Introducción a la Programación

Tema 10. Introducción a la recursividad y su relación con la iteración

1. Introducción

En los capítulos anteriores hemos visto solamente algoritmos iterativos. En este capítulo vamos a introducir el concepto de recursividad. En ese capítulo aprenderemos a definir problemas recursivamente e introduciremos el concepto de tamaño del problema.

Un algoritmo recursivo es aquél que expresa la solución de un problema en términos de llamada o llamadas a sí mismo. Cada llamada a sí mismo se denomina llamada recursiva. Definir recursivamente un problema es definirlo en términos de problemas de la misma naturaleza pero más pequeños.

En este capítulo volveremos a ver, también, los algoritmos iterativos, su relación con los recursivos y algunos conceptos adicionales para los mismos.

2. Algunos detalles de implementación y de notación

Los lenguajes imperativos tienen una semántica que debemos comprender para manipular adecuadamente los programas que construimos. Para ello vamos a introducir los conceptos de *bloque básico, conjunto de restricciones, operador de asignación paralela, posibilidad de eliminar variables y asignaciones en un bloque básico y la posibilidad de reordenar las asignaciones en un bloque básico*.

Bloque básico

En los lenguajes imperativos hay sentencias que producen un cambio del valor de una variable dada. La más usual es la sentencia de asignación. Una asignación se representa en la mayoría de los lenguajes de la forma:

$$x = e(x, a);$$

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 de la variable permanece fija pero el valor va cambiando a lo largo de la ejecución de un programa. Los sucesivos valores que una variable x va tomando los designaremos por x, x', x'', x''', … Usaremos

indistintamente x para representar la variable o su valor. Esto puede ser confuso en algunas ocasiones.

Como sabemos el operador asignación (=) no es simétrico. La expresión anterior cambia completamente si cambiamos el orden. En una expresión de asignación se evalúa el valor de la expresión de la derecha y ese valor se asigna a la variable de la izquierda. El antiguo valor que tuviera se pierde, se olvida.

Llamaremos bloque básico a una secuencia de sentencias de asignación. Por tanto en un bloque básico no hay *if* ni *while*. Un primer ejemplo de bloque básico es:

$$a = x;$$
$$b = y;$$
$$x = b;$$
$$y = a;$$

Otro segundo ejemplo es:

$$a = x;$$
$$x = y;$$
$$y = a;$$

Y un tercero:

$$x = y;$$
$$a = x;$$
$$y = a;$$

En un bloque básico puede haber un conjunto de variables, que llamaremos básicas, que son las que pretendemos manipular (cambiar su valor de alguna forma) y otras, que llamaremos variables intermedias que sólo sirven como contenedores pero de las que no nos interesa su valor al final del bloque. Normalmente estamos interesados en el valor final de las variables básicas en función del valor que tenían al principio del bloque y de posiblemente otras variables no relevantes. Necesitamos una notación para expresar el cambio que conseguimos (o queremos conseguir) en los valores de un conjunto de variables tras ejecutar un bloque básico. Para ello designaremos un conjunto de variables como $x = (x_1, x_2, \dots, x_m)$, por $x_i{}'$ el valor de la variable x_i tras ejecutar un bloque de código y por x' el valor final de todas ellas. La relación entre los valores finales y los originales la expresamos por un conjunto de restricciones entre ellos. Estas restricciones por ahora serán de igualdad. Más adelante aparecerán restricciones de otro tipo. Usaremos el operador == para representar la igualdad entre valores. El operador == si es simétrico a diferencia del operador de asignación (=) que no lo es, como hemos dicho. El operador == representa la igualdad entre valores es decir representa lo mismo que el operador == en Java usado entre tipos básicos o *.equals()* usado entre tipos objeto.

Intentemos deducir el valor final, x', y', de las variables x, y tras ejecutar los bloques básicos anteriores. Podemos comprobar que en el bloque 1 tenemos $x' == y; y' == x;$. También podemos comprobar que en el bloque 2 resulta igualmente $x' == y; y' == x;$. En ambos casos la conclusión es la misma: es decir los valores resultantes son los antiguos

intercambiados. Podemos concluir que ambos bloque básicos producen el mismo cambio en al variables básicas. Podemos pensar que el bloque básico 1 es equivalente al 2. Podemos definir entonces que dos bloques básicos son equivalentes si producen el mismo cambio en los valores de las variables básicas. O dicho de otra forma la relación entre los valores originales y finales es la misma.

Si examinamos el bloque 3 las relaciones entre los valores originales y finales es $x' == y; y' = = y;$. Es decir en este caso los nuevos valores son iguales al antiguo valor de y. Como vemos las relaciones son completamente distintas a los bloques anteriores. El bloque 3, que es el 2 con las asignaciones cambiadas de orden, no es equivalente al 2 ni al 1.

De lo anterior concluimos que el orden es muy importante en un bloque básico y que hay diferentes formas, unas más simples que otras, de conseguir un determinado cambio en los valores de un conjunto de variables.

Conjunto de restricciones

Una restricción entre varias variables es un predicado sobre las mismas. También se le suele llamara aserto. Un conjunto de restricciones es un conjunto de predicados combinados con el operador lógicos *and* que no haremos explícito pero suponemos que existe implícitamente combinando las restricciones que forman el conjunto. Más adelante introduciremos adicionalmente el operador lógico *or* para combinar predicados. Un ejemplo de conjunto de restricciones es el visto arriba:

$$x' == y; y' == y;.$$

En este caso el conjunto de restricciones se compone de dos predicados de igualdad combinados, como hemos comentado, por el operador *and*.

Los conjuntos de restricciones tienen propiedades completamente diferentes del conjunto de asignaciones que forman un bloque básico. La primera propiedad importante es que el orden no importa en el conjunto de restricciones y la segunda es que el conjunto de restricciones puede ser manipulado simbólicamente. Es decir podemos despejar una variable en una restricción de igualdad y sustituirla en el resto de las restricciones como solemos hacer en matemáticas. Representaremos por $e[a|y]$ queremos indicar la sustitución simbólica de a por y en la expresión e.

En un bloque básico no es posible la manipulación simbólica por los efectos laterales que va produciendo el operador de asignación. Las asignaciones en un bloque básico tienen un orden que no se puede cambiar. En un conjunto de restricciones el orden es irrelevante.

Un bloque básico se puede transformar en 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 una variable nueva, se renombran los usos posteriores de esa variable y se añade una restricción de igualdad al conjunto de restricciones. Las variables nuevas las representaremos por x', x'', x''', …

Veamos un ejemplo:

$$\begin{cases} b = x; \\ x = y; \\ y = b; \end{cases} \equiv \begin{matrix} b' == x; \\ x' == y; \\ y' == b'; \end{matrix}$$

La parte izquierda es un bloque básico. La parte derecha es un conjunto de restricciones. Es éste podemos despejar variables y sustituir. En el bloque básico no. En el conjunto de restricciones podemos despejar x', y' en función de x, y resultando el conjunto de restricciones $x' == y; y' == x;$. Estas restricciones nos dan el resultado neto del bloque básico sobre los valores iniciales x, y.

Como hemos dicho dos bloques básicos son equivalentes si dan lugar al mismo conjunto de restricciones entre los valores iniciales y finales de las variables. Esto podemos usarlo para simplificar un bloque básico. Algunas simplificaciones son:

- *Eliminación de asignaciones y variables intermedias*. Si una variable es definida en un punto mediante la asignación $u = e(z)$, y en un segmento posterior bloque básico las variables u, z no son definidas pero si usadas, entonces la asignación y la variable definida pueden ser eliminadas si sustituimos simbólicamente la variable u por la expresión $e(z)$ en el segmento de bloque básico referido. La sustitución simbólica la representaremos por $r[u|e(z)]$. Veamos la simplificación del bloque básico siguiente:

$$\begin{matrix} a = y; \\ b = x; \\ x = a; \\ y = b; \end{matrix} \equiv \begin{matrix} \\ b = x; \\ x = y; \\ y = b; \end{matrix}$$

  En el bloque básico de la izquierda la variable a es definida en $a = y;$. En las dos asignaciones siguientes no se definen las variables a, y por lo que la primera asignación puede ser eliminada junto con la variable definida haciendo la sustitución simbólica correspondiente. La corrección de la transformación puede comprobarse obteniendo el conjunto de restricciones equivalente a cada bloque básico y viendo que son iguales.

  La aplicación de la técnica anterior puede llevar en algunos casos a eliminar código inútil como en el caso siguiente donde la variable a es definida en la primera asignación pero sin ser usada se define de nuevo. Entonces la primera asignación puede ser eliminada.

$$\begin{array}{l} a = y; \\ b = x; \\ x = h; \\ a = b; \end{array} \equiv \begin{array}{l} b = x; \\ x = h; \\ a = b; \end{array}$$

- *Cambio de orden de asignaciones.* Dos asignaciones de la forma $x = f(x, a); y = g(y, a); z = h(z, a)$; pueden ser cambiadas de orden. Es decir ninguna de las variables definidas x, y, z es usada (no aparece) en la definición del resto de variables definidas.

El operador de asignación paralela

Algunos lenguajes de programación cuentan con el denominado operador de asignación paralela. Con este operador se pueden llevar a cabo varias asignaciones en paralelo como:

$$(x, y) \doteq (y, x);$$

El operador $\doteq$ es la asignación paralela. Es decir la asignación de los valores de una tupla de expresiones a una *tupla* de variables. El efecto neto de la asignación paralela anterior es el conjunto de restricciones $x' == y; y' = x;$. En un primer momento la notación puede ser confusa porque podemos darnos cuenta que la secuencia de asignaciones:

$$\begin{array}{l} x = y; \\ y = x; \end{array}$$

No consigue el efecto deseado. Es decir no es equivalente a una asignación paralela ya que da lugar al conjunto de restricciones $x' == y; y' = y;$. En general el operador de asignación paralela tiene la forma:

$$(x_1, x_2, \dots, x_m) \doteq (e_1(x_1, x_2, \dots, x_m), e_2(x_1, x_2, \dots, x_m), \dots, e_m(x_1, x_2, \dots, x_m))$$

Es conveniente conocer con detalle la forma de implementar este operador, su relación con las restricciones entre los valores de las variables antes y después de su ejecución y, también, su relación con los **bloque básicos** de código. Veamos cada uno de estos conceptos y sus relaciones.

Si designamos por $x_i{}'$ el valor de la variable $x_i{}'$ después de la asignación paralela

$$(x_1, x_2, \dots, x_m) \doteq (e_1(x_1, x_2, \dots, x_m), e_2(x_1, x_2, \dots, x_m), \dots, e_m(x_1, x_2, \dots, x_m))$$

Entonces los valores posteriores de esas variables son:

$$x_i{}' = e_i(x_1, x_2, \dots, x_m)$$

Es decir en una asignación paralela se toman los valores previos de cada una de las expresiones de la derecha, de forma independiente (paralela) se calculan los valores finales de cada una de las variables asignadas y se les asigna. La propiedad anterior implica que la variables de la parte izquierda pueden reordenarse de cualquier manera siempre que se reordenen de la misma forma las expresiones de la derecha. Así la asignación paralela anterior es equivalente a:

$$(x_n, x_2, \dots, x_1) \doteq (e_n(x_1, x_2, \dots, x_m), e_2(x_1, x_2, \dots, x_m), \dots, e_1(x_1, x_2, \dots, x_m))$$

Por otra parte un bloque básico (en los lenguajes imperativos) es una secuencia de asignaciones que van produciendo efectos laterales. Una asignación paralela y un bloque básico son conjuntos de asignaciones bastante diferentes. Los lenguajes de programación usuales disponen de bloques básicos pero no de asignación paralela. Debemos aprender a convertir una asignación paralela en un bloque básico y viceversa. Las asignaciones paralelas son más cómodas cuando hablamos de esquemas algorítmicos o las transformaciones de unos en otros.

Veamos ahora como conseguir un **bloque básico equivalente** a una asignación paralela. La idea general es usar variables nuevas, asignar a estas variables los valores de las expresiones, posteriormente asignar las nuevas variables a las antiguas y simplificar el bloque básico. El esquema es entonces:

$$(x_1, x_2, \dots, x_m) \doteq (e_1, e_2, \dots, e_m) \equiv \begin{cases} a_1 = e_1; \\ a_2 = e_2; \\ \quad \dots \\ a_m = e_m; \\ x_1 = a_1; \\ x_2 = a_2; \\ \quad \dots \\ x_m = a_m; \end{cases}$$

Por ejemplo

$$(x, y) \doteq (y, x) \equiv \begin{cases} a = y; \\ b = x; \\ x = a; \\ y = b; \end{cases} \equiv \begin{cases} b = x; \\ x = y; \\ y = b; \end{cases}$$

En el tercer paso hemos eliminado la variable a y la ecuación donde se definía y hemos sustituido su uso por su valor en la ecuación que define la x.

Como podemos comprobar el operador de asignación paralela se reduce a un bloque básico formado por una secuencia de asignaciones, en cualquier orden, cuando cada variable definida no usa ninguna de las demás variables definidas.

Veamos otro ejemplo:

$$(a, b) \doteq (b, a\%b)$$

Como podemos ver la implementación directa como secuencia de asignaciones es incorrecta. En efecto la secuencia de asignaciones siguiente no es equivalente a la asignación paralela anterior.

$$a = b;\ b = (a\%b);$$

La razón es que la primera asignación cambia el valor de la a y este valor se usa en la segunda en lugar del valor original. Otra forma de verlo es que el conjunto de restricciones equivalente a la asignación paralela es $a' == b; b' == a\%b$; mientras que el equivalente a la secuencia de dos asignaciones anteriores se obtiene despejando a', b' en $a' == b; b' == a'\%b$; para obtener $a' = b; b' == b\%b == 0;$. Es decir las restricciones resultantes son: $a' = b; b' = 0;$ y por lo tanto completamente distintas a las de la asignación paralela.

La implementación correcta es:

$$c = b;\ d = (a\%b); a = c; b = d;$$

Que puede ser simplificada eliminando la variable d:

$$c = b;\ b = (a\%b); a = c;$$

Si las variables en una asignación paralela (o en el equivalente conjunto de restricciones) pueden ser ordenadas en tal forma que cada variable x_i , $i \in [1..m]$ sólo usa en su definición las variables x_j , $j \in [1..i]$ entonces la asignación paralela es equivalente a una secuencia de asignaciones en el orden $x_m, x_{m-1}, \dots, x_1$. Como ejemplo tenemos:

$$(i, r) \doteq (i + 1, r * i);$$

Que por las razones dadas anteriormente se puede implementar como:

$$r = r * i;$$
$$i = i + 1;$$

Esto se puede comprobar deduciendo el conjunto de restricciones equivalente que en ambos casos es: $r' = r * i; i' = i + 1;$

En siguiente lugar veamos la forma de transformar un bloque básico en un asignación paralela que defina un conjunto de variables escogidas de entre todas las que aparecen en el bloque básico. Sean x, y las variables definidas en el bloque básico y a otras variables usadas pero no definidas. Si las variables y no van a ser usadas posteriormente y sólo estamos interesados en el valor final de las variables x entonces se trata de encontrar el conjunto de restricciones equivalente y despejar los valores finales de las x (x') en función de los valores originales de las y, a. A partir de ahí la asignación paralela buscada es inmediata.

$$\begin{cases} x_1' == e_1(x,y,a) \\ x_2' == e_2(x,y,a) \\ \quad\quad\dots \\ x_m' == e_m(x,y,a) \end{cases} \equiv (x_1, x_2, \dots, x_m) \doteq (e_1, e_2, \dots, e_m)$$

Aplicando esas ideas podemos transformar un bloque básico en una asignación paralela como podemos observar en el siguiente ejemplo.

$$\begin{matrix} a = y; \\ b = x; \\ x = a; \\ y = b; \end{matrix} \equiv \begin{matrix} a' == y; \\ b' == x; \\ x' == a'; \\ y' == b'; \end{matrix} \equiv \begin{matrix} x' = y \\ y' = x \end{matrix} \equiv (x,y) \doteq (y,x)$$

En un programa los bloques básicos se combinan en sentencias de control. Veamos primero la sentencia *if.* La sentencia *switch* la podemos considerar como una secuencia de sentencias *if-then-else.* Veamos ahora la forma de manipular programas constituidos de bloques básicos combinados con sentencias *if-then-else*. Asumimos que cada rama de la sentencia *if-then-else* contiene un bloque básico u otra sentencia *if-then-else*. Según la guarda sea verdadera o falsa en cada sentencia *if-then-else* el flujo de control de un programa continuará por la primera o la segunda rama. Llamaremos *camino* a la secuencia de bloques básicos que recorre el flujo de control de un programa. El camino recorrido depende del valor inicial de las variables y cada camino viene definido por la conjunción de las guardas correspondientes de las sentencias *if-then-else.* Como cada sentencia *if-then-else* tiene dos ramas un programa con n sentencias *if-then-else* tiene 2^n caminos.

Cada camino, escogido de forma individual, se compone de una secuencia de bloques básicos (por lo tanto un bloque básico) y por lo tanto asociado a cada camino podemos deducir un conjunto de restricciones entre los valores iniciales y finales de las variables como hemos visto anteriormente. La restricciones asociadas a cada camino contienen, además, las restricciones asociadas a las guardas que lo definen. El conjunto de restricciones asociado al programa es la combinación mediante el operador *or* (∨) de las restricciones asociadas a cada camino.

Veamos el siguiente ejemplo:

```
int max3(int a, int b, int c) {
       int r;
       if(a>=b)r = a;
       else r = b;
       if(r<c)r = c;
       return r;
}
```

El ejemplo anterior calcula el máximo de los tres parámetros de entrada. Hay cuatro caminos cuyas restricciones asociadas son:

$$r1: a \geq b; r' == a; r' < c; r'' == c;$$
$$r2: a \geq b; r' == a; r' \geq c;$$
$$r3: a < b; r' == b; r' < c; r'' == c;$$
$$r4: a < b; r' == b; r' \geq c;$$
$$r: r1 \vee r2 \vee r3 \vee r4$$

Que puede ser transformado a:

$$r1: a \geq b; a < c; r'' == c;$$
$$r2: a \geq b; a \geq c; r' == a;$$
$$r3: a < b; b < c; r'' == c;$$
$$r4: a < b; b \geq c; r' == b;$$
$$r: r1 \vee r2 \vee r3 \vee r4$$

Podemos comprobar que las restricciones resultantes especifican el resultado como el valor mayor o igual a los otros dos. Podemos pensar en otras restricciones adicionales para especificar el resultado del problema pero podemos ver que se pueden deducir de las anteriores por lo que no añaden nada a las anteriores. Estas restricciones pueden ser convertidas en un programa equivalente al anterior de la forma:

```
int max32(int a, int b, int c) {
	int r;
	if(a>=b && a<c)r = c;
	if(a>=b && a>=c)r = a;
	if(a<b && b<c)r = c;
	if(a<b && b>=c)r = b;
	return r;
}
```

Además de la sentencia *if-then-else* la siguiente de sentencia de control que usamos es la sentencia *while* (y sus equivalentes *for* clásico y *for* extendido y también los *while* anidados). La sentencia *while* podríamos tratarla asociándole un conjunto de caminos. Cada camino estaría asociado a i iteraciones del bucle comenzando con la guarda con valor verdadero y acabando con valor falso. De esta forma el conjunto de restricciones asociado al bucle sería la conjunción de las restricciones asociadas a cada camino.

El problema de este enfoque es que el número de caminos puede llegar a ser muy grande y cada camino en particular muy largo. Aunque este enfoque puede ser de interés en algunos casos es más frecuente un enfoque alternativo. En este segundo punto de vista se trata de asociar a cada bucle una restricción sobre las variables del bucle (las de la guarda y las del cuerpo) que debe ser verdadera al comenzar el bucle, al acabar y al finalizar cada una de las iteraciones. Esta restricción la llamaremos *Invariante* y la representaremos por $I(x)$. El invariante no se deduce del código del bucle. Hay que imaginarlo. Además hay que asociar una función sobre las variables del bucle y valores enteros positivos. Esta función denominada *Función de Cota* y que representaremos por $C(x)$ debe disminuir su valor en cada iteración. Tampoco la función de cota se deduce del código del bucle. En este capítulo aprendemos a deducir el código del bucle a partir de un *Invariante* y una *Función de Cota* dadas y por otra parte a deducir el conjunto de restricciones entre los valores iniciales y finales de las variables

de un programa con bloques básicos, sentencias *if* y *while* (siempre asumiendo que podemos asociar un invariante u una función de cota a cada *while*).

En el ejemplo siguiente queremos deducir las relaciones entre el valor inicial del parámetro de entrada n y el valor final de a, el valor devuelto.

```
int factorial(int n) {
       int i, a;
       i=0;
       a=1;
       while(i<n){
              i = i+1;
              a = a * i;
       }
       return a;
}
```

La variables del bucle son a, i, n siendo n sólo usada pero no definida por lo se comporta como una constante. Proponemos como invariante y función de cota las siguientes:

$$I(a,i,n): a == i!$$
$$C(a,i,n): n - i$$

Debemos comprobar que el invariante se cumple antes de la primera iteración y que suponiendo que se cumple a principio del cuerpo del bucle se cumple al final. Igualmente la función de cota desciende al ejecutarse el cuerpo. Los valores de a, i antes de empezar el bucle son $1, 0$. Se cumple, por lo tanto el invariante puesto que $1 == 0!$.

Para ver si este invariante y esta función de cota cumplen los requisitos introducimos, al principio del cuerpo una variable entera $c = n - i$, el valor en ese punto de la función de cota. Obtenemos las restricciones asociadas al cuerpo, las completamos con el invariante y vemos si es posible deducir que el invariante es válido con los valores finales y el nuevo valor de la función de cota es más bajo que el anterior.

$$I(a,i,n): a == i!;$$
$$c' == n - i;$$
$$i' == i + 1;$$
$$a' == a * i';$$

De dónde podemos concluir que

$$I(a',i',n):\ a' == i'!\ \equiv a * (i+1) == (i+1)! == i! * (i+1)$$
$$c'' == n - i' == n - (i+1) == n - i - 1 == c' - 1 > c'$$

Una vez que invariante y cota cumple los requisitos podemos sustituir, para conseguir las restricciones asociadas al programa, el bucle *while* por la restricción $I(x) \wedge \overline{g(x)}$. Dónde hemos usado ahora $\wedge$ para representar el operador lógico *and*, $g(x)$ es la guarda del bucle, y $\overline{g(x)}$ su negación. La restricción es entonces: $a == i!\,; \overline{i < n;}$. Puesto que $i == n;$ es el contrario de $\overline{i < n;}$ (en realidad el contrario es $i \geq n;$ pero como los valores de i van

aumentando de uno en uno el valor alcanzado es la igualdad) podemos concluir que la restricción final es $a == i! ; i == n$; o equivalentemente $a == n!$;. Es decir el valor devuelto es el factorial de n.

Con la ideas anteriores podemos obtener la relación entre los valores iniciales de las variables (o de los parámetros de entrada) con los valores finales (o retornados) de un programa compuesto de bloques básicos combinados mediante sentencias *if* y *while*. Como hemos comentado es necesario imaginar un invariante y una función de cota asociado a cada bucle *while*.

El camino que hemos seguido ha sido partir del código y deducir la relación entre valores iniciales y finales. El uso que haremos de estas técnicas es comprobar que un determinado código lleva a cabo una transformación de valores de las variables especificada.

El camino inverso consistente en dada una transformación buscada, y posiblemente un invariante y una cota, encontrar un código (compuesto de bloques básicos, *if* y *while*) que la lleve a cabo. Esta tarea se llama Diseño Iterativo y la veremos más adelante en este capítulo.

3. Otros detalles de implementación en C y lenguajes similares

Hay varios temas a considerar si usamos lenguajes de implementación de más bajo nivel como C: las tuplas de valores, la implementación de las listas, los mecanismos para devolver varios resultados a la vez por parte de un algoritmo, la distintas formas de implementar variables compartidas entre distintas funciones y la forma de tratar con las excepciones.

En muchos casos conviene tratar varias propiedades de forma agrupada formando tuplas. Una tupla, que representaremos por $(a1, a2, \ldots an)$, es un agregado de valores de diferentes tipos a diferencia de una lista, que representaremos de la misma forma, dónde todos los valores del agregado son del mismo tipo. Algunos lenguajes disponen de tuplas pero Java y C no las tienen. Las *tuplas* se pueden implementar como objetos en Java y como *struct* en C.

Las listas pueden ser implementadas como tales en Java y en C mediante *arrays*. Los *arrays* son objetos en Java que tienen el número de elementos como propiedad asociada y disparan excepciones si se intenta acceder a un índice no permitido. En C un array no dispone de esa información y para representar una lista de tamaño variable se la tenemos que añadir. En C modelamos una lista de tamaño variable por la *tupla* $(T * dt,\ int\ n,\ int\ m)$ o simplemente por las tres variables que la constituyen. El primer campo son los datos del array y suele ser un puntero al tipo de datos de las casillas, *n* es el número de elementos que contiene y *m* el número máximo de elementos que puede contener según el tamaño de la memoria reservada. Si *m* es una constante conocida no la haremos explícita.

Otra cuestión es los parámetros de entrada y entrada-salida en C y en Java. En Java son parámetros de entrada los tipos básicos y los tipos objetos inmutables. Son parámetros de entrada-salida los tipos mutables.

En C son parámetros de entrada los tipos básicos, los *struct* y todos aquellos parámetros formales que lleven el calificativo *const*. Los parámetros de entrada-salida son aquellos especificados como tipo puntero y sin calificativo *const*.

En algunos esquemas algorítmicos las funciones que aparecen toman unos parámetros de entrada y pueden devuelven uno o varios de salida, pudiendo producir efectos laterales en propiedades compartidas.

Los parámetros de entrada en los esquemas se implementan como parámetros de entrada en el lenguaje de programación escogido. Los parámetros de salida se pueden convertir en parámetros de entrada salida en lenguajes tipo C de la forma:

```
R f(T x) {
        ...
        return s;
}
```

Puede ser trasformada en

```
void f(T x, R * s) {
        ...
}
```

Donde, s en un parámetro de entrada-salida.

En Java un parámetro de salida se puede convertir en otro se entrada salida si es de tipo mutable. Si no lo es hay que convertir el tipo inmutable en otro mutable equivalente.

Otra forma de implementar los parámetros de salida, cuando hay varios, es agruparlos en una *tupla*.

En C no existe *static,* que en Java es adecuado para modelar propiedades compartidas, pero puede ser emulado declarando variables globales con un ámbito adecuado.

En el caso de Java la no solución de un problema se puede implementar mediante el disparo de una excepción. Pero las excepciones no están disponibles en *C*. Para modelar las excepciones en C es adecuado que todas las funciones que puedan disparar excepciones devuelvan un entero. Si el valor del entero es positivo la terminación ha sido normal y si es negativo lo consideramos equivalente al disparo de una excepción codificada en el valor negativo devuelto.

4. Elementos de recursividad

La definición recursiva de un problema es una especificación de la solución del mismo 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 una solución. En toda definición recursiva

aparecen los conceptos de caso base, caso recursivo y tamaño de un problema. Veamos para ir aclarando estos conceptos un ejemplo. Queremos definir el problema *n!* (factorial de *n*) y queremos hacerlo de forma recursiva. La definición es de la forma:

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

La idoneidad de la definición podemos verla con un ejemplo:

$$3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1$$

Lo primero que debemos tener en cuenta es que cuando vamos a hacer una definición recursiva de un problema siempre debemos tener en cuenta un **conjunto de problemas**. Al conjunto de problemas los llamamos también **dominio**. En este caso el conjunto de problemas viene dado por todos los problemas del tipo *n!* para todo *n>=0.*

En lo que sigue representaremos los problemas por $p, p_1, p_2, \dots, p_r$. Un conjunto de problemas lo representaremos por *P*. Cada problema tendrá unas propiedades *x*. Cada propiedad específica la representaremos mediante un superíndice: $x = (x^1, \dots, x^m)$. Dentro de un conjunto de problemas *P* los valores de sus propiedades identifican al problema de manera única.

Un problema podemos pensarlo como un objeto. El diseño de los problemas se hace con la misma metodología orientada a objetos que hemos explicado en capítulos anteriores. Por la misma razón las propiedades de un problema pueden clasificarse en básicas y derivadas, individuales y compartidas, consultables y modificables, etc. Para cada problema del conjunto de problemas podemos indicar un **invariante del problema**, $I(x)$, que es una expresión lógica que debe ser válida para las propiedades de cada problema en particular. También podemos indicar el **dominio**, $D(x)$, que es una expresión que es válida para las propiedades de todos los problemas que están incluidos en el conjunto de problemas de interés. En general llamaremos **aserto**, $A(x)$, a cualquiera expresión lógica construida con la propiedades del problema. Al escribir los asertos usamos los operadores lógicos *not*, *and*, *or* representados como ¬,Λ,V.

A cada problema podemos asociar el concepto de **tamaño** que es una nueva propiedad derivada del mismo. 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 sobre las propiedades del mismo. Lo representamos por $n = t(x)$ o $n = t(p)$. El tamaño del problema deber ser un entero mayor o igual que cero que nos dé una idea de la complejidad del mismo. 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. En el conjunto de problemas $n!, n \geq 0$ el problema *0!* es un caso base. Su solución es 1. Puede haber más de un caso base**.** El resto de

problemas del conjunto considerado (en este caso todos los que tienen $n > 0$) 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 sub-problemas. Un mismo problema puede tener diferentes definiciones recursivas. Otra definición recursiva para la factorial es:

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

En este caso hay dos casos base. Y otra más es:

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

Esta última definición recursiva es menos evidente pero, como veremos más adelante, importante. Ahora el conjunto de problemas es $(n,m), n \geq 0, m \geq 1$ y la definición recursiva de la solución:

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

Con esa definición la solución del problema original, $n!$, es igual la del problema $fa(n,1)$. Es decir hemos definido la solución del problema *n!,* que tiene una sola propiedad, en base a otro que tiene dos: $fa(n,m)$. El tamaño del problema $n!$ es *n* y el de (n,m) también es *n*.

Podemos comprobar con un ejemplo que la definición es adecuada. En efecto:

$$3! = (3,1) = (2,3*1) = (1,2*3*1) = (0,1*2*3*1) = 1*2*3*1$$

En este capítulo aprenderemos a transformar unas definiciones recursivas en otras y a escoger la mejor de ellas para diferentes propósitos.

Pero no todas las definiciones recursivas son adecuadas para construir un algoritmo. Para que una definición recursiva pueda convertirse en un algoritmo debe tener al menos un caso base, y cada caso recursivo definirse en base a otro u otros de menor tamaño. Las siguientes son propiedades de la factorial

$$\begin{cases} 0! = 1 \\ \dfrac{(n+1)!}{n+1} = n! \end{cases}$$

Pero juntas no forman un algoritmo recursivo.

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

La definición anterior no constituye un algoritmo porque el caso recursivo se ha definido en base a otros problemas de tamaño mayor. En un algoritmo es necesario que los sub-problemas usados para definir el caso recursivo tengan un tamaño menor. Así en cada **paso recursivo** (paso de un problema a los sub-problemas que lo definen) se reduce el tamaño. Como éste deber ser mayor o igual a cero para todos los problemas del conjunto considerado en algún momento llegaremos al caso base y el algoritmo acabará. Esto no ocurre en la definición incorrecta anterior.

Un algoritmo puede usarse para encontrar la solución del problema en cuestión. Veremos algoritmos recursivos escritos en C y en Java.

```
int factorial1(int n) {
	int r;
	assert(n>=0);
	if(n==0){
		r = 1;
	} else {
		r = n*factorial1(n-1);
	}
	return r;
}
```

Este algoritmo recursivo es la transcripción mimética de la primera definición recursiva que dimos para la factorial. Las otras definiciones tienen algoritmos similares. Hemos escrito la aserción *assert(n>=0)* para insistir en que el conjunto de problemas que hemos considerado está formado por aquellos que tienen *n>=0.*

Como segundo ejemplo de problema definido recursivamente tenemos el máximo común divisor. Son conocidas varias propiedades del máximo común divisor:

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

Para diseñar un algoritmo recursivo debemos escoger el conjunto de problemas y el tamaño. Cada problema lo representamos por dos propiedades enteras, (a, b), que son los dos enteros cuyo máximo común divisor queremos calcular. Suponemos que el conjunto de problemas está formado para todos los pares *a, b* que son mayores o iguales a cero pero no ambos iguales a cero. El tamaño de un problema dado lo escogemos como el valor de b. Con estas ideas, y las propiedades anteriores, una definición recursiva es:

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

Vemos que el algoritmo está bien definido: el sub-problema tiene un tamaño menor que el del problema (dado que el resto de una división entera es menor que el dividendo y el divisor) y todos los problemas del dominio tienen un tamaño mayor o igual a cero. Podemos comprobar la idoneidad de la definición con un par de ejemplos:

$$mcd(9,6) = mcd(6,3) = mcd(3,0) = 3$$
$$mcd(8,12) = mcd(12,8) = mcd(8,4) = mcd(4,0) = 4$$

El algoritmo equivalente a esa definición se muestra en el ejemplo siguiente.

```
int mcd(int a, int b) {
	int r;
	assert(a>=0 && b>=0 && !((a==0) && (b==0)));
	if(b==0){
		r = a;
	} else {
		r = mcd(b,a%b);
	}
	return r;
}
```

5. Esquemas recursivos

4.1 Introducción

En general los algoritmos recursivos siguen el siguiente esquema:

```
R g(P p){
  return h(f(i(p)));
}
S f(X x) {
	S r;
	LS s;
	LX y;
	assert(D(x));
	if(b(x)){
		r = sb(x);
	} else {
		y = sp(x);
		assert(t(y) < t(x));
		s = f_L(y);
		r = c(x,s);
	}
	return r;
}
```

Llamaremos a éste esquema, **esquema recursivo sin memoria**. Este denominado usualmente como **divide y vencerás sin memoria**.

Una versión más compacta del mismo esquema es:

$$f(x) = \begin{cases} sb(x), & b(x) \\ c(x, f_L(sp(x))) & !\, b(x) \end{cases}$$
$$g(p) = h(f(i(p)))$$

Como hemos comentado arriba al hacer una definición recursiva debemos considerar un conjunto de problemas. Cada problema viene definido por un conjunto de propiedades p. Cada propiedad específica la representaremos mediante un superíndice: $p = (p^1, \dots, p^l)$. El problema generalizado tiene $m \geq l$ propiedades que representaremos por $x = (x^1, \dots, x^m)$. Representamos los problemas por $p, p_1, p_2, \dots$ y los problemas generalizados por $x, x_1, x_2, \dots$ Puede haber k sub-problemas $sp(x) = (sp_1(x), sp_2(x), \dots, sp_k(x))$ para cada problema generalizado.

En el párrafo anterior hemos escogido la notación (a, b, c) para representar *tuplas* de valores.

En el esquema recursivo aparecen las siguientes funciones y variables:

- *P* es el tipo de las propiedades del problema. Normalmente son un conjunto de propiedades. Como sabemos las propiedades son individuales y compartidas. Las primeras se implementan como parámetros de entrada. Las segundas según los casos pueden ser de entrada o de entrada-salida. X es el tipo de las propiedades del problema generalizado. Por LX representamos una secuencia de problemas generalizados.
- *R* es el tipo de los resultados buscados. Puede ser uno o varios.
- *LX* es una secuencia de X. Representa las diversos sub-problemas.
- *LS* es una lista de S. Representa los resultados intermedios de los sub-problemas.
- *D(x):* Es una función lógica que especifica el dominio de conjunto de problemas generalizados. Es decir verdadero si el problema pertenece al conjunto de problemas. Se ha recogido en un aserto el hecho de que el problema sea uno de los considerados.
- *t(x):* Es una función que especifica el tamaño del conjunto de problemas. Cada sub-problema debe ser de un tamaño inferior al problema de partida. Esto lo hemos recogido en un aserto del esquema.
- *b(x):* Es una función lógica que devuelve verdadero si el problema es un caso base.
- *sb(x):* Es una función que devuelve un valor de tipo *R* que es la solución del caso base.
- *sp(x):*Es una función que calcula las propiedades del sub-problema (o de los sub-problemas) al que se reduce el problema original.
- *s*: Es una variable de tipo *LS* (una secuencia) que guardan la solución de los sub-problemas.
- *y:* Es una variable de tipo *LX* (una secuencia) que guarda los sub-problemas al que se reduce el problema original
- *r*: Es una variable de tipo S que guarda la solución del problema generalizado.
- *s*: Es una variable de tipo LS, una secuencia de soluciones, que guarda la solución de los sub-problemas.
- *c(x,s):* Es una función, que llamaremos función de combinación, que obtiene el resultado combinando las propiedades del problema con el resultado del sub-problemas (resultado de las llamada recursiva).

- *i(p)* es la función de instanciación que a partir del problema original escoge un problema generalizado.
- *h(r)* es la función que transforma la solución del problema generalizado en la del problema original.

Donde el operador = se entenderá en general como ≔ el operador de asignación paralela que hemos visto arriba y f_L como la aplicación de f a una secuencia.

4.2 Diseño Recursivo

Con los elementos vistos arriba es posible ver los elementos que componen el diseño recursivo de un algoritmo. Este tipo de diseño se le suele llamar **Técnica de Divide y Vencerás** porque busca la solución de un problema a partir de la de otros subproblemas más pequeños según su tamaño. Estos elementos son:

- A partir de un problema inicial definido por la propiedades *p* de tipo *P* generalizamos el problema a otro con propiedades *x* de tipo *X*. Para precisar el conjunto de problemas de interés definimos un domino *D(x).*
- A cada uno de los problemas generalizados le asociamos un tamaño *t(x).* Este tamaño debe ser mayor o igual que cero para cada problema del dominio.
- Escogemos un conjunto de problemas de tamaño pequeño para los que conocemos su solución. Estos problemas vienen especificados por el predicado *b(x)* y su solución por *sb(x).*
- Para cada problema escogemos un conjunto de subproblemas. Estos tienen que ser de tamaño menor y adecuados para poder construir la solución del problema a partir de las soluciones de los sub-problemas. Los subproblemas vienen dados por *sp(x).*
- Diseñamos una función de combinación *c(x,s)* capaz de calcular la solución de un problema a partir de las soluciones de los subproblemas y sus propias propiedades.
- Por último se diseña una función de instanciación *i(p)* y la correspondiente *h(r).*

Por motivos de diseño las propiedades de los problemas generalizados las podemos clasificar en **individuales** y **compartidas**. Las propiedades individuales son específicas para cada problema de tal forma que cada problema puede ser identificado de forma única por sus propiedades individuales. Las propiedades compartidas son comunes a todos los problemas generalizados. Ambos tipos de propiedades tienen técnicas de implementación distintas tal como veremos más abajo.

6. Ejemplos de algoritmos recursivos

Veamos en algunos ejemplos anteriores como identificar cada una de estas funciones. En el ejemplo del factorial que volvemos a reproducir aquí.

```
int factorial1(int n) {
	int r;
	assert(n>=0);
	if(n==0){
		r = 1;
	} else {
		r = n*factorial1(n-1);
	}
	return r;
}
```

Las funciones anteriores son:

$$D(n) \equiv n \geq 0, t(n) = n, sb(n) = 1, sp(n) = n - 1, c(n, s) = n * s$$

Y en el problema del máximo común divisor:

```
int mcd(int a, int b) {
	int r;
	assert(a>=0 && b>=0 && !((a==0) && (b==0)));
	if(b==0){
		r = a;
	} else {
		r = mcd(b,a%b);
	}
	return r;
}
```

Cada problema tiene las propiedades (a, b). Las funciones del esquema son:

$$D(a,b) \equiv a \geq 0 \ \wedge \ b \geq 0 \ \wedge !(a = 0 \ \wedge \ b = 0),$$
$$t(a,b) = b, \quad b(a,b) \equiv b = o, \quad sb(a,b) = a,$$
$$sp(a,b) = (b, a\%b),$$
$$c(a,b,s) = s.$$

7. Generalización de problemas

Una definición recursiva necesita considerar 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. Para encontrar el conjunto de problemas de interés es necesario, en la mayoría de los casos, generalizar el problema original. Generalizar un problema es añadir propiedades al problema original para considerarlo un caso particular de un conjunto de problemas más amplio.

Generalización de secuencias indexables

Un conjunto de generalizaciones interesantes son las generalizaciones de secuencias indexables. Una secuencia indexable podemos definirla en general como un agregado de datos dt de tamaño n y donde podemos obtener los elementos a partir de un índice de la forma $dt[i], \ 0 \leq i < n$. Ejemplos de secuencias indexables son las listas, los arrays, las hileras de

caracteres, etc. El problema original lo asumimos definido por (dt, n), los datos y el tamaño de los mismos.

En lo que sigue asumiremos la convención Java: cualquier sublista (i, j) incluirá la casilla *i* pero no la *j*. Por lo tanto su tamaño será *j-i*.

Las generalizaciones posibles, el tamaño de los problemas asociados, y los posibles subproblemas son los siguientes:

- Prefijo: El problema se generaliza a $(dt, n, i), 0 \leq i \leq n$. Cada problema generalizado representa al prefijo de la secuencia formado por los elementos con índices $[0, i)$ de tamaño i. Por cada problema generalizado (dt, n, i) uno más pequeño al que se puede reducir es $(dt, n, i-1)$. Casos base posibles son $(dt, n, 0), (dt, n, 1), (dt, n, 2)$. La definición recursiva es:

$$f(i) = \begin{cases} h0, & i = 0 \\ h1(dt[0]), & i = 1 \\ h2(dt[0], dt[1]), & i = 2 \\ f(i-1) \odot dt[i-1], & i > 2 \end{cases}$$

$$g(dt, n) = f(n)$$

- Sufijo: El problema se generaliza a $(dt, n, i), 0 \leq i \leq n$. Cada problema generalizado representa al prefijo de la secuencia formado por los elementos con índices $[i, n)$ de tamaño $n - i$. Por cada problema generalizado (dt, n, i) uno más pequeño al que se puede reducir es $(dt, n, i+1)$. Casos base posibles son $(dt, n, n), (dt, n, n-1), (dt, n, n-2)$. La definición recursiva es:

$$f(i) = \begin{cases} h0, & n - i = 0 \\ h1(dt[n-1]), & n - i = 1 \\ h2(dt[n-1], dt[n-2]), & n - i = 2 \\ f(i+1) \odot dt[i], & n - i > 2 \end{cases}$$

$$g(dt, n) = f(0)$$

- Subsecuencia central: El problema se generaliza a $(dt, n, i, j), 0 \leq i \leq n, i \leq j \leq n$. Cada problema generalizado representa la Subsecuencia central formada por los elementos con índices $[i, j)$ de tamaño $j - i$. Por cada problema generalizado (dt, n, i, j) uno más pequeño al que se puede reducir es $(dt, n, i+1, j-1)$. Casos base posibles son $(dt, n, i, i), (dt, n, i, i+1), (dt, n, i, i+2)$. La definición recursiva es:

$$f(i, j) = \begin{cases} h0, & j - i = 0 \\ h1(dt[i]), & j - i = 1 \\ h2(dt[i], dt[i+1]), & j - i = 2 \\ dt[i] \odot f(i+1, j-1) \odot dt[j-1], & j - i > 2 \end{cases}$$

$$g(dt, n) = f(0, n)$$

- Subsecuencia mitad: El problema se generaliza a $(dt, n, i, j), 0 \leq i \leq n, i \leq j \leq n$. Cada problema generalizado representa la subsecuencia mitad formada por los elementos con índices $[i, j)$ de tamaño $j - i$. Por cada problema generalizado (dt, n, i, j) dos más pequeños al que se puede reducir es $(dt, n, i, k), (dt, n, k, j), k = \frac{i+j}{2}$. Casos base posibles son $(dt, n, i, i), (dt, n, i, i + 1), (dt, n, i, i + 2)$. La definición recursiva es:

$$f(i,j) = \begin{cases} h0, & j - i = 0 \\ h1(dt[i]), & j - i = 1 \\ h2(dt[i], dt[i+1]), & j - i = 2 \\ f(i,k) \odot f(k,j), & j - i > 2 \end{cases}$$

$$g(dt, n) = f(0, n)$$

Generalización de secuencias no indexables

Otro conjunto de generalizaciones son las de secuencias que no son directamente indexables pero pueden ser divididas en subsecuencias de tamaño más pequeño siguiendo los esquemas anteriores. Asumimos que el tamaño de una secuencia lo representamos por |dt| aunque en munchos casos este dato aparecerá como una propiedad adicional del problema. Ahora buscamos los subproblemas a los que reducir un problema.

Las variantes posibles, el tamaño de los problemas asociados, y los posibles subproblemas son, como anteriormente, los siguientes:

- Prefijo: El problema se descompone en $dt = dt' \oplus e$. Dónde e es una secuencia de tamaño 1. Casos base posibles son $dt, = (), dt = (e), dt = (e1, e2)$. La definición recursiva es:

$$f(dt) = \begin{cases} h0, & dt = () \\ h1(e), & dt = (e) \\ h2(e2, e2), & dt = (e1, e2) \\ f(dt') \odot e, & |dt| > 2 \end{cases}$$

- Sufijo: El problema se descompone en $dt = e \oplus dt'$. Dónde e es una secuencia de tamaño 1. Casos base posibles son $dt, = (), dt = (e), dt = (e1, e2)$. La definición recursiva es:

$$f(dt) = \begin{cases} h0, & dt = () \\ h1(e), & dt = (e) \\ h2(e2, e2), & dt = (e1, e2) \\ e \odot f(dt'), & |dt| > 2 \end{cases}$$

- Subsecuencia central 1: El problema se descompone en $dt = e1 \oplus dt' \oplus e2$. Dónde $e1, e2$ son secuencias de tamaño 1. Casos base posibles son $dt, = (), dt = (e), dt = (e1, e2)$. La definición recursiva es

$$f(dt) = \begin{cases} h0, & dt = () \\ h1(e), & dt = (e) \\ h2(e2, e2), & dt = (e1, e2) \\ e1 \odot f(dt') \odot e2, & |dt| > 2 \end{cases}$$

- Subsecuencia central 2: El problema se descompone en $dt = dt' \oplus dt''$. Casos base posibles son $dt, = (), dt = (e), dt = (e1, e2)$. La definición recursiva es:

$$f(dt) = \begin{cases} h0, & dt = () \\ h1(e), & dt = (e) \\ h2(e2, e2), & dt = (e1, e2) \\ f(dt') \odot f(dt''), & |dt| > 2 \end{cases}$$

Para poder implementar las generalizaciones anteriores debemos disponer de funciones que calculen dt', dt'', e a partir de dt.

Máximo de una lista de enteros

En primer lugar veamos una definición recursiva del cálculo del máximo de una lista de enteros. Consideramos, entonces, que la información de entrada son los datos dt y un entero n que indica el número de los mismos. El problema planteado puede representarse por (dt, n) y la solución buscada por $maxL(dt, n)$. El dominio de problemas considerados es $\mathrm{D(dt, n)} \equiv n \geq 0$. Sabemos que para $n = 0$ el problema planteado no tiene solución.

Con esos datos es prácticamente imposible considerar un conjunto de problemas adecuado para plantear la definición recursiva. Por lo tanto generalizamos el problema considerando alguna de las generalizaciones vista arriba. Escogemos como ejemplo la generalización sufijo anterior. Con esta generalización el conjunto de problemas está formado por las sub-listas (i, n).

Ahora cada problema generalizado podemos representarlo por la propiedades (i) y asumiendo dt, n como variables compartidas, Cada problema generalizado consiste, por lo tanto, en buscar el máximo valor en la sub-lista (i, n) de *dt*.

Y el tamaño

$$t(i) = n - i$$

De las tres propiedades la primera i es una propiedad individual. Las dos siguientes $\mathrm{dt, n}$ propiedades compartidas del conjunto de problemas. Usaremos, por claridad, la función $\mathrm{max(a, b) = a \geq b ? a : b}$ que el calcula el máximo de dos valores con respecto a un orden. Con la notación $\mathrm{dt[i]}$ representamos el valor de la casilla i.

$$maxLG(i) = \begin{cases} dt[i], & n - i = 1 \\ max(dt[i], maLG(i + 1)), & n - i > 1 \end{cases}$$

La definición recursiva anterior se basa en la siguiente idea: el máximo de los valores contenidos en las casillas de un sub-lista (i,n) es el máximo entre primera casilla y el máximo del valor contenido en las casillas restantes excluyendo la primera.

El paso final consiste en escoger el problema generalizado equivalente al problema original

$$maxL(dt,n) = \begin{cases} \bot, & n = 0 \\ maxLG(0), & n \geq 0 \end{cases}$$

El algoritmo correspondiente escrito en C, convirtiendo en valor máximo devuelvo en un parámetro de entrada-salida, es:

```
int max(int a, int b){
	return a >= b ? a : b;
}

int maxL(int * dt, int n, int * m) {
	int r = -1;
	if((n>0){
		*m = maxLG(0,dt,n);
		r = 1;
	}
	return r;
}

int maxLG(int i, int * dt, int n){
	int r;
	assert(i>=0 && i<n && n>0);
	if(n-i == 1){
		r = dt[i];
	} else {
		r = max(dt[i],maxLG(i+1,dt,n));
	}
	return r;
}
```

Como vemos hemos diseñado una función para la solución del problema generalizado, *maxLG*, y otra, *maxL*, para instanciar el problema original y disparar las posibles excepciones.

Resumamos los elementos de este algoritmo recursivo. En primer lugar hemos generalizado el problema. El problema original era encontrar el máximo valor de las casillas de una lista *dt* con *n* elementos. El problema generalizado es encontrar el máximo valor de las casillas de las sub-lista (i,n). El problema original se obtiene dando el valor *0* al parámetro *i* del problema generalizado. Los parámetros del problema generalizado podemos clasificarlos en compartidos *(dt, n)* e individuales *(i).* Por lo tanto un problema concreto dentro del conjunto considerado lo podemos identificar con el parámetro *i.* Los otros dos parámetros son compartidos por todos los problemas. El resto de las funciones del esquema son:

```
D(dt,n,i)= n>0 && i>=0 && i<n
b(i,dt,n) = n-i == 1
sb(i,dt,n) = dt[i]
(i+1,dt,n) = sp(i,dt,n)
c(i,dt,n,s)= max(dt[i],s)
```

El código anterior podemos escribirlo en Java. Ahora podemos usar métodos genéricos (siendo *T* el tipo de los elementos de la lista, podemos usar excepciones y hacer que el orden sea un parámetro más del problema. El código quedaría ahora como:

```
private static <T> T max(T a, T b, Comparator<T> ord){
	return ord.compare(a,b)>=0 ? a : b;
}

public static <T extends Comparable<? super T> T maxL(List<T> dt) {
	Preconditions.checkArgument(dt.size()>0);
	Comparator<T> ord = Comparator.naturalOrder();
	return max(0,dt,ord);;
}

public static <T> T maxL(List<T> dt, Comparator<T> cmp) {
	Preconditions.checkArgument(dt.size()>0);
	return maxLG(0,dt,ord);;
}

private static <T> T maxLG(int i, List<T> dt, Comparator<T> ord){
	int r;
	int n = dt.size();
	Preconditions.checkElementIndex(i,n);
	if(n-i == 1){
		r = dt.get(i);
	} else {
		r = max(dt.get(i),maxLG(i+1,dt),ord);
	}
	return r;
}
```

8. Acumulación de secuencias transformadas y filtradas mediante un operador binario

Dada una secuencia abstracta finita y un operador binario es posible pensar esquemas algorítmicos recursivos, no finales y finales, muy generales que acumulen los valores de la misma posiblemente transformados por una función $g(x)$ y filtrados por el predicado $p(x)$. Dependiendo de cómo caractericemos la secuencia y el operador binario tendremos varios tipos de esquemas.

Sea la secuencia abstracta definida por los valores:

$$(x_0, x_1, \dots, x_{b-1}, x_b)$$

Es decir hay un primer elemento x_0 y un último elemento x_b que lo hemos representado así porque tiene relación con el caso base de los esquemas recursivos. Este tipo de secuencias aparece de forma natural en los esquemas recursivos. Los sucesivos valores de la secuencia son las propiedades de los diferentes subproblemas alcanzados hasta llegar al último x_b asociado a las propiedades del caso base. En los esquemas recursivos normalmente se un valor calculado a partir de las propiedades del problema que denotaremos por $g(x)$ y finalmente, en muchos casos los valores resultantes posiblemente filtrados se acumulan para dar el

resultado. Según como caractericemos la secuencia, el tipo de operador para acumular los valores y si hay filtro o no resultan esquemas recursivos bastante generales. Veamos algunos de ellos.

Primer Esquema

Caracterizamos la secuencia abstracta por un primer elemento, x_0, el elemento siguiente de uno dado representado por la función $s(x)$, y un predicado $b(x)$ que nos indica si ese elemento es el último o no. Es decir $b(x_b)$ es verdadero y para los demás valores falso. La secuencia la podemos representar por $sq = (x_o, s(x), b(x))$. La definición anterior es la adecuada para secuencias no indexadas. Las indexadas se pueden modelar dentro del esquema anterior como un rango de enteros $i \in [0..b]$ más una función que asocie un valor a cada entero. Adicionalmente disponemos de un operador $\otimes_I$ asociativo por la izquierda y con elemento neutro por la izquierda e. Pretendemos calcular el valor:

$$g(x_0) \otimes_I \dots \otimes_I g(x_{b-1}) \otimes_I g(x_b)$$

Con esto elementos el esquema algorítmico final es:

$$f(x) = fg(x_0, e)$$

$$fg(x,a) = \begin{cases} a \otimes_I{}' x, & b(x) \\ fg(s(x), a \otimes_I{}' x), & !\, b(x) \end{cases}$$

$$a \otimes_I{}' x = \begin{cases} a \otimes_I g(x), & p(x) \\ a, & !\, p(x) \end{cases}$$

Y el correspondiente esquema iterativo es:

```
A f(T x){
        A a = e;
        while(!b(x)){
                if(p(x)) a = a⊗Ig(x);
                x = s(x);
        }
        a = a⊗Ig(x);
        return a;
}
```

Segundo Esquema

Alternativamente caractericemos la secuencia por $sq = (x_o, s(x), d(x))$. Dónde $d(x)$ es un predicado que es verdadero para todos los elementos de la secuencia dentro del dominio de la misma pero que es falso para el siguiente al último. Es decir es falso $d(s(x_b))$ y verdadero $d(x_i), i \in [0..b]$. Las secuencias indexadas se pueden modelar, igualmente, dentro del esquema anterior como un rango de enteros $i \in [0..b]$ más una función que asocie un valor a cada entero. Adicionalmente disponemos igualmente de un operador $\otimes_I$ asociativo por la izquierda y con elemento neutro por la izquierda e. Pretendemos calcular el valor:

$$g(x_0) \otimes_I \dots \otimes_I g(x_{b-1}) \otimes_I g(x_b)$$

Con estos nuevos elementos el esquema algorítmico final es:

$$f(x) = fg(x_0, e)$$
$$fg(x,a) = \begin{cases} a\,, & !\,d(x) \\ fg(s(x), a \otimes_I {}'x), & d(x) \end{cases}$$
$$a \otimes_I {}'x = \begin{cases} a \otimes_I g(x), & p(x) \\ a, & !\,p(x) \end{cases}$$

Y el correspondiente esquema iterativo es:

```
A f(T x){
        A a = e;
        while(d(x)){
                if(p(x)) a = a⊗Ig(x);
                x = s(x);
        }
        return a;
}
```

Este segundo esquema es más compacto que el anterior. Ahora elemento neutro es más claramente el valor acumulado de la secuencia vacía.

Tercer Esquema

Con la definición de secuencia $sq = (x_o, s(x), b(x))$ vista anteriormente y con un operador de asociativo por la derecha $\oplus_D$ con elemento neutro por la derecha e pretendemos calcular el valor acumulado:

$$g(x_0) \oplus_D \ldots \oplus_D g(x_{b-1}) \oplus_D g(x_b)$$

Con estos elementos el esquema algorítmico no final es:

$$f(x) = \begin{cases} e\,, & b(x) \wedge !\,p(x) \\ g(x)\,, & b(x) \wedge !\,p(x) \\ f(s(x)) \oplus_D {}'a, & !\,b(x) \end{cases}$$
$$x \oplus_D {}'a = \begin{cases} x \oplus_D a, & p(x) \\ a, & !\,p(x) \end{cases}$$

Cuarto Esquema

Ahora con la definición de secuencia $sq = (x_o, s(x), d(x))$ vista anteriormente y con un operador de asociativo por la derecha $\oplus_D$ con elemento neutro por la derecha e pretendemos calcular el valor acumulado:

$$g(x_0) \oplus_D \ldots \oplus_D g(x_{b-1}) \oplus_D g(x_b)$$

Con estos elementos el esquema algorítmico no final es:

$$f(x) = \begin{cases} e\,, & !\,d(x) \\ f(s(x)) \oplus_D{}' a, & d(x) \end{cases}$$
$$x \oplus_D{}' a = \begin{cases} x \oplus_D a, & p(x) \\ a, & !\,p(x) \end{cases}$$

Algunas consideraciones

Como podemos observar los esquemas segundo y cuarto definen la secuencia como un conjunto de valores que cumplen un predicado. No tratan como elemento distinguido el último elemento de la secuencia. Los esquemas primero y tercero definen la secuencia como un conjunto de valores hasta que se encuentra el último (el caso base). El valor asociado a la secuencia vacía será el elemento neutro e.

Los esquemas primero y segundo son esquemas finales (que como veremos más adelante son equivalentes a esquemas iterativos).

En muchos casos nos interesará transformar un esquema del tipo tercero o cuarto en otro de tipo primero o segundo. Si el operador es $\oplus_D$ es asociativo (por la derecha y por la izquierda) y tiene elemento neutro por la izquierda entonces los esquemas tercero y cuarto se transforman en el primero y segundo generalizando la función $f(x)$ a $fg(x,a)$. En este caso el operador $\otimes_{\mathrm{I}}$ es el mismo que el $\oplus_D$ pero toman los operandos en distinto orden. El operador $\otimes_{\mathrm{I}}$ toma primero el acumulador y luego el valor obtenido de las propiedades del problema y $\oplus_D$ toma el acumulador en segundo lugar. Esto es irrelevante si los operadores son conmutativos pero clave si no lo son.

Esquema quinto

Los esquemas recursivos simples (con un solo subproblema) son más generales que los anteriores al ser de esta forma:

$$f(x) = \begin{cases} sb(x)\,, & b(x) \\ c(f(s(x)), x), & !\,b(x) \end{cases}$$

O definiendo la secuencia mediante un dominio $d(x)$ y asociando un elemento neutro a la secuencia vacía. Los esquemas anteriores hemos asumido que $c(r,x) = r \oplus_D g(x)$ dónde r es el resultado de la llamada recursiva. Si la función de combinación es de una forma más general, en algunos casos, es posible definir una generalización $fg(x,y)$ y una nueva secuencia $s'(x,y) = (s(x), h(y))$ tal que el esquema anterior sea equivalente a:

$$f(x) = fg(x_0, h_0, e)$$
$$fg(x,y,a) = \begin{cases} a \otimes_I (x,y), & b(x) \\ fg(s(x), h(y), a \otimes_I (x,y), & !\, b(x) \end{cases}$$

O en la forma siguiente si la secuencia la definimos por un dominio:

$$f(x) = fg(x_0, h_0, e)$$
$$fg(x,y,a) = \begin{cases} e, & !\, d(x) \\ fg(s(x), h(y), a \otimes_I (x,y), & d(x) \end{cases}$$

Dónde los operadores $\oplus_D, \otimes_I$ pueden contener una etapa de filtro si es necesario. Para que los esquemas transformados y los originales sean equivalentes será necesario demostrarlo en cada problema particular. Veremos ejemplos más adelante.

9. Esquema recursivo con memoria

La recursividad puede ser de diferentes tipos. En primer lugar la vamos a clasificar según el número de sub-problemas en **recursividad simple** (también llamada **recursividad lineal**) cuando el número de sub-problemas es uno y **recursividad múltiple** cuando el número de sub-problemas es mayor que uno.

Un ejemplo de recursividad múltiple es el cálculo de los números de Fibonacci cuya definición recursiva es:

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

Vemos que cada problema puede ser identificado por un parámetro entero *n*. Este entero podemos tomarlo como el tamaño del problema. El dominio de problemas en el que estamos interesados es el conjunto de problemas con *n>=0*. La definición recursiva tiene dos casos base y dos sub-problemas y está bien en la medida que los sub-problemas tienen un tamaño menor que el problema original.

El algoritmo en C que es copia mimética de la definición recursiva es el de abajo.

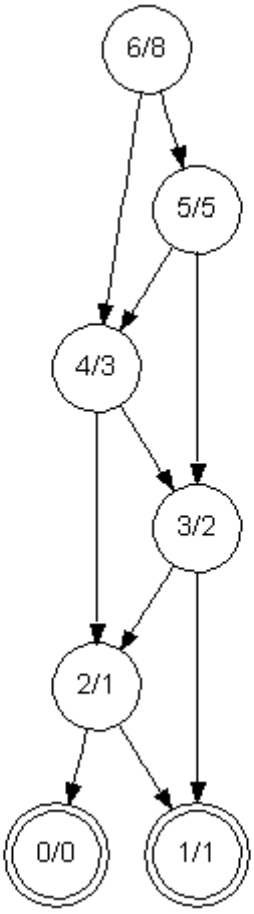
```
long fib1(int n){
	long r;
	if(n==0){
		r = 0;
	} else if(n==1){
		r = 1;
	} else {
		r = fib1(n-1) + fib1(n-2);
	}
	return r;
}
```

Hemos declarado el resultado de tipo *long* porque los números de *Fibonacci* crecen muy rápidamente y desbordan las posibilidades de un *int.* Incluso el tipo *long* se desborda para valores de *n* no muy grandes. Dejamos como ejercicio averiguar los sub-problemas, y la función de combinación.

Los números de Fibonacci calculados resultan ser *0, 1,1,2,3,5,8,13,21,34,55,89,144,233,377,….* Observemos en cualquier caso cómo se hacen las sucesivas llamadas recursivas. Hemos dibujado un grafo dirigido cuyos vértices tienen el número del problema y la solución correspondiente. Así *6/8* indica que la solución del problema *fib(6)* es *8*. Vemos como para calcular la solución *fib(6)* hay que calcular *fib(4)* y *fib(5).*

Vemos que con la forma del algoritmo recursivo se hacen muchos cálculos repetidos. Así *fib(3)* se hace repite tres veces cuando intentamos calcular *fib(6)* (el número de caminos de 6 a 3 en el grafo). No habría cálculos repetidos si el grafo anterior fuera un árbol.

El hecho de que haya tantos cálculos que se repiten da lugar a que el tiempo de ejecución del algoritmo crezca exponencialmente cuando aumenta el tamaño n como veremos en el tema siguiente.



Para evitar los cálculos repetidos podemos mejorar el esquema de divide y vencerás 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 de tipo *Map<E,R>* que guarde la solución *r* para cada problema *x* ya resuelto. En Java tenemos disponible ese tipo de datos. En *C* habría que disponer de una implementación.

De forma general un variable *m* de tipo *Map* dispone, entre otras, de tres funciones para gestionarla (métodos si estamos en Java):

- *boolean contains(m,x):* Ha sido resuelto el problema x?
- *r = get(m,x):* Solución del problema ya resuelto x.
- *put(m,x,r):* Se ha obtenido la solución *r* para el problema *x*.

En Java la notación sería *m.contains(x), m.get(x), m.put(x,r).*

```
R g(P p){
	Map<X,S> m = m0;
	return h(f(m,i(p)));
}
S f(X x, Map<X,R> m) {
	S r;
	LS s;
	LP y;
	assert(D(x));
	if(contains(m,x)){
		r = get(m,x);
	} else if(b(x)){
		r = sb(x);
		put(m,x,r);
	} else{
		y = sp(x);
		assert(t(y) < t(x));
		s = f_L(y,m);
		r = c(x,s);
		put(m,x,r);
	}
	return r;
}
```

El algoritmo para calcular los números de Fibonacci usando divide y Vencerás con Memoria los mostramos a continuación. La implementación es en Java. También usamos el tipo *BigInteger* disponible en Java para representar enteros tan grandes como queramos.

```
public static BigInteger fib(Integer n){
	Map<Integer,BigInteger> m = Maps.newHashMap();
	return fib2(n,m);
}

private static BigInteger fib2(Integer n, Map<Integer,BigInteger> m){
	BigInteger r;
	if(m.contains(n){
		r = m.get(n);
		m.put(n,r);
	} else if(n==0){
		r = BigInteger.ZERO;
```

```
        } else if(n==1){
              r = BigInteger.ONE;
        } else{
              r = fib2(n-1) + fib2(n-2);
              m.put(n,r);
        }
        return r;
}
```

En el capítulo siguiente estimaremos los tiempos de ejecución de *fib1* y *fib2*.

10. Algoritmos iterativos

En los capítulos anteriores hemos visto solamente algoritmos iterativos. Fundamentalmente hemos visto los que hemos llamado **Esquemas Secuenciales**. Ahora vamos a dar alguna notación y conceptos adicionales para el diseño y comprensión de los algoritmos iterativos. Aprenderemos, también, a obtenerlos a partir de algún tipo de algoritmos recursivos.

Como en el caso de los algoritmos recursivos aquí partimos de un problema y lo generalizamos para resolverlo más fácilmente. Como antes un problema tendrá un conjunto de propiedades (de distintos tipos), un tamaño, posiblemente un invariante y también usaremos un dominio del conjunto de problemas considerado.

Un **algoritmo iterativo** parte de un problema inicial, para cada problema actual busca un único problema siguiente hasta alcanzar un problema final. Resumiremos estas ideas en un **esquema iterativo.**

En los algoritmos iterativos se denomina **estado** al problema actual, **estado inicial** al problema inicial y **estado final** al problema final. Se llama **transición** al paso de un estado (problema) al siguiente.

Con los elementos anteriores el **esquema de un algoritmo iterativo** es:

```
R f(P p) {
        X x = i(p);
        assert(D(x));
        assert(I(x));
        while(g(x)){
              int n = t(x);
              assert(I(x));
              x ≐ s(x);
              assert(t(x) < n));
              assert(I(x));
        }
        return r(x);
}
```

Donde ≐ es el operador de asignación paralela que vimos más arriba.

Los algoritmos iterativos siguen el esquema anterior en el que se puede distinguir un bloque inicial, una condición del bucle y un cuerpo del bucle. El algoritmo iterativo sigue los siguientes pasos: inicializa el estado en el **bloque inicial** a uno que cumpla el invariante y esté dentro del dominio definido y mientras que no llegue al estado final, lo que comprueba con la **condición del bucle**, va haciendo transiciones de un estado al siguiente en el **cuerpo del bucle**. Tras cada transición exigimos que disminuya el tamaño del problema y siga cumpliéndose el invariante. Al terminar el bucle calcula la solución a partir del estado en la **sentencia final.**

En esquema iterativo puede aparecer un *for* en lugar del *while* o varios *for* o *while* anidados en lugar del *while*. El esquema es esencialmente el mismo.

En el esquema iterativo aparecen las siguientes funciones y variable (muchas compartidas con los algoritmos recursivos vistos anteriormente):

- *P* es el tipo de las propiedades del problema original
- *X* es el tipo de las propiedades del problema generalizado.
- *R* es el tipo de los resultados buscados. Puede ser uno o varios.
- *D(x):* Es una función lógica que especifica el dominio de conjunto de problemas generalizados
- *I(x):* Es una función lógica que especifica el invariante del problema generalizado.
- *t(x):* Es una función que especifica el tamaño del conjunto de problemas.
- *s(x):* Es una función que calcula las propiedades del siguiente problema. Es similar a la función *sp(x)* para el caso de un solo sub-problema.
- *i(p):* Es una función que calcula las propiedades del problema generalizado inicial a partir de los datos del problema original. Es similar a la instanciación del problema generalizado a partir de un problema generalizado.
- *g(x):* Es una función lógica verdadera si *x* no es el problema final. Es decir *g(x) = !f(x)*
- *r(x):* Es una función que calcula la solución a partir de los datos de un problema.

Los algoritmos iterativos están diseñados para que se cumpla el invariante al principio y al final del bucle. Cuando el bucle termina se cumple $I(x) \wedge \neg g(x)$

Diseño Iterativo

Con los elementos vistos arriba es posible ver los elementos que componen el diseño iterativo de un algoritmo. Estos elementos son:

- A partir de un problema inicial definido por las propiedades *p* de tipo *P* generalizamos el problema a otro con propiedades *x* de tipo *X*. Para precisar el conjunto de problemas de interés definimos un domino *D(x).* También indicamos un problema inicial (dónde comenzará el bucle) y un predicado f(x) que define el problema final (dónde termina el bucle).
- Diseñamos un Invariante *I(x)* para los problemas generalizados. Este invariante debe cumplirse antes de comenzar el bucle y tras terminar el mismo. Elegir adecuadamente este invariante es la parte más compleja del diseño iterativo.

- A cada uno de los problemas generalizados le asociamos un tamaño *t(m).* Este tamaño debe ser mayor o igual que cero para cada problema del dominio y debe decrecer en cada iteración. Esto es necesario para asegurar que el algoritmo termine.
- Tenemos que escoger una función siguiente *s(x)* que a partir de un problema nos de otro de tamaño menor y que también cumpla el invariante.
- Escogemos la función de instanciación *i(p)*. Es una función que a partir de las propiedades del problema original nos da un problema generalizado que cumple el invariante.
- Por último diseñamos la función *r(x)* que a partir de un problema generalizado calcula la solución del problema original.

Con esos elementos a partir del problema original se calcula uno generalizado que cumple el invariante y a partir de él se va pasando por problemas generalizados de tamaño cada vez menor que cumplen el invariante hasta que alcanzamos el problema final. Cuando esto ocurre hemos encontrado un problema final y que cumple el invariante. A partir de ahí es posible encontrar la solución.

Ejemplos

Factorial

En primer lugar un algoritmo iterativo para calcular la factorial. El problema original tiene las propiedades (n) y estamos interesados en los problemas $D(n) \equiv n \geq 0$. El problema generalizado tendrá las propiedades (i, a) con el invariante $I(i, a) \equiv a = i!$, el tamaño $t(i, a) = n - i$. La propiedades compartidas son n y las individuales (i, a). El problema inicial, el final y el resultado vienen dados por $i(n) = (0,1),\ f(i, a) \equiv i = n,\ r(i, a) = a$.

Para obtener el problema siguiente podemos hacerlo teniendo en cuenta que el invariante se debe cumplir al principio del cuerpo del bucle y al final y que decidimos que la variable i aumente en 1 para reducir el tamaño del problema $t(i, a) = n - i$. De esto tenemos:

$$\begin{matrix} a = i! \\ a' = i'! \\ i' = i + 1 \end{matrix} \equiv \begin{matrix} a = i! \\ a' = (i+1)! = (i+1) * i! \\ i' = i + 1 \end{matrix} \equiv \begin{matrix} a' = (i+1) * a \\ i' = i + 1 \end{matrix} \equiv$$

$$(i, a) \doteq (i + 1, a * (i + 1))$$

El problema siguiente es por lo tanto de la forma $s(i, a) = (i + 1, a * (i + 1))$. Recordemos que la asignación $(i, a) \doteq (i + 1, a * (i + 1))$ es una asignación paralela y se tiene que implementar con las ideas que comentamos más arriba. Como podemos comprobar la implementación correcta es $i = i + 1; a = a * i$. O alternativamente $a = a * (i + 1); i = i + 1$. Con esos elementos resulta el código escrito en C siguiente.

```
int factorial2(int n) {
      int i, a;
      i=0;
      a=1;
```

```
        while(i<n){
                i = i+1;
                a = a * i;
        }
        return a;
}
```

Maximo común divisor

```
int mcd2(int a, int b) {
        int r;
        while(b>0){
                r = a%b;
                a = b;
                b = r;
        }
        return a;
}
```

Como en el caso recursivo consideramos los problemas representados por los parámetros (a, b) con el dominio $D(a, b) = a \geq 0 \ \wedge \ b \geq 0 \wedge \neg(a = 0 \wedge b = 0)$. Ahora el problema original y el generalizado es el mismo. El invariante es $I(a, b) \equiv DC(a, b) = M$. Donde $DC(a, b)$ representa los divisores comunes de a, b. El problema siguiente es $sp(a, b) = (b, a\%b)$ y el tamaño $t(a, b) = b$.

Todos los problemas $(a, 0)$ son finales. Es conveniente convencerse de que se mantiene el invariante después de la transformación del estado. Es decir de pasar de un problema al siguiente. Esto es debido a una propiedad de los enteros: dos números a y b tienen los mismos divisores comunes que uno de ellos y el resto de su división.

Ordenar una lista

Veamos un algoritmo iterativo para ordenar una lista que representaremos como antes por (dt, n) cuyo dominio es $D(dt, n) \equiv n \geq 0$. Asumimos que la lista es de enteros y la queremos ordenar de menor a mayor. Asumimos dt y n variables compartidas.

Vamos a enfocar el problema mediante una aproximación simple que pueda ser usada en muchos casos aunque no sea la más eficiente. Es lo que podríamos denominar la solución obtenida a *fuerza bruta.*

Generalizamos el problema con dos parámetros i, j. El problema generalizado será (i, j, dt, n). Los problemas generalizados tienen $D(i, j, dt, n) \equiv i \geq 0 \wedge i < j \wedge \ i < n \wedge j < n.$ Los pares de casillas distintas (i, j) podemos organizarlos en forma de una secuencia tal que

$$s(i,j) = \begin{cases} (i, j+1), & j < n-1 \\ (i+1, i+2), & j = n-1 \end{cases}$$

En esta secuencia la casilla inicial es $(0,1)$ y la final $(n-2, n-1)$. Con esta secuencia definida cada problema (i, j, dt, n) pretende colocar en el orden correcto las casillas posteriores a i, j asumiendo el invariante que las anteriores ya están bien colocadas. El algoritmo es de la forma:

```
void sort2(int * dt, int n){
	int i, j;
	for(i = 0;i < n; i++){
		for(j = i+1; j < n; j++){
			if(dt[i] > dt[j]){
				it(dt,i,j);
			}
		}
	}
}
```

Este tamaño $t(i,j)$ es el número de iteraciones que falta para llegar a la última casilla de la secuencia. Este método de ordenación se conoce también como método de la burbuja.

Suma de los elementos de una lista

El problema original tiene las propiedades (ls, n). Es decir una lista de tamaño n. Estamos interesados en los problemas $D(n) \equiv n \geq 0$. El problema generalizado tendrá las propiedades (i, a, n) con el invariante $I(i,a) \equiv a = \sum_{x=0}^{i-1} ls[x]$, el tamaño $t(i,a) = n - i$. El problema inicial, el final y el resultado vienen dados por $i(n) = (0,0)$, $f(i,a,n) \equiv i = n$, $r(i,a) = a$.

El problema siguiente se puede obtener del invariante con las ideas vistas anteriormente:

$$\begin{matrix} a = \sum_{x=0}^{i-1} ls[x] \\ a' = \sum_{x=0}^{i'-1} ls[x] \\ i' = i+1 \end{matrix} \equiv \begin{matrix} a = \sum_{x=0}^{i-1} ls[x] \\ a' = \sum_{x=0}^{i'-1} ls[x] = \sum_{x=0}^{i} ls[x] = a + ls[i] \\ i' = i+1 \end{matrix} \equiv \begin{matrix} a' = a + ls[i] \\ i' = i+1 \end{matrix} \equiv$$

$$(i,a) \doteq (i+1, a+ls[i])$$

El problema siguiente es de la forma $s(i,a) = (i+1, a+ls[i])$. Recordemos que la asignación $(i,a) \doteq (i+1, a+ls[i])$ es una asignación paralela y se tiene que implementar con las ideas que comentamos más arriba. Como podemos comprobar, la implementación correcta es $a = a + ls[i]; i = i+1$. Con esos elementos resulta el código escrito en C siguiente.

```
int suma(int* ls,int n) {
	int i, a;
	i=0;
	a=0;
	while(i<n){
		a= a+ ls[i];
		i = i+1;
```

```
        }
        return a;
}
```

Aquí el problema generalizado tiene las propiedades (i, a). Un problema generalizado representa la sublista $[i, n)$ y a es la suma de loa elementos de la sublista $[0, i)$.

11. Diseño de tipos recursivos

A la vez que los algoritmos recursivos existen tipos recursivos. Es decir tipos tales que algunas de sus propiedades son del tipo que estamos definiendo. Normalmente los algoritmos asociados a los tipos recursivos son también recursivos. Veamos algunos ejemplos y una propuesta de diseño para esos tipos.

Veamos, en primer lugar, el tipo árbol binario, *BinaryTree<T>,* que podemos definir en una notación similar a la usada para diseñar algoritmos recursivos como:

$$t = \begin{cases} empty() \\ unary(e) \\ binary(e, t1, t2) \end{cases}$$

Donde hemos asumido que $t, t1, t2$ son árboles y e un objeto del tipo T. Es decir un árbol binario es un árbol vacío, un árbol con un solo elemento o un árbol compuesto por un elemento, un árbol $t1$ y otro árbol $t2$.

Junto con la definición anterior es conveniente definir predicados como $isEmpty(t)$, $isUnary(t)$, $isBinary(t)$ que, respectivamente, deciden si t es un árbol vacío, unario o binario. Adicionalmente si un árbol t es binario asumimos que ha sido construido de la forma $t = binary(e, t1, t2)$ y por lo tanto no podemos referir a sus elementos $e, t1, t2$. Igualmente para los caso unario.

Una forma de implementar la idea anterior es mediante una factoría con los métodos especificados.

Con esa definición del tipo podemos diseñar algoritmos recursivos que calculen propiedades derivadas del tipo como $size(t)$ que calcula el número de elementos que tiene el tipo y que puede definirse así:

$$size(t) = \begin{cases} 0, & isEmpty(t) \\ 1, & isUnary(t) \\ size(t1) + size(t2) + 1, & isBinary(t) \end{cases}$$

Una manera de diseñar los tipos recursivos es la siguiente:

- Diseñar una clase abstracta asociada al tipo y clases concretas que hereden de la anterior por cada caso particular del tipo. Cada clase concreta tendrá los atributos

necesarios para guardar las propiedades básicas de ese tipo concreto, un constructor para inicializar esas propiedades y una implementación específica de los métodos asociados a propiedades derivadas definidas en el tipo padre.

- Añadir a la clase abstracta métodos de factoría para crear cada uno de los casos particulares del tipo y un método abstracto por cada propiedad del tipo.

Veamos como ejemplo el código para implementar un árbol binario y para calcular la propiedad *size*.

```
public abstract class BinaryTree<T> {

	public static <T> Binary<T> binary(T label, BinaryTree<T> left,
			BinaryTree<T> right) {
		return new Binary<T>(label, left, right);
	}

	public static <T> Empty<T> empty() {
		return new Empty<T>();
	}

	public static <T> Unary<T> unary(T label) {
		return new Unary<T>(label);
	}

	public static <T> boolean isEmpty(BinaryTree<T> tree){
		return (tree instanceof Empty<?>);
	}

	public static <T> boolean isUnary(BinaryTree<T> tree){
		return (tree instanceof Unary<?>);
	}

	public static <T> boolean isBinary(BinaryTree<T> tree){
		return (tree instanceof Binary<?>);
	}

	public abstract int size();

}

public class Empty<T> extends BinaryTree<T> {

	Empty(){
		super();
	}

	@Override
	public String toString() {
		return "()";
	}

	public int size(){
		return 0;
	}
}

public class Unary<T> extends BinaryTree<T> {

	private T label;

	Unary(T label) {
		super();
```

```
		this.label = label;
	}

	public T getLabel() {
		return label;
	}

	public void setLabel(T label) {
		this.label = label;
	}

	@Override
	public String toString() {
		return "("+label+")";
	}

	public int size(){
		return 1;
	}

}

public class Binary<T> extends BinaryTree<T> {

	private T label;
	private BinaryTree<T> left;
	private BinaryTree<T> right;

	Binary(T label, BinaryTree<T> left, BinaryTree<T> right) {
		super();
		this.label = label;
		this.left = left;
		this.right = right;
	}
	public T getLabel() {
		return label;
	}
	public void setLabel(T label) {
		this.label = label;
	}
	public BinaryTree<T> getLeft() {
		return left;
	}
	public void setLeft(BinaryTree<T> left) {
		this.left = left;
	}
	public BinaryTree<T> getRight() {
		return right;
	}
	public void setRight(BinaryTree<T> right) {
		this.right = right;
	}
	@Override
	public String toString() {
		return "("+label+left+right+")";
	}

	public int size(){
		return this.left.size()+this.right.size()+1;
	}
}

public class TreeTest {

	public static void main(String[] args) {
		BinaryTree<Integer> t =
		   BinaryTree.binary(
				2,
```

```
                        BinaryTree.binary(
                            4,
                            BinaryTree.binary(
                                    7,
                                    BinaryTree.empty(),
                                    BinaryTree.unary(8)),
                            BinaryTree.unary(3)),
                        BinaryTree.unary(5));
            System.out.println(t);
            System.out.println(t.size());
        }
}
```

Otros ejemplos de tipos recursivos son las expresiones con valores enteros que podemos definir como:

$$ex = \begin{cases} cons(v) \\ unary(opu, ex) \\ binary(opb, ex1, ex2) \end{cases}$$

La expresión anterior sólo tiene constantes y operadores unarios y binarios sobre enteros. Podría ampliarse con variables de tipo entero. Una posible propiedad sería el cálculo del valor de la expresión.

Una lista también puede ser definida de forma recursiva como

$$ls = \begin{cases} empty() \\ unary(e) \\ add(ls1, e) \end{cases}$$

O como

$$ls = \begin{cases} empty() \\ unary(e) \\ add(e, ls1) \end{cases}$$

O incluso de la forma

$$ls = \begin{cases} empty() \\ unary(e) \\ concat(ls1, ls2) \end{cases}$$

Cada definición recursiva tendrá interés dependiendo del problema en cuestión. Teniendo en cuenta las definiciones recursivas anteriores de las listas podemos encontrar algoritmo recursivos para el cálculo de las propiedades de las secuencias.

12. Transformación recursivo-iterativa y entre diferentes tipos de recursividad

La recursividad puede ser de diferentes tipos. En primer lugar la vamos a clasificar según el número de sub-problemas en **recursividad simple** (también llamada **recursividad lineal**) cuando el número de sub-problemas es uno y **recursividad múltiple** cuando el número de sub-problemas es mayor que uno.

La recursividad lineal puede, a su vez, clasificarse en **recursividad lineal final** y **recursividad lineal no final**. Cuando la forma de la función de combinación es de la forma $c(x,r) = r$, es decir no depende de las propiedades del problema y siempre es igual a r, el resultado del sub-problema, entonces la recursividad lineal es final. En otro caso es no final.
Otra forma de distinguir los dos tipos de recursividad lineal: en la recursividad lineal final el resultado del sub-problema es el mismo que el del problema. En la recursividad lineal no final el resultado del problema se obtiene haciendo algún tipo de cálculo (la función de combinación) a partir del resultado del sub-problema y los parámetros del problema.

Una propiedad importante de la recursividad lineal final es que la solución del problema es la misma que la del caso base puesto que cada problema tiene la misma solución que el sub-problema al que se reduce. Este hecho será importante posteriormente.

Hemos visto ejemplos de recursividad lineal final y no final. Recordemos algunos algoritmos recursivos lineales finales y otros no finales.

Máximo común divisor

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

Factorial de n

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

Veamos ahora en primer lugar la transformación entre determinados algoritmos recursivos a otros iterativos y posteriormente algunas transformaciones de algoritmos recursivos.

Transformación de Recursividad Lineal Final a Iterativo

La más sencilla, por ser directa, es la transformación de un algoritmo recursivo lineal final a otro iterativo equivalente en el sentido de que resuelve el mismo problema. Normalmente tenemos un problema original, otro generalizado y una forma de instanciar el original a partir del generalizado. Con esos elementos el esquema de transformación es:

$$f(p) = g\big(i(p)\big), \qquad g(x) = \begin{cases} sb(x), & b(x) \\ g(sp(x)), & \neg b(x) \end{cases}$$

$$\equiv$$

$$x = i(p);\ while(!\,b(x))\{\ x \doteq sp(x);\}\ return\ sb(x);$$

En el esquema anterior el problema original es p, el generalizado es x, y la función de instanciación $i(p)$. La función de instanciación escoge el problema generalizado concreto que corresponde al problema original. Es decir calcula las propiedades del problema generalizado que corresponden a las del problema original. El algoritmo iterativo resultante es de la forma:

```
R f(P p){
        X x = i(p);
        while(!b(x)){
                x ≔ sp(x);
        }
        return sb(x);
}
```

Transformación de Recursividad Lineal no Final a Final mediante generalización con acumuladores y enfoque top-down

En algunos casos es posible transformar un algoritmo lineal no final en otro final (con el objetivo de obtener finalmente un algoritmo iterativo) generalizando el problema al añadir parámetros acumuladores. Solamente veremos algunas transformaciones básicas para casos concretos.

Como ya sabemos una definición recursiva lineal no final es la de la forma:

$$f(x) = \begin{cases} sb(x), & b(x) \\ c(x, f(sp(x)), & !\, b(x) \end{cases}$$

El esquema recursivo obtiene, en primer lugar, a partir del problema inicial los sucesivos problemas hasta alcanzar el caso base son: $x_0, x_1, \dots, x_b$. Estos problemas forman una secuencia cuyo primer elemento es x_0. Dado un elemento x el siguiente es $sp(x)$ y el último cumple $b(x_b)$. Estas ideas podemos representarlas de forma compacta como $[sp(x), x_0, b(x)]$. Posteriormente los elementos de la secuencia se van combinando, mediante la función de combinación $c(x,r)$ de forma ascendente. La solución del problema original es por tanto:

$$c\left(x_0, c\left(x_1, \dots, c(x_{b-1}, sb(x_b)\right)\right)\right)$$

En la función de combinación $c(x,r)$ las propiedades del problema vienen representadas por x y r es el valor devuelto por el subproblema.

En el caso de que la función de combinación tenga la forma $c(x,r) = g(x) \oplus_D r$ podemos ver que la solución del problema es:

$$c\left(x_0, c\left(x_1, \dots, sb(x_b)\right)\right) = g(x_0) \oplus_D g(x_1) \oplus_D \dots \oplus_D g(x_{b-1}) \oplus_D sb(x_b) = s$$

Como vemos se define implícitamente la secuencia

$$(g(x_0), g(x_1), \dots, g(x_{b-1}), sb(x_b))$$

Y el resultado s del problema se obtiene acumulando los valores de esa secuencia de derecha a izquierda con el operador $\oplus_D$ que asumimos que es asociativo por la derecha.

Si podemos imaginar una nueva función de combinación $c'(u,x) = u \otimes_I g(x)$ con un elemento neutro e tal que $e \otimes_I g(x) = g(x)$ y el operador $\otimes_I$ asociativo por la izquierda, entonces podemos escribir la expresión siguiente evaluada de izquierda a derecha:

$$s' = e \otimes_I g(x_0) \otimes_I \dots \otimes_I g(x_{b-1}) \otimes_I sb(x_b)$$

Si podemos demostrar que ambos resultados, s, s', son iguales disponemos de un esquema de transformación de recursividad no final a final de la forma:

$$f(x) = \begin{cases} sb(x), & b(x) \\ c(x, f(sp(x))), & \neg b(x) \end{cases} \equiv fg(x,u) = \begin{cases} c'(u, sb(x)), & b(x) \\ fg(sp(x), c'(u,x)), & \neg b(x) \end{cases}$$
$$f(x) = fg(x,e)$$

Si el operador original $\oplus_D$ es asociativo, conmutativo y con elemento neutro los operadores $c(x,r)$ y $c'(u,x)$ son exactamente iguales e igualmente los son los operadores $\oplus_D, \oplus_I$. Operadores que cumplen estas propiedades son entre otros: *suma, producto, and, or, mínimo, máximo, etc.* Pero si no es conmutativo entonces el esquema transformado usa $c'(u,x)$ en vez del original $c(x,u)$. Es decir en $c'(x,u)$ los parámetros u, x cambian de orden con respecto a su posición en $c(x,u)$. Operadores como la *concatenación de cadenas* son asociativos y con elemento neutro pero no conmutativos.

Hay que tener en cuenta que el problema original se obtiene instanciando el generalizado de la forma:

$$f(x) = fg(x,e)$$

Como podemos ver en cada problema generalizado $fg(x,u)$ el parámetro u es el valor acumulado asociado al prefijo de la secuencia recorrido $u = e \otimes_I g(x_0) \otimes_I g(x_1) \otimes_I \dots \otimes_I g(x_{i-1})$. Y en el caso concreto de del problema inicial $u = e$.

Las ideas tras esta transformación son muy similares a las que expusimos en el epígrafe de acumulación de los valores de una secuencia mediante un operador binario.

Aplicando esta transformación podemos encontrar una versión recursiva final para la factorial que ya vimos en el ejemplo 1.

$$fac(n) = \begin{cases} 1, & n = 0 \\ n * fac(n-1), & n > 0 \end{cases} \equiv facg(n,w) = \begin{cases} w * 1, & n = 0 \\ fg(n-1, w*n), & n > 0 \end{cases}$$

El problema original es $fac(n) = facg(n,1)$. Como podemos ver la secuencia de llamadas para $fac(4)$ y el papel del parámetro acumulador es:

$$fac(4) = facg(4{,}1) = facg(3{,}1*4) = facg(2{,}1*4*3) = fac(1{,}1*4*3*2)$$
$$= fac(0{,}1*4*3*2*1) = 1*1*4*3*2*1$$

Al final del capítulo veremos un ejemplo de este tipo de transformación.

Transformación bottom-up de Recursividad Múltiple (o de forma particular Simple) a Recursividad Final e Iterativa

Vamos a aplicar una transformación consistente en calcular la solución del problema recursivo de abajo arriba para conseguir, en determinados casos, una versión iterativa y posteriormente una recursiva lineal final. Es decir desde los casos base hasta el problema planteado. El esquema recursivo general con k sub-problemas es:

$$f(x) = \begin{cases} sb(x), & b(x) \\ c(x, f(sp_0(x)), f(sp_1(x)), \dots, f(sp_{k-1}(x))) & \neg b(x) \end{cases}$$

La idea es generalizar el problema incorporando los subproblemas, sus soluciones calculadas, establecer un invariante y usar las técnicas de diseño iterativo. La versión recursiva final se puede obtener de la versión iterativa.

Veamos como ejemplo el problema de Fibonacci

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

Definimos el problema generalizado con las propiedades (i, a, b) y establecemos el invariante $a = fib(i+1), b = fib(i)$. 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. El problema siguiente lo podemos deducir como en problemas anteriores:

$$\begin{matrix} a = fib(i+1) \\ b = fib(i) \\ a' = fib(i'+1) \\ b' = fib(i') \\ i' = i+1 \end{matrix} \equiv \begin{matrix} a = fib(i+1) \\ b = fib(i) \\ a' = fib(i+2) = fib(i+1) + fib(i) \\ b' = fib(i+1) \\ i' = i+1 \end{matrix} \equiv \begin{matrix} a' = a+b \\ b' = a \\ i' = i+1 \end{matrix} \equiv$$

$$(i, a, b) \doteq (i+1, a+b, a)$$

Y de aquí el esquema iterativo es:

```
long fib3(int n){
       long i, a, b;
       (i,a,b)≐ (0,1,0);
       while(i < n){
              (i,a,b)≐(i+1,a+b,a);
       }
       return b;
}
```

Desplegando la asignación paralela

```
long fib3(int n){
       long i, a, b, a0;
       i = 0;
       a = 1;
       b = 0;
       while(i < n){
              i = i+1;
              a0 = a;
              a = a0+b;
              b = a0;
       }
       return b;
}
```

La idea anterior se puede generalizar a un conjunto de parámetros tan grande como se quiera siempre que se usen los tipos de datos adecuados. Lo importante es que los nuevos parámetros $w'_0, w'_1, \dots, w'_m$ se puedan calcular a partir de los antiguos $w_0, \dots, w_m$ siempre asumiendo que se mantiene el invariante establecido y en particular $w_0 = f(x)$.

13. Conversión de entero a binario

Veamos, en primer lugar, algoritmos para obtener la representación binaria de un entero en una lista de caracteres y el valor entero asociado a una lista de caracteres binarios. Sea n es un número entero, + el operador suma de enteros, d un entero cero o uno, $b, b1$ listas de dígitos binarios, $+_L$ el operador de concatenación de listas, $[d]$ una lista unitaria que contienen los caracteres cero o uno y $[n]$ una lista unitaria que contienen la representación en caracteres de n suponiendo que $0 \leq n \leq 1$.

Una definición recursiva de una lista puede ser:

$$b = \begin{cases} [] \\ [d] \\ b1 +_L [d] \end{cases}$$

Es decir una secuencia binaria es vacía, está formada por un dígito binario o añadiendo un dígito binario a otra secuencia por la derecha. Para hacer operativa esta definición es necesario disponer de predicados para discriminar si es la primera opción (longitud de la

cadena cero, la segunda (longitud de la cadena 1) o la tercera segunda (longitud de la cadena mayor que 1) y de funciones que calculen $b1, d$ a partir de b. Dejamos esta tarea como ejercicio.

A partir de la definición recursiva anterior vamos a definir una función que calcule el valor entero de una secuencia de dígitos binarios. La definición es:

$$val(b) = \begin{cases} d, & si\ \ b = [d] \\ val(b1) * 2 + d, & si\ \ b = b1 +_L [d] \end{cases}$$

La definición es intuitiva si consideramos que el valor entero es un dígito binario es simplemente la conversión a entero del dígito y el valor de secuencia de dígitos binarios es el doble del valor de la secuencia prefijo más el valor el dígito final. La demostración se deja como ejercicio.

Para encontrar una versión recursiva final seguimos las ideas explicadas en el esquema quinto de la sección 8 con $h(r) = 2 * r$ el operador binario la suma de enteros y $g(b) = d$ siendo $b = b1 +_L [d]$ o $b = [d]$ y

$$val(b) = valg(b, 1, 0)$$
$$valg(b, y, a) = \begin{cases} a + y * d, & si\ \ b = [d] \\ valg(b1, 2 * y, a + y * d, & si\ \ b = b1 +_L [d] \end{cases}$$

La intuición tras esa transformación viene de la expresión

$$n = \sum_{i=0}^{p} d_i 2^i$$

La secuencia asociada al problema original es $s(b) = b1$ que podemos extender a $s'(b, y) = (b1, 2 * y)$. Como vemos las componentes de la secuencia (b, y) son los sucesivos prefijos de b y las potencias de 2 respectivamente. Ahora podemos definir un operador asociativo por la izquierda que acumula los valores transformados de la secuencia extendida en la forma $a \otimes_I (b, y) = a + y * d$.

Asumiendo que modelamos las secuencias de caracteres por un array s y un entero i. La versión iterativa correspondiente es:

```
int val(char * s, int t) {
	int i=n-1;
	int a =0;
	int y =1;
	while(i>0){
		a = a + y*s[i];
		i = i-1;
		y = 2*y;
	}
	a = a + y*s[i];
	return a;
}
```

Para convertir un entero a una lista binaria podemos partir de la definición de entero en la forma (pensada para este caso particular):

$$n = \begin{cases} 0 \\ 1 \\ \frac{n}{2} + 0 \\ \frac{n}{2} + 1 \end{cases}$$

El predicado para discriminar entre la tercera y cuarta forma es $n\%2 = 0$. Si es verdadero es la tercera y si no es la cuarta. O en forma más compacta, siendo d cero o uno:

$$n = \begin{cases} d \\ \frac{n}{2} + d \end{cases}$$
$$d = \begin{cases} 0, & n\%2 = 0 \\ 1, & n\%2 = 1 \end{cases}$$

Sobre esa definición de entero podemos definir la función $bin(n)$ que devuelva una lista con su representación binaria. Esta función es:

$$bin(n) = \begin{cases} [d], & n \leq 1 \\ bin\left(\frac{n}{2}\right) +_L [d], & n > 1 \end{cases}$$

Las definiciones anteriores pueden generalizarse a cualquier base. Esto se deja como ejercicio.

Transformemos ahora el esquema recursivo no finales a otro final. El operador $+_L$ es asociativo, con elemento neutro [] pero no conmutativo. El esquema transformado es:

$$bin(n) = bing(n, [])$$

$$bing(n, a) = \begin{cases} [d] +_L r, & n \leq 1 \\ bing\left(\frac{n}{2}, [d] +_L r\right), & n > 1 \end{cases}$$

Asumiendo que modelamos las secuencias de caracteres por un array s y un entero i. La versión iterativa correspondiente es:

```
char * val(char * s, int t, int n) {
	int i=t-1;
	char c;
	while(n>1){
		c = n%2;
		s[i] = c;
		i = i-1;
		n = n/2;
	}
	s[i] = c;
```

```
        return s+i;
}
```

En los problemas anteriores téngase en cuenta las siguientes propiedades de un número n de $p+1$ cifras representado en una base r:

$$n = a_p a_{p-1} \dots a_0 = \sum_{i=0}^{p} a_i r^i = \big((a_p r + a_{p-1})r + \cdots + a_1\big)r + a_0$$

$$a_0 = n\%r$$

$$a_i = \left(\frac{n}{r^i}\right)\%r$$

$$d(n) = \underline{\log_r n} + 1$$

$$a_p a_{p-1} \dots a_k = \frac{n}{r^k}$$

$$a_{k-1} a_{k-2} \dots a_0 = n\%r^k$$

Donde $d(n)$ calcula el número de dígitos de n en base r siendo $\underline{\log_r n}$ el entero más pequeño o igual a $\log_r n$. Por otra parte $a_p a_{p-1} \dots a_k$ es el prefijo del número y $a_{k-1} a_{k-2} \dots a_0$ el sufijo.

14. El problema de la potencia entera

El problema de la potencia entera es el cálculo de a^n donde n es un entero no negativo.

Dado su interés veamos diferentes algoritmos recursivos e iterativos para resolver el problema. El primer algoritmo se basa en las propiedades de la potencia entera:

$$a^n = \begin{cases} 1, & n = 0 \\ a a^{n-1}, & n > 0 \end{cases}$$

Estas propiedades permiten formular un algoritmo recursivo y transformarlo a iterativo. Ambos se dejan para ser resueltos como ejercicios.

La segunda posibilidad, más compleja pero que dará lugar a algoritmos más eficientes, se basa en las propiedades:

$$a^n = \begin{cases} 1, & n = 0 \\ (a^{n/2})^2, & n > 0, \quad n\ es\ par \\ a(a^{n/2})^2, & n > 0, \quad n\ es\ impar \end{cases}$$

Definiendo la función $h(n, a)$ como:

$$h(a, n) = \begin{cases} 1, & n\ es\ par \\ a, & n\ es\ impar \end{cases}$$

La definición puede ser compactada a:

$$a^n = \begin{cases} 1, & n = 0 \\ h(a,n) * (a^{n/2})^2, & n > 0 \end{cases}$$

Como podemos ver es una definición recursiva lineal no final. La función de combinación es: $c(a,n,r) = h(a,n) * r^2$.

Vamos a intentar encontrar un nuevo operador que nos permita reordenar los cálculos y obtener una versión recursiva final. Para ello seguimos las ideas del epígrafe 8. Observemos que recordando la expresión en binario de n tenemos:

$$n = \sum_{i=0}^{p} d_i 2^i$$

$$a^n = a^{\sum_{i=0}^{p} d_i 2^i} = \prod_{i=0}^{p} a^{d_i 2^i}$$

$$d_i = \begin{cases} 0, & n\%2 = 0 \\ 1, & n\%2 = 1 \end{cases}$$

$$a^{2^{i+1}} = a^{2^i 2} = (a^{2^i})^2$$

También podemos observar que a^{2^i} es la secuencia $(a, a^2, a^4, a^8, \dots)$ que podemos definir mediante el primer elemento a y un siguiente elemento $s(y) = y\text{^}2$. Combinando esta idea con la secuencia original del problema podemos definir la secuencia extendida $s'(n,y) = \left(\frac{n}{2}, y^2\right)$. Podemos ver entonces que el valor de la potencia es el valor acumulado de la secuencia extendida anterior mediante el operador binario $u \otimes_I (n,y) = u * a^{d_i} * y$ que teniendo en cuenta la expresión para a^{d_i} anterior podemos considerar que la secuencia está filtrada para los valores $a^{d_i} = 1$ es decir $n\%2 = 1$.

De donde obtenemos la solución recursiva final

$$pot(a,n) = pt(a,n,a,1)$$

$$pt(a,n,y,u) = \begin{cases} u, & n = 0 \\ pt(a, \frac{n}{2}, y^2, u), & n > 0, n\%2 = 0 \\ pt(a, \frac{n}{2}, y^2, u * y), & n > 0, n\%2 \neq 0 \end{cases}$$

La secuencia de llamadas para n = 9 es:

$$pot(a,9) = pt(a,9,a,1) = pt(a,4,a^2,a) = pt(a,2,a^4,a) = pt(a,1,a^8,a) \\ = pt(a,0,a^{16},a^9) = a^9$$

La definición propuesta es recursiva final. Ahora podemos obtener una solución iterativa. Esta solución es:

```
long pot(int a, int n){
       long r, u;
       y = a;
       u = 1;
       while( n > 0){
              if(n%2==1){
                   u = u * y;
              }
              y = y * y;
              n = n/2;
       }
       return u;
}
```

15. Corrección de los algoritmos

Corrección de algoritmos recursivos

Veamos primero la corrección de algoritmos recursivos. La corrección de un algoritmo de *Divide y Vencerás* se puede probar por inducción matemática: en primer lugar demostramos que sin correctas las soluciones de los casos base, posteriormente en los casos recursivos asumiendo que es correcta la solución de los problemas debemos demostrar que lo es la del problema y por último justificamos que los casos base son suficientes. Esto en el sentido que a partir de un problema dado y tras una secuencia de llamadas recursivas alcanzamos siempre los casos base indicados.

Veamos la definición de potencia entera. Pretendemos calcular a^n teniendo en cuenta el dominio $D(a,n) \equiv a > 0 \wedge n \geq 0$ y tamaño $t(a,n) = n$. En principio suponemos a, n enteros. La definición es:

$$a^n = \begin{cases} 1, & n = 0 \\ a, & n = 1 \\ \left(a^{\frac{n}{2}}\right)^2 cd(n,a), & n > 1 \end{cases}$$

Donde $cd(n,a) = (n\%2 = 0)?\, 1 : a$. Estamos asumiendo que $\frac{n}{2}$ es una división de números enteros con resultado entero y $n\%2$ es el resto de dividir n por 2. Es decir la función $cd(n,a)$ devuelve a si n es impar y 1 si es par.

Primero veamos los casos base. Por las propiedades de la potencia entera $a^0 = 1$, $a^1 = a$ concluimos que las soluciones de los casos base son correctas.

Para estudiar los casos recursivos tenemos que identificar en primer lugar el o los sub-problemas, su tamaño y la función de combinación de los resultados. A partir de aquí debemos comprobar que el tamaño de cada sub-problema es menor que el del problema y asumiendo que la solución del o de los sub-problemas es correcta entonces lo es la del problema.

Podemos comprobar que hay un sub-problema y que

$$sp(a^n) = a^{\frac{n}{2}}, \qquad c(a,n,r) = r^2 (n\%2 = 0)?\, 1 : a$$

Como caso concreto tenemos

$$sp(a^5) = a^2, \qquad c(a,5,r) = a\, r^2$$

Tenemos que justificar que si $r = a^{\frac{n}{2}}$ entonces $a^n = c(a,n,r)$ para $n > 1$. La justificación podemos obtenerla de las propiedades de potencia:

$$a^n = c(a,n,r) = \begin{cases} r^2, & n\%2 = 0 \\ a\, r^2, & n\%2 \neq 0 \end{cases} = \begin{cases} a^{2\frac{n}{2}}, & n\%2 = 0 \\ a^{2\frac{n}{2}+1}, & n\%2 \neq 0 \end{cases}$$

Y la de los números enteros

$$n = \begin{cases} 2\dfrac{n}{2}, & n\%2 = 0 \\ 2\dfrac{n}{2} + 1, & n\%2 = 0 \end{cases}$$

Por último partiendo de un problema de tamaño n entonces la secuencia de tamaño del problema y los sub-problemas es $n, \frac{n}{2}, \frac{n}{4}, \ldots, 0$. Como vemos sería suficiente con le caso base de tamaño 0. El caso base de tamaño 1 se ha añadido por eficiencia aunque no es estrictamente necesario.

Corrección de algoritmos iterativos

Para justificar la corrección de los algoritmos iterativos usamos el invariante definido sobre el problema generalizado. Hemos de justificar que el problema inicial cumple el invariante, suponiendo que un problema generalizado cumple el invariante hemos de justificar que también lo cumple el problema siguiente (para ello es más cómodo poner el bloque básico, o generalizado, del en forma de asignación paralela) y por último concluir de las propiedades de la solución del problema se derivan del invariante, la negación de la guarda y la expresión final.

Veamos el caso de fib3 visto arriba.

```
long fib3(int n0){
       long r1, r2;;
       int n;
       (n,r1,r2)≔ (1,1,0);
       while(n < n0){
              (n,r1,r2)≔(n+1,r1+r2,r1);
       }
       return r1;
}
```

Como dijimos el invariante y el problema siguiente son:

$$I(n, r1, r2) \equiv r1 = fib(n) \wedge r2 = fib(n-1)$$

$$s(n, r1, r2) = (n+1, r1+r2, r1)$$

Tenemos que justificar que si $I(n, r1, r2)$ es verdadero también los es $I(n, r1, r2)[n|n+1, r1|r1+r2, r2|r1]$.

Es decir

$$r1 = fib(n) \wedge r2 = fib(n-1) \rightarrow r1 + r2 = fib(n+1) \wedge r1 = fib(n)$$

Como vemos lo anterior se deriva de las propiedades de los números de Fibonacci. Por último al finalizar el bucle se cumple

$$I(n, r1, r2) \wedge !\,(n < n0) \equiv r1 = fib(n) \wedge r2 = fib(n-1) \wedge n = n0$$

Por lo que concluimos que el resultado es correcto.

16. Problemas propuestos

Para los problemas siguientes diseñar algoritmos recursivos de distintos tipos y algoritmos iterativos, directamente o deducidos de los recursivos.

1. Dar una definición recursiva para la potencia entera de base a, es decir a^n, apoyándose en las siguientes propiedades de la misma: $a^0 = 1$, $a^n = aa^{n-1}$. Encontrar a partir de la definición anterior una definición recursiva lineal final y posteriormente un algoritmo iterativo.
2. Obtener un algoritmo recursivo para la potencia entera de base a, es decir a^n, apoyándose en las siguientes propiedades de la misma: $a^0 = 1$, $a^n = cd(n, a)(a^{n/2})^2$. Donde la función $cd(n, a)$ devuelve a si n es impar y 1 si es par. ¿Qué tipo de definición recursiva hemos hecho?
   En las definiciones recursivas sobre listas o arrays siguientes considerar subproblemas de tamaño mitad y, alternativamente, subproblemas de tamaño una unidad menor.
3. Dar definiciones recursiva para el producto escalar de dos vectores de tamaño n.
4. Dar una definición recursiva para la suma de las casillas de un vector de tamaño n.
5. Dar una definición recursiva para el producto de las casillas de un vector de tamaño n.
6. Dar una definición recursiva para el máximo de las casillas de un vector de tamaño n.
7. Dar una definición recursiva para decidir si todas la casillas de un vector de enteros de tamaño n son pares.
8. Dar una definición recursiva para decidir si existe alguna casilla de un vector de enteros de tamaño n que sea par.
9. Dado dos listas diseñar un algoritmos recursivo para decidir si son iguales
10. Dada una lista de enteros diseñad un algoritmo recursivo para decidir si el valor de alguna casilla es igual al valor del índice.
11. Diseñar algoritmos para encontrar la inversa de una cadena de caracteres.
12. Dada una cadena de caracteres dar una definición recursiva que decida si es un palíndromo. Una cadena es un palíndromo si es igual a su inversa.

13. Diseñar un algoritmo para obtener la representación en binario de un entero. La solución debe ser una cadena de caracteres '0' y '1'.
14. Diseñar un algoritmo iterativo para encontrar los números de *Fibonacci*. Posteriormente encontrar un algoritmo iterativo mediante la adaptación del algoritmo de la potencia entera. Los números de *Fibonacci* cumplen la propiedad $f_n = f_{n-1} + f_{n-2}, f_1 = 1, f_0 = 0$
15. Dada la siguiente propiedad de la multiplicación entera

$$xy = \begin{cases} 2x\left(\frac{y}{2}\right), & si\ y\ es\ par \\ 2x\left(\frac{y}{2}\right) + x, & si\ y\ es\ impar \end{cases}$$

Encontrar una definición recursiva para la misma y posteriormente diseñar un algoritmo iterativo.

16. Dada la siguiente propiedad de la multiplicación entera

$$ab = \begin{cases} a + a(b-1), & b > 0 \\ 0, & b = 0 \end{cases}$$

Encontrar una definición recursiva para la misma y posteriormente diseñar un algoritmo iterativo.

17. Diseñar un algoritmo iterativo, 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.
18. Diseñar un algoritmo iterativo, con y sin memoria, y posteriormente encontrar un algoritmo iterativo que calcule los valores de la recurrencia $f_n = 4f_{n-1} + f_{n-2} + f_{n-3}, f_2 = 1, f_1 = 1, f_0 = 2$ mediante el cálculo de abajo arriba de los valores de la recurrencia.
19. Dar un definición recursiva y obtener un algoritmo recursivo para el cálculo de un número combinatorio basándose en las propiedades

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

20. Dar una definición recursiva para obtener el máximo de los elementos de un array genérico sobre un orden que se pasa como parámetro. Encontrar el algoritmo iterativo correspondiente en Java y para un tipo concreto en C.
21. Dar una definición recursiva para decidir si todos los elementos de un array genérico cumplen una propiedad que se pasa como parámetro. Encontrar el algoritmo iterativo correspondiente en Java y para un tipo concreto en C.
22. Ordenar un array genérico con respecto a un orden que se pasa como parámetro adaptando el algoritmo del ejemplo 5 (transformado a iterativo) y 9. En Java.
23. Buscar un elemento dado en una lista. Asumir, primero que la lista no está ordenada. En segundo lugar que está ordenada. Diseñar algoritmos recursivos e iterativos.
24. Diseñar un algoritmo para decidir si una lista está ordenada con respecto a un orden.

25. Diseñar un algoritmo para decidir si los elementos de una lista de enteros forman una progresión aritmética.
26. Diseñar un algoritmo iterativo para fusionar dos listas ordenadas en otra también ordenada. Obtener una definición recursiva del problema.
27. Dar una definición recursiva de una lista.
28. Dar una definición recursiva de un árbol binario.
29. Dar una definición recursiva de las sucesivas potencias de 2.
30. Dar una definición recursiva de un número primo.