Capítulo 10

Recursividad

En este capítulo se introducen los algoritmos recursivos y su implementación como métodos recursivos. Mediante la recursividad es posible efectuar la repetición de algunos cálculos sin utilizar estructuras de iteración explícitas. Por ello y debido además a que los algoritmos recursivos pueden facilitar la solución de algunos problemas, la recursividad constituye una herramienta fundamental en la resolución de múltiples problemas computacionales.

En la naturaleza y en las matemáticas es frecuente encontrar ejemplos en los que una definición de objetos por enumeración es complicada. Un ejemplo típico es la definición del conjunto de los números naturales que, al ser infinito, es imposible hacer por enumeración. Los axiomas de Peano, basándose en la definición de *sucesor*, establecen los términos básicos de la definición de un número natural:

1. El 1 es un número natural.
2. Si n es un número natural, entonces el sucesor de n también es un número natural.

Como se puede ver, la generalidad de la definición de número natural se apoya a su vez en la definición de número natural (además de en la de sucesor). Sólo el caso del número 1 se define por sí mismo.

A este tipo de definiciones, en las que un concepto se define basándose en sí mismo (excepto posiblemente para una serie de casos triviales), se les denomina *definiciones recursivas*. También se suele hablar de *razonamiento inductivo*, ya que a partir de unos casos sencillos se es capaz de encontrar una definición general que se apoya en un caso más simple; en el ejemplo de los naturales, el caso más sencillo de partida es el del número 1, y utilizándolo es posible generar por inducción, empleando la definición de sucesor, el resto de números naturales.

La *definición recursiva* o *inductiva* de la solución de un problema se caracteriza por que, a partir de un conjunto finito de casos sencillos del problema para los que se conoce la solución, se realiza una *hipótesis* de cuál es la solución general de dicho problema:

- La *hipótesis de inducción* para el caso general se expresa en términos de la(s) solución(es) del mismo problema para el(los) caso(s) más sencillo(s).
- Debe ser validada o demostrada mediante un proceso de *prueba por inducción*, sin el cual no deja de ser más que una hipótesis.

De igual manera que hay definiciones recursivas, hay problemas que pueden resolverse mediante el uso de *algoritmos recursivos*. La ventaja que presenta el uso de algoritmos recursivos es que se definen de forma inductiva, algo que en muchos casos es una ventaja frente a la formulación deductiva de un problema. Así, ante un problema complejo en ocasiones es mucho más sencillo formular su solución de forma inductiva (en términos de soluciones para casos más simples del problema) que de forma deductiva (por enumeración de los pasos necesarios para obtenerla).

La idea clave es que para resolver un problema complejo mediante un algoritmo recursivo se descompone el problema en una serie de problemas más simples que se resuelven *empleando el mismo algoritmo*, para luego componer la solución del problema complejo en base a las soluciones de los problemas más simples. Evidentemente, llegará un momento en que el problema se habrá simplificado lo suficiente como para que su solución sea inmediata, lo cual pone fin a la resolución recursiva y da la finitud necesaria para que el proceso sea considerado algorítmico (recuérdese que todo algoritmo se caracteriza por ser finito).

Tanto en definiciones matemáticas como en algoritmos se puede ver que hay dos situaciones distintas a la hora de resolver el problema planteado:

- El problema o dato es lo suficientemente simple para poder ser resuelto de manera trivial (en el caso de los números naturales, es así para el 1); este caso (o conjunto de casos) se denomina *caso base*.
- El problema o dato requiere del uso de la resolución de una instancia más simple para hallar su solución; a este caso se le llama *caso general*.

Una definición recursiva, al igual que un algoritmo recursivo, puede presentar varios casos base y casos generales, aunque de momento la mayor parte de los ejemplos que se expondrán tendrán un único caso base y caso general.

Una vez planteado un algoritmo recursivo es necesario verificar:

1. La *terminación* del algoritmo, es decir, que en algún momento se llegará al caso base a partir de cualquier caso general planteado inicialmente.
2. La *corrección* del algoritmo, es decir, que para cualquier subproblema que se dé, la solución del algoritmo es correcta (que suele requerir una demostración matemática por *inducción* sobre algún parámetro del algoritmo).

10.1 Diseño de un método recursivo

A la hora de resolver un problema recursivo, se suele partir de una formulación formal del mismo que plantea la recursividad. Un ejemplo clásico es la definición recursiva de la función factorial ($n!$) de un número n, entero mayor o igual que 0, que sigue la siguiente recurrencia:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

A partir de esta formulación, se debe conseguir un algoritmo que resuelva el problema de obtener el factorial de un número n (entero mayor o igual que 0). La implementación de estos algoritmos recursivos en Java requiere el uso de *métodos recursivos*. Un método recursivo se caracteriza por ser un método donde se ejecuta al menos *una llamada a sí mismo*, generalmente variando los parámetros de la llamada.

Un método recursivo debe constar en su cuerpo de una o varias condiciones que permiten distinguir en qué caso (base o general) se encuentra y una o varias instrucciones que permiten obtener la solución del caso correspondiente. A continuación, se muestra una posible implementación recursiva del factorial, indicando qué partes del código constituyen cada una de las partes generales de un algoritmo recursivo.

```
/** n >= 0. */
public static int factorial(int n) {
    if (n == 0) {                    // Condición del caso base
        return 1;                    // Instrucciones del caso base
    }
    else {
        return n * factorial(n - 1); // Instrucciones del caso general
    }
}
```

Así pues, a la hora de diseñar un método recursivo, han de tenerse en cuenta:

- Las condiciones de los distintos casos base y las condiciones de los distintos casos generales.
- Las acciones de los distintos casos base (que pueden ser vacías en algunos casos) y las acciones de los distintos casos generales (que como mínimo deben de incluir una llamada al mismo método, lo que dará la naturaleza recursiva).

Todo método recursivo debe tener un caso base que garantice el fin de la recursividad. Además, las llamadas que se producen en el caso general deben de garantizar que se van acercando cada vez más, de forma estricta, a la condición del caso base, a fin de garantizar la finitud del algoritmo. Todo esto lleva a plantear que los métodos recursivos siguen un esquema de código semejante al que sigue:

```
[modificadores] [static] TipoRetorno nomMetRecursivo(listaParams) {
    TipoRetorno resMet, res1, res2, ..., resK;
    if (casoBase(listaParams)) {
        resMet = solucionBase(listaParams);
    }
    else {
        res1 = nomMetRecursivo(anterior1(listaParams));
        res2 = nomMetRecursivo(anterior2(listaParams));
        ...
        resK = nomMetRecursivo(anteriorK(listaParams));
        resMet = combinar(listaParams, res1, res2, ..., resK);
    }
    return resMet;
}
```

en donde:

- `listaParams` es la lista de parámetros del método recursivo.
- `casoBase` verifica si la condición del caso base es cierta.
- `solucionBase` obtiene la solución trivial para un caso base.
- `anterior`i indica la descomposición del caso actual (`listaParams`) en el i-ésimo caso más sencillo (estrictamente más cercano al caso base que el actual).
- `combinar` indica la recombinación de las soluciones de los casos más sencillos para obtener la solución del caso actual.

Como se puede ver, la estructura fundamental consiste en tener una instrucción condicional que permita distinguir entre caso base y general. Por otro lado, las llamadas recursivas deben efectuarse cada vez para casos estrictamente más simples (los métodos `anterior`i tienen que garantizar que los datos sobre los que se opera estén estrictamente más cercanos al caso base que los de la llamada previa al método recursivo).

Generalmente, cuando se desea resolver un problema utilizando un algoritmo (o método) recursivo se consideran las siguientes etapas:

1. Enunciar completamente el problema, declarando la cabecera del método que se va a construir y estableciendo tanto las *condiciones de entrada* para las que deberá ejecutarse el algoritmo, como el *resultado esperado* de dicha ejecución.

   Además de por motivos obvios, la definición del problema es importante porque a menudo durante la misma se pone de relieve la estructura recursiva que puede darse en el problema, o se facilita su determinación.

2. Análisis de casos. Consiste en hacer explícito el caso base y el caso general de la recursividad, estableciendo para cada caso las instrucciones pertinentes para resolver el problema. Hay que comprobar que se cubre cualquier caso posible, esto es, que ante cualquier posible entrada del problema, se efectúa el caso base o el general.

3. Transcribir el algoritmo que se está describiendo a un método del lenguaje utilizado.

4. Determinar que en cada nueva llamada recursiva se cumple que:

   - El nuevo problema que se debe resolver es estrictamente más cercano al caso base que el problema original, y
   - los datos de entrada de la nueva llamada cumplen las condiciones de entrada en las que se ha establecido que se deberá ejecutar el algoritmo.

10.2 Tipos de recursividad

En un algoritmo (o método) recursivo, alguna de sus instrucciones hace referencia directa o indirecta a si mismo. Esto es, alguna de sus instrucciones es una llamada explícita a si mismo (en cuyo caso se denomina *recursividad directa*), o bien es una llamada a otro algoritmo, a través del cual puede llegarse a hacer una llamada al primero (denominándose entonces *recursividad indirecta*). Aunque ambas situaciones son posibles, este capítulo se centra en los algoritmos (o métodos) recursivos directos.

En función de cuántas llamadas recursivas se hacen y de cómo se recombinan las soluciones de los casos más simples se pueden distinguir varias categorías de algoritmos recursivos. Estas categorías suelen relacionarse con su complejidad computacional (capítulo 11) y la facilidad de convertirlos en algoritmos iterativos equivalentes. La clasificación es la siguiente:

- *Recursividad lineal*: hay a lo sumo una sola llamada recursiva en cada ejecución del algoritmo, generando una *secuencia de llamadas* (figura 10.1). Según la recombinación usada, a su vez, se clasifica en:
  - *Final*: el resultado de la llamada recursiva es el propio resultado de la llamada actual; es decir, la solución del caso más simple es a su vez la del caso más complejo, con lo que no hay combinación de resultados.
  - *No final*: el resultado de la llamada recursiva se emplea para calcular el resultado de la llamada actual, arrojando un resultado posiblemente distinto al de la llamada recursiva.
- *Recursividad múltiple*: hay más de una llamada recursiva en cada ejecución del algoritmo y sus resultados han de combinarse para obtener la solución del caso actual. Se genera un *árbol de llamadas* (figura 10.3).

Atendiendo a esta clasificación, el método de cálculo del factorial que se ha presentado es un método lineal (se hace una única llamada recursiva a `factorial(n - 1)` en cada ejecución) no final (el resultado de la llamada se multiplica por `n` para devolverse). En la figura 10.1 se muestra gráficamente la secuencia de llamadas realizadas al método `factorial`, a partir de la llamada inicial `factorial(4)`. La siguiente es una traza de la ejecución del método para calcular 4!, en donde se indica el orden en el que se producen y el orden en el que finalizan las llamadas recursivas ejecutadas, así como el resultado de cada llamada.

Llamadas ejecutadas	Orden de ejecución	Orden de finalización	Resultado
`factorial(4)`	1	5	4 * `factorial(3)` = 24
`factorial(3)`	2	4	3 * `factorial(2)` = 6
`factorial(2)`	3	3	2 * `factorial(1)` = 2
`factorial(1)`	4	2	1 * `factorial(0)` = 1
`factorial(0)`	5	1	1

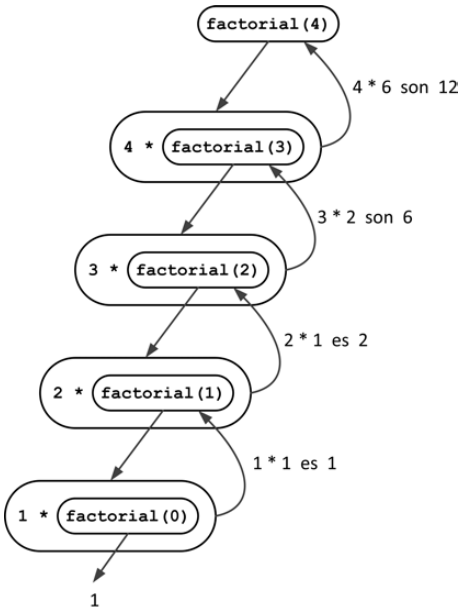


Figura 10.1: Secuencia de llamadas del método recursivo `factorial`.

10.3 Recursividad y pila de llamadas

La ejecución de un método recursivo es similar a la de cualquier otro método, aunque conviene tener presente que en el caso recursivo un método efectuará llamadas a si mismo. Las características más relevantes de dicha ejecución pueden resumirse del modo siguiente:

- Cada vez que comienza la ejecución de un método, debido a una llamada al mismo, se apila un registro de activación en la pila de llamadas. Nótese que cada una de las diferentes ejecuciones recursivas de un mismo método está, por lo tanto, asociada a un registro de activación propio. Es decir, existen tantos registros de activación como llamadas pendientes. De todos ellos, en cada momento sólo hay uno activo, el que está en el tope de la pila.
- Un método puede, durante su ejecución, modificar la información local asociada al mismo sin que, por ello, quede alterada la información local asociada a otras llamadas pendientes, incluso del mismo método. Recuérdese que el paso de parámetros en Java se efectúa siempre por valor.
- Cuando una ejecución de un método finaliza, se liberan los recursos propios de dicha llamada. En particular, deja de existir su registro de activación (se desapila). La ejecución del programa pasa a reanudarse a partir de la

instrucción en la que se efectuó la llamada al método cuya ejecución ha finalizado. Nótese que en el caso recursivo, la ejecución puede reanudarse en una ejecución inmediatamente anterior del mismo método, que habrá dejado de estar pendiente.

- La información se transmite entre las distintas ejecuciones de los métodos a través de los parámetros cuando se efectúa la llamada, o por los valores devueltos cuando se efectúa el retorno.

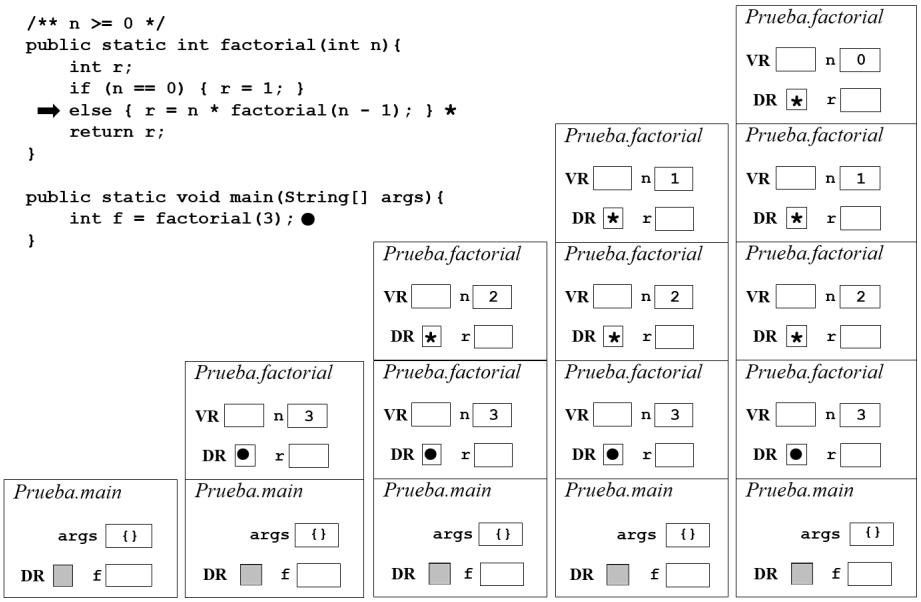
Todo esto acarrea un uso intensivo de la pila de llamadas que, en ocasiones, crece de manera extraordinaria y puede llegar a provocar serios problemas por agotamiento de la memoria, produciéndose un *desbordamiento de la pila* (en inglés, *stack overflow*). La causa habitual del desbordamiento de la pila es la recursividad infinita. Si un método realiza infinitas llamadas recursivas, en algún momento el tamaño de los registros de activación apilados llegará a sobrepasar el tamaño de la pila, provocando la excepción `StackOverflowError`.

Un ejemplo de cómo se comporta la pila para la ejecución del método `factorial(int)` presentado previamente se muestra en la figura 10.2. Para facilitar la discusión, se ha reescrito el código anterior de la siguiente forma:

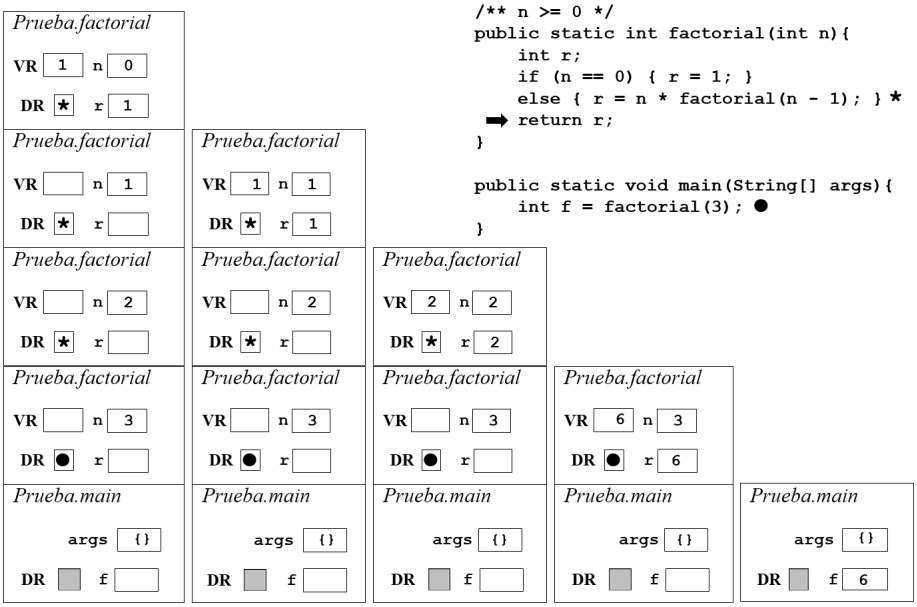
```
/** n >= 0. */
public static int factorial(int n) {
    int r;
    if (n == 0) { r = 1; }
    else { r = n * factorial(n - 1); }
    return r;
}
```

En la figura 10.2(a) se muestran los estados de la pila tras cada llamada al método `factorial(int)`. Se puede observar como va creciendo la pila al apilarse un registro de activación por cada llamada al método `factorial(int)`. El primer estado se corresponde con el registro de activación del método `main`, justo antes de la llamada a `factorial(3)`. A continuación, cada columna representa la secuencia de registros de activación cada vez que se detiene la ejecución del método debido a su punto de ruptura (representado por ➡). En el último estado, el registro activo (el de la cima de la pila) se corresponde a la llamada a `factorial(0)`, es decir, la del caso base.

En la figura 10.2(b) se muestran los estados de la pila justo antes de devolver el resultado de cada llamada a `factorial(int)`. La pila va decreciendo al ir desapilándose los registros de activación a medida que se van resolviendo las llamadas al método `factorial(int)`, hasta llegar al estado en el que el único registro de la pila es el correspondiente al método `main` y se ha obtenido el resultado de `factorial(3)`.



(a) Estados de la pila tras cada llamada a `factorial` hasta alcanzar el caso base.



(b) Estados de la pila justo antes de devolver el resultado de cada llamada a `factorial`.

Figura 10.2: Evolución de la ejecución y la pila de llamadas.

Si se compara el uso de la pila que provoca la llamada `factorial(n)` en la versión iterativa y recursiva del método, se puede concluir que dicho uso es mayor en el caso recursivo que en el iterativo. En la pila de llamadas de la versión iterativa de `factorial(int)` coexisten simultáneamente en memoria, como mucho, el registro de activación del método `factorial(int)` y el registro de activación del método `main`. En la pila de llamadas de la versión recursiva de `factorial(int)` pueden llegar a coexistir simultáneamente `n` +1 registros de activación de las distintas llamadas a `factorial(int)` más el registro de activación del método `main`. Es decir, el consumo de memoria del método `factorial(int)` iterativo es siempre el mismo mientras que el del `factorial` recursivo depende del valor de `n`.

10.4 Algunos ejemplos

Por simplicidad, los métodos que se presentan en esta sección son métodos estáticos; pero un método recursivo puede ser también un método de instancia.

Potencia n-ésima

Dado un número natural $n \geq 0$ y un número real $a \neq 0$, la potencia a^n puede definirse de forma recursiva mediante la recurrencia:

- Si $n = 0$, entonces $a^n = 1$.
- Cuando $n > 0$, entonces $a^n = a^{n-1} * a$.

Con esta recurrencia, un posible método recursivo que lo soluciona sería:

```
/** n >= 0 y a != 0. */
public static double potencia(double a, int n) {
    if (n == 0) { return 1; }
    else { return potencia(a, n - 1) * a; }
}
```

La condición del caso base es `n == 0` y la del caso general es la condición contraria (de ahí que se use el `else`). La acción del caso base consiste en devolver el valor 1. La acción del caso general consiste en devolver el resultado de la llamada recursiva multiplicado por `a` (es decir, se trata de una recursividad lineal no final), cambiando el valor del segundo parámetro por `n - 1`. Esto hace que en cada llamada el valor del segundo parámetro vaya decreciendo y se garantiza que en algún momento llegará a ser 0, alcanzando el caso base y finalizando el algoritmo. Esta prueba informal de terminación debe demostrarse matemáticamente para garantizar la finitud del algoritmo, al igual que debe hacerse con su corrección.

A continuación, se muestra una traza de la ejecución del método para calcular 2^4.

Llamadas ejecutadas	Orden de ejecución	Orden de finalización	Resultado
`potencia(2, 4)`	1	5	`potencia(2, 3)` * 2 = 16
`potencia(2, 3)`	2	4	`potencia(2, 2)` * 2 = 8
`potencia(2 ,2)`	3	3	`potencia(2, 1)` * 2 = 4
`potencia(2, 2)`	4	2	`potencia(2, 0)` * 2 = 2
`potencia(2, 0)`	5	1	1

Al tratarse de un tipo de recursividad lineal no final, cuando una llamada recursiva finaliza, se reanuda la ejecución en el punto del método en el que se efectuó la llamada, realizándose aún operaciones posteriores en dicho método para poder devolver el resultado y finalizar.

Resto de la división entera

Dados dos números naturales $a \geq 0$ y $b > 0$, el resto de su división entera a/b puede definirse de forma recursiva mediante la recurrencia siguiente:

- Si $a < b$, el resto de a/b es a.
- En otro caso, el resto de a/b es el resto de $(a - b)/b$.

El siguiente es un posible método recursivo que implementa esta recurrencia:

```
/** a >= 0 y b > 0. */
public static int resto(int a, int b) {
    if (a < b) { return a; }
    else { return resto(a - b, b); }
}
```

La condición del caso base es `a < b` y la del caso general es la contraria. La acción del caso base consiste en devolver el valor `a`. La acción del caso general consiste en devolver el resultado de la llamada recursiva (es decir, se trata de una recursividad lineal final), cambiando el valor del primer parámetro a `a - b`. Así, en cada llamada el valor del primer parámetro va decreciendo y se garantiza que en algún momento será inferior al segundo parámetro, alcanzando el caso base y finalizando el algoritmo. Esta prueba informal de terminación debe demostrarse matemáticamente para garantizar la finitud del algoritmo, al igual que debe hacerse con su corrección.

A continuación, se muestra una traza de la ejecución del método para los valores 75 y 18.

Llamadas ejecutadas	Orden de ejecución	Orden de finalización	Resultado
`resto`(75, 18)	1	5	3
`resto`(57, 18)	2	4	3
`resto`(39, 18)	3	3	3
`resto`(21, 18)	4	2	3
`resto`(3, 18)	5	1	3

Al tratarse de una recursividad lineal final, cuando la ejecución de una llamada al método devuelve un valor como resultado (caso base), entonces dicho valor es también el de retorno del resto de llamadas (ya que sobre ninguno de ellos se efectúa otra operación posterior).

Algoritmo de Euclides

En el capítulo 8, se presentaron dos versiones iterativas (ejemplos 8.4 y 8.5) del algoritmo de Euclides para el cálculo del máximo común divisor (m.c.d.) de dos números naturales a y b mayores que cero. La estrategia de resolución planteada en la segunda versión (ejemplo 8.5) se puede seguir para describir el algoritmo de Euclides de forma recursiva según la siguiente recurrencia:

- Si el resto de a/b es 0, el m.c.d. es b.
- En otro caso, el m.c.d. es el m.c.d. de b y el resto de a/b.

Por tanto, una posible implementación de esta recurrencia es:

```
/** a > 0 y b > 0. */
public static int euclides(int a, int b) {
    if (a % b == 0) { return b; }
    else { return euclides(b, a % b); }
}
```

La condición del caso base es `a % b == 0`, siendo el caso general el opuesto. En el caso base se devuelve como resultado el valor del segundo parámetro. En el caso general se devuelve el resultado de la llamada recursiva poniendo como primer parámetro `b` y como segundo parámetro `a % b`; esto garantiza que el segundo parámetro va siempre decreciendo (pues `a % b` $\in$ `[0, b - 1]`) y se va acercando a la condición del caso base, pues en algún momento llegará como mínimo a valer 1 y en ese caso `a % b = 0`. Se trata de nuevo de una recursividad lineal final.

A continuación, se muestra una traza de la ejecución del método para los valores 152 y 247.

Llamadas ejecutadas	Orden de ejecución	Orden de finalización	Resultado
`euclides`(152, 247)	1	7	19
`euclides`(247, 152)	2	6	19
`euclides`(152, 95)	3	5	19
`euclides`(95, 57)	4	4	19
`euclides`(57, 38)	5	3	19
`euclides`(38, 19)	6	2	19
`euclides`(19, 19)	7	1	19

Sucesión de Fibonacci

La sucesión de Fibonacci fue definida por el matemático italiano Leonardo de Pisa, conocido como Fibonacci, que introdujo en Europa el sistema de numeración indo-arábigo actualmente utilizado. Dicha sucesión es la siguiente:

$$0\ 1\ 1\ 2\ 3\ 5\ 8\ 13\ 21\ \ldots$$

De manera informal, cada término de la sucesión de Fibonacci se define a partir de la suma de sus dos términos predecesores en la misma sucesión. Los dos términos iniciales, al no tener predecesores suficientes, se definen con un valor de 0 y 1, respectivamente. Por tanto, la sucesión de Fibonacci admite la siguiente formulación recursiva para el término n-ésimo de la sucesión (asumiendo que el término 0-ésimo es el primer término de la sucesión):

- Si $n \leq 1$, el valor del término n-ésimo es n.
- En otro caso, es la suma de los términos $(n-1)$-ésimo y $(n-2)$-ésimo.

Con esta definición, el método recursivo que puede plantearse para encontrar el término n-ésimo de la sucesión de Fibonacci es:

```
/** n >= 0. */
public static int fibonacci(int n) {
    if (n <= 1) { return n; }
    else { return fibonacci(n - 1) + fibonacci(n - 2); }
}
```

La condición del caso base es `n <= 1`, siendo la del caso general la condición opuesta. A diferencia de los ejemplos previos, este método presenta en el caso general dos llamadas recursivas (la de (`n - 1`) y la de (`n - 2`)), y el resultado

devuelto se construye a partir de la suma de los resultados de las dos llamadas; se trata, por tanto, de una recursividad múltiple. En cada llamada se va tendiendo a términos inferiores de la sucesión, por lo que se garantiza (informalmente) que en algún momento se cumplirán las condiciones asociadas al caso base.

Se muestra, a continuación, una traza del método para calcular el término 4-ésimo de la sucesión de Fibonacci.

Llamadas ejecutadas	Orden de ejecución	Orden de finalización	Resultado
`fibonacci(4)`	1	9	`fibonacci(3)` + `fibonacci(2)` = 3
`fibonacci(3)`	2	5	`fibonacci(2)` + `fibonacci(1)` = 2
`fibonacci(2)`	3	3	`fibonacci(1)` + `fibonacci(0)` = 1
`fibonacci(1)`	4	1	1
`fibonacci(0)`	5	2	0
`fibonacci(1)`	6	4	1
`fibonacci(2)`	7	8	`fibonacci(1)` + `fibonacci(0)` = 1
`fibonacci(1)`	8	6	1
`fibonacci(0)`	9	7	0

En la figura 10.3 se muestra el árbol de llamadas al método `fibonacci` generadas por la llamada inicial `fibonacci(4)`, el orden en el que se realizan y el orden en el que finalizan dichas llamadas. Se puede observar cómo se repiten los mismos cálculos en llamadas recursivas idénticas. Por ejemplo, el subárbol generado por la llamada `fibonacci(2)` está duplicado. Para llamadas a `fibonacci` con valores de `n` más grandes (por ejemplo, para `fibonacci(30)`), la repetición de los cálculos será más frecuente, suponiendo un mayor consumo de tiempo y de memoria (por ejemplo, calcular `fibonacci(20)` requiere 21.891 llamadas y calcular `fibonacci(30)` requiere 2.692.537 llamadas).

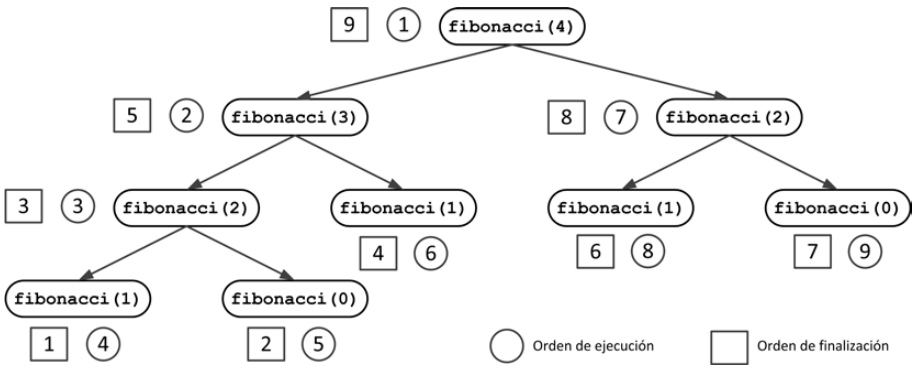


Figura 10.3: Árbol de llamadas del método recursivo `fibonacci`.

10.5 Recursividad con arrays: recorrido y búsqueda

Todos los problemas con arrays, clasificados como *problemas de recorrido y búsqueda*, que se han estudiado en el capítulo 9, pueden también resolverse haciendo uso de esquemas recursivos en lugar de iterativos.

Para facilitar la discusión subsiguiente, se supondrá dada la siguiente definición en Java:

```
tipoBase[] a = new tipoBase[num];
```

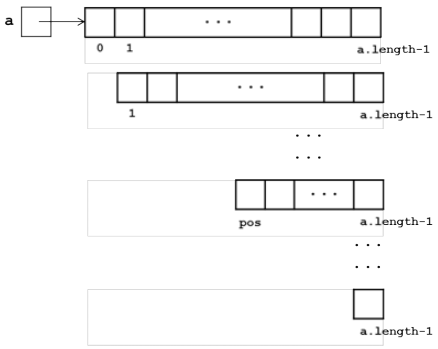
con la que se declara y crea un array `a` de `num` elementos de tipo `tipoBase`. Dada dicha declaración, se puede considerar que el array `a` representa una secuencia de `a.length` elementos de tipo `tipoBase` (`a[0]`, `a[1]`, `a[2]`, ..., `a[a.length - 2]`, `a[a.length - 1]`), denotada como `a[0..a.length - 1]`.

Una secuencia como la anterior se puede definir recursivamente como la secuencia formada por la primera componente de `a`, `a[0]`, y la subsecuencia (que se denomina habitualmente *subarray*) definida por el resto de componentes de `a`, esto es, el array `a` menos su primera componente o subarray `a[1..a.length - 1]`. Nótese que, a su vez, el subarray `a[1..a.length - 1]` puede definirse recursivamente del mismo modo y así, sucesivamente, hasta que en una última descomposición factible el subarray correspondiente esté vacío, sin componentes. Esta descomposición de un array se denomina *descomposición recursiva ascendente*.

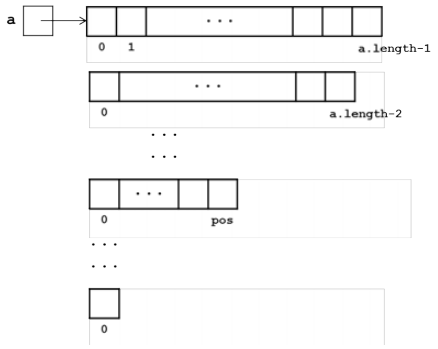
Análogamente, el array `a` se puede definir de forma recursiva considerando el subarray `a[0..a.length - 2]` y su última componente `a[a.length - 1]`, lo que se denomina *descomposición recursiva descendente*. En las figuras 10.4(a) y 10.4(b) se ilustra gráficamente la descomposición recursiva de un array `a` de manera ascendente y descendente, respectivamente.

Naturalmente, para poder diseñar un método recursivo que realice un recorrido o búsqueda sobre un (sub)array `a` se debe disponer de tres datos o parámetros formales del método: el propio (sub)array `a` y las posiciones que marcan el `inicio` y el `fin` del recorrido o la búsqueda a realizar sobre él en cada llamada. Estos tres parámetros definen en cada llamada la *talla* del problema o número de componentes del (sub)array `a[inicio..fin]` sobre las que se realiza el recorrido o la búsqueda, que viene dada por la expresión `t = fin - inicio + 1`; por ejemplo, si en la llamada inicial a un método recursivo `inicio = 0` y `fin = a.length - 1` entonces se explora todo el array `a`, pero si `inicio = (a.length - 1) / 2` y se tiene que `fin = a.length - 1` entonces el subarray de `a` a explorar es su última mitad.

Además de los tres parámetros mencionados, el diseño de un método recursivo exige determinar el tipo de descomposición recursiva que se realiza, ascendente



(a) Ascendente



(b) Descendente.

Figura 10.4: Descomposición recursiva de un array `a`.

o descendente, pues tanto el caso base como la forma de progresar hacia éste en el caso general varían según se elija uno u otro. Específicamente, en una descomposición ascendente de un array `a` el caso base se alcanza cuando `inicio = fin + 1`, es decir, para una talla `t = 0`, y la forma de progresar hacia él en cada llamada es incrementar el valor de `inicio`, con lo que la talla `t` decrece en una unidad; así, el rango de variación de `inicio` será $0 \leq$ `inicio` $\leq$ `a.length`, mientras que `fin` mantiene constante su valor en la llamada inicial, `fin = a.length - 1`. Sin embargo, en una descomposición descendente, el caso base se alcanza cuando `fin = inicio - 1` y la forma de progresar hacia él en cada llamada será decrementar el valor de `fin`; así, el rango de variación de `fin` será `-1` $\leq$ `fin` $\leq$ `a.length - 1`, mientras que `inicio` mantiene constante su valor en la llamada inicial, `inicio = 0`.

A continuación, se presentan e instancian los esquemas recursivos de recorrido y búsqueda sobre un array. Cabe señalar que, por simplicidad, ambos esquemas se presentan como métodos estáticos y que, como se verá en los ejemplos, cuando se

definan como métodos de instancia de un objeto con un array como atributo de instancia, dicho array no será necesario como parámetro formal del método.

10.5.1 Esquemas recursivos de recorrido

En base a la definición de recorrido de un array `a` y la descomposición recursiva ascendente de `a`, el *esquema recursivo de recorrido ascendente* del array `a` desde una posición `izq` hasta una posición `der`, $0 \leq$ `izq` $\leq$ `der` $<$ `a.length`, es el siguiente:

```
/** 0 <= inicio <= der + 1 y fin = der. */
public static void recorrer(tipoBase[] a, int inicio, int fin) {
    if (inicio > fin) { tratarVacio(); }
    else {
        tratar(a[inicio]);
        recorrer(a, inicio + 1, fin);
    }
}
```

donde `tratarVacio()` indica la operación a realizar para un (sub)array sin elementos y `tratar(a[inicio])` indica la operación a realizar con el elemento que ocupa la posición `inicio` del array; siendo la primera llamada o *llamada inicial* `recorrer(a, izq, der)`, esto es, inicialmente `inicio = izq` y `fin = der`. Nótese que si se trata de un recorrido de todos los elementos del array `a`, en la llamada inicial `inicio = 0` y `fin = a.length - 1`, es decir, la talla inicial es `t = a.length`. En este caso, es habitual definir un método público homónimo, denominado *guía* o *lanzadera*, que realiza la llamada inicial, con el fin de ocultar la estructura recursiva del array `a` que muestran los parámetros `inicio` y `fin` de la cabecera del método recursivo `recorrer` anterior que ahora se define privado.

```
public static void recorrer(tipoBase[] a) {
    recorrer(a, 0, a.length - 1);
}
```

El esquema presentado se puede simplificar sustituyendo cualquier referencia al parámetro formal `fin` por su valor en la llamada inicial `der`, como sigue:

```
/** 0 <= inicio <= der + 1. */
public static void recorrer(tipoBase[] a, int inicio) {
    if (inicio == der + 1) { tratarVacio(); }
    else {
        tratar(a[inicio]);
        recorrer(a, inicio + 1);
    }
}
```

El *esquema recursivo de recorrido descendente* es el siguiente:

```
/** inicio = izq y izq - 1 <= fin < a.length. */
public static void recorrer(tipoBase[] a, int inicio, int fin) {
    if (fin < inicio) { tratarVacio(); }
    else {
        tratar(a[fin]);
        recorrer(a, inicio, fin - 1);
    }
}
```

donde `tratarVacio()` indica la operación a realizar para un (sub)array sin elementos y `tratar(a[fin])` indica la operación a realizar con el elemento que ocupa la posición `fin` del array; siendo la llamada inicial `recorrer(a, izq, der)`, esto es, inicialmente `inicio = izq` y `fin = der`. Nótese que, al igual que en el recorrido ascendente, si se trata de un recorrido de todos los elementos del array `a`, en la llamada inicial `inicio = 0` y `fin = a.length - 1`, es decir, la talla inicial es `t = a.length`; pudiéndose definir también, en este caso, un método guía idéntico al del esquema ascendente.

La versión simplificada en la que `inicio` desaparece como parámetro formal, sería:

```
/** izq - 1 <= fin < a.length. */
public static void recorrer(tipoBase[] a, int fin) {
    if (fin == izq - 1) { tratarVacio(); }
    else {
        tratar(a[fin]);
        recorrer(a, fin - 1);
    }
}
```

En general, la elección del esquema ascendente o descendente, o de la descomposición recursiva subyacente del array, debe realizarse del modo que mejor sirva a los propósitos del diseño ya que forma parte de éste y se basa en el análisis de casos que se realiza para resolver de la manera más natural y eficaz posible cada problema particular que se plantee; los ejemplos que siguen servirán para ilustrarlo.

Ejemplo 10.1. En la clase `SecuenciaDeCirculos` (figura 9.18 del capítulo 9) se definió un método iterativo `area` para calcular la suma de las áreas de todos los círculos de una secuencia de círculos; dicha secuencia se representaba mediante un array de `Circulo` (`elArray`) y un valor entero indicando el número de círculos (`talla`).

Considérese ahora el mismo problema pero en su versión recursiva. Para resolverlo, se siguen a continuación los pasos señalados en la sección 10.1 para abordar el diseño de un algoritmo recursivo:

1. Enunciado del problema. Se puede replantear el problema inicial de forma que se haga referencia explícita a la estructura recursiva del array que representa la secuencia de círculos en los términos siguientes: "determinar la suma de las áreas de todos los círculos del array `elArray[0..talla - 1]`". Siguiendo el esquema general recursivo de recorrido ascendente, se puede establecer como cabecera del método la siguiente:

   ```
   /** 0 <= inicio <= talla y fin = talla - 1. */
   private double area(int inicio, int fin) {
       double resMetodo, resLlamada;
       ...
       return resMetodo;
   }

   // area(inicio, fin) == ∑_{i = inicio}^{fin} elArray[i].area()
   ```

   donde, a diferencia del esquema general, se puede observar, por una parte, que al tratarse de un método de instancia no es necesario pasar `elArray` como parámetro y, por otra parte, que devuelve un resultado explícito de tipo `double`.

   El método guía público que lo lanza sería:

   ```
   public double area() { return area(0, talla - 1); }
   ```

2. Análisis de casos:

   - Caso base. Como la suma de las áreas de cero círculos es cero, si `elArray` está vacío o su talla es `0`, `inicio` $>$ `fin` y `tratarVacio()` es `resMetodo = 0;`
   - Caso general. Cuando `inicio` $\leq$ `fin`, la suma de las áreas de los círculos del subarray `elArray[inicio..fin]` se puede expresar como la suma del área de su primer círculo `elArray[inicio].area()` y la de las áreas de los círculos del subarray `elArray[inicio + 1..fin]`. Equivalentemente,

     ```
     resLlamada = area(inicio + 1, fin);
     resMetodo = elArray[inicio].area() + resLlamada;
     ```

3. Transcripción del algoritmo a un método en Java, en el que no se utilizará el parámetro `fin`, pues queda claro tras el análisis de casos que no es necesario:

```
/** 0 <= inicio <= talla. */
private double area(int inicio) {
    double resMetodo, resLlamada;
    if (inicio == talla) { resMetodo = 0; }
    else {
        resLlamada = area(inicio + 1);
        resMetodo = elArray[inicio].area() + resLlamada;
    }
    return resMetodo;
}
// area(inicio) == ∑ talla-1 elArray[i].area()
                  i = inicio
```

   Alternativamente, escrito de forma más compacta se tendría:

```
private double area(int inicio) {
    if (inicio == talla) { return 0; }
    else { return elArray[inicio].area() + area(inicio + 1); }
}
```

   y su método guía público sería el siguiente:

```
public double area() { return area(0); }
```

4. Validación del diseño. Al examinar el código se observa lo siguiente:

   - en el caso general, en cada llamada la talla del problema decrece una unidad porque `inicio` se incrementa en una unidad; por tanto, como antes de efectuarse la llamada `inicio` $<$ `talla`, tras incrementarse puede llegar a alcanzar el valor `talla`;
   - en cualquiera de los dos casos establecidos, el valor del parámetro formal siempre cumple la precondición: `inicio` toma valores en el rango `[0..talla]` porque su valor en la llamada inicial, `area(0)`, se incrementa siempre en uno hasta llegar en el caso base a `talla`.

Ejemplo 10.2. Se quiere añadir un método recursivo a la clase `SecuenciaDeCirculos` que muestre por pantalla en orden inverso el área de todos los círculos de la secuencia. La forma más natural de resolver este problema será basarse en una descomposición descendente del array y seguir el esquema general recursivo de recorrido descendente.

1. Enunciado del problema. Para hacer referencia explícita a la estructura recursiva del array que representa la secuencia de círculos, se replantea el problema como sigue: "mostrar por pantalla en orden inverso el área de todos los círculos del array `elArray[0..talla - 1]`". Siguiendo el esquema general

recursivo de recorrido descendente, se puede establecer como cabecera del método la siguiente:

```
/** inicio = 0 y -1 <= fin < talla. */
private void mostrarArea(int inicio, int fin)
```

donde, a diferencia del esquema general, se puede observar que, al tratarse de un método de instancia, no es necesario pasar `elArray` como parámetro.

El método guía público que lo lanza sería:

```
public void mostrarArea() { return mostrarArea(0, talla - 1); }
```

2. Análisis de casos:

   - Caso base. Como no hay ningún círculo, no hay ningún área que mostrar, si `elArray` está vacío o su talla es `0`, `fin` $<$ `inicio` y `tratarVacio() == ;`
   - Caso general. Cuando `fin`$\geq$`inicio`, mostrar por pantalla en orden inverso el área de los círculos del subarray `elArray[inicio..fin]` se puede expresar como mostrar por pantalla el área de su último círculo `elArray[fin].area()` y las áreas de los círculos del subarray `elArray[inicio..fin - 1]`. Equivalentemente,

     ```
     System.out.println(elArray[fin].area());
     mostrarArea(inicio, fin - 1);
     ```

3. Transcripción del algoritmo a un método en Java, en el que no se utilizará el parámetro `inicio`, pues queda claro tras el análisis de casos que no es necesario:

   ```
   /** -1 <= fin < talla. */
   private void mostrarArea(int fin) {
       if (fin == -1) { ; }
       else {
           System.out.println(elArray[fin].area());
           mostrarArea(fin - 1);
       }
   }
   ```

   Alternativamente, escrito de forma más compacta se tendría:

   ```
   private void mostrarArea(int fin) {
       if (fin >= 0) {
           System.out.println(elArray[fin].area());
           mostrarArea(fin - 1);
       }
   }
   ```

y su método guía público sería el siguiente:

```
public void mostrarArea() { return mostrarArea(talla - 1); }
```

4. Validación del diseño. Al examinar el código se observa lo siguiente:

   - en el caso general, en cada llamada la talla del problema decrece una unidad porque `fin` se decrementa en una unidad; por tanto, como antes de efectuarse la llamada `fin` $\geq$ `0`, tras decrementarse puede llegar a alcanzar el valor `-1`;
   - en cualquiera de los dos casos establecidos, el valor del parámetro formal siempre cumple la precondición: `fin` toma valores en el rango `[-1..talla - 1]` porque su valor en la llamada inicial, `mostrarArea(talla - 1)`, se decrementa siempre en uno hasta llegar en el caso base a `-1`.

10.5.2 Esquemas recursivos de búsqueda

Para obtener un esquema recursivo de búsqueda se puede plantear el siguiente análisis de casos: sea `a[inicio..fin]`, `0` $\leq$ `inicio` $\leq$ `a.length` y `-1` $\leq$ `fin` $<$ `a.length`, el subarray de talla `t = fin - inicio + 1` donde se busca la posición de un elemento que cumpla una cierta propiedad; si `t = 0`, es decir, el subarray está vacío, obviamente ningún elemento cumple la propiedad y el resultado de `buscar` en su caso base es `-1`; si por el contrario `t` $>$ `0`, esto es, el subarray contiene al menos una componente, en base a una descomposición ascendente de éste, bien `a[inicio]` cumple la propiedad y el resultado de `buscar` es `inicio` o bien `a[inicio]` no la cumple y se deberá continuar buscando en el subarray `a[inicio + 1..fin]`. En base a este análisis, se propone el siguiente *esquema recursivo de búsqueda ascendente* de la posición de un elemento que cumpla una cierta propiedad en un array `a` desde una posición `izq` hasta una posición `der`, `0` $\leq$ `izq` $\leq$ `der` $<$ `a.length`:

```
/** 0 <= inicio <= der + 1 y fin = der. */
public static int buscar(tipoBase[] a, int inicio, int fin) {
    int resMetodo = -1;
    if (inicio <= fin) {
        if (propiedad(a[inicio])) { resMetodo = inicio; }
        else { resMetodo = buscar(a, inicio + 1,fin); }
    }
    return resMetodo;
}
```

donde `propiedad(a[inicio])` comprueba si el elemento que ocupa la posición `inicio` del array cumple la propiedad enunciada.

Nótese cómo el análisis de casos realizado se refleja en su diseño: por motivos puramente sintácticos del lenguaje Java `resMetodo` se inicializa en cada llamada a `-1`, el resultado de la búsqueda para el caso base; dicho valor sólo se modifica si el caso que expresan los parámetros de la llamada es distinto del base, es decir, si se busca un elemento que cumpla la propiedad en un subarray donde `inicio` $\leq$ `fin`. Además, al tratarse de una búsqueda también en el caso general se puede obtener el resultado del método sin efectuar llamada: como ya se ha indicado, si `a[inicio]` cumple la propiedad el resultado es `inicio`; sino, se debe seguir recorriendo `a` en busca de algún elemento que la cumpla –recuérdese que en el peor de los casos, cuando ningún elemento cumple la propiedad, cualquier búsqueda se transforma en recorrido–.

A partir de este esquema se pueden obtener fácilmente los esquemas recursivos de búsqueda ascendente simplificada (en la que se obvia el parámetro `fin`), descendente (basada en la descomposición descendente del array `a`) y descendente simplificada; basta con seguir los pasos y premisas del recorrido recursivo.

Para concluir con la presentación de los esquemas recursivos y con el fin de subrayar una vez más que la elección de un tipo de esquema ascendente o descendente no es en general arbitraria, se aborda a continuación el diseño de un método recursivo que obtenga la posición de la última aparición de un dato dado en un array.

Ejemplo 10.3. Se quiere añadir un método recursivo a la clase `SecuenciaDeCirculos` que obtenga la posición de la última aparición de un círculo de color `col` de la secuencia.

Tras una breve reflexión sobre el problema enunciado es fácil concluir que éste admite, al menos, los dos tipos de soluciones siguientes:

- en base a una descomposición recursiva *ascendente* de `elArray` no cabe más que plantear la solución requerida como un *recorrido* del subarray `elArray[inicio..fin]`, donde para cada elemento visitado se debe comprobar si es el último círculo del subarray cuyo color coincide con el dado;
- en base a una descomposición recursiva *descendente* de `elArray` la solución requerida es una *búsqueda* de la primera aparición de un círculo de color `col` en el subarray `elArray[inicio..fin]`.

Aunque ambos tipos de soluciones son correctas, como se verá en el capítulo 11, el criterio de eficiencia es el que lleva a elegir una u otra y, por tanto, el tipo de descomposición recursiva del array sobre el que se efectuará el diseño. Así, la solución óptima es la de búsqueda que, en su peor caso, tiene que visitar todos los elementos del array como si de un recorrido se tratara y, en su mejor caso, en la llamada inicial, inmediatamente obtiene la solución.

Una vez elegido el tipo de solución, se refinará hasta convertirse en un método ejecutable Java siguiendo las siguientes etapas:

1. Enunciado del problema. Se replantea el problema como sigue: "obtener la posición de la última aparición de un círculo de color `col` en el array `elArray[0..talla - 1]`". Siguiendo el esquema general recursivo de búsqueda descendente, se puede establecer como cabecera del método:

   ```
   /** inicio = 0 y -1 <= fin < talla. */
   private int ultimaP(String col, int inicio, int fin) {
       int resMetodo;
       ...
       return resMetodo;
   }
   ```

   Su método guía, público, sería el siguiente:

   ```
   public int ultimaP(String col) { return ultimaP(col, 0, talla - 1); }
   ```

2. Análisis de casos.

   - Caso base. Si `elArray` es un array vacío obviamente no hay ningún círculo de color `col`. Por tanto, el caso base de `ultimaP` es aquel en que `fin` $<$ `inicio` y tiene como resultado `resMetodo = -1;`
   - Caso general. Cuando `fin` $\geq$ `inicio`, en base a una descomposición descendente de `elArray`, bien `elArray[fin]` es la primera aparición de un círculo de color `col` en `elArray` –por lo que el resultado de `ultimaP` es `fin`– o bien no lo es y se deberá continuar con su búsqueda en `elArray[inicio..fin - 1]`. Equivalentemente,

     ```
     if (elArray[fin].getColor().equals(col)) {
         resMetodo = fin;
     }
     else { resMetodo = ultimaP(col, inicio, fin - 1); }
     ```

3. Transcripción del algoritmo a un método en Java, en el que no se utilizará el parámetro `inicio` pues queda claro tras el análisis de casos que no es necesario:

   ```
   private int ultimaP(String col, int fin) {
       int resMetodo = -1;
       if (fin > -1) {
           if (elArray[fin].getColor().equals(col)) {
               resMetodo = fin;
           }
           else { resMetodo = ultimaP(col, fin - 1); }
       }
       return resMetodo;
   }
   ```

Su método guía público sería:

```
public int ultimaP(String col) { return ultimaP(col, talla - 1); }
```

4. Validación del diseño. Al examinar el código se observa que:
   - en el caso general, en cada llamada la talla del problema decrece una unidad porque se decrementa `fin`; por tanto, como antes de la llamada `fin` $>$ `-1`, tras decrementarse en uno puede llegar a alcanzar el valor `-1`;
   - en cualquiera de los dos casos establecidos, el valor del parámetro formal siempre cumple la precondición: `fin` toma valores en el rango `[-1..talla - 1]` porque su valor en la llamada inicial, `ultimaP(col, talla - 1)`, se decrementa siempre en uno hasta llegar en el caso base al valor `-1`.

10.6 Recursividad con objetos de tipo `String`

Como se puede suponer, las `String` admiten un tratamiento recursivo similar al de los arrays ya que es posible acceder por su posición a sus caracteres constitutivos y, de ahí, que muchas de las estrategias vistas en la sección anterior también les serán aplicables.

Sin embargo, en Java, arrays y `String` tienen características diferenciales que provocan que su tratamiento, tanto el iterativo como el recursivo, pueda ser distinto.

La diferencia fundamental entre los arrays y las `String` proviene del hecho de que los primeros son mutables mientras que las segundas no lo son. Esto se hace evidente cuando se observa que el acceso a la posición i-ésima de cierto array `a`, que se efectúa mediante la expresión `a[i]`, sirve tanto para obtener el elemento i-ésimo como para sobrescribirlo; mientras que el acceso a la posición i-ésima de cierta `String` `s`, se puede hacer exclusivamente para obtener el carácter (y no para modificarlo), ejecutando en ese caso `s.charAt(i)`. Debido a esta inmutabilidad, no es posible modificar una `String` sino tan solo construir una nueva, tal vez reemplazando la antigua.

En la práctica, lo anterior significa que cualquier problema que se deba resolver alterando el contenido de una `String` deberá, en su resolución, retornar un valor que consistirá en una versión nueva de la `String` construida a partir de la original en la que se habrá modificado el o los elementos necesarios.

Otra particularidad evidente de las `String` en Java es que se puede aplicar a las mismas un grupo de operaciones definidas en la clase `String` orientadas a su tratamiento. Por supuesto, operaciones predefinidas tales como `concat(String)`, `substring(int)` y `startsWith(String)` podrían definirse sobre arrays, para ello bastaría con escribir los métodos equivalentes necesarios.

Como ejemplo de tratamiento recursivo de una `String`, considérese el siguiente método que, dada cierta `String s`, devuelve la inversión de la misma desde cierta posición `pos` en adelante. El análisis de casos es muy parecido al que se habría efectuado si se hubiera considerado un array en lugar de una `String`, por lo que simplemente se han señalado en el código los casos base y general. Obsérvese que el método puede devolver como resultado de su ejecución una `String` vacía:

```
/** pos >= 0. */
public static String inversion(String s, int pos) {
    // Caso base: String vacía.
    if (pos >= s.length()) { return ""; }
    // Caso general: No vacía. Tratar la substring posterior.
    else { return inversion(s, pos + 1) + s.charAt(pos); }
}
```

Un método así no debería utilizarse en la práctica ya que el proceso de concatenación efectuado en el caso general puede ser ineficiente al suponer, cada vez que se efectúa, la copia completa de las `String` implicadas. Sería más eficiente, copiar la `String` en un nuevo array de caracteres, invertir el mismo y después crear a partir de este último una nueva `String`.

Algo parecido ocurriría en el método recursivo siguiente en el que se cambian los caracteres en mayúscula por sus respectivas minúsculas e, idénticamente, aquellos que estén en minúscula por sus correspondientes mayúsculas; permaneciendo inalterados todos aquellos que, como los dígitos o los signos de puntuación, no sean caracteres alfabéticos. Nótese el uso de los métodos de clase, pertenecientes a la clase envoltorio `Character`: `isLetter(char)`, `isUpperCase(char)`, `toLowerCase(char)` y `toUpperCase(char)`, cuyo significado es el evidente. En esta ocasión, la concatenación para formar la nueva `String` se efectúa por su inicio (en lugar de por el final como en el método anterior) para así construir la nueva `String` respetando el orden de la original:

```
/** pos >= 0. */
public static String mayMin_MinMay(String s, int pos) {
    // Caso base: String vacía
    if (pos >= s.length()) { return ""; }
    // Caso general: No vacía. Tratar la substring posterior.
    else {
        char c = s.charAt(pos);
        if (Character.isLetter(c)) {
            if (Character.isUpperCase(c)) {
                c = Character.toLowerCase(c);
            } else { c = Character.toUpperCase(c); }
        }
        return c + mayMin_MinMay(s, pos + 1);
    }
}
```

Naturalmente, el valor del parámetro `pos` debe ser 0 en la llamada inicial a cualquiera de los dos métodos anteriores, tanto si se desea invertir toda la `String` utilizando el primer método, como si lo que se desea es intercambiar mayúsculas por minúsculas y viceversa en toda la `String`, utilizando el segundo. Una vez más, puede utilizarse un método lanzadera adicional para ejecutar en las condiciones iniciales deseadas los métodos recursivos de forma similar a como se ha hecho en ejemplos anteriores en este capítulo.

Debido a la representación interna de las `String` puede ocurrir que alguna de las operaciones definidas sobre las mismas pueda ser más eficiente que su equivalente si se realizara con arrays. Eso es lo que ocurre, como se verá a continuación, con la operación `substring(int)` en las versiones de Java 6 y anteriores.[1]

10.6.1 Representación de objetos de tipo `String` y su implicación en la operación `substring`

Los objetos de tipo `String` se representan internamente en Java (en las versiones 6 y anteriores) utilizando un array de `char`, soporte de la secuencia de caracteres, y dos valores enteros que indican, respectivamente, la posición del array en la que comienza la `String` y cuántos caracteres ocupa la misma.

Por ejemplo, dada la `String s`, `"Hola mundo"`, la substring de la misma `s1`, `"la mun"`, se representa internamente en Java (véase la captura de *BlueJ* [KR16] en la figura 10.5) mediante los tres valores, atributos privados del objeto `s1`:

- `"Hola mundo"`, que es el array de caracteres asociados,
- 2, entero que indica la posición de comienzo de los caracteres de la `String s1` en el array anterior, y
- 6, entero que indica el número de caracteres que tiene la `String s1`.

Gracias a esta forma de representación, la operación `substring` no realiza copia de los caracteres de la `String` original. Al ser así, es posible aplicar de forma eficiente dicha operación en la resolución de problemas relacionados con cadenas de caracteres. En cambio la obtención de un subarray a partir de otro dado sí que implica la copia de los elementos del primero en el segundo.

Así, dado el array: `{'H','o','l','a',' ','m','u','n','d','o'}`, obtener un subarray del mismo, tal y como se hizo antes en la versión con `String`, implica la creación de nueva memoria y la copia, elemento a elemento, de los caracteres correspondientes.

[1] En Java 7.0 se ha modificado la representación interna de las `String`, lo que ha provocado que la operación `substring` haya pasado de tener un coste constante a tenerlo lineal.

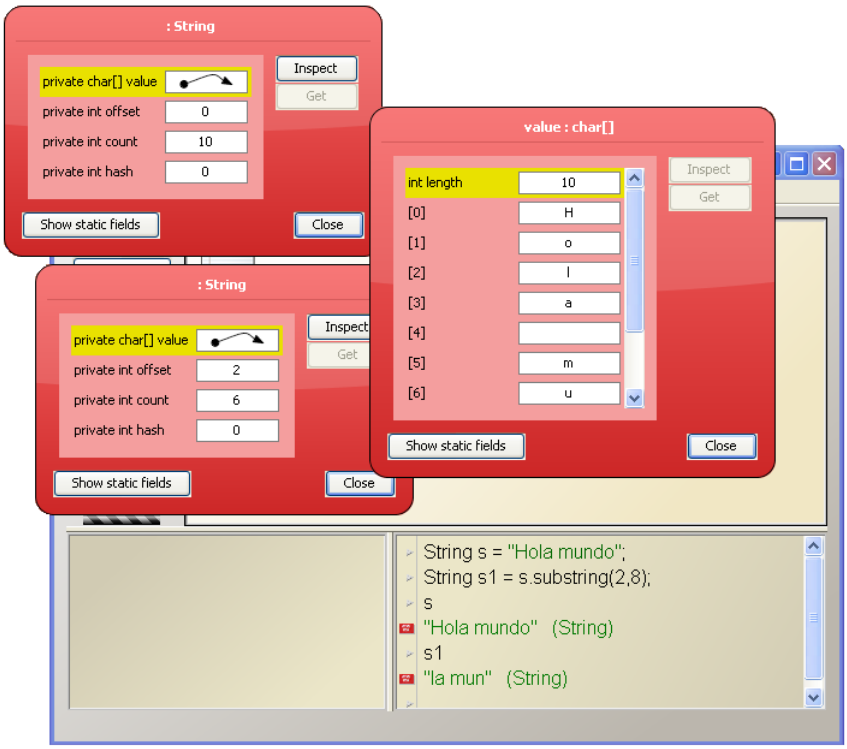


Figura 10.5: Inspección en *BlueJ* de la `String "Hola mundo"` y su substring `"la mun"`.

Ejemplo 10.4. Considérese ahora el problema de contar el número de caracteres `'a'` en cierta `String s`. Si se realiza un análisis para una solución recursiva del mismo, se tiene:

- Caso base: la `String s` es vacía, por lo que el número de `'a'`s que contiene es 0.
- Caso general: la `String s` no es vacía, por lo que:
  - si el primer carácter de `s` es `'a'`, entonces el total de `'a'`s es 1 más las `'a'`s que contenga la substring de `s` obtenida a partir del carácter de índice 1 en adelante;
  - en cambio, si el primer carácter no es `'a'`, entonces el total de `'a'`s es las que contenga la substring de `s` desde el carácter de índice 1 en adelante.

Es posible implementar el análisis anterior mediante el método siguiente:

```
public static int cuentaAs(String s) {
    // Caso base: String vacía.
    if (s.length() == 0) { return 0; }
    // Caso general: No vacía. Tratar la substring posterior.
    else if (s.charAt(0) == 'a') {
        return 1 + cuentaAs(s.substring(1));
    }
    else { return cuentaAs(s.substring(1)); }
}
```

Como ejemplo, se muestra la solución del mismo problema utilizando un parámetro posicional que indique dónde comienza la parte de la `String` sobre la que se trabaja, tal y como se ha hecho en los ejemplos iniciales de la sección. Ambos métodos: `cuentaAs(String)` y `cuentaAs(String, int)` son, en principio, igual de eficientes.[2]

```
/** pos >= 0. */
public static int cuentaAs(String s, int pos) {
    // Caso base: String vacía.
    if (pos >= s.length()) { return 0; }
    // Caso general: No vacía. Tratar la substring posterior.
    else if (s.charAt(pos) == 'a') {
        return 1 + cuentaAs(s, pos + 1);
    }
    else { return cuentaAs(s, pos + 1); }
}
```

Es obvio que también se puede efectuar un análisis y solución prácticamente idénticos a los anteriores, pero considerando el último carácter de la `String` en lugar del primero, así como la substring inicial en lugar de la final.

El método siguiente implementa dicho análisis. El estudio de los casos aparece reflejado en los comentarios del método.

```
public static int cuentaAs2(String s) {
    // Caso base: String vacía.
    if (s.length() == 0) { return 0; }
    // Caso general: No vacía. Tratar la substring anterior.
    else if (s.charAt(s.length() - 1) == 'a') {
        return 1 + cuentaAs2(s.substring(0, s.length() - 1));
    }
    else { return cuentaAs2(s.substring(0, s.length() - 1)); }
}
```

Recuérdese que `s.substring(i, j)` es un objeto `String` que representa la substring de `s` formada con los caracteres comprendidos entre el `i` y el `j - 1`.

[2]Según lo que se verá en el capítulo 11, ejemplo 7, cuando se utilice una versión de Java anterior a la 7.0.

10.7 Recursividad versus iteración

Al comparar la recursividad con iteración, se observa que presentan algunas similitudes:

- Tanto la recursividad como la iteración hacen uso de estructuras de control: la recursividad usa como instrucción principal una instrucción de selección (condicional) y la iteración usa como instrucción principal una instrucción de repetición (bucle).
- Ambas requieren una condición de terminación: la condición del caso base en la recursividad y la condición de la guarda del bucle en la iteración.
- Ambas se aproximan gradualmente a la terminación: la iteración conforme se acerca al cumplimiento de una condición y la recursividad conforme se divide el problema en otros más pequeños.
- Ambas pueden tener (por error) una ejecución potencialmente infinita.

Se puede demostrar que la solución algorítmica de cualquier problema algorítmicamente resoluble, se puede expresar tanto recursivamente como iterativamente. En este sentido, se dice que la recursividad como la iteración son equivalentes y, por ello, alternativos.

No es posible afirmar en general qué es lo más conveniente o sencillo. Es frecuente encontrar problemas para los que la solución iterativa es más sencilla de estructurar, y de expresar en el lenguaje de programación, que un equivalente recursivo. Además, la recursividad supone, en general, más carga computacional (espacio en memoria) que la iteración y, a veces, conduce con más facilidad a soluciones redundantes (algunas soluciones recursivas resuelven repetidamente un (sub)problema) que la iteración. Pero, en otras ocasiones, la recursividad es la alternativa más sencilla para resolver ciertos problemas cuya formulación iterativa resulta complicada. En estos casos, la versión recursiva refleja de manera más natural, concisa y elegante la solución al problema, lo que hace que sea más fácil de depurar y entender. Además, el planteamiento recursivo de muchos problemas es sólo un primer paso para poder desarrollar una solución iterativa posterior de mayor eficiencia mediante el uso de técnicas algorítmicas avanzadas (como la *programación dinámica*). Cabe también señalar que hay modelos de cómputo y lenguajes de programación cuyo único mecanismo de repetición es de carácter recursivo .

Se puede concluir, por lo tanto, que recursividad e iteración, además de alternativas, son complementarias.