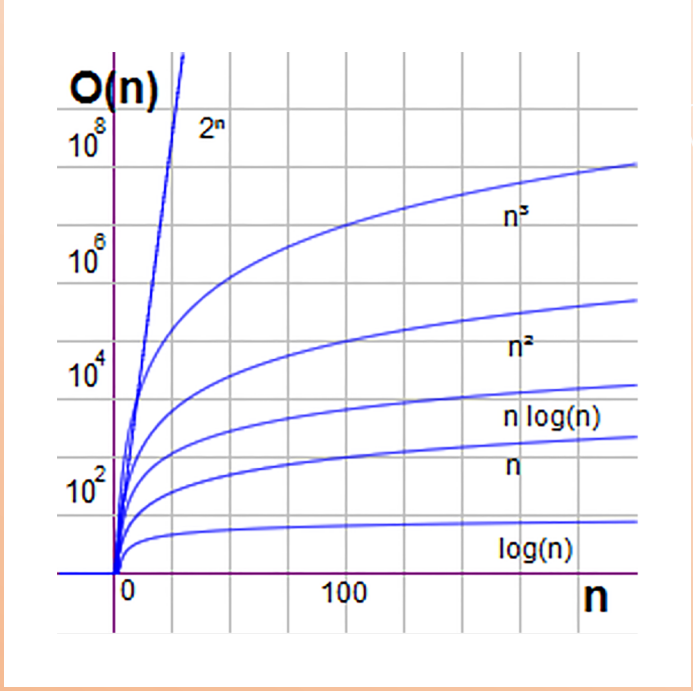


ANÁLISIS Y DISEÑO DE ALGORITMOS Y TIPOS DE DATOS

Miguel Toro Bonilla

Editorial Universidad de Sevilla

Análisis y diseño de algoritmos y tipos de datos

Editorial Universidad de Sevilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática

Análisis y diseño de algoritmos y tipos de datos // Miguel Toro Bonilla

Miguel Toro Bonilla

Análisis y diseño de algoritmos y tipos de datos

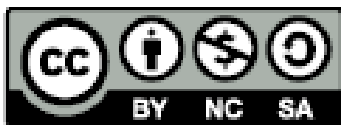
Editorial Universidad de Sevilla

Sevilla 2023

Análisis y diseño de algoritmos y tipos de datos // Miguel Toro Bonilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática

Núm.: 3



Editorial Universidad de Sevilla 2023
c/ Porvenir, 27 - 41013 Sevilla.
Tlfs.: 954 487 447; 954 487 451; Fax: 954 487 443
Correo electrónico: eus4@us.es
Web: https://editorial.us.es

DOI: https://dx.doi.org/10.12795/9788447225064

Maquetación: Miguel Toro

Diseño de cubierta y edición electrónica:
referencias.maquetacion@gmail.com

Índice

Índice de figuras

Notación

- **Rangos**. Son secuencias de números enteros.
  Los representaremos por $[a, b), [a, b]$. El primero no incluye el extremo derecho el segundo sí.
- **Tuplas.** Son agregados inmutables de valores posiblemente de diferentes tipos. Las representaremos por $t = (t_0, t_1, \dots, t_{n-1})$. Si t es una tupla t_i o $t[i]$ representa el i-ésimo elemento y $|t| = n$ el número de elementos.
- **Listas.** Son agregados indexados de elementos. Las representaremos por $ls = [e_0, e_1, \dots, e_{n-1}]$. Donde e_i son los elementos de la lista y $|ls|$ su longitud. La sublista delimitada por dos índices la representamos por $ls[a, b]$ que incluye las casillas de índices a al b sin la última casilla. El elemento i de la lista lo representaremos por $ls[i]$ y por [] la lista vacía.
- **Conjuntos**. Son agregados de elementos sin repetición y sin orden. Los representaremos por es un conjunto representaremos por $ss = \{e_0, e_1, \dots, e_{n-1}\}$, por $|ss|$ su cardinal y por $\{\}$ vacío.
- **Multiconjuntos.** Son agregados de elementos sin orden, pero con posible repetición de elementos. Las representaremos por $ms = e_0 : m_0, e_1 : m_1, \dots, e_{n-1} : m_{n-1}$, donde m_i es un entero positivo o cero que indica el número de veces que se repite e_i. Hacemos $ms[e_i] = m_i$. Una implementación se ofrece en el API.

- **Diccionarios.** Son conjuntos de pares clave-valor dónde las claves no están repetidas. Mediante $m = \{k_0: v_0, k_1: v_1, \dots, k_{n-1}: k_{n-1}\}$ representamos un diccionario, donde $(k_1: v_1)$ son pares clave-valor. Por $m[k_i]$ representamos el valor asociado a la clave k_i, y por $m.keys, m.items$ el conjunto de sus claves y de sus valores respectivamente y por {} un diccionario vacío. En Java este tipo se representa por el tipo *Map<K,V>.*
- **Agregados de datos**. Son grupos de datos organizados de diversas maneras. Pueden reales o virtuales. Reales si necesitan estar en memoria todos los elementos que los componen. Virtuales si solo declaramos sus propiedades y algún mecanismo para obtener sus elementos uno tras otro. Las listas y los conjuntos son agregados reales. Los rangos de números puede ser un agregado virtual. Otro agregado que consideraremos serán los ficheros. Los agregados de datos serán del tipo *D* y los representaremos por d. Los agregados de datos suelen ser iterables que quiere decir que nos ofrecen uno, o varios, flujos de datos para poder recorrer sus elementos.
- **Flujos de datos secuenciales.** Son agregados de datos virtuales que proporcionan uno tras otro sus elementos. Normalmente son vistas de agregados de datos que nos permiten recorrer sus elementos uno tras de otro. Los rangos, las listas, los conjuntos, el conjunto de los pares de un diccionario y sus claves ofrecen vistas en forma de flujos de datos.
- **Secuencias.** Son agregados de datos definidos por un primer elemento, una función que obtiene el siguiente y un predicado que indica hasta donde se extiende la secuencia. Por ahora usaremos secuencia y flujo de datos secuencial como conceptos intercambiables.

Agradecimientos

Este libro está orientado a la enseñanza de la asignatura Análisis y Diseño de Datos y Algoritmos que es la continuación natural de Fundamentos de Programación. El lector interesado puede consultar Fundamentos de Programación: Python y Fundamentos de Programación: Java. Estos conceptos se concretan en el lenguaje de programación Java; en otro volumen se verán en el lenguaje C.

Para abordar el diseño de algoritmos, es necesario tener asimilados los elementos de la programación en algún lenguaje. Para seguir el contenido hace falta conocer el lenguaje Java y sus peculiaridades. Aprenderemos las técnicas para el diseño de algoritmos iterativos, las técnicas de diseño de algoritmos recursivos, su análisis y las transformaciones de un tipo de algoritmo en otros.

Junto al diseño de algoritmos, el diseño de tipos y el uso de tipos diseñados por otros ocuparán un lugar importante en este material. Al final, se incluyen ejemplos propuestos y algunos resueltos de diseños de algoritmos y de diseño de tipos de datos.

Este texto deriva de la experiencia de varios años de enseñanza de la asignatura Análisis y Diseño de Algoritmos en la Universidad de Sevilla. El material procede de versiones anteriores que han sido transformadas hasta alcanzar la forma actual, que le debe mucho a los profesores Carmelo del Valle, Irene Barba, Andrés Jiménez y otros muchos que han

detectado errores y sugerido cambios. A todos ellos les quiero agradecer sus esfuerzos y dedicación. Los defectos son responsabilidad única del autor.

En https://github.com/migueltoro/adda_v3 puede encontrarse el código de los ejemplos. Próximas versiones se encontrarán en https://github.com/migueltoro/*.

Miguel Toro

Sevilla, mayo de 2023

Introducción

En cursos anteriores, estudiamos diferentes tipos de algoritmos para hacer cálculos sobre flujos de datos. Vimos dos estilos de implementar estos algoritmos: el *estilo imperativo* y el *estilo funcional.*

El estilo imperativo construye los algoritmos mediante tipos de datos (básicos y agregados), sentencias de asignación y las sentencias de control: *if, while, for*.

El estilo funcional obtiene del agregado de datos de partida un flujo de datos, un objeto de tipo *Stream<E>*, y a este objeto le aplica diferentes operaciones de filtro, transformación y acumulación.

Los algoritmos que hemos estudiado, ya en forma imperativa o funcional, eran *algoritmos iterativos*. Junto con ellos existen *algoritmos recursivos*. Un algoritmo es recursivo si se llama a sí mismo. Es decir, en un algoritmo recursivo aparece una llamada a sí mismo dentro de su cuerpo.

Los algoritmos recursivos son más generales que los iterativos. Como iremos viendo, algunos algoritmos recursivos tienen un equivalente iterativo directo. Otros no.

Los algoritmos iterativos en su forma imperativa tienen un equivalente en su forma funcional y viceversa.

Veamos un ejemplo para sumar una lista en sus diferentes versiones:

```
Double sumLista(List<Double> list) {
	Integer i = 0;
	Double b = 0.;
	while(i < list.size()){
	Double e = list.get(i);
		i = i+1;
		b = b + e;
	}
	return b;
}
```

Como vemos, el anterior es un algoritmo iterativo, no se llama a *sumLista* en su cuerpo. Está escrito en notación imperativa porque usa sentencias de asignación, *while* y podría usar *if* y *for*.

Veamos el mismo algoritmo en notación funcional.

```
Double sumLista(List<Double> ls) {
	return ls.stream().mapToDouble(x->x).sum();
}
```

Es iterativo porque no llama a *sumLista* en su cuerpo, pero está escrito en notación funcional: obtiene un flujo de datos, un *stream*, del agregado, la lista, y aplicando sucesivos métodos obtiene el resultado.

Veamos su versión recursiva.

```
Double sumLista(List<Double> list) {
	return sumLista(list,0,0.);
}
Double sumLista(List<Double> list, Integer i, Double b) {
	if(i<list.size()){
		Double e = list.get(i);
		b = sumLista(list,i+1,b+e);
	}
	return b;
}
```

La versión anterior se compone de dos funciones: la primera llama a la segunda que tiene algunos parámetros adicionales; la segunda función se llama a sí misma dentro de su cuerpo. Es una función recursiva.

Vamos a aprender a transformar algoritmos de un estilo a otro cuando eso sea posible.

En muchos casos los algoritmos, principalmente en los casos recursivos, los escribiremos en forma de esquemas como:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0 \end{cases}$$

Que es equivalente a código:

```
Integer f(Integer n) {
	Integer r;
	if(n==0){
		r = 1;
	} else {
		r = n*f(n-1);
	}
	return r;
}
```

Como podemos ver, un esquema de un algoritmo es un boceto de este. Algunos detalles, el tipo de las variables, por ejemplo, quedan implícitos pero a partir del esquema es fácil escribir el algoritmo en un lenguaje de programación dado. Aquí lo haremos en Java.

Los algoritmos recursivos se pueden clasificar en *simples* y *múltiples*. Los primeros tienen una sola llamada recursiva dentro de su cuerpo; los segundos, varias llamadas recursivas.

A su vez, los algoritmos recursivos simples se clasifican en *recursivos finales* y *recursivos no finales*. Un algoritmo recursivo simple es recursivo final si el resultado es igual al de la llamada recursiva sin ninguna operación posterior. En un algoritmo recursivo simple no final el resultado se obtiene aplicando algunas operaciones al valor devuelto por la llamada recursiva. Veamos algunos ejemplos.

Algoritmo recursivo final:

$$mcd(a,b) = \begin{cases} a, & b = 0 \\ mcd(b, a\%b), & a, b > 0 \end{cases}$$

Algoritmo recursivo simple no final:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0 \end{cases}$$

Algoritmo recursivo múltiple:

$$fib(n) = \begin{cases} n, & 0 \leq n \leq 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

En muchos casos, por simplicidad y generalidad del esquema algorítmico, es conveniente compactar varias variables en una tupla y una secuencia de asignaciones en una asignación de tuplas. Dado el ejemplo anterior:

```
Double sumLista(List<Double> list) {
       Integer i = 0;
       Double b = 0.;
       while(i < list.size()){
          Double e = list.get(i);
          i = i+1;
          b = b + e;
       }
       return b;
}
```

Podemos escribirlo como

```
Double sumLista(List<Double> list) {
       (Integer,Double) (i,b) = (0,0.);
       while(i < list.size()){
          (i,b) = (i+1,b+list.get(e));
       }
       return b;
}
```

Una secuencia de sentencias de asignación se le llama un *bloque básico*. Más adelante aprenderemos a transformar bloques básicos en asignaciones de tuplas y viceversa.

Un primer detalle que queremos enfatizar es que *los algoritmos iterativos,* imperativos o funcionales*, y los algoritmos recursivos simples finales* son equivalentes entre sí en el sentido de que dado un algoritmo en una de las formas podemos encontrar sistemáticamente el algoritmo equivalente en las otras dos.

Un segundo detalle importante es que en cualquier algoritmo podemos sustituir los bloques básicos por asignaciones de tuplas lo que nos permitirá disponer de un esquema más compacto del algoritmo. Aprenderemos más adelante como hacerlo

Vemos, por tanto, que las tuplas desempeñan un papel muy importante en el diseño de algoritmos. Algunos lenguajes como Python ya proporcionan el tipo tupla. Java no lo proporciona directamente, pero podemos simularlas mediante *records*, concepto que ya estudiamos en cursos anteriores.

Así la declaración y asignación de tuplas siguiente:

```
(Integer,Double) (i,b) = (0,0.);
```

Podemos simularlo por

```
record T(Integer i,Double b){}
T t = new T(0,0.);
```

El acceso a las propiedades de la tupla se hace mediante los métodos *t.i(), t.b().* Normalmente la declaración del tipo record se complementa de un método de factoría y entonces la asignación de tuplas queda en la forma:

```
record T(Integer i,Double b){
	public static T of(Integer i,Double b) {
		return new T(i,b);
	}
}
T t = T.of(0,0.);
```

Algoritmos iterativos

Como vamos a ver, un algoritmo iterativo y los recursivos simples se pueden pensar como *combinaciones de secuencias y acumuladores*. Veamos estos conceptos en primer lugar y su relación con los algoritmos.

En esta sección vamos a ver los conceptos relacionados con los algoritmos iterativos ly los recursivos simples asociados, su forma general y diversos casos particulares, los mecanismos para transformarlos y la forma de verificarlos y diseñarlos. Veremos la forma clásica, imperativa, de estos algoritmos, en su versión iterativa o recursiva, y su equivalente funcional que es más adecuada en los lenguajes más modernos.

Partamos del ejemplo anterior:

```
Double sumLista(List<Double> list) {
       Integer i = 0;
       Double b = 0.;
       while(i < list.size()){
          Double e = list.get(i);
          i = i+1;
          b = b + e;
       }
       return b;
}
```

Su versión con tuplas

```
Double sumLista(List<Double> list) {
       (Integer,Double) (i,b) = (0,0.);
       while(i < list.size()){
          (i,b) = (i+1,b+list.get(i));
       }
       return b;
}
```

O en su versión recursiva final.

```
Double sumLista(List<Double> list) {
       return sumLista(0,0.,list);
}
Double sumLista(Integer i, Double b, List<Double> list) {
       if(i<list.size()){
              b = sumLista(i+1,b+list.get(i),list);
       }
       return b;
}
```

En la versión recursiva solo hay que tener en cuenta, por ahora, que un algoritmo recursivo es aquel (como *sumLista(list,i,b))* que hace una llamada a si mismo dentro de su cuerpo).

Podemos observar la relación estrecha entre el algoritmo imperativo en su formato de tuplas y la versión recursiva final. Vemos que la función recursiva tiene todos los parámetros de la tupla más los parámetros de entrada de la función. La llamada a la función se hace con los parámetros de la parte derecha de la asignación de tuplas. El *while* se ha sustituido por *if*. El *while*, en el algoritmo iterativo, representa iteración que se sustituye por el *if* de la llamada recursiva que indica que se hace una llamada a sí misma mientras que se cumpla la misma condición que para el *while*.

O su versión funcional

```
Double sumListaFunc(List<Double> ls) {
       return ls.stream()
              .mapToDouble(e->e)
              .sum();
}
```

Para obtener la versión funcional hemos tenido que darnos cuenta de que existe una *secuencia*, los elementos de la lista, que podemos obtener mediante el método *stream()* y un *acumulador*, en este caso la variable *b* que se inicializa a *cero* y va acumulando, sumando en este caso, los sucesivos elementos de la secuencia.

Las tres versiones (iterativa, recursiva simple final y funcional) son algoritmos equivalentes en el sentido que producen el mismo resultado cuando se le proporciona la misma entrada.

Para poder transformar de una versión a otra debemos encontrar sus elementos comunes: *el acumulador y la secuencia*.

El *acumulador* viene definido por una variable *b*, que puede ser una tupla de tipo *B*, con un *valor inicial* b0, y una *función de acumulación c(b,e)*. En este caso *B* es Integer, el valor inicial es *0* y la función de acumulación *(b,e)->b+e.* El tipo *B* puede ser mutable o inmutable. Si el acumulador es mutable la función de acumulación toma forma de método *b.c(e).*

El acumulador acumula los sucesivos valores de la secuencia a partir del valor inicial dado. Cada acumulador tiene un nombre y viene definido por el tipo de acumulador, *B,* su valor inicial*, b0,* y la función de acumulación *c(b,e).* A partir de la versión funcional podemos deducir que el acumulador es un sumador y sus características son *Integer, 0, (b,e)->b+e*. Esto nos permite completar una parte del algoritmo en su versión imperativa.

Otro elemento es la *secuencia*. En la versión funcional una secuencia se implementa en el lenguaje Java mediante un objeto de tipo *Stream<E>.* Podemos obtener streams, flujos de datos, de los diferentes agregados de datos mediante métodos adecuados. En el caso de las colecciones este método es *stream().*

En la versión imperativa no disponemos de stream para representar secuencias. En ese caso usamos un iterador. Un iterador, en Java, es un

objeto de tipo *Iterator<E>* que nos puede proporcionar uno tras otro los objetos de un agregado. En Java el tipo Iterator viene definido como:

```
interface Iterator<E> {
	boolean hasNext();
	E next();
}
```

Un iterador es un objeto mutable que implementa el tipo *Iterator<E>.* El método *hasNext* nos indica si hay más elementos. El método *next* nos da el siguiente elemento, si hay alguno disponible, y actualiza el estado.

Los agregados de datos, reales o virtuales, que pueden ser recorridos se dice que son iterables. En Java implementan el tipo *Iterable<E>* que es de la forma:

```
interface Iterable<E> {
	Iterator<E> iterator();
}
```

Un iterable es, en definitiva, una factoría de iteradores. Un iterador también puede ser obtenido de un objeto de tipo *Stream<E>* mediante del método *iterator().*

Disponiendo de un iterador podemos escribir las formas imperativas del algoritmo de una manera que nos permitirá poder generalizarlo.

```
Double sumLista(List<Double> list) {
	Iterator<Double> it =list.iterator();
	Double b = 0.;
	while(it.hasNext()){
		Double e = it.next();
		b = b + e;
	}
	return b;
}
```


Java ya aporta una sentencia *for* para recorrer iterables por lo que el algoritmo anterior se puede escribir:

```
Double sumLista(List<Double> list) {
	Double b = 0.;
	for(Integer e: list){
		b = b + e;
	}
	return b;
}
```

La sentencia for, por lo tanto, obtiene un iterador del iterable y de él va obteniendo un elemento tras otro mientras haya elementos.

En este epígrafe aprenderemos a encontrar, dada una de las versiones, las otras dos que son equivalentes. Para poder hacerlo debemos encontrar la secuencia y el acumulador asociados al algoritmo.

La versión recursivo final usando el iterador se compone de dos funciones. Una primera en cuyo cuerpo se obtiene el iterador y se hace una llamada a una segunda función que tiene como parámetros el acumulador y el iterador.

```
Double sumLista(List<Double> list) {
	Iterator<Double> it = list.iterator();
	return sumLista(it,0.,list);
}
```

```
Double sumLista(Iterator<Double> it,Double b,
		List<Double> list) {
	if(it.hasNext()){
		Double e = it.next();
		b = sumLista(list,it,b+e);
	}
	return b;
}
```

Vemos, por lo tanto, que disponemos de tres formas equivalentes de escribir algoritmos iterativos: en forma imperativa, en forma funcional y de manera recursivo final.

En las tres formas existe un acumulador y una secuencia. La secuencia se puede implementar de forma general mediante un objeto de tipo

Iterator<E> y con él podemos diseñar las formas imperativa y recursivo final. Los algoritmos iterativos en su versión imperativa tienen la estructura clásica general:

```
R alg(x) {
	Iterator<E> it = it(x);
	B b = b0;
	while(it.hasNext() && !f(b)){
		E e = it.next();
		b = c(b,e); // b.c(e) si el tipo B es mutable
	}
	return r(b);
}
```

En este esquema, y de ahora en adelante, hemos de tener en cuenta que las funciones que aparecen, aunque no lo hagamos explícito, pueden depender de los parámetros iniciales *x* del algoritmo.

La implementación de iteradores y la posibilidad de hacer explícito su estado lo veremos más adelante.

Su versión recursiva final equivalente consta de dos funciones:

```
R alg(X x){
	B b = alg(x,it(x),b0);
	return r(b);
}
```

En la primera, arriba, se inicializa el acumulador, se obtiene el iterador a partir de los parámetros iniciales y se llama a la segunda función. Esta segunda tiene dos parámetros adicionales: el iterador y el acumulador.

```
B alg(X x,Iterator<E> it, B b) {
	if(it.hasNext() && !f(b)){
		E e = it.next();
		b = c(b,e); // b.c(e) si el tipo B es mutable
		b = alg(x,it,b);
	}
	return b;
}
```

En la versión funcional la secuencia se implementa mediante un objeto de tipo Stream<E>. Su equivalente funcional en Java expresado de una forma muy abstracta:

```
R alg(x) {
       S s = Stream.iterate(e0, e->g(e), e->s(e));
       A a = (b0,(b,e)->c(b,e), b->r(b), b->f(b));
       return s.collect(a);
}
```

En las tres versiones encontramos dos elementos repetidos: una *secuencia* y un *acumulador*. La *secuencia* de elementos de tipo *E* viene definida por la tupla *s = (e0, e->g(e), e->nx(e))*, que de forma compacta representaremos por *s = (e0, g(e), nx(e))*. Es decir, un elemento inicial, un predicado y una función siguiente. Si implementamos la secuencia mediante un iterador aparecen las funciones *it, hasNext y next.*

Un *acumulador*, por ahora un *acumulador secuencial*, viene definido por tres tipos: el tipo *B* de la base, el tipo *R* del resultado y el tipo *E* de los elementos a acumular. Junto a esos tipos un acumulador viene definido por una tupla *a = (()->b0, (b,e)->c(b,e), b->r(b), b->f(b))*, que de forma compacta representaremos por *a = (b0, c(b,e), r(b), f(b))*.

En lo anterior aparecen tipos. Sea *E* el tipo de los elementos de la secuencia, *B* el tipo de la base del acumulador y R el tipo del resultado del algoritmo. Las funciones que definen el acumulador tienen los tipos:

- $b0: () \rightarrow B$
- $c: (b: B, e: E) \rightarrow B$
- $r: (b: B) \rightarrow R$
- $f: (b: B) \rightarrow Boolean$

Que representan, respectivamente, el *valor inicial* b_0 de tipo *B*, la *función de acumulación*, la *función de retorno* y la *función de cortocircuito* del acumulador. Veremos más detalles de cada una de ellas.

Combinando una secuencia con un acumulador obtenemos un algoritmo que lleva a cabo una *operación de acumulación* sobre la secuencia. Decimos que acumulamos la secuencia *s* mediante el acumulador *a*. El tipo

del resultado será *R* el tipo de los resultados de la función *r* del acumulador.

En algunos casos podemos combinar en una tupla el estado del iterador y la base de acumulador. Siendo *Tp* el tipo de esa tupla un algoritmo iterativo tiene el aspecto general siguiente en su manera imperativa:

```
R alg(x) {
       Tp e = e0(x);
       while(g(e)){
          e = nx(e);
       }
       return r(e);
}
```

Es decir *Tp = (it,b).* La versión recursiva final tiene asociadas dos funciones

```
R alg(x) {
       Tp e = alg(e0(x));
       return r(e);
}
```

```
Tp alg(Tp e) {
       if(g(e)){
              e = alg(nx(e));
       }
       return e;
}
```

Y en la versión funcional

```
R alg(x) {
       Stream<Tp> s = Stream.iterate(e0(x),e->g(e),e->nx(e));
       Tp e = s.filter(e->!g(e)).findFirst().get();
       return r(e));
}
```

Veamos el ejemplo resuelto previamente, la suma de los elementos de una lista, con este enfoque en sus variantes imperativa, recursiva final y

funcional. Diseñamos en primer lugar una tupla que incluya el estado del iterador, i, ls, y el acumulador.

```
record Tp(Integer i, Double b, List<Double> ls) {
   public static Tp of(Integer i, List<Double> ls, Double b) {
       return new Tp(i,ls,b);
   }
}
```

La versión imperativa usando la tupla es:

```
Double sum0(List<Double> ls) {
       Tp e = Tp.of(0, 0.,ls);
       while(e.i()<e.ls().size()){
             e = Tp.of(e.i()+1,e.b()+e.ls().get(e.i()),e.ls());
       }
       return e.b();
}
```

Las dos funciones de la versión recursiva:

```
Double sum1(List<Double> ls) {
       Tp e = sum1(Tp.of(0,ls,0.));
       return e.b();
}
```

```
Tp sum1(Tp e) {
       if(e.i()<e.ls().size()){
         e=sum1(Tp.of(e.i()+1,e.b()+e.ls().get(e.i())),e.ls());
       }
       return e;
}
```

La versión funcional

```
Double sum2(List<Double> ls) {
       Tp e0 = Tp.of(0,ls,0.);
       Predicate<Tp> g = e->e.i() < e.ls().size();
       UnaryOperator<Tp> nx = e->Tp.of(e.i()+1,
             ls,e.b()+e.ls().get(e.i()));
       Stream<Tp> s = Stream.iterate(e0,nx);
       Tp e = s.filter(!g).findFirst().get();
       return e.b();
}
```

Podemos, entonces, concluir que un algoritmo iterativo devuelve el primer valor de una secuencia infinita, definida por un primer valor y un operador unario que obtiene el siguiente, que no cumple la guarda (predicado en el *while*) transformado por la función de retorno. Esto lo tendremos en cuenta a la hora de verificar algoritmos iterativos.

En muchos casos no será necesaria la tupla que en la versión imperativa la sustituiremos por tantas variables como campos y en la versión recursiva por parámetros adicionales de la función recursiva.

---

1. *Obtener la versiones imperativa y recursiva final del siguiente algoritmo en su versión funcional*

```
Set<Punto2D> ejemplo(List<Punto2D> ls){
	return ls.stream().collect(Collectors.toSet());
}
```

En el ejemplo anterior podemos ver que la secuencia puede ser implementada por un iterador con estado *(i,ls)* y un acumulador de tipo Set<Punto2D>. El estado del iterador indica que el siguiente elemento que obtendremos será *ls.get(i)* y el siguiente estado del mismo *(i+1,ls).* El acumulador está definido por una variable de tipo Set<Punto2D> inicializada a vacío.

Con estas ideas podemos escribir la versión imperativa como:

```
Set<Punto2D> ejemplo(List<Punto2D> ls){
	Integer i = 0;
	Integer n = ls.size();
	Set<Punto2D> s = new HashSet<>();
	while(i<n) {
		Punto2D p = ls.get(i);
		i= i+1;
		s.add(p);
	}
	return s;
}
```

Hemos incluido la variable n para guardar un cálculo que se puede repetir.

La versión recursivo final se obtiene de la versión imperativa fácilmente.

```
Set<Punto2D> ejemplo(List<Punto2D> ls){
	Integer i = 0;
	Integer n = ls.size();
	Set<Punto2D> b = new HashSet<>();
	s = ejemplo(i,b,ls,n);
	return s;
}
```

Esta función llama a otra que incluye como parámetros la *i* del iterador y la *b* del acumulador.

```
Set<Punto2D> ejemplo(Integer I, Set<Punto2D> b,
		List<Punto2D> ls, Integer n){
	if(i<n) {
		Punto2D p = ls.get(i);
		i= i+1;
		b.add(p);
		b = ejemplo(i,b,ls,n)
	}
	return b;
}
```

La cuestión clave es poder identificar la secuencia y el acumulador.

Secuencias, iterables y streams

Como ya hemos dicho antes las secuencias son flujos secuenciales de elementos que se pueden recorrer uno tras otro. Es decir, secuencialmente. Las secuencias son elementos abstractos que nos sirven para modelar nuestros algoritmos, pero necesitamos tipos concretos que las implementen. Ya hemos visto dos posibilidades un iterador o un objeto de tipo stream. Tienen los siguientes elementos:

- Sus valores son de tipo E con valor inicial e_o.
- Los sucesivos valores que toma el estado se van obteniendo por la aplicación del operador unario $e \rightarrow nx(e)$.
- Un predicado $e \rightarrow g(e)$ que indica que la secuencia se extiende mientras $g(e)$ es verdadero. Si en predicado es de la forma $e \rightarrow true$ estamos definiendo una secuencia infinita.

Algunos ejemplos:

- Secuencia aritmética entre dos valores a y b con paso c: *s = (a, e->e<b, e->e+c)*
- Secuencia infinita de números primos: *(2, e->true, e->siguientePrimo(e))*

La primera conclusión que queremos destacar es:

Un algoritmo iterativo tiene asociada una secuencia finita definida por $s = (e_0,\ e \to g(e),\ e \to nx(e))$.

Es importante destacar que un algoritmo, para que lo sea, deber tener asociada una secuencia finita. Y debe ser finita porque el algoritmo, para que lo sea, tiene que acabar.

También podemos darnos cuenta de que algunas secuencias pueden ser vacías. No tienen ningún elemento. Ni siquiera el primer elemento indicado pertenece a la secuencia. Ejemplo: *s= (10, e-> e < 5, e-> e+1).*

La descripción anterior de una secuencia se define, como hemos visto, de tres elementos: elemento inicial, un predicado y una función que da el elemento siguiente.

Implementación de secuencias

Ya hemos visto dos posibilidades: un iterador o un objeto de tipo stream.

En Java el tipo *Stream<E>* representa un flujo de datos general y, junto con la vista secuencial, ofrece otra vista paralela que veremos más adelante. Lo detalles internos de la implementación de *streams* los veremos más adelante. Por ahora si queremos diseñar un algoritmo en notación funcional usaremos las *streams* que nos proporciona Java.

En la versión imperativa no usamos *streams* para representar secuencias. En ese caso usamos un iterador. Un iterador, en Java, es un objeto de tipo *Iterator<E>* que nos puede proporcionar uno tras otro los objetos de un agregado. En Java el tipo *Iterator* viene definido como:

```
interface Iterator<E> {
       boolean hasNext();
       E next();
}
```

Un iterador es un objeto mutable que implementa el tipo *Iterator<E>.* El método *hasNext* nos indica si hay más elementos. El método *next* nos da el siguiente elemento, si hay alguno disponible, y actualiza el estado.

Los agregados de datos, reales o virtuales, que pueden ser recorridos se dice que son iterables y, en Java implementan el tipo *Iterable<E>* que es de la forma:

```
interface Iterable<E> {
	Iterator<E> iterator();
}
```

Un iterable es, en definitiva, una factoría de iteradores. Podemos poner todo junto e implementar el tipo *ListIterator<E>.*

En primer elegimos un estado para el iterador. Este estado se concreta en los atributos privados de la clase que implementa el iterador. En este caso el estado es *(i,list)* lo que implica que el siguiente elemento devuelto por el iterador es *list.get(i).* Demos implementar el método de factoría iterator().

```
class ListIterator<E> implements Iterator<E>, Iterable<E> {
	public static <E> Iterator<E> of(List<E> list){
		return new ListIterator<E>(0, list);
	}
	private Integer i;
	private List<E> list;
	private ListIterator(Integer i, List<E> list) {
		this.i = i;
		this.list = list;
	}
	@Override
	public Iterator<E> iterator() {
		return new ListIterator<E>(0, list);
	}
	…
}
```

El siguiente paso es implementar los métodos *hasNext()* y *next()*. Con el estado elegido habrá más elementos mientras que *I* se un índice válido de la lista: *i<list.size().* El método *next()* actualizará el estado y devolverá el siguiente elemento previsto.

```
@Override
public boolean hasNext() {
	return i < list.size();
}
@Override
public E next() {
	E e = list.get(i);
	i = i + 1;
	return e;
}
```

Se deja como ejercicio implementar un iterador sobre listas que recorre los elementos desde los índices mayores a los menores.

Un iterador, en general, se diseña para ocultar los detalles de su estado. Pero hay muchos casos en los que, por alguna razón, queremos hacerlo explícito.

En este caso un iterador viene definido por un estado t de tipo T, con valor inicial y tres funciones sin efectos laterales $boolean\ hasNext1(T\ t)$, $E\ next1(T\ t)$, $E\ next2(T\ t)$.

Hemos escogido los nombres para que sean similares a los correspondientes métodos de un iterador implementado como un objeto. De forma compacta las representaremos por *hn, n1, n2.*

Por los tanto un iterador podemos caracterizarlo por el tipo del T estado, el tipo E de los elementos devueltos y las tres funciones indicadas previamente. Dados T y E un iterador se define de por las funciones:

$$it = (t0, t \to hn(t), t \to nx1(t), t \to nx2(t))$$

Que de forma compacta lo representaremos por

$$it = (t0, hn, nx1, nx2)$$

Que definen, respectivamente el valor inicial del estado, si hay o no más elementos, *hasNext1*, el siguiente elemento a partir del estado, *next1*, y el estado siguiente, *next2*.

En el ejemplo anterior del iterador que proporciona los elementos de una lista en orden de sus índices de menor a mayor hemos considerado como

estado del iterador la tupla *(Integer, List<E>)*. Es el tipo T. Y las funciones que definen el iterador

$$(0, ls), (i, ls) \rightarrow i < ls.size(),$$
$$(i, ls) \rightarrow ls.get(i), (i, ls) \rightarrow (i + 1, ls)$$

A esta información la llamaremos iterador explícito. El algoritmo iterativo general con iterador implícito es:

```
R alg(x) {
        Iterator<E> it = it(x);
        B b = b0;
        while(it.hasNext()&& !f(b)){
                e = it.next();
                b = c(b,e);
        }
        return r(b);
}
```

Un algoritmo que use un iterador implícito, como el anterior, se puede transformar en otro que use un iterador explícito si conocemos la información pertinente (el tipo del estado, su valor inicial, y las funciones *hn, nx1, nx2*).

```
R alg(x) {
        T t = it(x);
        B b = b0;
        while(hn(t)&& !f(b)){
                E e =nx1(t);
                t = nx2(t);
                b = c(b,e);
        }
        return r(b);
}
```

En definitiva hacemos las siguientes sustituciones:

- Sustituimos *Iterator<E> it* por su estado *T t* en la declaración y en el uso *it* po *t*.
- Sustituimos *it.hasNext()* por *hn(t).*
- Sustituimos *e = it.next()* por la secuencia *E e = nx1(t); t = nx2(t);*

En lo que sigue diseñaremos los algoritmos con iteradores implícitos, pero si tenemos la información adecuada, podemos hacer explícito el iterador

de forma sistemática. También podemos hacerlo en sentido contrario: implementar un iterador para ocultar el estado.

Hay muchos tipos en Java iterables. Es decir que implementan Iterable<E>. Entre ellos tenemos listas, conjuntos, maps, etc. Otros tienen métodos asociados que producen iteradores: ficheros, cadenas, arrays, etc.

Un iterador también puede ser obtenido de un objeto de tipo *Stream<E>* mediante del método *iterator().*

Disponiendo de un iterador podemos escribir las formas imperativas del algoritmo de una manera que nos permitirá generalizarlo:

```
Double sumLista(List<Double> list) {
	Iterator<Double> it = list.iterator();
	Double b = 0.;
	while(it.hasNext()){
		Double e = it.next();
		b = b + e;
	}
	return b;
}
```

Si hacemos explicito el estado del iterador el esquema anterior se convierte en:

```
Double sumLista(List<Double> list) {
	Integer i = 0;
	Double b = 0.;
	while(i<list.size()){			//hasNext1
		Double e = ls.get(i);		//next1
		(i,ls) = (i+1,ls);		//next2
		b = b + e;
}
	return b;
}
```

Java ya aporta una sentencia *for* para recorrer iterables por lo que el algoritmo anterior, al ser la lista iterable, se puede escribir:

```
Double sumLista(List<Double> list) {
       Double b = 0.;
       for(Integer e: list){
          b = b + e;
       }
       return b;
}
```

Igualmente, la versión recursivo final usando el iterador es:

```
Double sumLista(List<Double> list) {
       Iterator<Double> it = list.iterator();
       return sumLista(it,0.);
}
```

```
Double sumLista(Iterator<Double> it,Double b) {
       if(it.hasNext()){
              Double e = it.next();
              b = sumLista(it,b+e);
       }
       return b;
}
```

Si queremos hacer explícito el estado del iterador incluimos como parámetros las componentes de la tupla que define el estado del iterador además de los parámetros iniciales y otros cálculos útiles.

```
Double sumLista(List<Double> list) {
       Integer n = list.size();
       return sumLista(i,0.,list,n);
}
```

Y la función recursiva con todos los parámetros añadidos.

```
Double sumLista(Integer i,Double b,List<Double> list,
              Integer n) {
       if(i<n){                                  //hasNext1
              Double e = list.get(i);            //next1
              i = i+1;                           //next2
              b = sumLista(i,b+e,list,n);
       }
       return b;
}
```

En muchos casos es necesario conseguir un iterable a partir de un iterator. En Java un iterable se puede conseguir de un *Stream<E>* o de un *Iterator<E>* con los métodos siguientes:

```
public static <E> Iterable<E> ofIterator(Iterator<E> it) {
	return ()->it;
}

public static <E> Iterable<E> ofStream(Stream<E> stream) {
	return () -> stream.iterator();
}
```

Por otra parte, un Stream<E> se puede conseguir de un Iterator<E> o de un Iterable<E> con los métodos:

```
public static <E> Stream<E> of(Iterable<E> iterable) {
	return StreamSupport.stream(
		Spliterators.spliteratorUnknownSize(
			iterable.iterator(),
			Spliterator.ORDERED
		),
		false
		);
}
```

```
public static <E> Stream<E> of(Iterator<E> iterator) {
	return StreamSupport.stream(
		Spliterators.spliteratorUnknownSize(
			iterator,
			Spliterator.ORDERED
		),
		false
	);
}
```

Tanto los objetos de tipo *Iterator<E>* como del tipo *Stream<E>* se consumen una vez que se recorren. Es decir, solo se pueden recorrer una vez. Si necesitamos recorrerlo más de una vez necesitamos un *iterable*, o una factoría de *streams*, que nos proporcione un iterador o un stream nuevos cada vez que los necesitemos. En Java todos los agregados iterables tienen un método *iterator()* que proporciona un *Iterator<E>*. También disponen de un método que actúa como factoría de streams. Para muchos agregados es el método *stream()* aunque para otros tiene otro nombre.

En definitiva, implementar una secuencia es desarrollar una clase que implemente *Iterator<E>.* A partir de un iterador podemos obtener un stream. Más adelante veremos cómo implementar un *stream* directamente si queremos que el flujo de datos al que está asociada la secuencia pueda verse como secuencial o paralelo.

Factoría de secuencias

En primer lugar, necesitamos una factoría de secuencias y sus iteradores o *streams* asociados. Diseñar una factoría de secuencias consiste en definir, e implementar, un conjunto de secuencias iniciales y otras obtenidas a partir de las primeras mediante algunas operaciones. Para definir una secuencia nueva, ya sea inicial o a partir de otra, debemos implementar un iterador y a partir de él (o directamente) obtener un stream.

Veamos algunos de ellos junto con sus propiedades y el código iterativo equivalente para producir los mismos elementos de la secuencia. Los iteradores asociados se pueden ver en el repositorio.

Rango: Es una secuencia, de enteros o reales, que comienza en a hasta b ($b>a$) sin incluir.

$$range(a,b) \equiv (a, e \rightarrow e < b, e \rightarrow e + 1)$$

Es la secuencia definida en Java por *IntStream.range(a,b).* El código imperativo correspondiente es:

```
for(int i = a; i<b; i++){
        …
}
```

Iterate: Es una secuencia general, cuyos elementos son de tipo *E,* definida por un valor inicial, un predicado y un operador unario.

$$iterate(e0,p,nx) \equiv (e0, e \rightarrow g(e), e \rightarrow nx(e))$$

El código imperativo correspondiente es:

```
E e = e0;
while(e->g(e)){
        e = nx(e);
        …
}
```

Un ejemplo:

```
Stream<Double> iterate(Double a, Double b, Double c){
        Double e0 = a;
        Predicate<Double> g = e->e<b;
        UnaryOperator<Double> nx = e->e*c;
        return Stream.iterate(e0,g,nx);
}
```

Su equivalente imperativo:

```
Double e = a;
while(e<b){
        e = e*c;
        …
}
```

Concatenar: Se representa en Java por *s3 = Stream.concat(s1,s2)* y obtiene otra secuencia con los elementos de *s1* seguidos por los de *s2*. En la forma imperativa tendremos que usar dos *for* consecutivos: uno para simular cada uno de los streams.

Secuencias asociadas a los agregados: Un elemento importarte para hacer posible la abstracción de datos es que los agregados exporten un iterador, o un *stream*, para poder recorrer los elementos que contienen y ocultar los detalles del estado de este. Es decir los detalles de implementación. Un agregado puede proporcionar varios iteradores para ofrecer distintos recorridos posibles. En la medida que un agregado exporta un iterador se dice que es iterable. Además los agregados ofrecen también un objeto de tipo stream. En Java esto se consigue con el método *c.stream()* para las colecciones, *Files.lines(file)* para los ficheros, *text.chars()* para las cadenas de caracteres, *Arrays.stream(a)* para los *arrays*, etc. Junto a los iteradores o streams ofrecidos por un lenguaje en particular podemos implementar

otros como hemos visto. Algunos pueden ser complejos debido a la dificultad de la implementación de los agregados.

En el caso de una lista el iterador asociado es sencillo como se puede ver arriba. En otros casos no tanto por lo que es crucial usar una implementación de confianza. Una posibilidad es obtener un stream y a partir de éste conseguir un iterador. En Java todos los objetos que implementan *Iterable<E>* nos proporcionan un iterador. Estos incluyen las colecciones.

Operaciones sobre secuencias

Las secuencias, y los iteradores y stream asociados, pueden ser sometidas a algunas operaciones para obtener otras secuencias. Veamos algunas de ellas presentando la versión funcional, su equivalente imperativo y en algunos casos la definición del iterador resultante obtenido en la operación.

Los lenguajes actuales disponen, en mayor o menor grado, de iteradores e iterables. Las implementaciones sobre iteradores en *Java* que proponemos pueden verse en el repositorio. Para otros lenguajes como *Python* también pueden verse en el repositorio. *Scala* y *Kotlin* ya disponen de un abanico muy amplio de operaciones sobre iterables. *C*, sin embargo, no dispone de estas construcciones.

Transformación (map): Obtiene una nueva secuencia a partir de otra aplicando a cada elemento una función dada. En la notación funcional se representa en Java por *s2 = s1.map(t),* siendo *t* una función, y sustituye cada elemento de la secuencia por el valor obtenidos al aplicar la función.

Ejemplo:

```
Stream<R> map(Stream<E> st, Function<E,R> f){
       return st.map(f);
}
```

Y la versión imperativa asumiendo que disponemos de un objeto *it* que sea iterable

```
for(E e: it.iterator() )
       R r = f(e);
        …
}
```

Filtro (filter): Obtiene una nueva secuencia a partir de otra quedándose solo con los elementos que cumplen un predicado. Se representa en Java por *s2 = s1.filter(h),* siendo *h* un predicado, y elimina de la secuencia *s* todos los elementos que no cumplan el predicado.

```
Stream<R> filter(Stream<E> st, Predicate<E> p){
       return st.filter(p);
}
```

Y la versión imperativa asumiendo que el objeto *it* sea iterable

```
for(E e: it.iterator() )
       if(p(e)){
       …
       }
}
```

Aplanamiento (flatMap): Obtiene una nueva secuencia concatenando los elementos de las secuencias asociadas a cada elemento mediante una función. Se representa en Java por *s2 = s1.flatMap(seq),* siendo *seq* una función que por cada elemento de la secuencia original obtiene una nueva secuencia. Sustituye los elementos de la secuencia por los valores de la nueva secuencia obtenidos al aplicar la función.

```
Stream<R> flatMap(Stream<E> st, Function<E,Stream<R>> f){
       return st.flatMap(f);
}
```

Y la versión imperativa asumiendo que el objeto it es iterable

```
for(E e: it.iterator() )
       for(R r: f(e)){
              …
       }
}
```

Límite (limit): Se representa en Java por *s2 = s1.limit(n)* y obtiene otra secuencia con los primeros *n* elementos de *s*.

```
Stream<E> limit(Stream<E> st, Integer lm){
    return st.limit(n);
}
```

Y la versión imperativa asumiendo que el objeto it es iterable.

```
Integer i = 0;
Iterator<E> itt = it.iterator();
while(itt.hasNext() && i< n)
       i++;
       E e = itt.next();
       …
}
```

Sorted(ordenado): Se representa en Java por *s2 = s1.sorted(orden)* y obtiene otra secuencia con elementos de *s* ordenados

```
Stream<E> sorted(Stream<E> st,Comparator<E> cmp){
    return st.sorted(cmp);
}
```

La versión imperativa siendo it iterable:

```
List<E> ls = toList(it);  //un acumulador visto más adelante
Collections.sort(ls,cmp);
for(E e:ls)
       …
}
```

Distinct: Se representa en Java por *s2 = s1.distinct()* y obtiene otra secuencia con elementos de *s* no repetidos.

```
Stream<E> distinct(Stream<E> st){
    return st.distinct();
}
```

La versión imperativa es:

```
Set<E> set = new HashSet<>();
for(E e: it.iterator()){
	if(set.contains()) continue;
	set.add(e);
	…
}
```

Las siguientes operaciones sobre secuencias son de uso común en los lenguajes actuales, aunque no se ofrecen en Java y tenemos que implementarlas. La implementación de los iteradores asociados a las secuencias resultantes puede verse en el repositorio.

Enumerate: Devuelve la secuencia formada por los pares de elementos formados por los elementos de *s* y su posición en la misma.

```
Stream<Enumerate<E>> Stream2.enumerate(Stream<E> stream,
	Integer start)
```

Donde Enumerate<E> es el record

```
record Enumerate<E>(Integer counter, E value) {
	public static <E> Enumerate<E> of(Integer num,E value) {
		return new Enumerate<E>(num, value);
	}
}
```

Pares consecutivos: *consecutivePairs(s)* devuelve una secuencia formada por los pares de elementos formados por cada elemento de *s* y su siguiente.

```
<E> Stream<Pair<E,E>> Stream2.consecutivePairs(Stream<E> sm)
```

Producto Cartesiano: *cartesianProduct(s1, s2, f)* devuelve cada elemento de *s1* operado con todos los de *s2* mediante la *bifunción f*

```
Stream<R> Stream2.cartesianProduct(Stream<T> s1, Stream<U> s2,
	BiFunction<T, U, R> f)
```

Zip: *zip(s1, s2, f):* Una secuencia formada combinando los elementos de las secuencias de entrada situados en las mismas posiciones mediante la *bifunción f*.

```
Stream<T> Stream2.zip(Stream<L> leftStream,
       Stream<R> rightStream, BiFunction<L, R, T> combiner)
```

Todos los pares: *allPairs(n1,n2,m1,m2).* Una secuencia de todos los pares formados por enteros en los rangos *n1..n2, m1..m2*

```
Stream<IntPair> Stream2.allPairs(Integer n1, Integer n2,
Integer m1, Integer m2)
```

Todos lo trios: *allTrios(n1,n2,m1,m2,r1,r2).* Una secuencia de todos los tríos formados por enteros en los rangos *n1..n2, m1..m2, r1..r2*

```
Stream<IntTrio> allTrios(Integer n1, Integer n2, Integer m1,
Integer m2, Integer r1, Integer r2)
```

Los iteradores asociados a las secuencias anteriores pueden verse en Java en el paquete *us.lsi.iterables*. Los *streams* asociados a las operaciones anteriores en *us.lsi.streams*.

Acumuladores secuenciales

Según las propiedades de las funciones asociadas al acumulador podemos clasificarlos en *acumuladores secuenciales* y *paralelos*. Cada uno de ellos es adecuado al tipo de flujo, secuencial o paralelo, que tengamos. Los *algoritmos secuenciales* que estamos estudiando tienen asociados un flujo de datos secuencial, que define una secuencia, y un acumulador secuencial. Los acumuladores paralelos se asocian a flujos de datos paralelos y a algoritmos paralelos de acumulación de datos que veremos más adelante.

Veremos ahora, entonces, los *acumuladores secuenciales*. En el apartado anterior hemos visto las relaciones de los algoritmos iterativos y las secuencias, las operaciones sobre las mismas y los esquemas imperativos asociados a esas operaciones. Otro elemento que aparece en los algoritmos iterativos es el *acumulador secuencial*.

Como ya dijimos una forma general de algoritmo iterativo explicitando el estado del iterador y asumiendo que la base del acumulador es un tipo inmutable es

```
R alg(x) {
       T t = it(x);
       B b = b0;
       while(hn(t)&&!f(b)){
          E e = nx1(t);
          t = nx2(t);
          b = c(b,e);
       }
       return r(b);
}
```

Y sin hacer explícito el estado del iterador:

```
R alg(x) {
       Iterator<E> it = it(x);
       B b = b0;
       while(it.hasNext()&& !f(b)){
       e = it.next();
             b = c(b,e);
       }
       return r(b);
}
```

Y su equivalente funcional, de forma abstracta, es:

```
R alg(x) {
       A a = (b0,(b,e) → c(b,e),b → r(b),b → f(b));
       S s = =(() → e0,e → g(e),e → s(e));
       R r = s.collect(a);
       return r;
}
```

Como ya hemos comentado en este esquema hay una serie de elementos que definen una secuencia finita, como ya hemos visto, definida por $s = (e_0, e \to g(e), e \to nx(e))$. El resto de los elementos definen *un acumulador secuencial.* Esencialmente un acumulador secuencial puede ser definido por:

$$a = (b0, (b,e) \to c(b,e), b \to r(b), b \to f(b))$$

La forma anterior es la adecuada si el tipo B es inmutable. Si B es mutable la forma del acumulador es:

$$a = (() \to b0, (b,e) \to b.c(e), b \to r(b), b \to f(b))$$

Es decir el valor inicial es de tipo un *Supplier<B>* en los mutables y de tipo *B* en los inmutables y la función de combinación de tipo *BiConsumer<B,E>* en los mutables frente a *BiFunction<B,E,B>* en los inmutables.

En lo que sigue los escribiremos asumiendo que la base del acumulador es inmutable. Si fuera mutable habrá que sustituir *b0* por *()->b0* y *(b,e)->c(b,e)* por *(b,e)->b.c(e).*

Veamos cada uno de los elementos:

- El acumulador tiene un estado, la base del acumulador, de tipo B y con *valor inicial* b_0. El tipo B puede ser mutable o inmutable.
- Los elementos e, de tipo E, de la secuencia s definida previamente se acumulan en la base mediante la *función de acumulación* $(b, e) \rightarrow c(b, e)$. En principio asumimos que el tipo de la base es inmutable. Si fuera mutable la función de acumulación se convierte en *b.c(e)*
- El algoritmo devuelve, al terminar, el valor de la base del acumulador transformado por la *función de retorno* $b \rightarrow r(b)$. El valor devuelto es del tipo R. Por lo tanto un acumulador es un tipo genérico basado en los tipos E, B, R.
- El algoritmo puede terminar antes de que la secuencia acabe si se cumple alguna condición sobre la base del acumulador. Esta condición la establece el predicado $b \rightarrow f(b)$, que es verdadero si con el valor de la base podemos considerar que el trabajo está terminado y no es necesario seguir. Por similitud con las características de cortocircuito de algunos operadores llamaremos a esta función la *función de cortocircuito* del acumulador.
- Hay muchos acumuladores, que llamaremos simples, donde se cumple que la función de retorno es la función identidad y la función de cortocircuito devuelve siempre false. En este caso sólo indicaremos los dos primeros elementos del acumulador. Es decir $a = (() \rightarrow b0, (b, e) \rightarrow c(b, e))$

Como vemos un acumulador se define mediante un un valor inicial (o un *supplier* si la base es mutable), una *bifunction* como función de acumulación (o un *consumer* si la base del acumulador es mutable), una *function* para definir el retorno y un *predicate* para definir la función de cortocircuito.

Factorías de acumuladores secuenciales

Un acumulador de tipo A viene definido por el tipo de los elementos a acumular E, el tipo de la base B, y el tipo de retorno R, más el valor inicial de la base y el resto de las funciones comentadas.

$$A = (E, B, R), \qquad a = (b0, (b, e) \rightarrow c(b, e), b \rightarrow r(b), b \rightarrow fn(b))$$

Definir un acumulador es concretar todos sus detalles.

Es conveniente disponer de una factoría de acumuladores secuenciales. Cada acumulador vendrá definido por todas sus propiedades o por una versión reducida.

En la versión reducida asumimos que $r(b) = b, f(b) = false$.

Una factoría de acumuladores secuenciales dispondrá de métodos adecuados para crear los acumuladores más usados. En Java se dispone de la factoría *Collectors*. Pero los acumuladores creados por ella son, a la vez, paralelos y secuenciales, por lo que los veremos más adelante.

Es conveniente conocer un catálogo de acumuladores. Un conjunto de acumuladores conocidos se muestra a continuación. Para cada acumulador se muestra un algoritmo iterativo que acumula una secuencia con ese acumulador. Mostramos el algoritmo en sus formas imperativa, recursiva final y funcional.

All(p)

All(p): $B = R = Boolean, a = (true, (b, e) \rightarrow p(e), b \rightarrow b, b \rightarrow !\, b)$

Este acumulador devuelve true si el predicado es verdadero al ser aplicado a cada elemento de la secuencia. El esquema iterativo, funcional e imperativo, con este acumulador es:

```
Boolean all(X x, Predicate<E> p) {
       Steam<E> st = st(x);
       return st.allMatch(p);
}
```

```
Boolean all(X x, Predicate<E> p) {
	Iterator<E> it = it(x);
	Boolean b = true;
	while(it.hasNext()&& b){
		E e = it.next();
		b = p(e);
	}
	return b;
}
```

La version recursiva final:

```
Boolean all(X x, Predicate<E> p) {
	Iterator<E> it = it(x);
	Boolean b = true;
	b = all(it,b,p);
	return b;
}
```

```
Boolean all(Iterator<E> it, Boolean b, Predicate<E> p){
	if(it.hasNext()&& b){
		E e = it.next();
		b = p(e);
		b = all(it,b,p);
	}
	return b;
}
```

---

1. *Diseñar una función que compruebe si todos los elementos de una secuencia son iguales*

La idea es comprobar que se verifica que todos los elementos son iguales al primero. El primero, sin consumir la secuencia, lo podemos obtener con la función *peek*.

```
<E> Boolean allEquals(Stream<E> st) {
	MutableType<E> first = MutableType.of(null);
	return st.peek(e->{if(first.value()==null)
			first.setValue(e);})
		.allMatch(e->e.equals(first.value()));
}
```

Esta función *allEquals* es muy útil para resolver problemas. La versión imperativa podemos obtenerla teniendo en cuenta el esquema para *allMatch* anterior y ampliando el estado del algoritmo con la variable que guarda el primer elemento.

```
Boolean allEquals(Iterable<E> st) {
	Iterator<E> it = st.iterator();
	Boolean b = true;
	E first = null;
	while(it.hasNext()&& b){
		E e = it.next();
		if(first == null) first = e;
		b = e.equals(first);
	}
	return b;
}
```

2. *Decidir si los enteros de una secuencia forman una progresión aritmética.*

La solución de este problema usa el ejercicio anterior. La secuencia es aritmética si formada la secuencia de diferencias entre elementos consecutivos todos los elementos son iguales.

```
Boolean esAritmetica(Stream<Integer> st) {
	Stream<Integer> stc = Stream2.consecutivePairs(st)
		.map(p->p.second()-p.first());
	return Stream2.allEquals(stc);
}
```

La versión imperativa se puede conseguir escogiendo un estado del algoritmo que incluya el elemento anterior y el primer elemento tal como se hace en las funciones *consecutivePairs* y *allEquals*. Además, usar es esquema del acumulador *all*.

```
Boolean esAritmetica(Iterable<Integer> iterable) {
	Iterator<Integer> it = iterable.iterator();
	Integer last,e;
	if(it.hasNext()) e = it.next(); else return true;
	Boolean b = true;
	Integer r = null;
	while(it.hasNext()  && b){
		last = e;
		e = it.next();
		if(r == null) r = e-last;
		b = e-last == r;
	}
	return b;
}
```

None(p)

None(p): $B = RBoolean, a = (false, (b, e) \rightarrow p(e), b \rightarrow !b, b \rightarrow b)$

Este acumulador devuelve true si el predicado es falso al ser aplicado a cada elemento de la secuencia. El esquema iterativo con este acumulador es:

```
Boolean none1(X x, Predicate<E> p) {
	Iterator<E> it = it(x);
	Boolean b = false;
	while(it.hasNext()&& !b){
		e = it.next();
		b = p(e);
	}
	return !b;
}
```

```
Boolean none2(X x, Predicate<E> p) {
	Iterator<E> it = it(x);
	Boolean b = false;
	b = none(it,b,p);
	return !b;
}
```

```
Boolean none2(Iterator<E> it, Boolean b, Predicate<E> p){
	if(it.hasNext()&& !b){
		E e = it.next();
		b = p(e);
		b = alg(it,b,p);
	}
	return b;
}
```

```
Boolean none3(X x, Predicate<E> p) {
	Steam<E> st = st(x);
	return st.noneMatch(p);
}
```

3. *Decidir si un número es primo*

Un número n es primo si no es divisible por ningún número entero en el rango $[2, \sqrt{n}]$.

Vemos que una secuencia adecuada es la que recorre los valores $2 \ldots \sqrt{n}$ y el acumulador *none* con el predicado *esDivisible*. Esta secuencia y este acumulador se pueden definir como:

$s = (2, e \to e \leq \sqrt{n}, e \to e + 1)$

$B = Boolean, a = (false, (b, e) \to p(e), b \to !\, b, b \to b)$

$p(e) = Math2.esDivisible(n, e)$

La secuencia puede ser implementada con un iterador con estado formado por un entero que se va incrementando de uno en uno. A partir de ahí podemos obtener la versión funcional, imperativa, recursiva final siguiendo los pasos genéricos anteriores.

```
boolean esPrimo1(Long n){
	Long sqrt = (long)Math.sqrt((double)n);
	return LongStream.rangeClosed(2, sqrt)
			.noneMatch(x->Math2.esDivisible(n, x));
}
```

```
boolean esPrimo2(Long n){
	Long sqrt = (long)Math.sqrt((double)n);
	Long e = 2L;
	Boolean b = false;
	while(e <= sqrt && !b){
		b = Math2.esDivisible(n, e);
		e = e + 1;
	}
	return !b;
}
```

```
boolean esPrimo3(Long n){
	Long sqrt = (long)Math.sqrt((double)n);
	Long e = 2L;
	Boolean b = false;
	return !esPrimo3(n,sqrt,e,b);
}
boolean esPrimo3(Long n, Long sqrt, Long e, Boolean b){
	if(e <= sqrt && !b){
		b = esPrimo3(n,sqrt,e+1,Math2.esDivisible(n, e));
	}
	return b;
}
```

Any(p)

Any(p): $B = R = Boolean, a = (false, (b, e) \rightarrow p(e), b \rightarrow b, b \rightarrow b)$

Este acumulador devuelve true si el predicado es verdadero al ser aplicado a algún elemento de la secuencia. El esquema funcional y el imperativo con este acumulador son:

```
Boolean any1(X x, Predicate<E> p) {
	Stream<E> st = st(x);
	return st.anyMatch(p);
}
```

```
Boolean any2(X x, Predicate<E> p) {
	Iterator<E> it = it(x);
	Boolean b = false;
	while(it.hasNext()&& !b){
		e = it.next();
		b = p(e);
	}
	return b;
}
```

La version recursive final

```
Boolean any3(X x, Predicate<E> p) {
	Iterator<E> it = it(x);
	Boolean b = true;
	b = alg(it,b);
	return b;
}
```

```
Boolean any3(Iterator<E> it, Boolean b, Predicate<E> p){
	if(it.hasNext()&& !b){
		E e = it.next();
		b = p(e);
		b = any(it,b,p);
	}
	return b;
}
```

Sum

Sum(): $B = R = Number, a = (0, (b, e) \rightarrow b + e)$

Este acumulador devuelve la suma de los elementos de la secuencia. Las diferentes versiones del esquema iterativo con este acumulador son:

```
Double sum(X x) {
	Stream<E> st = st(x);
	return st.mapToDouble(e->e).sum();
}
```

```
Double sum(X x) {
	Iterator<E> it = it(x);
	Double b = 0.;
	while(it.hasNext()){
	   e = it.next();
	   b = b+e;
	}
	return b;
}
```

```
Double sum alg(X x) {
	Iterator<Double> it = it(x);
	Double b = 0.;
	b = sum(it,b);
	return b;
}
```

```
Boolean sum(Iterator<Double> it, Double b){
	if(it.hasNext()){
		Double e = it.next();
		b = b+e;
		b = sum(it,b);
	}
	return b;
}
```

Count

Count(): $B = R = Integer, a = (0, (b, e) \rightarrow b + 1)$

Este acumulador devuelve el número de elementos de la secuencia. Las diferentes versiones del esquema iterativo con este acumulador son:

```
Long count1(X x) {
	Steam<E> st = st(x);
	return st.count();
}
```

```
Long count2(X x) {
	Iterator<E> it = it(x);
	Long b = 0L;
	while(it.hasNext()){
		e = it.next();
		b = b+1;
	}
	return b;
}
```

```
Long count3(X x) {
	Iterator<E> it = it(x);
	Long b = 0L;
	b = count(it,b);
	return b;
}
```

```
Long count3(Iterator<E> it, Long b){
       if(it.hasNext()&& b){
              E e = it.next();
              b = b +1;
              b = count(it,b);
       }
       return b;
}
```

First

First(): $B = (E, Boolean)$, R=Optional<E>

Este acumulador devuelve el primer elemento de la secuencia. Es interesante cuando se aplica a una secuencia que puede haber sido transformada y filtrada. Las diferentes versiones del esquema iterativo con este acumulador son:

```
Optional<E> first1(X x) {
       Steam<E> st = st(x);
       return st.findFirst();
}
```

```
Optional<E> first2(X x) {
       Iterator<E> it = it(x);
       E b = null;
       while(it.hasNext()&& b==null){
          b = it.next();
       }
       return Optional.ofNullable(b);
}
```

```
Optional<E> first3(X x) {
       Iterator<E> it = it(x);
       E b = null;
       b = alg(it,b);
       return Optional.ofNullable(b);
}
```

```
B first3(Iterator<E> it, Boolean b){
       if(it.hasNext()&& b == null){
              b = it.next();
              b = first3(it,b);
       }
       return b;
}
```

4. Diseñar una función que dado un entero devuelva el siguiente primo

El siguiente primo de un número *n* mayor o igual que dos es el primero de la secuencia de impares mayores que *n* filtrada por los que sean primos.

Partimos de la secuencia de impares posteriores al número dado, la filtramos por los primos y encontramos el primero. Como tenemos la seguridad de que siempre existe el siguiente primo el *Optional* resultante lo convertimos a entero. Abajo se incluye la versión funcional, imperativa y recursiva final.

```
Long siguientePrimo1(Long n){
	Long e0 = n%2==0?n+1:n+2;
	return Stream.iterate(e0, e->e+2)
		.filter(e->Math2.esPrimo(e))
		.findFirst()
		.get();
}
```

```
Long siguientePrimo2(Long n){
	Long e = n%2==0?n+1:n+2;
	Long r = null;
	while(r == null){
		if(esPrimo2(e)) r = e;
		e = e+2;
	}
	return r;
}
```

```
Long siguientePrimo3(Long n){
	Long e = n%2==0?n+1:n+2;
	Long r = null
	return siguientePrimo2(n,e,r);
}
```

```
Long siguientePrimo3(Long n, Long e, Long r){
	if(r==null){
		if(esPrimo2(e)) r = e;
		r = siguientePrimo3(n,e+2,r);
	}
	return r;
}
```

Last

Last(): $B = (E)$, R=Optional<E>

Este acumulador devuelve el último elemento de la secuencia. Es interesante cuando se aplica a una secuencia que puede haber sido transformada y filtrada. La versión imperativa es:

```
Optional<E> last(X x) {
       Iterator<E> it = it(x);
       E b = null;
       while(it.hasNext()){
          b = it.next();
       }
       return Optional.ofNullable(b);
}
```

La versión funcional es:

```
public static <E> Optional<E> findLast(Stream<E> stream){
             return stream.reduce((first,second)->second);
}
```

Las versión recursivo final queda como ejercicio.

---

5. Diseñar una función que devuelva el mayor primo menor que m

---

Podemos definir la secuencia de primos mayores menores que m y quedarnos con el último elemento de la secuencia. La secuencia puede empezar en 2 o en otro primo menor que m.

```
Optional<Integer> p1 =
       Stream2.findLast(Stream.iterate(2,e->e<32,
              e->Math2.siguientePrimo(e)));
```

```
Optional<Long> primeroMayor(Long m) {
	E b = null;
	Integer n = 2;
	while(n<m){
		b = n;
		n = siguientePrimo(n);
	}
	return Optional.ofNullable(b);
}
```

Joining

Joining(sp,pf,sf): $B = (String, Boolean), E = R = String$

Donde los valores *(sp,pf,sf)* son el separador, el prefijo y el sufijo.

Este acumulador se aplica a una secuencia cuyos elementos son de tipo *String* y devuelve otra cadena formada por los elementos de la secuencia separados por el separador con el prefijo al principio y el sufijo al final. Las diferentes versiones del esquema iterativo con este acumulador son:

```
String join2(X x, String sp, String pf, String sf) {
	Stream<E> st = st(x);
	return st.collect(Collectors.joining(sp,pf,sf);
}
```

```
String join1(X x, String sp, String pf, String sf) {
	Iterator<E> it = it(x);
	String b = pf;
	Boolean isFirst = true;
	while(it.hasNext()){
		e = it.next();
		if(isFirst){
			b = b+e;
			isFirst = false;
		}else b = b+sp+e;
	}
	return b+sf;
}
```

```
String join3(X x, String sp, String pf, String sf) {
	Iterator<Stream> it = it(x);
	String b = pf;
	Boolean isFirst = true;
	b = alg(sp,it,b,isFirst);
	return b+sf;
}
```

```
String join3(String sp, Iterator<Stream> it, String b,
             Boolean isFirst){
      if(it.hasNext()){
             E e = it.next();
             if(isFirst){
                    b = b+e;
                    isFirst = false;
             }else b = b+sp+e;
             b = join3(sp,it,b,isFirst);
      }
      return b;
}
```

Reduce

Es un acumulador muy genérico de los elementos de una secuencia mediante un operador binario. Tiene varias variantes

Reduce(b0,op): E, B =R, $op: (B, B) \rightarrow B$

Es una reducción con elemento inicial. La función de acumulación es un operador binario.

Reduce(op): $B = E, op: (B, B) \rightarrow B, R =$ *Optional<E>*

Es una reducción sin elemento inicial donde el tipo de la base del acumulador es el mismo que el de los elementos de la secuencia. En este caso la función de acumulación es un operador binario y el resultado es *Optional<E>.*

Reduce(b0,f,op): $b0: B, f: E \rightarrow B, op: (B, B) \rightarrow B, R = B$

Es una reducción con elemento inicial. Ahora los elementos de la secuencia son transformados al tipo de la base antes de ser acumulados por el operador binario. La función de acumulación es ahora *op(b,f(e)).*

Veremos sólo las versiones imperativas y recursivas finales de cada variante.

El esquema de la primera versión es:

```
E  reduce0(X x, BinaryOperator<E> op, E b0) {
      Stream<E> st = st(x);
      return st.collect(Collectors.reduce(b0,op));
}
```

```
E reduce1(X x, BinayOperator<E> op, E b0) {
	Iterator<E> it = it(x);
	E b = b0;
	while(it.hasNext()){
		e = it.next();
		b = op(b,e);
	}
	return b;
}
```

```
E reduce2(X x, BinayOperator<E> op, E b0) {
	Iterator<E> it = it(x);
	E b = b0;
	b = reduce2(it,b,op);
	return b;
}
E reduce2(Iterator<E> it, E b, BinayOperator<E> op){
	if(it.hasNext()){
		E e = it.next();
		b = op(b,e);
		b = reduce2(it,b,op);
	}
	return b;
}
```

El acumulador anterior tiene muchos casos particulares para B inmutable: *sum, count* entre otros y para *B* mutable: *toList, toSet, counting*, y los diversos *grouping* vistos más abajo.

La segunda versión es similar pero la base comienza en *null* y se actualiza al primer elemento cuando lo encuentra. Su esquema es:

```
Optional<E>  reduce0(X x, BinaryOperator<E> op) {
	Stream<E> st = st(x);
	return st.collect(Collectors.reduce(op));
}
```

```
Optional<E> reduce1(X x, BinaryOperator<E> op) {
	Iterator<E> it = it(x);
	E b = null;
	while(it.hasNext()){
		e = it.next();
		if(b == null) b = e;
		else b = op(b);
	}
	return Optional.ofNullable(b);
}
```

```
Optional<E> reduce2(X x, BinaryOperator<E> op) {
	Iterator<E> it = it(x);
	B b = null;
	b = alg(it,b,op);
	return Optional.ofNullable(b);
}
```

```
B reduce2(Iterator<E> it, B b, BinaryOperator<B> op){
	if(it.hasNext()){
		E e = it.next();
		if(b==null)  b = e;
		else b = op(b,e);
		b = reduce2(it,b,op);
	}
	return b;
}
```

Casos particulares de este *reduce* son los acumuladores *min* y *max* escogiendo como operadores binarios *min(a,b)* o *max(a,b).*

La tercera versión transforma el elemento de entrada antes de acumularlo.

```
B  reduce0(X x, Function<E,B> f, BinaryOperator<B> op, E b0) {
	Stream<E> st = st(x);
	return st.collect(Collectors.reduce(b0,f,op));
}
```

Dejamos como ejercicio los esquemas imperativos y recursivos finales de ambas versiones.

Las versiones recursiva final y funcional de esta variante se dejan como ejercicio.

Un caso aún más general es un reduce que incluya la función de cortocircuito

```
B reduce(X x, B b0, Function<E,B> f, BinaryOperator<B> op,
		Predicate<B> p) {
	Iterator<E> it = it(x);
	B b = b0;
	while(it.hasNext() && !p(b)){
		e = it.next();
		b = op(b,f(e));
	}
	return b;
}
```

Casos particulares de este último son *all, any, none, findFirst*, etc. Las versiones recursiva final y funcional de este último se dejan como ejercicio.

---

6. Obtener un String con lo primeros de caracteres de una lista de String

---

```
public static String ts(List<String> ls) {
              return ls.stream()
                     .collect(Collectors.reducing("",
                            s->s.substring(0, 1),(x,y)->x+y));
}
```

Las versiones imperativa y recursiva final quedan como ejercicio. Pueden hacerse siguiendo los esquemas anteriores.

ToList, ToSet, ToMultiset

ToList: B=R=List<E>
ToSet: B=R=Set<E>
ToMultiset: B=R=Multiset<E>

Este acumulador devuelve los elementos de la secuencia en una lista. El esquema *ToSet* y *ToMultiset* son similares, pero sustituyendo la lista por un conjunto o un M*ultiset*. Al acumular los elementos de un agregado en un conjunto podemos encontrar los elementos distintos que hay en el iterable. Al acumular en un *Multiset* obtenemos la frecuencia de cada elemento. El esquema iterativo en su versión funcional es;

```
List<E>  toList(X x) {
       Stream<E> st = st(x);
       return st.toList();
}
```

La versión imperativa;

```
List<E> toList(X x) {
       Iterator<E> it = it(x);
       List<E> b = new ArrayList<>();
       while(it.hasNext()){
          e = it.next();
          b.add(e);
       }
       return b;
}
```

Y la version recursive final:

```
List<E> toList(X x) {
       Iterator<E> it = it(x);
       List<E> b = new ArrayList<>();
       b = alg(it,b);
       return b;
}
```

```
List<E> toList(Iterator<E> it, List<E> b){
       if(it.hasNext()){
              E e = it.next();
              b.add(e);
              b = toList(it,b);
       }
       return b;
}
```

---

7. Diseñar una función que indique si todos los elementos de un iterable son diferentes

---

Esta acumulación se puede conseguir acumulando en un conjunto y a la vez contando los elementos. Los elementos serán diferentes si el número

de elementos en el conjunto es igual al número de elementos contados. La versión imperativa es:

```
Boolean allDifferents(X x) {
       Iterator<E> it = it(x);
       Integer n = 0;
       Set<E> b = new HashSet<>();
       while(it.hasNext()){
          e = it.next();
          b.add(e);
          n = n+1;
       }
       return n==b.size();
}
```

8. *Encontrar las veces que aparece cada elemento en un iterable*

Si agrupamos el iterable en un *multiset* obtenemos las frecuencias

```
Multiset frecuencias(X x) {
       Iterator<E> it = it(x);
       Multiset<E> b = Multiset.empty();
       while(it.hasNext()){
          e = it.next();
          b.add(e);
       }
       return b;
}
```

GroupsSize

GroupsSize(key): key: $E \rightarrow K$, B=R=Map<K,Integer>

Este acumulador devuelve el número de elementos en cada uno de los grupos de los elementos definidos por la función *key*. El esquema en su versión funcional es:

```
Map<K,Integer> groupsSize1(X x, Function<E,K> key) {
       Stream<E> st = st(x);
       return st.collect(
          Collectors.groupingBy(key,
             Collectors.collectingAndThen(
                 Collectors.counting(),Long::intValue)));
}
```

La versión recursiva final

```
Map<K,Integer> groupsSize3(X x, Function<E,K> key) {
       Iterator<E> it = it(x);
       Map<K,Integer> b = HashMap<>();
       b = GroupsSize3(it,b);
       return b;
}
```

```
Map<K,Integer> groupsSize3(Iterator<E> it,Map<K,Integer>  b){
       if(it.hasNext()){
              e = it.next();
              K k = key(e);
              if(b.containsKey(k))
                     b.put(k,b.get(k)+1);
              else
                     b.put(k, 1);
              b = GroupsSize3(it,b);
       }
       return b;
}
```

La versión imperativa es:

```
Map<K,Integer> GroupsSize2(X x, Function<E,K> key) {
       Iterator<E> it = it(x);
       Map<E,Integer> b = HashMap<>();
       while(it.hasNext()){
          e = it.next();
          K k = key(e);
          if(b.containsKey(k)){
              b.put(k,b.get(k)+1);
           else
              b.put(k, 1);
       }
       return b;
}
```

9. *Dado un iterable de ciudades encontrar el número de ellas en cada provincia*

Siendo el tipo Ciudad

```
record Ciudad(String nombre, String provincia) {
    public static Ciudad of(String nombre, String provincia){
                    return new Ciudad(nombre, provincia);
    }
}
```

```
List<Ciudad> = …
Map<String,Integer> m =
             groupsSize(ls.stream(),c->c.provincia());
```

GroupingList, GroupingSet

GroupingList(key): key: $E \rightarrow K$, B = R = Map<K,List<E>>.
GroupingSet(key): key: $E \rightarrow K$, B = R = Map<K,Set<E>>.

Este acumulador devuelve grupos en forma de listas de elementos de la secuencia definidos por la función *key*. El esquema *GroupingSet* es similar y forma un conjunto por grupo. El esquema en su versión funcional e imperativa es;

```
Map<K,Integer> groupingList(X x, Function<E,K> key) {
       Stream<E> st = st(x);
       return st.collect(Collectors.groupingBy(key));
}
```

```
Map<K,List<E>> groupingList(X x, Function<E,K> key) {
       Iterator<E> it = it(x);
       Map<K,List<E>> b = HashMap<>();
       while(it.hasNext()){
          e = it.next();
          K k = key(e);
          if(b.containsKey(k))
               b.get(k).add(e);
          else {
               List<E> ls = new ArraList<>();
               ls.add(e)
               b.put(e,ls);
          }
       }
       return b;
}
```

Y en su versión recursivo final:

```
Map<K,List<E> groupingList(X x, Function<E,K> key) {
        Iterator<E> it = it(x);
        Map<K,List<E>> b = HashMap<>();
        b = alg(it,b);
        return b;
}
```

```
Map<K,List<E>> groupingList(Iterator<E> it, Map<K,List<E>> b){
        if(it.hasNext()){
                e = it.next();
                K k = key(e);
                if(b.containsKey(k))
                        b.get(k).add(e);
                else {
                        List<E> ls = new ArraList<>();
                        ls.add(e)
                        b.put(e,ls);
                }
                b = groupingList(it,b);
        }
        return b;
}
```

GroupingReduce

GroupingReduce(key,f,op,v0): B = R = Map<K,V>

Este acumulador forma grupos con los elementos de la secuencia definidos por la función *key.* Posteriormente acumula cada grupo mediante una reducción con transformación.

La versión funcional es:

```
Map<K,R> groupingReduce1(X x, Function<E,K> key,
                Function<E,R> f,BinaryOperator<R> op){
        Stream<E> st = st(x);
        return st.collect(Collectors.groupingBy(key,
                Collectors.mapping(f,
                        Collectors.collectingAndThen(
                                Collectors.reducing(op),
                                        g->g.get()))));
}
```

La versión imperativa.

```
Map<K,R> groupingReduce2(X x, Function<E,K> key,
              Function<E,R> f, BinaryOperator<R> op){
       Iterator<E> it = it(x);
       Map<K,R> b = new HashMap<>();
       while(it.hasNext()){
              E e = it.next();
              K k = key.apply(e);
              R r = f.apply(e);
              if(b.containsKey(k))
                     b.put(k,op.apply(b.get(k),r));
              else
                     b.put(k,r);
              }
       return b;
}
```

La recursive final:

```
Map<K,R> groupingReduce3(X x, Function<E,K> key,
              Function<E,R> f,BinaryOperator<R> op){
       Iterator<E> it = it(x);
       Map<K,R> b = HashMap<>();
       b = groupingReduce(it,b);
       return b;
}
```

```
Map<K,R> groupingReduce3(Iterator<E> it, Function<E,K> key,
              Function<E,R>f, BinaryOperator<R>){
       if(it.hasNext()){
              e = it.next();
              K k = key.apply (e);
              R r = f.apply (e);
              if(b.containsKey(k))
                     b.put(k,op.apply(b.get(k),r);
              else
                     b.put(k,r);
              b = groupingReduce(it,b);
       }
       return b;
}
```

10. Dado un iterable de personas encontrar los de mayor edad para cada nombre

Sea el tipo Persona definido por el record

```
record Persona(String nombre, Integer edad) {
	public static Persona of(String nombre, Integer edad) {
			return new Persona(nombre,edad);
	}
}
```

El objetivo es agrupar a las personas por su nombre y encontrar el máximo para cada grupo según la edad. Es un caso concreto de *GroupsReduce* con la clave el nombre y la operación de reducción obtenida aplicando el operador binario mínimo para la propiedad edad.

```
List<Persona> ls = …
Map<String,Integer> m = groupingReduce(ls.stream(),
	p->p.nombre(),p->p.edad(),
	BinaryOperator.maxBy(Comparator.naturalOrder()));
```

Transformación de algoritmos iterativos

Como hemos estado viendo hay tres formas de representar un algoritmo iterativo: imperativa, funcional o recursiva final. Veamos como transformar de una a otra,

De la notación funcional a la imperativa

Como hemos visto existe una fuerte correlación entre la notación funcional y la imperativa de los tipos de algoritmos iterativos que estamos viendo aquí. Veamos cómo transformarlas de una a otra. Ya sea en un sentido o en otro se trata de identificar la secuencia y el acumulador y a partir de aquí escribir el algoritmo.

Si partimos de la notación funcional la identificación de la secuencia y el acumulador suele ser sencilla. A partir de este paso y de las equivalencias de la sección anterior podemos escribir el algoritmo imperativo. Veamos el siguiente ejemplo.

1. Obtener una lista con los primos menores que n.

Para resolver este problema podemos considerar una secuencia formada por los números primos, que comienza en dos y continua con el siguiente primo, hasta el primo que supere n sin incluir y escoger el acumulador *toList.*

```
List<Long> primos(Long n) {
	return Stream.iterate(2L,e<n,
			e->Math2.siguientetePrimo(e))
	.toList();
}
```

Y la versión imperativa

```
List<Long> primos2(Long n) {
	Long e = 2L;
	List<Long> r = new ArrayList<>();
	while(e<n) {
		r.add(e);
		e = Math2.siguientePrimo(e);
	}
	return r;
}
```

De la notación imperativa a la funcional

El proceso inverso suele ser más complejo debido a los aspectos imperativos. La idea es reducir cada bloque básico a una asignación de tuplas para encontrar una forma estandarizada de los algoritmos imperativos.

Para poder sistematizar el proceso debemos introducir algunos conceptos adicionales y recordar otros. En primer lugar, recordemos que una *tupla* es un agregado de elementos, que pueden ser de distintos tipos, pero de tamaño fijo.

Muchos lenguajes actuales ya disponen del tipo tupla implementado, pero Java no lo tiene, pero podemos implementarlo mediante **records** que ya hemos estudiado previamente. Una tupla de tres elementos la

representamos por $(Integer\ a, Double\ b, String\ c)$, o (a, b, c) si los tipos ya están definidos. Suponemos que *t, t2, t2* son tuplas y *T* su tipo.

Declaración e inicialización:

```
public static record T(Integer a, Double b, String c) {
       public static T of(Integer a, Double b, String c){
              return new T(a,b,c);
}


T t = T.of(2,5.4,"Hola");
T t2 = t;
String s = t.c();
```

Una *restricción entrada-salida* es un predicado que relaciona los parámetros de entrada con los de salida de un algoritmo o un trozo de este. Si x son los parámetros de entrada y w los de salida, entonces la restricción entrada-salida es un predicado de la forma:

$$R(x, w)$$

Veamos, primer lugar, como deducir la restricción de entrada-salida de un *bloque básico* y la versión del bloque como una asignación de tuplas. Un bloque básico es una secuencia de sentencias de asignación. Por tanto, en un bloque básico no hay *if, while, for* ni llamadas a métodos. Un primer ejemplo de bloque básico es:

```
a=x;
b=y;
x=b;
y=a;
```

Otro segundo ejemplo es:

```
a=x;
x=y;
y=a;
```

Un tercero:

```
x=y;
a=x;
y=a;
```

Y un cuarto

```
x=y;
y=x;
```

Tenemos que recordar que una variable tiene dos aspectos que debemos tener en cuenta. Por una parte, es una ubicación dónde se puede guardar un valor de un determinado tipo. Por otra parte, es un valor que se guarda en dicha ubicación. La ubicación en memoria de la variable permanece fija pero el valor va cambiando a lo largo de la ejecución de un programa. Si la variable aparece en una expresión en la parte derecha de una asignación diremos que la estamos *usando*. Estamos usando su valor. Si parece en la parte izquierda diremos que la estamos *definiendo*. Estamos dando un nuevo valor a la ubicación en memoria de la variable. Los sucesivos valores que va tomando una variable x los designaremos por $x, x', x'', x''', \ldots$ Cada vez que una variable se define toma un valor nuevo. Es importante recordar que el orden de las asignaciones en un bloque básico es muy importante.

Un bloque básico se puede transformar en una restricción entrada-salida formada por un conjunto de restricciones de igualdad siguiendo las pautas siguientes:

- Comenzamos con un conjunto de restricciones vacío.
- Cada vez que se define variable (está en la izquierda de una asignación) se crea un identificador nuevo para esa variable, que usaremos en adelante, y se añade una restricción de igualdad al conjunto de restricciones.
- Los identificadores nuevos los representamos por $x', x'', x''', \ldots$
- La restricción entrada-salida se obtiene eliminando, en el conjunto de restricciones anterior, los identificadores considerados intermedios.

Una *asignación de tuplas* tiene una restricción de entrada salida asociada que podemos obtener sustituyendo la asignación por igualdad y las variables a la izquierda por sus variables primadas.

Siguiendo esa pauta podemos transformar un bloque básico en una restricción entrada-salida y ésta en una asignación de tuplas. Dos bloques

básicos son equivalentes si tienen la misma restricción entrada salida o lo que es igual tienen asociada la misma asignación de tuplas.

Veamos varios ejemplos en los que deducimos para cada bloque básico su conjunto de restricciones asociado, las restricciones entrada-salida tras eliminar *a* y *b*, considerando *a, b* como variables intermedias a eliminar, y las asignaciones entre tuplas correspondientes:

```
1.
a=x;
b=y;
x=b;
y=a;
_____________
1: a' = x; b' = y; x' = b'; y' = a'; //eliminando a, b
2: x' = y; y' = x';
(x,y) = (y,x);  //asignación de tuplas equivalente
```

```
2.
a=x;
x=y;
y=a;
_____________
1: a' = x; x' = y; y'= a'
2: x' = y; y' = y;
(x,y) = (y,x);   //asignación de tuplas equivalente
```

```
3.
x=y;
a=x;
y=a;
_________________
1: x' = y; a' = x'; y'= a'
2: x' = y; y' = y;
(x,y) = (y,y);  //asignación de tuplas equivalente
```

```
4.
x=y;
y=x;
_________________
1: x' = y; y' = x';
2: x' = y; y' = y;
(x,y) = (y,y);
```

Vemos que los bloques básicos 1 y 2 son equivalentes y también los 2 y 4.

Podemos resumir lo anterior en:

- Todo bloque básico tiene asociado una única restricción entrada-salida
- Todo bloque básico tiene asociado una única asignación entre tuplas
- De una restricción entrada-salida formada por un conjunto de restricciones de igualdad se deduce una única asignación entre tuplas y viceversa.
- Dos bloques básicos son equivalentes si tienen la misma restricción entrada-salida.
- Una asignación entre tuplas, o su restricción entrada-salida equivalente, pueden tener muchos bloques básicos asociados.

Veamos ahora como conseguir un bloque básico equivalente a una asignación entre tuplas. La idea general es usar variables nuevas, asignar a estas variables los valores de las expresiones de la derecha, y, posteriormente, asignar las nuevas variables a las antiguas. Es decir:

```
(x1, x2, …, xm) = (e1, e2, …, em);
====
a1 = e1;
a2 = e2;
…;
am = em;
x1 = a1;
x2 = a2;
…;
xm = am;
```

Posteriormente el bloque básico puede ser simplificado. Algunas reglas de simplificación son:

- *Eliminación de asignaciones y variables intermedias*. Si una variable es definida en un bloque básico, mediante la asignación *u=e(z);* podemos sustituir la variable *u* por *e(z)* en el segmento de bloque básico hasta la siguiente definición de las variables *u* o *z* y eliminar la asignación.
- Una variable que se define y no se usa puede ser eliminada
- Si una variable se define y se vuelve a definir antes de ser usada la primera asignación puede ser eliminada
- *Cambio de orden de asignaciones*. Una secuencia de asignaciones de la forma siguiente puede ser cambiadas de orden:

```
x=f(x,a);
y=g(y,a);
z=h(z,a);
```

- *Cambio de orden de asignaciones*. Una secuencia de asignaciones de la forma siguiente puede ser cambiadas de orden haciendo la sustitución simbólica correspondiente.

```
x=f(x,a);
y=g(x,y,a);
=== Equivalente a
y=g(f(x,a),y,a);
x=f(x,a);
```

Veamos como ejemplo:

```
(x,y)=(y,x);
```

Es equivalente a

```
a = y;
b = x;
x = a;
y = b;
```

Que simplificando el bloque básico resultante obtenemos otro equivalente que es:

```
b = x;
x = y;
y = b;
```

Pero la asignación entre tuplas no es equivalente a:

```
x = y;
y = x;
```

Podemos usar los elementos anteriores para *normalizar un algoritmo* imperativo siguiendo la pauta:

- Convertir los bloques básicos a asignación de tuplas
- Definir las funciones y predicados sobre tuplas necesarios

Un algoritmo iterativo imperativo normalizado tiene la forma:

```
R alg(x) {
       E e = e0(x);
       while(g(e)){
          e = nx(e);
       }
       return r(e);
}
```

Al *e* lo llamamos estado del algoritmo. Donde *e* puede ser una tupla, *s, r* funciones sobre tuplas y *g* un predicado sobre tuplas. El equivalente funcional del anterior algoritmo imperativo normalizado es

```
R alg(x) {
       E e = Stream.iterate(e0(x),x->nx(x))
                   .filter(x->!g(x))
                   .findFirst()
                   .get();
       return r(e);
}
```

En muchos casos es posible separar la tupla en dos partes: acumulador y secuencia. Cuando esto es posible hacemos explícito el acumulador y la secuencia el algoritmo normalizado queda:

```
R alg(x) {
       E e = e0(x);
       B b = b0(x);
       while(g(e)){
              b = c(b,e);
              e = nx(e);
       }
       return r(b,e);
}
```

De la forma iterativa a recursiva final y viceversa

Los algoritmos iterativos tienen siempre algoritmos equivalentes recursivos finales. Aquí mostramos la equivalencia junto a la notación funcional. Como vemos para hacer la transformación es necesario identificar claramente la secuencia, es decir el iterador y el acumulador.

$$S\ s = (e_0, e \to g(e), e \to nx(e))$$
$$A\ a = (b0, (b, e) \to c(b, e), b \to r(b), b \to d(b))$$

Por simplificar la presentación de los esquemas siguientes consideramos que *s* y *a* vienen dados. Como ya hemos dicho una secuencia la podemos implementar mediante un iterador o un stream. Asumimos que tenemos un iterador que implementa la secuencia y también un stream. Podemos obtener ambos de los parámetros iniciales del problema con las funciones *it(x)* e *st(x)* que nos dan, respectivamente un iterador y un stream en su estado inicial. Como ya comentamos un iterador viene definido por un estado de tipo *T*, un valor inicial *t0* y las funciones *hn, next1, next2* que ya hemos estudiado y que tienen una fuerte relación con las que definen la secuencia.

$$It\ it = (() \to t0, t \to hn(t), t \to nx1(t), t \to nx2(t))$$

Haciendo explícito el estado del acumulador tenemos, como ya hemos visto, equivalencias que nos permiten, si hemos identificado la secuencia y el acumulador, obtener las equivalencias imperativas y recursivo final a partir de un algoritmo en notación funcional.

La versión imperativa

```
R alg(X x) {
       T t = it(x);
       B b = b0;
       while(hn(t)&&!f(b)){
          E e = nx1(t);
          t = nx2(t);
          b = c(b,e);   // Si t e mutable b.c(e)
       }
       return r(s(t,b));
}
```

La versión recursivo final

```
R alg(X x) {
	T t = it(x);
	B b = b0;
	B b = alg(t,b);
	return r(b);
}
B alg(T t,B b) {
	if(hn(t)&&!f(b)){
		E e = nx1(t);
		t = nx2(t);
		b = c(b,e); // Si t e mutable b.c(e)
		b = alg(t,b);
	} else {
		b = s(t,b); //En muchos casos s(t,b) = b
	}
	return b;
}
```

La versión funcional

```
R alg(X x) {
	Stream<E> st = st(x);
	return s.collect(a);
}
```

El proceso anterior de combinar una secuencia con un acumulador se le denomina *acumulación* como hemos comentamos arriba. Esta acumulación la llamamos *acumulación por la izquierda***.** Si el acumulador es uno de reducción, ya visto, lo llamaremos *reducción por la izquierda*.

Los esquemas anteriores nos permiten pasar de versiones imperativas de los algoritmos a otras funcionales o recursivas finales o viceversa.

Como ejemplo veamos el algoritmo para calcular el máximo común divisor:

$$mcd(a,b) = \begin{cases} a, & b = 0 \\ mcd(b, a\%b), & a, b > 0 \end{cases}$$

El esquema tiene el equivalente en código

```
Integer mcd(a,b) {
       R r = a;
       if(b>0)
              r = mcd(b,a%b);
       return r;
}
```

Es recursivo final y por lo tanto admite la versión iterativa equivalente.

```
Integer mcd(a,b) {
       while(b > 0){
          (a,b) = (b,a%b)
       }
       return a;
}
```

En este caso el estado del iterador es de tipo *(Integer,Integer)* con valor inicial *(a,b),* la función de guarda es $g(a,b) \equiv b > 0$, la función siguiente $nx(a,b) \equiv (b, a\%b)$ y la función de retorno $r(a,b) \equiv a$.

Ahora tenemos que convertir la asignación de tuplas a un bloque básico. Según lo explicado introducimos las dos variables *a1, b1* y posteriormente eliminamos *a1* de la siguiente manera:

```
a1 = b;
b1 = a%b;
a = b1;
b = a1;

_____
a1 = a;
a = a%b;
b = a1;
```

En las transformaciones anteriores hemos usado las equivalencias entre asignación de tuplas a una secuencia de asignaciones y posteriormente las reglas para simplificar un bloque básico.

Quedando el algoritmo:

```
Integer mcd(a,b) {
       while(b > 0){
          a1 = b;
          b = a%b;
          a = a1;
       }
       return a;
}
```

La versión funcional usa la tupla:

```
record Tm(Integer a, Integer b){
       public static Tm of(Integer a, Integer b){
              return new Tm(a,b);
       }
}
```

```
Integer mcd(Integer a, Integer b) {
       Stream<Tm> st = Stream.iterate(Tm.of(a,b),
              t->Tm.of(t.b(),t.a()%t.b()));
       Tm tr = st.filter(t->t.b()<=0).findFirst().get();
       return tr.a();
}
```

En las transformaciones anteriores hemos usado las equivalencias entre la versión recursiva final y la iterativa usando tuplas y posteriormente la equivalencia de la asignación de tuplas a una secuencia de asignaciones.

Complejidad

Cuando diseñamos algoritmos es necesario demostrar en primer lugar que acaba y hacen el cometido especificado. Pero en segundo lugar es conveniente estimar el tiempo que tardará en ejecutarse en función del tamaño del problema a resolver. El análisis de la complejidad de los algoritmos trata de hacer esa estimación y es lo que vamos a estudiar en este capítulo.

En primer lugar vamos a introducir un conjunto de conceptos y herramientas necesarias para el estudio de la complejidad de un algoritmo.

Órdenes de complejidad

En general estamos interesados en el tiempo de ejecución como una función del tamaño del problema cuando el tamaño es grande. Representaremos el tamaño de un problema por *n*. En general *n* será una función de los valores de las propiedades *x* del problema. Es decir, $n = f(x)$. Representaremos por $T(n)$ la función que nos da el tiempo de ejecución en función del tamaño. En los estudios de complejidad de algoritmos asumimos que todas las funciones T(n) son monótonas crecientes y

normalmente sólo estaremos interesados en los aspectos cualitativos de *T(n),* es decir en su comportamiento para valores grandes de *n*. Para ello clasificamos las funciones según su comportamiento para grandes valores de *n.* Esta clasificación agrupará las funciones *T(n)* en órdenes de complejidad. Cada orden de complejidad es un conjunto de funciones con comportamiento equivalente para grandes valores de *n*.

Para concretar lo anterior introduciremos una relación de orden total entre las funciones. Este orden define implícitamente una relación de equivalencia y unas operaciones de mínimo y máximo asociadas. Representaremos este orden por el símbolo $<_\infty$. Una función h(n) es menor que otra g(n) según este orden cuando el límite de su cociente es cero. Es decir:

$$h(n) <_\infty g(n) \leftrightarrow \lim_{n\to\infty} \frac{h(n)}{g(n)} = 0$$

De la misma manera definimos el concepto de mayor, equivalente, mayor igual y menor igual.

$$h(n) >_\infty g(n) \leftrightarrow \lim_{n\to\infty} \frac{h(n)}{g(n)} = \infty$$

$$h(n) \approx_\infty g(n) \leftrightarrow 0 < \lim_{n\to\infty} \frac{h(n)}{g(n)} < \infty$$

$$h(n) =_\infty g(n) \leftrightarrow \lim_{n\to\infty} \frac{h(n)}{g(n)} = 1$$

$$h(n) \leq_\infty g(n) \leftrightarrow 0 \leq \lim_{n\to\infty} \frac{h(n)}{g(n)} < \infty$$

$$h(n) \geq_\infty g(n) \leftrightarrow \lim_{n\to\infty} \frac{h(n)}{g(n)} > 0$$

Es decir dos funciones son equivalentes en el infinito, que representaremos por $\approx_\infty$, cuando el límite de su cociente es un número mayor que cero y menor que infinito. Podemos definir una noción más fuerte de esta equivalencia, que representaremos por $=_\infty$, para el caso en que el límite del cociente es 1.

La relación de equivalencia anterior define un conjunto de clases de equivalencia. Cada clase de equivalencia es un *Orden de Complejidad Exacto*. Más exactamente, el *orden de complejidad exacto* de una función *g(n),* que representaremos por $\Theta(g(n))$, es el conjunto de funciones que son equivalentes a ella en el infinito según la definición anterior. Así:

$$\Theta(g(n)) = \{ f(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = k,\ 0 < k < \infty\}$$

De entre todas las funciones que forman parte de un mismo orden de complejidad exacto escogemos una de ellas como representante de la clase. Así por ejemplo $\Theta(n)$ representa el conjunto de funciones equivalentes a *n* en el infinito. Entre estas funciones se encuentran los polinomios de primer grado en *n*.

El orden anterior $<_\infty$ definido entre funciones se puede extender a los órdenes de complejidad. La relación de equivalencia entre funciones se convierte en relación de igualdad entre órdenes de complejidad. Cuando quede claro por el contexto usaremos $<$ en lugar de $<_\infty$ y $\approx$ en vez de $\approx_\infty$. También definimos los operadores binarios max_∞ y min_∞ entre órdenes de complejidad *x, y* de la forma:

$$max_\infty(x,y) = \begin{cases} x, & x \geq_\infty y \\ y, & x <_\infty y \end{cases}$$

$$min_\infty(x,y) = \begin{cases} x, & x \leq_\infty y \\ y, & x >_\infty y \end{cases}$$

Igualmente definimos los operadores binarios max_∞ y min_∞ los usaremos indistintamente entre órdenes de complejidad o entre funciones y si, debido al contexto, no hay duda los sustituiremos por *max* y *min*.

A partir de la definición podemos comprobar los siguientes ejemplos:

- $\Theta(n) < \Theta(n^2)$
- $\Theta(n) \approx \Theta(an + b)$ para dos constantes *a,* b positivas cualesquiera
- $\Theta(n) > \Theta(\log_a n)$ para cualquier *a*
- $\Theta(n^a) < \Theta(a^n)$ para cualquier entero *a > 1*
- $\Theta(a^n) \approx \Theta(0) < \Theta(1)$ para cualquier entero *a < 1*

- $\Theta(n^{d+\varepsilon}) > \Theta(n^d log_a^p n)$ para cualquier d > 0, a > 0, p $> 0, \varepsilon > 0$
- $\Theta(n) \approx \max(\Theta(n), \Theta(\log n))$

Para facilidad posterior introducimos el concepto de contante multiplicativa mediante la definición:

$$f(n) =_\infty kg(n) \leftrightarrow \lim_{n\to\infty} \frac{kg(n)}{f(n)} = 1$$

En la definición anterior *f(n)* y *g(n)* son del mismo orden de complejidad y *k* es denominado constante multiplicativa.

Jerarquía de órdenes de complejidad exactos

Como hemos explicado anteriormente los órdenes de complejidad exactos son conjuntos de funciones entre los cuales se puede definir una relación de igualdad y otra de orden. En el análisis de algoritmos hay varios órdenes de complejidad exactos que reciben nombres especiales. Éstos, a su vez, podemos organizarlos en una jerarquía de menor a mayor:

Tabla 1: Clases de Complejidad

- $\Theta(1)$ orden constante
- $\Theta(\log n)$ orden logarítmico
- $\Theta(n)$ orden lineal
- $\Theta(n \log n)$ orden cuasi-lineal
- $\Theta(n^2)$ orden cuadrático
- $\Theta(n^a)$ orden polinómico *(a > 2)*
- $\Theta(2^n)$ orden exponencial
- $\Theta(a^n)$ orden exponencial *(a > 2)*
- $\Theta(n!)$ orden factorial
- $\Theta(n^n)$

Para hacernos una idea intuitiva de la jerarquía de órdenes de complejidad y las relaciones entre ellos veamos la siguiente tabla. En ella se muestra, en primer lugar, el efecto, en el tiempo *T(n)*, de una

duplicación de *n*, el tamaño del problema. Para ello escogiendo las constantes multiplicativas adecuadas se muestran los valores de las *T(n)* para un valor de n=200 siempre que para n=100 el valor sea de 1s. En segundo lugar se muestra el tamaño del problema que puede ser resuelto en un tiempo t = 2s suponiendo que el tiempo necesitado para un problema de tamaño n = 100 es de 1s. La observación más importante es la gran diferencia entre los algoritmos de orden exponencial o mayor y los de orden polinómico o menor.

Tabla 2: Tiempos

T(n)	n=100	n=200	t = 1 s	t = 2 s
k1 log n	1 s	1,15 s	n=100	n=10000
k2 n	1 s	2 s	n=100	n=200
k3 n log n	1 s	2,30 s	n=100	n=178
k4 n^2	1 s	4 s	n=100	n=141
k5 n^3	1 s	8 s	n=100	n=126
k6 2^n	1 s	1,27·1030 s	n=100	n=101

Otros órdenes de complejidad

Junto con el orden de complejidad exacto, $\Theta(g(n))$, se usan otras notaciones $O(g(n))$ (cota superior), $\Omega(g(n))$ (cota inferior). Todos definen conjuntos de funciones:

$$f(n) \in O(g(n)) \quad si \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in \Omega(g(n)) \quad si \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

La notación $O(g(n))$ define un conjunto de funciones que podemos considerar menores o iguales a $g(n)$ en su comportamiento para grandes

valores de *n*. En concreto son las funciones que cumplen que $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ es finito pudiendo ser cero. Por eso se le llama *cota superior* en el sentido de que *g(n)* es mayor o igual que todas las funciones *O(g(n)).*

Como vimos, la notación *Θ(g(n))* define un conjunto de funciones equivalentes a *g(n)* en su comportamiento para grandes valores de *n*. En concreto son las funciones que cumplen que $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ es finito y mayor que cero. Se le llama orden exacto en el sentido de que *g(n)* es *igual* a todas las funciones *Θ(g(n)).*

La notación *Ω(g(n))* define un conjunto de funciones que podemos considerar mayores o equivalentes a *g(n)* en su comportamiento para grandes valores de *n*. En concreto son las funciones que cumplen que el $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ es infinito o finito mayor que cero. Por eso se le llama cota inferior en el sentido de que *g(n)* es menor o igual que todas las funciones *Ω(g(n)).*

Para indicar que una función *f(n)* es de un orden de complejidad dado *g(n)* se indica indistintamente como $O\big(f(n)\big) = g(n)$ *o* $f(n) \in O(g(n))$. Igualmente con las otras notaciones Θ, Ω.

Algunas propiedades de los órdenes de complejidad

Dadas las definiciones anteriores podemos comprobar las siguientes propiedades de los órdenes de complejidad que podemos deducir de las propiedades de los límites:

Tabla 3: Propiedades de los órdenes de complejidad

- $\Theta(ag(n)+b) = \Theta\big(g(n)\big), \Omega(ag(n)+b) = \Omega\big(g(n)\big),$ $\mathrm{O}(ag(n)+b) = \mathrm{O}\big(g(n)\big)$
- $\Theta\big(g(n)\big) = \Omega\big(g(n)\big) \cap \mathrm{O}(g(n))$
- $\Theta\big(g(n)\big) \subset \Omega\big(g(n)\big),\ \Theta\big(g(n)\big) \subset \mathrm{O}(g(n))$
- $f(n) = max_{\infty}\big(f(n)+g(n)\big) \rightarrow \Theta\big(f(n)+g(n)\big) = f(n)$
- $\Theta\big(f(n)*g(n)\big) = \Theta\big(f(n)\big)*\Theta\big(g(n)\big)$
- $\Theta\big(g(n)\big) > 1 \rightarrow \Theta\big(f(n)*g(n)\big) > f(n)$

- $\Theta\big(g(n)\big) = 1 \rightarrow \Theta\big(f(n) * g(n)\big) = f(n)$
- $\Theta\big(g(n)\big) \geq 1 \rightarrow \Theta\left(\frac{f(n)}{g(n)}\right) = \Theta\big(f(n)\big)/\Theta\big(g(n)\big)$
- $\Theta\big(g(n)\big) > 1 \rightarrow \Theta\big(f(n)/g(n)\big) < f(n)$
- $\Theta\big(g(n)\big) = 1 \rightarrow \Theta\left(\frac{f(n)}{g(n)}\right) = f(n)$

Ecuaciones en diferencias: recurrencias lineales

En los cálculos de complejidad aparecen las ecuaciones de recurrencia lineales de la forma:

$$T(n) = aT(n-b) + p(n)$$

Siendo $p(n)$ un polinomio de grado d. Las ecuaciones con la forma anterior se denominan ecuaciones en diferencias lineales o recurrencias lineales. Si la parte de la derecha es cero se denominan homogéneas.

La ecuación característica generalizada asociada a esa ecuación de recurrencia es:

$$\big(x^b - a\big)(x-1)^{d+1} = 0$$

De manera más general tenemos la ecuación de recurrencias lineal

$$T(n) = a_1T(n-1) + a_2T(n-2) + \ldots + a_rT(n-1) + p(n)$$

Que tienen una ecuación característica generalizada asociada:

$$(x^r - a_1x - a_2x^2 - \ldots . - a_r)(x-1)^{d+1} = 0$$

En ambos caos el orden de complejidad de las soluciones es

$$\Theta(n^{m-1}r^n)$$

Donde r es la raíz real de mayor valor absoluto de la ecuación característica asociada y m su multiplicidad.

Para el primer caso las raíces reales de la ecuación son $1, a^{\frac{1}{b}}$. El resto de raíces de $(x^b - a) = 0$ son complejas. Si $a = 1$, la raíz de mayor valor es 1 y tiene de multiplicidad $d + 2$. Si $a < 1$ la raíz de mayor valor es 1 y su multiplicidad es $d + 1$. Por último cuando $a > 1$ la raíz mayor es a y la multiplicidad es 1. Aplicando la regla dada arriba tenemos, según sea a mayor, igual o menor a uno:

$$T(n) = \begin{cases} \Theta(a^{n/b}), & si \quad a > 1 \\ \Theta(n^{d+1}), & si \quad a = 1 \\ \Theta(n^d), & si \quad a < 1 \end{cases}$$

Una variante del caso anterior es la recurrencia

$$T(n) = aT(n-b) + n^d (log\, n)^p$$

Observando que $(\log n)^p \cong (\log(n-b))^p$ podemos ver que la solcuión tendrá la forma $h(n)(\log n)^p$. Donde $h(n)$ verifica la ecuación previa. La solución es entonces

$$T(n) = \begin{cases} \Theta(a^{n/b}(\log n)^p), & si \quad a > 1 \\ \Theta(n^{d+1}(\log n)^p), & si \quad a = 1 \\ \Theta(n^d(\log n)^p), & si \quad a < 1 \end{cases}$$

Usualmente aparecen recurrencias lineales de la forma:

$$a_0 T(n) + a_1 T(n-1) + \cdots + a_k T(n-k) = p(n)$$

Estas la reducimos al caso anterior. Veamos un ejemplo que es fácilmente generalizable. En el problema de Fibonacci, $f(n) = f(n-1) + f(n-2)$, la complejidad de la solución viene dada, como hemos visto anteriormente, por $T(n) = T(n-1) + T(n-2) + k$.

Las soluciones de esa recurrencia pueden será acotadas por las soluciones de las recurrencias mostradas abajo cuyas soluciones mostramos:

$$T(n) = 2T(n-1) + k, \qquad \Theta(a^n)$$
$$T(n) = 2T(n-2) + k, \qquad \Theta(a^{n/2})$$

Concluimos que $\Theta\left(\sqrt{2}^{n}\right) < \Theta(T(n)) < \Theta(2^n)$.

La solución exacta, como vimos arriba, es $\Theta((\frac{1+\sqrt{5}}{2})^n)$ que cae dentro de las cotas establecidas.

La complejidad exacta de la ecuación original las podemos encontrar. En efecto su ecuación característica generalizada es

$$(x^2 - x - 1)(x - 1)$$

Cuya solución real de mayor valor es $\frac{1+\sqrt{5}}{2} \approx 1.62$ y la complejidad $\Theta((\frac{1+\sqrt{5}}{2})^n) = \Theta(1.62^n)$. Podemos comprobar que está acotada por las soluciones encontradas antes $\Theta\left(\sqrt{2}^{n}\right) <_\infty \Theta((\frac{1+\sqrt{5}}{2})^n) <_\infty \Theta(2^n)$.

En la mayoría de los casos nos bastará con las cotas para la complejidad.

Ecuaciones recurrencia solubles por el Teorema Maestro (Master Theorem)

Suponiendo que un problema de tamaño *n* se divide en *a* subproblemas, todos del mismo tamaño *n/b,* y que el coste de dividir el problema en subproblemas más el coste de combinar las soluciones es *g(n)* entonces la ecuación de recurrencia será para los casos recursivos ($n > n_0$) :

$$T(n) = aT\left(\frac{n}{b}\right) + g(n), \qquad n > n_0, a \geq 1, b > 1, g(n) > 0$$

Este tipo de ecuaciones de recurrencia, a diferencia de las del apartado anterior, los argumentos de T están relacionados mediante factores multiplicativos. Son denominadas *q-difference equations*.

Un caso particular de mucha utilidad es cuando $g(n) \in \Theta\left(n^d log^p n\right)$, en cuyo caso tenemos

$$T(n) = \begin{cases} \Theta\left(n^{log_b a}\right) & si \quad a > b^d \\ \Theta\left(n^d \, log^{p+1} n\right) & si \quad a = b^d \\ \Theta\left(n^d \, log^p n\right) & si \quad a < b^d \end{cases}$$

Haciendo $p = 0$ en la tabla anterior obtenemos un caso aún más particular para cuando $g(n) \in \Theta(n^d)$. En este caso tenemos

$$T(n) = \begin{cases} \Theta(n^{log_b a}) & si \quad a > b^d \\ \Theta(n^d \ log\, n) & si \quad a = b^d \\ \Theta(n^d) & si \quad a < b^d \end{cases}$$

Como en las recurrencias lineales también podemos encontrar cotas superiores e inferiores con la misma técnica. Imaginemos que

$$f(n) = a1f(n/b1) + a2f(n/b/2) + \ \Theta(n^p log^d n)$$

Las soluciones de esa recurrencia pueden será acotadas por las soluciones de las recurrencias mostradas abajo cuyas soluciones mostramos:

$$T(n) = (a1 + b1)T(n/b1) + \Theta(n^p log^d n)$$
$$T(n) = (a1 + b1)T(n/b2) + \Theta(n^p log^d n)$$

Sumatorios y recurrencias

Como consecuencia de la transformación recursiva lineal final en iterativa podemos obtener relaciones entre algunos sumatorios y algunas ecuaciones de recurrencia. En el caso de un sumario cuyo índice recorre una expresión aritmética:

$$T(n) = T(n-b) + n^d(\ln n)^p \rightarrow \sum_{x=i_0+ri}^{n} n^d(\ln n)^p \cong T(n)$$

Y de aquí concluir que el orden de complejidad de la solución de

$$T(n) = T(n-b) + n^d(\ln n)^p$$

Es

$$\Theta(T(n)) = n^{d+1}\,(ln\, n)^p$$

En el caso de un sumatorio cuyo índice recorre una expresión geométrica y como consecuencia de la transformación recursiva lineal final en iterativa tenemos:

$$T(n) = T(n/b) + n^d (ln\, n)^p \rightarrow \sum_{x=i_0 r^i}^{n} f(x) \cong T(n)$$

Podemos aplicar el *Master Theorem* con $a = 1$, $f(n) = n^d (\ln\, n)^p$, resultando

$$\sum_{x=i_0\, r^i}^{n} x^d\, (ln\, x)^p \cong \begin{cases} (ln\, n)^{p+1}, & d = 0 \\ n^d\, (ln\, n)^p, & d > 0 \end{cases}$$

Hay otros sumatorios que aparecen en ocasiones:

$$\sum_{i=0}^{n} a^i = \begin{cases} 1, & a < 1 \\ a^n, & a \geq 1 \end{cases}$$

$$\sum_{i=0}^{log_r\, n} a^i \cong a^{log_r\, n} \cong n^{log_r\, a}$$

Complejidad de los algoritmos

Los problemas que vamos a resolver se agrupan en conjuntos de problemas. Cada problema tendrá unas propiedades *x*. Cada propiedad específica la representaremos mediante un superíndice: $x = x^1, \ldots, x^k$. Dentro de un conjunto de problemas los valores de sus propiedades identifican al problema de manera única.

Asociado a un problema podemos asociar el concepto de tamaño que es una nueva propiedad derivada del problema. El tamaño de un problema es una medida de la cantidad de información necesaria para representarlo. Normalmente representaremos el tamaño de un problema mediante *n* y lo calcularemos, mediante una función $n = t(x)$. Por lo tanto, cada problema, dentro de un conjunto de problemas, tendrá un tamaño.

En los algoritmos recursivos podemos entender que cada llamada recursiva resuelve un problema distinto y el tamaño de cada uno de los subproblemas debe ser menor que el tamaño del problema original. Por analogía, los algoritmos iterativos van transformando un problema en otro, también de tamaño más pequeño, hasta que se encuentra la solución.

Dado un conjunto de problemas y un algoritmo para resolverlos el tiempo que tardará el algoritmo para resolver un problema dado dependerá del tamaño de este $n = t(x)$. El tiempo que tarda el algoritmo en función del tamaño del problema que resuelve lo representamos por la función $T(n)$.

Complejidad de los algoritmos iterativos y recursivos sin memoria

Un algoritmo se compone de secuencia de bloques que clasificaremos en tres tipos:

- *Bloque básico*: una secuencia de asignaciones
- *Bloque básico con llamada a función*: uno con asignaciones y llamadas a función pero sin *if* ni *while*.
- *Bloque while*: Una sentencia while con un cuerpo de alguno de los tres tipos.

El flujo de control de la ejecución de un algoritmo se define un camino. Un camino es una secuencia de bloques básicos, bloques *con* llamadas a función o bloques while. El camino viene definido por las sucesivas ramas de los bloques *if* elegidas según los parámetros de entrada. Para un problema dado y unas propiedades de entrada algoritmo fijadas, el camino recorrido por algoritmo es único. El coste de ejecutar un algoritmo para resolver un problema es el coste de ejecutar los bloques en el camino recorrido. Por lo tanto la suma del coste de los bloques que lo forman. Veamos cada uno de ellos.

Asumimos que las propiedades del problema son x, su tamaño $n = t(x)$ y el coste de ejecutarlo $T(n)$. Es decir calculamos el coste en función del tamaño del problema. Recordemos que estamos interesados en el orden de complejidad exacto, es decir $\Theta(T(n))$, o alternativamente $O(g(n))$ o $\Omega(g(n))$, pero no en el coste $T(n)$ directamente.

Complejidad de un bloque básico.

Como hemos dicho un *bloque básico* es una secuencia de asignaciones sin *if,* sin *while* y sin llamadas a funciones. Su tiempo de ejecución es constante independientemente del tamaño del problema. Su complejidad es, por tanto, $\Theta(1)$.

Como hemos dicho un *bloque con llamada a función es* una secuencia de asignaciones y llamada a funciones, pero sin *if* ni *while*. Su tiempo de ejecución es la suma de los tiempos de cada una de las sentencias. La llamada a función tardará un tiempo que dependerá de los parámetros. En definitiva, dependerá del tamaño del problema que resuelve obtenido de los parámetros.

Complejidad del bloque while. La estructura de un bloque *while* se puede escribir en la forma:

```
while(g){
   s;
}
```

Donde g es una expresión lógica y s un bloque de código de alguno de los tipos anteriores. El boque *while* es el elemento central en todo algoritmo iterativo. A cada bloque while podemos asociar una variable entera i que va contando el número de iteraciones. Esta variable se va a mover en algún tipo de secuencia (aritmética, geométrica, etc.) que llamaremos I. En cada iteración del bucle el algoritmo transforma un problema con propiedades x_i en otro con propiedades x_j hasta llegar a un problema cuyas propiedades no satisfagan la guarda. Cada uno de los sucesivos problemas tiene asociado un tamaño $t(x_i)$ que debe ser cada vez más pequeño para que el algoritmo acabe.

El tiempo de ejecución de un bloque while dependerá del número de iteraciones que tenga. La guarda más el cuerpo de la forma un bloque de alguno de los tres tipos anteriores. Podemos calcular su tiempo de ejecución como hemos ido explicando. En tiempo de ejecución de la iteración i, ejecución de la guarda más el bloque s, es $T_s(i)$. Con estas ideas

El tiempo de ejecución del bucle *while* es entonces la suma del tiempo de sus ejecuciones:

$$\Theta(T(n)) = \Theta(\sum_{i \in I} T_s(i))$$

1. *Ejemplo de cálculo de complejidad*

```
for (int i = 1; i <= n; i++) {
    r += a * i;
}
```

$$\sum_{i=1}^{n} k = k\,n \ \in \Theta(n)$$

Dependiendo del problema concreto a resolver, en definitiva, de las condiciones iniciales del algoritmo, se seguirá un camino de ejecución u otro. Estos caminos dependen del problema concreto y como puede haber diferentes problemas con el mismo tamaño, pero diferentes propiedades pueden ocurrir que para diferentes problemas del mismo tamaño se sigan caminos diferentes. Distinguimos, entonces, el *caso mejor*, el *caso peor* y el *caso medio* a la hora de calcular la complejidad de un algoritmo.

- El caso mejor es el que ejecuta el camino de complejidad más baja
- El caso peor es el que ejecuta el camino de complejidad más alta
- El caso medio es el que ejecuta todos los caminos posibles cada uno de ellos con una probabilidad que tendremos que establecer

Como segundo ejemplo sea una lista de números enteros de tamaño n y los algoritmos siguientes que pretenden encontrar si existe algún múltiplo de s.

```
boolean contieneMultiploDe(List<Integer> ls, Integer s) {
       return contieneMultiploDe3(0,ls,r);
}
boolean contieneMultiploDe(Integer i, List<Integer> ls,
              Integer s) {
       boolean r;
       if(lis.get(i)%s==0) r= true;
       else r = contieneMultiploDe(i+1,ls);
       return r;
}
```

Observando el código del algoritmo generalizado vemos que el tamaño es $n - i$. Siendo n el tamaño de la lista. Podemos ver, también, que hay dos caminos posibles según la guarda $ls.get(i)\%s == 0$ sea verdadera o falsa.

Aquí podemos estudiar el caso mejor, pero y medio.

- Caso mejor: El primer elemento es múltiplo de r. Complejidad $\Theta\big(T(n)\big) = \Theta(1)$
- Caso peor: No hay múltiplo de r. Complejidad $T(n) = T(n-1) + 1$ cuya solución es $\Theta(n)$
- Complejidad del caso medio. Asumiendo que la probabilidad de que un número entero sea múltiplo de s es $1/s$ entonces la complejidad verifica $T(n) = \frac{s-1}{s}T(n-1) + \frac{1}{s}$. Cuya solución es $\Theta(1)$.

Como $\frac{s-1}{s} < 1$ y recordando el tipo de recurrencias a aplicar en este caso

$$T(n) = aT(n-b) + (log\, n)^p\, \Theta(n^d)$$

Cuya solución es: $T(n) = \begin{cases} \Theta\big(a^{n/b}(\log n)^p\big), & si \quad a > 1 \\ \Theta\big(n^{d+1}(\log n)^p\big), & si \quad a = 1 \\ \Theta\big(n^d(\log n)^p\big), & si \quad a < 1 \end{cases}$

El algoritmo tiene una versión secuencial siendo su complejidad igual:

```
boolean contieneMultiploDe(List<Integer> ls, Integer s) {
	boolean r = false;
	for(Integer e: lis){
		r = e%s==0;
		if(r) break;
	}
	return r;
}
```

- Caso mejor: El primer elemento es múltiplo de 3. Hay cero iteraciones. Complejidad $\Theta\big(T(n)\big) = \Theta(1)$
- Caso peor: No hay múltiplo de 3. Hay n iteraciones. Complejidad

$$\Theta\big(T(n)\big) = \Theta(\sum_{i=0}^{n-1} 1)) = \Theta(n)$$

- Complejidad del caso medio. Puede haber de 0 a n iteraciones. Cada iteración tiene coste constante pero la probabilidad de que ocurra es que no se haya encontrado en las iteraciones anteriores es decir $\left(\frac{s-1}{s}\right)^i$. Luego la complejidad del caso medio es $\Theta(\frac{1}{s}\sum_{i=0}^{n-2}(\frac{s-1}{s})^i)) = \Theta(1)$

Complejidad de los algoritmos recursivos con memoria

En los algoritmos con memoria el cálculo de la complejidad es distinto. Ahora cada problema se resuelve una vez y debemos tener en cuenta el conjunto de problemas P que debemos resolver para encontrar la solución de uno dado. Si $T(p)$ es el tiempo que tarda el problema p en resolverse asumiendo constante las llamadas a los subproblemas entonces la complejidad de un problema es

$$\Theta(\sum_{p\in P} T(p))$$

2. Ejemplo: Problema de Fibonacci

En el problema de *Fibonacci* cada problema es representado por su propiedad *n.* Podemos comprobar que el conjunto de problemas a resolver es $P = \{0,1,2, \dots, n\}$. Según lo explicado antes $T(p) = k$. Luego la complejidad del problema de Fibonacci resuelto con memoria es $\sum_{p \in P} k = k \sum_{p \in P} 1 \cong n$.

En el problema de la Potencia Entera cada subproblema se representa por su propiedad n. Ahora $P = \{n, \frac{n}{2}, \frac{n}{4}, \dots, 1\}$ y $|P| = log_2 n + 1$. Por lo tanto la complejidad de la *Potencia Entera con Memoria* es $\sum_{p \in P} k = k \sum_{p \in P} 1 \cong log\, n$.

Veremos otros ejemplos en el apéndice.

Estas ideas nos permiten decidir cuándo usar Divide y Vencerás con y sin Memoria. Sólo usaremos la Memoria cuando la complejidad del problema se reduzca (caso del problema de *Fibonacci)*. Si la complejidad es la misma (caso del problema de la *Potencia Entera*) no es conveniente usar *Memoria*. Las razones para que la complejidad sea distinta (usando memoria y sin usarla) es la aparición de subproblemas repetidos al resolver un problema. Esto ocurre en el problema de *Fibonacci* y no ocurre en el Problema de la *Potencia Entera. Por lo tanto, un criterio directo para decidir si usar memoria o no es verificar si aparecen subproblemas repetidos o no.*

Fórmulas de resumen

- Expresiones importantes con sumatorios

$$\sum_{i=0}^{n} a^i = \begin{cases} 1, & a < 1 \\ a^n, & a \geq 1 \end{cases}$$

$$\sum_{i=0}^{log_r n} a^i \cong a^{log_r n} \cong n^{log_r a}$$

$$\sum_{x=i_0+r\,i}^{n} x^d\,(ln\,x)^p \cong \frac{1}{r(d+1)}\,n^{d+1}\,(ln\,n)^p$$

$$\sum_{x=i_0\,r^i}^{n} x^d\,(ln\,x)^p \cong \begin{cases} (ln\,n)^{p+1}, & d=0 \\ n^d\,(ln\,n)^p, & d>0 \end{cases}$$

- Recurrencias lineales y su solución

$$T(n) = aT(n-b) + \Theta(n^d)log^p n$$

$$T(n) = \begin{cases} \Theta\left(a^{n/b}log^p n\right), & si \quad a>1 \\ \Theta\left(n^{d+1}log^p n\right), & si \quad a=1 \\ \Theta\left(n^d log^p n\right), & si \quad a<1 \end{cases}$$

- Recurrencias no lineales y su solución

$$T(n) = aT(n/b) + \Theta(n^d)log^p n$$

$$T(n) = \begin{cases} \Theta\left(n^{log_b a}\right) & si \quad a > b^d \\ \Theta\left(n^d\,log^{p+1} n\right) & si \quad a = b^d \\ \Theta\left(n^d\,log^p n\right) & si \quad a < b^d \end{cases}$$

3. *Un ejemplo*

```
double f (int n, double a) {
    double r;
    if (n == 1) {
        r = a;
    } else {
        r = f (n/2, a+1) - f (n/2, a-1);
        for (int i = 1; i <= n; i++) {
                r += a * i;
         }
    }
    return r;
}
```

La parte iterativa, según el Ejemplo 1, es $\Theta(n)$. Por lo tanto:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Cuya solución es: a= 2, b= 2, d =1, $T(n) \in \Theta(n \log n)$

4. Un segundo ejemplo es

```
int f (int n) {
    int i, j, z, r;
    if (n < 1) {
        r = 1;
    } else {
        z = 0;
        for (i = 1; i < n; i++) {
           for (j = 1; j < i * i; j++) {
                z ++;
           }
        }
        r = z * (( f (n - 2)) ^ 2);
    }
    return r;
fin
```

Tomando el resultado del ejemplo 5 tenemos:

$$T(n) = T(n-2) + \Theta(n^3)$$

Con a *= 1, b= 2, d = 3*, tenemos:

$$T(n) \in \Theta(n^4)$$

En el caso anterior si la expresión *f (n – 2)) ^ 2* la escribiéramos de la forma *f (n – 2))* f (n – 2))* la complejidad sería

$$T(n) = 2T(n-2) + \Theta(n^3)$$

Cuya solución es

$$T(n) \in \Theta\left(2^{\frac{n}{2}}\right) = \Theta\left(\sqrt{2}^{n}\right)$$

En ambos casos si utilizamos memoria la complejidad es

$$\sum_{i=2u}^{n} i^3 \cong \Theta(n^4)$$

En este caso asumimos que los subproblemas están calculados. Entonces cada problema tiene un coste de $\Theta(i^3)$ y los subproblemas que es necesario calcular están en una secuencia aritmética de paso 2.

Diseño y verificación de algoritmos iterativos

Las ideas vistas previamente nos van a permitir usar algunas estrategias para diseñar algoritmos iterativos. Hay varias posibilidades que vamos a ver ahora. Dejamos para más adelante la posibilidad de diseñar, con estrategias recursivas, un algoritmo recursivo final para obtener a partir de él uno iterativo en forma imperativa o funcional.

Diseño de Algoritmos Iterativos

Hay varias estrategias

- *Buscar un estado, una secuencia*, con su iterador asociado, *un acumulador secuencial y combinarlos* en un algoritmo iterativo. A veces *una función de partición* que a partir de un problema obtenga otro más pequeño puede ayudarnos a definir una secuencia formada de problemas y subproblemas
- *Buscar un estado y sobre él definir un invariante*. Un invariante es un predicado que debe cumplirse en todos los estados que va recorriendo el algoritmo. Dado el estado y el invariante se trata de encontrar la secuencia cuyos estados cumplen el invariante.

Veamos la primera estrategia.

El paso de buscar un estado lo llamamos *generalizar el problema*. Generalizar consiste, en definitiva, en imaginar propiedades que nos permitan definir un estado. Teniendo en estado debemos escoger un valor inicial, una guarda y una función siguiente para definir una secuencia.

La secuencia elegida puede ser obtenida mediante operaciones de filtro o transformación, etc., a partir de una secuencia básica y hay que demostrar que es finita. Para demostrar que la secuencia es finita es conveniente introducir el concepto de *tamaño*: función del estado que devuelve un entero positivo o cero que debe disminuir en cada paso de la secuencia.

A partir de la secuencia, que es un concepto abstracto, debemos escoger un iterador y un stream que la implementen.

Combinando una secuencia y un acumulador obtenemos un algoritmo iterativo. Para obtener la versión funcional el método *s.collect(a)* de Java hace el trabajo, pero asumiendo que s es de tipo *Stream<E>* y a de tipo *Collector<E,B,R>*. El tipo *Collector<E,B,R>* representa un acumulador que acumula valores de tipo *E* en una base de tipo *B* y con un valor de retorno de *R*. Por ahora sólo consideraremos los aspectos secuenciales de los tipos *Stream<E> y Collector<E,B,R>*. Más adelante veremos los aspectos paralelos. Recordemos que Java aporta, con Collectors, una amplia factoría de acumuladores de tipo *Collector<E,B,R>.*

Anteriormente ya hemos visto como concretar la versión imperativa partiendo de un iterador y un acumulador secuencial.

Veamos un ejemplo de esta estrategia.

Dado un fichero con varios números enteros por línea separados por comas sumar los que sean primos

Aquí para definimos una secuencia partiendo de un agregado de datos, un fichero en este caso, y aplicándole varias transformaciones: *flatMap, map y filter*. El acumulador elegido es *sum*.

La primera versión que haremos será la funcional:

```
Integer sumaPrimos(String file) {
       Intstream st = Streams.file(file)
                     .flatMap(e->Streams.split(e,[ ,”]);
                     .mapToInt(e->Integer.parseInt(e))
                     .filter(e->Math2.esPrimo(e));
       return       st.sum()
}
```

La versión imperativa requiere un iterador para recorrer el fichero y otro para recorrer las partes en las que se divide cada línea que es la forma de traducir *flatMap* a la notación imperativa. En cada lenguaje estos iteradores tomarán una forma particular. En Java sería:

```
Integer sumaPrimos(String file) {
       Iterator<String> fileIt =
       Iterables.file(file).iterator();
       Integer suma = 0;
       while(fileIt.hasNext()){
              String linea = fileIt.next();
              Iterator<String> lineaIt =
                     Iterables.split(linea,"[ ,]").iterator();
              while(lineaIt.hasNext()){
                     String p = lineaIt.next();
                     Integer e = Integer.parseInt(p);
                     if(Math2.esPrimo(e)){
                            suma = suma + e;
                     }
              }
       }
       return suma;
}
```

Podemos escribir el código con *for extendido* para recorrer iterables en este caso.

```
Integer sumaPrimos(String file) {
       Integer suma = 0;
       for(String linea: Iterables.file(file)) {
              for(String e: Iterables.split(linea, "[ ,]")) {
                     Integer en = Integer.parseInt(e);
                     if (Math2.esPrimo(en)) {
                            suma = suma + en;
                     }
              }
       }
       return suma;
}
```

Por último, podemos plantear la versión recursiva final con un iterador construido que compuesto a partir del iterador del fichero y el *flatMap* y un acumulador. La implementación de estos iteradores compuestos puede encontrarse en el repositorio.

```
Integer sumaPrimos3(String file) {
   Iterable<String> fileIt = Iterables.file(file);
   Iterable<String> ff = Iterables.flatMap(fileIt,
              linea->Iterables.split(linea,"[ ,]"));
   return sumaPrimos3(ff.iterator(),0);
}
Integer sumaPrimos3(Iterator<String> ff, Integer b) {
   if(ff.hasNext()) {
        String p = ff.next();
        Integer en = Integer.parseInt(p);
        if (Math2.esPrimo(en)) {
             b = b + en;
        }
        b = sumaPrimos3(ff,b);
    }
    return b;
}
```

La segunda estrategia es *buscar un estado*, lo que hemos llamado *generalización y sobre él definir un invariante*. Un invariante es un predicado que debe cumplirse en todos los estados que va recorriendo el algoritmo. Dado el estado y el invariante se trata de encontrar la secuencia cuyos estados cumplen el invariante

Como ejemplo veamos el algoritmo de la *búsqueda binaria*. Este problema consiste en, dada una lista ls, ordenada con respecto a un orden, encontrar, si existe, la posición de un elemento dado o -1 si no lo encuentra.

Partimos de la lista ordenada ls de tamaño n y un elemento e a buscar. El primer paso es escoger un estado. Es lo que llamamos *generalizar el problema*. Escogemos un estado representado por la tupla (i, j, k) con $i \in [0, n]$, $j \in [0, n], k \in [0, n]$ y siendo $n = ls.size()$.

En segundo lugar, escogemos un invariante:

$e \in ls[i, j] \;||\; e \notin ls,\ i \leq j,\ k = \frac{i+j}{2}$.

Es decir aseguramos que el elemento buscado está en la sublista $ls[i, j]$ o no está en la lista completa. El tamaño del problema(o función de cota) será $j - i$. El segundo paso es escoger el estado inicial, la guarda y la función siguiente para que se mantenga el invariante y la función de cota cumpla las propiedades exigidas.

Escogemos como estado inicial $(0, n, n/2)$.

La función que nos indica si hay siguiente es

$$hn(i, j, k) = j - i > 0 \ \wedge\ e \neq ls[k]$$

Escogemos como función siguiente (se llamará sólo si hay siguiente

$$nx(i, j, k) = \begin{cases} (i, k, \frac{i+k}{2}), & e < ls[k] \\ (k+1, j, \frac{k+1+j}{2}), & e > ls[k] \end{cases}$$

La intuición sobre la función siguiente es que si el elemento a buscar no es igual al que está en la posición k, la lista no está vacía y es menor que el que está en la posición k debemos buscar en la mitad de la izquierda y en otro caso en la mitad de la derecha. Así se mantiene el invariante porque si e está en la lista estará en la sublista escogida.

Con esos elementos planteamos el algoritmo de la forma:

```
Integer bb(ls,e) {
       n = |ls|;
       (i, j, k) = (0, n, n/2);
       while(j-i >0 && e != ls[k]){
       if(e < ls[k]){
                     (i,j,k) = (i,k,(i+k)/2);
              } else {
                     (i,j,k) = (k+1,j,(k+1+j)/2);
              }
       }
       return j-i>0?k:-1;
}
```

Podemos comprobar que se cumple el invariante y las condiciones sobre la función de cota. Las condiciones al terminar el bucle implican el invariante más la negación de guarda. Es decir

$$(e \in ls[i,j] \;||\; e \notin ls) \wedge (j - i = 0 \;||\; e = ls[k])$$

De aquí podemos deducir que si $j - i > 0$ entonces $e = ls[k]$ y el resultado k. Por otra parte, si $j - i = 0$ se debe cumplir $e \notin ls$ y resultado debe ser -1.

Un esquema de la versión recursiva final es:

```
Integer bb(ls,e) {
       n = |ls|;
       (i, j, k) = (0, n, n/2);
       (i,j,k) = bb(i,j,k,ls,e);
       return j-i>0?k:-1;
}
(Integer,Integer,Integer) bb(i,j,k,ls,e) {
       if(j-i >0 && e != ls[k]){
              if(e < ls[k]){
                     (i,j,k) = (i,k,(i+k)/2);
              } else {
                     (i,j,k) = (k+1,j,(k+1+j)/2);
              }
              (i,j,k) = bb(i,j,k,ls,e);
       }
       return (i,j,k);
}
```

Y la versión funcional

```
T =(Integer, Integer, Integer, List<E>, E)
so = (0,n,n/2,ls,e)
hn = (i,j,k,ls,e) -> j-i >0 && e != ls[k];
hnn = (i,j,k,ls,e) -> j-i ==0 || e == ls[k];
nx = (i,j,k,ls,e) -> if(e < ls[k]) (i,k,(i+k)/2,ls,e)
                                 else (k+1,j,(k+1+j)/2,ls,e);
rt = (i,j,k,ls,e) -> j-i>0?k:-1;

Integer bb(ls,e){
       Stream() st = Stream.iterate(so,x->true,nx);
       T r = st.filter(hnn).findFirst().get();
       return rt(r);
}
```

El algoritmo anterior se le suele llamar *búsqueda dicotómica*. Tiene muchos usos. Uno en particular es encontrar la solución de la ecuación $f(x) = k$ suponiendo que f es una *función monótona* y se ha especificado un rango de búsqueda $[x_0, x_1]$.

Una función es $f(x)$ es monótona creciente si $x_1 < x_2 \rightarrow f(x_1) \leq f(x_2)$ y monótona decreciente cuando $x_1 < x_2 \rightarrow f(x_1) \geq f(x_2)$. Para que podamos definir una función como monótona debe existir una relación de orden en su dominio e imagen. El algoritmo puede ser ajustado para cuando x toma valores discretos y buscamos la solución más cercana si no existe la exacta, etc.

Un ejemplo complejo de esta segunda estrategia, generalización más invariante, es el algoritmo de la bandera holandesa que usaremos posteriormente para diseñar algoritmos de ordenación.

El problema de la *bandera holandesa* se enuncia así:

Dada una lista y un elemento del mismo tipo que las casillas, que llamaremos pivote, reordenarla, de menor a mayor, para que resulten tres bloques: los menores que el pivote, los iguales al pivote y los mayores que el pivote. El algoritmo debe devolver dos enteros que son las posiciones de las casillas que separan los tres bloques formados.

Partimos de la lista ls de tamaño n y pivote p. Escogemos el estado formado por la tupla (a, b, c) y el invariante que asegura que en la sublista $ls[0, a]$ todos los elementos son más pequeños que el pivote, en la sublista

$ls[a,b]$ iguales al pivote, en la sublista $ls[c,n]$ mayores al pivote y en $ls[b,c]$ la relación de los elementos con el pivote es desconocida.

Tabla 4: Invariante de la bandera holandesa

0	*a*	*b*	*c*	*n*
<p	*=p*	*?*	*>p*	

El tamaño del problema será $c - b$. Iremos reduciendo el tamaño de la zona desconocida, el tamaño, en cada iteración. La función existe siguiente, la guarda, es $hn(a,b,c) = c - b > 0$. Con los elementos anteriores se propone se siguiente algoritmo:

```
bh(ls, p){
       n = |ls|;
       (a,b,c) = (0,0,n);
       while(c-b>0){
              if(ls[b] < p ){
                  it(a,b,ls);
                  a++;
                  b++;
              }else if(ls[b] == p){
                  b++;
              }else{
                  it(b,c-1,ls);
                  c--;
              }
       }
       return (a,b);
}
```

Donde la función *it* intercambia los valores de las dos casillas a, b y es de la forma:

```
it(a,b,ls){
       (ls[a],ls[b]) = (ls[b],ls[a]);
}
```

Podemos comprobar que la función de cota se reduce en cada iteración. Veamos que se mantiene el invariante. Hay tres casos dependiendo de la relación del elemento *ls[b]*, el primero de la zona desconocida, con el pivote *p*:

- *ls[b] < p*. Reubicamos el elemento en la casilla *b* en la zona *[0,a)* intercambiando el elemento en *b* por el que hay en *a,* e incrementando los valores de *a, b.*
- *ls[b] = p*. Reubicamos el elemento en la casilla *b* en la zona *[a,b)* simplemente aumentando *b*.
- *ls[b] > p*. Reubicamos el elemento en la casilla *b* en la zona *[c,n)* intercambiando el elemento en *b* por el que hay en *c-1* y decrementando el valor de *c*.

Cada vez que ubicamos un elemento disminuye $b - c$ y por lo tanto el tamaño del problema. Como vemos el acumulador va produciendo cambios en la lista.

Un esquema de la versión recursiva final es de forma análoga:

```
(Integer, Integer) bh(ls, p){
       n = |ls|;
       (a,b,c) = (0,0,n);
       (a,b,c) = bh(a,b,c,ls,p);
       return (a,b);
}
```

```
(Integer, Integer, Integer) bh(a,b,c,ls,p){
       if(c-b>0){
              if(ls[b] < p ){
                  it(a,b,ls);
                  a++;
                  b++;
              }else if(ls[b] == p){
                  b++;
              }else{
                  it(b,c-1,ls);
                  c--;
              }
            (a,b,c) = bh(a,b,c,ls,p);
       }
       return (a,b,c);
}
```

Y finalmente la versión funcional asumiendo que la función *it* devuelve la lista modificada:

```
T = (Integer, Integer, Integer, List<E>, E)
n = ls.size();
s0 = (0,0,n,ls,p);
hn = (0,0,n,ls,p) -> c-b>0;
hnn = (0,0,n,ls,p) -> c-b==0;
nx = (a,b,c,ls,p) -> if(ls[b] < p ) (a++,b++,c, it(a,b,ls),p)
             else if(ls[b] == p) (a,b++,c,ls,p)
             else (a,b,c--,it(b,c-1,ls),p);
rt = (a,b,c,ls,p) -> (a,b);
(Integer,Integer( bb(ls,e){
Stream(T) st = Stream.iterate(so,x->true,nx);
T r = st.filter(hnn).findFirst().get();
return rt(r);
}
```

Una variante de este algoritmo es la siguiente:

Dado un array y un elemento del mismo tipo que las casillas, que llamaremos pivote, reordenarlo para que queden dos bloques: los menores que el pivote, y los iguales o mayores al pivote. El algoritmo debe devolver un entero que separa los dos bloques formados.

El diseño de este algoritmo sigue los mismos pasos que el anterior.

Verificación de algoritmos iterativos

Para verificar un algoritmo iterativo, en su forma imperativa o funcional, partimos de un algoritmo dado y una *restricción de entrada-salida*. Verificar un algoritmo iterativo supone demostrar que cumple la restricción entrada-salida dada. Por comodidad para hacer la verificación es conveniente normalizar el algoritmo en primer lugar.

Para hacer esto tenemos que deducir, a partir del algoritmo, el estado, la secuencia y la función de retorno y si es el caso el acumulador correspondiente. Luego, debemos imaginar un invariante, demostrar por inducción que se cumple en todos los estados. La demostración por inducción implica demostrar que el invariante es válido en el estado inicial y supuesto válido en un estado demostrar que es válido en el siguiente.

Junto con lo anterior, para que la demostración por inducción sea válida debe acabar, necesitamos imaginar un *tamaño*, llamado a menudo *función de cota*, y demostrar que decrece en cada iteración. El tamaño es una función del estado del algoritmo y mide el trabajo que falta por hacer: las iteraciones que faltan.

Finalmente deducir la restricción entrada-salida de los elementos anteriores. Para ello hay que tener en cuenta que al final del bucle se cumple el invariante, la negación de la guarda y la relación establecida por la función de retorno entre el estado y los parámetros de salida.

Veamos, en primer lugar, la verificación de algoritmos que calculen la factorial de un número n. La restricción de salida es $R(n,r) \equiv r = n!$. El algoritmo a verificar el que se incluye abajo.

```
Long f(n) {
       t = 1;
       b = 1;
       while(t <= n){
          e = t;
          t = t + 1;
          b = b * e;
       }
       return b;
}
```

Transformando los bloques básicos en asignación de tuplas y eliminando la variable t tenemos:

```
Long factorial(n) {
       (e,b) = (1,1);
       while(e <= n){
          (e,b) = (e + 1, b*e);
       }
       return a;
}
```

Diseñando el record *A(Integer e, Integer b)* y factoría *of,* podemos escribirlo como:

```
Long factorial(Integer n) {
       A t  = A.of(1,1);
       while(t.e() <= n){
          t = A.of(t.e() + 1, t.b()*t.e());
       }
       return t.b();
}
```

O usando el acumulador *reduce* con valor inicial:

```
Long factorial(Integer n) {
       return LongStream.rangeClosed(1, n)
              .reduce(1,(x,y)->x*y);
}
```

Los elementos para la verificación, siguiendo la forma normalizada son:

- El estado es *(e,a)*
- La secuencia viene definida por (e0, a0) = (1, 1), g(e, a) = e ≤ n , s(e, a) = (e+1, a∗e);
- Un invariante puede ser *I(e,a) ≡ a = (e−1)!*
- Una función de cota *C(e,a) = n−e*
- El invariante se cumple en el estado inicial. En efecto *I(1,1), 1 = 0!*

Si el invariante se cumple en un estado se cumple en el siguiente. Para ello usamos la restricción entrada salida asociada a la asignación de tuplas interior del bucle tenemos las restricciones:

```
// Si esto se cumple
I(e,a) ≡ a =(e−1)! //
// Debe cumplirse
I(s(e,a)) ≡ I(e+1,a∗e) ≡ a∗e =(e+1−1)! = e!,    a∗e = e!
//Usando las propiedades de la factorial y usando lo supuesto
a =(e−1)!, e! =(e−1)!∗e  ≡  a∗e = e!
```

La función de cota disminuye en cada iteración

```
C(s(e,a)) < C(e,a) ≡  n−(e+1) = n−e−1 < n−e
```

La restricción de entrada salida es:

```
R(r,n)≡      r = a, a = (e-1)!,!(e <= n)
             r = a, a = (e-1)!, e = n +1
             r = n!
```

Donde se ha eliminado *a, e* y que *!(e <= n)* implica *e = n+1* dada la secuencia del algoritmo.

Como podemos comprobar podemos hacer explícita una secuencia y un acumulador:

```
s = (1, e -> e <= n, e -> e +1)
a = (1, (a,e) -> a*e)
```

Es decir, el algoritmo acumula la secuencia con el acumulador dado. Este acumulador acumula los productos de los elementos de la secuencia.

La idea de la demostración es válida para todos los algoritmos que explicitan la secuencia y el acumulador. En esos algoritmos la secuencia es acumulada por el acumulador y siempre es la misma demostración.

La demostración también es válida en el caso que se haga explícito el filtro, la transformación y el aplanamiento.

Esquemas recursivos

La definición recursiva de un problema es una especificación de la solución de este en base a la de otros problemas de la misma naturaleza, pero de un tamaño más pequeño. Todo problema tiene un conjunto de propiedades y posiblemente una solución. En toda definición recursiva aparecen los conceptos de *caso base*, *caso recursivo* y *tamaño de un problema*.

Lo primero que debemos tener en cuenta en una definición recursiva de un problema es que siempre debemos partir de un *conjunto de problemas*. Llamaremos *dominio* a un predicado que nos indica si un problema pertenece al conjunto de problemas. En lo que sigue representaremos los problemas por *p, p1, p2, ..., pr*. Un conjunto de problemas lo representaremos por *P*. Cada problema tendrá unas propiedades *x* que puede ser una tupla *x = (x1, ..., xm)*. Como en el diseño de tipos estas propiedades pueden ser individuales, compartidas, básicas, derivadas, etc. Un problema concreto podemos pensarlo como un objeto del tipo que representa todos los problemas posibles del conjunto de problemas *P*. El dominio lo representamos por *D(x).*

Dentro de un conjunto de problemas *P* los valores de sus propiedades identifican al problema de manera única. El dominio, *D(x)*, es una

expresión lógica válida para las propiedades de todos los problemas que están incluidos en el conjunto de problemas de interés. A cada problema le asociamos el concepto de *tamaño* que es una nueva propiedad derivada del mismo. Normalmente representaremos el tamaño de un problema mediante n y lo calcularemos mediante una función sobre sus propiedades. El tamaño del problema deber ser un entero mayor o igual que cero que nos dé una idea de la complejidad de este. Problemas de tamaño mayor serán más complejos que otros de tamaño menor. Puede haber distintas formas para escoger el tamaño de un problema.

Dentro de un conjunto de problemas aquellos que tienen una solución directa los llamamos *casos base*. Estos suelen tener un tamaño pequeño. Puede haber más de un caso base. El resto de los problemas del conjunto de problemas considerado los denominaremos *casos recursivos*. La solución de un caso recursivo se define en función de la de otros problemas de tamaño menor. Estos los denominaremos subproblemas. Un mismo problema puede tener diferentes definiciones recursivas.

Como hemos dicho antes, una definición recursiva necesita partir de un conjunto de problemas de interés para poder expresar la solución de un problema en base a la de otro u otros de tamaño más pequeño. En muchos casos debemos imaginar ese conjunto de problemas a partir del problema original. A ese proceso lo llamamos *generalizar*. Desde este punto de vista, generalizar un problema es añadir propiedades al problema original para considerarlo un caso particular de un conjunto de problemas más amplio. La generalización ya la hemos visto previamente cuando teníamos que escoger un estado para diseñar un algoritmo iterativo.

Un algoritmo recursivo donde cada caso recursivo se define en base a un solo problema se dice que es de *recursivo simple* y si se define en base a más de uno *recursivo múltiple*. Un algoritmo *recursivo simple* es recursivo final si el resultado es igual al de la llamada recursiva sin ninguna operación posterior.

Algoritmo recursivo final:

$$mcd(a,b) = \begin{cases} a, & b = 0 \\ mcd(b, a\%b), & a, b > 0 \end{cases}$$

Algoritmo recursivo simple no final:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0 \end{cases}$$

Algoritmo recursivo múltiple:

$$fib(n) = \begin{cases} n, & 0 \leq n \leq 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

El esquema general de un algoritmo recursivo múltiple es:

```
R f(x) {
        if(b(x)){
                s = sb(x);
        } else {
                (y0,y1,..) = sp(x);
                (s0,s1,..) = (f(y0),f(y1),..)
                s = c(s1, s2,…, x);
        }
        return s;
}
```

En un algoritmo recursivo aparecen los siguientes conceptos:

- $b(x)$*:* Es una función lógica que devuelve verdadero si el problema *es un caso base*
- $sb(x)$*:* Es una función que devuelve la *solución del caso base*.
- $sp(x)$*:* Es una función que devuelve una tupla de los *subproblemas* al que se reduce el problema original. Esta función la podemos denominar, también, función de partición.
- $c(s0, s1, \dots, x)$*:* Es una función, que llamaremos *función de combinación*, que obtiene la solución del problema combinando las soluciones de los subproblemas con las propiedades del problema.
- $y0, y1, \dots$ Son los subproblemas.
- $s0, s1, \dots$ Son las soluciones de los subproblemas.

En muchos algoritmos recursivos puede que se repitan muchos subproblemas lo que hace que el algoritmo tarde más en ejecutarse. Para evitar los cálculos repetidos podemos mejorar el esquema recursivo

anterior intentando recordar los cálculos ya realizados. Para ello diseñamos un nuevo algoritmo recursivo que llamaremos *divide y vencerás con memoria* (en anterior era *divide y vencerás sin memoria*). Para ello necesitamos una variable *m*, de tipo diccionario, que guarde la solución *s* para cada problema *x* ya resuelto. El esquema es:

```
S f(x){
        Map<P,S> m = …;
        return fm(x,m);
}
S fm(x,m) {
        …
        if(x en m.keys()) {
                s = m[x];
        } else if(b(x)){
                s = sb(x);
                m[x] = s;
        } else {
                (y0,y1,..) = sp(x);
                (s0,s1,..) = (f(y0),f(y1),..)
                s = c(s1, s2,…, x);
                m[x] = s;
        }
        return s;
}
```

Diseño de algoritmos recursivos

Con las estrategias de diseño de algoritmos recursivos obtenemos algoritmos recursivos en muchos casos múltiples y en otros casos simples no finales o finales.

Para diseño de algoritmos recursivos disponemos de las siguientes estrategias

- *Generalizar el problema* añadiendo más propiedades, y tras definir un tamaño del problema, y los casos base, reducir cada problema a subproblemas y definir la solución de problema dadas las de los subproblemas.
- *Definir una función de partición* que obtenga los subproblemas de un problema dado y a partir de ahí definir la solución de un

problema en base a la de sus problemas. Está muy relacionada con la primera estrategia.

- *Partir de un conjunto de propiedades, encontrar una definición recursiva* y a partir de ella encontrar un algoritmo recursivo tras definir un tamaño del problema y los casos base.
- *Definir una restricción entrada salida* y a partir de ella encontrar un algoritmo que la verifique. Una restricción entrada salida es un predicado que relaciona los parámetros de entrada con los resultados del algoritmo. Esto implica encontrar un estado, un invariante por relajación de la restricción de entrada salida y a partir de ellos encontrar una secuencia.

Generalización de un problema

Es la estrategia más general para diseñar algoritmos recursivos. Como ya hemos dicho la definición recursiva de un problema es una especificación de la solución de este en base a la de otros problemas de la misma naturaleza, pero de un tamaño más pequeño. Lo primero que debemos hacer en una definición recursiva de un problema es encontrar el *conjunto de problemas* uno de los cuales será el problema inicial. Dentro del conjunto de problemas aquellos que tienen una solución directa los llamamos *casos base*. Estos suelen tener un tamaño pequeño. Puede haber más de un caso base. El resto de los problemas del conjunto de problemas considerado los denominaremos *casos recursivos*.

Recordamos que llamamos *dominio*, *D(x),* a un predicado que nos indica si un problema pertenece al conjunto de problemas *P*. En lo que sigue representaremos los problemas por *p, p1, p2, ..., pr*. Cada problema tendrá unas propiedades *x* que puede ser una tupla *x = (x1, ..., xm)*. Como en el diseño de tipos estas propiedades pueden ser individuales, compartidas., básicas, derivadas, etc. Un problema concreto podemos pensarlo como un objeto del tipo que representa todos los problemas posibles del conjunto de problemas *P*. Dentro de un conjunto de problemas *P* los valores de sus propiedades identifican al problema de manera única. A cada problema le asociamos el concepto de *tamaño* que es una nueva propiedad derivada del mismo. Normalmente representaremos el tamaño de un problema mediante *n* y lo calcularemos mediante una función sobre sus

propiedades. También usaremos *nu*, el tamaño umbral, que delimita los problemas cuyo tamaño es pequeño y pueden resolverse fácilmente y entre ellos incluimos a los casos base. El tamaño del problema deber ser un entero mayor o igual que cero que nos dé una idea de la complejidad de este. Problemas de tamaño mayor serán más complejos que otros de tamaño menor. Puede haber distintas formas para escoger el tamaño de un problema.

En realidad, el conjunto P tiene una estructura de grafo dirigido donde cada problema está conectado con sus subproblemas que deben ser de un tamaño menor. Los casos base no tienen hijos. La solución de un caso recursivo se define en función de sus subproblemas. Esta relación de un problema con sus subproblemas tiene que completarse con una definición de la solución de un problema dada la solución de sus subproblemas. Todos estos detalles nos permiten concretar una definición recursiva.

La *generalización* es la técnica para obtener un conjunto de problemas a partir de un problema inicial. Un mismo problema puede tener diferentes definiciones recursivas dependiendo del conjunto (en realidad grafo) de problemas imaginado en la generalización.

Aunque solo es un caso particular es conveniente conocer algunas generalizaciones usuales para problemas que usan secuencias indexables: son secuencias indexables aquellas en las que podemos acceder al elemento en posición i, como las listas, los arrays, los *String*, los rangos, etc.

Generalización sufija o prefija en secuencias indexables. Dada una secuencia indexable s de tamaño n se generaliza a (i, s) con $0 \leq i \leq n$. El problema generalizado (i, s) representa a la subsecuencia $s[i, n]$ de tamaño $n - i$. Cada problema generalizado (i, s) se puede reducir al $(i + 1, s)$. Los casos bases posibles son: (n, s), $(n - 1, s)$, $(n - 2, s)$. El problema inicial es $(0, s)$.

A esta generalización la llamamos *generalización sufija* en el sentido que el subproblema es el sufijo de la secuencia original.

De la misma manera podemos tener la *generalización prefija*. En este caso el problema generalizado (i, s) representa a la subsecuencia $s[0, i]$ de tamaño i. Cada problema generalizado (i, s) se puede reducir al $(i - 1, s)$.

Los casos bases posibles son: $(0, s), (1, s), (2, s)$. El problema inicial es (n, s).

Con esta generalización sufija los esquemas resultantes son:

```
R algr(S s) {
        B b = algr(s,0);
        return r(b);
}
B algr(int i, S s){
        B r;
        if(n-i < nu){
                r = sb(i,s);
        }else{
                r = algr(i+1,s);
                r = c(r,i,s);
        }
return r;
}
```

Este tipo de esquemas produce, en general, algoritmos recursivos simples no finales y con una complejidad dada por la ecuación:

$$T(n) = T(n-1) + \Theta(n^k)$$

Con solución $\Theta(n^{k+1})$. Donde $\Theta(n^k)$ asumimos que es la complejidad de la función de combinación.

Generalización central en secuencias indexables. Un problema sobre una secuencia indexable s de tamaño n se generaliza a (i, j, s) con $0 \leq i \leq n, i \leq j \leq n$. Un problema generalizado (i, j, s) representa a la subsecuencia $s[i, j]$ de tamaño $j - i$. Cada problema generalizado (i, j, s) se puede reducir a uno o a los dos más pequeños (i, k, s) , (k, j, s) con $k = (j + i)/2$. Los casos bases posibles son aquellos que cumplen: $j - i = 0, j - i = 1, j - i = 2$. Hay otras formas de definir los subproblemas. Una de ellas es que k se escoja aleatoriamente. Otra posibilidad es que haya un solo subproblema que sea $(i + 1, j - 1, s)$ y muchas más posibilidades.

Con esta generalización central los esquemas resultantes son:

```
R algr(S s) {
       B b = algr(s,0,n);
       return r(b);
}
B algr(int i, int j, S s){
       B r;
       if(n-i < nu){
              r = sb(i,j,s);
       }else{
              int k = (j+i)/2;
              R r1 = algr(i,k,s);
              R r2 = algr(k,j,s);
              r = c(r1,r2,i,j,s);
       }
       return r;
}
```

Este tipo de esquemas produce, en general, algoritmos recursivos múltiples y con una complejidad en el caso peor dada por la ecuación:

$$T(n) = 2T(n/2) + \Theta(n^d)$$

Con solución dependiendo del valor de k:

$$T(n) = \begin{cases} \Theta(n) & si \quad d = 0 \\ \Theta(n \ log \, n) & si \quad d = 1 \\ \Theta(n^d) & si \quad d > 1 \end{cases}$$

Si se consigue hacer solo una de las llamadas recursivas entonces la complejidad vine dada por la ecuación:

$$T(n) = T(n/2) + \Theta(n^d)$$

Con solución dependiendo del valor de *k*:

$$T(n) = \begin{cases} \Theta(\ log \, n) & si \quad d = 0 \\ \Theta(n^d) & si \quad d \geq 1 \end{cases}$$

Donde $\Theta(n^d)$ asumimos que es la complejidad de la función de combinación.

Como ejemplos de esta estrategia de generalización y obtención de un conjunto de problemas, veamos algunos algoritmos recursivos muy conocidos. Se trata de los siguientes problemas:

1. Dada una lista ordenarla con respecto a un orden. Veremos para resolver este problema el algoritmo conocido como *Quicksort*.
2. Dada una lista y un orden encontrar, sin ordenarla, que ocuparía la posición *k* si la lista se ordenara. Veremos un algoritmo conocido como el *k-ésimo*.

El algoritmo del *Quicksort* se basa en el de la Bandera Holandesa que vimos anteriormente. La idea es:

- Se generaliza el problema según la generalización de la subsecuencia central escogiendo k mediante la definición de un pivote.
- Se reordena la lista con la Bandera Holandesa
- Se vuelve a aplicar Quicksort los segmentos menores y mayores al pivote.
- El caso base será cuando el tamaño del problema sea pequeño. En ese caso aplicaremos cualquier método elemental de ordenación.

```
<E> void quickSort(List<E> lista, int i, int j,
       Comparator<? super E> ord){
       Preconditions.checkArgument(j>=i);
       if(j-i <= 4){
              ordenaBase(lista, i, j, ord);
       }else{
              E pivote = escogePivote(lista, i, j);
              Tuple2<Integer,Integer> p =
                     banderaHolandesa(lista, pivote, i, j, ord);
              quickSort(lista,i,p.v1,ord);
              quickSort(lista,p.v2,j,ord);
       }
}
```

El algoritmo del *k-ésimo* se basa también en el de la Bandera Holandesa.

```
<E> E escogeKesimo(List<E> lista, int i, int j, int k,
              Comparator<? super E> ord){
       Preconditions.checkArgument(j>=i);
       E r;
       if(j-i == 0){
              r = null;
       } else if(j-i == 1){
              r = lista.get(i);
       }else{
              E pivote = escogePivote(lista, i, j);
              Tuple2<Integer,Integer> p =
                     banderaHolandesa(lista, pivote, i, j, ord);
              if(k < p.v1){
                     r = escogeKesimo(lista,i,p.v1,k,ord);
              }else if(k >= p.v2){
                     r = escogeKesimo(lista,p.v2,j,k,ord);
              }else{
                     r = lista.get(k);
              }
       }
       return r;
}
```

La idea es:

- Se generaliza el problema según la generalización de la subsecuencia central
- Se escoge un pivote
- Se reordena la lista con la Bandera Holandesa
- Según el valor de k se busca en el segmento de los menores o mayores que el pivote. Se vuelve a aplicar *Quicksort* los segmentos menores y mayores al pivote.
- Los casos base son: que el tamaño sea 1 en cuyo caso ese es el elemento que buscamos o que la posición esté en el segmento de los iguales al pivote.

Las secuencias no indexables se generalizan utilizando un iterador. Dado el agregado *d* con iterador asociado *it* la generalización se consigue con os problemas *(it,d)* donde *it* un iterador al resto de los elementos del agregado excluyendo el primero. Los esquemas resultantes son de la forma:

```
R algr(D d) {
        Iterator<E> it = d.iterator();
        B b = algr(it,d);
        return r(b);
}
B algr(Iterator<E> it, D d){
        B r = sb(d);
        if(it.hasNext()){
                E e = it.next();
                r = algr(it,d);
                r = c(e,r,d);
        }
        return r;
}
```

Definición de una función de partición

Esta estrategia es similar a la anterior pero es más general que la anterior de generalización para secuencial indexables en el sentido que se puede aplicar a agregados de datos no indexables y a otros agregados que son secuencial.

Asumimos que tenemos un agregado de datos de tipo *D* compuesto de elementos de tipo *E*. Esencialmente una función de partición es una función que dado un agregado de datos de tipo *D* nos devuelve una tupla con sus partes y quizás algún elemento de particular. A cada resultado de una función de partición le llamamos una *vista del agregado de datos* y asumimos que el tipo D tiene el método *size()* que nos indica su tamaño. Estas vistas podemos clasificarlas en:

- *Vistas de tipo 1*, *record View1<D,E> (E e, D rs).* Donde *e* un elemento que puede quede ser definido en cada caso y *rs* el resto del agregado. En los casos más frecuentes *e* es el primer o el último elemento del agregado.
- *Vistas de tipo 2*, *record View2<D> (D left,D right).* Donde *left* es la parte izquierda del agregado y *right* la parte derecha. Según los casos ambas partes pueden ser del mismo o diferente tamaño.
- *Vistas de tipo 2 con elemento*, *record View2E<D> (E e,D left,D right).* Donde *e* es un elemento del agregado, *left* es la parte izquierda del agregado y *right* la parte derecha. Según los casos ambas partes pueden ser del mismo o diferente tamaño. Igualmente la vista puede ser cerrada o abierta. Abierta si *e* no pertenece a *left* ni a

right y estas son disjuntas. Cerrada si *left* y *right* tienen una intersección que es justamente *e*.

- *Vistas de tipo 4*, *record View4<D> (D a, D b, D c, D d).* Donde *a, b, c, d* son cada una de las cuatro partes en las que se divide el agregado.

Algunas de las generalizaciones anteriores para secuencias indexables pueden ser reinterpretadas como vistas. Así tenemos algunas posibilidades:

- $(s[i],(i+1,s)) = view1((i,s))$
- $(((i,k,s),k,(k,j,s) = view2\big((i,j,s)\big), k = (i+j)/2.$
- $(s[k],((i,k,s),k,(k,j,s) = view2e\big((i,j,s)\big), k = (i+j)/2.$

Los esquemas para este tipo de vistas son similares a los vistos para las generalizaciones de secuencias y la complejidad la misma. Repetimos, como ilustración, el caso de *view2e.*

```
R algr(D d) {
	B b = algr2(s);
	return r(b);
}
B algr(D d){
	B r;
	if(d.size() < nu){
		r = sb(d);
	}else{
		View2E<D,E> (e,left,right) = view2e(d);
		R rl = algr(left);
		R rr = algr(right);
		r = c(rl,rr,e);
	}
	return r;
}
```

Además de las secuencias indexables hay vistas de tipo 1 para otros tipos de agregados dando lugar a los esquemas:

```
R algr(D d) {
       B b = algr2(s);
       return r(b);
}
B algr(D d){
       B r;
       if(d.size() < nu){
              r = sb(d);
       }else{
              View1<D,E> (e,rs) = view1(d);
              R r = algr(rs);
              r = c(r,e);
       }
       return r;
}
```

Un caso de ellas vistas de tipo 1 es un iterable. Así podemos tener *View1<Iterable<E>,E> (e,it2) = view1(it)* con e el elemento obtenido con *it.next()* y *it2* el iterador resultante. Las generalizaciones prefija o sufija de secuencias y las que usan iteradores dan lugar a esquemas similares usando vistas.

La ventaja que tiene usar vistas es que pueden estar implementadas para cada tipo de datos y reutilizadas. En el repositorio hay implementadas para algunos tipos de datos.

El uso de la View4 da lugar a esquemas del tipo:

```
R algr(D d) {
        B b = algr2(s);
        return r(b);
}
B algr(D d){
        B r;
        if(d.size() < nu){
                r = sb(d);
        }else{
                View4<D> (abcd) = view2e(d);
                R ra = algr(a);
        R rb = algr(a);
        R rc = algr(a);
                R rd = algr(a);
                r = c(a,b,c,d);
        }
        return r;
}
```

Este tipo de esquemas produce, en general, algoritmos recursivos múltiples y con una complejidad en el caso peor dada por la ecuación:

$$T(n) = 4T(n/2) + \Theta(n^d)$$

Con solución dependiendo del valor de d. Recordemos que la solución es:

$$T(n) = \begin{cases} \Theta\left(n^{log_b a}\right) & si \quad a > b^d \\ \Theta\left(n^d \, log^{p+1} n\right) & si \quad a = b^d \\ \Theta\left(n^d \, log^p n\right) & si \quad a < b^d \end{cases}$$

Suponemos que cada parte de la vista tiene la mitad del tamaño que el agregado completo como es el caso con las matrices. Entonces $a = 4, b = 2$. Por lo tanto para $d = 2$ se cumple $a = b^2$, para $d < 2, a > b^d$ y para $d > 2$ entonces $a < b^d$. La solución anterior se concreta en:

$$T(n) = \begin{cases} \Theta(n^2) & si \quad d < 2 \\ \Theta(n^2 \, log\, n) & si \quad d = 2 \\ \Theta\left(n^d\right) & si \quad d > 2 \end{cases}$$

Veamos un ejemplo: obtener la multiplicación de dos matrices:

- De forma iterativa

- De forma recursiva dividiéndolas en cuatro submatrices

Para enfocar el problema proponemos un sistema de coordenadas para nombrar las casillas de la matriz. Nombremos las filas de arriba abajo por i y las columnas por j de tal forma que (0,0) es la casilla superior izquierda. Una matriz vendrá definida por su número de filas y su número de columnas: : $nf \times nc$. Una submatriz es una vista de una matriz definida por un vértice superior izquierdo y un tamaño de filas y columnas $(m, i0, nf, j0, nc)$. Es una idea similar a *sublist*.

Una matriz la podemos dividir en cuatro vistas que denominaremos a, *b, c, d*. La vista *a* vendrá definida por $(0, nf/2, 0, nc/2)$ y de forma similar el resto. Cada submatriz tiene asociadas unas coordenadas locales tales que el vértice superior izquierdo es (0,0).

Llamemos *Matrix<E>* al tipo que representa una matriz o una vista de esta. Las funciones para operar con este tipo son:

- *Matrix<E> of(E[][] datos):* Una nueva matriz definida por un array
- *Matrix<E> zero(int nf, int nc):* Una nueva matriz con todos sus elementos cero
- *Matrix<E> one(int n):* Una nueva matriz unidad de *n x n*.
- E get(int i, int j): El valor de la casilla i, j
- *void set(int i, int j, E value):* Actualización de la casilla i, j con value.
- *View4<E> views():* Las cuatro vistas de una matriz
- Matrix<E> compose(Matrix<E> a, Matrix<E> b, Matrix<E> c, Matrix<E> d): Construye una matriz a partir de sus vistas

Con View4, como hemos comentado, de la forma:

```
public record View4<D>(D a,D b,D c,D d) {}
```

El método *views()* es el que consigue los subproblemas. Diseñamos la multiplicación de las dos matrices cuadradas a, b. Teniendo en cuenta que

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} * \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix} = \begin{pmatrix} a1 * a2 + b1 * c2 & a1 * b2 + b1 * d2 \\ c1 * a2 + d1 * d2 & c1 * b2 + d1 * d2 \end{pmatrix}$$

Un algoritmo iterativo y otro recursivo son:

```
Matrix<E> multiply(Matrix<E> in2){
  Matrix<E> r = Matrix.zero(this.nf,in2.nc);
  for (int i = 0; i < this.nf; i++) {
    for (int j = 0; j < in2.nc; j++) {
        r.set(i,j,this.field.getZero());
        for (int k = 0; k < this.nc; k++){
          E val = (this.get(i,k).multiply(in2.get(k,j)))
              .add(r.get(i,j));
          r.set(i,j,val);
        }
    }
  }
  return r;
}
```

```
Matrix<E> multiply_r(Matrix<E> m2){
  Matrix<E> r;
  if(this.nc < 2 || this.nf < 2 || m2.nf < 2 || m2.nc < 2) {
       r = this.multiply(m2);
  } else {
       View4<Matrix<E>> v1 = this.views();
       View4<Matrix<E>> v2 = m2.views();
       Matrix<E> a = v1.a.multiply_r(v2.a)
              .add(v1.b.multiply_r(v2.c));
       Matrix<E> b = v1.a.multiply_r(v2.b)
              .add(v1.b.multiply_r(v2.d));
       Matrix<E> c = v1.c.multiply_r(v2.a)
              .add(v1.d.multiply_r(v2.c));
       Matrix<E> d = v1.c.multiply_r(v2.b)
              .add(v1.d.multiply_r(v2.d));
       r = Matrix.compose(a, b, c, d);
  }
       return r;
}
```

Como podemos que también la multiplicación iterativa y la recursiva tienen complejidad a $\Theta(n^3)$

$$T(n) = 8T(n/2) + n^2, \qquad \Theta\left(n^{\log_2 8}\right) = \Theta(n^3)$$

Esta complejidad puede reducirse aún más reordenado el cálculo en la forma (es el denominado algoritmo de *Strassen*):

$$p1 = a(f-h), p2 = (a+b)h, p3 = (c+d)e, p4 = d(g-e)$$

$$p5 = (a+d)(e+h), p6 = (b-d)(g+h), p7 = (a-c)(e+f)$$

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} * \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix} = \begin{pmatrix} p5+p4-p2+p6 & p1+p2 \\ p3+p4 & p1+p5-p3-p7 \end{pmatrix}$$

Para la multiplicación de matrices podemos seguir, también, el esquema siguiente que como podemos comprobar tendrá asociada la ecuación de recurrencia

$$T(n) = 7T(n/2) + n^2, \;\; \Theta\left(n^{\log_2 7}\right) = \;\; \Theta(n^{2.8})$$

Partir de un conjunto de propiedades

Esta es una estrategia muy adecuada cuando se disponen de propiedades sobre el problema que queremos resolver. Pero disponer de propiedades no es lo mismo que disponer de una *definición recursiva*. Una definición recursiva especifica la solución de un problema en base a otros problemas de tamaño menor y, además, da la solución para los casos base. Un conjunto de propiedades normalmente tiene que ser completada para convertirlo en una definición de recursiva. Pero cuando disponemos de un conjunto de propiedades debemos aprovecharlas.

Las siguientes son propiedades de la factorial, pero no constituyen una definición recursiva.

$$\begin{aligned} n! &= \begin{cases} \dfrac{(n+1)!}{n+1}, & n \geq 0 \end{cases} \\ n! &= n * (n-1)! \\ 1! &= 1 \\ 0! &= 1 \end{aligned}$$

Escogiendo n como tamaño podemos construir un algoritmo con la segunda y la cuarta, pero no podemos usar la primera porque indica la solución en base a la de otro problema de tamaño mayor. La definición recursiva es:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 1 \end{cases}$$

Otro ejemplo es calcular el máximo común divisor de *a, b* partiendo de las propiedades:

- El entero *a%b* tiene los mismos divisores que los divisores comunes entre *a, b.* Esto puede ser fácilmente demostrado porque cualquier número que divida a *a,b* también divide a *a%b.*
- El máximo común divisor de a, 0 es 0. Lo cual es evidente.

Con lo anterior podemos escoger un estado de tipo *(a,b)* y comprobar que sus divisores comunes lo son también a *(b, a%b).* Si escogemos como tamaño b vemos que podemos definir una secuencia de estado que van manteniendo el conjunto de divisores comunes hasta llegar a un del tipo *(a,0).* La definición recursiva es:

$$mcd(a,b) = \begin{cases} a, & b = 0 \\ mcd(b, a\%b), & b > 0 \end{cases}$$

Esta definición es recursiva final como ya hemos visto anteriormente y tiene un equivalente iterativo. Es el llamado algoritmo de Euclides.

Un ejemplo más complejo es el número combinatorio. Los números combinatorios tienen las propiedades:

$$\begin{cases} \binom{n}{0} = \binom{n}{n} = 1, & n \geq 0 \\ \binom{n}{1} = \binom{n}{n-1} = n, & n \geq 1 \\ \binom{n}{k} = \binom{n}{n-k}, & n \geq k \\ \binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}, & n \geq k \end{cases}$$

Escogiendo como tamaño *n* tenemos el algoritmo recursivo:

$$\binom{n}{k} = \begin{cases} 1, & k, n-k = 0 \\ n, & k, n-k = 1 \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & k > 1 \end{cases}$$

A partir de una restricción de entrada salida

Como ejemplo de la última estrategia definir una restricción entrada salida y a partir de ella encontrar un algoritmo que la verifique veamos el ejemplo de más abajo.

Una *restricción entrada salida* es un predicado que relaciona los parámetros de entrada con los resultados del algoritmo ignorando las variables intermedias del mismo. Esto implica encontrar un estado y un invariante. Este invariante se suele obtener por relajación de la restricción de entrada salida y a partir de ellos encontrar una relación recursiva. La relación recursiva implica un concepto de tamaño porque el subproblema debe ser de un tamaño más pequeño que el problema.

Como ejemplo veamos cómo obtener un algoritmo para que dados dos enteros *a, b* podamos encontrar su cociente y su resto: *c, r*. Los parámetros de entrada son a, b y lo de salida *c,r.* La restricción entrada salida:

$$R(a,b,c,r) \equiv a = b*c+r \wedge r < b \wedge r < c \wedge a \geq 0 \wedge b > 0$$

Imaginemos un estado formado por $(a1,b1,c1,r1)$ con un invariante relajado de la relación anterior

$$I(a1,b1,c1,r1) \equiv a1 = b1*c1+r1 \wedge a \geq 0 \wedge b > 0$$

Si expresamos la restricción en la forma

$$a1 = b1*c1+b1-b1+r1 = b1*(c1+1)+r1 - b1$$

Lo cual quiere decir que la tupla *(a1,b1,c1+1,r1-b1)* cumple el invariante y además ha decrecido la cantidad *r1-b1* que ha pasado a ser r1 a *r1-b1.* Teniendo en cuenta que *a, b* no cambian tenemos la *definición recursiva:*

$$cr(c1,r1) = \begin{cases} (c1,r1), & r1-b < 0 \\ cr(c1+1,r1-b1), & r1-b \geq 0 \end{cases}$$

Claramente el problema requiere un problema inicial. Este podría ser *(a,b,0,a).* Tal como podemos ver la definición es recursiva final y por lo tanto podemos escribir un algoritmo recursivo.

```
public static record Cr(Integer c, Integer r) {}

public static Cr cr(Integer a, Integer b) {
              return cr(a,b,0,a);
}
public static Cr cr(Integer a,Integer b,Integer c,Integer r) {
       Cr s = new Cr(c,r);
       if(r-b>0){
              s = cr(a,b,c+1,r-b);
       }
       return s;
}
```

El algoritmo tiene un equivalente iterativo. La asignación de tuplas puede desplegarse en este caso fácilmente y para el resultado usamos un record:

```
public static record Cr(Integer c, Integer r) {}
Cr cr(Integer a, Integer b) {
       Integer c = 0;
Integer r = a;
       while(r-b>0){
              c = c+1;
              r = r-b;
      }
       return new Cr(c,r);
}
```

La versión funcional se deja como ejercicio.

En este caso el diseño recursivo nos ha llevado a un algoritmo iterativo por resultar la definición recursiva final.

Transformaciones recursivo-iterativo en recursividad simple

Después de aplicadas las estrategias de diseño de algoritmos obtenemos algoritmos recursivos de diversos tipos: recursivos simples y múltiples y entre los simples finales y no finales. Veamos ahora estrategias de transformación de estos algoritmos recursivos. En primer lugar veamos las transformaciones de los algoritmos recursivos simples. Si son finales

ya hemos visto como obtener versiones iterativas y funcionales equivalentes. Abordemos, por tanto, las estrategias de transformación de los algoritmos recursivos simples no finales.

Abajo incluimos el esquema general de un algoritmo recursivo simple no final con iterador implícito:

```
R alg(X x) {
       Iterator<E> = it(x);
       B b = alg(x,it);
       return r(b);
}
B agl(X x,Iterator<E> it) {
       if(it.hasNext())){
              E e = it.next;
              b = alg(x,it);
              if(!f(b)){
                     b = c(b,e); // b.c(e) si B es mutable
              }
       } else {
              b = sb(x,it);
       }
       return b;
}
```

O con iterador explícito:

```
R alg(X x) {
       T t = it(x);
       B b = alg(x,it);
       return r(b);
}
B alg(X x, T t) {
       if(hn(t)){
              E e = nx1(t);
              t = nx2(t);
              b = alg(x,t);
              if(!f(b)){
                 b = c(b,e); // b.c(e) si B es mutable
              }
       } else {
              b = sb(x,t);
       }
       return b;
}
```

En lo anterior *sb* es la solución del caso base. Esta solución depende de los parámetros de entrada y, si el iterador es explícito, del estado del iterador.

Hay muchas semejanzas con el caso iterativo. De nuevo aparece la secuencia y el acumulador.

$s = (e_0, e \to g(e), e \to nx(e))$
$a = (b0, (b, e) \to c(b, e), b \to r(b), b \to f(b))$

Y el iterador que implementa la secuencia si se hace explícito.

$it = (t_0, t \to hn(t), t \to nx1(t), t \to nx2(t))$

La diferencia ahora es que la acumulación se hace posteriormente, es decir es la *acumulación por derecha*. No hay un algoritmo iterativo directamente equivalente. Aunque ahora veremos cuándo podemos encontrar uno. La acumulación que se hace con los algoritmos iterativos es una *acumulación por la izquierda*.

De lo anterior podemos deducir que, si el acumulador da el mismo resultado acumulándolo por la izquierda que por la derecha, existe un algoritmo iterativo equivalente al recursivo simple no final. Es el algoritmo asociado a la aplicación del acumulador por la izquierda a la secuencia. Cuando la operación de acumulación es de la forma $c(b, e) = b \ op \ f(e)$ y op es asociativo y conmutativo se puede demostrar que la aplicación a izquierda y derecha del acumulador es equivalente. Para que pueda ser conmutativo f debe tener tipo $f: E \to B$.

Ejemplos de acumuladores que cumplen las propiedades anteriores son: sumar, multiplicar, min, max, all, any, none, contar, agrupar en conjuntos, agrupar con los operadores anteriores aplicados a los grupos, counting, etc.

Al algoritmo los llamamos *acumulación por la derecha* y el que se hace iterativamente *acumulación por la izquierda*. Llamamos reducción a un caso particular de este algoritmo cuando el acumulador es de tipo reduce. Es decir, el tipo *B = R y* la función de retorno es la identidad. Un caso más concreto todavía es si *B = E y R = Optional<E>*. Estos algoritmos más concretos se llaman *reducción por la izquierda* o *reducción por la derecha* que puede ser con elemento inicial o no.

Hay casos en los que dado un acumulador a podemos encontrar otro $a' = (b0', c'(b, e))$, y posiblemente otra secuencia s', tal que a aplicado por la derecha a s de el mismo resultado que a' aplicado por la izquierda s'. También en este caso podemos encontrar un algoritmo iterativo equivalente al recursivo simple no final. Veremos algunos ejemplos.

Podemos resumir sin demostración que:

La acumulación por la izquierda se hace mediante un algoritmo iterativo o uno equivalente recursivo final. La acumulación por la derecha se hace mediante un algoritmo recursivo no final.

Cada algoritmo iterativo puede ser visto como la acumulación por la izquierda de una secuencia y un acumulador.

Igualmente, cada algoritmo recursivo final puede ser visto como la acumulación por la izquierda de una secuencia y un acumulador.

Cada algoritmo recursivo no final puede ser visto como la acumulación por la derecha de una secuencia y un acumulador.

De las definiciones de acumulación por la derecha y por la izquierda podemos concluir que:

Si la función de combinación del acumulador es de la forma $(b, e) = b\ op\ f(e), siendo\ \ op$ un operador asociativo y conmutativo entonces es igual la acumulación por la izquierda y por la derecha sobre cualquier secuencia.

Si la función de combinación del acumulador es de la forma $(b, e) = b\ op\ f(e)$, siendo op un operador asociativo y no conmutativo entonces hay un algoritmo iterativo equivalente con la misma secuencia y un acumulador con la función de combinación $(b, e) = f(e)\ op\ b$.

Si s^{-1} es la secuencia inversa de s entonces se cumple:

$$acumulaIzquierda(s, a) = acumulaDerecha(s^{-1}, a)$$

Si un acumulador a acumula por la derecha una secuencia s y se obtiene el mismo resultado si acumulamos a' por la izquierda con la secuencia s', entonces a', s' nos permitirán definir un algoritmo iterativo equivalente al recursivo definido por la acumulación por la derecha de a, s.

Con las ideas anteriores si existe una función de combinación $c'(b,e)$ equivalente a la $c(b,e)$ existe un algoritmo recursivo final equivalente que usa un acumulador con esa función de combinación.

```
R alg(X x) {
	T t = it(x);
	B b = alg(x,it,b0);
	return r(b);
}
B alg(X x, T t, B b) {
	if(hn(t) && !f(b)){
		E e = nx1(t);
		t = nx2(t);
		b = c'(b,e); // b.c(e) si B es mutable
		b = alg(x,t,b);
	} else {
		b = c'(b,sb(x,t));
	}
	return b;
}
```

En el esquema anterior $b0$ debe cumplir $b == c'(b.b0)$. Es decir $b0$ es elemento neutro del operador *c'(_,_).* Como hemos visto anteriormente todos esquema recursivos finales tiene uno iterativo equivalente.

```
R alg(X x) {
	T t = it(x);
	B b = b0;
	while(hn(t) && !f(b)){
		E e = nx1(t);
		t = nx2(t);
		b = c'(b,e);
	}
	b = c'(b,sb(x,t));
	return r(b);
}
```

Este tiene un equivalente funcional

```
record Te(T t, B b, X x){
	public static Te of(T t, B b, X x){
		return new Te(t, b, x);
	}
	public Te next(){
		E e = nx1(t);
		t = nx2(t);
		b = c'(b,e);
		return Te.of(t,b,x);
	}
}
R alg(X x) {
	T t = it(x);
	B b = b0;
	Te r = Stream.iterate(Te.of(t,b,x),t->t.next()).
		.filter(t->!(hn(t) && !f(b))
		.findFirst()
		.get();
	b = c'(r.b(),sb(x,r.t()));
	return r(b);
}
```

Veamos un ejemplo de estas ideas: *implementar un algoritmo para calcular la inversa de una lista.*

Una definición recursiva del problema es:

$$inv(i, ls) = \begin{cases} [], & |ls| - i = 0 \\ inv(i+1, ls) + [ls[i]], & |ls| - i > 0 \end{cases}$$

Es una definición recursiva no final. Dada una lista *ls* y un elemento *e* el operador *ls+[e]* (añadir al final *add*) no es conmutativo. Usando lo explicado sobre operadores asociativos no conmutativos podemos un algoritmo iterativo equivalente al recursivo no final.

```
List<E> inversa(List<E> ls) {
	return inversa(0, ls);
}
List<E> inversa(List<E> ls, int i) {
	List<E> r;
	if(i < ls.size()) {
		r = inversa(i+1,ls);
		r.add(ls.get(i));
	} else {
		r = new ArrayList<>();
	}
	return r;
}
List<E> inversa(List<E> ls) {
	Integer i = 0;
	List<E> b = new ArrayList<>();
	while(i < ls.size()){
		b.add(0,ls.get(i));
		i++;
	}
	return b;
}
```

Transformación recursivo múltiple a iterativo

La idea es generalizar el problema incorporando el tamaño del problema y las soluciones calculadas de algunos subproblemas de tamaño menor. Establecemos un invariante sobre el estado elegido y usamos las técnicas de diseño iterativo. Para ello hacemos que el invariante se cumpla para los casos base y vamos aumentando, en el programa iterativo, el tamaño de los problemas hasta llegar al problema original. La versión recursiva final se puede obtener de la versión iterativa.

---

1. Transformar a iterativo el esquema recursivo siguiente

$$fb(n) = \begin{cases} d_0, & n = 0 \\ d_1, & n = 1 \\ a * fb(n-1) + b * fb(n-2), & n > 1 \end{cases}$$

Este esquema se puede implementar con memoria y sin memoria. Llamémoslos *fbsm, fbm*.

El de problema de *Fibonacci* es el caso concreto *a=b=1, d0 =0, d1 = 1.*

Definimos el problema generalizado con el estado $e = (i, u, v)$ y el invariante $u = fbi(i+1)$, $v = fbi(i)$. El problema final es $i = n$. Un problema inicial que cumple el invariante es *(0,1,0)*.

Decidimos incrementar en cada paso la variable *i* en *1*. Pretendemos deducir de la invariante y del incremento decidido para *i* la función siguiente.

```
(i',u',v') = (i+1, fbi(i+2), fbi(i+1))
           = (i+1, fbi(i+1)+fbi(i), fib(i+1))
           = (i+1, a*u+b*v, u)
```

Y de aquí el esquema resultante es:

```
fblin(n,a,b,d1,d0){
        (i,u,v) = (0,d1,d0);
        while(i < n){
                (i,u,v) = (i+1,a*u+b*v,u);
        }
        return v;
}
```

Desplegando la asignación paralela:

```
fblin(n,a,b,d1,d0){
        i = 0;
        u = d1;
        v = d2;
        while(i < n){
                i = i+1;
                u0 = u;
                u = a*u+b*v;
                v = u0;
        }
        return v;
}
```

Aunque con el método anterior es posible hay otro método más sencillo y general usando un *Map* de la forma:

```
Long rc(Integer n,Integer a,Integer b,Long d0,Long d1) {
       Map<Integer,Long> m = new HashMap<>();
       for(int i=0; i<=n; i++) {
              if(i==0) m.put(0,d0);
              else if(i==1) m.put(1,d1);
              else {
                     Long r = a*m.get(i-1)+b*m.get(i-2);
                     m.put(i,r);
              }
              m.remove(i-3);
       }
       return m.get(n);
}
```

La clave está en recorrer los valores de *i bottom-up*. En esta estrategia es muy importante liberar los datos de la memoria que no son necesarios

Números combinatorios

La idea anterior se puede generalizar a un conjunto variable de subproblemas usando un *Map* para mantener las soluciones ya calculadas. Veamos como ejemplo de este caso el cálculo del número combinatorio.

$$\binom{n}{k} = \begin{cases} 1, & k, n-k = 0 \\ n, & k, n-k = 1 \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & k > 1 \end{cases}$$

esquema iterativo equivalente comienza con los casos base y va calculando todos los problemas de tipo $\binom{i}{j}$ para los valores posibles de i en $[0, n]$ y j en $[0, i]$ a partir de las soluciones de los problemas $\binom{i-1}{j-1}$, $\binom{i-1}{j}$ que ya han debido ser calculados. Vamos incrementando uno a uno el valor de *i* desde 0 hasta alcanzar *n y* de *j de 0 a i.* Hemos de tener en cuenta que para los casos base ya tenemos la solución y para los casos recursivos la tenemos en la memoria. El esquema iterativo es:

```
record Bn(Long a, Long b) {
	public static Bn of(Long a, Long b) {
		return new Bn(a,b);
	}
}

Long binom(Long n, Long k) {
	Map<Bn,Long> m = new HashMap<>();
	for(long i=0; i<=n; i++) {
		for(long j=0; j<=i;j++) {
			if(j==0) m.put(Bn.of(i, 0L),1L);
			else if(j==1 || i-j ==1)
				m.put(Bn.of(i,j),i);
			else {
				Long r = m.get(Bn.of(i-1, j-1))+
					m.get(Bn.of(i-1, j-1));
				m.put(Bn.of(i,j),r);
			}
			m.remove(Bn.of(i-2,k));
		}
	}
	return m.get(Bn.of(n, k));
}
```


Recursividad múltiple y acumuladores paralelos

Los *agregados de datos* pueden ser convertidos en *flujos de datos* para ser acumulados. Los flujos de datos obtenidos pueden ser secuenciales o paralelos. Un *flujo de datos secuencial* va proporcionando uno tras otro, en algún orden, los elementos del agregado. Esto se implementa, como ya hemos visto, construyendo un iterable a partir del agregado. Un agregado de datos también puede proporcionar un *flujo de datos paralelo*. Este divide el agregado en partes, que se puede volver a subdividir las veces que sean necesarias hasta que cada parte es suficientemente pequeña y posteriormente ofrece uno tras otro los datos de cada parte.

En Java un flujo de datos se representa por el tipo *Stream.* El tipo *Stream* puede representar un flujo secuencial de datos y también un flujo de datos paralelo.

La vista secuencial del tipo *Stream<E>,* que se obtiene con el método *iterator(),* devuelve un *Iterator<E>* como ya hemos visto.

Para entender el funcionamiento paralelo de *Stream<E>* y el acumulador paralelo asociado recordemos el esquema que vimos asociado a la vista *View2*

```
R algr(D d) {
       B b = algr2(s);
       return r(b);
}
B algr(D d){
       B r;
       if(d.size() < u){
              r = reduce(d,f,b0);
       }else{
              View2<D> (left,right) = view2e(d);
              R rl = algr(left);
              R rr = algr(right);
              r = c(rl,rr);
       }
       return r;
}
```

Para acercarnos aún más al tipo *Stream<E>* asumimos que el tipo *B* es mutable, que *b0* es un *Supplier<B>,* es decir *()->b0*, que f es *BiConsumer<B,E>,* es decir *((b,e)->b.c(e))* y c es *BinaryOperator<B>,* es decir *(b1,b2) -> op(b1,b2).*

Supongamos además que disponemos de *m* procesadores. La idea del algoritmo anterior es la siguiente: divide el agregado de datos de dos partes y continúa dividiendo recursivamente hasta que el tamaño del agregado este por debajo de un umbral *u*. Cuando lo consigue pasa ese agregado a un procesador para que resuelva el problema iterativamente usando un algoritmo de tipo *reduce*, que ya hemos estudiado, con parámetros *f* y *b0*.

Los resultados conseguidos por cada procesador serán de tipo *B* y se irán combinando con el operador binario *c*. Intuitivamente podemos ver que si el algoritmo se ejecuta sobre un solo procesador y tarda un tiempo *T* entonces al disponer de m procesadores conseguiremos un tiempo de *T/m*. Evidentemente habrá costes adicionales que no vamos a tener en cuenta de paso de un procesador a otro.

El algoritmo se puede sintetizar en un flujo de datos paralelo, un *Spliterator<E> y* un acumulador paralelo, un *Collector<E,B,R>.*

La vista paralela del tipo *Stream<E>* se puede obtener mediante el método *spliterator()* que devuelve un *Spliterator<E>.* Un *Spliterator<E>* sobre un agregado de datos está asociado a la *View2* de es *agregado.* Tiene los métodos:

```
interface Spliterator<E> {
       boolean tryAdvance(Consumer<? super T> action);
              // Intenta consumir un elemento
       Spliterator() trySplit();
              // Intenta partir el agregado de datos. Si puede
              // devuelve una   parte y si no puede null
}
```

El método *trySplit()* es equivalente al siguiente código que utiliza una vista:

```
if(d.size() < nu){
       B b = reduce(d,f,b0);
}else{
       View2<D> (left,right) = view2e(d);
```

El método *trySplit()* divide el agregado de datos en dos partes si su tamaño es suficientemente grande. Una parte queda cubierta por el nuevo *Spliterator* y la otra por el antiguo. Si el tamaño del agregado es pequeño el método *trySplit()* devuelve *null*.

El método *tryAdvance* va consumiendo los elementos del agregado uno tras otro para conseguir una reducción iterativa. Este método usa un iterador del flujo de datos, una función de combinación y un elemento inicial. El iterador se pude conseguir del *Spliterator* mediante el método Es decir es el esquema *reduce(d,f,b0)*

```
It it = it(d);
B b = ()->b0();
do {
boolean r = this.tryAdvance(f);
} while(r);
```

El método *tryAdvance* intenta consumir un elemento. Si lo consigue devuelve true y si no lo consigue false. Al consumir un elemento ejecuta la acción *f*.

A su vez los acumuladores podemos clasificarlos en *acumuladores secuenciales y paralelos*. Cada uno de ellos es adecuado al tipo de flujo,

secuencial o paralelo, que tengamos. Los *algoritmos secuenciales* que vimos anteriormente tienen asociados un flujo de datos secuencial, que define una secuencia, y un acumulador secuencial. Los acumuladores paralelos se asocian a flujos de datos paralelos y a algoritmos recursivos. Ya vimos las propiedades de un acumulador secuencial. Vemos ahora las del paralelo. Muchas de ellas son similares.

Siendo e un elemento del flujo a acumular, de tipo E, b la base del acumulador de B y r el valor devuelto, de tipo R, los acumuladores secuenciales tienen las funciones:

- *Valor inicial* b_0 de la base del acumulador
- Función de acumulación $(b, e) \rightarrow c(b, e)$.
- Función de retorno $b \rightarrow r(b)$.
- Función de cortocircuito $b \rightarrow d(b)$

Los acumuladores paralelos están pensados para poder dividir el flujo de entrada en varias partes, acumular cada parte y posteriormente combinar los resultados. Tienen las funciones:

- *Valor inicial de la base* b_0 del tipo B de la base del acumulador
- Función de acumulación secuencial de un elemento en la base $(b, e) \rightarrow f(b, e)$.
- Función de combinación de dos bases $(b1, b2) \rightarrow c(b1, b2)$. Deber ser asociativa y tener elemento neutro.
- Función de retorno $b \rightarrow r(b)$.

Por otra parte, el acumulador será mutable o inmutable según lo sea el tipo B de la base del acumulador.

Estos acumuladores se implementan por el tipo *Collector<E,B,R>* que tiene los siguientes métodos:

- *Supplier<B> supplier().* Proporciona el valor inicial de la base. Es $b0$.
- *BiConsumer<B,E> consumer().* Acumula secuencialmente en la base el nuevo elemento. Es $f(b, e)$
- *BinaryOperator<B> combiner().* Combina dos valores de la base obtenidos de resultados procesados en paralelo. Es $c(b1, b2)$.

- *Function<B,R> finisher().* Calcula el resultado a partir del valor de la base. Es $r(b)$.

En Java se nos proporciona un método de factoría para crear acumuladores:

```
<E,B,R> Collector<E,B,R> Collector.of(
       Supplier<B> supplier,
       BiConsumer<B,E> accumulator,
       BinaryOperator<B> combiner,
       Function<B,R> finisher
 )
```

Este método asume que el tipo de la base es mutable. Si la base es inmutable un adaptador adecuado es:

```
Collector<E,MutableType<B>,R> ofBaseInmutable(
                    Supplier<B> supplier,
                    BiFunction<B,E,B> consumer,
                    BinaryOperator<B> combiner,
                    Function<B,R> finisher){
   return Collector.of(
       ()->MutableType.of(supplier.get()),
       (b,e)->b.newValue(consumer.apply(b.value,e)),
       (x,y)->MutableType.of(combiner.apply(x.value,y.value)),
       x->finisher.apply(x.value));
}
```

Y el método del tipo *Stream* para acumular el flujo de datos mediante un acumulador paralelo es:

```
<R,B> R collect(Collector<? super E,B,R> collector);
```

Que se usa en la forma:

```
R f(x) {
       Collector<E,B,R>  a = Collectors…;
       R r = Stream.iterate(e0, e->g(e), e->s(e)).collect(a);
     return r;
}
```

Una implementación secuencial de la acumulación crearía la base de un acumulador utilizando la función de creación proporcionada, invocaría la función de acumulación una vez para cada elemento del flujo de datos. Una

implementación paralela dividiría el flujo de datos, crearía una base del acumulador para cada parte, acumularía el contenido de cada parte en su respectivo acumulador, y luego usaría la función de combinación para fusionar los resultados parciales. Una implementación paralela posible de ese método *collect* es:

```
R collect(Stream<E> flow, Collector<E,B,R> c) {
       B b = collect(flow.spliterator(),c);
       return c.finisher().apply(b);
}

B collect(Spliterator<E> flow, Collector<E,B,R> c) {
       B b;
       Spliterator<E> f2 = flow.trySplit();
     if(f2 != null){
              B b1 = collect(f2,c);
              B b2 = collect(flow,c);
              b = c.combiner().apply(b1,b2);
       } else {
              b = c.supplier().get();
              Boolean r;
              do {
                 r = flow.tryAdvance(e->c.accumulator()
                           .accept(b,e));
              }while(r);
       }
       return b;
}
```

La implementación paralela es, en realidad, más compleja. Aquí la acumulación paralela se ha presentado como un algoritmo recursivo. Para que sea paralelo, además, cada vez que se divide el flujo hay que crear una nueva hebra que pueda ser asignada a uno de los procesadores disponibles.

Para crear un acumulador paralelo que acumule un flujo en un *multiconjunto* los divisores de todos los números enteros entre *a* y *b*.

```
<E> Collector<E,Multiset<E>,Multiset<E>> toMultiset() {
       return Collector.of(
              ()->Multiset.create(),
              (x,e)->x.add(e),
              (x,y)->Multiset.add(x, y),
              x->x);
}
Multiset<Long> rr4 = Stream.iterate(a,x->x<b,x->x+1)
                    .flatMap(x->divisores(x))
                    .collect(Collectors2.toMultiset());
```

Un *problema de recursividad múltiple* se puede transformar en muchos casos en un *flujo de datos paralelo y un acumulador paralelo*. Las condiciones para que esto sea posible son:

- Un problema de tamaño grande debe poder ser dividido en dos partes separadas y su solución debe ser una combinación de la solución de las dos partes. La solución del problema se obtendrá combinando la de las dos partes mediante una función (*combiner*).
- Un problema de tamaño pequeño debe poder considerado como un flujo de datos secuencial y su solución se puede obtener acumulando por la izquierda ese flujo secuencial mediante un acumulador secuencial de base mutable.

En definitiva un problema de recursividad múltiple se puede transformar en un *flujo de datos paralelo* si podemos implementar un *SplitIterator<E>* con los datos del problema. El algoritmo de ordenación *mergeSort* puede ponerse en forma de flujo paralelo más acumulador. Para ello necesitamos un flujo de datos paralelo. Es decir un *Stream<E>* y una implementación de *Collector<E,B,R>* con los detalles:

```
Collector<E,List<E>,List<E>> mergeSort(Comparator<E> cmp) {
     return Collector.of(
              ()->new ArrayList<>(),
              (x,e)->Lists2.insertOrdered(x,e,cmp),
              (x,y)->Lists2.mergeOrdered(x,y,cmp),
            x->x);
}
```

Las funciones usadas pueden verse en el repositorio. Y la forma de usarlo:

```
public static void main(String[] args) {
       Random r = new Random(System.nanoTime());
       Stream<Integer> s = r.ints(5000).boxed();
       List<Integer> rr = s.parallel()
              .collect(Collectors2.mergeSort());
       Strings2.toConsole(rr, "Ordenada");
}
```

Como hemos visto hay una gran relación entre un *Spliterator<E>* y las vistas *View1<D,E>* y *View2<D,E>*. Es posible hacer una implementación de

un *Spliterator<E>* dadas funciones que nos proporcionen esas vistas de un agregado de datos, su tamaño y un umbral para decidir si partir o no:

```
<D,E> SpliteratorOfView<D,E> of(D d,
       Function<D,View2<D>> view2,
       Function<D,View1<D,E>> view1,
       Function<D,Integer> size,
       Function<D,Integer> u) {
   return new
        SpliteratorOfView<D,E>(d,view2,view1,size,u);
}
```

Para el caso que el agregado es una lista

```
<E> SpliteratorOfView<List<E>,E> ofList(List<E> d) {
       return new SpliteratorOfView<List<E>,E>(d,
              ls->List2.view2(ls),ls->List2.view1(ls),
              ls->ls.size(),ls->5);
}
```

Los detalles de la implementación incluyen los atributos privados:

```
static class SpliteratorOfView<D,E> implements Spliterator<E>{

       private D d;
       private Function<D,View2<D>> view2;
       private Function<D,View1<D,E>> view1;
       private Function<D,Integer> size;
       private Function<D,Integer> nu;
```

El constructor:

```
private SpliteratorOfView(D d,
       Function<D, View2<D>> view2,
       Function<D, View1<D,E>> view1,
       Function<D,Integer> size,
       Function<D,Integer> nu) {
       this.d = d;
       this.view2 = view2;
       this.view1 = view1;
       this.size = size;
       this.nu = nu;
}
```

El método *tryAdvance*:

```
@Override
public boolean tryAdvance(Consumer<? super E> action) {
	Boolean r = false;
	if(this.size.apply(this.d) > 0) {
		View1<D,E> vw = this.view1.apply(d);
		E e = vw.e();
		this.d = vw.r();
		action.accept(e);
		r = true;
	}
	return r;
}
```

El método *trySplit*:

```
@Override
public Spliterator<E> trySplit() {
	Spliterator<E> r = null;
	if(this.size.apply(this.d) >
			this.nu.apply(this.d)) {
		View2<D> vw = this.view2.apply(d);
		r = Spliterators2.of(vw.left(),
			view2,view1,size,nu);
		this.d = vw.right();
	}
	return r;
}
```

Y por último el método de factoría:

```
public Stream<E> stream(){
		return Spliterators2.asStream(this);
}
```

El algoritmo de *Quicksort*, sin embargo, es más complejo de ajustar a este esquema.

Diseño de tipos: tipos de datos recursivos

Un buen diseño de tipos es básico para que los programas sean comprensibles y fáciles de mantener. Veamos algunas pautas para este diseño y algunos ejemplos que puedan servir de guía. Tras una presentación general de la problemática del diseño de tipos nos dedicaremos especialmente a los tipos recursivos. Aquellos en los que algunas de las propiedades son del mismo tipo que está siendo diseñado.

Diseño de tipos

Al diseñar un tipo nuevo debemos partir de los ya existentes. Es necesario decidir a qué otros tipos *extender* o que tipos *usar*. Decimos que el nuevo tipo *usa* aquellos tipos con los que declara sus propiedades. También puede diseñarse el nuevo tipo extendiendo algunos de los disponibles. Si un tipo extiende a otro (mediante *extends o implements*) decimos que el nuevo tipo es un *subtipo* de los tipos de los que hereda o también decimos que *refina* esos tipos.

Un tipo puede ser diseñado para que sea *mutable* o *inmutable*. Un tipo inmutable no tiene operaciones para cambiar sus propiedades. Una vez creado el objeto no puede cambiarlas. Un tipo mutable sí.

Un tipo puede ser recursivo, si algunas de sus propiedades son del mismo tipo diseñado, o no recursivo.

Para un tipo podemos diseñar propiedades que nos devuelvan *vistas* del objeto original. Un objeto *a* se dice una vista de *b* si los cambios en las propiedades de *a* se hacen también en *b,* pero *a* puede ofrecer una funcionalidad diferente a *b*. Por ejemplo, un tipo inmutable podría ofrecer una vista mutable.

Todo tipo tiene, además de las heredadas de los tipos que refina, unas propiedades y posiblemente unas operaciones nuevas. Cada propiedad tiene un nombre, un tipo, puede ser *consultada* y además modificada o sólo consultada, y puede ser una propiedad *simple* o una propiedad *derivada*. Además, las propiedades pueden ser *individuales* y *compartidas*. Las propiedades individuales son específicas de un objeto individual. Las propiedades compartidas son comunes a todos los objetos de la población del tipo. Las propiedades derivadas pueden ser calculadas a partir de las otras propiedades. Las simples o básicas no. Las propiedades son usualmente consultables y pueden ser también modificables. Las propiedades pueden tener parámetros y una *precondición* y una *restricción*. Una precondición es una expresión lógica construida sobre el resto de las propiedades del tipo que indica en qué condiciones es posible obtener el valor de la propiedad. Una *restricción* es una expresión lógica que indica los valores permitidos de la propiedad. Un tipo mutable tiene adicionalmente un conjunto de operaciones. Una *operación* es un mecanismo para cambiar las propiedades del tipo. Una operación puede tener una precondición, es decir una expresión lógica sobre las propiedades del tipo que indica cuando podemos usar la operación, y una poscondición que es una expresión lógica sobre las propiedades del tipo, antes y después de usar la operación y posiblemente del valor devuelto. que asegura cuáles serán los valores de las propiedades tras la llamada a la operación posiblemente a partir de los valores previos. En las poscondiciones los valores previos de las propiedades los representaremos añadiendo una prima a la propiedad. Un tipo tiene también un conjunto de *métodos de factoría* que son los mecanismos de creación de los valores del tipo. Y por último un tipo puede tener un *invariante***.** Un invariante es una expresión lógica construida sobre las de

propiedades del tipo que indica una restricción sobre los valores permitidos de las distintas propiedades. Una precondición la representaremos por @pre, una poscondición por @post, un invariante por @*inv*, una restricción por @*res* y el valor devuelto por una operación por @*ret.*

Es importante definir cuál es el *criterio de igualdad* de dos valores del tipo. De este criterio deduciremos los métodos *equals* y *hashCode*.

Si lo tuviera es importante indicar cuál es el *orden natural* del tipo. Este orden tiene que ser coherente con la igualdad definida para él.

Con todos ellos definiremos un nuevo tipo. Esta definición debería ser independiente de las posibles implementaciones.

El diseño de un tipo, su definición, se puede resumir en forma de un contrato. Un contrato es un documento en el que se define la funcionalidad ofrecida por un tipo y por lo tanto su uso por parte de sus posibles clientes.

Una forma práctica de ese contrato es crear una documentación tipo *Javadoc* dónde cada método y constructor del tipo se documenta con precondiciones, postcondiciones, invariantes, constructores, condiciones de disparo de excepciones, etc.

Veamos algunos elementos del diseño del tipo *List<E>.*

```
List<E>: Mutable.
   Propiedades:
       int size();
       Boolean contains(E e);
       E get(int i); Preconditions = i>=0, i < size();
       int indexOf(E e);
       List<E> subList(int i, in j);
          @pre i,j >=0,i,j < size(), i<j;
   Operaciones
       boolean add(E e);
          @post size()' == size()+1),get(size()-1).equals(e)
       boolean remove(E e); @post size()'==size()-1,
   Métodos de factoría:
      List<E> empty(): Crea una lista vacía
       List<E> of(E… elems): Crea una lista con los elementos
             dados
   Notas:
       La propiedad subList devuelve una vista de la lista
             original
```

Árboles

Un tipo recursivo es aquel que tiene alguna propiedad con valores del mismo tipo. Veremos aquí ejemplos de estos y técnicas para su diseño e implementación.

Los tipos de datos recursivos dan lugar a algoritmos recursivos específicos. Estos algoritmos están guiados por la estructura del tipo de datos recursivo. Veamos varios tipos recursivos: *BinaryTree, Tree* y *Exp*.

Veamos, en primer lugar, los árboles. Un árbol es un tipo de datos jerárquico, con una raíz y subárboles hijos. Vamos a definir dos tipos de árboles: binarios (*BinaryTree<E>*) y n-arios (*Tree<E>*). Los primeros tienen cero o dos hijos. Los segundos pueden tener un número variable de hijos. En particular los árboles n-arios pueden ser vacíos, tener un elemento, que llamaremos *etiqueta*, o una etiqueta y *n* hijos. Cada tipo anterior es un subtipo de árbol. Los árboles binarios pueden ser vacíos, tener una *etiqueta*, o una etiqueta y dos hijos. Cada tipo anterior es un subtipo de árbol. Un ejemplo de árbol nario es:

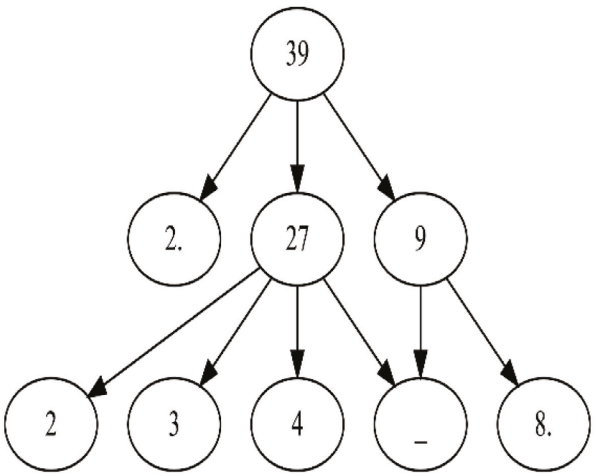


Ilustración 1: Un árbol n-ario

Un árbol n-ario se implementa mediante un interface sellado que permite varios subtipos: uno por cada tipo de árbol.

```
public sealed interface Tree<E> permits TEmpty<E>,
              TLeaf<E>, TNary<E>{
       public static <R> Tree<R> empty() {
              return new TEmpty<>();
       }
       public static <R> Tree<R> leaf(R label) {
              return new TLeaf<>(label);
       }
       public static <R> Tree<R> nary(R label,
              List<Tree<R>> elements) {
              if(elements.isEmpty()) return Tree.leaf(label);
              else return new TNary<R>(label,
                     new ArrayList<>(elements));
       }
       int size();
       List<Tree<E>> elements();
       int numElements();
       Optional<E> optionalLabel();
       …
}
```

El árbol vacío se diseña mediante un record implementa las propiedades del árbol

```
public static record TEmpty<E>() implements Tree<E> {
       public boolean isEmpty() {return true;};
       public Optional<E> optionalLabel() {
              return Optional.empty(); }
       public List<Tree<E>> elements() {
              return new ArrayList<>();}
       public int size() { return 0; }
       public int numElements() { return 0;}
       …
}
```

De la misma forma los árboles hoja y n-arios implementan la propiedades del árbol.

```
public static record TLeaf<E>(E label) implements Tree<E> {
       public boolean isEmpty() {return false;};
       public Optional<E> optionalLabel() {
              return Optional.of(this.label()); }
       public List<Tree<E>> elements() {
              return new ArrayList<>();}
       public int size() { return 1; }
       public int numElements() { return 0;}
       …
}
```

```
public static record TNary<E>(E label, List<Tree<E>> elements)
              implements Tree<E> {
       public boolean isEmpty() {return false;};
       public Optional<E> optionalLabel() {
              return Optional.of(this.label()); }
       public int size() {
              return  1+(int)elements().stream()
                     .mapToInt(x->x.size()).sum();
       }
       public Tree<E> element(int index) {
              return elements.get(index);
       }
       public int numElements() {
              return elements.size();
       }
       …
}
```

Algunas nociones sobre árboles:

- *El tamaño de un árbol* (*size*) es el número de etiquetas que tiene.
- Un *camino en un árbol* es cualquier secuencia de árboles*, t0, t1,…, tr−1,* tal que cada uno es padre del siguiente. Entre cada padre y uno de sus hijos existe una arista. La longitud del camino se define como el número de sus aristas.
- La *altura de un árbol* (*Height*) se define como la longitud del camino más largo que comienza en la raíz y termina en una hoja. La altura de una hoja será de cero. La altura de un árbol se define como la altura de su raíz.
- La *profundidad (Depht) de un árbol* dentro de otro se define como la longitud del camino que comienza en la raíz y termina en él. La profundidad de la raíz es cero. A la profundidad de un árbol también se la denomina *nivel* del árbol en el árbol que lo contiene.
- Un árbol binario se dice *ordenado* si es vacío, hoja o es binario, sus hijos están ordenados y su etiqueta es mayor que todas las de su hijo izquierdo y menor que las de su hijo derecho.
- Un árbol binario es *equilibrado* si es vacío, es hoja o si es binario sus hijos están equilibrados y sus alturas no difieren en más de una unidad.
- Un árbol es *raíz* si no tiene padre.

Para implementar las propiedades diseñamos un código estructurado alrededor de un *switch* sobre los subtipos de árbol. Los algoritmos sobre árboles, y en general sobre todos los tipos recursivos, son algoritmos recursivos. Para un árbol n-ario los casos base son los árboles *Empty, Leaf*. El caso recursivo el árbol *Nary*. Como en todos los algoritmos recursivos la solución de los casos base debe ser inmediata, la solución del caso recursivo se obtendrá componiendo las soluciones asociadas a cada uno de los hijos. Veamos ahora la definición recursiva de algunas propiedades anteriores.

Hay dos formas de hacerlo: implementando un método para cada uno de los subtipos de árbol o implementar un método con un switch.

El tamaño de un árbol se ha implementado arriba incluyendo el método correpondiente en cada subtipo de árbol. Alternativamente se puede codificar como:

```
int size(){
       return switch(tree) {
              case TEmpty<E> t -> 0 ;
              case TLeaf<E> t -> 1;
              case TNary<E> t -> 1 + t.elements().stream()
                     .mapToInt(x->size(x)).sum();
              };
}
```

Como vemos los casos base son el árbol sea vacío o una hoja, en cuyo caso el número de etiquetas es cero o uno respectivamente. En el caso recursivo el número de etiquetas de un árbol es la suma del número de etiquetas de sus hijos más uno.

Veamos el cálculo de la altura:

```
public static <E> Integer height(Tree<E> tree) {
       return switch(tree) {
       case TEmpty<E> t -> 0;
       case TLeaf<E> t -> 0;
       case TNary<E> t -> 1+ tree.elements().stream()
              .mapToInt(x->x.height())
              .max().getAsInt();
       };
}
```

Para el cálculo de la altura los casos base son, como antes, el árbol vacío o una hoja, en cuyo caso la altura es cero. En el caso recursivo la altura de un árbol es el máximo de las alturas de sus hijos más uno.

En el caso recursivo copiar un árbol es copiar cada uno de sus hijos y construir un árbol a partir de las copias. Se puede implementar una vista de un árbol. La copia de un árbol se puede hacer incluyendo un método en cada uno de los subtipos o de la siguiente forma.

```
public static <E> Tree<E> copy(Tree<E> tree) {
       return switch (tree) {
       case TEmpty<E> t -> Tree.empty();
       case TLeaf<E> t -> Tree.leaf(t.label());
       case TNary<E> t -> {
              List<Tree<E>> ch = t.elements().stream()
              .map(x -> copy(x)).collect(Collectors.toList());
              yield Tree.nary(t.label(), ch);
       }
       };
}
```

Es conveniente diseñar varios iteradores para un árbol. La estrategia para diseñarlos ya los explicamos anteriormente. Se ha propuesto dos iteradores: uno primero en profundidad por la izquierda y otro por niveles que nos va devolviendo las tuplas de los árboles de un nivel junto con su nivel.

Veamos los detalles del iterador en profundidad. Vemos que en el estado del iterador usamos una pila: *Stack<Tree<E>>.*

```
class DepthPathTree<E> implements Iterator<Tree<E>>,
       Iterable<Tree<E>> {
       public static <E> DepthPathTree<E> of(Tree<E> tree){
              return new DepthPathTree<E>(tree);
       }
       private Stack<Tree<E>> stack;
       public DepthPathTree(Tree<E> tree) {
              this.stack = new Stack<>();
              this.stack.add(tree);
       }
       @Override
       public Iterator<Tree<E>> iterator() {
              return this;
       }
       …
}
```

No hay más elementos si la pila está vacía:

```
public boolean hasNext() {
       return !this.stack.isEmpty();
}
```

La implementación del método next tiene algunos detalles a resaltar. En primer lugar se saca un elemento de la pila con *stack.pop().* El elemento obtenido es el que se va a devolver, pero antes de devolverlo hay que insertar sus hijos en la pila. Solo hay que hacer esta acción si el árbol es *Nary*. La disciplina sacar el primero de la pila y añadir sus hijos produce un recorrido en profundidad en preorden. Es decir, primero la etiqueta de la raiz antes que las de sus hijos y así sucesivamente para sus hijos.

Este recorrido tiene otras variantes. Para árboles binarios los recorridos alternativos. Uno es inorden: la etiqueta de la raiz se recorre despues de todas las de los hijos izquierdos y antes de todas las del hijo dererecho. Otro es postorden: la etiqueta de la raiz se recorre despues de todas las de los hijos izquierdos y antes de las del hijo derecho. Estos recorridos alternativos se pueden conseguir recursivamente y también iterativamente. Ambos se dejan como ejercicios.

```
@Override
public Tree<E> next() {
	Tree<E> actual = stack.pop();
	switch(actual) {
	case TNary<E> t:
		for(Tree<E> v:List2.reverse(t.elements())) {
			stack.add(v);
		}
		break;
	case TEmpty<E> t:break;
	case TLeaf<E> t:break;
	}
	return actual;
}
```

El recorrido por niveles usa una cola frente al recorrido en profundidad que usaba una pila.

```
class BreadthPathTree<E> implements Iterator<TreeLevel<E>>,
		Iterable<TreeLevel<E>> {
	public static <E> BreadthPathTree<E> of(Tree<E> tree){
		return new BreadthPathTree<E>(tree);
	}
	private Queue<TreeLevel<E>> queue;
	public BreadthPathTree(Tree<E> tree) {
		this.queue = new LinkedList<>();
		this.queue.add(TreeLevel.of(0,tree));
	}
	@Override
	public Iterator<TreeLevel<E>> iterator() {
		return this;
	}
```

Para mantener la información del nivel usamos el tipo *TreeLevel<E>* que guarda un árbol y su nivel.

```
record TreeLevel<E>(Integer level, Tree<E> tree){
	public static <R> TreeLevel<R> of(Integer level,
		Tree<R> tree){
		return new TreeLevel<R>(level,tree);
	}
}
```

No hay más elementos cuando la cola está vacía

```
@Override
public boolean hasNext() {
	return !this.queue.isEmpty();
}
```

El siguiente elemento es el que se saca de la cola pero antes de devolverlo debe introducir sus hijos, junto con el siguiente nivel, en la cola.

```
public TreeLevel<E> next() {
	TreeLevel<E> actual = queue.remove();
	switch(actual.tree()) {
	case TNary<E> t:
		for(TreeLevel<E> v:children(actual)) {
			queue.add(v);
		}
		break;
	case TEmpty<E> t:break;
	case TLeaf<E> t:break;
	}
	return actual;
}
```

La función chidren siguiente obtiene a partir de un árbol actual junto con su nivel los árboles hijos con el nivel siguiente.

```
private static <E> List<TreeLevel<E>> children(
	TreeLevel<E> actual){
	return actual.tree()
		.elements()
		.stream()
		.map(t->TreeLevel.of(actual.level()+1,t))
		.toList();
}
```

Algoritmos sobre árboles

Un árbol es un agregado de datos que puede ser tratado iterativa o recursivamente. Para los tratamientos iterativos necesitamos implementar uno o varios iteradores y con ellos podemos llevar a cabo todos los tipos de tratamientos iterativos que hemos visto. Para los tratamientos recursivos podemos implementar esquemas ya hemos estudiado.

Provistos de iteradores son agregados de datos que admiten los mismos tratamientos iterativos y recursivos que ya hemos visto. Los algoritmos sobre árboles usan la estructura *switch* que hemos visto antes. En este caso los árboles vacíos y hojas son los casos base y el resto los casos recursivos. En los casos recursivos hay que combinar la solución obtenida de los hijos para obtener la solución del problema.

Veamos algunos ejemplos:

1. *La etiqueta más pequeña, o la mayor, con respecto a un orden suponiendo el árbol no ordenado.*

Esta búsqueda se puede llevar a cabo con un iterador como para cualquier agregado de datos.

```
<E> E minLabel(Tree<E> tree, Comparator<E> cmp) {
      return tree.byDepth()
      .map(tt -> tt.optionalLabel())
      .filter(x-> x.isPresent())
      .map(x->x.get())
      .min(cmp).get();
}
```

2. *El problema anterior si el árbol está ordenado*

```
<E> Optional<E> minLabelOrdered(BinaryTree<E> tree,
            Comparator<E> cmp) {
      return switch (tree) {
      case BEmpty<E>  t -> Optional.empty();
      case BLeaf<E>  t ->  Optional.of(t.label());
      case BTree<E>  t && t.left() instanceof BEmpty<E>  tl ->
            Optional.of(t.label());
      case BTree<E>  t && t.left() instanceof BLeaf<E>  tl ->
            Optional.of(tl.label());
      case BTree<E>  t && t.left() instanceof BTree<E>  tl ->
            minLabelOrdered(tl, cmp);
      };
}
```

En el problema 1 la búsqueda tiene una complejidad $\Theta(n)$ en el caso peor. Si el arbol está ordenado se puede hacer una búsqueda más eficiente como vemos en el código arriba.

Para comprender el código anterior hemos de tener en cuenta que en un árbol binario ordenado las etiquetas del árbol izquierdo son menores que la etiqueta de la raiz y que no hay etiquetas repetidas. Si el árbol está vacío no hay solución por eso el algortimo devuelve *Optional<E>.*

De forma similiar podemos diseñar *maxLabelOrdered.*

3. Decidir si un árbol binario está ordenado.

Un árbol binario está ordenado si está vacío, es una hoja o es binario y su raiz es mayor que las etiquetas del hijo izquierdo y menor que las etiquetas del hijo derecho. Esto puede ser enunciado de otra forma: la etiqueta de un árbol es mayor que la del hijo izquierdo si no está vacío y menor que la del hijo derecho si no está vacío y además ambos hijos están ordenados.

Este problema puede resolverse mediante un algoritmo recursivo que usa *switch*.

```
<E> Boolean estOrdenado(BinaryTree<E> tree, Comparator<E> cmp) {
        BinaryType type = tree.getType();
        return switch (type) {
        case Empty -> true;
        case Leaf -> true;
        case BTree<E> t -> {
                Optional<E> lfLabel = maxLabelOrdered(t.left(), cmp);
                Optional<E> rgLabel = minLabelOrdered(t.right(), cmp);
                Boolean r = (lfLabel.isEmpty() ||
                        cmp.compare(lfLabel.get(), t.label()) < 0) &&
                             (rgLabel.isEmpty() ||
                        cmp.compare(rgLabel.get(), t.label()) > 0);
                yield r && isOrdered(t.left(), cmp) &&
                        isOrdered(t.right(), cmp);
        };
}
```

4. *Comprobar si un árbol (no suponemos que esté ordenado) contiene la etiqueta a*

Esta comprobación se hace como para cualquier agregado de datos usando un iterador.

```
<E> Boolean containsLabel(BinaryTree<E> tree, E label) {
      return tree.byDeph()
             .filter(t->!t.isEmpty())
             .map(t->t.optionalLabel().get())
             .anyMatch(e->e.equals(label));
}
```

Si el árbol está ordenado podemos usar un algoritmo recursivo más eficiente.

```
<E> Boolean containsLabelOrdered(BinaryTree<E> tree, E label,
             Comparator<E> cmp) {
      return switch (tree) {
      case BEmpty<E> t -> false;
      case BLeaf<E> t -> t.label().equals(label);
      case BTree<E> t && cmp.compare(label, t.label()) == 0 ->
                  true;
      case BTree<E> t && cmp.compare(label, t.label()) < 0 ->
                  containsLabelOrdered(t.left(), label, cmp);
      case BTree<E> t && cmp.compare(label, t.label()) > 0  ->
             containsLabelOrdered(t.right(), label, cmp);
      };
}
```

5. *Obtener la copia de un árbol n-ario*

```
<E> Tree<E> copy(Tree<E> tree) {
      return switch (tree) {
      case TEmpty<E>  t -> Tree.empty();
      case TLeaf<E>  t -> Tree.leaf(t.label());
      case TNary<E>  t -> Tree.nary(t.label(),t.elements()
             .stream().map(x->Trees.copy(x)).toList());
      };
}
```

Generalizaciones en algoritmos sobre árboles

En muchos problemas sobre árboles es conveniente generalizar el algoritmo. Esta generalización, como en los algoritmos recursivos vistos anteriormente, consiste en añadir parámetros adicionales. En muchos problemas estos parámetros son de dos tipos: parámetros de contexto que guardan la ubicación en el árbol y parámetros acumuladores que son usualmente valores de un tipo mutable adecuados para acumular el resultado. Veamos algunos ejemplos.

---

6. *Diseñe un algoritmo que, dado un árbol binario de enteros, determine el camino del árbol desde la raíz a una hoja no vacía tal que el producto de sus etiquetas sea máximo.*

El resultado pedido en este algoritmo es una tupla de un entero, el producto de los elementos de la lista, y una lista de enteros. Diseñamos para ello un record.

```
record TT(Integer p, List<Integer> ls) {
	public static TT of(List<Integer> ls) {
		Integer p = ls.stream().reduce(1,(x,y)->x*y);
		return new TT(p,ls);
	}
}
```

Con este tipo generalizamos el algoritmo.

```
TT maxCamino(BinaryTree<Integer> tree) {
	return maxCamino(tree,List.of());
}
```

La generalización la hacemos añadiendo un parámetro de contexto que lleve anotado la lista de etiquetas desde la raiz. Como vemos en la llamada original la lista está vacía. En éste tipo de problemas es conveniente trabajar con listas inmutables para no compartir las listas entre las distintas llamadas recursivas. Por eso inicializamos con List.of() y creamos una lista nueva al añadir un elemento.

```
TT maxCamino(BinaryTree<Integer> tree, List<Integer> camino) {
	return switch(tree) {
	case BEmpty<Integer> t -> null;
	case BLeaf<Integer> t ->
		TT.of(List2.addLast(camino,t.label()));
	case BTree<Integer> t -> {
		camino = List2.addLast(camino,t.label());
		TT left = maxCamino(t.left(),camino);
		TT right = maxCamino(t.right(),camino);
		yield Stream.of(left,right)
			.filter(x->x!=null)
			.max(Comparator.comparing(x->x.p()))
			.orElse(null);
		}
	};
}
```

7. *Diseñe un algoritmo que dado un árbol n-ario de caracteres devuelva un conjunto de cadenas de caracteres que contenga todas las cadenas palíndromas que se formen desde la raíz a una hoja no vacía.*

Ahora vamos a hacer la generalización incluyendo una lista para anotar el contexto y un conjunto que haga de acumulador como primera opción. Posteriormente veremos otra opción sin incluir el acumulador como parámetro de generalización.

```
Set<String> palindromas(Tree<Character> tree) {
	Set<String> st = Set2.of();
	palindromas(tree, "", st);
	return st;
}
```

El problema generalizado:

```
public static void palindromas(Tree<Character> tree,
              String camino, Set<String> st) {
       switch(tree) {
       case TEmpty<Character> t: break;
       case TLeaf<Character> t:
              Character label = t.label();
              camino = camino+label;
              if(IterativosyRecursivosSimples
                     .esPalindromo1(camino)) st.add(camino);
              break;
       case TNary<Character> t:
              label = t.label();
              camino = camino+label;
              for(Tree<Character> tt: t.elements())
                           palindromas(tt,camino,st);
              break;
       }
}
```

La segunda solución sin incluir un parámetro acumulador es de la forma:

```
public static Set<String> palindromas2(Tree<Character> tree) {
       return palindromas2(tree, "");
}
```

Los casos base del algoritmo generalizado son:

```
Set<String> palindromas2(Tree<Character> tree,String camino) {
       return switch(tree) {
       case TEmpty<Character> t -> Set2.of();
       case TLeaf<Character> t -> {
              Character label = t.label();
              camino = camino+label;
              if(IterativosyRecursivosSimples
                     .esPalindromo1(camino))
                           yield Set2.of(camino);
              else yield Set2.of();
       }
       …
```

El caso recursivo:

```
case TNary<Character> t -> {
              Character label = t.label();
              camino = camino+label;
              Set<String> st = Set2.of();
              for(Tree<Character> tt: t.elements()) {
                     Set<String> st1 = palindromas2(tt,camino);
                     st.addAll(st1);
              }
              yield st;
       }
       };
}
```

Expresiones y árboles de sintaxis abstracta

Una expresión es una combinación de constantes, variables, operadores y funciones de acuerdo con sus reglas particulares de prioridad y de asociación.

Cada expresión correctamente formada tiene un tipo y un valor. Este proceso de calcular el valor se llama evaluación.

Informalmente podemos dar las siguientes definiciones para las propiedades y métodos:

- Una expresión bien formada tiene un tipo que viene devuelto por t*ype().* Por simplicidad los tipos que consideraremos son: *Integer, Double, Boolean y String*. Los tipos de los operandos y los operadores de una expresión bien formada tienen que cumplir unas reglas que debemos especificar. El cálculo del tipo resultante más la comprobación de las reglas de tipo deben estar contenidas en el código de *type().*
- Una expresión que tiene un tipo tiene también un valor de tipo *R* que viene devuelto por *val().*
- La copia de una expresión es otra expresión igual pero no idéntica.
- A partir de una expresión se puede generar un fichero que represente el grafo de la expresión en formato *.gv (toDOT).*

Un ejemplo es:

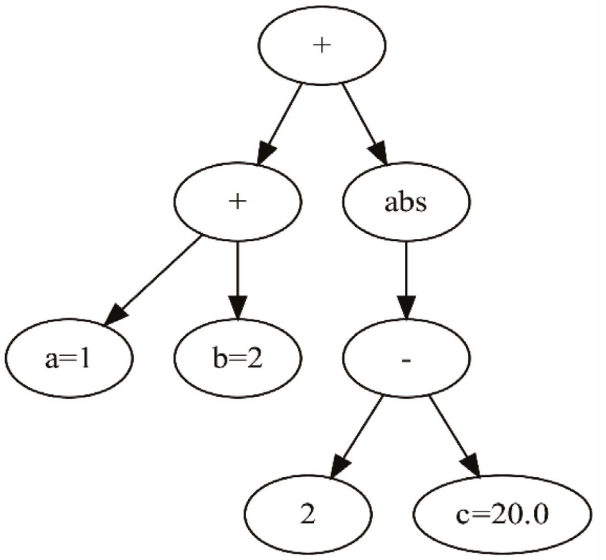


Ilustración 2: Arbol de sintaxis abstracta

El diseño del tipo podría ser:

```
public sealed interface Exp permits Unary, Binary, Var, Const,
       CallFunction, Nary{

       Set<Var> vars();
       Operator operator();
       Object value();
       Type type();
       String name();
       void toDot(PrintStream file, Map<Object,Integer> map);
       Boolean isConst();
       Exp simplify();
…
```

Los subtipos de Exp son *Unary, Binary, Var, Const, CallFunction, Nary.* Veamos la forma de implementar *Exp copy()* en cada subtipo concreto. En los tipos *Var, Unary, Binary* tenemos:

```
Exp copy() {
       return Exp.var(this.name(), this.type());
}
Exp copy() {
       return Exp.unary(this.op.copy(), this.operator);
}
Exp copy() {
       return Exp.binary(this.op1.copy(), this.op2.copy(),
              this.operator);
}
```

La recursividad queda distribuida en la estructura del tipo recursivo. El código completo se puede encontrar en el paquete Tipos Recursivos.

Se deja como ejercicio implementar el resto de las propiedades.

De forma similar a las expresiones un programa se puede representar como un tipo de datos diseñado adecuadamente.

El tipo de datos se diseña con las mismas ideas que la expresión. Sus valores se llaman *árboles de sintaxis abstracta (Abstract Sintax Tree, Ast).* Los detalles de este tipo pueden verse en el repositorio. Un programa es un bloque que a su vez es una lista de sentencias. El tipo *Sentence* lo diseñamos mediante un interface sellado con los subtipos permitidos *Assign, IfThenElse, While, Block.*

```
public sealed interface Sentence permits Assign, IfThenElse,
              While, Block{
       String name();
       void toDot(PrintStream file, Map<Object,Integer> map);
}
```

Las expresiones y los *árboles de sintaxis abstracta* pueden tener diversas propiedades y ser transformados en otro tipo de objetos. Una de ellas sería en código en algún lenguaje de bajo nivel adecuado para ejecutar ese programa. Un ejemplo de árbol de sintaxis abstracta es:

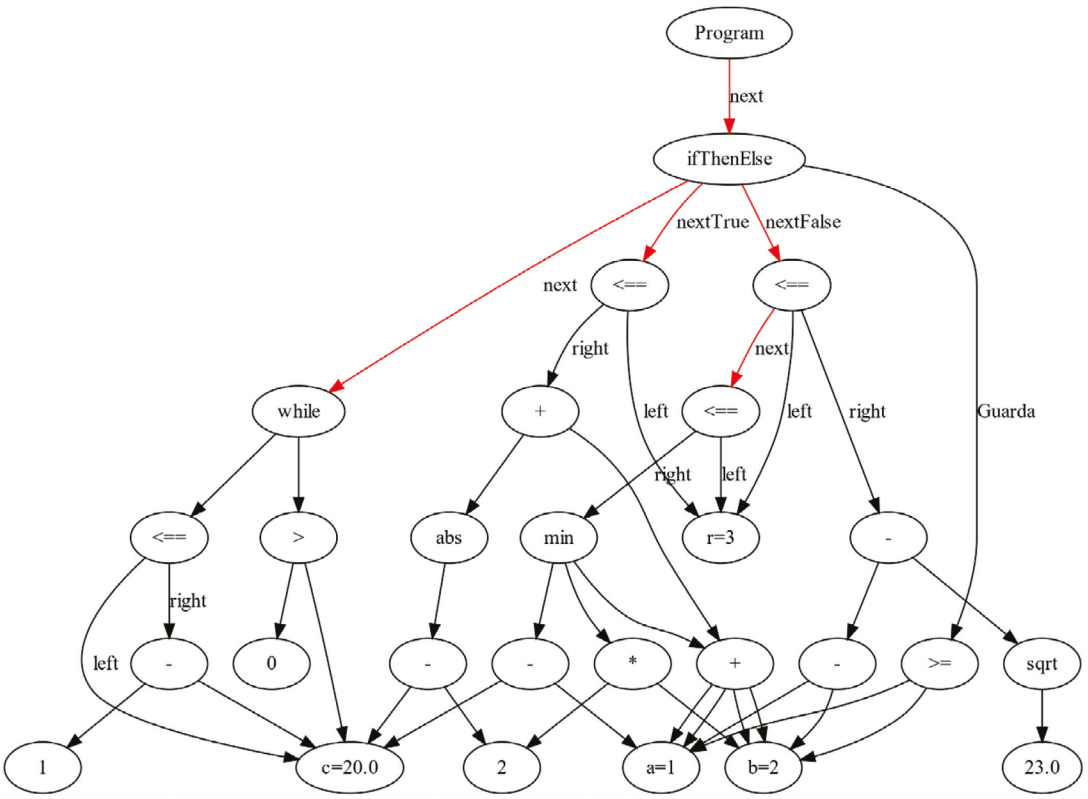


Ilustración 3: Arbol de sintaxis abstracta de un programa

Algoritmos sobre expresiones y árboles de sintaxis abstracta

Los algoritmos sobre expresiones y árboles de sintaxis abstracta siguen patrones similares.

En el repositorio se incluyen algoritmos para el cálculo del tipo de una expresión y de su valor.

Patrones sobre tipos recursivos

Una vez definido un tipo recursivo podemos construir valores aplicando sucesivamente los constructores disponibles. Pero puede interesarnos describir expresiones formadas por valores y variables, donde cada variable puede tomar un valor del tipo correspondiente. Esas expresiones las denominamos patrones. Sustituyendo en un patrón las variables por valores obtenemos nuevos valores que decimos que concuerdan con el

patrón (*match*). Un patrón es, también, un tipo recursivo. Veamos, por ejemplo, el patrón *BinaryPattern<E>* adecuado para describir patrones sobre árboles binarios:

```
public sealed interface  BinaryPattern<E> permits PTree<E>,
       PEmpty<E>, PLeaf<E>, PVarLabel<E>, PVarTree<E> {
       …
}
```

Los detalles del tipo pueden verse en el API. Un ejemplo de patrón podría ser:

p = -43.7(2.1,56(_e0(_,2), _T0))

Dónde *_e0* representa una variable de tipo *Double* y *_T0* una variable de tipo *BinaryTree<Double>.* La representación gráfica del ejemplo anterior es:

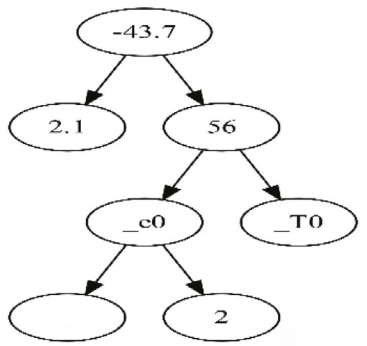


Ilustración 4: Patrón binario

Por otra parte, podemos tener el árbol binario:

b = -43.7(2.1,56(-27.3(_,2),78.2(3,4)))

Un patrón concuerda con un árbol si existen sustituciones en las variables del patrón que puedan dar lugar al árbol. Efectivamente el árbol *b* concuerda con el patrón p haciendo *_e0 = -27.3* y *_T0 = 78.2(3,4).*

Si tenemos un patrón adicional podemos transformar un árbol que concuerda con un patrón en otro descrito por un segundo patrón. Esto lo conseguimos sustituyendo en el segundo las variables por los valores

obtenidos en la operación de concordancia entre el árbol y el primer patrón. Si tenemos el patrón:

r = -43.7(_T0, _e0(_e0(_, _T0), _T0)) y
b2 = *transform*(b,p,r);
b2 =-43.7(78.2(3.0,4.0),-27.3(-27.3(_,78.2(3.0,4.0)),78.2(3.0,4.0)))

Algunos patrones sobre árboles:

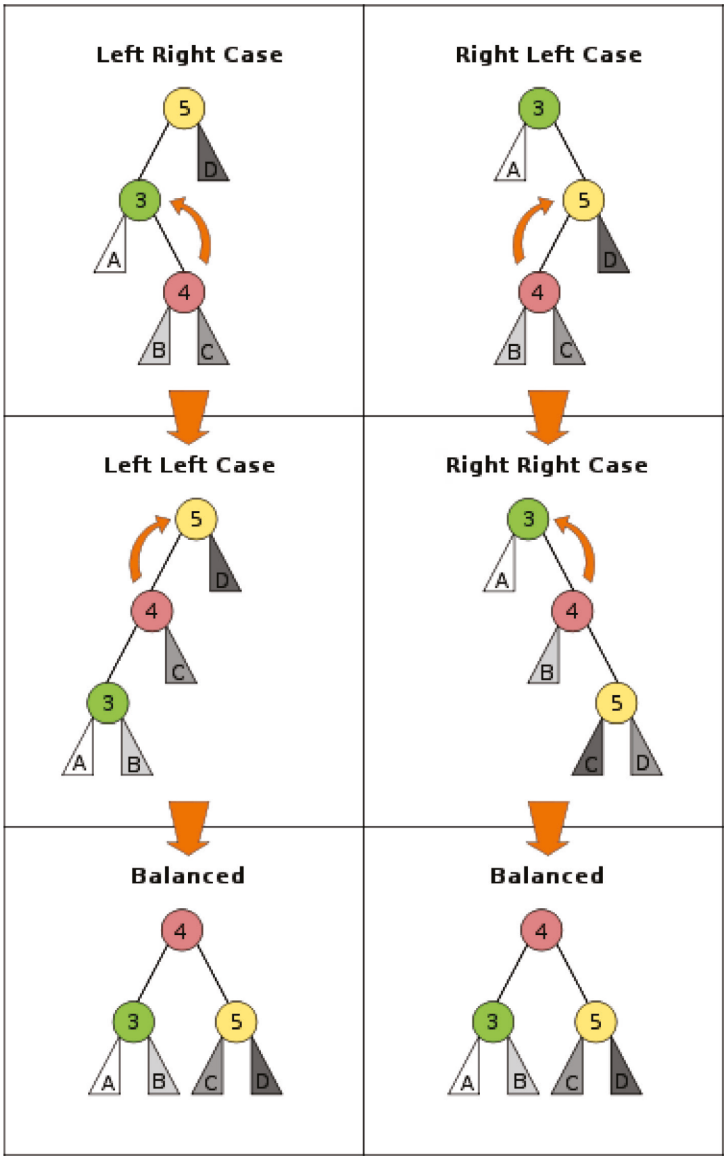


Ilustración 5: Patrones de árboles no equilibrados

Estos patrones tienen la representación textual:

```
leftRight === "_e5(_e3(_A,_e4(_B,_C)),_D)"
rightLeft === "_e3(_A,_e5(_e4(_B,_C),_D))"
leftLeft === "_e5(_e4(_e3(_A,_B),_C),_D)"
rightRight === "_e3(_A,_e4(_B,_e5(_C,_D)))"
result === "_e4(_e3(_A,_B),_e5(_C,_D))"
```

Las operaciones de *match* y *transform* vienen implementadas en el API.

Los patrones sobre árboles binarios son muy adecuados para implementar el método equilibrate de un *BinaryTree*. Un árbol binario que está ordenado y equilibrado queda en alguna de estas situaciones tras añadir o eliminar una nueva etiqueta. Cada uno de ellos, y el árbol equilibrado, puede ser representada por el patrón:

La implementación del método *equilibrate* consiste en identificar el tipo de situación de las cuatro anteriores, hacer el *matching* y luego transformar el árbol según el patrón según *result*. La identificación que cuál de las cuatro posibilidades es la adecuada se hace según la diferencia de alturas en los hijos izquierdo y derecho de niveles 1 y 2.

Representación textual de los tipos recursivos

Para la reconstrucción de instancias de tipos recursivos a partir de sus representaciones textuales necesitamos técnicas y herramientas adecuadas. Estas técnicas se basan en el diseño de una gramática para especificar la descripción textual de los valores de un tipo. Una herramienta adecuada es Antlr4. No entraremos en mucha profundidad en esta herramienta de la que recomendamos su estudio. Solo presentaremos las gramáticas para representar los tipos *Tree* y *Exp*. Los detalles pueden verse en el repositorio en que también aparece el tipo *BinaryTree* y *Program*.

```
nary_tree : '_'   #emptyTree
          | label     #labelTree
          | treeLabel=label '(' nary_tree (',' nary_tree)* ')'
#naryTree
          ;

label : ('+'|'-')? INT         #intLabel
      | ('+'|'-')? DOUBLE      #doubleLabel
      | ID                     #idLabel
      ;
ID    :  ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_'
| '0'..'9')*
      ;
INT   : ('0'..'9')+
      ;
DOUBLE   : INT '.' INT?
         ;
WS    : [ \t\r\n]+ -> skip
      ;
```

Como podemos ver una gramática se compone de reglas. Una regla tiene una parte izquierda, su nombre, y una parte derecha que describe la estructura del nombre indicado. Las que tienen el nombre en mayúsculas se llaman reglas léxicas. En la parte derecha de una regla se usan los operadores de las expresiones regulares: **, +, ?,[], ()*. Que significan respectivamente repetición cero o más veces, repetición una o más veces, opción sí o no, clase de caracteres y un bloque. Los caracteres entre comillas simples representan texto final. Hacemos notar la diferencia ente *()* y *'(' ')'*. Los primeros indican un bloque. Los segundos dos caracteres.

Es importante ver que las reglas son recursivas. Es decir en una regla se puede hacer referencia a ella misma o a otra.

La gramática para el tipo *Exp* es:

```
exp   : op='(int)' right=exp                                  #unaryExpr
      | op='(double)' right=exp                               #unaryExpr
      | op=('+'|'-' |'!') right=exp                           #unaryExpr
      | left=exp op=('*'|'/'|'%') right=exp                   #binaryExpr
      | left=exp op=('+'|'-') right=exp                       #binaryExpr
      | left=exp op=('<='|'<'|'>='|'>') right=exp             #binaryExpr
      | left=exp op=('!='|'=') right=exp                      #binaryExpr
      | left=exp op=('&&'|'||') right=exp                     #binaryExpr
      | '(' exp ')'                                           #parenExpr
      | name=ID '(' real_parameters? ')'                      #callExpr
      | id=ID                                                 #idExpr
      | DOUBLE                                                #doubleExp
      | INT                                                    #intExpr
      ;
…
```

Usando Antlr4, las gramáticas anteriores y un patrón de diseño conocido como *Visitor* podemos obtener los valores de uno de los tipos anteriores a partir de su representación textual. Los detalles pueden verse en el repositorio.

Implementación de Tipos

Arriba hemos visto los pasos para diseñar un tipo. Con el diseño indicamos todos los detalles públicos del tipo que recogemos en un contrato. Tras el diseño pasamos a la implementación donde concretamos los detalles internos del mismo.

Para implementar la funcionalidad expresada por un contrato en Java, debemos construir una clase que será la implementación del contrato. Una implementación es una relación entre un *contrato*, que puede ser especificado por una interface más un conjunto de precondiciones y poscondiciones, y una *clase* concreta. La clase deberá cumplir el contrato definido para un tipo dado.

Implementar un tipo es decidir un conjunto de propiedades privadas y un invariante sobre la mismas. Un tipo se implementará mediante una clase. Las propiedades privadas se convertirán en atributos y a partir de ellos debemos diseñar los constructores y métodos (públicos y privados) y los métodos de factoría. Puede haber tipos privados diseñados para la implementación.

Una técnica muy frecuente de implementación es la *reutilización* de otro u otros tipos para delegar en ellos algún tipo de funcionalidad. Mediante esta técnica creamos atributos privados de los tipos reutilizados e

invocamos sobre ellos los métodos necesarios para construir la funcionalidad requerida. Otra técnica de implementación es la *herencia*. Consiste en partir de un tipo y extenderlo para tener un nuevo tipo. Esta técnica es menos útil que la reutilización, como técnica de implementación, porque el tipo que va a ser implementado debe ser un subtipo del que queremos partir y eso solo se cumple en casos muy particulares.

Veamos como ejemplo *AList<E>* la implementación de una lista de tamaño variable *List<E>.* Los detalles del código pueden verse en el repositorio.

```
AList<E>: Mutable. Es un tipo adecuado para implementar una
       lista mediante un array cuyo tamaño se ajusta
       automáticamente al tamaño de los datos.
       Propiedades Privadas
              E[] data;
              int tam;
              int size;
       Invariante:
              tam = data.lenght,
              size < tam,
              size = número de elementos;
       Operaciones privadas:
              E[] grow(E[] old, int newTam);
       Métodos de Creación:
              AList<E> empty();
              AList<E> of(E[] data, int size);
```

```
       Propiedades:
              int size();
              Boolean contains(E e);
              E get(int i); Preconditions = i>=0, i < size();
              int indexOf(E e);
              AList<E> subList(int i, in j);
                 @pre i,j >=0,i,j < size(), i<j;
              Devuelve una vista
```

```
       Operaciones
              boolean add(E e);
                @post &ret == (size()' == size()),contains(e)'
              boolean remove(E e);
                @post &ret== (size()'==size()), !contains(e)'
       Notas:
              La propiedad subList devuelve una vista de la
              lista original
```

El primer detalle para tener en cuenta es que el tamaño de array de datos (data) debe ser suficiente para almacenar los datos. Es decir, *size < tam.* Esta restricción del invariante se puede mantener llamando a la operación *grow* al comienzo de la operación *add.*

Es importante discutir las complejidades de cada una de las operaciones y propiedades. En particular la de *sublist*. Los detalles de esta implementación nos permiten ver la manera de implementar vistas. Un objeto de tipo *V* es una vista de otro objeto de tipo *T* si ambos objetos comparten el estado completo o una parte de este de tal forma que el cambio de las propiedades del objeto de tipo *T* implica un cambio de las propiedades del objeto de tipo *V* y viceversa. Una sublista es una vista de una lista.

Grafos

El tipo *grafo* está formado un conjunto de vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto. Es posible que haya varias aristas entre cada par de vértices y aristas que unen un vértice consigo mismo. Tanto los vértices como las aristas son objetos que pueden tener información propia. Denominaremos V el tipo de los vértices y E el tipo de las de las aristas.

Las aristas pueden ser de dos tipos: *aristas dirigidas* y *aristas no dirigidas*. Las primeras tienen una orientación es decir tienen un vértice origen y un vértice destino. Las segundas no tienen orientación, sólo tienen dos vértices que son sus extremos. Los grafos que sólo tienen aristas dirigidas se denominan *grafos dirigidos* y los que tienen aristas dirigidas y no dirigidas *grafos mixtos*.

Cuando conocemos a priori el conjunto de vértices y las aristas de un grafo decimos que tenemos un *grafo explícito*. En este caso describimos el grafo de forma global y disponemos en memoria de toda la información sobre el conjunto de vértices y aristas y sus informaciones asociadas.

Hay otros grafos, que llamaremos *grafos implícitos*, de los que no conocemos a priori todos sus vértices y aristas. Este tipo de grafos están

descritos por las propiedades que cumplen los vértices y por una función que dado un vértice nos calcula sus vecinos y las aristas que lo conectan a ellos. El grafo estará constituido por el conjunto de vértices alcanzables desde el vértice dado. Los vértices del grafo se irán descubriendo a medida que el grafo se vaya recorriendo.

Términos de un grafo

Veamos, en primer lugar, un conjunto de términos sobre grafos:

- *Camino*: Secuencia de vértices donde cada uno está conectado con el siguiente por una arista.

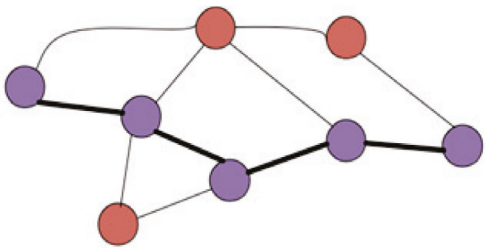


Ilustración 6: Camino en un grafo

- *Longitud de un Camino*: Se llama longitud del camino al número de aristas que contiene.
- *Camino Cerrado*: Camino donde el vértice primero coincide con el último.

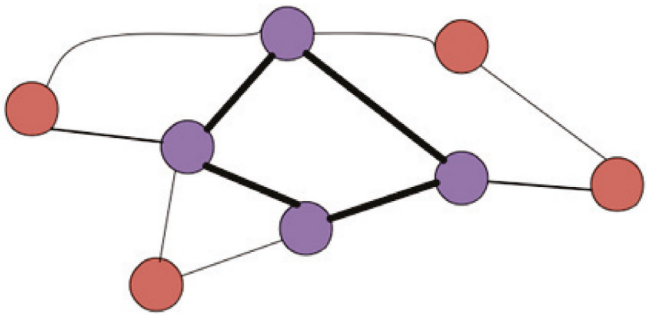


Ilustración 7: Camino cerrado

- *Camino simple*: No tiene vértices repetidos a excepción del primero que puede ser igual al último.

- *Bucle*: Arista que une vértice consigo mismo.
- *Ciclo*: Camino cerrado donde no hay vértices repetidos salvo el primero que coincide con el último.
- *Grafos Dirigidos*: Si todas sus aristas son dirigidas.
- *Grafos No Dirigidos*: Si todas sus aristas son no dirigidas.
- *Grafos Mixtos*: Puede haber aristas dirigidas y no dirigidas
- *Grafos simples*: Desde un vértice v1 hasta v2 hay como máximo una arista que puede ser dirigida o no dirigida. No pueden contener bucles.

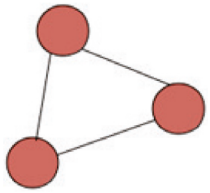


Ilustración 8: Bucle en un grafo

- *Multigrafos*: Son grafos donde puede haber más de una arista entre dos vértices.
- *Pseudografos*: Son grafos donde puede haber más de una arista entre dos vértices y bucles.
- *Grafo Completo*: Grafo simple en el que hay una arista entre capar de vértices cada par de vértices

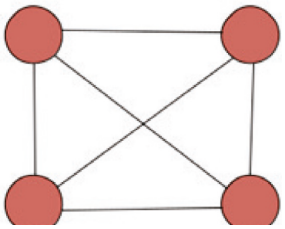


Ilustración 9: Grafo completo

- *Grafo Acíclico*: Grafo que no tiene ciclos.
- *Grafo Conexo*: Grafo no dirigido en el que hay al menos un camino entre cada dos nodos.
- *Grafo Débilmente Conexo*: Grafo dirigido en el cual al remplazar cada arista dirigida por una no dirigida resulta un grafo no dirigido conexo.

- *Grafo Fuertemente Conexo*: Grafo dirigido dónde para cada par de vértices u, v existe un camino dirigido desde u a v y otro de v hasta u.
- *Árbol Libre*: Es un grafo simple conexo y sin ciclos
- *Bosque*: Es un grafo sin ciclos. Es un conjunto de árboles libres.

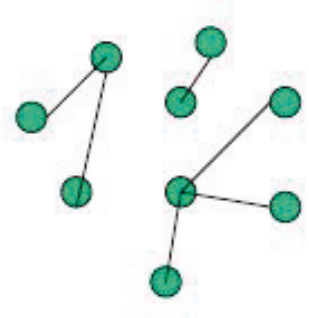


Ilustración 10: Bosque

- *Subgrafo*: Un grafo g1 es subgrafo de otro g2 si los vértices y aristas de g1 están incluidos en los vértices y aristas de g2.
- *Componentes Conexas*: Subgrafos maximales de un grafo no dirigido formados por los vértices entre los que hay al menos un camino, más las aristas que los conectan.
- *Componente fuertemente conexa de un grafo dirigido*: Es un subgrafo maximal fuertemente conexo de un grafo dirigido.
- *Componente débilmente conexa:* Es un subgrafo maximal débilmente conexo de un grafo dirigido.

Para todos estos tipos nos basaremos en *JGraphT*, http://jgrapht.org/. Es importante conocer los siguientes tipos:

Graph<V,E>: Un grafo con vértices de tipo V y aristas de tipo E. Es importante tener en cuenta que los tipos concretos que ocupen el papel de V y de E deberían ser tipos inmutables y tener diseñado un *equals* y un *hashCode*. Dos vértices del grafo (o dos aristas) tienen que ser distintos.

GraphPath<V,E>: Un camino en un grafo.

Formatos de ficheros para representar grafos y su visualización

Hay diferentes tipos de formatos para representar grafos. Aquí veremos uno de los más conocidos: el formato *dot*. Además, usaremos otra representación sencilla que denominaremos *lsi.* Ambos formatos tienen propósitos diferentes. El segundo tiene como objetivo representar la información asociada a los datos del grafo. Es decir, las propiedades de los vértices y las aristas, el conjunto de vértices el conjunto de aristas y sus conexiones. El primero está más orientado a representar la información que se necesita para representar gráficamente un grafo. Es decir, posibles colores, figuras geométricas asociadas a los vértices, estilo de las aristas, etc. Veamos los detalles de cada uno de ellos.

Los ficheros estilo *lsi* para representar grafos tienen una estructura con dos secciones: la sección de vértices y la de aristas. Ambas secciones están encabezadas por #VERTEX# y #EDGE#. Un fichero *lsi* es:

```
#VERTEX#
Sevilla,1535379
Cadiz,647241
Huelva,230466
Cordoba,363326
…
#EDGE#
Cadiz,Huelva,Mar,150.2
Sevilla,Huelva,A-49,92.850
Sevilla,Cordoba,A-4,140.543
Sevilla,Cadiz,AP-4,121.066
Sevilla,Antequera,A-92,159.947
Cordoba,Jaen,A-316,108.120
Cordoba,Granada,N-432,207.838
…
```

El primer campo de cada vértice es un identificador y debe ser único. Este primer campo puede ser también una propiedad del vértice. Los siguientes campos son propiedades del vértice. En la sección de aristas los dos primeros campos son índices de vértices, los siguientes campos son propiedades de la arista correspondiente.

Para poder diseñar un algoritmo general que permita construir grafos de distintos tipos a partir de la información anterior diseñamos el método *newGraph* de la clase *GraphReader.*

```
<V, E, G extends Graph<V,E>> G newGraph(
        String file,
        Function<String[],V> vf,
        Function<String[],E> ef,
        Supplier<G> creator,
        Function<E,Double> ew);
}
```

Los elementos de ese método son:

- *String file:* El nombre del archivo
- Function<String[],V> vf: Una función que construye un vértice a partir de las partes de una línea de fichero de la sección *#VERTEX#*
- Function<String[],E> *ef:* Una función que construye un vértice a partir de las partes de una línea de fichero de la sección *#EDGE#*
- *Supplier<G> creator:* Un mecanismo para crear un grafo vacío del tipo que queramos construir
- *Function<E,Double> ew):* Una función que dada una arista calcula su peso. Esta función puede ser null.

Cada vértice y cada arista deben ser de tipos que debemos diseñar. El diseño debe tener en cuenta que dos vértices o dos aristas del grafo no pueden ser iguales. Para la implementación de los tipos es adecuado usar record porque deben ser tipos inmutale y además tenemos implementado un equals, hashCode, constructor, etc. Veamos el diseño de Ciudad y Carretera. Diseñamos un método de factoría que construya un objeto a partir de un *String[]* que contiene los trozos de una línea del fichero. La primera propiedad actua con identificador y debe ser única en los datos de entrada del fichero.

```
public record Ciudad(String nombre, Integer habitantes)  {
	public static Ciudad ofFormat(String[] formato) {
		String nombre = formato[0];
		Integer habitantes =
			Integer.parseInt(formato[1]);
		return new Ciudad(nombre,habitantes);
	}
	public static Ciudad of(String nombre,
			Integer habitantes) {
		return new Ciudad(nombre,habitantes);
	}
	@Override
	public String toString() {
		return this.nombre;
	}
}
```

El tipo Carretera los diseñamos de manera similar pero ahora debemos generar automáticamente una nueva propiedad para conseguir que dos carreteras sean distintas aunque tengan todas las propiedades leidas del fichero iguales.

```
public record Carretera(Integer id, Double km,String nombre) {
	private static int num =0;
	public static Carretera of(Double kms) {
		Double km = kms;
		String nomb = null;
		Integer id = num;
		num++;
		return new Carretera(id, km, nomb);
	}
	…
}
```

Como vemos la técnica consiste en mantener una variable num que vaya proporcionando números enteros consecutivos cada vez qu se construye un objeto. Junto con el método de factoría anterior diseñamos otro que cosntruya objetos a partir de los fragmentos de una línea del fichero.

```
public static Carretera ofFormat(String[] formato) {
	Double km = Double.parseDouble(formato[3]);
	String nomb = formato[2];
	Integer id = num;
		num++;
		return new Carretera(id, km, nomb);
	}

	@Override
	public String toString() {
		String nn = this.nombre==null?"":this.nombre+",";
		return nn+this.km+")";
	}


}
```

En algunos casos las propiedades de los vértices pueden estar repetidas. Si fuera así añadimos un índice para cada vértice. El fichero quedaría asi:

```
#VERTEX#
1,Sevilla
2,Córdoba
3,Cádiz
4,Málaga
#EDGE#
1,2,180.
1,3,120.
2,4,140.
4,1,210.
```

La representación visual de un grafo ya construido es un tema mucho más complicado, pero de gran interés. Los grafos construidos mediante *jgrapht* pueden ser exportados a un fichero con estructura *gv* y visualizados con la herramienta anterior. Esto puede hacerse con el método *toDot* de la clase *GraphColors*:

```
<V,E> void toDot(Graph<V,E> graph,
	String file,
	Function<V,String> vertexLabel,
	Function<E,String> edgeLabel);
```

Donde los parámetros indican

- *Graph<V,E> graph:* El grafo a exportar
- *String file:* El fichero *gv* al que se exporta

- Function<V,String> vertexLabel: Las etiquetas de los vértices
- Function<E,String> edgeLabel: Las etiquetas de las aristas

En el repositorio se encuentran otros métodos *toDot* para indicar propiedades como colores y detalles de los vértices y aristas.

```
<V,E> void toDot(Graph<V,E> graph,
       String file,
       Function<V,String> vertexLabel,
       Function<E,String> edgeLabel,
       Function<V,Map<String,Attribute>> vertexAttribute,
       Function<E,Map<String,Attribute>> edgeAttribute);
```

Los parámetros adicionales *vertexAttribute* y *edgeAttribute* proporcionan los atributos gráficos de aristas y vértices. La clase GraphColors proporciona métodos para obtener colores, formas o estilos para asignar a vértices y aristas. Estos colores, formas o estilos pueden ser escogidos mediante un predicado o una función. Un ejemplo es:

```
GraphColors.toDot(graph,
       "ficheros/andaluciaAStar.gv",
       x->x.nombre(),
       x->String.format("%.2f",x.km()),
       v->GraphColors.color(Color.black),
       e->GraphColors.styleIf(Style.bold,
              carreteras.contains(e)));
```

Este método genera un fichero con una estructura similar al siguiente que conteniene información gráfica específica para indicar el tipo de vista del grafo que queremos. Estos ficheros tienen extensión *gv* por el tipo de formato que usan.

```
graph andalucia {
       size="100,100";
       Sevilla [style = filled, color = blue];
       Cordoba -- Sevilla [label= "100+2", color= red,
              style= bold];
       Granada -- Cordoba [label= 10, color= red, style= bold];
       Sevilla -- Huelva [label= 15, color= red, style= bold];
       Sevilla -- Cadiz [label= 30];
       Almeria -- Granada [label= 25, color= red, style= bold];
       Granada -- Jaen[label= 32];
       Almeria -- Malaga [label= 51, color= red, style= bold];
}
```

El fichero se compone de una cabecera y un conjunto de líneas. Cada línea describe un vértice con sus propiedades entre *[]* o una arista con sus propiedades también entre *[]* e indicando los vértices que conecta. El símbolo – representa una arista no dirigida. El símbolo -> (que no aparece en el texto) representa una arista dirigida.

Las propiedades y sus valores se representan por pares nombre = valor. Hay muchas propiedades predefinidas que tienen una semántica ya fijada.

Para visualizar el gráfico con los detalles indicados en el fichero de extensión *gv* es necesario aplicar algunos algoritmos complejos para calcular la disposición (*layout*) más adecuada de los vértices y aristas. Podemos usar la herramienta *ZGRViewer* (visualizador de *Graphviz - Graph Visualization Software*) que lee ficheros con esa extensión y visualiza el grafo correspondiente siendo posible ver el resultado como una imagen. La documentación puede encontrarse en:

http://www.graphviz.org/Download.php

El gráfico siguiente es el resultado de procesar el fichero anterior con la herramienta comentada.

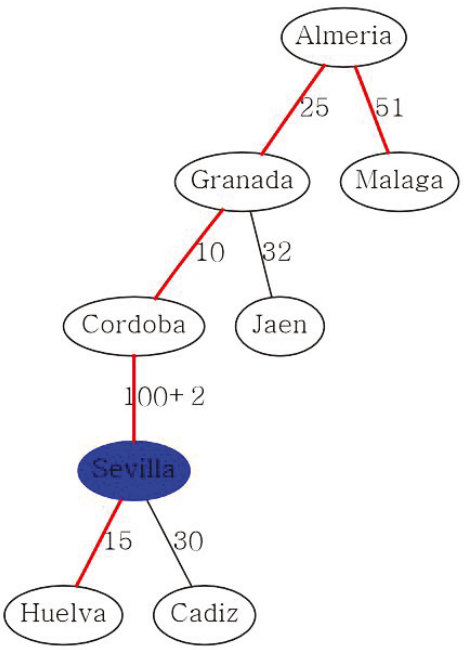


Ilustración 11: Visualización de grafos

Vistas de grafos

Dado un grafo podemos obtener diversas vistas de este que están disponibles en el repositorio. Algunas de ellas son:

- *SubGraphView<V,E>*: Es una vista de un grafo (*Graph<V,E>*). La vista sólo contiene los vértices y aristas que cumplen un predicado. Alternativamente la vista solo contiene los vértices de un conjunto dado y las aristas que los conectaban en el grafo original.
- *CompleteGraph<V,E>:* Es una vista inmutable de un grafo (*Graph<V,E>*). La vista ofrece un grafo completo es decir que contiene aristas entre todos los vértices. Si el grafo original no tenía una arista la vista ofrece una con un peso dado.
- *IntegerVertexGraphView<V,E>:* Es una vista inmutable de un grafo (*Graph<V,E>*) de tipo *Graph<Integer,SimpleEdge<Integer>>*. La vista tiene todos sus vértices de tipo *Integer*, las aristas de tipo *SimpleEdge<Integer>* con los pesos de las aristas del grafo original. Adicionalmente la vista ofrece dos propiedades adicionales: *vertex(i)*, que da el vértice de índice *i* e *index(v)* que da el índice asociado a *v*.

Recubrimientos y componentes conexas

Veamos ahora algunos algoritmos sobre grafos que están disponibles en *jgrapht*.

Recubrimiento mínimo

Un bosque de recubrimiento mínimo de un grafo es subgrafo que sea un bosque, incluya todos los vértices del grafo y minimice la suma total de los pesos de las aristas escogidas. El algoritmo devuelve el conjunto de aristas que definen el bosque.

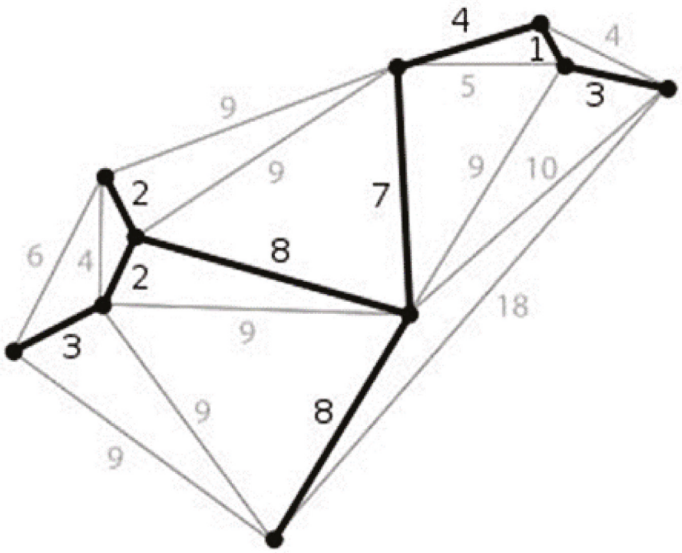


Ilustración 12: Recubrimineto mínimo

Aplicación del Recubrimiento Mínimo

La aplicación usual es un problema de diseño de una red telefónica. Suponga que tiene varias ciudades que desea conectar mediante líneas telefónicas. La compañía telefónica cobra diferentes cantidades de dinero para conectar diferentes pares de ciudades. ¿Qué ciudades conectar para que todas estén conectadas entre sí al mínimo precio?

Algoritmo de Kruskal

El algoritmo de *Kruskal* busca un *bosque de recubrimiento mínimo* de un grafo y devuelve el conjunto de aristas del recubrimiento. Es aplicable para grafos no conexos o conexos (en ese caso devuelve un árbol de recubrimiento mínimo.

Dado un grafo la forma de obtener las aristas del recubrimiento mínimo es:

```
SpanningTreeAlgorithm<Carretera> ast =
      new KruskalMinimumSpanningTree<>(graph);
SpanningTree<Carretera> r = ast.getSpanningTree();
System.out.println(r.getEdges());
```

El resultado del algoritmo es un *SpanningTree<Carretera>* que entre otras propiedades tiene *r.getEdges()* el conjunto de aritas que constituyen el recubrimiento mínimo.

Algoritmo de Prim

El algoritmo de *Prim* busca un *árbol de recubrimiento mínimo* de un grafo y devuelve el conjunto de aristas del recubrimiento. Es aplicable para grafos no conexos.

Recubrimiento de vértices

El problema del recubrimiento mínimo anterior consistía en escoger un subconjunto de aristas cuyos extremos incluyeran todos los vértices y cuya suma de pesos fuera mínima. Podríamos llamarlo recubrimiento mínimo de aristas. De forma similar el problema llamado recubrimiento de vértices consiste en escoger el mínimo número de vértices que tocan todas las aristas del grafo.

Los vértices llenos son el recubrimiento de vértices mínimo del grafo.

Ilustración 13: Recubrimineto de vértices

Aplicación

Si tenemos un grafo cuyas aristas representan carreteras, los vértices representan cruces entre ellas, pretendemos colocar cámaras de seguridad en algunos cruces de tal manera que puedan ver todas las carreteras que acceden al cruce. Si pretendemos colocar el mínimo

número de cámaras podemos obtener la solución mediante recubrimiento de vértices.

Algoritmo

Hay varios algoritmos para resolver este problema que se pueden encontrar en el API.

```
Map<Ciudad,Double> habitantes = Map2.of(x->1./x.habitantes());
VertexCoverAlgorithm<Ciudad> vc = new
      RecursiveExactVCImpl<>(graph,habitantes);
VertexCover<Ciudad> r2 = vc.getVertexCover();
System.out.println(r2.toString());
```

El resultado del algoritmo, un *VertexCover<Ciudad>,* es un conjunto de vértices con algunas propiedades adicionales. En el ejemplo anterior se ha hecho un recubrimiento de vértices minimizando la suma de la propiedad *1./x.habitantes().*

Componentes conexas

Un grafo no dirigido es conexo si entre cada dos vértices existe un camino del grafo. Una *componente conexa de un grafo no dirigido* es un *subgrafo maximal* conexo. Un grafo conexo tiene una sola componente conexa.

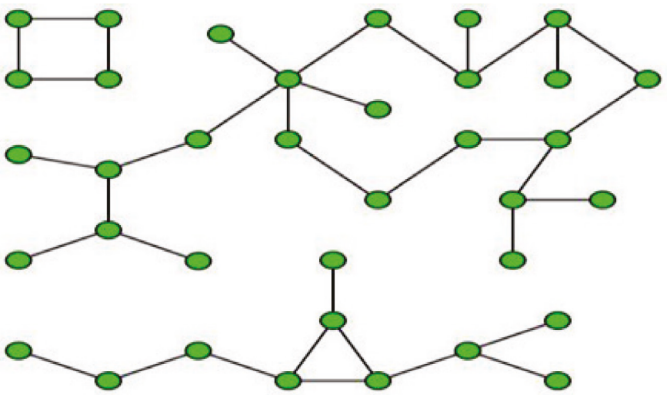


Ilustración 14: Componentes conexas

Un *grafo dirigido es débilmente conexo* si al reemplazar cada arista dirigida por una no dirigida resulta un grafo conexo. Un grafo dirigido es fuertemente conexo si para cada par de vértices *u, v* existe un camino

dirigido de *u* a *v* o de *v* a *u*. Una *componente fuertemente conexa* de un grafo dirigido es un subgrafo maximal fuertemente conexo de un grafo dirigido. Una *componente débilmente conexa* es un subgrafo maximal débilmente conexo de un grafo dirigido.

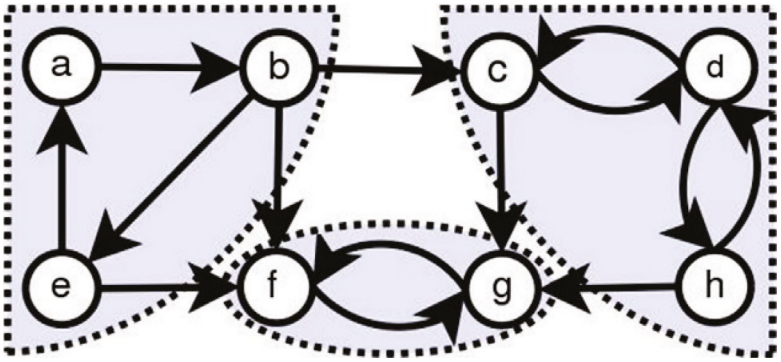


Ilustración 15: Componentes fuertemente conexas

Algoritmos

La clase *ConnectivityInspector* calcula componentes conexas de un grafo no dirigido y componentes débilmente conexas de un grafo dirigido. Puede responder a las siguientes preguntas:

- Conjunto de vértices conectados a un dado
- Lista de componentes conexas. Donde una componente conexa viene dada por un conjunto de vértices
- ¿Es conexo un grafo?
- ¿Existe un camino entre dos vértices dados?

El tipo *StrongConnectivityInspector* calcula las componentes fuertemente conexas de un grafo dirigido. Puede responder a las siguientes preguntas

- Lista de componentes fuertemente conexas. Donde una componente fuertemente conexa viene dada por un conjunto de vértices
- ¿Es fuertemente conexo un grafo?

Ejemplos

Un ciclo es un camino cerrado en un grafo. Se pueden encontrar en el API varios algoritmos para obtener los ciclos de un grafo, si hay caminos entre dos vértices, si el grafo es conexo, etc.

Veamos algunos ejemplos:

1. *Decidir un grafo no dirigido es conexo*

```
ConnectivityInspector<Ciudad, Carretera> a =
       new ConnectivityInspector<>(graph);
Boolean r1 = a.isConnected();
```

2. *Encontrar las componentes conexas de un grafo no dirigido*

```
ConnectivityInspector<Ciudad, Carretera> a =
       new ConnectivityInspector<>(graph);
List<Set<Ciudad>> r3 = a.connectedSets();
```

3. *Encontrar los vértices que están en la misma componente conxa que uno dado*

```
Ciudad start = …;
ConnectivityInspector<Ciudad, Carretera> a =
       new ConnectivityInspector<>(graph);
Set<Ciudad> r4 = a.connectedSetOf(start);
```

4. Decidir si hay un camino entre do vértices de un grafo no dirigido

```
Ciudad start = …;
Ciudad end = …;

ConnectivityInspector<Ciudad, Carretera> a =
       new ConnectivityInspector<>(graph);
Boolean r2 = a.pathExists(start,end);
```

Preguntas similares para grafos dirigidos pueden ser respondidas con la clase *KosarajuStrongConnectivityInspector<V,E>*.

Coloreado de grafos

Dado un grafo se define su *número cromático* como el mínimo número de colores necesario para dar un color a cada vértice de tal forma que dos vértices vecinos tengan colores distintos.

Los dos métodos implementan algoritmos de aproximación para resolver el problema. El primero devuelve el número cromático (número de colores distintos). El segundo devuelve para cada posible color el conjunto de vértices que lo tienen asignado. Cada posible color viene representado por un entero *0, 1, …*

Dado un grafo la forma de usar este algortimo es:

```
VertexColoringAlgorithm<Ciudad> vca = new
       GreedyColoring<>(graph);
VertexColoringAlgorithm.Coloring<Ciudad> vc =
       vca.getColoring();
Map<Ciudad,Integer> colorDeCiudad = vc.getColors();
Integer nc = vc.getNumberColors();
```

Aplicaciones del coloreado de grafos

Scheduling

Problemas donde se trata de programar la ejecución de un conjunto de tareas en distintas franjas horarias. Las tareas pueden ser ejecutadas en cualquier orden pero hay pares de tareas que tienen recursos compartidos. Estos

problemas se pueden representar mediante un grafo no dirigido con un vértice por cada tarea y una arista entre aquellos vértices que tengan recursos compartidos. El problema se puede resolver mediante coloreado de grafos. El número cromático nos dará el número mínimo de franjas horarias necesarias y cada grupo de vértice con el mismo color representa tareas que se pueden ejecutar en la misma franja horaria.

Sudoku

El Sudoku es un problema conocido. Un ejemplo se muestra abajo. En un Sudoku las diferentes casillas no en la misma fila, en la misma columna, o en la misma caja deben tener distinto valor asociado. Cada casilla debe tener un valor del 1 al 9. Un ejemplo de Sudoku es:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Ilustración 16: Un sudoku

Para ver la conexión entre Sudoku y coloreado de grafos, primero describiremos un grafo asociado al Sudoku. A cada casilla se le asocia un vértice. El grafo tiene 81 vértices. Dos casillas que están en la misma fila, en la misma columna, o en la misma caja no pueden tener el mismo valor por lo que añadimos una arista entre los vértices asociados. Al comienzo algunos vértices ya asignado un valor. El número asociado a una casilla representa su color. Hay por lo tanto nueve colores.

Los vértices con el mismo color pueden tener el mismo número.

Los vértices con distinto color deben tener distinto valor asociado. Si dentro de los vértices con el mismo color ya hay uno con color

previamente definido los demás toman el mismo color. El grafo se colorea con 9 colores diferentes.

Problema de la Reinas

Con las mismas ideas se podría resolver el problema de las 8 reinas. El problema consistente en colocar 8 reinas en un tablero de ajedrez sin que ninguna de ellas amenace a otra. A partir del problema definimos un grafo cuyos vértices son cada una de las casillas. Añadiremos una arista entre dos vértices si ubicadas dos reinas en las correspondientes casillas se amenazan entre sí. Coloreamos el grafo anterior. Si escogemos una casilla en la fila 0 y todas las de su mismo color obtendremos la solución del problema.

Recorridos sobre grafos

Los grafos son agregados de datos y, como todos los agregados, pueden ofrecer diferentes maneras de recorrer los elementos de este. En concreto diferentes maneras de recorrer los vértices de un grafo. En Java cada forma concreta de recorrer los elementos de un agregado se consigue implementado un objeto de tipo Iterable. Cada recorrido puede considerarse, también, como un problema de búsqueda de un vértice con propiedades específicas entre los vértices de un grafo. Por eso hablaremos indistintamente como recorridos o búsqueda. De entre ellos los más conocidos son: recorrido en anchura, recorrido en profundidad, y en orden topológico.

Recorrido en anchura

La búsqueda en anchura visita cada nodo, posteriormente sus hijos, luego los hijos de estos, etc. Es, por lo tanto, un recorrido por niveles o si se quiere por distancia al nodo origen. Primero los que están a distancia cero, luego los que están a distancia 1, etc. El recorrido puede continuar cuando acabe con una componente conexa con algún vértice no recorrido hasta completar el grafo.

Este recorrido se puede aplicar a grafos dirigidos o no dirigidos.

En el gráfico se muestra un recorrido en anchura. Dado un grafo y un vértice inicial hay varios recorridos que cumplen las restricciones anteriores dependiendo del orden en que se recorran los hijos de un nodo.

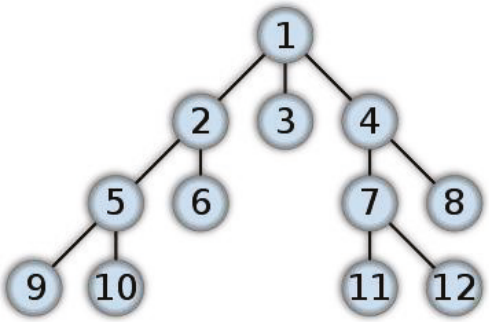


Ilustración 17: Numeración de vértices en anchura

A partir de un grafo la búsqueda en anchura define un árbol de recubrimiento que une cada vértice con aquellos alcanzables desde él, pero con un nivel más. Este árbol de recubrimiento no es único porque depende de la forma de recorrer los vecinos de un vértice. La distancia al vértice inicial en el árbol de recubrimiento es el nivel de cada vértice.

El recorrido en anchura tiene la propiedad que el camino de la raíz a un vértice en el árbol de recubrimiento es el más corto posible.

El recorrido en anchura usa un conjunto y una cola. En el conjunto, inicialmente vacío, se van guardando los vértices ya visitados. Dado un vértice actual se añaden a la cola todos sus vecinos no visitados y el mismo al conjunto de visitados. Luego se saca un vértice de la cola y se continua por él. Se acaba cuando la cola está vacía.

Usos del recorrido en Anchura

El recorrido en anchura, el orden definido a partir del mismo tiene diferentes aplicaciones:

- Cálculo de las componentes conexas de un grafo no dirigido. Una componente conexa está formada por todos los vértices alcanzables desde uno dado.

- Obtención de un bosque de recubrimiento en anchura de un grafo no dirigido.
- Cálculo de los caminos más cortos medidos en número de aristas de un vértice a los demás en un grafo no dirigido
- En un grafo dirigido encontrar todos los vértices alcanzables desde uno dado.

Código del recorrido en anchura

Dado un grafo el recorrido en anchura se puede hacer con la clase *BreadthFirstIterator* que implementa un iterador a partir del cual podemos obtener un stream.

```
Ciudad c = …;
Ciudad c2 = …;

BreadthFirstIterator<Ciudad, Carretera> ra =
       new BreadthFirstIterator<>(graph,c);
Stream<Ciudad> sa = Stream2.of(ra);
List<Ciudad> la = sa.toList();
Integer pa = ra.getDepth(c2);
Ciudad pva = ra.getParent(c2);
Carretera sta = ra.getSpanningTreeEdge(c2);
```

El algoritmo de recorrido en anchura construye además un árbol de recubrimiento en anchura. Este es un árbol de recubrimiento. Contiene todos los vértices del grafo. Cada vértice tiene como hijos sus vecinos en el recorrido en anchura. Los caminos desde cada vértice a la raiz en este árbol son los caminos más cortos en el grafo, medidos en número de aristas, entre ambos vértices. La variable *ra* tiene algunos métodos adicionales que pueden ser de interés:

- *Integer pa = ra.getDepth(c2):* Proporciona la profundidad del vértice en el árbol de recubrimiento en profundidad.
- *Ciudad pva = ra.getParent(c2):* Proporciona el padre del vértice en el árbol de recubrimiento en profundidad.
- *Carretera sta = ra.getSpanningTreeEdge(c2):* Proporciona la arista hacia el padre del vértice en el árbol de recubrimiento en profundidad.

Implementación del recorrido en anchura

El recorrido en anchura de un grafo se implimenta usando una cola de forma similar al que hicimos para árboles. Seguimos el esquema de un iterador: establecemos un estado que incluye una cola y un *Map<V,E>* que mantiene en sus claves lo vértices ya visitados y las aristas del árbol de recubrimiento en anchura.

La clase debe implementar los métodos del iterador: hasNext y next. El recorrido acaba cuando la cola está vacía.

```
public boolean hasNext() {
	return !stack.isEmpty();
}
```

El método next devuelve un elemento sacado de la cola y añade a ella hijos del que ha sacado que no han sido vstado previamente. Por cada uno de ellos añade una arista al árbol de recubrimineto que se va costruyendo en el Map.

```
@Override
public V next() {
	V actual = stack.pop();
	for(V v:Graphs.neighborListOf(graph, actual)) {
		if(!this.edgeToOrigin.containsKey(v)) {
				stack.add(v);
				this.edgeToOrigin.put(v,
					graph.getEdge(actual, v));
		}
	}
	return actual;
}
```

El estado es de la forma:

```
public class DephtSearch<V, E>   implements
       Iterator<V>, Iterable<V> {

       protected Map<V,E> edgeToOrigin;
       public Graph<V,E> graph;
       protected Stack<V> stack;
       protected V startVertex;
       DephtSearch(Graph<V, E> g, V startVertex) {
              this.graph = g;
              this.startVertex = startVertex;
              this.edgeToOrigin = new HashMap<>();
              this.edgeToOrigin.put(startVertex, null);
              this.stack = new Stack<>();
              this.stack.add(startVertex);
       }
       public Iterator<V> iterator() {
              return this;
       }
```

Recorrido en profundidad

La búsqueda en profundidad visita cada nodo, posteriormente luego uno de sus hijos, luego uno de los hijos de éste, etc. Cuando termina con todos los descendientes de un hijo continua con los del siguiente. El recorrido puede continuar cuando acabe con una componente conexa con algún vértice no recorrido hasta completar el grafo.

Este recorrido se puede aplicar a grafos dirigidos o no dirigidos.

En el gráfico se muestra un recorrido en profundidad. Dado un grafo y un vértice inicial hay varios recorridos que cumplen las restricciones anteriores dependiendo del orden en que se recorran los hijos de un nodo.

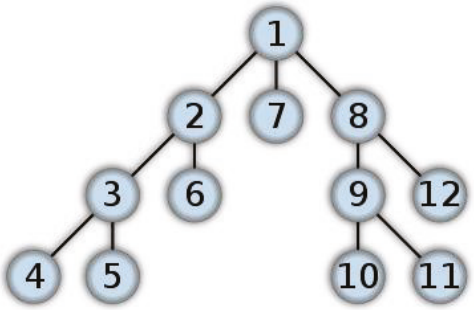


Ilustración 18: Numeración de vértices en profundidad

A partir de un grafo la búsqueda en profundidad define un árbol de recubrimiento que une cada vértice con aquellos alcanzables desde él. Este árbol de recubrimiento no es único porque depende de la forma de recorrer los vecinos de un vértice.

El recorrido en profundidad usa un conjunto y una pila. En el conjunto, inicialmente vacío, se van guardando los vértices ya visitados. Dado un vértice actual se añaden a la pila todos sus vecinos no visitados y el mismo al conjunto de visitados. Luego se saca un vértice de la pila y se continua por él. Se acaba cuando la pila está vacía.

Usos de del recorrido en profundidad

La búsqueda en profundidad y los órdenes definidos a partir del mismo tienen diferentes aplicaciones:

- Obtención de un bosque de recubrimiento de un grafo dirigido o no.
- Cálculo de las componentes conexas de un grafo no dirigido.
- Cálculo de las componentes fuertemente conexas de un grafo dirigido.
- Cálculo de caminos de un vértice a los demás en un grafo dirigido o no dirigido o decidir que no existe camino.
- Comprobar si un grafo no dirigido es conexo. No es conexo si existe más de una componente conexa.
- Comprobar si un grafo dirigido es conexo. No es conexo si existe más de una componente fuertemente conexa. Si un grafo dirigido es conexo existen caminos entre cada par de vértices.
- Cálculo de las componentes débilmente conexas de un grafo dirigido G. Es calcular las componentes conexas del grafo G' obtenido a partir de G haciendo todas las aristas no dirigidas.

Código del recorrido en profundidad

```
Ciudad c = …;

DepthFirstIterator<Ciudad, Carretera> rp = new
       DepthFirstIterator<>(graph,c);
Stream<Ciudad> sp = Stream2.of(rp);
List<Ciudad> lp = sp.toList();
```

El algoritmo de recorrido en profundidad ofrece en cada momento la pila de elementos apilados.

Implementación del recorrido en profundidad

El recorrido en profundidad de un grafo se implimenta usando una pila de forma similar al que hicimos para árboles. Seguimos el esquema de un iterador: establecemos un estado que incluye una pila y un *Map<V,E>* que mantiene en sus claves lo vértices ya visitados y las aristas del árbol de recubrimiento en profundidad.

La clase debe implementar los métodos del iterador: hasNext y next. El recorrido acaba cuando la cola está vacía.

El estado de la clase es de la forma:

```
public class DephtSearch<V, E> implements Iterator<V>,
       Iterable<V> {
       protected Map<V,E> edgeToOrigin;
       public Graph<V,E> graph;
       protected Stack<V> stack;
       protected V startVertex;
       DephtSearch(Graph<V, E> g, V startVertex) {
              this.graph = g;
              this.startVertex = startVertex;
              this.edgeToOrigin = new HashMap<>();
              this.edgeToOrigin.put(startVertex, null);
              this.stack = new Stack<>();
              this.stack.add(startVertex);
       }
       public Iterator<V> iterator() {
              return this;
       }
```

Y los métodos del iterador

```
public boolean hasNext() {
       return !stack.isEmpty();
}
```

```
@Override
public V next() {
	V actual = stack.pop();
	for(V v:Graphs.neighborListOf(graph, actual)) {
		if(!this.edgeToOrigin.containsKey(v)) {
			stack.add(v);
			this.edgeToOrigin.put(v,
					graph.getEdge(actual, v));
		}
	}
	return actual;
}
```

Orden topológico

Es un tipo de recorrido que se aplica a grafos dirigidos acíclicos. En estos grafos podemos definir los *vértices posteriores y anteriores* a uno dado. Dado un vértice sus vértices posteriores son los alcanzables con un camino dirigido desde él. Los vértices anteriores son aquellos desde los que existe un camino dirigido hasta él.

En este recorrido cada vértice va después que los vértices anteriores y después que sus vértices posteriores.

Ordenación topológica de un grafo dirigido: la ordenación topológica de los vértices de un grafo no dirigido viene dada por el *Postorden Inverso*

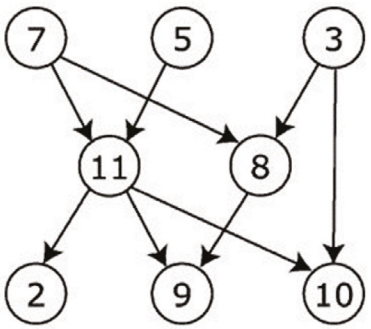


Ilustración 19: Orden topológico

Con las restricciones anteriores hay varios recorridos posibles. Algunos son:

```
7, 5, 3, 11, 8, 2, 9, 10
3, 5, 7, 8, 11, 2, 9, 10
3, 7, 8, 5, 11, 10, 2, 9
5, 7, 3, 8, 11, 10, 9, 2
7, 5, 11, 3, 10, 8, 9, 2
7, 5, 11, 2, 3, 8, 9, 10
```

Una aplicación del orden topológico es en la programación de una secuencia de tareas que tienen precedencias entre ellas. Las tareas están representadas por vértices, y hay un arco (o arista) desde *x* a *y* si la tarea *x* debe completarse antes que la tarea *y* comience.

El orden topológico se aplica como hemos dicho a grafos dirigidos acíclicos. Sobre estos grafos podemos usar el recorrido en anchura para encontrar todos los vértices alcanzables desde uno dado. Aplicando la misma idea sobre el grafo inverso podemos obtener los vértices anteriores a uno dado.

5. *Comprobar que un grafo dirigido es acíclico*

La librería jgrapht aporta la clase CycleDetector que puee decidir si hay ciclos o no.

```
CycleDetector<Integer, SimpleEdge<Integer>> cd =
       new CycleDetector<>(graph);
Boolean c = cd.detectCycles();
```

6. Dado un grafo dirigido acíclico obtener su grafo inverso

```
<V, E> SimpleDirectedGraph<V, E> inversedDirectedGraph(
       SimpleDirectedGraph<V, E> graph){
       SimpleDirectedGraph<V, E> gs =
              Graphs2.simpleDirectedGraph();
       for (V v : graph.vertexSet()) {
              gs.addVertex(v);
       }
       for (E e : graph.edgeSet()) {
              V s = graph.getEdgeSource(e);
              V t = graph.getEdgeTarget(e);
              gs.addEdge(t, s, e);
       }
       return gs;
}
```

7. Dado un vértice en un grafo dirigido acíclico encontrar sus vértices posteriores

```
<V,E>List<V> verticesPost(SimpleDirectedGraph<V,E> graph,V v){
       return Stream2.of(
              new BreadthFirstIterator<>(graph,v)).toList();
}
```

8. Dado un vértice en un grafo dirigido acíclico encontrar sus vértices anteriores

```
<V,E> List<V> verticesPre(SimpleDirectedGraph<V,E> graph,V v){
       SimpleDirectedGraph<V,E> graph2 =
              Graphs2.inversedDirectedGraph(graph);
       return Stream2.of(
              new BreadthFirstIterator<>(graph2,v)).toList();
}
```

Camino Mínimo

El problema de camino mínimo se puede enunciar del siguiente modo: Dado un grafo y dos vértices encontrar el camino (secuencia de aristas) con longitud mínima entre ambos vértices.

Algoritmo de Dijkstra

El algoritmo de Dijkstra encuentra el camino más corto entre dos vértices de un grafo. Los pesos deben ser no negativos.

Este es el algoritmo más utilizado para encontrar el camino más corto entre dos puntos de un grafo, por ejemplo, en grafos que representen mapas, nodos de una red, etc.

Esencialmente el algoritmo de Dijkstra funciona pasando de un vértice al siguiente vecino más cercano al origen. Antes de pasar a un nuevo vértice actualiza el camino más corto al origen. El algoritmo para cuando encuentra el vértice final. Si los pesos de las aristas son todos 1 entonces el algoritmo de Dijkstra se reduce a la búsqueda en anchura.

Usualmente queremos buscar el camino más corto, pero si queremos buscar el más largo podemos conseguirlo cambiando el signo de los pesos de las diferentes aristas.

Más adelante veremos con más detalle otros algoritmos de camino mínimo como el *A**.

Código de uso del algoritmo de Dijkstra

```
ShortestPathAlgorithm<Ciudad,Carretera> a =
       new DijkstraShortestPath<Ciudad,Carretera>(graph);
Ciudad from = …;
Ciudad to = …;
GraphPath<Ciudad,Carretera> gp =  a.getPath(from,to);
```

El algoritmo de Dijkstra da como resultado un GraphPath que es un camino en un grafo. Este tipo tiene las propiedades: lista de vértices, lista de aristas, peso, etc.

Problema del viajante

Es un problema muy conocido en teoría de grafos. Hay dos problemas relacionados que buscan caminos cerrados en grafos que incluyan todos lo vértices del grafo. El primero es el *ciclo Hamiltoniano* que es un camino cerrado que visita todos los vértices del grafo una sóla ves aunque puede repetir el paso por alguna arista. Un *ciclo Euleriano* es un camino cerrado que recorre cada arista exactamente una vez. Los grafos que tienen un ciclo Hamiltoniano se llaman grafos hamiltonianos y los que tienen un ciclo Euleriano grafos eulerianos.

Se denomina *Problema del Viajante* a encontrar en ciclo Hamiltoniano de peso mínimo. Es decir, un camino cerrado en un grafo que pase por todos los vértices del grafo una sola vez y tenga la longitud mínima.

El API se ofrecen algunos algoritmos aproximados para grafos sea completo y se verifique la desigualdad triangular. Es decir, para tres vértices x, y, z entonces $d(x,y)+d(y,z) > d(x,z)$.

La forma de obtener un grafo completo que sea una vista de otro que no lo es se deja como ejercicio.

Una de las clases que implementa este algoritmo en jgrapht es la siguiente:

```
HamiltonianCycleAlgorithm<Ciudad, Carretera> a =
       new TwoApproxMetricTSP<>();
GraphPath<Ciudad, Carretera> r = a.getTour(graph2);
```

Para encontrar un camino Euleriano podemos usar la siguiente clase de jgrapht.

```
EulerianCycleAlgorithm<Ciudad, Carretera> a2 =
       new HierholzerEulerianCycle<>();
GraphPath<Ciudad, Carretera> r2 = a2.getEulerianCycle(graph);
```

Problemas de complejidad

Recordemos las expresiones más relevantes sobre sumatorios y recurrencias

Sumatorios

$$\sum_{i=0}^{n} a^i = \begin{cases} 1, & a < 1 \\ a^n, & a \geq 1 \end{cases}$$

$$\sum_{i=0}^{log_r n} a^i \cong a^{log_r n} \cong n^{log_r a}$$

$$\sum_{x \in pa(a,r)}^{n} x^d (ln\, x)^p \cong \frac{1}{r(d+1)} n^{d+1} (ln\, n)^p$$

$$r > 1, \sum_{x \in pg(a,r)}^{n} x^d (ln\, x)^p \cong \begin{cases} (ln\, n)^{p+1}, & d = 0 \\ n^d (ln\, n)^p, & d > 0 \end{cases}$$

Recurrencias del tipo:

$$T(n) = aT(n-b) + \Theta(n^d)log^p n$$

Cuya solución es:

$$T(n) = \begin{cases} \Theta\left(a^{n/b} log^p n\right), & si \quad a > 1 \\ \Theta\left(n^{d+1} log^p n\right), & si \quad a = 1 \\ \Theta\left(n^d log^p n\right), & si \quad a < 1 \end{cases}$$

Recurrencias de tipo:

$$T(n) = aT(n/b) + \Theta(n^d \log^p n)$$

Solución

$$T(n) \in \begin{cases} \Theta(n^d \log^p n) & \text{si } a < b^d \\ \Theta(n^d \log^{p+1} n) & \text{si } a = b^d \\ \Theta(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

Ejemplos de complejidad de algoritmos iterativos

1. *Calcular la complejidad*

```
for (int i = 1; i <= n; i++) {
    r += a * i;
}
```

$$\sum_{i=1}^{n} 1 = n \in \Theta(n)$$

2. Calcular la complejidad

```
i = n;
while (i >= 1) {
    s = s + i;
    i = i / r;
}
```

La complejidad es

$$\sum_{i \in pg(1,r)}^{n} 1 \cong \Theta(log\, n =$$

También podemos plantearlo mediante una ecuación recursiva. El tamaño asociado al bucle directamente la variable *i*

$$T(n) = T\left(\frac{n}{r}\right) + 1$$

Cuya solución es $T(n) \in \Theta(\log n) si r \geq 2$ puesto que $a = 1, b = r, d = 0$. El problema original tiene la complejidad asociada al primer valor del tamaño $T(n) \in \Theta(\log n)$

3. Calcular la complejidad

```
i = 1;
while (i <= n ) {
    s = s + i;
    i = i + r;
}
```

La complejidad es:

$$\sum_{i \in pa(1,r)}^{n} 1 \cong \frac{1}{r} n \in \Theta(n)$$

También podemos plantearlo mediante una ecuación recursiva considerando que el tamaño es *m=n-i.*

$$T(m) = T(m-r) + 1$$

Cuya solución es $T(m) \in \Theta(m)$ si $r \geq 1$ puesto que $a = 1, b = r, d = 0$. La complejidad del problema original es $T(n) \in \Theta(n-1) = \Theta(n)$

4. Calcular la complejidad

```
for (int i = 1; i <= n - 1; i++) {
        for (int j = i + 1; j <= n; j++) {
            for (int k = 1; k <= j; k++) {
                r = r + 2;
            }
        }
   }
```

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j} 1 \cong \sum_{i=1}^{n}\sum_{j=i}^{n} j \cong \sum_{i=1}^{n}(n^2 - i^2) \cong \left(n^3 - \frac{n^3}{3}\right) = \frac{2}{3}n^3 \in \Theta(n^3)$$

5. Calcular la complejidad

```
for (i = 1; i < n; i++) {
       for (j = 1; j < i * i; j++) {
                z ++;
       }
}
```

$$\sum_{i=1}^{n}\sum_{j=1}^{i^2} 1 \cong \sum_{i=1}^{n} i^2 \cong n^3 \in \Theta(n^3)$$

6. *Suponiendo que los valores de a[i] pertenecen a 0..k-1 (k<n) y todos son igualmente probables calcular la complejidad del caso peor, mejor y medio para valores grandes de n y k.*

```
r = 0;
for (int i = 0; i < n; i++) {
    if (a[i] == k) {
       for (j = 1; j < n; j++) {
           r += a[i + j];
        }
     }
}
```

En este algoritmo el caso peor es que siempre entre en el *if.* Es decir que *a[i]==k* para todos los valores de *i*. En este caso la complejidad es:

$$T(n) = \sum_{i \in pa(1,1)}^{n} \sum_{j \in pa(1,1)}^{n} 1 \cong \Theta(n^2)$$

El caso mejor es que nunca entre en el *if.* Es decir que a*[i]≠k* para todos los valores de *i*.

$$T(n) = \sum_{i \in pa(1,1)}^{n} 1 \cong \Theta(n)$$

Las probabilidades de que *a[i]* sea igual o distinto a *k* son: $P(a[i] == k) = \frac{1}{k}$, $P(a[i] \neq k) = \frac{k-1}{k}$. Luego estas son las probabilidades de que el algoritmo entre o no en la sentencia *if.* El tiempo medio *T(n)* es

$$T(n) = \sum_{i=0}^{n} \left(\frac{1}{k} \sum_{j=1}^{n} 1 + \frac{k-1}{k} 1 \right) \cong \sum_{i=0}^{n} 1 \cong \Theta(1)$$

Los anterior se deduce que asumiendo que $ak \approx n \to \infty$:

$$\frac{1}{k} \sum_{j=1}^{n} 1 \cong_{\infty} 1, \qquad \frac{k-1}{k} \cong_{\infty} 1$$

7. Suponiendo que A(i) ∈ Θ(i) calcular la complejidad

```
i = 1;
x = 0;
while ((i * i) <= n) {
      x = x + A(i);
      i = i + 1;
}
```

El índice varía de 1 a $\sqrt{n}$ luego

$$T(n) \cong \sum_{i=1}^{\sqrt{n}} i \cong (\sqrt{n})^2 \cong n \in \Theta(n)$$

8. Suponiendo B(n, m) ∈ Θ(n) para todo valor de j calcular la complejiad

```
i = n;
while (i > 1) {
       x = x + x;
       i = i / 4;
       for (int j=1; j <= n; j++) {
                  x = x * B(j, n);
       }
}
```

La complejidad es

$$\sum_{i \in pg(1,4)}^{n} \sum_{j=1}^{n} j \cong n^2 \sum_{i=\in pg(1,4)}^{n} 1 \in \Theta(n^2 \log n)$$

9. Calcular la complejidad

```
int busca(int [] a, int n, int c) {
    int i = 0;
    int j = n;
    int k, r;
    boolean fin = false;
    while (j > i && !fin) {
        k = (i + j) / 2;
        if (a[k] == c) {
            r = k;
            fin = true;
        } else if (c < a[k]) {
            j = k - 1;
        } else {
            i = k + 1;
        }
    }
    if (!fin) {
        r = - 1;
    }
    return r;
}
```

El caso peor ocurre cuando entra siempre en el bucle. Esto ocurre cuando *a[k] !=* c para todos los valores de i. En este caso sea la variable $x = j - i$ que representa el tamaño de un subarray. Observando las iteraciones tenemos la secuencia $[n, \frac{n}{2}, \frac{n}{2^2}, \dots, 1]$,

$$T(n) = \sum_{x \in pg(1,4)}^{n} 1 \cong \ln n \in \Theta(\log n)$$

El caso mejor ocurre cuando entra en el bucle solo una vez. Esto ocurre cuando *a[n/2] ==* c. En ese caso:

$$T(n) \in \Theta(1)$$

En el caso medio hay tres posibilidades que en una iteración ocurra que *a[n/2] ==* c y si no ocurre que vaya hacia la izquierda o hacia la derecha. Para abordar el problema tenemos que hacer hipótesis sobre las tres probabilidades. Dependiendo de esto obtendremos una solución u otra. Si asumimos que los enteros en el array están en el rango *0..r-1*, c es un

entero en ese rango que tiene la misma probabilidad de ser igual a cualquiera de ellos, las probabilidades serían $\frac{1}{r}, \frac{r-1}{2r}, \frac{r-1}{2r}$. Aunque es un problema iterativo podemos plantear una ecuación de recurrencia para el caso medio de la forma:

$$T(n) = \frac{1}{r} + \frac{r-1}{2r} T(\frac{n-1}{2}) + \frac{r-1}{2r} T(\frac{n-1}{2}) = \frac{1}{r} + \frac{r-1}{r} T(\frac{n}{2}) \in \Theta(1)$$

Puesto que $\frac{r-1}{r} < 1$

10. Calcular la complejidad

```
i = n;
while (i >= 1) {
    j = 1;
    while (j <= i ) {
       s = s + j;
       j = j + 2;
    }
    i = i / 3;
}
```

La complejidad es:

$$\sum_{i \in pg(1,3)}^{n} \sum_{j \in pa(1,2)}^{j=i} 1 \cong \sum_{x \in pg(1,3)}^{n} i \cong n \in \Theta(n)$$

Ejemplos de complejidad de algoritmos recursivos

11. Calcular la complejidad del algoritmo recursivo siguiente

```
int f (int [] a, int n, int i, int j) {
    int k, t, s;
    if(j-i = 0){
        s= 0;
    } else if (j - i = 1) {
        s = a[i];
    } else if (j - i = 2) {
        s = a[i]+a[i+1];
    } else {
        k = (i + j)/2;
        t = (j - i)/3;
        s = a[k] + f (a, n, i, i + t) + f (a, n, j - t, j);
    }
    return s;
}
int f0(int [] a, int n){
    return f(a, n, 0, n);
}
```

Suponemos que x es el tamaño del subintervalo del array correspondiente $x = j - i$. Vemos que:

$$t = \frac{j-i}{3} = \frac{x}{3}$$

$$x_1 = i + t - i = t = \frac{x}{3}$$

$$x_2 = j - (j - t) = t = \frac{x}{3}$$

Por lo tanto

$$T(x) = 2T\left(\frac{x}{3}\right) + 1$$

Cuya solución es:

a= 2, b=3, d= 0, $\quad T(x) \in \Theta(x^{\log_3 2}) \approx \Theta(x^{0.63})$

La complejidad del problema original es por lo tanto $T(n) \in \Theta(n^{\log_3 2}) \approx \Theta(n^{0.63})$

Para hacer el cálculo de la complejidad con memoria debemos tener en cuenta que el tamaño de los problemas va disminuyendo en progresión geométrica y por lo tanto el número de pasos hasta el caso base es $\log_3 n$. En cada paso se duplican el número de problemas por lo que el número de problemas totales es:

$$\sum_{i=0}^{log_3 n} 2^i \cong n^{log_3 3}$$

Vemos cómo la complejidad con y sin memoria es la misma. A esta conclusión podríamos haber llegado observando que no hay subproblemas duplicados.

---

12. Calcular la complejidad del algoritmo recursivo siguiente

---

```
double f (int n, double a) {
    double r;
    if (n == 1) {
        r = a;
    } else {
        r = f (n/2, a+1) - f (n/2, a-1);
        for (int i = 1; i <= n; i++) {
                    r += a * i;
         }
    }
    return r;
}
```

La parte iterativa, según el Ejemplo 1, es $\Theta(n)$. Por lo tanto:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Cuya solución es:

a= 2, b= 2, d =1, $T(n) \in \Theta(n \log n)$

13. Calcular la complejidad del algoritmo recursivo siguiente

```
int f (int a, b) {
    int r;
    if (a == 0 || b == 0) {
        r = 1;
    } else {
        r = 2 * f (a-1, b-1);
    }
    return r;
}
```

Sea $n = \min(a, b)$. Tengamos en cuenta que $\min(a-1, b-1) = n-1$. Por lo tanto:

$$T(n) = T(n-1) + 1$$

Por lo tanto con *a= 1, b= 1, d=0*

$$T(n) \in \Theta(n)$$

14. Calcular la complejidad del algoritmo recursivo siguiente

```
int f (int n) {
    int i, j, z, r;
    if (n < 1) {
        r = 1;
    } else {
        z = 0;
        for (i = 1; i < n; i++) {
           for (j = 1; j < i * i; j++) {
                z ++;
           }
        }
        r = z * (( f (n - 2)) ^ 2);
    }
    return r;
fin
```

Tomando el resultado del ejemplo 5 tenemos:

$$T(n) = T(n-2) + \Theta(n^3)$$

Con a *= 1, b= 2, d = 3*, tenemos:

$$T(n) \in \Theta(n^4)$$

En el caso anterior si la expresión f (n – 2)) ^ 2 la escribiéramos de la forma f (n – 2))* f (n – 2)) la complejidad sería

$$T(n) = 2T(n-2) + \Theta(n^3)$$

Cuya solución es

$$T(n) \in \Theta\left(2^{\frac{n}{2}}\right) = \Theta\left(\sqrt{2}^{n}\right)$$

En ambos casos si utilizamos memoria la complejidad es

$$\sum_{i=2u}^{n} i^3 \cong \Theta(n^4)$$

En este caso asumimos que los subproblemas están calculados. Entonces cada problema tiene un coste de $\Theta(i^3)$ y los subproblemas que es necesario calcular están en una secuencia aritmética de paso 2.

15. Calcular la complejidad del algoritmo recursivo siguiente

```
int f (int [] a, int n) {
    return g (a, n, 0);
}
int g (int [] a, int n, int i) {
    int s;
    if (n-i==0) {
        s = r;
    } else {
        r += a[i];
        s = g (a,i+1);
    }
    return s;
}
```

Escogiendo el tamaño para g es $x = n - i$, tenemos $x_1 = n - (i + 1) = x - 1$.

$$T_g(x) = T_g(x - 1) + 1$$

Cuya solución es:

a= 1, b= 1, d= 0, $\quad T_g(x) \in \Theta(x)$

Para la función *f* escogemos como tamaño n por lo que:

$$T_f(n) = T_g(n - 0) = T_g(n) \in \Theta(n)$$

El cálculo con memoria se deja como ejercicio.

16. Calcular la complejidad del algoritmo recursivo siguiente

```
void p0(int[] a, int n){
    p(a,n,n);
}
void p(int [] a, int n, int i) {
    if (i > 0) {
        ps(a, n, i);
        p(a, n, i - 1);
    }
}
void ps(int [] a, int n, int i) {
    int temp;
    if (n - i > 1) {
        if(a[i] > a[i + 1]) {
            temp = a[i];
            a[i] = a[i + 1];
            a[i + 1] = temp;
        }
        ps(a, n, i + 1);
    }
}
```

Sea el tamaño de *ps es* $n_{ps} = n - i$. Por lo que en el caso peor.

$$T_{ps}(n_{ps}) = T_{ps}(n_{ps} - 1) + 1$$

Cuya solución es $T_{ps}(n_{ps}) \in \Theta(n_{ps})$.

Por lo tanto asumiendo que el tamaño de problema es $n_p = i$ tenemos:

$$T_p(i) = T_p(i-1) + \Theta(n-i)$$

Esta ecuación no tiene la forma de las que hemos dado en el resumen del principio pero por la equivalencia de la recursividad final y la iteratividad, y teniendo en cuenta que la variable i se mueve de 0 a n tenemos que

$$T_{p0}(n) = \sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} n - \sum_{i=0}^{n} i = n\sum_{i=0}^{n} 1 - \sum_{i=0}^{n} i \cong n^2 - \frac{1}{2}n^2 \cong n^2$$

El cálculo con memoria se deja como ejercicio.

17. Calcular la complejidad del algoritmo recursivo siguiente

```
int F(int n) {
    int x, j, i;
    if (n < 10) {
        i = n;
    } else {
        i = 1;
        j = 0;
        while ((i * i) <= n) {
            j = j + A(i);
            i = i + 1;
        }
        x = n;
        while (x > 1) {
            j = j + x;
            x = x / 4;
            for (int ii=1; ii<=n;ii++) {
                j = j * B(ii, n);
            }
        }
        i = 2 * F(n / 2) + j;
    }
    return i;
}
```

Asumiendo los resultados de los problemas 7 y 8 anteriores tenemos:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$$

Donde hemos tenido en cuenta que $\Theta(n) + \Theta(n^2 \log n) \in \Theta(n^2 \log n)$

Por lo tanto, la solución es:

a= 1, b= 2, d= 2, p = 1, $T(n) \in \Theta(n^2 \log n)$

El cálculo con memoria se deja como ejercicio.

Problemas de algoritmos recursivos simples a iterativos

Introducción

Los problemas recursivos lineales y los problemas iterativos tienen muchas relaciones. Los problemas iterativos se pueden ver como una secuencia y un acumulador acumulado por la izquierda. Los recursivos finales equivalentes siguen siendo acumulaciones por la izquierda de un acumulador sobre una secuencia. Los recursivos lineales no finales se pueden ver como una secuencia y un acumulador acumulado por la derecha. Según las propiedades de la función de acumulación podemos obtener algoritmos iterativos equivalentes a los recursivos lineales no finales.

Para el diseño de este tipo de algoritmos podemos seguir dos vías:

- Hacer un diseño iterativo y posteriormente conseguir una versión funcional y otra recursiva final
- Hacer un diseño recursivo y si resulta un algoritmo recursivo lineal transformarlo, si se puede, para conseguir una versión recursivo final y posteriormente una iterativa y su correspondiente funcional.

Recordemos algunas ideas vistas anteriormente:

Si la función de combinación del acumulador es de la forma $(b, e) = b\ op\ f(e)$, siendo op un operador asociativo y conmutativo entonces es igual la acumulación por la izquierda y por la derecha sobre cualquier secuencia.

Si s^{-1} ***es la secuencia inversa de s entonces se cumple***:

$$acumulaIzquierda(s, a) = acumulaDerecha(s^{-1}, a)$$

La acumulación por la izquierda se hace mediante un algoritmo iterativo. La acumulación por la derecha se hace mediante un algoritmo recursivo no final.

Cada algoritmo iterativo puede ser visto como la acumulación por la izquierda de una secuencia y un acumulador.

Igualmente, cada algoritmo recursivo final puede ser visto como la acumulación por la izquierda de una secuencia y un acumulador.

Cada algoritmo recursivo no final puede ser visto como la acumulación por la derecha de una secuencia y un acumulador.

Si tenemos un algoritmo recursivo no final y la función de combinación del acumulador es de la forma $(b, e) = b\ op\ e$, siendo b, e del mismo tipo y op conmutativo y asociativo entonces hay un algoritmo iterativo equivalente y otro recursivo final con la misma secuencia y el mismo acumulador.

Si tenemos un algoritmo recursivo no final y la función de combinación del acumulador es de la forma $(b, e) = bopf(e)$, siendo op un operador asociativo y no conmutativo entonces hay un algoritmo iterativo con la misma secuencia y un acumulador con función de acumulación $(b, e) = f(e)\ op\ b$

Algunos problemas sencillos

Busquemos, en primer lugar, un diseño iterativo para los siguientes problemas:

1. Obtener la suma de los primos mayores o iguales que *m* y menores que *n*.
2. Obtener una lista de pares de un primo y su siguiente, tal que el primero sea mayor o igual que *m* y menor que *n,* y que la diferencia entre ellos sea una cantidad *k >= 2* dada.
3. Obtener el número de veces que repite cada divisor, mayor que 1, entre los divisores de los números que va de *m* a *n*.
4. Encontrar las frecuencias de un elemento en una secuencia

---

1. Obtener la suma de los primos mayores o iguales que m y menores que n.

Se aborda partiendo de la secuencia formada por el primer primo posterior a *m*, continuada por el siguiente primo hasta *n*, y acumulada con el acumulador *sum*.

---

2. Obtener una lista de pares de un primo y su siguiente primo, tal que el primero sea mayor o igual que m y menor que n, y que la diferencia entre ellos sea una cantidad k >= 2 dada.

Podemos resolverlo partiendo de una secuencia de números primos en el rango especificado, obteniendo a partir de ella la secuencia de pares consecutivos, filtrándola con la propiedad pedida y acumulándola en una lista.

```
List<LongPair> primosPar(Long m, Long n, Integer k) {
	Stream<Long> r = Stream.iterate(
		Math2.siguientePrimo(m - 1), x -> x < n,
			x -> Math2.siguientePrimo(x));
	List<LongPair> r2 = Stream2.consecutivePairs(r)
		.map(p->LongPair.of(p))
		.filter(t -> t.second() - t.first() == k)
		.toList();
	return r2;
}
```

Ya vimos cómo obtener la secuencia de pares consecutivos. La versión imperativa requerirá desplegar esta secuencia de pares consecutivos.

```
List<LongPair> primosPar2(Long m, Long n, Integer k) {
	List<LongPair> r = new ArrayList<>();
	Long e1 = null;
	Long e2 = Math2.siguientePrimo(m - 1);
	while(e2<n) {
		LongPair p = LongPair.of(e1, e2);
		e1 = e2;
		e2 = Math2.siguientePrimo(e2);
		if(p.first() != null &&
				p.second() - p.first() == k) {
			r.add(p);
		}
	}
	return r;
}
```

---

3. Obtener el número de veces que repite cada divisor, mayor que 1, entre los divisores de los números que va de m a n.

Podemos obtener la solución funcional recorriendo los enteros de *m* a *n,* desplegando cada entero en sus divisores, *flatmap*, y acumulandolos en un *Multiset* para guardar las frecuencias de cada uno.

```
Stream<Long> divisores(Long n){
	return Stream.iterate(2L,x-> x <= (long) Math.sqrt(n),
		x -> x+1)
	.filter(x->n%x==0);
}
Multiset<Long> rr4 = Stream.iterate(m,x->x<n,x->x+1)
	.flatMap(x->divisores(x))
	.collect(Collectors2.toMultiset());
```

Dejamos como ejercicio las versiones imperativas.

4. *Encontrar las frecuencias con las que aparece una propiedad en los elementos de una secuencia*

En esencia consiste en partir de una secuencia, formar grupos a partir de una función que define la propiedad, la clave, que calcula el grupo, y aplicar una operación de acumulación a cada grupo que en este caso es contar cuantos hay. Es la acumulación que denominamos GroupSize.

Las frecuencias con la que aparecen los restos al dividir por *r* en una secuencia de enteros en un rango *r1..r2*, es de la forma:

```
Map<Integer,Integer> frecuencias(
             Integer r1,Integer r2,Integer r) {
     return IntStream.range(r1,r2).boxed().
       .collect(Collectors.groupingBy(e-> e%r,
         Collectors.collectingAndThen(
             Collectors.counting(),Long::intValue)));
}
```

Su versión imperativa

```
Map<Integer,Integer> frecuencias(Integer r1, Integer r2,
Integer r) {
       Integer i = r1;
       Map<Integer,Integer> b = new HashMap<>()
       while(i < r2){
              Integer key = i%r;
              i = i+1;
              if(b.containsKey(key) m.put(key,m.get(key)+1)
              else m.put(key,1)
       }
       return b;
}
```

Y su versión recursiva final:

```
Map<Integer,Integer> frecuencias(Integer r1, Integer r2,
Integer r) {
       Integer i = r1;
       Map<Integer,Integer> b = new HashMap<>()
       return frecuencias(r1, r2, i, b);
}
```

```
Map<Integer,Integer> frecuencias(Integer r1,Integer r2,
          Integer i, Map<Integer,Integer> b) {
       if(i<r2){
              Integer key = i%r;
              if(b.containsKey(key) m.put(key,m.get(key)+1)
              else m.put(key,1)
              b = frecuencias(r1,r2,i+1,b);
       }
       return b;
}
```

Las operaciones de *grouping* son, como ya vimos, operaciones de acumulación donde la base del acumulador tiene tipo *Map<K,V>* y la función de acumulación es para este caso concreto:

```
Integer key = i%7;
if(b.containsKey(key) m.put(key,m.get(key)+1)
else m.put(key,1)
```

Alternativamente las operaciones de *grouping* podemos verlas como una partición de la secuencia en grupos mediante la clave más un acumulador para acumular cada grupo en un valor.

Problemas de lectura de ficheros

5. *Dado un fichero con varios números enteros por línea separados por comas hacer varios cálculos sobre el mismo:*

- Sumar los que sean primos
- Agrupar los enteros del fichero según el resto al ser divididos por un entero n

La primera versión que haremos será la funcional:

```
Integer sumaPrimos(String file) {
       return Streams.file(file)
              .flatMap(e->Streams.split(e,[ ,"]);
              .mapToInt(e->Integer.parseInt(e))
              .filter(e->Math2.esPrimo(e))
              .sum()
}
```

La versión imperativa requiere un iterador para recorrer el fichero y otro para recorrer las partes en las que se divide cada línea que es la forma de traducir *flatMap* a la notación imperativa. En cada lenguaje estos iteradores tomarán una forma particular. En Java sería:

```
Integer sumaPrimos(String file) {
	Iterator<String> fileIt = Iterators.file(file);
	Integer suma = 0;
	while(fileIt.hasNext()){
		String linea = fileIt.next();
		Iterator<String> lineaIt =
			Iterators.split(e,[" ,"]);
		while(lineaIt.hasNext()){
			String e = lineaIt.next();
			Integer en = Integer.parseInt(e))
			if(Math2.esPrimo(e)){
				suma = suma + en;
			}
		}
	}
	return suma;
}
```

Cuando usamos iteradores e iterables es más cómodo usar el for extendido proporcionado por Java.

```
Integer sumaPrimos(String file) {
	Iterable<String> fileIt = Iterables.file(file);
	Integer suma = 0;
	for(String linea: fileIt) {
		for(String e: Iterables.split(linea, "[ ,]")) {
			Integer en = Integer.parseInt(e);
			if (Math2.esPrimo(en)) {
				suma = suma + en;
			}
		}
	}
	return suma;
}
```

La versión recursiva final la podemos obtener consiguiendo un iterador a partir del iterador del fichero y un *flatmap*.

```
Integer sumaPrimos(String file) {
	Iterable<String> fileIt = Iterables.file(file);
	Iterable<String> ff =
		Iterables.flatMap(fileIt,
			linea->Iterables.split(linea,"[ ,]"));
	return sumaPrimos3(ff.iterator(),0);
}
Integer sumaPrimos(Iterator<String> ff, Integer b) {
	if(ff.hasNext()) {
		String p = ff.next();
		Integer en = Integer.parseInt(p);
		if (Math2.esPrimo(en)) {
			b = b + en;
		}
		b = sumaPrimos3(ff,b);
	}
	return b;
}
```

Veamos ahora la forma de agrupar los números del fichero.

```
Map<Integer,List<Integer> agrupaPorResto(
		String file,Integer n) {
	return Streams.file(file)
		.flatMap(e->Streams.split(e,"[ ,]");
		.map(e->Integer.parseInt(e))
		.collect(Collectors.groupingBy(e->e%n));
}
```

Y la versión imperativa usando for extendido:

```
Map<Integer, List<Integer>> agrupaPorResto2(String file,
Integer n) {
	Iterable<String> fileIt = Iterables.file(file);
	Map<Integer, List<Integer>> grupos = new HashMap<>();
	for(String linea:fileIt) {
		for(String e:Iterables.split(linea, "[ ,]")) {
			Integer en = Integer.parseInt(e);
			Integer key = en % n;
			List<Integer> ls;
			if (grupos.containsKey(key)) {
				ls = grupos.get(key);
			} else {
				ls = new ArrayList<>();
				grupos.put(key, ls);
			}
			ls.add(en);
		}
	}
	return grupos;
}
```

La estructura es similar teniendo en cuenta la función acumulación

```
Integer en = Integer.parseInt(e);
Integer key = en % n;
List<Integer> ls;
if (grupos.containsKey(key)) {
	ls = grupos.get(key);
} else {
	ls = new ArrayList<>();
	grupos.put(key, ls);
}
ls.add(en);
```

Comprobar si una lista está ordenada.

Para comprobar si una lista está ordenada podemos usar un diseño recursivo con los siguientes elementos:

- Podemos generalizar la secuencia mediante una generalización prefija. Los problemas que parecen son de la forma (i, ls) siendo ls la lista.

- Con esta generalización obtenemos la definición recursiva que expresa la solución de un problema en base a la solución de otro menor y concretamos los casos base

$$ord(i, ls) = \begin{cases} true, & n - i < 2 \\ ls[i] \leq ls[i+1] \wedge ord(i+1, ls), & n - i \geq 2 \end{cases}$$

La definición recursiva indica que una lista es ordenada si tiene un tamaño menor que 2 o dado un i se cumple que la casilla i es menor o igual que la siguiente y el resto de la lista está ordenada.

```
Boolean menorOIgual(E a, E b, Comparator<E> cmp){
       return cmp.compare(ls.get(i),ls.get(i+1))<=0;
}

Boolean estaOrdenada(List<E> ls, Comparator<E> cmp) {
       return estaOrdenada(0,ls,cmp);
}
Boolean estaOrdenada(int i, List<E> ls, Comparator<E> cmp) {
   Boolean r = true;
   Integer n = ls.size();
   if(n-i > 1) {
       r = menorOIgual(ls.get(i),ls,get(i+1), cmp) &&
              estaOrdenada(i+1,ls,cmp);
   }
   return r;
}
```

Observamos que es un algoritmo recursivo lineal no final con una operación de acumulación de la forma $b = b \wedge ls[i] \leq ls[i+1]$. Como el operador $\wedge$ es conmutativo podemos obtener una versión iterativa en a la forma:

```
Boolean ord(List<E> ls, Comparator<E> cmp) {
       Integer i = 0;
       Boolean b = true;
Integer n = ls.size();
       while(n-i>1) {
              E e1 = ls.get(i);
              i++;
              b = b && cmp.compare(e1,e2)<=0;
       }
       return b;
}
```

Pero también sabemos que el operador &&, implementación del operador matemático ∧ tiene propiedades de cortocircuito. Es decir, no evalúa el segundo operando si el primero es *false.* Estas propiedades permiten introducir una función de cortocircuito en el *while* y simplificar la función de acumulación.

```
Boolean ord(List<E> ls, Comparator<E> cmp) {
       Integer i = 0;
       Boolean b = true;
Integer n = ls.size();
       while(n-I > 1 && b) {
              E e1 = ls.get(i);
              i++;
              b = cmp.compare(e1,e2)<=0;
       }
       return b;
}
```

Aquí aparece un acumulador (*all*) lo que nos permite obtener la versión funcional. Esta versión la podríamos haber obtenido usando un diseño iterativo y combinando la secuencia de enteros $[0, n-1)$ con el acumulador *all.*

```
Boolean ord(List<E> ls, Comparator<E> cmp){
       return IntStream.range(0,ls.size()-1)
              .allMatch(
                     i->cmp.compare(ls.get(i),ls.get(i+1))<=0);
}
```

Encontrar el mayor en una lista ordenada y rotada

---

6. Dado un vector ordenado y rotado k veces, diseñar un algoritmo Θ(log n) que encuentre el elemento mayor del vector, suponiendo que no conocemos el valor de k.

Este problema lo podemos reducir a la búsqueda dicotómica teniendo en cuenta que el vector ordenado y rotado *k* veces se compondrá en general de dos partes ordenadas pero el extremo izquierdo de una de ellas tendrá sus elementos mayores que el extremo izquierdo. Tenemos que comprobar por esa propiedad en que zona estamos para seguir buscando

en una zona u otra. Usando una generalización central tenemos la definición recursiva

$$my(i,j,k,ls) = \begin{cases} ls[i], & j-i=1 \\ my2(ls[i], ls[i+1]), & j-i=2 \\ my(k,j,\frac{k+j}{2}ls), & j-i>2 \wedge ls[0] < ls[k] \\ my(i,k,\frac{i+k}{2}ls), & j-i>2 \wedge ls[0] > ls[k] \end{cases}$$

Problema del palíndromo

Dada una secuencia indexable se dice que es un palíndromo si la secuencia es simétrica respecto del centro. Es decir, si son iguales el primer y último elemento, el segundo y el penúltimo y así sucesivamente. Abordamos el problema con una generalización central y reduciendo en problema a otro de tamaño $n-2$. La definición recursiva es de la forma:

$$pl(i,j,ls) = \begin{cases} true, & j-i<2 \\ ls[i] = ls[j-1] \wedge pl(i+1, j-1, ls), & j-i \geq 2 \end{cases}$$

La definición tiene muchos elementos en común con problemas anteriores por lo que seguimos las mismas ideas.

```
boolean esPalindromo(String p) {
	return esPalindromo4(0,p.length(),p);
}
boolean esPalindromo4(int i, int j, String p) {
	boolean b = true;
	if (j - i > 1) {
		b = p.charAt(i) == p.charAt(j-1) &&
			esPalindromo4(i + 1, j - 1, p);
	}
	return b;
}
```

La versión iterativa

```
boolean esPalindromo(String p) {
	int i = 0;
	int j= p.length();
	Boolean b = true;
	while(j - i > 1 && b) {
		i = i + 1;
		j = j - 1;
		b = p.charAt(i) == p.charAt(j-1);
	}
	return b;
}
```

Y la funcional

```
boolean esPalindromo1(String st) {
	UnaryOperator<IntPair> next =
		p -> IntPair.of(p.first()+1, p.second()-1);
	Stream<IntPair> s =
		Stream.iterate(IntPair.of(0,st.length()),
			p->(p.second()-p.first())> 1, next);
	return s.allMatch(p->st.charAt(p.first()) ==
		st.charAt(p.second()-1));
}
```

Índice de un elemento en una secuencia

Se trata de encontrar el índice del elemento *e* en la secuencia *ls* y si no lo contiene entonces devolver -1. Abordemos el problema siguiente con diferentes hipótesis

- Asumir que la secuencia es indexable y ordenada
- Asumir que la secuencia es indexable y no está ordenada
- Asumir que la secuencia es no indexable y ordenada
- Asumir que la secuencia es no indexable y no ordenada

Secuencia indexable.

Para las secuencias indexables usamos una generalización central o sufija según que esté ordenada o no.

Para la generalización central escogemos la tupla (i, j, k), $i \in [0, n]$, $j \in [0, n]$, $k \in [0, n]$, siendo $n = ls.size()$ y podemos conseguir la siguiente definición recursiva:

$$idx(i,j,k,ls,e) = \begin{cases} -1, & j-i=0 \\ -1, & j-i=1 \ \wedge ls[k] \neq e \\ k, & j-i \geq 1 \ \wedge ls[k] = e \\ idx(i,k,\frac{i+k}{2},ls), & j-i>1 \wedge ls[k] < e \\ idx(k,j,\frac{k+j}{2},ls), & j-i>1 \wedge ls[k] > e \end{cases}$$

En esta generalización siempre se cumple $k = \frac{j+i}{2}$. La deinición sigue del hecho que si $ls[k] < e$ habrá que buscar en el lado izquierdo y en caso contrario en el derecho.

Con la generalización sufija podemos conseguir la siguiente definición:

$$idx(i,ls,e) = \begin{cases} -1, & n-i=0 \\ i, & n-i>0 \ \wedge ls[i] = e \\ idx(i+1,ls,e), & n-i>0 \wedge ls[k] \neq e \end{cases}$$

Ambas definiciones son recursivas finales por lo que podemos obtener directamente la versión iterativa y la funcional. Claramente las complejidades son muy distintas. La primera es $\Theta(log\ n)$ y es adecuada para secuencias indexables ordenadas. La segunda es $\Theta(n)$ y es adecuada para secuencias indexables no ordenadas.

La versión iterativa equivalente a la primera con elementos innecesarios eliminados es:

```
<E> Integer index5(List<E> ls,E elem, Comparator<E> cmp) {
      Integer b = -1;
      Integer i = 0;
      Integer j = ls.size();
      while(j-i>0 && b==-1){
            Integer k = (j+i)/2;
      if (Comparator2.isEQ(ls.get(k),elem,cmp)) b = k;
            else if (
               Comparator2.isLT(ls.get(k),elem,cmp)) i = k;
            else j = k;
      }
      return b;
```

La versión iterativa equivalente a la segunda es:

```
<E> Integer index(List<E> ls, E elem) {
Integer i = 0;
Integer n = ls.size();
Integer b = -1;
while(i < n && b == -1){
       E e = ls.get(i);
       if(e.equals(elem)) b = i;
       i = i+1;
}
return b;
}
```

Secuencia no indexable.

Si la secuencia no es indexable debemos disponer de un iterador. Como hemos visto en capítulos anteriores un iterador tiene asociados dos métodos: *next* y *hasNext*. Si queremos hacer una definición recursiva necesitamos una versión de iterador mediante funciones sin efectos laterales. Estas eran: *hn, nx1, nx2*.

Una definición recursiva final para secuencias no ordenadas descritas por un iterador la obtenemos generalizando el problema a (it, i, b, ls, el)

$$idx(t,i,ls,el) = \begin{cases} -1, & !\,hn(t) \\ i, & hn(t) \wedge nx1(t) = el \\ idx(nx2(t), i+1, ls, el), & hn(t) \wedge nx1(t) \neq el \end{cases}$$

Que puede ser inicializada con $idx(it(d),0,-1,ls,el)$

Si la secuencia está ordenada podemos incluir una condición más

$$idx(t,i,ls,el) = \begin{cases} -1, & !\,hn(t) \\ i, & hn(t) \wedge nx1(t) = el \\ -1, & hn(t) \wedge nx1(t) > el \\ idx(nx2(t), i+1, ls, el), & hn(t) \wedge nx1(t) < el \end{cases}$$

Las versiones iterativas se pueden obtener directamente de la definición recursiva. Pero hay que tener en cuenta que tenemos los métodos hasNext y next del iterador si este tiene oculto el estado. Para hacerlo posible introducimos la variable b inicializada a -1 que recuerda si se ha cumplido

la condición nx1(t) = el o no. Una versión iterativa para secuencias indexables no ordenadas es por tanto

```
Integer index(Iterable<E> iterable, E elem) {
	Integer i = 0;
	Integer b = -1;
	Iterator<E> it = iterable.iterator();
	while(it.hasNext() && b == -1){
		T e = it.next();
		if(e.equals(elem)) b = i;
		i = i + 1;
	}
	return b;
}
```

El algoritmo anterior puede ser obtenido directamente con las técnicas iterativas combinando la secuencia definida por el iterador con un acumulador con base (i, b) cuya función de acumulación en (i, b, e)−> $(i + 1, e = elem? b + 1: b)$ y función de retorno b.

Para secuencias ordenadas introducimos la variable e que recuerda el último valor devuelto por el iterador.

```
Integer index(Iterable<E> it, E elem, Comparator<E> cmp) {
	Integer i = 0;
	Integer b = -1;
	Iterator<E> itt = it.iterator();
	E a = null;
	while(itt.hasNext() && b == -1 && ((a==null) ||
		cmp.compare(a,elem)<=0)){
			a = itt.next();
			if(a.equals(elem)) b = i;
			i = i +1;
	}
	return b;
}
```

El algoritmo anterior puede ser obtenido directamente con las técnicas iterativas combinando la secuencia definida por el iterador con un acumulador con base (i, b) cuya función de acumulación en (i, b, a, e)−> $(i + 1, e = elem? b + 1: b, e)$ y función de retorno b.

Problema del entero más cercano

Este problema busca el elemento más cercano a uno dado en una lista. La lista puede estar ordenada o no.

Si la lista está ordenada el problema es similar al de la búsqueda dicotómica y como en aquel caso siendo una secuencia indexable podemos usar generalización central cerrada. Es decir que los dos subproblemas comparten el elemento central. Dada la lista ls la generalizamos a (i, j, ls). Los subproblemas serán $(i, k+1, ls)$ y (k, j, ls) con $k = (j+i)/2$ una propiedad derivada. El tamaño del problema es $t = j - i$. Podemos seguir las ideas de la búsqueda dicotómica, pero en este caso, para ver otra perspectiva, los vamos a resolver usando vistas. En este caso una *View2E* y usando los esquemas correspondientes.

```
Integer cercano2(List<Integer> ls, Integer e) {
       int n = ls.size();
       Integer r = null;
       if(n<1) Preconditions.checkArgument(!ls.isEmpty());
       else if (n == 1) r = ls.get(0);
       else if (n == 2) r = masCercano(ls.get(0),
                     ls.get(1), e);
       else if (n > 2) {
              View2E<List<Integer>, Integer> w =
                     List2.view2eOverlap(ls);
              if (e == w.e()) r = w.e();
              else if (e < w.e()) r = cercano2(w.left(), e);
              else r = cercano2(w.right(), e);
       }
       return r;
}
```

Inversión de funciones monótonas

Una función $f(x)$ es monótona si $x_1 < x_2 \rightarrow f(x_1) \leq f(x_2)$. O también $x_1 < x_2 \rightarrow f(x_2) - f(x_1) \geq 0$. Podemos usar el algoritmo anterior del entero más cercano para resolver el siguiente problema: dado un valor y_0 encontrar x_0 en el intervalo $[a, b]$ tal que sea mínimo $|f(x_0) - y_0|$. Planteado el problema así se reduce al problema del valor más cercano. Dado $f(x)$ y y_0 para que sea posible encontrar x_0 debemos imaginar los valores a, b que definan un intervalo de búsqueda que contenga la solución.

Casos particulares del problema anterior, asumiendo que $f(x)$ es monótona, son:

- Buscar la existencia o no de un x_0 tal que $f(x_0) = y_0$.
- Buscar x_0 en $[a, b)$ tal que se minimice $|f(x_0) - y_0|$.
- Buscar el máximo valor de x_0 tal que $f(x_0) \leq y_0$.

Veamos algunos ejemplos de aplicación de estas ideas.

---

*7. **Buscar un elemento que coincida con su índice en una lista de enteros ordenados decrecientemente**.*

Sea *ls* la lista de entrada ordenada decrecientemente. Vemos que la función $f(i) = i - ls[i]$ es monótona creciente. En efecto

$$i_1 < i_2 \rightarrow f(i_2) - f(i_1) = i_2 - ls[i_2] - i_1 + ls[i_1]$$
$$= i_2 - i_1 + ls[i_1] - ls[i_2] > 0$$

---

*8. **Dados los enteros positivos** a, n **calcular** $s_0 = \sqrt[n]{a}$. **Siendo** s_0 **el entero que hace mínima la expresión** $(|a - {s_0}^n|)$*

Podemos ver que la función s^n es monótona crciente respecto a s. El intervalo de búsqueda será $[0, a)$.

---

9. Dados los enteros positivos a, b calcular $s_0 = log_b\, a$ con una precisión ε.

Podemos ver que la función a^s es monótona creciente con respecto a s.

10. Dado un número m, compruebe que el número es Stella Octangula o no. Un número es Stella Octangula si es de la forma $m = n(2n^2 + 1)$ *donde n es un número entero.*

Como antes podemos comprobar que la función $f(n) = n(2n^2 + 1)$ es monótona con respecto a n. En efecto:

$$n_1 < n_2 \rightarrow f(n_2) - f(n_1) = n_2(2{n_2}^2 - 1) - n_1(2{n_1}^2 - 1)$$
$$= 2({n_2}^3 - {n_1}^3) + n_2 - n_1 > 0$$

La solución del problema es buscar el valor s_0 tal que sea mínimo $|f(s_0) - m|$ y comprobar que $f(s_0) = m$.

Definiciones recursivas a partir de propiedades

11. Diseñar un algoritmo para multiplicar enteros a partir de las propiedades siguientes:

$$xy = \begin{cases} 2x\frac{y}{2}, & y\%2 = 0 \\ 2x\frac{y}{2} + x, & y\%2 = 1 \end{cases}$$

$$x0 = 0$$

$$xy = yx$$

Una definición recursiva es:

$$ml(x, y) = \begin{cases} 0, & y = 0 \\ 2 * ml(x, \frac{y}{2}), & y > 0 \ \wedge \ y\%2 = 0 \\ 2 * ml(x, \frac{y}{2}) + x, & y > 0 \ \wedge \ y\%2 = 1 \end{cases}$$

Como podemos observar se define una secuencia $y, \frac{y}{2}, \frac{y}{4}, \ldots$ que es acumulada por la derecha con la función de acumulación $b = 2b + (y\%2 = 0?\ 0: x)$ cuyo operador no es asociativo. El algoritmo recursivo es:

```
Integer ml(Integer x, Integer y){
	Integer r = 0;
	if(y > 0) {
		r = 2*ml(x,y/2)+(y % 2 == 0? 0: x);
	}
	return r;
}
```

Pero no podemos encontrar directamente un equivalente iterativo. Para hacerlo buscamos otra forma de expresar la multiplicación descomponiendo y en potencias de 2.

$$xy = x\sum_{i=0}^{r} d_i 2^i = \sum_{i=0}^{r} x d_i 2^i$$

La expresión es el valor de un polinomio para *v=2* y coeficientes xd_i. Donde d_i son los dígitos en binario de y. Los d_i son *0 o 1,* luego los coeficientes son 0 o x. Podemos obtener un algoritmo iterativo evaluando el polinomio con los coeficientes de menor a mayor. Este algoritmo de evaluación de polinomios lo veremos más adelante. También veremos la forma de obtener los dígitos binarios de un entero de menor a mayor relevancia. Con todo ello obtenemos el algoritmo:

```
Integer ml(Integer x, Integer y){
	Integer b = 0;
	Integer p = 1;
	while(y>0) {
		Integer cf = y % 2 == 0? 0: x;
		p = p*2;
		b = b + cf;
	}
	return b;
}
```

12. Obtener una defición recursiva para evaluar $f(x) = e^x - 1, 0 < x < 1$ con una precisión $\boldsymbol{\varepsilon}$ dada la propiedad

$$f(x) = e^x - 1 = \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

La definición recursiva

$$f(x,i,t) = \begin{cases} 0, & t\frac{x}{i+1} \leq \varepsilon \\ f\left(x, i+1, t\frac{x}{i+1}\right) + t\frac{x}{i+1}, & t\frac{x}{i+1} > \varepsilon \end{cases}$$

Esto nos proporciona un algoritmo iterativo directamente por las propiedades de la función de acumulación.

```
Double exp(Double x, Double eps){
       Double r = 0.;
       Integer i = 0;
       Double t = 1.;
       while(t>eps) {
              i = i+1;
              t = t*x/i;
              r = r + t;
       }
       return r;
}
```

Las ideas anteriores pueden servir de base para calcular otras funciones como:

- e^x con x no restringido a $0 \leq x < 1$. Ya que $e^{n+y} = e^n e^y$ con $n = int(x), y = x - n$.
- $a^x = e^{x \ln a}$
- El cálculo de a^n con n entero lo veremos abajo

13. Diseñar una definición recursiva para decidir si a es múltiplo de b a partir de las propiedades

$$esMultiplo(a,b) = esMultiplo(a-b,b), a >= b$$
$$esMultiplo(0,b) = true$$
$$esMultiplo(a,b) = false, 0 < a < b$$

La definición recursiva es

$$em(a,b) = \begin{cases} false, & a < b, \\ true, & a = 0, b > 0 \\ em(a-b,b), & a \geq b, a > 0 \end{cases}$$

La definición es recursiva final por lo que disponemos de una versión iterativa.

Fusión de secuencias ordenadas

La fusión de secuencias ordenadas pretende obtener una secuencia ordenada a partir de otras dos que también lo están. La secuencia puede ser indexable o no. En ambos casos podemos obtener un diseño iterativo con una generalización y un invariante. Si las secuencias son indexables y adoptamos una generalización sufija para las listas entrada aparecen los problemas $(i1, ls1), (i2, ls2)$ y la lista de salida $ls3$. El invariante enunciado verbalmente es:

- *ls3* está ordenada
- Las frecuencias de cada elemento en *ls3* es la suma de las que tiene en *ls1[0,i1]* y *ls2[0,i2]*

El tamaño del problema es *n1+n2-i1-i2.* Partimos del problema inicial $(0, ls1), (0, ls2), ls3 = []$. En cada paso debemos escoger el menor entre el primer elemento de $(i1, ls1)$ y el de $(i2, ls2)$. El elemento escogido pasa a la lista e salida y índice de la correspondiente lista de entrada aumenta en 1. De esta forma se mantiene el invariante y se reduce el tamaño del problema en 1.

```
List<E> mezclaOrdenada(List<E> ls1, List<E> ls2, Comparator<E>
cmp) {
	List<E> ls3 = new ArrayList<>();
	Integer i1, i2;
	Integer n1 = ls1.size();
	Integer n2 = ls2.size();
	i1 = 0;
	i2 = 0;
	while (n1 + n2 – i1 -i2 > 0) {
		E e;
		if (i2 >=n2 ||i1 <n1 &&
Comparator2.isLE(ls1.get(i1),ls2.get(i2), cmp)) {
			e = ls1.get(i1);
			i1 = i1 + 1;
		} else {
			e = ls2.get(i2);
			i2 = i2 + 1;
		}
		ls3.add(e);
		}
	return ls3;
}
```

Si la secuencia es no indexable usamos un iterador para cada secuencia de entrada y seguimos las mismas ideas que en el caso anterior.

```
void mezclaOrdenada(Iterable<E> it1, Iterable<E> it2,String
fileOut,Comparator<E> cmp) {
	Iterator<E> itt1 = it1.iterator();
	Iterator<E> itt2 = it2.iterator();
	PrintStream f = Printers.file(fileOut);
	E e1 = itt1.hasNext()?itt1.next():null;
	E e2 = itt2.hasNext()?itt2.next(): null;
	while(e1 != null || e2 != null){
		E e=null;
		if(e2==null || Comparator2.isLE(e1,e2,cmp)){
			e = e1;
			e1 = itt1.hasNext()?itt1.next():null;
		} else {
			e = e2;
			e2 = itt2.hasNext()?itt2.next():null;
		}
		f.println(e.toString());
	}
	f.close();
}
```

Algunos problemas conocidos

Evaluación de polinomios

Un polinomio puede ser evaluado para un valor v de la variable. Un polinomio es una expresión de la forma:

$$p(x) = c_r x^r + c_{r-1} x^{r-1} + \ldots + c_0 = \sum_{i=0}^{r} c_i x^i$$

Su valor en *p(v)* es el que se obtiene sustituyendo *x* por *v*.

Los coeficientes del polinomio forman la secuencia $c = [c_0, c_1, .., c_r]$ si los ordenamos de menos significativo a más. La secuencia inversa que contiene los coeficientes ordenados de más a menos será $d = [c_r, c_{r-1}, \ldots, c_0]$. Diseñaremos diferentes algoritmos que usarán *c* o *d*. Sean *(i,c)* e *(i,d)* las generalizaciones sufijas de *c* y *d* respectivamente.

Para evaluar el polinomio una primera idea es combinar c con la secuencia $s = (1, e \to e * v)$ mediante zip(c,s), transformarla obteniendo el producto de ambas componentes del zip y acumular el resultado mediante un acumulador de suma.

```
Double polf(List<Double> c, Double v){
       return Stream2.zip(c.stream(),
                     Stream.iterate(1.,e->e*v),(e1,e2)->e1*e2)
              .mapToDouble(x->x).sum();
}
```

Desplegando el código anterior escogiendo un iterador que vaya recorriendo los índices de la lista y se componga mediante zip con la secuencia *s* nos da lugar a un estado $t = (i, p)$ cuyas funciones son:

- hn(i,p) = $i < n$
- nx1(i,p) = p*c[i]
- nx2(i,p) = (i+1,p*v)

La versión iterativa con ese iterador es:

```
Double polI(List<Double> c, Double v){
	Double b = 0.;
	Integer i = 0;
	Double p = 1.;
	Integer n = c.size();
	while(i<n) {
		Double cf = p*c.get(i);
		i = i+1;
		p = p*v;
		b = b + cf;
	}
	return b;
}
```

Otra posibilidad es basarlos en la propiedad:

$$c_r v^r + c_{r-1} v^{r-1} + \ldots + c_0 = c_0 + v(c_1 + \cdots + v(c_{n-1} + v(0 + c_n))$$

y generalizando la lista de coeficientes con una generalización sufija tenemos la definición recursiva

$$pol(i, c, v) = \begin{cases} 0, & r - i = 0 \\ c[i] + v * pol(i+1, c, v), & x \geq 0 \end{cases}$$

Con $r = |c| = n + 1$. Esta definición es recursiva lineal no final. Podemos ver que recorre la secuencia de índices de la lista de coeficientes y usa la función de acumulación $(b, i) \to b * v + c[i]$. Este acumulador lo llamaremos el *acumulador de Horner*. El código resultante es:

```
// Coeficientes de menor a mayor
Double polHD(List<Double> c, Double v){
	return polHD(c,v,0,c.size());
}
Double polHD(List<Double> c,Double v,Integer i,Integer r){
	Double b = 0.;
	if(i<r) {
		b = polHD(c,v,i+1,r);
		Double d = c.get(i);
		b = b*v + d;
	}
	return b;
}
```

Por lo explicado en la teoría anterior el *acumulador de Horner* puede ser acumulado por la izquierda siempre que usemos la secuencia inversa (coeficientes de mayor a menor):

```
// Coeficientes de mayor a menor
Double polHI(List<Double> d, Double v){
	Double b = 0.;
	Integer i = 0;
	Integer r = d.size();
	while(i<r) {
		Double dd = d.get(i);
		i = i + 1;
		b = b*v + dd;
	}
	return b;
}
```

Las dos estrategias anteriores ofrecen distintas posibilidades. La estrategia primera requiere formar una nueva secuencia a partir de la secuencia de coeficientes de menor a mayor y la secuencia de potencias. La secuencia no es fácilmente invertible pero el operador dela acumulador (+) es asociativo y conmutativo. Podríamos obtener una acumulación por la derecha con la misma secuencia y el mismo acumulador.

El *acumulador de Horner* no es asociativo y conmutativo. Si queremos acumular por la izquierda debemos invertir primero la secuencia. Es decir, tomar los coeficientes de mayor a menor.

Por lo explicado arriba si partimos de la lista de coeficientes ordenados de menor a mayor podemos acumularla por la derecha con el *acumulador de Horner*. Si partimos de la lista de coeficientes ordenados de mayor a menor podemos y acumularla por la izquierda con ese acumulador.

La secuencia formada mediante *zip* anterior, que usa los coeficientes ordenados de menor a mayor, podemos acumularla por la derecha o la izquierda con el acumulador *sum*.

Los algoritmos anteriores son adecuados para evaluar para un valor dado *x* de productos de la forma

$$\prod_{i=0}^{r} c_i x^i$$

Es evidente que si alguno de c_i es cero el producto también lo será. Cuando todos los c_i son distintos de cero podemos usar los mismos algoritmos que antes pero ahora el acumulador será producto en lugar de suma. Tomado los coeficientes de menor a mayor y acumulando por la izquierda tenemos el algoritmo siguiente que usaremos más abajo:

```
Double prodI(List<Double> c, Double v){
       Double b = 1.;
       Integer i = 0;
       Double p = 1.;
       Integer n = c.size();
       while(i<n) {
              Double cf = p*c.get(i);
              i = i+1;
              p = p*v;
              b = b * cf;
       }
       return b;
}
```

Problemas con dígitos de un número entero

Veamos ahora diversos problemas con dígitos de enteros en base *a*.

- Contar cuantos dígitos son cero en la representación en base *a* de *n*
- Obtener una lista con los dígitos de un número *n* en base *a*.
- A partir de una lista con los dígitos de un número *n* en base *a* obtener *n*.
- Dado un entero *n* en base *a* obtener otro *m* cuyos dígitos en la misma base estén invertidos. Ejemplo: sean inversos invierte (1024,10) =4201.

Sea *n* el entero, *a* la base, d_i los dígitos de n en esa base y r el orden del primer dígito entonces podemos expresar:

$$n = d_r a^r + \cdots + d_1 a^1 + d_0 = d_0 + d_1 a^1 + \cdots + d_r a^r$$

$$n\%a = d_0$$

En definitiva, un entero puede ser expresado como un polinomio cuyos coeficientes son los dígitos en una determinada base *a*. La secuencia de los

dígitos de menor a mayor significación, $d_0, d_1, \ldots, d_r$, puede ser generada por:

$$s = \left(n, e \to e > 0, e \to \frac{e}{a}\right).map(e \to e\%a)$$

Esta secuencia genera los dígitos de menor a mayor: es decir $[d_0, d_1, .., d_r]$. Si en esa secuencia filtramos los dígitos que son cero y la acumulamos con el acumulador contador obtenemos el primer resultado. El acumulador contador puede acumularse por la izquierda o por la derecha. Por la izquierda en forma iterativa resulta:

```
Integer numeroDeCeros(Integer n, Integer a){
	Integer b = 0;
	while(n>0) {
		Integer d = n%a;
	if(d == 0){
		b = b+1;
	}
		n = n/a;
	}
	return b;
}
```

14. Obtener la representación de n en base a.

Si acumulamos la secuencia *s* anterior con el acumulador *toList* obtendremos una lista con los dígitos. Dado que el operador concatenar listas es no conmutativo las listas que se obtienen acumulando por la izquierda y por la derecha son inversas. Como *s* genera los dígitos de menor a mayor y queremos la lista con los dígitos de mayor a menor acumulamos la secuencia por la derecha con el acumulador *toList*.

El código recursivo es:

```
List<Integer> digitos(Integer n, Integer a){
	List<Integer> r;
	if(n>a) {
		r = digitos(n/a,a);
		Integer d = n%a;
		r.add(d)
	} else {
		r = new Arraylist<>();
	}
	return r;
}
```

Este es un algoritmo recursivo no final, pero teniendo en cuentas las propiedades del operador de concatenación podemos disponer de un algoritmo iterativo:

```
List<Integer> digitosI(Integer n, Integer a){
	List<Integer> b = new ArrayList<>();
	while(n>0) {
		Integer d = n%a;
		b.add(0,d);
		n = n/a;
	}
	return b;
}
```

15. Obtener el entero asociado a una lista de dígitos

Es similar a evaluar el polinomio correspondiente donde la variable toma la base como valor. Como la secuencia está de mayor a menor acumulamos por la derecha. A partir de los dígitos de mayor a menor y acumulando por

la derecha la secuencia $[d_r, d_{r-1}, .., d_0]$ con el acumulador de Horner $ac = (0, (b, d) \to b * a + d)$ y resulta el código siguiente:

```
Long entero(List<Integer> digitos, Integer a){
	return entero(digitos, a,0,d.size());
}
Long entero(List<Integer> digitos,
		Integer a,Integer i,Integer n){
	Long b = 0;
	if(n-i> 0){
		b = entero(digitos, a, i + 1, n);
		Integer d = digitos[i);
		b = b * a + d;
	}
	return b;
}
```

---

16. Construir un entero m cuya representación en una base a sea inversa que la de n.

Partimos de la secuencia de dígitos de *n* generados por la secuencia

$$s = \left(n, e \to e > 0, e \to \frac{e}{n}\right).map(e \to e\%a)$$

Y la acumulamos por la izquierda con el acumulador ac anterior la secuencia de dígitos de mayor a menor obtenemos m que se ha pedido. Esto nos da:

```
Integer inverso(Integer n, Integer a){
	Integer b = 0;
	while(n > 0){
		Integer d =n%a;
		b = b * a + d;
		n = n/a;
	}
	return b;
}
```

El problema de la potencia entera

Veamos la forma de obtener un algoritmo iterativo de complejidad $\log n$ para la potencia entera. Una definición para la potencia entera es:

$$a^n = \begin{cases} 1, & n = 0 \\ (a^{n/2})^2 a, & n > 0, n\%2 == 1 \\ (a^{n/2})^2, & n > 0, n\%2 == 0 \end{cases}$$

Esta definición nos proporciona la secuencia $sq = (n, n \to n > 0, n \to n/2)$ y un acumulador por la derecha $ac = (1, cb(b, e))$ cuya la función de acumulación es:

$$c(b, n) = \begin{cases} ab^2, & n\%2 == 1 \\ b^2, & n\%2 == 0 \end{cases}$$

El código equivalente es:

```
Long pot(Long a, Integer n) {
       Long r = 1;
       if(n>0){
              r = pot(a,n/2);
              r = r*r;
              if(n%2 == 1) r = r * a;
       }
       return r;
}
```

En este caso podemos encontrar una secuencia *s'* y un acumulador a' que acumulados por la izquierda dan el mismo resultado que la anterior secuencia y acumulador acumulados por la derecha. Estas secuencia y acumulador son:

$$s1 = \left((n, a), (n, e) \to n > 0, (n, e) \to \left(\frac{n}{2}, e * e\right)\right)$$
$$s' = s1.filter((n, e) \to n\%2 == 1)$$
$$a' = (1, (b, (n, e)) \to b * e)$$

Y el algoritmo iterativo equivalente:

```
Long pot(Long a, Integer n){
       Long e = a;
       Long b = 1;
       while(n > 0){
              if(n%2==1){
                     b = b * e;
              }
              e = e * e;
              n = n/2;
       }
       return b;
}
```

La razón de lo anterior es tener en cuenta que:

$$a^n = a^{\sum_0^p d_i 2^i} = \prod_0^p a^{d_i 2^i} = \prod_0^p a^{d_i 2^i}$$

Podemos ver que la expresión de la derecha podemos considerarla como el producto (acumulador) de una secuencia cuyos elementos son: $a^{d_i 2^i}$. Siendo d_i la secuencia invertida de los dígitos binarios de n. Es decir lo valores 0, 1 por lo que los elementos de la secuencia son 1 si $d_i = 0$ y a^{2^i}, si $d_i = 1$. Observemos que $a^{2^{i+1}} = a^{(2^i * 2)} = (a^{2^i})^2 = a^{2^i} * a^{2^i}$. La secuencia invertida de dígitos binarios podemos encontrarla mediante la secuencia:

```
sd = (n,n->n>0,n/2).map(x->x%2);
```

Reuniendo las ideas anteriores la secuencia de los factores de la expresión de la derecha puede ser descrita por:

$$s = \left((n,a),(n,e) \to n > 0, (n,e) \to \left(\frac{n}{2}, e * e\right)\right)$$

$$.filter((n,e) \to n\%2 == 1)$$

El filtro es debido a que sólo debemos tener en cuenta los $d_i = 1$ ya que si $d_i = 0$ el factor se reduce a 1. Acumulando la secuencia anterior con el acumulador producto $a = (1,(b,(n,e)) \to b * e)$ obtenemos el resultado buscado y el algoritmo iterativo.

Problemas de algoritmos recursivos múltiples

Suma y multiplicación de enteros grandes

Los números enteros solemos representarlos en los algoritmos como *Integer* o *Long*: pero estos tipos tienen un conjunto de valores limitado. Los valores de *Integer* van de -2^31= -2,147,483,648 a 2*31-1=2,147,483,647 que ocupan 4 bytes de memoria

Los *Long* van de -2^63 = -9223372036854775808 a 2^63-1 = 9223372036854775807 que ocupan 8 bytes.

Para usar valores enteros más grandes necesitamos un tipo de datos de tamaño variable que pueda representar cualquier valor entero. Java nos proporciona el tipo *BigInteger.* Las operaciones de suma y multiplicación se hacen con complejidad $\Theta(1)$ pero los de tipo *BigInteger* tienen unas complejidades que dependen de su número de dígitos.

Un valor de tipo *BigInteger*, un entero largo n, lo podemos representar como una lista de dígitos $d_r d_{r-1} \dots d_0$ en una base b. El número de dígitos, comovimos en problemas anteriores, es $r = log_b\, n \cong \Theta(log\, n)$.

Queremos diseñar algoritmos de suma, reta, multiplicación, conciente, resto y potencia de enteros representados por listas de dígitos de tipo

Long y de base $b = 2\text{^}31$. En el repositorio se ofrece el tipo *LargeInteger* que tiene una funcionalidad similar a BigInteger.

La suma de valores de este tipo se puede hacer con el algoritmo clásico que se aprende en la escuela. Se van sumando los dígitos comenzando por los menos significativos y si se sobrepasa la base se anota una unidad, el acarreo, para acumularlo con el dígito siguiente. El algoritmo asumiendo que los dos sumandos tienen el mismo número de dígitos es:

```
LargeInteger sum(LargeInteger x, LargeInteger y) {
	Integer n = x.size();
	List<Long> r = new ArrayList<>();
	Long ac = 0L;
	for(int i=n-1;i>=0;i--) {
		Long sm = ac+x.digit(i)+y.digit(i);
		Long d = sm%base;
		ac = sm/base;
		r.add(0,d);
	}
	r.add(0,ac);
	return LargeInteger.ofLong(r);
}
```

Como podemos ver la suma tiene complejidad $\Theta(r)$. Es decir lineal en el número de dígitos y equivalentemente $\Theta(log\, n)$. Un algoritmo más completo para cuando el número de dígitos es diferente o algunos de los sumandos es cero está en el repositorio.

La multiplicación también se podría diseñar con el algoritmo aprendido en la escuela que tiene complejidad $\Theta(r^2)$. En este caso podemos diseñar un algoritmo recursivo con una complejidad más baja. Es el algoritmo de *Karatsuba*. Sean x, y los valores a multiplicar. Asumamos que las listas de dígitos las dividimos por la mitad y sea m el tamaño de la segunda mitad. Podemos observar que:

$$x = x_1 * b^m + x_0, \qquad y = y_1 * b^m + y_0$$

Y la multiplicación x ∗ y es:

$$x * y = z_2 * b^{2m} + z_1 * b^m + z_0$$

$$z_2 = x_1 * y_1,\ z_1 = (x_1 + y_0) * (x_0 + y_1)\ , z_0 = x_0 * y_0$$

A partir de esas propiedades podemos deducir una definición recursiva. El caso base será cuando x, y tengan un dígito. Vemos que el tamaño del problema es el número de dígitos. Cada problema se divide en dos partes iguales, hay cuatro llamadas recursivas y la función de combinación es a suma la suma de complejidad $\Theta(r)$.

La recurrencia definida es:

$$T(r) = 4T(r/2) + \Theta(r)$$

Cuya solución es $\Theta(r^2)$. Con este esquema recursivo no hemos ganado nada sobre el algortimo elemental. Pero podemos reordenar el cáculo de z_1 aprovechando los cálculos ya hechos para z_0, z_2 en la forma:

$$z_1 = (x_1 + x_0) * (y_0 + y_1) - z_2 - z_0$$

Ahora podemos observar que la ecuación de recurrencia es

$$T(r) = 3T(r/2) + \Theta(r)$$

Cuya solución es $\Theta(r^{\log_2 3}) = \Theta(r^{1.58})$.

La potencia de enteros largos se puede implementar con el algoritmo de la potencia entera usando la suma y multiplicación anteriores. Observando el algoritmo de la potencia entera y teniendo en cuenta que los valores de n siguen una progresión geométrica de razón 2, la complejidad de a^n, siendo a un entero largo de r dígitos y n un entero conrto, será:

$$\sum_{i=0}^{log_2 n} (r * 2^i)^{1.58} \cong f(n) * \theta(r^{1.58})$$

Con $f(n) = \sum_{i=0}^{log_2 n} 2^{1.58\, i}$. La explicación de lo anterior es que el número de pasos del algoritmo es $log_2 n$ y que los dígitos de un número se duplican al elevarse al cuadrado.

Problemas de listas

1. *Encontrar un conjunto con loe elementos de una lista en un rango*

Dada una lista ordenada de tipo genérico (List<E> lista), diseñar un algoritmo que devuelva un conjunto que incluya los elementos de dicha lista que se encuentren en un rango [a, b) dado (siendo a y b del mismo tipo que los elementos de la lista). Discutir los distintos algoritmos posibles y sus complejidades.

- La primera posibilidad es usar una generalización sufija

$$s(i, ls) = \begin{cases} \emptyset, & n - i = 0 \\ \{ls[i]\} \cup s(i+1, ls), & n - i > 0 \ \wedge ls[i] \in [a, b) \\ s(i+1, ls), & n - i > 0 \ \wedge ls[i] \notin [a, b) \end{cases}$$

Dadas las propiedades del operador ∪ podemos encontrar un algoritmo iterativo equivalente. La complejidad es $\Theta(n)$ como puede comprobarse

- La segunda posibilidad es usar una generalización central

$$s(i, j, k, ls)$$

$$= \begin{cases} \emptyset, & n - i = 0 \\ \emptyset, & n - i = 1 \ \wedge ls[i] \notin [a, b) \\ \{ls[i]\}, & n - i = 1 \ \wedge ls[i] \in [a, b) \\ s(i, k, \frac{k+i}{2}, ls) \cup s(k, j, \frac{j+k}{2}, ls), & n - i > 1 \ \wedge ls[k] \in [a, b) \\ s(i, k, \frac{k+i}{2}, ls), & n - i > 0 \ \wedge ls[k] > [a, b) \\ s(k, j, \frac{j+k}{2}, ls), & n - i > 0 \ \wedge ls[k] < [a, b) \end{cases}$$

La complejidad del caso peor es la solución de $T(n) = 2T(n/2) + 1$.

Es decir $\Theta(n)$. La del caso mejor $T(n) = T(n/2) + 1$. Es decir $\Theta(logn)$. La complejidad del caso medio depende de las probabilidades de

$$ls[k] \in [a, b), ls[k] > [a, b), ls[k] < [a, b)$$

Suponiendo que son $p_0, \frac{1-p_0}{2}, \frac{1-p_0}{2}$ la complejidad verifica

T(n)= $2p_0T(n/2) + (1-p_0)T(n/2) + 1 = (1+p_0)T(n/2) + 1$

Donde se cumple que $(1+p_0) < 2$ y por los tanto la complejidad media es $\Theta(n)$.

---

2. Dada una lista de elementos de tamaño n decidir si hay alguno de ellos que se repita n/2 veces o más.

- Usando un *Multiset*
- Usando el algoritmo de la bandera holandesa

La primera posibilidad consiste en construir un Multiset a partir de una lista y posteriormente revisar que existe un elemento cuya frecuencia es igual o superior a la mitad del tamaño de la lista. Como podemos comprobar complejidad $\Theta(n)$

```
E masDeLaMitad(List<E> ls) {
	Integer n = ls.size();
	Multiset<E> ms = Stream2.toMultiSet(ls.stream());
	return ms.elementSet().stream()
		.map(e->Pair.of(e,ms.count(e)))
		.filter(p->p.second()>=n/2)
		.findFirst()
		.map(p->p.first())
		.orElse(null);
}
```

La segunda posibilidad es usar el algoritmo de la *Bandera Holandesa.* Este algoritmo, como vimos, tras escoger un pivote reordena la lista en tres partes cuyos límites vienen definidos por un *IntPair*. Según el tamaño de cada parte continuará la búsqueda

```
E masDeLaMitad2(List<E> ls,Comparator<? super E> cmp) {
	IntPair p = masDeLaMitad2R(ls,cmp);
	return p==null?null:ls.get(p.first());
}

IntPair masDeLaMitad2R(List<E> ls,Comparator<? super E> cmp) {
	Integer n = ls.size();
	IntPair r;
	IntPair p = banderaHolandesa(ls,cmp);
	if(p.size() >= n/2) r = p;
	else if(p.first() >=n/2) r =
		masDeLaMitad2R(ls.subList(0,p.first()),cmp);
	else if(n-p.second() >=n/2) r =

	masDeLaMitad2R(ls.subList(p.second(),n),cmp);
	else r = null;
	return r;
}
```

Vemos que es un algoritmo recursivo con complejidad $\Theta(n)$ puesto que le problema tiene asociada la ecuación de recurrencia

$$T(n) = T(n/2) + \Theta(n)$$

Donde hemos supuesto que la bandera holandesa divide el problema en tres partes con la central pequeña y las otras dos cercanas a la mitad del tamaño. Esto no es siempre así por lo que haría falta un estudio más detallado del cao medio que no haremos aquí.

Como vemos ambos algoritmos tienen la misma complejidad. Preferimos el algoritmo iterativo.

Subsecuencia de suma máxima

El problema de la subsecuencia de suma máxima consiste en encontrar una sublista (en posiciones consecutivas) cuya suma sea máxima dentro de una lista original. Por ejemplo: en la lista *[-1.,6.,-2.,5.,-1.,4.,3.,-4.,3.]* . La subsecuencia de suma máxima es [*6.,-2.,5.,-1.,4.,3.]*, cuya suma es *15*.

Consideremos las tuplas (i, j, s) con $j > i$ y s la suma de la sublista $ls[i, j]$. Estas tuplas las ordenamos según s y escogemos la mayor. Si recorremos los valores de j dado un i fijo, actualizando s y la tupla mayor obtenemos un algoritmo de complejidad $\Theta(n^2)$.

Veamos ahora una solución recursiva:

```
SubSecuencia getSubSecuenciaMaxima(List<Double> lista){
	return subSecuenciaMaxima(lista,0,lista.size());
}

SubSecuencia subSecuenciaMaxima(
		List<Double> lista, int i, int j){
	SubSecuencia r = null;
	if(j-i <= 1){
		r = SubSecuencia.of(i,j,lista);
	}else{
		int k = (i+j)/2;
		SubSecuencia s1 =
			subSecuenciaMaxima(lista, i, k);
		SubSecuencia s2 =
			subSecuenciaMaxima(lista, k, j);
		SubSecuencia s3 =
			subSecuenciaMaximaCentrada(lista,i,j,k);
		r = Stream.of(s1, s2, s3)
			.max(Comparator.naturalOrder()).get();
	}
	return r;
}
```

Este algoritmo tiene una recurrencia asociada

$$T(n) = 2T(n/2) + \Theta(n^k)$$

Donde $\Theta(n^k)$ es la complejidad de *subsecuenciaMaximaCentrada*. Para rebajar la complejidad del algoritmo recursivo a $\Theta(n^2)$ hace falta que $k = 1$. De esta forma consiguiremos que sea $\Theta(n \log n)$.

El algoritmo *subSecuenciaMaximaCentrada* siguiente tiene la complejidad requerida de $\Theta(n)$.

```
SubSecuencia subSecuenciaMaximaCentrada(
              List<Double> ls, int i, int j, int k){
       Double suma = 0.;
       SubSecuencia smax = SubSecuencia.of(k,k,0., ls);
       for(Integer i1 = k-1;i1>=i;i1--) {
              suma = suma + ls.get(i1);
              SubSecuencia s = SubSecuencia.of(i1,k,suma, ls);
              if(s.compareTo(smax) >0) smax = s;
       }
       suma = smax.suma();
       Integer i1 = smax.from();
       for(Integer j1 = k;j1<j;j1++) {
              suma = suma + ls.get(j1);
              SubSecuencia s =
                     SubSecuencia.of(i1,j1+1,suma,ls);
              if(s.compareTo(smax) >0) smax = s;
       }
       return smax;
}
```

Problema del histograma

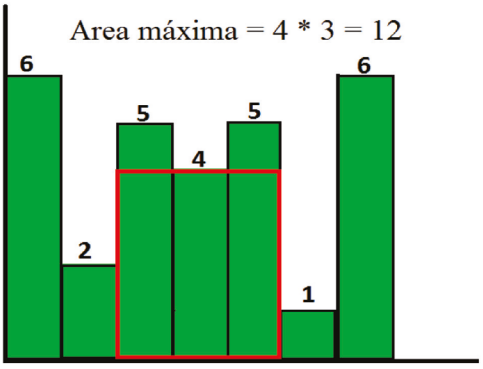


Ilustración 20: Histograma

Dado una lista de números enteros que representa un histograma, diseñe un algoritmo que determine la mayor área rectangular que se puede encontrar. Por simplicidad considere que todas las barras tienen anchura de 1 unidad.

Este es un problema complejo que requiere un buen diseño de tipos y funciones para poder la solución con éxito. Veamos, en primer lugar, los tipos y funciones necesarios.

Par concretar suponemos que las alturas del histograma vienen dadas en una lista de enteros *ls.* Diseñamos la solución del problema, el tipo *Sol*, como una tupla $(from, to, minI)$. Donde *from, to* son los dos extremos del área, *minI* el índice que tiene la altura mínima en el intervalo *[from,to).* La tupla Sol tiene, además la propiedad derivad área, *a = ls[minI]*(to-from).* A las tuplas anteriores le asignamos una relación de orden según la propiedad *a*.

Diseñamos el tipo *SolH* con esas características

```
record SolH(Integer from, Integer to,
            Integer minI, Integer a, List<Integer> ls)
      implements Comparable<SolH> {

      public static SolH of(Integer from, Integer to,
                  Integer minI,List<Integer> ls) {
            Integer h = ls.get(minI);
            Integer a = (to - from) * h;
            return new SolH(from, to, minI, a, ls);
      }

      public static SolH zero() {
            return of(-1, -1, -1, null);
      }

      @Override
      public int compareTo(SolH other) {
            return this.a().compareTo(other.a());
      }
}
```

Además, usaremos las funciones

- *int minIndex(int i, int j, List<Integer> ls):* el índice cuyo valor es menor
- *int minIndex(int i, int j, int k, List<Integer> ls):* el índice cuyo valor es menor

Con estos elementos podemos diseñar un primer algoritmo teniendo en cuenta que se trata de buscar el mínimo para todos los valores posibles de (i, j, k). Donde i define la parte inferior del intervalo, j la parte superior y k el índice donde el valor es minimo en el intervalo $[i, j)$. Podemos comprobar que el rango de estos índices es $i \in [0, n)$, $j \in [i + 1, n]$, $k \in [i, j)$.

```
SolH histograma0(List<Integer> ls) {
	Integer n = ls.size();
	SolH amax = SolH.zero();
	for (Integer i = 0; i < n; i++) {
		for (Integer j = i+1; j < n; j++) {
			Integer m = i;
			for(Integer k=i+1;k<j;k++) {
				m = minIndex(ls,m,k);
			}
			SolH e = SolH.of(i,j,m,ls);
			if (e.compareTo(amax) > 0) amax = e;
		}
	}
	return amax;
}
```

Que podemos comprobar que es de complejidad $\Theta(n^3)$.

Una segunda aproximación al problema recorrer los índices (i, j) con $i \in [0, n)$, $j \in [i+1, n]$ y mantener el invariante: m es el índice con menor valor en el intervalo $[i, j)$. Así resulta el algoritmo:

```
SolH histograma1(List<Integer> ls){
	Integer n = ls.size();
	SolH em = SolH.zero();
	for(Integer i = 0; i < n; i++){
		Integer m = i;
		for(Integer j = i+1; j <= n; j++){
			m = minIndex(ls,j-1,m);
			SolH e = SolH.of(i,j,m,ls);
			if (e.compareTo(em) > 0) em = e;
		}
	}
	return em;
}
```

Que podemos comprobar que es de complejidad $\Theta(n^2)$.

Una tercera aproximación al problema es enfocarlo de forma recursiva, generalizando el problema y dividiéndolo en dos partes iguales.

```
SolH histograma2(List<Integer> ls){
	Integer n = ls.size();
	return histograma2(0,n,ls);
}

SolH histograma2(Integer i, Integer j, List<Integer> ls){
	SolH rm;
	if(j-i>1){
		Integer k = (j+i)/2;
		SolH r1 = histograma2(i,k,ls);
		SolH r2 = histograma2(k,j,ls);
		SolH r3 = centro(i,j,k,ls);
		rm = List.of(r1,r2,r3).stream()

	.max(Comparator.naturalOrder()).get();
	} else if (j-i == 1) {
		rm = SolH.of(i,j,i,ls);
	} else {
		rm = SolH.zero();
	}
	return rm;
}
```

Como vemos la solución del problema será la de mayor área de la parte izquierda, o de la parte derecha o la solución de área máxima centrada en *k*. Es decir, con *i* a la izquierda de *k* y *j* a la derecha. Para acabar de resolver el problema tenemos que diseñar el algoritmo que calcula la solución centrada. Una primera posibilidad es usar el algoritmo histograma1 anterior con $i1$, $j1$ moviéndose en los intervalos $i1 \in [i, k)$, $j1 \in [k+1, j]$. Pero al ser $histograma1$ de complejidad $\Theta(n^2)$ la ecuación de recurrencia asociada es:

$$T(n) = 2T(n/2) + n^2, \qquad \Theta(n^2)$$

Por esta vía no mejoramos en complejidad. Para que podamos mejorar la complejidad necesitamos que el algoritmo centro sea de complejidad $\Theta(n)$ para que la ecuación de recurrencia asociada sea:

$$T(n) = 2T(n/2) + n, \qquad \Theta(n \log n)$$

Busquemos un algoritmo que calcule la solución centrada con complejidad $\Theta(n)$. La idea es expandir la solución centrada más pequeña ubicada en los limites $(k-1, k+1)$ y el índice con valor mínimo en $k-1$. La idea es expandir esos límites hacia izquierda o derecha según el valor mayor de la nueva casilla y mantener la solución con mayor área. El algortimo resultante de complejidad $\Theta(n)$ es:

```
SolH centro(Integer i, Integer j, Integer k,
              List<Integer> ls) {
       Integer i1 = k-1, j1 = k+1;
       Integer min_index = minIndex(ls,i1,j1-1);
       SolH amax = SolH.zero();
       while(i1 >= i && j1 <= j) {
       min_index = minIndex(ls,min_index,i1,j1-1);
              SolH r = SolH.of(i1,j1,min_index,ls);
              if(r.compareTo(amax) > 0) amax = r;
              if (i1 == i) j1++;
              else if (j1 == j) i1--;
              else {
                     if (ls.get(i1-1) >= ls.get(j1))  i1--;
                     else j1++;
               }
       }
       return amax;
}
```

El algoritmo anterior mantiene el invariante: *min_index* es el índice con menor valor en todo el intervalo *[i1,j1)* y *amax* es la solución con mayor área en el intervalo *[i1,j1).* Como podemos comprobar que el algoritmo no recorre todos los pares *(i1,j1)* posibles (eso daría complejidad de $\Theta(n^2)$). Frente a eso amplia el intervalo a izquierda a derecha en la dirección que puede crecer el área. Esto lo consigue eligiendo el nuevo índice cuyo valor es mayor.

Problemas de matrices

3. *Obtener la suma de dos matrices:*

- De forma iterativa
- De forma recursiva dividiéndolas en cuatro submatrices

Para enfocar el problema proponemos un sistema de coordenadas para nombrar las casillas de la matriz. Nombremos las filas de arriba abajo por i y las columnas por j de tal forma que (0,0) es la casilla superior izquierda. Una matriz vendrá definida por su número de filas y columnas: $nf \times nc$. Una *submatriz* es una vista de una matriz definida por un vértice superior izquierdo y un tamaño de filas y columnas. Es decir dada una matriz m una posible submatriz es $sm = (m, i0, nf, j0, nc)$ con la restricciones adecuadas. Es una idea similar a *sublist.*

Una matriz la podemos dividir en cuatro vistas que denominaremos *m0, m1, m2, m3*. La vista *m0* vendrá definida $(m, 0, nf/2, 0, nc/2)$ y de forma similar el resto. Cada submatriz tiene unas coordenadas locales tales que el vértice superior izquierdo es (0,0). Evidentemente una matriz se puede ver como una submatriz $sm = (m, 0, nf, 0, nc)$. Llamemos *Matrix<E>* al tipo que representa una matriz o una vista de esta. Las funciones para operar con este tipo son:

- *Matrix<E> of(E[] datos, Integer nf, Integer nc):* Una nueva matriz definida por un array de tamaño nf x nc.
- *E get(int i, int j):* El valor de la casilla i, j
- *void set(int i, int j, E value):* Actualización de la casilla i, j con el valor value.
- *View4<Matrix<E>> views4():* Las cuatro vistas de una matriz

En primer lugar diseñamos la suma de las dos matrices cuadradas a, b.

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} * \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix} = \begin{pmatrix} a1 + a2 & b1 + b2 \\ c1 + c2 & d1 + d2 \end{pmatrix}$$

Un primer algoritmo iterativo y otro recursivo son:

```
Matrix<E> add(Matrix<E> m2){
     Matrix<E> r = Matrix.zero(this.nf,m2.nc);
       for (int i = 0; i < this.nf; i++) {
           for (int j = 0; j < this.nc; j++) {
              E val = this.get(i,j).add(m2.get(i,j));
              r.set(i,j,val);
           }
       }
       return r;
}
```

```
Matrix<E> add_r(Matrix<E> m2){
	Matrix<E> r;
	if(this.nc > 1 && this.nf > 1) {
		View4<Matrix<E>> v1 = this.views();
		View4<Matrix<E>> v2 = m2.views();

		Matrix<E> a = v1.a.add_r(v2.a);
		Matrix<E> b = v1.b.add_r(v2.b);
		Matrix<E> c = v1.c.add_r(v2.c);
		Matrix<E> d = v1.d.add_r(v2.d);
		r = Matrix.compose(a, b, c, d);
	} else {
		r = this.add(m2);
	}
	return r;
}
```

Como podemos comprobar ambos algoritmos tienen la complejidad $\Theta(n^2)$. El recursivo tiene asociada la ecuación de recurrencia y la solución:

$$T(n) = 4T(n/2) + 1, \qquad \Theta(n^2)$$

4. *Obtener la multiplicación de dos matrices:*

- De forma iterativa
- De forma recursiva dividiéndolas en cuatro submatrices según se indica

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} * \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix} = \begin{pmatrix} a1 * a2 + b1 * c2 & a1 * b2 + b1 * d2 \\ c1 * a2 + d1 * d2 & c1 * b2 + d1 * d2 \end{pmatrix}$$

Una versión iterativa está en el algoritmo:

```
Matrix<E> multiply(Matrix<E> in2){
       Matrix<E> r = Matrix.zero(this.field,this.nf,in2.nc);
       for (int i = 0; i < this.nf; i++) {
           for (int j = 0; j < in2.nc; j++) {
               r.set(i,j,this.field.getZero());
               for (int k = 0; k < this.nc; k++){
                  E val= (this.get(i,k).multiply(in2.get(k,j)))
                           .add(r.get(i,j));
                   r.set(i,j,val);
               }
           }
       }
       return r;
}
```

Como podemos comprobar el algoritmo es de complejidad $\Theta(n^3)$. Una versión recursiva, con las ideas indicadas es:

```
Matrix<E> multiply_r(Matrix<E> m2){
   Matrix<E> r;
   if(this.nc < 2 || this.nf < 2 || m2.nf < 2 || m2.nc < 2) {
       r = this.multiply(m2);
   } else {
           View4<Matrix<E>> v1 = this.views();
           View4<Matrix<E>> v2 = m2.views();
           Matrix<E> a = v1.a.multiply_r(v2.a)
                    .add(v1.b.multiply_r(v2.c));
           Matrix<E> b =
                v1.a.multiply_r(v2.b)
                    .add(v1.b.multiply_r(v2.d));
           Matrix<E> c =
                v1.c.multiply_r(v2.a)
                    .add(v1.d.multiply_r(v2.c));
           Matrix<E> d =
                v1.c.multiply_r(v2.b)
                    .add(v1.d.multiply_r(v2.d));
           r = Matrix.compose(a, b, c, d);
   }
   return r;
}
```

Como podemos que también la multiplicación iterativa la recursiva tienen complejidad a $\Theta(n^3)$:

$$T(n) = 8T(n/2) + n^2, \qquad \Theta(n^3)$$

Esta complejidad puede reducirse aún más reordenado el cálculo en la forma:

$$p1 = a(f-h), p2 = (a+b)h, p3 = (c+d)e, p4 = d(g-e)$$

$$p5 = (a+d)(e+h), p6 = (b-d)(g+h), p7 = (a-c)(e+f)$$

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} * \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix} = \begin{pmatrix} p5+p4-p2+p6 & p1+p2 \\ p3+p4 & p1+p5-p3-p7 \end{pmatrix}$$

Que como podemos comprobar tendrá asociada la ecuación de recurrencia

$$T(n) = 7T(n/2) + n^2, \ \Theta\left(n^{\log_2 7}\right) = \ \Theta(n^{2.8})$$

---

5. *Dada una matriz comprobar si todos sus elementos cumplen una propiedad dada*

Este problema podemos resolver recorriendo todos los elementos de la matriz y comprobando si cumplen la propiedad

```
Boolean allElements(Matrix<E> m, Predicate<E> pd) {
       return Stream2.allPairs(m.nf(),m.nc())
              .map(p->m.get(p))
              .allMatch(pd);
}
```

Igualmente se puede comprobar si algún elemento cumple la propiedad o ninguno la cumple.

6. *Dada una matriz de unos y ceros encontrar la mayor submatriz compuesta solo de unos*

0	1	1	0	1
1	1	0	1	0
0	1	1	1	0
1	1	1	1	0
1	1	1	1	1
0	0	0	0	0

Ilustración 21: Matriz binaria

Una manera de comenzar a enfocar el problema es recorrer todas las submatrices posibles, filtrar la que cumplan la propiedad y quedarse con la de mayor área. El código es:

```
Stream<Matrix<E>> allSubMatrix(Matrix<E> m){
       return Stream2.allQuartets(0,m.nf(),0,m.nc(),
              1,m.nf(),1,m.nc())
        .filter(q->q.third()<=m.nf()-q.first() &&
              q.fourth()<=m.nc()-q.second())
        .map(q->m.view(q.first(),q.second(),
              q.third(),q.fourth()));
}

Matrix<E> largeRectangle1(Matrix<E> m, Predicate<E> pd){
       return allSubMatrix(m)
              .filter(mx->allElements(mx,pd))
              .max(Comparator.comparing(my->my.area()))
              .get();
}
```

El algoritmo anterior tiene complejidad, como puede comprobarse, de $\Theta(n^6)$. Esto es debido a que para generar las submatrices debemos recorrer todas las combinaciones de (f, c, nf, nc), $\Theta(n^4)$ y para cada matriz comprobar que todos sus elementos cumpen la propiedad, $\Theta(n^2)$.

Una posibilidad de reducir la complejidad es mediante un esquema recursivo. Para escoger el algoritmo adecuado debemos decidir si dividir la matriz en 2 o en 4. Estos algoritmos serán de la forma:

```
Matrix<E> largeRectangle2(Matrix<E> m, Predicate<E> pd){
	Matrix<E> r;
	if(m.nf()>2 && m.nc()>2) {
		View2<Matrix<E>> vx = m.views2F();
		Matrix<E> ma = largeRectangle2(vx.left(),pd);
		Matrix<E> mb = largeRectangle2(vx.right(),pd);
		Matrix<E> ct = center2(m,pd);
		r = List.of(ma,mb,ct).stream()
			.max(Comparator.comparing(my->my.area()))
			.get();
	} else {
		r = largeRectangle1(m,pd);
	}
	return r;
}
```

```
Matrix<E> largeRectangle4(Matrix<E> m, Predicate<E> pd){
	Matrix<E> r;
	if(m.nf()>2 && m.nc()>2) {
		View4<Matrix<E>> vx = m.views4();
		Matrix<E> ma = largeRectangle4(vx.a(),pd);
		Matrix<E> mb = largeRectangle4(vx.b(),pd);
		Matrix<E> mc = largeRectangle4(vx.c(),pd);
		Matrix<E> md = largeRectangle4(vx.d(),pd);
		Matrix<E> ct = center4(m,pd);
		r = List.of(ma,mb,mc,md,ct).stream()
			.max(Comparator.comparing(my->my.area()))
			.get();
	} else {
		r = largeRectangle1(m,pd);
	}
	return r;
}
```

Para calcular la complejidad de este tipo de algoritmos recordemos que las ecuaciones de recurrencia del tipo

$$T(n) = aT(n/b) + \Theta(n^d)log^p n$$

Solución

$$T(n) = \begin{cases} \Theta\left(n^{log_b a}\right) & si \quad a > b^d \\ \Theta\left(n^d \, log^{p+1} n\right) & si \quad a = b^d \\ \Theta\left(n^d \, log^p n\right) & si \quad a < b^d \end{cases}$$

Concretado para $p = 0, a = 4, b = 2$ las complejidades en función de d son:

$$T(n) = \begin{cases} \Theta(n^2) & si \;\; d < 2 \\ \Theta(n^2 \; log\, n) & si \;\; d = 2 \\ \Theta(n^d) & si \;\; d > 2 \end{cases}$$

Concretado para $p = 0, a = 2, b = 2$ las complejidades son en función de d:

$$T(n) = \begin{cases} \Theta(n) & si \;\; d < 1 \\ \Theta(n \; log\, n) & si \;\; d = 1 \\ \Theta(n^d) & si \;\; d > 1 \end{cases}$$

La complejidad depende, por tanto, de la complejidad de la función *center* (2 o 4). Observamos que si *center* tiene complejidad $\Theta(n^d)$ el algoritmo diseñado $a = 2$, dividir en dos partes por filas o columnas, da complejidades mejores que para $a = 4$, dividir el cuatro partes. Escogemos dividir en dos partes. Buscamos para *center* un algoritmo de complejidad $\Theta(n)$. De esta forma conseguimos que nestro algoritmo de búsqueda del mayor bloque tenga una complejidad de $\Theta(n \; log\, n)$ más baja que $\Theta(n^6)$ del algoritmo original. Si para center solo comseguimos $\Theta(n^2)$ el algoritmo quedaría con complejidad $\Theta(n^2)$ que sigue siendo mejor que $\Theta(n^6)$.

El algoritmo *center2* tiene como objetivo buscar la mayor matriz de unos que incluya casillas de las dos subpartes en las que dividimos la matriz original. Esto se puede conseguir en dos pasos que dejamos como ejercicio aunque la solución se puede encontrar en el repositorio:

- Encontrar una lista de pares de enteros que delimiten los bloques de unos en la fila central (o en la columna central si dividimos por columnas)
- Por cada bloque de unos en la fila central encontrar la matriz de área mayor que incluyendo solo unos sea un subconjunto del bloque dado. La solución recuerda a la propuesta para el problema del histograma anterior

- Encontrar la mayor de las matrices asociadas a cada bloque.

Si dividiéramos por cuatro el problema sería similar pero ahora tendríamos que recorrer la fila y la columna central para buscar los bloques de uno y luego la mayor matriz. Es debido a que tendríamos que buscar la mayor de todas las matrices que se solapan entre las cuatro partes.

Dada una matriz de ceros y unos encontrar una submatriz rectangular número de filas y columnas mayor que 1, si la hay, que tenga todas sus esquinas iguales a 1

```
mat[][] =   { 1 0 0 1 0
              0 0 1 0 1
              0 0 0 1 0
              1 0 1 0 1
            }
```

Podemos abordar una solución enumerando las submatrices y comprobando si hay alguna que cumple la propiedad. Esto como dijimos antes se puede conseguir con complejidad $\Theta(n^4)$. El código sería:

```
Matrix<E> conPropiedad(Matriz<E> mt, Predicate<Matri<E)> pd){
       return mt.allSubMatrix()
              .filter(m->m.nf()>1 && m.nc()>1)
              .filter(pd)
              .findFirst()
              .get();
}
```

En este caso la propiedad sería

```
Predicate<Matri<E)> pd = m->m.corners().stream()
       .allMatch(e->e.equals(1))
```

Una solución recursiva se puede conseguir con el mismo esquema que en el problema anterior. Pero ahora la función center, en el caso peor, tendría complejidad $\Theta(n^4)$ con lo que no ganaríamos nada con la versión recursiva.

La función center, en este caso, necesitaría enumerar los rectángulos que se solapen con las partes de la matriz y que tengan la propiedad pedida. Enumerar los rectángulos se puede hacer enumerando los posibles vértices superior izquierdo e inferior derecho.

7. Decidir si una matriz cumple una propiedad recursiva

Dado una matriz de n x n números enteros (siendo n una potencia de 2), decidir si cumple que el valor en la casilla superior izquierda es menor al de la casilla inferior derecha y cada uno de los cuatro submatrices cumplen la propiedad.

Esta es una propiedad recursiva de una matriz. La forma de codificarla es:

```
Boolean cumplePropiedadRecursiva(Matrix<Integer> m) {
	Boolean r;
	if(m.nf()>2 && m.nc()>2) {
		View4<Matrix<Integer>> vx = m.views();
		r =    cumplePropiedadRecursiva(vx.a()) &&
			cumplePropiedadRecursiva(vx.b()) &&
			cumplePropiedadRecursiva(vx.c()) &&
			cumplePropiedadRecursiva(vx.d()) &&
			m.get(0,0) < m.get(m.nf()-1,m.nc()-1);
	} else {
		r =  m.get(0,0) < m.get(m.nf()-1,m.nc()-1);
	}
	return r;
}
```

En este caso es importante usar el operador *&&* para combinar las llamadas recursivas. De esa forma si una llamada recursiva devuelve false ya no se hará la llamada siguiente. Podemos ver que la complejidad en el caso peor es $\Theta(n^2)$

De algoritmos recursivos múltiples a iterativos

Diseñar un algoritmo recursivo, con y sin memoria, y posteriormente un algoritmo iterativo que calcule los valores de la recurrencia $f_n = 2f_{n-1} + 3f_{n-2} - f_{n-3}, f_2 = 1, f_1 = 1, f_0 = 2$ mediante el cálculo de abajo arriba de los valores de la recurrencia.

El esquema anterior se convierte en el algoritmo recursivo:

```
E rec1(Integer n){
E r;
       if (n==0) r = 2;
       else if (n==1) r = 1;
       else if (n==2) r = 1;
       else r = 2*rec1(n-1)+3*rec1(n-2)-rec1(n-3);
       return r;
}
```

En el algoritmo anterior hemos de concretar el tipo *E* que puede ser *Integer, Long* o *BigInteger*. Para un valor de *n* igual a 30 o superior debemos usar *BigInteger* para el tipo *E*. Asumiendo esto, en el cálculo de la complejidad hemos de tener en cuenta que el producto de dos enteros largos es $\Theta(d^{1.6})$ si se usa el algortimo de Karatsuba y la suma $\Theta(d)$. En ambos casos d es el número de dígitos. Por otra parte el número de dígitos

de un entero n es $\Theta(\log n)$. Por ello la complejidad de la suma y el producto en función de n es $\Theta(log\ n), \Theta((log\ n)^{1.6})$.

Con las consideraciones anteriores la complejidad del algoritmo anterior sin memoria tiene asociada la ecuación de recurrencia:

$$T(n) = T(n-1) + T(n-2) + T(n-3) + log\ n$$

Que podemos acotar por las ecuaciones:

$$T(n) = 3T(n-1) + log\ n\,,\ \ T(n) = 3T(n-3) + log\ n$$

Que tienen soluciones de $\Theta(3^n\ log\ n)$ y $\Theta\left(3^{\frac{n}{3}}\ log\ n\right) = \Theta(\sqrt[3]{3}^n)$ respectivamente.

La versión con memoria:

```
E rec2(Integer n){
       return rec3(n,new Hashmap<>());
}
E rec2(Integer n, Map<Integer,E> m){
       E r;
       if(m.containsKey(n) r = m.get(n);
       else if (n==0) r = 2;
       else if (n==1) r = 1;
       else if (n==2) r = 1;
       else r = 2*rec2(n-1,m)+3*rec2(n-2,m)-rec2(n-3,m);
       return r;
}
```

Cuya complejidad podemos expresar mediante el sumatorio:

$$T(n) = \sum_{i \in pa(0,1)}^{n} log\ i \simeq\ \Theta(n\ log\ n)$$

El primer algoritmo sin memoria es exponencial y no se puede usar por encima de $n \cong 40$. El algortimo con memoria tiene una complejidad asumible pero la cantidad de memoria necesaria lo hace inutilizable para valores $n \cong 600$ o superiores.

```
E rec(Integer n){
        Integer i;
        E a,b,c;
        (i,a,b,c) = (0,1,1,2);
        while(i < n){
                (i,a,b,c) = (i+1,2*a+3*b-c,a,b)
        }
        return c;
}
```

Cuya complejidad podemos expresar igualmente mediante el sumatorio:

$$T(n) = \sum_{i \in pa(0,1)}^{n} log\, i \simeq \Theta(n\ log\, n)$$

Una forma alternativa de enfocar el problema es reducirlo a un problema de potencia entera en la forma:

$$\begin{pmatrix} f_n \\ f_{n-1} \\ f_{n-2} \end{pmatrix} = \begin{pmatrix} 2 & 3 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \\ f_{n-3} \end{pmatrix}, \qquad \begin{pmatrix} f_2 \\ f_1 \\ f_0 \end{pmatrix} = \begin{pmatrix} d_2 \\ d_1 \\ d_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}$$

Siendo la matriz $M = \begin{pmatrix} 2 & 3 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ y $d = \begin{pmatrix} d_2 \\ d_1 \\ d_0 \end{pmatrix}$ la solución del problema es:

$$\begin{pmatrix} f_n \\ f_{n-1} \\ f_{n-2} \end{pmatrix} = M^n \begin{pmatrix} d_2 \\ d_1 \\ d_0 \end{pmatrix}$$

1. *Cálculo de un número combinatorio*

Dar una definición recursiva y obtener un algoritmo recursivo para el cálculo de un número combinatorio basándose en las propiedades

$$\begin{cases} \binom{n}{0} = \binom{n}{n} = 1, & n \geq 0 \\ \binom{n}{1} = \binom{n}{n-1} = n, & n \geq 1 \\ \binom{n}{k} = \binom{n}{n-k}, & n \geq k \\ \binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}, & n > k \end{cases}$$

La solución es de la forma:

$$\binom{n}{k} = \begin{cases} 1, & k, n-k = 0 \\ n, & k, n-k = 1 \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & k > 1 \end{cases}$$

Que tiene el código asociado:

```
E binom1(n,k) {
        E r;
        if(k == 0 || n-k == 0) r = 1;
        else if(k==1 || n-k == 1) r = n;
        else r = binom1(n-1,k-1)+binom1(n-1,k);
        return r;
}
```

Y la solución con memoria:

```
E binom2(n,k) {
        return binom2(n,k,new Hashmap<>());
}
E binom2(n,k,Map<(Int,Int),E> m){
        E r;
        if(m.containsKey((n,k)) r = m.get((n,k));
        else if(k == 0 || n-k == 0) r = 1;
        else if(k==1 || n-k == 1) r = n;
        else r = binom1(n-1,k-1)+binom1(n-1,k);
        return r;
}
```

Igualmente la solución iterativa que ya se ha visto anteriormente la volvemos a incluir aquí:

```
record Bn(Long a, Long b) {
	public static Bn of(Long a, Long b) {
		return new Bn(a,b);
	}
}

BigInteger binom(Long n, Long k) {
	Map<Bn,BigInteger> m = new HashMap<>();
	for(Long i=0L; i<=n; i++) {
		for(Long j=0L; j<=i;j++) {
			if(j==0)
				m.put(Bn.of(i,0L),BigInteger.ONE);
			else if(j==1 || i-j ==1)
				m.put(Bn.of(i,j),
					new BigInteger(i.toString()));
			else {
				BigInteger r =
				   m.get(Bn.of(i-1, j-1))
				       .add(m.get(Bn.of(i-1, j-1)));
				m.put(Bn.of(i,j),r);
			}
			m.remove(Bn.of(i-2,k));
		}
	}
	return m.get(Bn.of(n, k));
}
```

Asumiendo que para valores grandes de *n*, tomado como *n* el tamaño del problema, el tipo *E* debe ser *BigInteger* las complejidades de sin memoria es:

$$T(n) = 2T(n-1) + \log n\,, \Theta(2^n\ \log n)$$

Y la complejidad del algoritmo con memoria y el iterativo:

$$\sum_{i \in pa}^{n} \sum_{j \in pa}^{i} log\ i = \Theta(n^2\ log\ n)$$

Problemas de implementación de tipos

En esta sección se proponen implementaciones para tipos conocidos y ejercicios para ser resueltos

Implementación de listas

Las listas tienen dos implementaciones usuales: ArrayList y LinkedList. Los tipos siguientes, Alist y Llist, hacen este trabajo.

Recordemos que un tipo se implementa mediante otro que tiene unas propiedades privadas, un invariante, unas operaciones y unos métodos de factoría. El tipo implementador ofrece como públicas las propiedades y operaciones que requiere el tipo a implementar.

```
AList<E>: Mutable, Implementa List<E>
       Propiedades Privadas
              E[] data;
              int capacity;
              int size; // número de elementos
       Invariante:
              tam = data.lenght,
              size < tam,
       Operaciones privadas:
              E[] grow(E[] old, int newTam);
```

En esta implementación los datos se guardan en un array que está parcialmente lleno. El número de datos es *size*. El primer detalle para tener en cuenta es que el tamaño del array de datos (data) debe ser suficiente para almacenar los datos. Es decir, *size < tam.* Esta restricción del invariante se puede mantener llamando a la operación *grow* al comienzo de la operación *add.*

Con la implementación anterior se propone la implementación de las siguientes propiedades, operaciones y métodos de factoría del tipo *List<E>* discutiendo en cada caso la complejidad, las precondiciones y postcondiciones de cada uno de ellos.

- int size()
- Boolean contains(E e)
- E get(int i)
- boolean add(E e)
- boolean remove(E e)
- int indexOf(E e)
- List<E> subList(int i, in j)

Como ejemplo veamos los métodos *grow* y *add*.

```
private AList(int capacity) {
	this.capacity = capacity;
	this.size = 0;
	this.data = null;
}
private void grow(){
	if(size==capacity){
		E[] oldElements = data;
		capacity = capacity*2;
		data = Arrays.copyOf(oldElements, capacity);
	}
}
```

```
public boolean add(E e) {
	if(this.data == null) this.data =
		(E[]) Array.newInstance(e.getClass(), capacity);
	grow();
	data[size] = e;
	size++;
	return true;
}
```

En el repositorio se puede encontrar el código del resto de las operaciones.

El tipo *LList* es la implementación de una lista enlazada. Como se ve abajo, también hay que mantener el invariante, *size < tam,* de forma similar a antes. Ahora, sin embargo, los datos no están indexados. Están enlazados en las distintas entradas del tipo *Entry<E>.* Ahora las complejidades de las operaciones básicas y del cálculo de las propiedades son diferentes. También es más compleja la implementación de *sublist*.

```
LList<E>: Mutable, Implementa List<E>
	Propiedades Privadas
		Entry<E> first;
		Entry<E> last;
		int size;
	Invariante:
		…
	Propiedades privadas:
		Entry<E> entryInPos(int index);
	Tipos privados:
		Entry<K>
		Propiedades
			K key();
			Entry<K> next();
		Métodos de creación:
		Entry<K> of(K key, Entry<K>  next);
```

Como ejercicio se propone implementar las propiedades, operaciones y métodos de factoría de List<E>. Veamos como ejemplo algunos métodos privados y otros públicos.

Algunas propiedades privadas son importantes como *entryInPos* que encuentra la tupla en la posición index o devuelve *null* si no la encuentra. Podemos ver que esta operación tiene en el caso peor complejidad del orden $\Theta(n)$.

```
boolean add(E e){
	Entry<E> e1 = new Entry<E>(e);
	if(last==null){
		first = e1;
		last=e1;
	}else{
		last.setNext(e1);
		last = e1;
	}
		size++;
	return true;
}

private Entry<E> entryInPos(int index){
	Preconditions.checkElementIndex(index, size);
	Entry<E> pe = first;
	for(int p = 0 ; p < index; p++){
		pe = pe.next();
	}
	return pe;
}
```

```
public boolean add(int index, E e){
	Preconditions.checkPositionIndex(index, size);
	Entry<E> ne = new Entry<E>(e);
	if(index==size){
		add(e);
	} else if(index==0){
		ne.setNext(first);
		first = ne;
	} else {
		Entry<E> pe = entryInPos(index-1);
		ne.setNext(pe.next());
		pe.setNext(ne);
	}
	size++;
	return true;
}
```

Tablas hash

El tipo *HashTable* es el adecuado para implementar el tipo *Map<K,V>, Set<E>, Multiset<E>,* etc.

```
HashTable<K,V>: Mutable, Implementa Map<K,V>
       Propiedades Privadas:
              int groupsNumber;  //número de grupos
              int capacity;   //tamaño del array de datos
              int size;  //número de pares clave valor
              AList<Integer> groups;  //array de entradas a
                                      //grupos
              AList<EntryData<K,V>> data; //array de datos
              Integer firstFreeData; // primera tupla libre
              Integer group(K key); // grupo al que pertenece
                                    // la clave key
              EntryData<K,V> findEntry(K key);
                           //tupla con la clave key o null si no
                            //hay ninguna
       Invariante:
              size/capacity < loadFactorLimit;
       Operaciones privadas:
              void  rehash(int newCapacity);
       Tipos privados:
              EntryData<K,V>
              Propiedades:
                     K key();
                     V value();
                     Integer next();
              Métodos de creación:
                     EntryData<K,V> of(
                            K key,V value,Integer next);
       Métodos de Creación:
              HashTable<K,V> empty();
       Propiedades:
              …
       Operaciones
              …
```

El invariante es el siguiente:

- El grupo *i* está formado por la lista enlazada de tuplas (EntryData<K,V>) guardadas en *data* que comienza en *groups.get(i).*
- Un par (key,value) está en el grupo *i = key.hashCode()%groupsNumber* o no está.
- La primera casilla libre de *data* está en la posición *firstFreeData*.

- Todas las casillas libres en data están enlazadas entre sí.
- Debe cumplirse que *size/capacity < loadFactorLimit*. La operación privada *rehash* está diseñada para mantener esta relación si se intenta sobrepasar el factor de carga.
- Cada tipo que actúe como clave debe tener un hashCode y un equals.

El tipo usa una tupla de clave, valor y entero que denominamos *EntryData* y dos listas implementadas mediante arrays: una de enteros (*groups*) y otra de *EntryData (data).*

Para implementar la búsqueda del grupo asociado a un par clave valor definimos los métodos privados:

- *Integer group(K key):* Calcula el grupo asociado a la clave como el resto de la división entre el *hashCode* de la clave y el número de grupos.
- *EntryData<K,V> findEntry(K key):* Busca dentro del grupo asociado a la clave, que como sabemos es una lista enlazada de *EntryData* la tupla con la clave igual a *key*. Devuelve *null* si no la encuentra.

El método *get(K key)* se implementa fácilmente usando los dos métodos privados anteriores. El método *put(K key, V value)* se implementa con las siguientes ideas:

- Se busca una EntryData que tenga una clave igual a key usando findEntry.
    - Si se encuentra se modifica el valor en esta tupla y se devuelve el antiguo valor
    - Si no se encuentra se busca primera EntryData en la lista de la que están vacías (que comienza en firstFreeData) y sus campos se actualizan con los valores de key, value. Posteriormente se actualiza firstFreeData a la siguiente tupla libre y se inserta al principio de la lista enlazada asociada al grupo group(key).

Como ejercicio se propone implementar las propiedades, operaciones y métodos de factoría de Map<K,V>.

El tipo *Set<E>* se implementa también mediante un pequeño ajuste al tipo HashTable<E,V>. Ahora no hace falta el tipo V y el tipo EntryData<E,V> se sustituye por EntryData<E> con propiedades key, next.

Como ejercicio se propone implementar las propiedades, operaciones y métodos de factoría de Set<E>.

El tipo Multiset<E> se implementa teniendo en cuenta que es equivalente a Map<E,Integer>.

Como ejercicio se propone implementar las propiedades, operaciones y métodos de factoría de Set<E>.

Conjuntos de enteros: IntegerSet

El tipo IntegerSet es una implementación de Set<Integer> donde los valores enteros están por encima de un valor dado. El tipo es muy interesante porque ofrece operaciones con conjuntos de complejidad $\Theta(1)$. Para su implementación se usa un array de bits ya proporcionado por Java,

El **array de bits** es un tipo, que ya ofrece Java como *BitSet*, muy adecuado para implementar conjuntos de enteros dentro de un rango.

```
public class IntegerSet implements Set<Integer> {
       private final Integer infLimit;
       private final BitSet bits;
       …
}
```

La idea es reducir las operaciones sobre conjuntos (contains, unión, intersección, diferencia simétrica, complemento, …) a otras operaciones con bits (getBit, or, and, xor, complement, …).

Se propone implementar las propiedades, operaciones y métodos de factoría de *IntegerSet* que implementan las de *Set<E>.* Como ejemplo la operación add es:

```
@Override
public boolean add(Integer e) {
	Preconditions.checkArgument(e>=infLimit,
		"Fuera de rango");
	Integer ne = e-infLimit;
	Boolean c = this.bits.get(ne);
	this.bits.set(ne, true);
	return c != this.bits.get(ne);
}
```

Implementación de árboles

El tipo **Tree<E>** se puede implementar mediante *TreeImpl* que mantiene un conjunto de propiedades privadas adecuadas. Se propone como ejercicio implementar las operaciones básicas del tipo Tree<E>. La implementación puede verse en el repositorio.

```
TreeImpl<E>: Inmutable, Implementa Tree<E>
	Propiedades Privadas
		TreeType type;
		E label;
		List< TreeImpl<E>> children;
		TreeImpl<E> father;
	Invariante:
		(type == Nary && label !=null && children!=null)
		(type == Leaf && label !=null) ||
			type == Empty;
	Métodos de creación:
	…
	Propiedades:
		…
```

De la misma manera el tipo *BinaryTree<E>* se puede implementar mediante *BinaryTreeImpl*. Se propone como ejercicio implementar las operaciones básicas del tipo *BinaryTree<E>*. La implementación puede verse en el repositorio.

```
BinaryTreeImpl<E>: Inmutable, Implementa BinaryTree<E>

Propiedades Privadas
             BinaryType type;
             E label;
             BinaryTreeImpl<E> left;
       BinaryTreeImpl<E> right;
             BinaryTreeImpl father;
       Invariante:
       (type == Binary && label !=null && left!=null &&
                    right!=null) ||
              (type == Leaf && label !=null) ||
                    type == Empty;
Métodos de creación:
       …
       Propiedades:
             …
```

Para implementar conjuntos ordenados se suele usar un *AVLTree*. Este tipo se implementa reutilizando un *BinaryTree* que se mantiene equilibrado y ordenado. Se propone como ejercicio implementar las operaciones del tipo *SortedSet<E>.*

```
AVLTree<E>: Mutable, Implementa SortedSet<E>.
       Propiedades privadas
             BinaryTree<E> tree;
             Comparator<E> comparator;
       Invariante: tree está ordenado con respecto a comparator
             y equilibrado
       Operaciones privadas:
             BinaryTree<E> addOrdered(BinaryTee<E>
                    tree,Comparator<E> cmp);
             BinaryTree<E> removeOrdered(BinaryTee<E>
                    tree,Comparator<E> cmp);
             BinaryTree<E> equilibrate(BinaryTee<E> tree);
       Métodos de creación:
             …
       Propiedades:
             …
       Operaciones:
       …
```

Problemas de árboles y expresiones

Los problemas sobre árboles son de tres tipos fundamentales:

- Problemas iterativos sin tener en cuenta el nivel de cada subárbol. Estos se resuelven usando el stream, o el iterador, proporcionado por el árbol. Los algoritmos resultantes son similares a los algoritmos iterativos sobre un agregado de datos.
- Problemas iterativos que tienen en cuenta el nivel de cada subárbol. Estos se resuelven usando el stream *byLevel(),* o el iterador equivalente, proporcionado por el árbol. Los algoritmos resultantes son similares a los algoritmos iterativos sobre un agregado de datos pero donde cada elemento incorpora su nivel dentro del árbol
- Problemas recursivos. Todos los problemas de árboles pueden ser resueltos recursivamente. Aunque en muchos casos es más fácil y recomendable usar algoritmos iterativos hay otros algoritmos más fáciles desde el punto de vista recursivo.

1. Calcular el número de etiquetas del árbol (tamaño del árbol)

La versión iterativa

```
<E> Integer size(BinaryTree<E> tree) {
       return (int) tree.stream().filter(t->!t.isEmpty())
              .mapToInt(t->1).count();
}
```

La recursiva:

```
<E> Integer size(BinaryTree<E> tree) {
       return switch (tree) {
       case BEmpty<E>  t-> 0;
       case BLeaf<E>  t -> 1;
       case BTree<E> t -> 1 + size(t.left()) + size(t.right());
       };
}
```

2. Sumar las etiquetas de un árbol que cumplen un predicado

```
Integer sumIfPredicate(BinaryTree<Integer> tree,
              Predicate<Integer> predicate) {
       return switch (tree) {
       case BEmpty<Integer> t -> 0;
       case BLeaf<Integer> t ->
              predicate.test(t.label()) ? t.label() : 0;
       case BTree<Integer> t ->
              predicate.test(t.label()) ? t.label()
                     : 0 +
              sumIfPredicate(t.left(), predicate) +
              sumIfPredicate(t.right(), predicate);
       };
}
```

Y al versión iterativa:

```
Integer sumIfPredicate2(BinaryTree<Integer> tree,
            Predicate<Integer> predicate) {
      return tree.byDeph()
      .map(t -> t.optionalLabel().orElse(0))
      .filter(predicate)
      .mapToInt(lb -> lb).sum();
}
```

3. Añadir una etiqueta a un árbol binario inmutable ordenado

```
<E> BinaryTree<E> addOrdered(BinaryTree<E> tree, E element,
            Comparator<E> cmp) {
      BinaryTree<E> s = switch (tree) {
      case BEmpty<E> t -> BinaryTree.leaf(element);
      case BLeaf<E> t && cmp.compare(element, t.label()) == 0
            -> tree;
      case BLeaf<E> t && cmp.compare(element, t.label()) < 0
            -> BinaryTree.binary(t.label(),
            BinaryTree.leaf(element), BinaryTree.empty());
      case BLeaf<E> t && cmp.compare(element, t.label()) > 0
            -> BinaryTree.binary(t.label(),
            BinaryTree.empty(), BinaryTree.leaf(element));
```

Y los casos recursivos:

```
case BTree<E> t && cmp.compare(element, t.label()) == 0 ->
      tree;
case BTree<E> t && cmp.compare(element, t.label()) < 0 -> {
      BinaryTree<E> new_left = addOrdered(t.left(),
            element, cmp);
      BinaryTree<E> r = tree;
      if (t.left() != new_left) {
            r = BinaryTree.binary(t.label(), new_left,
                  t.right());
            r = r.equilibrate();
      }
      yield r;
}
```

```
case BTree<E> t && cmp.compare(element, t.label()) > 0 -> {
       BinaryTree<E> new_right = addOrdered(t.right(), element,
              cmp);
       BinaryTree<E> r = tree;
       if (t.right() != new_right) {
         r = BinaryTree.binary(t.label(), t.left(), new_right);
         r = r.equilibrate();
       }
       yield r;
}
};
return s;
}
```

4. *Diseñe un algoritmo que dado un árbol binario de enteros devuelva cierto en caso de que para cada nodo que tenga 2 hijos no vacíos se cumpla que su etiqueta es igual a la suma de las etiquetas de las raíces de sus 2 hijos.*

Una versión iterativa sería

```
Boolean sumaEtiquetas(BinaryTree<Integer> tree) {
       return tree.byDeph()
       .filter(e->e instanceof BTree<Integer> t &&
              !t.left().isEmpty() &&
              !t.right().isEmpty())
       .allMatch(e->e instanceof BTree<Integer> t &&
              t.label().equals(t.left().optionalLabel().get()+
                           t.right().optionalLabel().get()));
}
```

5. *Implemente una función booleana que, dados un árbol binario de caracteres y una lista de caracteres, determine si existe un camino en el árbol de la raíz a una hoja no vacía que sea igual a la lista.*

```
Boolean existeLista(BinaryTree<Character> tree,
              List<Character> ls) {
       Integer n = ls.size();
       return switch (tree) {
       case BEmpty<Character> t && ls.isEmpty() -> true;
       case BLeaf<Character> t && !ls.isEmpty() ->
              ls.get(0).equals(t.label());
       case BTree<Character> t && !ls.isEmpty() ->
              ls.get(0).equals(t.label())&&
                     (existeLista(t.left(), ls.subList(1, n)) ||
                     existeLista(t.right(), ls.subList(1, n)));
       };
}
```

6. *Buscar lista de propiedades en un árbol*

Diseñe un algoritmo que dado un árbol n-ario Tree<E> y un predicado sobre Tree<E> devuelva una lista List<Boolean> de forma que el elemento i-ésimo de la lista será true si todos los árboles del nivel i cumplen el predicado.

Una versión iterativa usando el iterador por niveles es

```
<E>List<Boolean> niveles(Tree<E> tree,Predicate<Tree<E>> pd) {
       Map<Integer,Boolean> m = tree.byLevel()
       .collect(Collectors.groupingBy(p->p.level(),
              Collectors.collectingAndThen(
                     Collectors.mapping(p->p.tree(),
                            Collectors.toList()),ls->ls.stream()
                                   .allMatch(pd))));
       return m.entrySet().stream()
              .sorted(Comparator.comparing(e->e.getKey()))
              .map(e->e.getValue()).toList();
}
```

7. Buscar listas de árboles con una propiedad

Diseñe un algoritmo recursivo que dado un árbol n-ario de tipo genérico devuelva un *Map<Integer, List<Tree<E>>>*. Dicho map en cada entrada tiene como clave el número de hijos de un árbol y como valor una lista con todos los árboles que tienen ese número de hijos.

Una versión iterativa es

```
<E> Map<Integer,List<Tree<E>>> parNumeroDeHijos(Tree<E> tree){
      return tree.stream()
             .collect(Collectors.groupingBy(
                    t->t.numElements ()));
}
```

Problemas de expresiones

Los problemas de expresiones y árboles de sintaxis abstracta pueden verse en el repositorio. Algunos de ellos son:

- Calcular el tipo de una expresión
- Calcular el valor de una expresión
- Generar un fichero .gv a partir de un árbol de sintaxis abstracta
- Determinar errores de tipo, si los hay, en un árbol de sintaxis abstracta
- Generar código en un lenguaje dado a partir de un árbol de sintaxis abstracta

Problemas de Grafos

1. *Dado un grafo obtener una vista con los vértices que cumplen una propiedad y las aristas que cumplen otra propiedad dada.*

Diseñaremos una vista inmutable. Delegaremos los métodos en una variable privada de tipo *Graph<V,E>,* ajustaremos los métodos necesarios y haremos que los métodos modificadores no se puedan llamar:

```
public class SubGraphView<V, E, G extends Graph<V,E>>
              implements Graph<V, E> {

       public static <V, E, G extends Graph<V,E>>
                     SubGraphView<V, E, G> of(
                     G graph, Predicate<V> vertices,
                     Predicate<E> edges) {
              return  new SubGraphView<V, E, G>(graph,
                     vertices, edges);
       }

       private G graph;
       private Predicate<V> vertices;
       private Predicate<E> edges;
       …
```

```
        public Set<E> edgeSet() {
                return graph.edgeSet().stream()
                        .filter(e->this.containsEdge(e))
                        .collect(Collectors.toSet());
        }

        public Set<E> edgesOf(V v) {
                return graph.edgesOf(v).stream()
                        .filter(e->this.containsEdge(e))
                        .collect(Collectors.toSet());
        }
        …
}
```

2. *Dado un grafo con pesos obtener una vista que contenga los mismos vértices y sea completo. Las nuevas aristas tendrán un peso muy grande.*

Usamos la misma técnica de antes

```
public class CompleteGraphView<V, E, G extends Graph<V,E>>
        implements Graph<V,E> {

        private G graph;
        private Supplier<E> edgeWeightFactory;
        private Double weightDefault;

        public E getEdge(V v0, V v1) {
                E edge = null;
                if (graph.containsEdge(v0, v1)) edge =
                        graph.getEdge(v0, v1);
                else edge = this.edgeWeightFactory.get()
                return edge;
        }
        …
```

Aquí se ha optado por construir una nueva arista cada vez que es necesario. Otra posibilidad es añadir las aristas nuevas que se construyen al grafo de partida cuando la necesitemos.

3. *A partir de un grafo no dirigido con pesos obtener una vista que sea un grafo dirigido con los mismos vértices y que por cada arista original tenga dos dirigidas con los mismos pesos.*

Se usa la misma técnica que antes. Se deja como ejercicio.

4. *Una red social imaginaria y pequeña, se representa con un grafo en el que los vértices modelan a los miembros de la red social y las aristas (sin etiqueta ni peso) modelan la amistad entre dos miembros.*

Desarrolle:

- Un método que devuelva un conjunto con aquellos miembros que no tengan ningún amigo.
- Un método que, dados dos miembros, devuelva la lista más corta de amigos que hay desde un miembro a otro miembro.

```
<V> Set<V> sinAmigos(Graph<V,Integer> graph){
      return graph.vertexSet().stream()
             .filter(v->graph.degreeOf(v)==0)
             .collect(Collectors.toSet());
}

<V> List<V> listaMasCorta(Graph<V,Integer> graph, V e1, V e2){
      ShortestPathAlgorithm<V,Integer> a =
             new DijkstraShortestPath<V,Integer>(graph);
      GraphPath<V,Integer> gp =  a.getPath(e1,e2);
      return gp.getVertexList();
}
```

5. *Lectura de grafos*

Dada una clase CV con los atributos: {a: String, b: Integer, c: Boolean}, una segunda clase CE con los siguientes atributos: {s: String, i: Integer} y un fichero datos.txt que contiene la información necesaria para representar

un grafo en el que los vértices sean objetos CV y las aristas objetos CE, implemente un método, y todos los elementos auxiliares necesarios, que permita cargar dicho grafo desde el fichero.

Asumimos que el tipo CV no define una propiedad que defina cada valor de manera unívoca. Cuando esto es así le añadimos un identificador de tipo entero que identifique cada vértice. El fichero es *datos.txt* es de la forma:

```
#VERTEX#
1,A,2,true
2,B,3,false
3,C,10,true
4,D,21,true
5,E,4,false
#EDGE#
1,3,v1,21
1,2,v2,2
2,5,v4,20
4,3,v5,3
2,4,v6,5
```

Los tipos CV, CE los diseñamos como records con un método de factoría que convierte una array de String en un valor.

```
record CV(Integer id, String a, Integer b, Boolean c) {
	public static CV of(String[] formato) {
		return new CV(
			Integer.parseInt(formato[0]),
			formato[1],
			Integer.parseInt(formato[2]),
			toBoolean(formato[3]));
	}
}
```

```
record CE(String s, Integer i) {
	public static CE of(String[] formato) {
			return new CE(formato[2],
				Integer.parseInt(formato[3]));
	}
}
```

```
Graph<CV, CE> leeGrafo(String file) {
	SimpleWeightedGraph<CV, CE> graph =
		GraphsReader.newGraph(file, CV::of,
			CE::of,
			Graphs2::simpleWeightedGraph,
			e->1.0);
	return graph;
}
```

6. *Rutas mínimas*

Una empresa de autobuses tiene andenes en algunas ciudades de un país. El mapa de carreteras de dicho país está modelado como un grafo en el que los vértices están etiquetados con el nombre de la ciudad y las aristas con la distancia entre dos ciudades. Desarrolle un método que permita calcular todas las rutas mínimas entre las diferentes ciudades, junto con su distancia.

Una manera adecuada es usar el algoritmo de *FloydWarshall:*

```
<V,E> Set<GraphPath<V,E>> distanciasMinimas(Graph<V,E> g){
	List<V> vertices = g.vertexSet().stream().toList();
	FloydWarshallShortestPaths<V,E>  a = new
		FloydWarshallShortestPaths<>(g);
	Set<GraphPath<V,E>> s = new HashSet<>();
	for(int i = 0; i<vertices.size();i++) {
		for(int j=i+1; j<vertices.size();j++) {
			s.add(a.getPath(vertices.get(i),
				vertices.get(j)));
		}
	}
	return s;
}
```

7. *Recubrimiento mínimo*

Se desea conectar un conjunto de ciudades mediante líneas telefónicas. La compañía telefónica cobra la cantidad $p(i,j)$ por conectar la ciudad i

con la *j*. ¿Qué ciudades conectar para que todas estén conectadas entre sí al mínimo precio?

El algoritmo de Kruskal nos resuelve el problema. Este algoritmo nos devuelve un *SpanningTree<E>* que no informa del conjunto de aristas elegidas y el peso del recubrimiento:

```
<V,E> SpanningTree<E> recubrimientoMinimo(Graph<V,E> g){
	KruskalMinimumSpanningTree<V,E> a =
		new KruskalMinimumSpanningTree<>(g);
	return a.getSpanningTree();
}
```

8. *Recubrimiento de vértices*

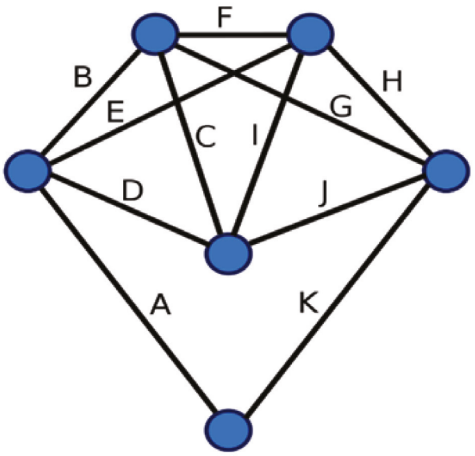
Sea un grafo cuyas aristas representan carreteras y los vértices representan cruces entre ellas. Queremos colocar cámaras de seguridad en algunos cruces de tal manera que puedan ver todas las carreteras que acceden al cruce. ¿Si pretendemos colocar el mínimo número de cámaras, como podemos obtener la solución?

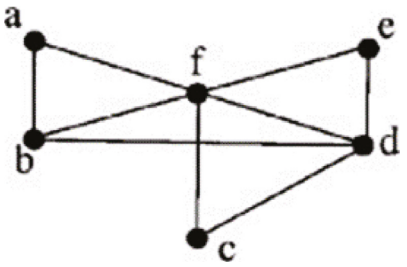
Un algoritmo adecuado es el recubrimiento de vértices. Una implementación de este es *RecursiveExactVCImpl.* Este algoritmo devuelve un *VertexCover<V>* que es un conjunto de vértices y alguna información adicional como el peso del recubrimiento.

```
<V,E> VertexCover<V> recubrimientoVertices(Graph<V,E> g){
	VertexCoverAlgorithm<V> a =
		new RecursiveExactVCImpl<>(g);
	return a.getVertexCover();
}
```

9. Recorrido

Una compañía de autopistas ha contratado a una empresa de seguridad para que patrulle la red de autopistas cuyo mapa está esquematizado en los siguientes grafos:





La empresa de seguridad quiere realizar el servicio con un solo vehículo y quiere determinar la existencia de un recorrido de manera que se vigilen todos los tramos de la autopista una única vez. ¿Cuál es ese recorrido? Dado un grafo no dirigido implementar un algoritmo que nos proporcione una lista de aristas, si existe, con el recorrido a seguir. En cuál de los dos grafos existe el camino pedido.

Un *camino euleriano* es un camino en un grafo que visita cada arista exactamente una vez (lo que permite volver a visitar los vértices). Es decir, en un camino euleriano pueden repetirse los vértices pero no las aristas. Alternativamente un *camino hamiltoniano* es un camino en un grafo que visita cada vértice exactamente una vez. En un camino hamiltoniano no se pueden repetir ni vértices ni aristas.

El problema que se propone es encontrar un camino euleriano para ese grafo. Es conocido que para que exista un camino euleriano todos los vértices deben ser de grado par. Escribir un algoritmo para decidir si un grafo tiene un camino *euleriano* y si lo tiene encontrarlo.

```
<V,E> Boolean esEuleriano(Graph<V,E> g){
	return g.vertexSet().stream()
		.allMatch(v->g.degreeOf(v)%2==0);
}

<V,E> GraphPath<V,E> caminoEuleriano(Graph<V,E> g){
	EulerianCycleAlgorithm<V,E> a =
		new HierholzerEulerianCycle<V,E>();
	return a.getEulerianCycle(g);
}
```

Podemos comprobar que el grafo de la derecha no es euleriano. El de la izquierda si y un camino euleriano está definido por la secuencia de aristas:

```
[K, J, I, H, G, C, D, E, F, B, A]
```

10. Plan de estudios

Un plan de estudios está estructurado de tal forma que existen asignaturas que deben haberse cursado (y aprobado) anteriormente para que un alumno se pueda matricular de una asignatura dada. Así, los prerrequisitos de una asignatura pueden ser 0 o más asignaturas. Implemente un método que, dado el grafo que representa el plan de estudios, con las asignaturas y las restricciones de matriculación entre las mismas, y la lista de asignaturas que un alumno tiene aprobadas, devuelva una lista con las asignaturas que puede cursar el próximo año.

Con la información proporcionada podemos representar el plan de estudios como un grafo dirigido acíclico. Debe ser acíclico porque si hubiera ciclos podríamos tener disponible una asignatura ya cursada. Se trata de encontrar las asignaturas vecinas hacia delante de las proporcionadas en la lista:

```
Set<String> asignaturasPosibles(
       DirectedAcyclicGraph<String,String> g, List<String> as){
       Set<String> r = new HashSet<>();
       as.stream()
              .flatMap(a->g.outgoingEdgesOf(a).stream()
              .map(t->g.getEdgeTarget(t)))
       .filter(a->!as.contains(a))
       .forEach(y->r.add(y));
       return r;
}
```

11. Problema de horarios

Problema de horarios. Se deben impartir varias asignaturas y no se pueden programar a la misma hora dos asignaturas que tienen alumnos matriculados en ambas. En este caso los vértices representan asignaturas y una arista entre dos vértices significa que hay alumnos matriculados en las asignaturas que representan los vértices.

Lo primero que queremos ver es la forma de leer un grafo cuando la información de la arista es irrelevante los vértices son de tipo String:

```
Graph<String,DefaultEdge> leeAlumnosComunes(String file) {
       return GraphsReader.newGraph(file,
              s->s[0],
              s->new DefaultEdge(),
              Graphs2::simpleGraph);
}
```

Disponiendo del grafo este problema se puede resolver mediante coloreado de grafos.

```
<V, E> Map<V, Integer> coloreado(Graph<V, E> g) {
       VertexColoringAlgorithm<V> vca =
              new GreedyColoring<V, E>(g);
       Coloring<V> vc = vca.getColoring();
       return vc.getColors();
}
```

El Map resultante asocia un color a cada asignatura. Las asignaturas con el mismo color pueden ser impartidas a la misma hora. Esto lo conseguimos con el código

```
Graph<String,DefaultEdge>  g = leeAlumnosComunes(String file);
Map<String, Integer> m = coloreado(g);
Map<Integer, List<String>> r =
      m.keySet().stream()
            .collect(Collectors.groupingBy(s->m.get(s)));
```

El map r está formado por grupos cada uno de los cuales es una franja horaria. El número de franjas horarias es el número de colores diferentes que han hecho falta. ¿Se puede hacer un horario de n horas sin incompatibilidades?

Miguel Toro es doctor en Ingeniero Industrial por la Universidad de Sevilla y catedrático del Departamento de Lenguajes y Sistemas Informáticos de la misma universidad.

Ha sido director de la Oficina de Transferencia de Resultados de la Investigación (OTRI) y Director General de Investigación, Tecnología y Empresa de la Junta de Andalucía.

Ha tenido un papel activo en la Agencia Andaluza de Evaluación de la Calidad y Acreditación Universitaria (AGAE), ha sido miembro del Consejo Asesor de la Agencia Nacional de Evaluación de la Calidad y Acreditación (ANECA) y colabora asiduamente con varias agencias nacionales de evaluación universitaria.

Ha sido el Presidente de Sistedes (Sociedad Nacional de Ingeniería del Software y Tecnologías de Desarrollo de Software) y el Presidente de la Sociedad Científica Informática de España (SCIE) que engloba a los informáticos de las universidades españolas.

Ha recibido el Premio Fama de la Universidad de Sevilla, el Premio Sistedes de la Sociedad Nacional de Ingeniería del Software y Tecnologías de Desarrollo de Software en reconocimiento a su labor de promoción y consolidación de la Informática en España.

Ha recibido el Premio Nacional de Informática José García Santesmases a la trayectoria profesional otorgado por la Sociedad Científica Informática de España.

Actualmente es director del Instituto de Ingeniería Informática de la Universidad de Sevilla.

Este libro está especialmente orientado hacia la enseñanza de la asignatura Análisis y Diseño de Datos y Algoritmos, que es la continuación natural de Fundamentos de Programación, impartidas ambas en los diferentes grados de Ingeniería Informática. En él se abordan las técnicas para el diseño de algoritmos iterativos, las de diseño de algoritmos recursivos, así como el análisis y las transformaciones de un tipo de algoritmo en otros. Para abordar el diseño, es necesario haber asimilado previamente los elementos de la programación en algún lenguaje, de manera que en el seguimiento del contenido será necesario conocer Java y sus peculiaridades. El lector interesado puede consultar otras obras del autor en esta misma colección que le permitirán abordar estos aspecto: *Fundamentos de Programación: Python* y *Fundamentos de Programación: Java*. Los conceptos sobre análisis y diseño de datos y algoritmos se concretarán, pues, en el lenguaje de programación Java; en otro volumen, lo harán en el lenguaje C.

UNIVERSIDAD DE SEVILLA
eus
Editorial Universidad de Sevilla