *A* tenemos [5, 2, 4] y concatenando con el padre y luego con *B* tenemos [5, 4, 2, 0, 14, 12] y obtuvimos el camino que buscábamos.

Estrategia 2

Buscamos directamente *A* y *B* tomando como padre a la raíz del árbol y luego concaténamos como en la estrategia 1. Si bien esto da el mismo resultado para el ejemplo anterior, si queremos ir del nodo 5 al 1, esto no es así, dado que nos quedaría [5, 2, 4, 0, 4, 2, 1]. Sin embargo, existe una forma de arreglar esto, dado que si nos fijamos, el nodo 2 es el elemento de izquierda a derecha que primero aparece repetido, y lo mismo de derecha a izquierda, y esto siempre es así porque tenemos una simetría. Deberíamos eliminar estos elementos.

Esta nueva estrategia nos permite una mejora sobre caminos que tienen la misma cantidad de elementos que niveles recorridos. Por ejemplo, el camino [1, 2, 4] tiene todos sus niveles distintos para sus nodos. Con este análisis de simetría, notamos que el camino 1, 2, 4, 0, 4 tiene la particularidad de que uno de sus extremos ya es un elemento repetido. Cuando esto suceda, podemos corregir el arreglo eliminando los valores desde ese extremo hacia el lado opuesto hasta encontrar el elemento repetido inclusive. No solo eso, sino que si queremos encontrar un camino de un nodo a sí mismo esto también funciona: [2, 4, 0, 4, 2] por ejemplo, tomando el lado izquierdo y sabiendo que este elemento está repetido, eliminamos su lado derecho hasta encontrar su repetido obteniendo [2, 2] y luego eliminando este elemento repetido también obtenemos [2].

Implementación

Vamos a implementar la segunda estrategia. Para esto necesitamos generar el camino. Dado que un arreglo es de tamaño fijo, vamos a utilizar una lista. Por otro lado, debido a la recursividad esta lista está invertida respecto a la que queremos, pero podemos rearmar el algoritmo no invirtiendo lo que queremos invertir, e invirtiendo la lista que queremos que quede intacta (invertir una lista es una función que se tiene de inversa a sí misma).

```

4834 private static LinkedList pathFromTheRoot(IBinaryTree binaryTree, int node) {
4835     if (binaryTree == null || binaryTree.isEmpty()) {
4836         return new LinkedList();
4837     }
4838     if (binaryTree.getValue() == node) {
4839         LinkedList result = new LinkedList();
4840         result.add(binaryTree.getValue());
4841         return result;
4842     }
4843
4844     if (binaryTree.getLeft() == null && binaryTree.getRight() == null) {
4845         return new LinkedList();
4846     }
4847
4848     if (binaryTree.getLeft() != null && binaryTree.getRight() == null) {
4849         LinkedList result = pathFromTheRoot(binaryTree.getLeft(), node);
4850         if (!result.isEmpty()) {
4851             result.add(binaryTree.getValue());
4852         }
4853         return result;
4854     }
4855
4856     if (binaryTree.getLeft() == null && binaryTree.getRight() != null) {
4857         LinkedList result = pathFromTheRoot(binaryTree.getRight(), node);
4858         if (!result.isEmpty()) {
4859             result.add(binaryTree.getValue());
4860         }
4861         return result;
4862     }
4863
4864     LinkedList result = pathFromTheRoot(binaryTree.getLeft(), node);
4865     if (result.isEmpty()) {
4866         result = pathFromTheRoot(binaryTree.getRight(), node);
4867     }
4868     if (result.isEmpty()) {
4869         return result;
4870     }
4871
4872     result.add(binaryTree.getValue());
4873     return result;

```

4874 }

Esta función incluye a la raíz, por lo que al momento de usarla lo haremos con los hijos de nuestra raíz original.

Necesitamos una función que nos revierta esta lista:

```
4875 private static LinkedList revert(LinkedList linkedList) {
4876     LinkedList aux = new LinkedList();
4877     for(int i = linkedList.size() - 1; i >= 0; i--) {
4878         aux.add(linkedList.get(i));
4879     }
4880     return aux;
4881 }
```

Ahora que tenemos la lista y una forma de revertirla, debemos hacer la concatenación:

```
4882 private static LinkedList concat(IBinaryTree binaryTree, int a, int b) {
4883     LinkedList listA = pathFromTheRoot(binaryTree.getLeft(), a);
4884     LinkedList listB = pathFromTheRoot(binaryTree.getRight(), b);
4885
4886     boolean leftToRight = true;
4887
4888     if(listA.isEmpty()) {
4889         listA = pathFromTheRoot(binaryTree.getRight(), a);
4890         listB = pathFromTheRoot(binaryTree.getLeft(), b);
4891         leftToRight = false;
4892     }
4893
4894     if(leftToRight) {
4895         listA.add(binaryTree.getValue());
4896         return concat(listA, revert(listB));
4897     }
4898
4899     listB.add(binaryTree.getValue());
4900     return concat(listB, revert(listA));
4901 }
4902
4903 private static LinkedList concat(LinkedList list, LinkedList list2) {
4904     LinkedList linkedList = new LinkedList();
4905     for(int i = 0; i < list.size(); i++) {
4906         linkedList.add(list.get(i));
4907     }
4908     for(int i = 0; i < list2.size(); i++) {
4909         linkedList.add(list2.get(i));
4910     }
4911     return linkedList;
4912 }
```

Por último hay que hacer la corrección de elementos repetidos. Vamos a unir todo en una sola función que nos da la solución:

```
4913 public static void path(IBinaryTree binaryTree, int a, int b) {
4914     LinkedList candidate = concat(binaryTree, a, b);
4915     if(candidate.size() == 1) {
4916         System.out.printf("[%d]", candidate.get(0));
4917         return;
4918     }
4919
4920     if(candidate.get(0) == candidate.get(candidate.size() - 1)) {
4921         System.out.printf("[%d]", candidate.get(0));
```

```
4922     return;
4923 }
4924
4925 if(exists(candidate, candidate.get(0), 1, candidate.size())) {
4926     LinkedList result = new LinkedList();
4927     result.add(candidate.get(0));
4928     for(int i = 1; i < candidate.size(); i++) {
4929         if(candidate.get(i) == candidate.get(0)) {
4930             break;
4931         }
4932         result.add(candidate.get(i));
4933     }
4934     printPath(result);
4935     return;
4936 }
4937
4938 if(exists(candidate, candidate.get(candidate.size() - 1), 0, candidate.size() - 1)
4939 ) {
4940     LinkedList result = new LinkedList();
4941     int index = firstIndex(candidate, candidate.size() - 1);
4942     for(int i = index; i < candidate.size(); i++) {
4943         result.add(candidate.get(i));
4944     }
4945     printPath(result);
4946     return;
4947 }
4948
4949 for(int i = 0; i < candidate.size(); i++) {
4950     for(int j = candidate.size() - 1; j > i; j--) {
4951         if(candidate.get(i) == candidate.get(j)) {
4952             LinkedList result = new LinkedList();
4953             for(int k = i; k <= j; k++) {
4954                 result.add(candidate.get(k));
4955             }
4956             printPath(result);
4957             return;
4958         }
4959     }
4960 }
4961 printPath(candidate);
4962 }
4963
4964 private static int firstIndex(LinkedList linkedList, int value) {
4965     for(int i = 0; i < linkedList.size(); i++) {
4966         if(linkedList.get(i) == value) {
4967             return i;
4968         }
4969     }
4970     return -1;
4971 }
4972
4973 private static void printPath(LinkedList linkedList) {
4974     System.out.print("[");
4975     if(linkedList.size() == 1) {
4976         System.out.print(linkedList.get(0));
4977     } else {
4978         for(int i = 0; i < linkedList.size() - 1; i++) {
4979             System.out.print(linkedList.get(i));
```

```

4980         System.out.print(",");
4981     }
4982     System.out.print(linkedList.get(linkedList.size() - 1));
4983   }
4984   System.out.print("]");
4985 }
4986
4987 private static boolean exists(LinkedList linkedList, int value, int init, int end) {
4988   for(int i = init; i < end; i++) {
4989     if(linkedList.get(i) == value) {
4990       return true;
4991     }
4992   }
4993   return false;
4994 }
```

Podemos hacer una pequeña prueba como la siguiente:

```

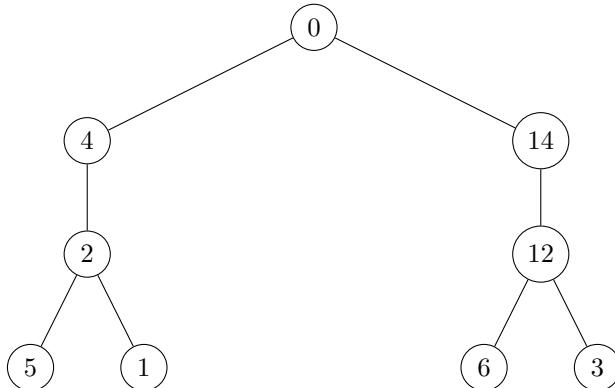
4995 public static void main(String[] args) {
4996   IBinaryTree binaryTree = new BinaryTree();
4997   binaryTree.create(0);
4998   binaryTree.addRight(14);
4999   binaryTree.addLeft(4);
5000   binaryTree.getLeft().addLeft(2);
5001   binaryTree.getRight().addRight(12);
5002   binaryTree.getRight().getRight().addLeft(34);
5003   binaryTree.getRight().getRight().addRight(3);
5004   binaryTree.getLeft().getLeft().addLeft(222);
5005   binaryTree.getLeft().getLeft().addRight(1);
5006
5007   path(binaryTree, 222, 34);
5008 }
```

Esta prueba imprime [222, 2, 4, 0, 14, 12, 34].

12.11.9. Diámetro de un árbol

Definimos el diámetro de un árbol como su camino más largo. Para conocerlo, una primera solución a fuerza bruta es iterar sobre cada camino posible, mapear estos valores a sus tamaños y luego buscar el máximo.

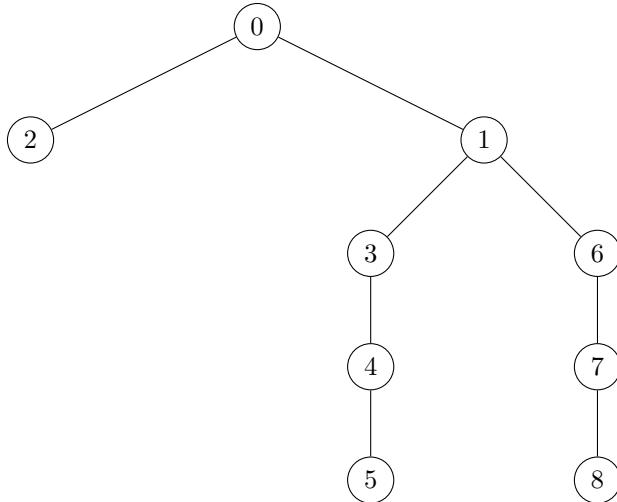
Retomemos el árbol anterior:



Aquí tenemos dos múltiples caminos de tamaño 7, como por ejemplo el camino [1, 2, 4, 0, 14, 12, 6, 3]. Tengamos en cuenta que su inverso también es un camino válido con la misma longitud.

Dado que solo nos interesa su longitud, algo que se nos puede ocurrir es que podemos usar lo que vimos en el problema de imprimir un camino arbitrario: siempre tenemos una raíz tal que tiene su hijo izquierdo con parte del camino y a su hijo derecho con parte del camino, y pensar que la raíz del árbol original es quien buscamos.

Sin embargo, un posible contraejemplo es el siguiente árbol:



Aquí el camino más largo es [5, 4, 3, 1, 6, 7, 8] y no pasa por la raíz 0.

Estrategia

Podemos considerar que los hijos del árbol pueden ser una posible raíz que si cumpla que pertenece al camino más largo. Por lo que, podemos comparar el camino más largo de considerar todo el árbol, solo su hijo izquierdo o solo su hijo derecho. En el caso de que el hijo sea único hay que considerar si vale la pena considerar al padre o no.

Pero ¿analizar esto de forma recursiva no llevaría más iteraciones de fuerza bruta? No necesariamente. Si consideramos el árbol que tiene como raíz incluida en su camino más largo, entonces podemos asumir que la longitud de ese camino es la altura de su hijo izquierdo, más la altura de su hijo derecho, incrementando en 1 para considerar la raíz.

Implementación

```

5009 public static int diameter(IBinaryTree binaryTree) {
5010     if (binaryTree == null || binaryTree.isEmpty()) {
5011         return 0;
5012     }
5013     if (binaryTree.getLeft() == null && binaryTree.getRight() == null) {
5014         return 1;
5015     }
5016     int candidate = height(binaryTree.getLeft()) + 1 + height(binaryTree.getRight());
5017     return Math.max(Math.max(candidate, diameter(binaryTree.getRight())), diameter(
5018         binaryTree.getLeft()));
  
```

Aquí `height` es la función previamente mostrada en este apunte para calcular la altura de un árbol. Podemos probar el caso anterior con el siguiente main:

```

5019 public static void main(String[] args) {
5020     IBinaryTree binaryTree = new BinaryTree();
5021     binaryTree.create(0);
  
```

```

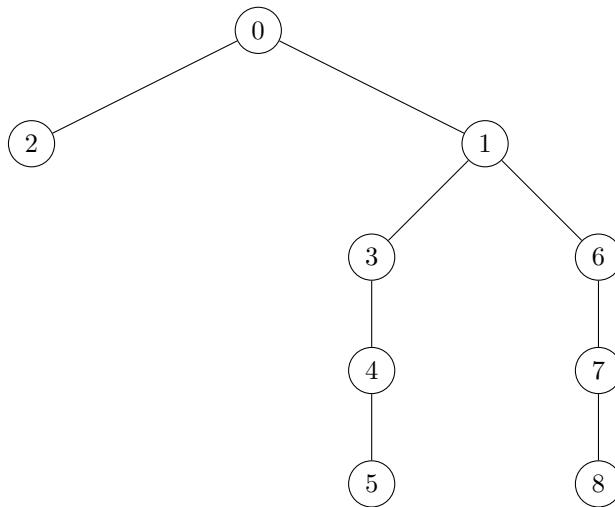
5022     binaryTree.addRight(1);
5023     binaryTree.addLeft(2);
5024     binaryTree.getRight().addLeft(3);
5025     binaryTree.getRight().getLeft().addLeft(4);
5026     binaryTree.getRight().getLeft().getLeft().addRight(5);
5027     binaryTree.getRight().addRight(6);
5028     binaryTree.getRight().getRight().addRight(7);
5029     binaryTree.getRight().getRight().addLeft(8);
5030
5031     System.out.println(diameter(binaryTree));
5032 }
```

12.11.10. Swap

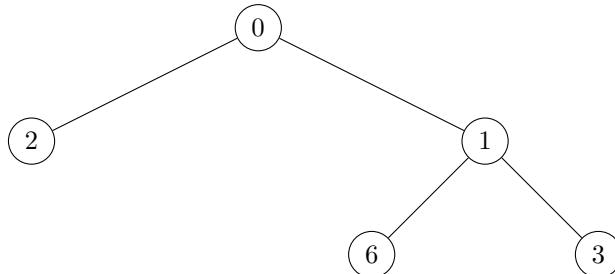
Dado un árbol binario y un número que representa un nodo del árbol, el problema consiste en hacer un Swap entre ese nodo y su hermano.

Tener en cuenta que no es intercambiar los valores de los nodos, sino los nodos. Este problema se podría solucionar sencillo si tuviésemos un setter para poder establecer quien es el hijo izquierdo y quien el derecho por referencia. De esa forma, podemos hacer algo similar a un intercambio de variables pero entre sus hijos.

Sin embargo, solo con las operaciones básicas de nuestro TDA no podemos hacerlo, dado que el método addLeft y el método addRight reemplazan todo el árbol izquierdo y derecho por un único nodo que contiene el valor especificado (ver la implementación en su capítulo correspondiente). Entonces, tenemos la dificultad de que el árbol se va a destruir parcialmente y nuestro trabajo es reconstruirlo. Es decir, supongamos el siguiente árbol:



Si queremos hacer *swap* entre los nodos 3 y 6, entonces luego de hacer un addLeft(3) y un addRight(6) nos quedará el subárbol:



Luego, tenemos que volver a agregar los nodos que faltan. Vamos a hacerlo de la siguiente forma:

