Estructuras de Datos y Algoritmos.
Unidad Didáctica I: Conceptos de Java para Estructuras de Datos.

Tema 5. Diseño recursivo y eficiente: soluciones Divide y Vencerás para la Ordenación y la Selección

Mabel Galiano Natividad Prieto

Departamento de Sistemas Informáticos y Computación
Escuela Técnica Superior de Ingeniería Informática

Índice

1. Razonamiento y Diseño recursivos: revisión de conceptos 3
1.1. Aplicación del razonamiento Inductivo a la Programación . . . 7
1.1.1. Esquema general recursivo . . . 8
1.1.2. Taxonomía de la Recursión . . . 11
1.1.3. Ejecución de un método recursivo: la Pila de la Recursión . . . 13
1.1.4. Iteración VS Recursión . . . 15
1.2. Etapas del diseño recursivo . . . 17
1.2.1. Esquema recursivo de Recorrido de un `array` . . . 19
1.2.2. Esquema recursivo de Búsqueda de un Dato `b` en un `array` . . . 21
1.3. Coste Espacial y Temporal de un método y su análisis . . . 24
1.3.1. Revisión de conceptos y notación . . . 24
1.3.2. Expresión de la Complejidad Temporal de un método recursivo: Ecuaciones o Relaciones de Recurrencia . . . 29

2. El coste como criterio de diseño: teoremas de coste y su aplicación al desarrollo de estrategias recursivas eficientes 31
2.1. Reducción Aritmética VS Geométrica de la talla para la Recursión Lineal: la estrategia de Reducción Logarítmica y algunos ejemplos de su aplicación . . . 32
2.2. Recursión Múltiple VS Recursión Lineal: la estrategia Divide y Vencerás (DyV) y su aplicación al problema de la Ordenación . . . 37
2.2.1. Ordenación por Fusión o *Merge*: métodos `mergeSort` y `fusionDyV` . . . 42
2.2.2. *Quick Sort*: métodos `quickSort` y `particion` . . . 45
2.3. Una solución eficiente al problema de la Selección: método `seleccionRapida` . . 51

Objetivos y Bibliografía

El objetivo principal de este tema es mostrar que la incorporación del criterio de eficiencia al proceso de diseño recursivo permite el desarrollo de estrategias eficientes y generales de resolución de problemas complejos y/o de gran peso específico en la Programación; la estrategia paradigma elegida para ilustrarlo es Divide y Vencerás por dos motivos: su instancia para el caso de Recursión Lineal es la estrategia de Reducción Logarítmica y permite resolver recursiva y eficientemente los problemas de Ordenación y Selección o Búsqueda del `k`-ésimo mínimo de una Colección de Datos.

Para conseguir el objetivo enunciado en este tema se amplían los conceptos sobre diseño recursivo estudiados en la asignatura Programación de primer curso. En concreto, tras haber revisado detenidamente en su primer apartado los principios del diseño recursivo y del uso del coste como criterio efectivo de comparación entre los distintos métodos que resuelven un mismo problema, en el segundo apartado del tema se presentan los (cuatro) Teoremas que establecen el coste Temporal Asintótico de un método recursivo sin necesidad de resolver y acotar la solución de las Relaciones de Recurrencia que expresan su Complejidad, el elemento imprescindible para aplicar el criterio de eficiencia ya en los primeros pasos del proceso de diseño recursivo y no tras su conclusión. Como resultado directo del análisis y confrontación de tales teoremas se deducirán las estrategias de Reducción Logarítmica para Recursión Lineal y la de Divide y Vencerás para Recursión Múltiple; ésta última se aplicará al diseño de métodos eficientes para la Selección (`seleccionRapida`) y Ordenación de un `array` genérico (métodos `mergeSort` y `quickSort`).

Como bibliografía básica del tema se proponen los siguientes apartados de los Capítulos `7` y `8` del libro de Weiss M.A. **Estructuras de datos en Java** (Adisson-Wesley, 2000): del `7.1` al `7.4` y del `8.5` al `8.7`. También se recomienda consultar para la parte de Divide y Vencerás el Capítulo `19` del libro de Sahni, S. **Data Structures, Algorithms, and Applications in Java** (McGraw-Hill, 2000).

En cuanto a bibliografía adicional se pueden consultar los Capítulos `4` y `7` del libro de Brassard G. y Bratley P. **Fundamentos de Algoritmia** (Prentice Hall, 1997) y el Capítulo `3` del texto de Ferri, F.J., Albert J.V. y Martín G. **Introducció a l'anàlisi i disseny d'algorismes** (Universitat de València, 1998); asimismo resultan de interés las siguientes referencias electrónicas: **la tabla de material didáctico del ítem 5 de los Contenidos del tema en la PoliformaT EDA**, donde figuran diversas clases Java que se pueden usar para profundizar en el estudio de la estructura e implementación recursiva de un método, y la página **http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html**, que contiene distintas demostraciones gráficas y métodos de Ordenación de un `array` en Java desarrollados por James Gosling y Jason Harrison.

NOTA. Los métodos `mergeSort` y `quickSort` se añadirán a la clase `Ordenacion` del paquete `librerias.util.operacionesArray` del proyecto BlueJ `eda` y el método `seleccionRapida` a la clase `Operaciones` del mismo paquete. Los restantes métodos recursivos que aparecen en los ejemplos y ejercicios propuestos de este tema formarán parte de las distintas clases que componen el (nuevo) paquete `tema5` de `ejemplos` de `eda`.

1. Razonamiento y Diseño recursivos: revisión de conceptos

Los términos Inducción, Recursión o Inferencia se utilizan indistintamente y en diversos ámbitos, desde el cotidiano al científico, para denominar una misma estrategia de resolución de problemas: a partir de un conjunto finito de ejemplos o casos sencillos de un problema para los que se conoce la solución, por trivial o sencilla, se realiza una hipótesis de cuál es la solución general de dicho problema, i.e. la solución aplicable a cualquiera de sus casos o caso general.

Para poner de manifiesto la naturalidad con la que el ser humano aplica la estrategia inductiva, a sabiendas o no, se enuncian a continuación algunos problemas específicos que se resuelven aplicándola y que, más que ajenos o novedosos, deben resultar familiares: un test psicotécnico, en el que se pide establecer el siguiente de una serie de elementos dada, números o figuras típicamente, el aprendizaje de la lengua materna que realizan los bebés y en Matemáticas la definición de muchas funciones, entre otras factorial y fibonacci, conjuntos como los Naturales y la Prueba por Inducción de una propiedad dada. Estos ejemplos ayudan a perfilar aún más las características del razonamiento inductivo:

1. La hipótesis de solución del problema para el caso general se expresa **explícitamente** en términos de la(s) solución(es) del mismo problema para un caso(s) más sencillo(s).
2. La hipótesis de solución del problema para el caso general debe ser validada o demostrada mediante un proceso de prueba, sin el cual no deja de ser más que eso, una hipótesis. Es precisamente por esta característica que, en muchas ocasiones, se denomina Prueba por Inducción al razonamiento inductivo.
3. Se reconoce como recursivo a cualquier ente (definición, proceso, estructura, etc.) que se define en función de sí mismo.
4. Aún de manera inconsciente, el ser humano lo aplica en un gran número de procesos de Aprendizaje.

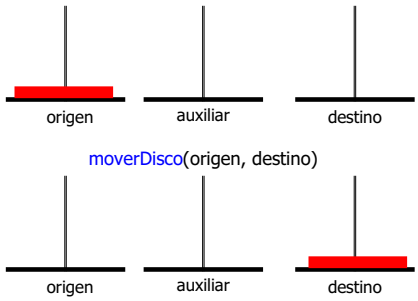
Finalmente, y antes de indicar su papel en la Programación, se debe presentar la ventaja más clara que tiene el razonamiento inductivo frente al deductivo: cuando un problema es lo suficientemente complejo, resulta más fácil expresar su solución inductiva que deductivamente, i.e. en términos de la solución del mismo problema para casos más sencillos que por enumeración de los pasos a seguir para obtenerla. Para ilustrar este hecho se propone resolver el **problema de las Torres de Hanoi**, cuyo enunciado es el siguiente:

> Se dispone de tres torres (`origen`, `destino` y `auxiliar`) y un cierto número de `discos` de diferentes tamaños. Inicialmente, todos los `discos` se encuentran en la torre `origen` apilados en forma decreciente según su diámetro. El problema consiste en desplazarlos a la torre `destino` utilizando la torre `auxiliar` y siguiendo las siguientes reglas: los discos se deben desplazar uno a uno y en ningún momento puede haber un disco de mayor diámetro situado sobre uno más pequeño.

Como se puede comprobar fácilmente, abordar la resolución general del problema tal como se ha enunciado, por ejemplo para `N = 30` discos, resulta imposible. Un primer paso plausible sería mover el disco más pequeño de la torre `origen` a cualquier torre que no sea la torre `destino`. Pero inmediatamente se plantean las siguientes cuestiones: suponiendo situado en la torre `auxiliar` el disco más pequeño, ¿cómo situar cada uno de los `29` discos restantes en cada una de las tres torres disponibles?; es más, en el momento en que se intente mover el más

pequeño de los 29 discos restantes la elección de la torre `auxiliar` para situar el disco más pequeño empieza a ser dudosa pero, si no se pone en ella ¿dónde se pone entonces el disco más pequeño? En resumen, describir paso a paso la solución de las Torres de Hanoi es demasiado complejo, no resulta posible si se afronta globalmente siguiendo las reglas establecidas.

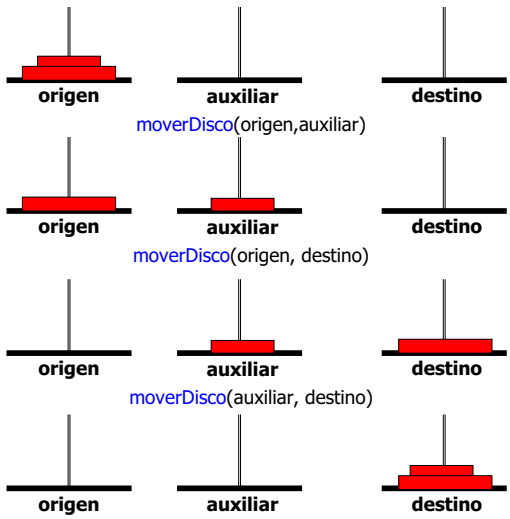
Dicho lo anterior, resulta bastante natural pensar en intentar resolver el mismo problema pero con menos discos, poco a poco, para ver si así se puede entender mejor el problema y darle una solución. Supóngase entonces que se empieza estudiando la solución al caso más fácil posible de Hanoi, esto es aquel en el que hay sólo un disco que mover, `N = 1`. Para este caso trivial, la solución es obvia y se ilustra gráficamente en la siguiente figura: mover el disco de la torre origen a la torre destino, operación básica que a partir de ahora se denotará como `moverDisco(origen, destino)`.



Para `N = 1`, la solución de Hanoi se puede expresar entonces como sigue:

```
hanoi(1, origen, auxiliar, destino)
    moverDisco(origen, destino)
```

Si se afronta ahora la resolución de Hanoi para el siguiente caso más fácil, éste sólo puede ser Hanoi con `N = 2`; a estas alturas ya se debe haber caído en la cuenta de que la talla del problema Hanoi es el número de discos a mover, $N > 0$. Como muestra la siguiente figura, la solución a este caso sencillo es realizar tres operaciones básicas `moverDisco`:



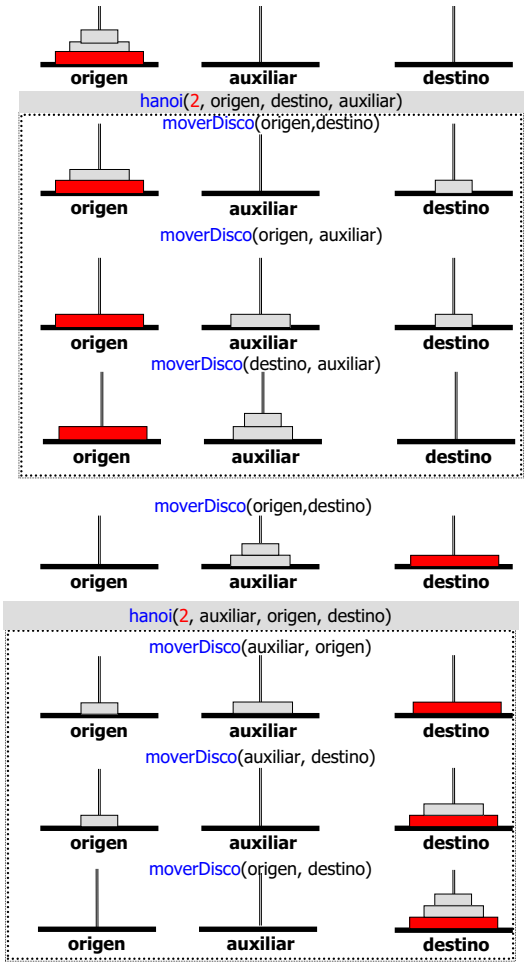
Más que la descripción paso a paso de la solución obtenida lo que interesa aprender de este caso es que su solución se expresa, *al igual que la de Hanoi con `N = 1` disco*, en términos de la operación básica `moverDisco` y, *a diferencia de la de Hanoi con `N = 1` disco*, usando la torre `auxiliar` como **destino** del primer disco que se mueve y como **origen** del último disco que se mueve; sólo en el segundo movimiento `auxiliar` juega el mismo papel que en Hanoi con `N=1`, esto es ninguno. Así, la solución de Hanoi con `N = 2` se puede expresar como sigue:

```
hanoi(2, origen, auxiliar, destino)
    hanoi(1, origen, destino, auxiliar)
    moverDisco(origen, destino)
    hanoi(1, auxiliar, origen, destino)
```

A partir de ésta, la hipótesis de solución de Hanoi con `N = 3` discos sería:

```
hanoi(3, origen, auxiliar, destino)
    hanoi(2, origen, destino, auxiliar)
    moverDisco(origen, destino)
    hanoi(2, auxiliar, origen, destino)
```

Para comprobar su validez, en la siguiente figura se muestra la solución de Hanoi con `N=3` discos pero enmarcando el conjunto de estados y movimientos que se pueden describir en términos de Hanoi con `N = 2` discos:



De la figura anterior resulta inmediato comprobar que, en efecto, la hipótesis realizada es válida. Por ello, generalizándola para cualquier número de discos N>0, la **Hipótesis de Inducción** sobre la solución general de Hanoi que se obtiene es la siguiente:

```
hanoi(N, origen, auxiliar, destino)
si N = 1 moverDisco(origen, destino)
sino hacer:
        hanoi(N-1, origen, destino, auxiliar)
        moverDisco(origen, destino)
        hanoi(N-1, auxiliar, origen, destino)
```

Lo que expresa esta hipótesis es algo razonable: para mover N discos de la torre origen a la torre destino, utilizando en ese proceso la torre auxiliar, hacer:

si N = 1 moverDisco(origen, destino)

sino hacer:

1. siguiendo las reglas establecidas, mover los N-1 primeros discos de la torre origen a la torre auxiliar, utilizando en ese proceso como torre auxiliar destino; formalmente: hanoi(N-1, origen, destino, auxiliar);
2. como tras realizar el paso 1 en la torre origen sólo queda el disco más grande, moverlo a la torre destino mediante moverDisco(origen, destino);
3. tras realizar el paso 2, mover los N-1 discos de la torre auxiliar, donde se dejaron en el paso 1, a la torre destino, utilizando como torre auxiliar origen; formalmente: hanoi(N-1, auxiliar, origen, destino).

   Tras realizar este último paso, como en la torre destino ya estaba el más grande de todos los discos, la resolución del problema concluye.

Al leer el algoritmo establecido, habituado como se está al razonamiento deductivo, es fácil pensar que no dice nada de cómo mover los N-1 > 1 discos restantes en los pasos 1 y 3, pero recuérdese que se está realizando una definición recursiva y, por tanto, si se pregunta cómo solucionar el movimiento de N-1 discos la respuesta sólo puede ser «pues haciendo para los N-2 restantes lo mismo que se hacía antes para los N-1» o, en otras palabras, seguir el algoritmo que se acaba de presentar hasta que sólo quede un disco que mover.

Una vez formulada la **Hipótesis de Inducción** se debe realizar **la demostración de su corrección**, lo que conlleva las siguientes pruebas:

- **la prueba de Terminación**: independientemente de cuánto tarde en hacerlo, si N>0 hanoi(N,origen,auxiliar,destino) termina o resuelve el problema en un tiempo finito;
- una vez demostrado que termina, se realiza **la prueba de la Hipótesis de Inducción**.

Una de las herramientas más eficaces para realizar ambas pruebas es, precisamente, la Prueba por Inducción. Así, para demostrar que si N>0 hanoi(N,origen,auxiliar,destino) termina en un tiempo finito, basta con aplicar el siguiente razonamiento inductivo:

si N = 1, como moverDisco(origen,destino) es una acción que se realiza en un tiempo finito, hanoi(N,origen,auxiliar,destino) termina en un tiempo finito

y sino, como el número de discos es finito y para resolver Hanoi con N > 1 se debe resolver Hanoi con N-1, tras un tiempo finito se alcanzará la resolución de Hanoi con N = 1. Así, la resolución de Hanoi con N > 1 termina en un tiempo finito.

Para intuir cuánto tarda Hanoi con `N > 1` en resolverse, se propone la siguiente **cuestión:**

> Cuenta la leyenda que en el templo de Brahma en Benares existían tres agujas de diamantes que servían para apilar 64 discos de oro puro. Incansablemente, los sacerdotes transferían discos de una aguja a otra obedeciendo siempre la ley de Brahma: *ningún disco se puede situar encima de uno menor*.
>
> Al inicio del mundo los 64 discos se dispusieron sobre la primera aguja, formando una torre de Brahma; según la leyenda, en el momento en que el más pequeño de los discos se colocara y se tuviera una nueva torre de Brahma llegaría el fin del mundo.
>
> Según la profecía de los brahmanes, suponiendo que los monjes emplearan 20 mseg. en mover un disco, **¿cuándo llegará el fin del mundo?**

Una vez demostrada por Inducción la terminación de Hanoi, utilizando de nuevo la Prueba por Inducción se realizará **la Prueba de la Hipótesis de Inducción**. Así,

> **si** `N = 1`, obviamente, `moverDisco(origen,destino)` es la solución correcta
>
> **y sino**, suponiendo que tanto `hanoi(N-1, origen, destino, auxiliar)` como `hanoi(N-1, auxiliar, origen, destino)` son las soluciones correctas para los respectivos problemas de Hanoi con `N-1` discos, como `moverDisco(origen, destino)` es correcta y `N>N-1` entonces, por construcción, `hanoi(N,origen,destino,auxiliar)` es también correcta.

1.1. Aplicación del razonamiento Inductivo a la Programación

De la introducción realizada y recordando que la Programación no es más que la actividad de resolución de problemas por ordenador se pueden deducir los siguientes hechos:

- el instrumento necesario y suficiente que debe proporcionar cualquier lenguaje de programación para implementar un razonamiento inductivo (recursivo) es el método, bien sea función o procedimiento;
- un método que resuelve un problema de talla dada es recursivo si se define invocándose a sí mismo, para resolver el mismo problema pero de talla menor. En otras palabras, recordando que la petición de ejecución de cualquier método, recursivo o no que sea, se denomina invocación (o llamada) al mismo, un método es recursivo si su ejecución requiere la ejecución previa del mismo método para una talla menor.

Así por ejemplo, un método Java para resolver el problema de Hanoi puede ser el siguiente:

```
static void hanoi(int n, String origen, String auxiliar, String destino){
  if ( n == 1 ) moverDisco(origen, destino);
  else {        hanoi(n-1, origen, destino, auxiliar);
                moverDisco(origen, destino);
                hanoi(n-1, auxiliar, origen, destino);
  }
}
static void moverDisco(String desde, String hasta){
  System.out.println("Moviendo disco de torre "+desde+" a torre "+hasta);
}
```

Para ser ejecutado éste método se debe invocar desde otro, como por ejemplo el que sigue:

```
package ejemplos.tema5;
import java.util.*;
public class TestHanoi {
 public static void main(String args[]){
  Scanner teclado = new Scanner(System.in);
  System.out.println("Introduzca el número de discos en la torre origen:");
  int N = teclado.nextInt();
  if ( N > 0 ) hanoi(N, "origen", "auxiliar", "destino");
  else System.out.println("El número de discos debe ser mayor que cero");
 }
 private static void hanoi(int n, String origen, String auxiliar, String destino){...}
 private static void moverDisco(String desde, String hasta){...}
}
```

Como se observa en el código de `hanoi`, los cambios introducidos para traducir una solución recursiva de un problema a un método Java se deben al cambio de procesador que ejecutará la solución, el cerebro humano VS la máquina (Java) del ordenador. Aunque mayor que la del ser humano, la potencia de un ordenador radica única y exclusivamente en la rapidez con la que es capaz de ejecutar cálculos incansablemente y sin posibilidad de error; esto es, carece de la flexibilidad y conocimiento propios de la mente humana, por lo que realizar una buena labor de programación supone respetar muchas restricciones y un alto grado de formalización. Quizás los exponentes más claros de este encorsetamiento del razonamiento son el cambio de lenguaje en la expresión de la solución, que pasa del lenguaje natural (pseudocódigo) al más restrictivo lenguaje de programación, y la demostración de que los cálculos realizados son correctos, terminan en un tiempo finito y consumen el mínimo de recursos posibles. En los siguientes apartados se detallan y formalizan todos estos cambios, avanzando así en la descripción y características del razonamiento inductivo aplicado al diseño de programas.

1.1.1. Esquema general recursivo

El método `hanoi` que se acaba de presentar no es más que una instancia del esquema general de método recursivo que se presenta a continuación y que plasma en Java la estructura esencial que debe tener cualquier método que se defina para formalizar una estrategia inductiva de resolución de un problema cuyos Datos representa `x`:

```
/** Dominio(x) == true */
public static TipoResultado metodoRecursivo( TipoDatos x ){
  TipoResultado resMetodo, resLlamada1, resLlamada2,..., resLlamadaK;
  if ( casoBase(x) ) resMetodo   = solucionBase(x);
  else{              resLlamada1 = metodoRecursivo( anterior1(x) );
                     resLlamada2 = metodoRecursivo( anterior2(x) );
                     ...
                     resLlamadaK = metodoRecursivo( anteriorK(x) );
                     resMetodo   = combinar( x, resLlamada1, ..., resLlamadaK );
  }
  return resMetodo;
}
// metodoRecursivo devuelve la solución correcta del problema para x
```

Para poder valorar la forma en que este esquema general transcribe a un lenguaje de programación un razonamiento inductivo conviene antes aclarar las siguientes cuestiones.

Sobre la representación de los Datos del problema y su talla: parámetros formales del método, sus tipos y posibles restricciones

Determinar la solución de un problema requiere, implícita o explícitamente, determinar la talla del problema o magnitud de sus Datos; aún más cuando la solución es recursiva, pues se expresa explícitamente en términos de la(s) solución(es) del mismo problema para talla(s) menor(es). Así,

> la cabecera de `metodoRecursivo` debe contener al menos un parámetro formal, `x`, para representar los Datos o talla del problema. Obviamente, el tipo de `x`, `TipoDatos`, debe representar el conjunto o dominio de los Datos del problema.

Ahora bien, no todos los dominios tienen un tipo equivalente en el lenguaje de programación; en `hanoi` por ejemplo la talla es el número de discos a mover y pertenece al dominio de los Naturales estrictamente positivos pero el parámetro formal `x = n` que la representa es de tipo `int`. Por tanto,

> cualquier discordancia entre el dominio de los Datos del problema y el tipo del parámetro formal que los representa debe ser corregida, pues la solución del problema sólo es válida para Datos que son del dominio; imagínese por ejemplo el resultado que obtendría el método `hanoi` cuando `n` $\leq$ `0`.

El primer paso para llevar a cabo esta corrección es advertir que la ejecución del método `metodoRecursivo` exige como condición previa (o Precondición) la comprobación de que el valor de `x` es, además de un valor posible de `TipoDatos x`, un valor del dominio de los Datos del problema; es por ello que se ha incluido como comentario previo a la cabecera de `metodoRecursivo` la expresión lógica `Dominio(x) == true`, que por ejemplo para `hanoi` sería `n` $\geq$ `0`. El segundo paso a dar es bastante obvio: **antes** de invocar a `metodoRecursivo` desde otro método, programar su Precondición. Recuérdese por ejemplo que en el `main` de `TestHanoi` se comprueba mediante un `if` el valor de la variable `N`, el parámetro actual con el que se pretende invocar a `hanoi`: si es estrictamente positivo, entonces se ejecuta `hanoi(N)` y sino se muestra por pantalla un mensaje en el que se recuerda al usuario que el número de discos de la torre origen debe ser positivo; con este simple test el programador consigue hacer coincidir el dominio Natural de Hanoi con el dominio `int` de `hanoi`.

Sobre la evaluación y solución de los casos del problema: variables locales y cuerpo del método

La instrucción fundamental de un método recursivo es una condicional simple `if` mediante la cual, tras comprobar si el caso actual del problema es el base (`casoBase(x)`) o el general (`!casoBase(x)`), se expresa la solución asociada a éste. Así,

> **si** `casoBase(x) == true`, si el valor de `x` es el de la talla del problema en su caso base, **entonces** `metodoRecursivo` devuelve en `resMetodo` la solución correcta del problema para el caso base, la que denota en el esquema `solucionBase(x)`
>
> **y sino**, al cumplirse la Precondición `metodoRecursivo` devuelve en `resMetodo` la «adecuada» combinación de `x` con los resultados de `metodoRecursivo` para tallas del problema estrictamente menores que la que representa `x`, la que denota en el esquema `combinar(x, resLlamada1, ..., resLlamadaK)`;

Una última observación: para evitar que en `metodoRecursivo` se confundan el resultado del método recursivo y el de cada una de las llamadas recursivas se han utilizado distintas variables de `TipoResultado`: `resMetodo` representa la solución del problema de talla `x` y `resLlamada1`, `resLlamada2`, `...`, `resLlamadaK`, representan la solución del mismo problema pero para tallas menores que `x`, respectivamente `anterior1(x)`, `anterior2(x)`, `...`, `anteriorK(x)`.

Sobre la corrección de la solución: coherencia de las llamadas, terminación y corrección

En principio, al igual que para cualquier otro tipo de definición recursiva, la demostración de la corrección de `metodoRecursivo` supone probar que tras finalizar su ejecución al cabo de un tiempo finito, el resultado que obtiene representa la solución correcta del problema; i.e. probar, por ejemplo por Inducción, que `metodoRecursivo` termina y es correcto. Sin embargo y aún así la compilación y ejecución de un método recursivo puede provocar errores si previamente no se ha comprobado **la coherencia de sus llamadas**. A saber, como en su caso general `metodoRecursivo` se invoca a sí mismo para las tallas menores `anterior1(x))`, `...`, `anteriorK(x)`, se tendrán que realizar las siguientes comprobaciones:

1. Cualquier parámetro actual o valores de `x` en cada llamada a `metodoRecursivo` es del dominio del problema. Lo que se puede denotar formalmente como sigue: sea `x` tal que `Dominio(x) == true AND !casoBase(x)`, **entonces**

```
      Dominio(anterior1(x)) == true
AND   Dominio(anterior2(x)) == true
AND   ...
AND   Dominio(anteriorK(x)) == true
```

   A modo de ejemplo, se comprueba ahora esta restricción para el parámetro formal `int n` de `hanoi`, el único que representa la talla del problema e interviene en la demostración: sea `x = n` $>$ `0` por Precondición; como en el caso general de `hanoi n` $>$ `1` **entonces** `anterior(x) = n-1` $\geq$ `1` y, por tanto, `x = n` $>$ `0` como se quería demostrar.
2. Las llamadas de talla menor que `x` obtienen resultados del mismo tipo que el definido en la cabecera de `metodoRecursivo` a partir del mismo número de parámetros, en el mismo orden y del mismo tipo que los definidos en la cabecera de `metodoRecursivo`. Por ejemplo, el método `hanoi` propuesto cumple claramente esta restricción: cada una de las dos llamadas de talla menor que `n`, `hanoi(n-1, origen, destino, auxiliar)` y `hanoi(n-1, auxiliar, origen, destino)`, retorna el mismo tipo de resultado que el definido en su cabecera, `void`, a partir del mismo número de parámetros, cuatro, en el mismo orden y del mismo tipo que los definidos en su cabecera.

Por motivos que se evidenciarán más tarde, resulta de interés presentar ahora dos pruebas de terminación de un método recursivo alternativas a la ya conocida Prueba de Inducción. La primera consiste en enumerar las llamadas que origina la llamada principal o más alta desde otro método; por ejemplo, el número de llamadas que se realizan al invocar `hanoi` con `n == N` viene dado por la siguiente Ecuación de Recurrencia:

$$\texttt{numLlamadas(N)} = \begin{cases} \texttt{2 * numLlamadas(N - 1)} & \text{si } \texttt{N} > 1 \\ \texttt{0} & \text{si } \texttt{N} = 1 \end{cases}$$

Resolviendo y acotando esta ecuación se obtiene que `numLLamadas( N )` $\simeq 2^{N}-1$, un número finito a pesar de ser muy grande; por tanto, el número de llamadas es finito, lo que asegura la terminación de la ejecución de `hanoi(N)`.

Ejercicio propuesto: para darse cuenta de la enorme cantidad de tiempo que puede llegar a consumir la ejecución del método `hanoi`, o de por qué el fin del mundo llegará cuando los sacerdotes del templo de Brahma construyan una nueva torre, basta realizar un pequeño experimento: observar qué ocurre cuando se ejecuta el programa `TestHanoi` sucesivamente para `N = 5`, `N = 10`, `N = 15` y `N = 20`.

**`TestHanoi` figura en la tabla de material didáctico de los Contenidos PoliformaT del tema.

La segunda prueba consiste en demostrar que los valores de los parámetros de las sucesivas llamadas conforman una sucesión ordenada decrecientemente que converge al valor para el que `casoBase(x) == true`; equivalentemente, si la talla del problema expresada por `x` decrece al menos una unidad en cada llamada recursiva, al cabo de un tiempo finito se debe alcanzar la talla del caso base. Formalmente, si x_{base} denota el valor para el que `casoBase(x) == true`, se debe demostrar que

$$\begin{aligned} & x_{base} \leq \texttt{anterior1(x)} < \texttt{x} \\ \texttt{AND}\quad & x_{base} \leq \texttt{anterior2(x)} < \texttt{x} \\ \texttt{AND}\quad & \ldots \\ \texttt{AND}\quad & x_{base} \leq \texttt{anteriorK(x)} < \texttt{x} \end{aligned}$$

1.1.2. Taxonomía de la Recursión

Para plasmar cualquier estrategia inductiva de diseño basta con instanciar el esquema general recursivo presentado. Es más, en función del número de llamadas que se realicen en el caso general y de si el método `combinar(x, resLlamada1, ..., resLlamadaK)` se utiliza o no, cualquier instancia del esquema general se puede clasificar en base a la siguiente taxonomía:

- **Recursión Lineal:** cuando en su caso general un método recursivo genera una **única** llamada. Además, en función de si el resultado de esta llamada se combina o no para obtener el resultado del método, se distinguen los siguientes **sub**tipos de Recursión Lineal:

  1. **Final:** cuando el resultado de la llamada recursiva **es** el resultado del método recursivo, i.e. `resMetodo = resLlamada = metodoRecursivo(anterior(x))`; al no utilizarse el método `combinar`, la llamada recursiva es la última operación que se realiza en cada invocación de un método recursivo Lineal Final. Por ejemplo, el siguiente método recursivo es Lineal Final:

```
/** n > 0 AND m > 0 */
static int maximoComunDivisor(int n, int m){
   int resMetodo, resLlamada;
   if ( n == m )       resMetodo = n;
   else{
          if ( n > m ) resLlamada = maximoComunDivisor(n-m, m);
          else         resLlamada = maximoComunDivisor(n, m-n);
          resMetodo =  resLlamada;
   }
   return resMetodo;
}
// maximoComunDivisor(n,m) calcula el mayor entero que divide a n y m
```

2. **No Final:** cuando el resultado de la llamada recursiva **no es** el resultado del método recursivo sino que debe combinarse de alguna forma con `x` para obtenerlo, i.e. `resMetodo = combinar(x, resLlamada)`. Por ejemplo, el siguiente método recursivo es Lineal No Final:

```
/** n >= 0 */
static int factorial(int n){
  int resMetodo, resLlamada;
  if ( n == 0 ) resMetodo = 1;
  else{
                resLlamada = factorial(n-1);
                resMetodo = n * resLlamada;
  }
  return resMetodo;
}
// factorial(n) calcula n!
```

- **Recursión Múltiple:** cuando en su caso general un método recursivo genera **más de una** llamada en secuencia, como el método `hanoi` ya presentado o el que figura a continuación:

```
/** n >= 0 */
static int fibonacci(int n){
  int resMetodo, resLlamada1, resLlamada2;
  if ( n <= 1 ) resMetodo = n;
  else {
             resLlamada1 = fibonacci(n-1);
             resLlamada2 = fibonacci(n-2);
             resMetodo   = resLlamada1 + resLlamada2;
  }
  return resMetodo;
}
// fibonacci(n) calcula el término n-ésimo de la serie de Fibonacci
```

Ejercicio propuesto: los métodos `maximoComunDivisor`, `factorial` y `fibonacci` que se acaban de presentar han sido diseñados instanciando textualmente el esquema recursivo general. En el programa `RecursionNumerica` que los contiene, modifíquense sus códigos para que no usen ninguna variable local y, por tanto, sean más compactos y efectivos.

**`RecursionNumerica` figura en la tabla de material didáctico de los Contenidos PoliformaT del tema.

El tipo de Recursión de un método determina la estructura del conjunto de llamadas que origina su llamada principal desde otro método. Así, como la Recursión Lineal se caracteriza porque por cada llamada al método origina sólo otra, la llamada más alta a un método recursivo lineal origina una **Secuencia de Llamadas**, cuyo primer elemento es la llamada más alta y cuyo último elemento es la llamada para resolver el caso base. Por ejemplo, al invocar `maximoComunDivisor(10,15)` se origina la siguiente Secuencia de Llamadas:

`maximoComunDivisor(10, 15)`→`maximoComunDivisor(10, 5)`→`maximoComunDivisor(5, 5)`

Al igual que `maximoComunDivisor(10, 15)`, cualquier método Lineal Final calcula el resultado demandado tras generar el último elemento de su Secuencia de Llamadas, lo que concuerda con su definición de método Final: aquel cuyo resultado coincide con el de la única llamada que origina. Para `maximoComunDivisor`, tras resolver `maximoComunDivisor(n-m, m)` o `maximoComunDivisor(n, m-n)` no es necesario ningún cálculo adicional.

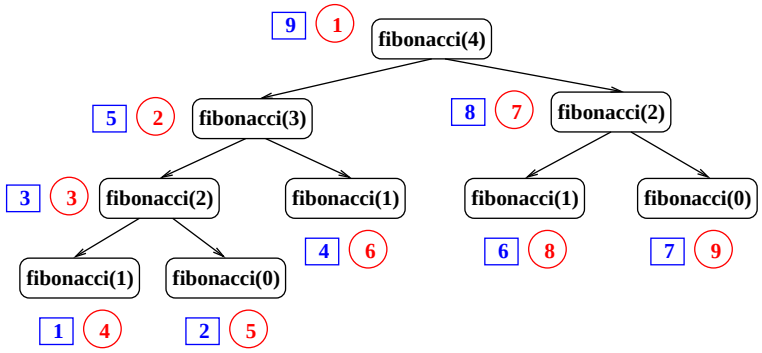
Sin embargo, si el método recursivo es No Final, por definición, el resultado de la última llamada no es el resultado del método, que deberá obtenerse recorriendo en sentido inverso hasta su primer elemento, la llamada más alta; en cada paso de este Recorrido el resultado de la llamada actual se obtiene combinando sus Datos con el resultado de la llamada anterior. Por ejemplo, al invocar `factorial(4)` se origina la siguiente Secuencia de Llamadas:

`factorial( 4 )` → `factorial( 3 )` → `factorial( 2 )` → `factorial( 1 )` → `factorial( 0 )`

Cuando se genera el último elemento de esta Secuencia, la llamada a `factorial(0)`, su resultado es `1` y para obtener el resultado demandado se recorrerá en sentido inverso la Secuencia y el resultado para cada llamada se calcula multiplicando su Dato por el resultado de la llamada anterior, $\forall$`1` $\leq$ `i` $\leq$ `4: factorial(n) = n * factorial(n-1)`.

`factorial( 4 )` → `factorial( 3 )` → `factorial( 2 )` → `factorial( 1 )` → `factorial( 0 )`
`4 * 6` ← `3 * 2` ← `2 * 1` ← `1 * 1` ← `1`

Finalmente, si un método es **recursivo Múltiple**, en lugar de una Secuencia tiene asociado un **Árbol de Llamadas**; cada invocación a un método de este tipo origina en su caso general `K>1` llamadas, lo que se puede representar jerárquicamente mediante un Árbol donde cada uno de sus Nodos tiene tantos sucesores, Hijos, como número de llamadas en secuencia origina el método en su caso general. Así, el Nodo Raíz del Árbol será la llamada más alta y sus Hojas las llamadas para resolver el caso base. Por ejemplo, si se realiza la invocación `fibonacci(4)` se origina el Árbol **Binario** que se muestra en la siguiente figura, y en cuyos Nodos se ha indicado dentro de un círculo el orden de generación y dentro de un cuadrado el de terminación:



1.1.3. Ejecución de un método recursivo: la Pila de la Recursión

En esta sección se estudia cómo implementa la máquina Java la ejecución de un método, recursivo o no, y qué recursos utiliza para ello. Como se verá pronto, conocer estos aspectos permite confrontar en términos de eficiencia los métodos iterativos y sus equivalentes recursivos.

En primer lugar se recordará que para poder ejecutar cualquier método, por ejemplo el método `main` de una aplicación, éste debe ocupar una zona establecida de la memoria RAM del ordenador. Dicha zona consta principalmente de tres partes:

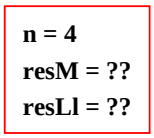
1. la zona reservada al código de la `JVM`, o Zona `1`;
2. la zona reservada para las variables globales o de clase, o Zona `2`;
3. la zona denominada ***Heap*** o memoria dinámica, subdividida a su vez en dos: la reservada a los objetos que se crean vía `new` y el ***Stack*** o **Pila**.

Cada vez que el interprete Java ejecuta la invocación a un método (siguiendo el código situado en la Zona `1`) se crea una estructura de Datos en el *Heap* que recibe el nombre de **Registro de Activación** y contiene toda la información local asociada a dicha ejecución; básicamente, una copia de todos los parámetros y variables locales presentes en la definición del método invocado y la dirección de la Zona `1` donde debe continuar el intérprete Java al finalizar la ejecución del método invocado. Lógicamente, el Registro de Activación desaparece del *Heap*, se borra, al finalizar la ejecución del método invocado, cuando se liberan los recursos que ésta consumía.

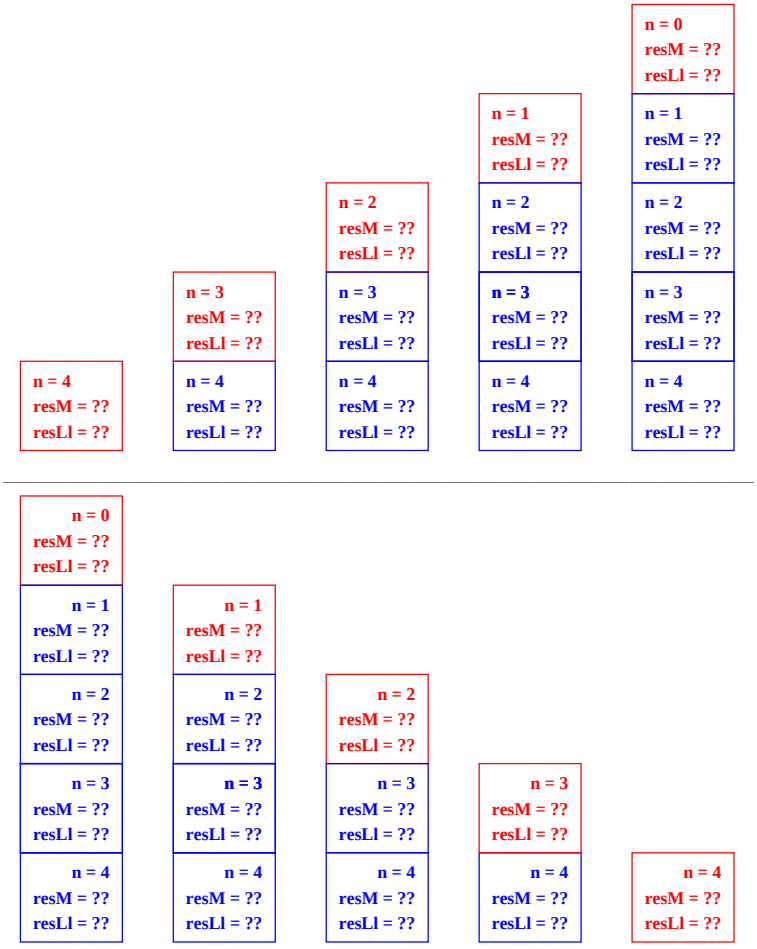
Como es fácil deducir, la ejecución de un método recursivo es similar a la de un método no recursivo, pero por cada invocación a sí mismo que realice se crea un nuevo Registro de Activación, propio de esa llamada, **sobre el último** registro asociado al mismo método, cuya ejecución queda pendiente. Y así sucesivamente hasta que se termina de ejecutar la invocación al mismo método en su caso base, momento en el cual se borra su registro asociado y se puede reanudar la ejecución de la llamada inmediatamente anterior del mismo método que quedó pendiente. Así, la última llamada pendiente será la primera en seguir ejecutándose. Esta mecánica **LIFO** (**L***ast* **I***n* **F***irst* **O***ut*) con la que se crean y borran los Registros de Activación hace que la subarea del *Heap* donde se almacenan se denomine **Pila de la Recursión** o ***Stack***. Asimismo, se dice que los Registros no se crean sino que se **apilan**, no se borran sino que se **desapilan** y en cada momento sólo puede haber **uno activo**, el último que entró.

Además de su mecánica, es importante resaltar que el uso del Registro de Activación permite implementar cada llamada a un método recursivo como un clon de éste, es decir una copia exacta pero distinta de la definición del método: los valores de sus parámetros formales y variables locales varían durante su ejecución, sin que por ello quede alterada la de otras llamadas pendientes; la comunicación entre llamadas queda garantizada perfectamente gracias al paso de parámetros cuando se efectúa la llamada y a la devolución de resultados cuando se efectúa el retorno.

Para ilustrar la implementación de la ejecución de un método recursivo que realiza la máquina Java considérese el método recursivo `factorial`; como muestra la siguiente figura para la invocación `factorial(4)`, cualquier Registro de Activación asociado a cualquier llamada a `factorial` contiene una copia del parámetro formal `n`, cuyo valor es el del parámetro actual de la llamada y que a todos los efectos se considera como una variable local, y las variables locales `resMetodo` y `resLlamada` (`resM` y `resLl` en la figura por abreviar), que se siguen utilizando en este ejemplo con el único propósito de facilitar la discusión:



Así los distintos estados de la Pila de la Recursión al ejecutar `factorial(4)` serían los que se ilustran en las siguientes figuras, en la primera los correspondientes a la inserción de Registros en la Pila y en la segunda los correspondientes a los borrados de su último Registro.



1.1.4. Iteración VS Recursión

Como se ha visto en el apartado anterior, un método recursivo es una forma alternativa de expresar un cálculo repetitivo que podría describirse mediante un método iterativo. De hecho, se puede demostrar que a partir de un método recursivo dado se puede obtener su equivalente iterativo mediante la denominada Transformación Recursivo-Iterativa. Aunque su forma general queda fuera del alcance de esta asignatura, es muy fácil obtenerla para un método recursivo Lineal Final: el cálculo generado por la llamada a un método de este tipo se puede interpretar como el tratamiento iterativo de su Secuencia de Llamadas asociada. Así,

- a cada llamada generada le corresponde un elemento de dicha Secuencia;
- la información asociada a cada elemento de dicha Secuencia está constituida por los valores de los parámetros actuales de la llamada;
- el primer elemento de esta Secuencia corresponde a la llamada principal y el último a la llamada para el caso base;
- el elemento siguiente a uno dado es el correspondiente a la llamada que éste origina.

Por ejemplo, la versión iterativa de `maximoComunDivisor(N,M)` se obtiene como sigue:

- cada elemento de la Secuencia de Llamadas asociada a `maximoComunDivisor(N,M)` se caracteriza por los dos valores Naturales que corresponden a los parámetros formales `n` y `m` de `maximoComunDivisor`;

- el primer elemento de dicha Secuencia queda caracterizado por los valores de la llamada principal: `n = N; m = M`;
- el último elemento de dicha Secuencia queda caracterizado por los valores de la llamada para el caso base, `n == m`;
- la llamada `maximoComunDivisor(n,m)` genera `maximoComunDivisor(n-m,m)` si `(n > m)`; sino, `maximoComunDivisor(n,m-n)`. Así el siguiente elemento de la Secuencia queda caracterizado por los valores `n-m` y `m` si `(n > m)`; sino, los valores son `n` y `m-n`.

De esta forma,

```
/** N > 0 AND M > 0 */
static int maximoComunDivisorIterativo(int N, int M) {
  int n = N, m = M;
  while ( n != m )
     if ( n > m ) n = n - m; else m = m - n;
  return n;
}
```

Sin embargo, si la Recursión es No Final, en muchos casos es necesario utilizar como Estructura de Datos auxiliar una Pila; ésta almacena los Datos de las llamadas anteriores y los mantiene accesibles para la correcta aplicación del método `combinar`. Más difícil aún resulta la Transformación Recursivo-Iterativa en el caso de Recursión Múltiple, pues requiere el tratamiento del Árbol de Llamadas asociado.

A pesar de las dificultades que puede entrañar la Transformación Recursivo-Iterativa, resulta necesaria siempre que la ejecución del método recursivo suponga, frente a la del iterativo, un mayor consumo de espacio y/o tiempo. Por ejemplo, frente al coste Espacial del método recursivo `factorial` propuesto su versión iterativa requiere una reserva de memoria para sólo un par de variables enteras y, lo que es más importante, esta necesidad de espacio es independiente del valor de `n` para el que se calcula el factorial; nótese que la cantidad de espacio requerido por la versión recursiva es proporcional al número de Registros de Activación que se almacenan en la Pila, esto es, es proporcional al valor de `n` para el que se calcule el factorial.

Otro ejemplo que demuestra lo inapropiada que puede resultar una versión recursiva de un método es `fibonacci`; nótese la gran cantidad de cálculos que se repiten al realizar una invocación a `fibonacci` para un número mayor o igual que tres. Una versión iterativa que evita todas estas repeticiones es la siguiente, mucho más eficiente que la recursiva pero sin su claridad de diseño:

```
/** n >= 0 */
static int fibonacciIterativo(int n){
  int f1 = 0, f0 = 1;
  while ( f1 < n ){
     f1 = f0 + f1; f0 = f1 - f0;
  }
  return f1;
}
```

Un último ejemplo significativo sobre el uso o no de la Recursión lo ofrece la solución del problema de las Torres de Hanoi, de la que se presenta ahora una posible implementación iterativa, para confrontarla con la recursiva `hanoi` ya estudiada:

```
/** n > 0 */
static void hanoiIterativo(int n, String origen, String auxiliar, String destino){
  int h = n; String a = origen, c = destino, b = auxiliar; String k; int no = 1;
  while ( no != 0 ){
    while ( h != 2 ){ h--; k = b; b = c; c = k; no = 2 * no; }
    moverDisco(a,b); moverDisco(a,c); moverDisco(b,c);
    while ( (no % 2) != 0 ){ h++; k = a; a = b; b = k; no = no/2; }
    if ( no != 0 ){
      moverDisco(a, b);
      k = a; a = c; c = b; b = k; no++;
    }
  }
}
```

Como es evidente, la versión iterativa de Hanoi, **aún tendiendo la misma Complejidad Temporal que la recursiva**, ni es clara ni parece la solución natural que se obtiene tras la lectura del enunciado del problema. Pues bien, al igual que Hanoi, existen otros muchos problemas complejos que no se pueden abordar más que utilizando una estrategia de diseño recursiva, aunque más tarde, vía Transformación Recursivo-Iterativa, se implementen iterativamente.

Ya para finalizar la sección y a modo de conclusión se ofrece la siguiente regla general:

> **aplíquese la estrategia Inductiva únicamente a la resolución de aquellos problemas lo suficientemente complejos como para merecerlo**

En base a ella, un problema que se puede solucionar mediante un sencillo bucle no lo merece; si es algo más complicado, una vez obtenido un primer diseño recursivo y tras analizar su coste Espacial y Temporal, decídase si se obtiene su versión iterativa equivalente.

1.2. Etapas del diseño recursivo

A la hora de realizar el diseño de un método recursivo con cierta garantía de éxito, conviene considerar a modo de guía las siguientes etapas:

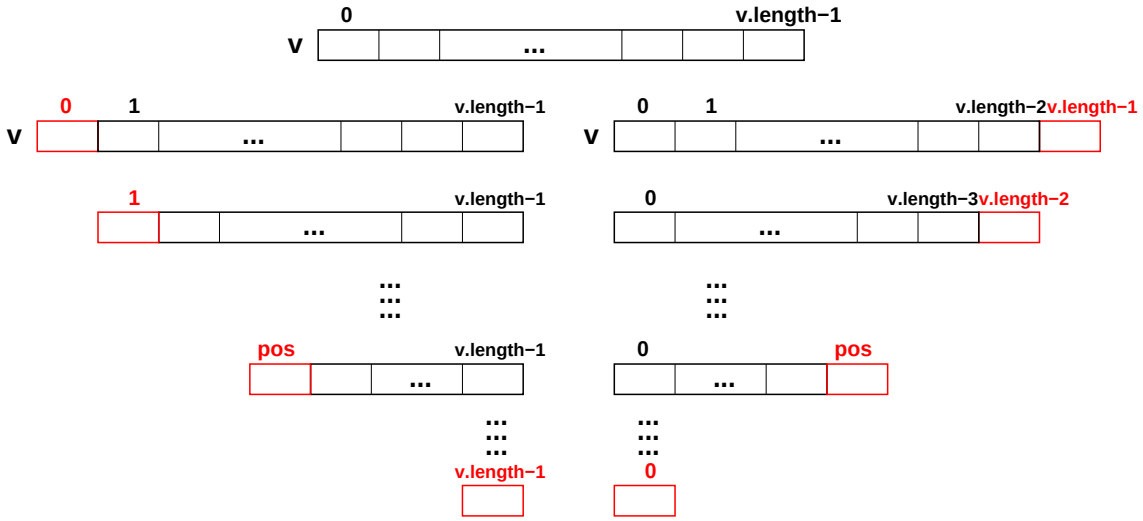
1. **Definición de la cabecera del método:** determinar su nombre, sus parámetros formales y el tipo de éstos, el tipo de su resultado y sus posibles Excepciones; además, tal como indica el esquema general recursivo, se debe especificar su Precondición pues representa las restricciones sobre el dominio de sus Datos.

   Esta primera etapa de definición es muy importante ya que en ella se pone de relieve la estructura recursiva del dominio de definición, talla del problema incluida, lo que facilita sobremanera la estrategia inductiva a seguir.
2. **Análisis de Casos:** hacer explícitos los casos base y general de la Recursión y establecer para cada uno de ellos las instrucciones pertinentes que los resuelven; en particular, hay que comprobar que cualquier elemento del dominio pertenece a uno u otro de los dos casos determinados.
3. **Transcripción del Análisis de Casos:** codificación en Java de la solución propuesta en la etapa anterior.
4. **Validación del diseño:** comprobar la coherencia y terminación de sus llamadas; se sugiere probar la terminación demostrando que la talla del problema decrece al menos una unidad en cada una de las llamadas del caso general y que alcanza la del caso base.

A continuación se ejemplifican estas etapas para el diseño de métodos recursivos de Recorrido y Búsqueda sobre un `array` genérico, que servirá de dominio de definición de tales métodos y cuya naturaleza recursiva se pone de manifiesto en la siguiente definición:

> Sea `T v[]` un `array` genérico que representa una Secuencia de `v.length` Datos de Tipo `T`, denotada como `v[0 ... v.length-1]`. Se puede definir recursivamente `v` como la Secuencia formada por la primera componente de `v`, `v[0]`, y la **sub**Secuencia definida por el resto de `v`, i.e. el `array v` menos su primer componente o **subarray** `v[1 ... v.length-1]`. Nótese que, a su vez, el **subarray** `v[1 ... v.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.

Análogamente, el `array v` se puede definir de forma recursiva y Descendente considerando el **subarray** `v[0 ... v.length-2]` y su última componente `v[v.length-1]`; en la siguiente figura se ilustra gráficamente esta descomposición, a la derecha de la Ascendente:



Naturalmente, para poder diseñar un método recursivo que realice un Recorrido o Búsqueda sobre un (**sub**)`array v` se debe disponer de tres Datos o parámetros formales del método: el propio (**sub**)`array v` 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 v[inicio...fin]` sobre las que se realiza el Recorrido o la Búsqueda, que viene dada por la expresión `x = fin-inicio+1`; por ejemplo, si en la llamada más alta a un método recursivo `inicio = 0` y `fin = v.length-1` entonces se explora todo el `array v`, pero si `inicio = (v.length-1)/2` y `fin = v.length-1` entonces el **subarray** de `v` 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 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 v` el caso base se alcanza cuando `inicio = fin+1`, i.e. para una talla `x = 0`, y la forma de progresar hacia él en cada llamada es incrementar el valor de `inicio`, con lo que la talla `x` decrece en una

unidad; así, el rango de variación de `inicio` será `0` $\leq$ `inicio` $\leq$ `v.length`, mientras que `fin` mantiene constante su valor en la llamada más alta, `fin = v.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` $\leq$ `v.length-1`, mientras que `inicio` mantiene constante su valor en la llamada más alta, `inicio = 0`.

Como aplicación de lo visto, se presentan e instancian en lo que sigue los esquemas recursivos de Recorrido y Búsqueda sobre un `array`.

1.2.1. Esquema recursivo de Recorrido de un `array`

En base a la definición de Recorrido de un `array v` y la descomposición recursiva Ascendente de `v`, el esquema recursivo de Recorrido Ascendente es el siguiente:

```
public static <T> void recorrer(T v[]){
  recorrer(v, 0, v.length-1);
}
/** 0 <= inicio <= v.length AND fin = v.length-1 **/
private static <T> void recorrer(T v[], int inicio, int fin){
  if ( inicio > fin ) visitarVacio();
  else{ visitar(v[inicio]); recorrer(v, inicio+1, fin); }
}
```

Nótese que el método público `recorrer`, el que se suele denominar **guía**, se define para ocultar la estructura recursiva del `array v` que muestran los parámetros `inicio` y `fin` de la cabecera del método homónimo pero recursivo y privado. Con ello no se hace más que definir la solución del problema de Recorrido de un `array v` de `v.length` componentes en términos del Recorrido recursivo Ascendente del **subarray** `v[inicio...fin]` de talla `x = fin-inicio+1` componentes; para hacer coincidir sus soluciones el método guía lanza o invoca al recursivo con los valores de `inicio = 0` y `fin = v.length-1`, i.e. para una talla inicial `X = v.length`.

Obsérvese también que el esquema presentado se puede simplificar substituyendo cualquier referencia al parámetro formal `fin` por su valor en la llamada más alta `v.length-1`, pues como se expresa en la Precondición `fin = v.length-1` en cualquier llamada recursiva a `recorrer`:

```
public static <T> void recorrer(T v[]){
  recorrer(v, 0);
}
/** 0 <= inicio <= v.length **/
private static <T> void recorrer(T v[], int inicio){
  if ( inicio == v.length )    visitarVacio();
  else{ visitar(v[inicio]); recorrer(v, inicio+1); }
}
```

Ahora bien, la simplificación realizada conduce a plantear un mismo problema, el del Recorrido de un `array v` de `v.length` componentes, en términos de un método recursivo **menos general**: **eliminar el parámetro `fin` supone que el método privado sólo represente la solución al problema para un tipo particular de subarray `v`**, aquel en el que forzosamente la última posición a visitar es `v.length-1`. Esta pérdida de generalidad no resulta significativa en un gran número de problemas, pero no se puede obviar a la hora de realizar diseños recursivos correctos.

Ejercicio propuesto: en el programa `TestRecorrido` se ha definido una instancia del esquema recursivo de Recorrido Ascendente en la que el método `visitar` sólo contiene la instrucción «`System.out.println(v[inicio]);`» y `visitarVacio()` no hace nada; además, el método guía `recorrer` se invoca sobre un `array` de `String v={"hoy","no","hace","sol"}`. Ejecútese dicho programa y obsérvese su resultado. Modifíquese entonces su método recursivo `recorrer` de forma que la instrucción «`System.out.println(v[inicio]));`» aparezca **detrás** de la llamada `recorrer(v,inicio+1, fin)` y no antes ¿Qué resultado se obtiene? ¿Por qué?

**`TestRecorrido` figura en la tabla de material didáctico de los Contenidos PoliformaT del tema.

Tras resolver el ejercicio anterior es fácil plantear el siguiente Esquema recursivo de Recorrido Descendente:

```
public static <T> void recorrer(T v[]){ recorrer(v, 0, v.length-1); }
/** inicio = 0 AND -1 <= fin < v.length */
private static <T> void recorrer (T v[], int inicio, int fin){
  if ( fin < inicio ) visitarVacio();
  else{ visitar(v[fin]); recorrer(v, inicio, fin - 1); }
}
```

Para simplificar este esquema, nótese, basta con eliminar de su código el parámetro `inicio`:

```
public static <T> void recorrer(T v[]){ recorrer(v, v.length-1); }
/** -1 <= fin < v.length */
private static <T> void recorrer (T v[], int fin){
  if ( fin == -1 ) visitarVacio();
  else{ visitar(v[fin]); recorrer(v, fin-1); }
}
```

Cuestión: presentados los esquemas recursivos de Recorrido, ¿cuál de ellos resulta más natural elegir para imprimir en orden inverso las componentes de un `array`?

Ya en general, la elección del esquema Ascendente o Descendente de Recorrido de un `array` para cada problema particular que se plantee, o la descomposición recursiva subyacente a realizar, debe ser aquella que permita resolverlo de la manera más natural y eficaz posible; el ejemplo que se plantea ahora servirá para terminar de ilustrar este punto.

Ejemplo: suma recursiva de las componentes de un `array` de Integer

1. **Definición de la cabecera del método:** por el enunciado del ejemplo, podría ser:

```
/** 0 <= inicio <= v.length AND fin = v.length-1 */
private static int sumar(Integer v[], int inicio, int fin){
  int resMetodo, resLlamada;
  ...
  return resMetodo;
}
// sumar(v, inicio, fin) == ∑ i = inicio, ..., fin v[i].intValue()
```

Entonces, el método guía (público) desde el que se lanza `sumar` sería:

```
public static int sumar(Integer v[]){
  return sumar(v, 0, v.length-1);
}
```

2. **Análisis de Casos:**
   - como la suma de cero elementos es cero, si el `array v` está vacío o su talla es `0`, `casoBase(x) == ( inicio > fin )` y `visitarVacio() == resMetodo = 0;`
   - sino, `!casoBase(x)`, cuando `inicio` $\leq$ `fin`, la suma de las componentes del **subarray** `v[inicio...fin]` se puede expresar como la suma de su primera componente `v[inicio]` y la de las componentes del **subarray** `v[inicio+1...fin]`. Equivalentemente,

     ```
     resLlamada = sumar(v, inicio+1, fin); resMetodo  = v[inicio] + resLlamada;
     ```

3. **Transcripción del Análisis de Casos** a Java, sin usar `fin` pues siempre vale `v.length-1`:

   ```
   /** 0 <= inicio <= v.length */
   private static int sumar(Integer v[], int inicio){
     int resMetodo, resLlamada;
     if ( inicio == v.length ) resMetodo = 0;
     else{ resLlamada = sumar(v, inicio+1); resMetodo = v[inicio] + resLlamada; }
     return resMetodo;
   }
   // sumar(v, inicio) == ∑_{i = inicio, ..., v.length-1} v[i].intValue()
   ```

   Alternativamente, escrito de forma más compacta se tendría:

   ```
   private static int sumar(Integer v[], int inicio){
     if ( inicio == v.length ) return 0; return v[inicio] + sumar(v, inicio+1)
   }
   ```

   y su método guía, público, sería el siguiente:

   ```
   public static int sumar(Integer v[]){ return sumar(v, 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` $<$ `v.length`, tras incrementarse puede llegar a alcanzar el valor `v.length`;
   - en cualquiera de los dos casos establecidos, los valores de los parámetros formales siempre cumplen la Precondición: `v` porque permanece inalterado en cualquier llamada y en cuanto a `inicio`, toma valores en el rango `[0 ... v.length]` porque su valor en la llamada más alta, `sumar(v, 0)`, se incrementa siempre en uno hasta llegar en el caso base a `v.length`.

1.2.2. Esquema recursivo de Búsqueda de un Dato `b` en un `array`

Para obtener un esquema recursivo de Búsqueda se puede plantear el siguiente Análisis de Casos: sea `v[inicio...fin]`, `0` $\leq$ `inicio` $\leq$ `v.length AND -1` $\leq$ `fin` $<$ `v.length`, el **subarray** de talla `x = fin-inicio+1` donde se busca el Dato `b`; si `x = 0`, i.e. el **subarray** está vacío, obviamente `b` no está en él y el resultado de `buscar` en su caso base es `-1`, el habitual en casos de Búsqueda fallida cuando no se lanza Excepción; si por el contrario `x` $>$ `0`, i.e. el **subarray** contiene al menos una componente, en base a una descomposición Ascendente de éste, bien `v[inicio].equals(b)` y el resultado de `buscar` es `inicio` o bien `!v[inicio].equals(b)` y se deberá continuar buscando en el **subarray** `v[inicio+1...fin]`. En base a este análisis, se propone el siguiente esquema recursiva de Búsqueda Ascendente de `b` en un `array v`:

```
public static <T> int buscar(T v[], T b){
  return buscar(v, b, 0, v.length-1);
}
/** 0 <= inicio <= v.length AND fin = v.length-1 */
private static <T> int buscar(T v[], T b, int inicio, int fin){
  int resMetodo = -1;
  if ( inicio <= fin )
   if ( v[inicio].equals(b) ) resMetodo=inicio; else  resMetodo=buscar(v,b,inicio+1,fin);
  return resMetodo;
}
//buscar(v,b,inicio,fin) == -1 AND ∀i: inicio≤i≤fin: v[i]!=b OR
 //buscar(v,b,inicio,fin) == i: inicio≤i≤fin: v[i]=b AND ∀j:inicio≤j<i: v[j]!=b
```

Obsérvese que, al igual que para el Recorrido recursivo y por los mismos motivos, el método recursivo y privado `buscar` es lanzado por el método guía homónimo y público. Nótese además 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, i.e. si se busca `b` 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 `v[inicio].equals(b)` el resultado es `inicio`; sino, se debe seguir recorriendo `v` en busca de `b` -recuérdese que en el Peor de los Casos, cuando `b` no está en `v`, cualquier Búsqueda se transforma en Recorrido.

Ejercicio propuesto: obtener los esquemas recursivos de Búsqueda Ascendente simplificada, sin parámetro `fin`, Descendente y Descendente simplificada.

Para concluir con la presentación de los esquemas recursivos, y para subrayar que la elección de un esquema Ascendente o Descendente no es arbitraria, se aborda el siguiente ejemplo.

Ejemplo: obtención de la posición de la última aparición de un Dato `b` en un `array`

El problema planteado admite en principio dos soluciones:

- eligiendo una descomposición recursiva **Ascendente** de `v`, un **Recorrido** del **subarray** `v[inicio...fin]` en el que se comprueba si cada elemento visitado es igual a `b`;
- eligiendo una descomposición recursiva **Descendente** de `v`, una **Búsqueda** de la primera aparición de `b` en el **subarray** `v[inicio...fin]`.

Aunque ambas son correctas, el criterio de eficiencia conduce a elegir la segunda solución por presentar un caso mejor de coste independiente de la talla. Para obtener entonces el método Java correspondiente los pasos son:

1. **Definición de la cabecera del método**, y la del método guía que lo lanzará:

```
/** 0 <= inicio <= v.length AND fin = v.length-1 */
private static <T> int ultimaP(T v[], T b, int inicio, int fin){
   int resMetodo; ... return resMetodo;
}
// ultimaP(v,b,inicio,fin) == -1 AND ∀i: inicio≤i≤fin: v[i] != b OR
// ultimaP(v,b,inicio,fin) == i:inicio≤i≤fin: v[i]=b AND ∀j:i<j≤fin: v[j]!=b
public static <T> int ultimaP(T v[], T b){ return ultimaP(v, b, 0, v.length-1); }
```

2. **Análisis de Casos:**
   - si `v` es un `array` vacío obviamente `b` no está en `v`. Por tanto, el caso base de `ultimaP` es aquel en que `fin` $>$ `inicio` y tiene como resultado `resMetodo = -1;`
   - sino, cuando `fin` $\geq$ `inicio`, en base a una descomposición Descendente de `v`, bien `v[fin]` es la primer aparición de `b` en `v` -por lo que el resultado de `ultimaP` es `fin` -o bien no lo es y se deberá continuar con su Búsqueda en `v[inicio...fin-1]`:

     ```
     if (v[fin].equals(x)) resMetodo = fin;
     else                  resMetodo = ultimaP(v, b, inicio, fin-1);
     ```

3. **Transcripción del Análisis de Casos** a Java, en la que no tiene sentido usar `inicio`:

   ```
   private static <T> int ultimaP(T v[], T b, int fin){
     int resMetodo = -1;
     if ( fin > -1 )
      if (v[fin].equals(b)) resMetodo = fin; else resMetodo = ultimaP(v, b, fin-1);
     return resMetodo;
   }
   ```

   Su método guía, público, sería:

   ```
   public static <T> int ultimaP(T v[], T b){ return ultimaP(v, b, v.length-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 producirse la llamada `fin` $>$ `-1`, tras decrementarse en uno puede llegar a alcanzar el valor `-1`;
   - en cualquiera de los dos casos establecidos, los valores de los parámetros formales siempre cumplen la Precondición: `v` y `b` porque permanecen inalterados en cualquier llamada y en cuanto a `fin`, toma valores en el rango `[-1 ... v.length-1]` porque su valor en la llamada más alta, `ultimaP(v, b, v.length-1)`, se decrementa siempre en uno hasta llegar en el caso base al valor `-1`.

Ejercicios propuestos: el programa `RecursionArray` contiene los métodos recursivos `ultimaP` y `sumar` presentados en esta sección. Se pide:

1. Añadir a dichos métodos las líneas de código necesarias para poder trazar su ejecución, i.e. para mostrar por pantalla la talla del problema que resuelven en cada llamada que origine su ejecución, los datos de tal llamada y sus resultados. Hecho esto, ejecutar `sumar` para el `array` de `Integer` {`1, 2, 3, 4`} y `ultimaP` para el `Integer b = 5` y los `array` {`1, 2, 3, 4`} y {`1, 2, 3, 5`}; en base a las trazas obtenidas, contestar para cada uno de los métodos las siguientes preguntas: ¿que tipo de Recursión presenta? ¿cuánto y cómo disminuye la talla en cada una de sus llamadas? ¿cuántas llamadas genera su invocación más alta?
2. Incluir en `RecursionArray` los diseños de los métodos recursivos que se especifican a continuación, con instrucciones de traza incluidas: `invertir`, que invierte *in-situ* las componentes de un `array` genérico dado `v`; `maximo`, que obtiene la posición del máximo de un `array` genérico dado `v`; `minimo`, que obtiene la posición del mínimo de un `array` genérico dado `v`; `esSumaIgualA`, que comprueba si un cierto número Natural `b` es igual a la suma de todas las componentes de un `array` de Naturales dado `v`; `capicua`, que comprueba si un `array` genérico dado `v` es capicúa.

**`RecursionArray` figura en la tabla de material didáctico de los Contenidos PoliformaT del tema.

1.3. Coste Espacial y Temporal de un método y su análisis

En esta última sección del apartado se revisan todos los conceptos relacionados con el coste Espacial y Temporal de un método, el criterio efectivo que permite comparar tal método con cualquier otro que resuelva el mismo problema y, por tanto, sin cuyo análisis o estimación no puede darse por concluido su diseño. Tal revisión sirve como imprescindible preámbulo al siguiente apartado, en el que el criterio de coste se incorporará al proceso de diseño para poder exigir que el método que de él resulte sea no sólo correcto sino el más eficiente de todos los que resuelven un determinado problema.

1.3.1. Revisión de conceptos y notación

El criterio fundamental que permite comparar los distintos métodos que resuelven un mismo problema es su eficiencia, o el consumo de recursos que su ejecución provocará: el método más eficiente, el mejor, será aquel que menos recursos requiera para su ejecución. Visto que los recursos básicos de un ordenador son la memoria (RAM) y el tiempo de CPU, la eficiencia de un método se expresa en términos de su **coste Espacial** o medida del espacio que ocupa en memoria a lo largo de su ejecución y su **coste Temporal** o medida del tiempo empleado por éste para ejecutarse y dar un resultado a partir de los Datos de entrada.

En general, el coste de un método dependerá de dos tipos de factores: los propios del problema que resuelve -su talla e instancias y la estrategia de resolución que implementa- y los dependientes del entorno de Programación donde se ejecutará finalmente -tipo de ordenador, carga del sistema, lenguaje y compilador o intérprete, etc. Por tanto, las dos aproximaciones al estudio de la eficiencia de un método son: el análisis Teórico o *a-priori*, consistente en obtener una expresión matemática (cota) del coste del método en función de los factores propios del problema que resuelve, y el análisis Experimental o *a-posteriori*, consistente en medir los tiempos de ejecución del método para diversos casos de prueba (entradas) en un ordenador, lenguaje y compilador concretos. Evidentemente un tipo de análisis no excluye al otro, son complementarios, pero sí que hay un orden de precedencia entre ellos: si el análisis Teórico ya indica que el coste intrínseco del método es impracticable o peor que el de otro método existente que resuelve el mismo problema no será necesario implantarlo en sistema alguno, lo que supone un ahorro de tiempo y esfuerzo considerable. Es por ello que en lo que sigue sólo se tratarán los aspectos relacionados con el análisis Teórico o *a-priori* del coste de un método, dejando para las prácticas el análisis *a-posteriori*.

Coste a-priori de un método

El coste *a-priori* de un método es una medida de los recursos (tiempo o memoria) que éste emplea para ejecutarse **independiente del entorno de programación** donde lo haga. Esta definición plantea una pregunta obvia: ¿cómo medir el tiempo o el espacio que un método emplea en su ejecución si éste no se está ejecutando en máquina alguna? La respuesta es que el coste *a-priori* de un método, Temporal o Espacial, no es un valor, sino una función matemática que expresa el tiempo o la memoria que su ejecución requiere en función de los factores propios de éste; a dicha función se le denomina **Complejidad**, Temporal o Espacial, quedando la sutil diferencia entre los términos Complejidad y coste oculta tras el abuso del lenguaje que se realiza en la práctica, donde un término se utiliza por otro sin más.

Como en su día se discutió en Programación de primer curso, la unidad de tiempo en la que se expresa el coste Temporal de un método es **el paso de programa** y, por tanto, cualquier método concreto de cálculo de la función Complejidad consiste en contar los pasos de programa que éste realiza durante su ejecución. Si se recuerda, para un método iterativo el cálculo de su Complejidad Temporal se basa en suponer que su **instrucción crítica**, i.e. la que se ejecuta por lo menos con tanta frecuencia como cualquier otra del método, tarda en ejecutarse un paso de programa; de donde el método tardará tantos pasos de programa como número de veces se repita su **instrucción crítica**. En el caso de un método recursivo, el planteamiento utilizado ocasionalmente hasta ahora era el siguiente: si una llamada recursiva tarda en ejecutarse un número dado de pasos de programa, `x` por ejemplo, el método tardará en ejecutarse `x` multiplicado por el número de llamadas que origina su llamada más alta (calculable por Inducción); como es fácil deducir, este producto no se precisará completamente hasta que no se determine el número de pasos de programa que tarda una llamada recursiva en ejecutarse, lo que se estudiará un poco más adelante.

Así mismo, y como también se discutió el curso pasado, el coste o número de pasos de programa que un método realiza durante su ejecución depende de dos factores: **el tamaño o talla del problema** y, para una talla dada, **la instancia del problema**.

El primero de ellos, la talla del problema, se define como la cantidad o magnitud de los Datos que se deben procesar y, como su nombre indica, es más una característica del problema que se quiere resolver y no tanto del método que se emplee para ello; por ejemplo e independientemente de la estrategia de resolución, la talla del problema de la Ordenación es el número de componentes del `array` que se desea ordenar, la del cálculo del factorial de un número es la magnitud de tal número, etc. Es por ello que la Complejidad Temporal (Espacial) de un método `a` se define como una función positiva no decreciente del tiempo (espacio) que `a` necesita para ejecutarse en base a la talla `x` del problema, lo que se denota como $T_a(x)$ generalmente.

En cuanto al segundo factor mencionado, una instancia de un problema es aquel conjunto de configuraciones de talla dada de sus Datos para las que el comportamiento del método que lo resuelve, y por tanto su coste, es el mismo. Así por ejemplo el problema de Recorrido de un `array` de talla dada presenta una única instancia, el conjunto de todas y cada una de las posibles presentaciones de las componentes del `array`, pues el tiempo que se emplea para resolverlo no depende de la presentación particular de las componentes del `array` sino de lo que se tarda en visitar cada una de ellas; sin embargo, si sobre el mismo `array` se realiza una Búsqueda entonces el número de instancias del problema es la talla del `array` más uno, pues además de aquella en la que el Dato a buscar no está en el `array` existen tantas instancias en las que el Dato que se busca sí está en el `array` como posiciones válidas tenga éste (puede ser el primero del `array` o el segundo o `...` o el último).

En función de lo expuesto, si un método resuelve un problema que para una talla dada sólo presenta una instancia entonces su coste Temporal es el número de pasos de programa que requiere la ejecución de tal instancia, que se calculará de una forma u otra según el método sea recursivo o iterativo. Si por el contrario se ha detectado más de una instancia para una talla dada su coste se establece calculando su Complejidad en los siguientes casos:

- **En el Peor de los Casos**, o para aquella instancia que obliga a que el método `a` realice el máximo número de cálculos, i.e. instancia que da lugar al Peor coste; en este caso la Complejidad Temporal se denota $T_a^P(x)$.

  A partir de esta definición es fácil deducir que el comportamiento del método `a` jamás puede ser peor que el que presenta en su Peor de los Casos, para ninguna otra instancia; por tanto, y esto es lo realmente importante, $T_a^P(x)$ **es una cota superior** del coste (comportamiento) Temporal general del método `a`.

- **En el Mejor de los Casos**, o para aquella instancia que obliga a que el método en cuestión realice el mínimo número de cálculos, i.e. instancia que da lugar al Mejor coste; en este caso la Complejidad Temporal se denota $T_a^M(x)$.

  De nuevo es fácil deducir que el comportamiento del método `a` jamás será mejor que el que presenta en su Mejor de los Casos, para ninguna otra instancia, de donde el coste $T_a^M(x)$ **es una cota inferior** del coste (comportamiento) Temporal general del método `a`; nótese que aunque no es baladí, esta información no resulta tan significativa como la que proporciona $T_a^P(x)$.

- **En Promedio o coste Medio**, que se define como la media de la Complejidad del método `a` para todas y cada uno de sus instancias. En este caso, la Complejidad Temporal se denotará $T_a^\mu(x)$, y representará el comportamiento medio del método; desgraciadamente, a pesar de que este coste es el que más información proporciona sobre el comportamiento de un método, su cálculo es en general difícil de realizar: no sólo exige conocer todas las instancias del problema y su Complejidad, sino también, y aquí reside el problema principal, la probabilidad con la que cada una de éstas se dan.

Coste Asintótico de un método y su notación

Como se acaba de enunciar, el coste *a-priori* de un método se expresa como una función no decreciente de la talla del problema, de donde se podría deducir que comparar costes es comparar directamente tales funciones. Sin embargo, si se atiende a las siguientes consideraciones, tal conclusión resulta precipitada y se puede mejorar sensiblemente:

- la determinación de la expresión exacta de la función Complejidad, además de una tarea difícil, no tiene mucho sentido si se considera que distintas decisiones sobre qué es un paso de programa conducen a funciones de coste que sólo difieren en constantes multiplicativas o aditivas. Es más, como se argumenta en los dos items siguientes, tales diferencias no resultan significativas a la hora de establecer si, para tallas suficientemente grandes e independientemente del entorno de Programación, un determinado método se comporta mejor, peor o equivalentemente que otro;
- para valores suficientemente grandes de la talla, el valor de la función Complejidad está completamente determinado por su término dominante;
- el valor exacto del coeficiente del término dominante de la función Complejidad no se conserva al cambiar de entorno de programación.

A partir de lo anterior, es fácil obtener las siguientes conclusiones:

1. Cuando se analiza el coste *a-priori* de un método determinado, el objetivo es conocer, más que la expresión exacta de su función Complejidad, qué tipo de dependencia muestra con la talla del problema o su **Tasa de Crecimiento**. Se habla entonces de **coste Asintótico** o **Complejidad Asintótica**, según se haga referencia, respectivamente, al comportamiento de un método o a la expresión formal que lo representa; aunque la diferenciación entre ambos términos se debe conocer, como es habitual, en la práctica se utilizan indistintamente en un claro abuso del lenguaje.
2. Comparar costes es comparar las Tasas de Crecimiento de las funciones Complejidad involucradas; por ejemplo, si el coste de un método viene expresado mediante una función Complejidad que es un polinomio, su monomio de mayor grado es el que determina su Tasa de Crecimiento o Complejidad Asintótica (constante, lineal, cuadrática, cúbica, etc.)
3. La **notación Asintótica** es la herramienta formal que permite expresar de forma sencilla y clara el coste Asintótico de un método: como, recuérdese, $O(f(x))$ define el conjunto de funciones en x que son como máximo del orden de $f(x)$, $\Omega(f(x))$ el de las que son como mínimo del orden de $f(x)$ y $\Theta(f(x))$ el de las que son exactamente del orden de $f(x)$, entonces, dado un método a, su coste Asintótico se puede obtener para cualquiera de sus instancias como sigue:
   - Obténgase a partir de $T_a^P(x)$ una función de cota $f(x)$ que sea exactamente de su mismo orden, que la acote superior e inferiormente, lo que se nota $T_a^P(x) \in \Theta(f(x))$; si no fuera posible obtener una cota tan precisa, $f(x)$ será al menos una cota superior de $T_a^P(x)$, esto es $T_a^P(x) \in O(f(x))$.

     Una vez obtenido el coste Asintótico en el Peor de los Casos, se puede deducir que para cualquier instancia a no se puede comportar peor que en el Peor de los Casos o, lo que es lo mismo, que $T_a(x) \in O(f(x))$.
   - Obténgase a partir de $T_a^M(x)$ una función de cota $f(x)$ tal que $T_a^M(x) \in \Theta(f(x))$; si no fuera posible obtener una cota tan precisa, al menos $T_a^M(x) \in \Omega(f(x))$.

     A partir de este resultado, como para cualquier instancia a no se puede comportar mejor que en el Mejor de los Casos, $T_a(x) \in \Omega(f(x))$.
   - En el caso de que el método a no presente instancias significativas, o que coincidan las cotas para el Peor y Mejor de sus Casos, entonces $T_a(x) \in \Theta(f(x))$.

Para terminar, a modo de recordatorio y para su uso posterior, en la siguiente tabla se muestran las funciones de cota más frecuentes en el análisis del coste Asintótico de un método -función, nombre y notación Asintótica; el hecho de que aparezcan ordenadas ascendentemente facilitará la comparación por eficiencia de los diversos métodos que resuelven un mismo problema.

Función	Nombre	Notación Asintótica
c	constante	$\Theta(1)$
$\log x$	logarítmica	$\Theta(\log x)$
$\log^2 x$	logarítmica al cuadrado	$\Theta(\log^2 x)$
x	lineal	$\Theta(x)$
$x * \log x$	$x * \log x$	$\Theta(x * \log x)$
x^2	cuadrática	$\Theta(x^2)$
x^3	cúbica	$\Theta(x^3)$
2^x	exponencial	$\Theta(2^x)$

Metodología general para el análisis del coste Temporal de un método

La aplicación ordenada y sistemática de los conceptos revisados hasta ahora proporciona una metodología general para el análisis del coste Temporal de un método dado. Tal metodología se presenta enunciando los pasos que la conforman y aplicándolos al análisis del coste del método de **Fusión** o **Mezcla Natural**, que dados dos `array a` y `b` de `int` ordenados ascendentemente construye uno nuevo también ordenado:

```
public static int[] fusion(int a[], int b[]){
  int max = a.length+b.length; int res[] = new int[max]; int i = 0, j = 0, k = 0;
  while ( i < a.length ) && ( j < b.length ) ){
   if ( a[i] < b[j] ) res[k] = a[i++]; else res[k] = b[j++];
   k++;
  }
  for ( int r = i; r < a.length; r++ ) res[k++] = a[r];
  for ( int r = j; r < b.length; r++ ) res[k++] = b[r];
  return res;
}
```

El análisis del coste Temporal de un método, y en particular el de `fusion`, se realiza siguiendo los pasos que se enuncian a continuación:

1. **Determinar la talla del problema que resuelve el método a analizar** y, siempre que sea posible, expresarla en términos de sus parámetros; para ello se deberá estudiar, básicamente, cómo y en función de qué parámetros se obtiene el resultado del método.

   Así por ejemplo, para determinar la talla del problema que resuelve `fusion` resulta imprescindible analizar cómo se construye `res`, el `array` resultado de la fusión de `a` y `b`: en un primer bucle, mientras queden componentes de `a` o `b` por visitar, en cada iteración `k` se deposita en `res` la `k`-ésima menor componente de la unión de `a` y `b`, `a[i]` o `b[j]`, pues por Precondición tanto `a` como `b` están ordenados ascendentemente; una vez termina el Recorrido de uno de los dos `array`, las componentes del otro se depositan directamente en `res` mediante un segundo bucle, pues obviamente sólo pueden ser mayores que las que ya forman parte de `res`. Por tanto, la talla del problema que resuelve `fusion`, o el número de datos que se deben manipular para construir su resultado, es la de `res`, la suma de las tallas de `a` y `b` dada por la expresión `x = a.length + b.length`.
2. **Determinar las posibles instancias significativas del problema**: se evalúa si la Complejidad del método varía, para una talla dada del problema, en función de la presentación de los Datos y, si es así, se identifican su Peor y Mejor de los Casos.

   En el caso de `fusion` queda patente de la discusión anterior que la construcción de `res` **no** presenta instancias significativas, pues siempre supone visitar todos y cada uno de los datos que componen `a` y `b`, independientemente de su presentación.
3. **Determinar el coste Asintótico del método** en dos fases: en la primera, calcular y acotar la Complejidad del método para cada instancia significativa detectada, $T_a(x)$ si no tiene ninguna y las funciones $T_a^P(x)$ y $T_a^M(x)$ para el Peor y Mejor de los Casos respectivamente; en la segunda, establecer el coste Asintótico del método como sigue:
   - si no existen instancias significativas o bien coinciden sus cotas de Complejidad, entonces $T_a(x) \in \Theta(f(x))$;
   - sino, en el caso en que $T_a^P(x) \in \Theta(f(x))$ y $T_a^M(x) \in \Theta(g(x))$ se tiene respectivamente que $T_a(x) \in O(f(x))$ y $T_a(x) \in \Omega(g(x))$.

Para `fusion`, al no haber instancias significativas el cálculo de su función Complejidad se realiza como sigue: si se toma como instrucción crítica `k++`, pues representa la posición de la siguiente componente de `res` a construir, se tiene que $T_{fusion}(x) = \Sigma_{k=0...x}\ 1 = x-1$, un polinomio que se puede acotar tanto superior como inferiormente por `x`, i.e. por su monomio de mayor grado. Por tanto, $T_{fusion}(x) \in \Theta(x)$.

A las etapas que se acaban de presentar cabe añadir una última en el caso de que se esté analizando el coste de un método recursivo: tras haber obtenido su coste *a-priori* se hace imprescindible valorar detenidamente si la cantidad de recursos extra, sobretodo de memoria RAM, que su ejecución requiere supone un coste extra lo suficientemente elevado, y que claro está no tiene su equivalente iterativo, como para desaconsejar su uso.

1.3.2. Expresión de la Complejidad Temporal de un método recursivo: Ecuaciones o Relaciones de Recurrencia

Como ya se ha indicado, la expresión y cálculo del coste Espacial y Temporal de un método recursivo se establece a partir de la expresión y cálculo del coste de una cualquiera de las llamadas que provoca la invocación de su ejecución. Específicamente, la función Complejidad Espacial de un método recursivo se expresa como la suma de las Complejidades de cada una de las llamadas recursivas que pueden coexistir en memoria como resultado de su llamada principal. Por ejemplo, sea el método recursivo `factorial` como sigue:

```
static int factorial(int n){
   if ( n == 0 ) return 1;
   return n * factorial(n-1);
}
```

Como `int n` es el parámetro que representa la talla del problema, la magnitud del valor del que se calcula el factorial, suponiendo que el número de posiciones de memoria que ocupa dicho `n` es constante e igual al que ocupa un `int`, `k` por ejemplo, la Complejidad Espacial Asintótica de una llamada a `factorial` será `k`; por tanto, si `n = 0` en la llamada para el caso base y `n = N` en la llamada principal o más alta, la Complejidad Espacial Asintótica del método `factorial` vendrá dada el sumatorio $\sum_{n=0...N}\ k = k*(N+1) \in \Theta(N)$ o, equivalentemente, será estrictamente del orden de `N`. Nótese que en una estimación experimental o *a-posteriori* del coste Espacial del método `factorial` `N` será el valor concreto con el que se realiza su llamada principal, `k` será el tamaño exacto del Registro de Activación que representa cada una de sus llamadas y, por tanto, la multiplicación de ambos valores será su coste Espacial efectivo.

En cuanto a la Complejidad Temporal de un método recursivo, para establecerla de forma general se utilizará el siguiente esquema:

```
public static TipoResultado metodoRecursivo( TipoDatos x ){
   TipoResultado resMetodo, resLlamada1, resLlamada2,..., resLlamadaK;
   if ( casoBase(x) ) resMetodo   = solucionBase( x );
   else{              resLlamada1 = metodoRecursivo( anterior1( x ) );
                      ...
                      resLlamadaK = metodoRecursivo( anteriorK( x ) );
                      resMetodo   = combinar( x, resLlamada1, ..., resLlamadaK );
   }
   return resMetodo;
}
```

Nótese que tal como ha sido definido su cuerpo, la Complejidad Temporal de cualquier llamada a `metodoRecursivo` sólo puede ser:

- bien la de la llamada que origina su caso base, cuando $\texttt{x} = \texttt{x}_{\texttt{base}}$ ó `casoBase(x)`, igual a la Complejidad Temporal de `solucionBase(x)` o número de pasos de programa que requiere obtener `resMetodo if ( casoBase(x) )`; utilizando la notación habitual se tendría que

$$\texttt{T}_{\texttt{metodoRecursivo}}(\texttt{x}_{\texttt{base}}) = \texttt{T}_{\texttt{solucionBase}}(\texttt{x}_{\texttt{base}});$$

- bien la de la llamada que origina su caso general, cuando $\texttt{x} > \texttt{x}_{\texttt{base}}$ ó `!casoBase(x)`, que depende de los siguientes factores: el número de llamadas recursivas $\texttt{a} = \texttt{K} \geq 1$ que se realizan para obtener el resultado del método para una talla `x`, la talla del problema en cada una de ellas, que si y sólo si se supone igual para todas se puede denotar como `anterior(x)` tal que $\texttt{anterior(x)} < \texttt{x}$, y, finalmente, la denominada **sobrecarga** o coste de todas las operaciones que, salvo las $\texttt{a} = \texttt{K}$ llamadas, se han de realizar para que `metodoRecursivo` obtenga `resMetodo` para la talla `x`. En lo que sigue, $\texttt{T}_{\texttt{restoOps}}(\texttt{x})$ denotará la sobrecarga, es decir la suma de los costes de evaluar si `!casoBase(x)`, calcular `anterior(x)` y `combinar` el resultado de cada una de las `K` llamadas realizadas con `x` para obtener y devolver `resMetodo`. Específicamente, la Complejidad de una llamada de talla $\texttt{x} > \texttt{x}_{\texttt{base}}$, se puede expresar en base a los factores enunciados como sigue:

$$\texttt{T}_{\texttt{metodoRecursivo}}(\texttt{x} > \texttt{x}_{\texttt{base}}) = \texttt{a} * \texttt{T}_{\texttt{metodoRecursivo}}(\texttt{anterior(x)}) + \texttt{T}_{\texttt{restoOps}}(\texttt{x})$$

Las dos ecuaciones que se acaban de presentar se denominan **Relaciones o Ecuaciones de Recurrencia** de `metodoRecursivo` y expresan la Complejidad Temporal de cualquiera de sus llamadas para una talla e instancia dadas. Instanciando `metodoRecursivo` a un método concreto se pueden obtener las Ecuaciones de Recurrencia que expresan su Complejidad Temporal; así por ejemplo, para `factorial` se tendría que

$$\texttt{T}_{\texttt{factorial}}(\texttt{x=0}) = \texttt{k'};\ \texttt{T}_{\texttt{factorial}}(\texttt{x>0}) = 1 * \texttt{T}_{\texttt{factorial}}(\texttt{x}-1) + \texttt{k}$$

donde,

- la constante $\texttt{k'} \geq 1$ representa el número de pasos de programa que se realizan para que `factorial` devuelva `1` cuando $\texttt{x} = 0$;
- la constante $\texttt{a} = 1$ es el número de llamadas en secuencia que efectúa `factorial` en su caso general, pues presenta un tipo de Recursión Lineal;
- $\texttt{anterior(x)} = \texttt{x}-1 < \texttt{x}$, pues la talla del problema decrece aritméticamente en cada llamada o restándole `1` a `n`;
- $\texttt{T}_{\texttt{restoOps}}(\texttt{x}) = \texttt{k} \geq 1$ es la sobrecarga constante que resulta de multiplicar por `n` el resultado de la llamada `factorial(n-1)` y devolverlo como resultado cuando $\texttt{n} > 0$.

Para resolver las Relaciones planteadas, o las de cualquier otro método recursivo que instancie a `metodoRecursivo`, se pueden aplicar los métodos de resolución de recurrencias estudiados en la asignatura **Análisis Matemático** de primer curso, cuando se mostró el comportamiento de las sucesiones y se aplicaron sus propiedades a la determinación de órdenes de magnitud y a la resolución de recurrencias; acotando su solución se obtendrá el coste Temporal Asintótico del método en cuestión para una instancia y talla dadas.

2. El coste como criterio de diseño: teoremas de coste y su aplicación al desarrollo de estrategias recursivas eficientes

Hasta ahora el coste de un método se ha utilizado como criterio de comparación *a-posteriori*, esto es como un indicador de la bondad de un método ya diseñado frente a otros que resuelven el mismo problema. Sin embargo es mucho más ventajoso utilizarlo para comparar las distintas estrategias de diseño que se pueden emplear para resolver un mismo problema, o como criterio de comparación *a-priori*, pues con ello se evita el baldío esfuerzo de transcribir y validar una estrategia ineficaz y, más importante aún, se pueden establecer las características que debe poseer la más eficiente o mejor de todas las estrategias que resuelven un problema dado.

Obviamente, no es posible comparar costes de estrategias recursivas sin incorporar el análisis y cálculo del coste al proceso de diseño o, equivalentemente, hacer del coste o la eficiencia un criterio de diseño. Para lograrlo basta recordar que tras la etapa de Análisis de Casos de un cierto `metodoRecursivo` ya se ha establecido una estrategia a partir de, exactamente, los mismos factores que definen las Ecuaciones de Recurrencia que expresan su Complejidad:

- la talla `x` e instancias significativas del problema a resolver;
- el número `a` de llamadas en secuencia que se realizan en el caso general, $a = 1$ si la estrategia recursiva es Lineal o $a > 1$ si es Múltiple;
- el tipo de reducción de la talla en cada llamada, aritmética cuando a `x` se le resta una cierta cantidad constante $c \geq 1$ (i.e. `anterior(x) = x - c`) o geométrica si `x` se divide por $c \geq 2$ (i.e. $anterior(x) = \frac{x}{c}$);
- la sobrecarga en cada llamada, $T_{restoOps}(x)$, que puede ser una constante $b \geq 1$ independiente de la talla o una función lineal de la talla $b * x + d$ con $b \geq 1$ y $d \geq 1$.

Así, para decidir si vale la pena continuar desarrollando la estrategia recursiva planteada para un cierto `metodoRecursivo` o es mejor plantear otra alternativa más eficiente simplemente hay que plantear, resolver y acotar la solución de las Relaciones de Recurrencia que expresan su Complejidad. Ahora bien, ¿cómo determinar si es posible aplicar una estrategia alternativa y, en su caso, si dicha estrategia es la óptima? La respuesta se puede obtener analizando y confrontando los cuatro **teoremas de coste** que se presentan a continuación, que además permiten obtener el coste Temporal Asintótico de una estrategia recursiva sin necesidad de resolver y acotar la solución de sus Ecuaciones de Recurrencia asociadas:

- **Teorema 1:** sea $T_{metodoRecursivo}(x) = a * T_{metodoRecursivo}(x-c) + b$, entonces
  si $a = 1$ $T_{metodoRecursivo}(x) \in \Theta(x)$ **y si** $a > 1$ $T_{metodoRecursivo}(x) \in \Theta(a^{\frac{x}{c}})$
- **Teorema 2:** sea $T_{metodoRecursivo}(x) = a * T_{metodoRecursivo}(x-c) + b * x + d$, entonces
  si $a = 1$ $T_{metodoRecursivo}(x) \in \Theta(x^2)$ **y si** $a > 1$ $T_{metodoRecursivo}(x) \in \Theta(a^{\frac{x}{c}})$
- **Teorema 3:** sea $T_{metodoRecursivo}(x) = a * T_{metodoRecursivo}(\frac{x}{c}) + b$, entonces
  si $a = 1$ $T_{metodoRecursivo}(x) \in \Theta(\log_c x)$ **y si** $a > 1$ $T_{metodoRecursivo}(x) \in \Theta(x^{\log_c a})$
- **Teorema 4:** sea $T_{metodoRecursivo}(x) = a * T_{metodoRecursivo}(\frac{x}{c}) + b * x + d$, entonces
  si $a < c$, $T_{metodoRecursivo}(x) \in \Theta(x)$, **si** $a = c$, $T_{metodoRecursivo}(x) \in \Theta(x * \log_c x)$
  y si $a > c$, $T_{metodoRecursivo}(x) \in \Theta(x^{\log_c a})$

Nótese ahora que sólo se pueden aplicar los Teoremas 1 y 3 a Ecuaciones de Recurrencia cuya sobrecarga sea una constante b independiente de la talla y los Teoremas 2 y 4 a las que tengan una sobrecarga lineal con la talla; asimismo, sólo se pueden aplicar los Teoremas 1 y 2 a Ecuaciones de Recurrencia cuya reducción de la talla sea aritmética y los Teoremas 3 y 4 a las que la tengan geométrica. Por ejemplo, el orden de la Ecuación de Recurrencia asociada con el caso general del método `factorial`, $T_{factorial}(x>0) = 1 * T_{factorial}(x-1) + k$, sería $T_{factorial}(x>0) \in \Theta(x)$ por el Teorema 1 con $a = c = 1$ y sobrecarga $b = k \geq 1$ constante.

Ejercicio propuesto: obténganse las Ecuaciones de Recurrencia que expresan la Complejidad Temporal de los métodos diseñados en `RecursionNumerica` y `RecursionArray` y resuélvanse utilizando alguno de los teoremas anteriores.

Una vez presentados, en las siguientes secciones de este apartado se analizan y confrontan los teoremas de coste para obtener dos estrategias recursivas que permiten resolver de forma óptima conocidos e importantes tipos de problemas: **Reducción Logarítmica** para Recursión Lineal y **Divide y Vencerás** para Recursión Múltiple.

2.1. Reducción Aritmética VS Geométrica de la talla para la Recursión Lineal: la estrategia de Reducción Logarítmica y algunos ejemplos de su aplicación

De la confrontación de los **Teoremas** 1 y 3 se deduce que la estrategia de diseño óptima de un método recursivo Lineal con sobrecarga constante es aquella en la que se hace decrecer la talla del problema geométricamente, pues se obtiene una cota logarítmica en lugar de la lineal que proporcionaría una disminución aritmética; de ahí el nombre de Reducción Logarítmica que recibe esta estrategia, que en lo que sigue se utilizará para diseñar los métodos recursivos más eficientes que resuelven una serie de conocidos problemas numéricos y de Búsqueda.

Búsqueda de un Dato `b` en un `array v` ordenado Ascendentemente

Tras una breve reflexión sobre el problema enunciado, la estrategia recursiva que de forma más inmediata se puede proponer es la de Búsqueda Ascendente de la posición de la primera componente de `v` que sea mayor o igual que `b`; nótese que esta doble condición de Búsqueda es factible pues `v` está ordenado ascendentemente, lo que para determinadas instancias del problema permite establecer que `b` no está en `v`, o equivalentemente entregar como resultado de la Búsqueda el valor `-1` antes de alcanzar el caso base de la Recursión.

Como es fácil deducir, el tipo de Recursión que establece esta estrategia es Lineal Final: si se busca a `b` en `v[inicio ... v.length-1]`, de talla `x = v.length-inicio` > 0, y `v[inicio]` es menor que `b` la Búsqueda deberá proseguir en el **subarray** `v[inicio+1 ... v.length-1]`, cuya talla es menor que la del anterior en una unidad, hasta que bien se alcanza el caso base o bien se encuentra la componente buscada. Por tanto, si se busca `b` en `v[0 ... v.length-1]` y, además, `b` es mayor estricto que todas las componentes de `v`, el coste de la Búsqueda en el Peor de los Casos $T_{buscar}{}^{P}(x)$ vendrá dado por el **Teorema** 1 con $a = 1 = c = 1$ y será $\Theta(v.length)$, por lo que para cualquier otra instancia $T_{buscar}(v.length) \in O(v.length)$.

Ahora bien, el criterio de eficiencia y las condiciones del problema indican que una Reducción Logarítmica puede mejorar la estrategia definida; como la opción más natural y eficiente para que la talla decrezca en cada llamada geométricamente es dividirla por `c` = 2, la estrategia a seguir será la siguiente: sea el **subarray** de Búsqueda `v[inicio...fin]` de talla `x = fin-inicio+1` y sea su posición central `mitad` = $\frac{(\texttt{inicio+fin})}{2}$ la que establece una descomposición recursiva del **subarray** de Búsqueda en dos **subarray** `v[inicio...mitad]` y `v[mitad...fin]` de la misma talla, exactamente $\frac{x}{2}$. A partir de esta descomposición recursiva, la Búsqueda de `b` en `v[inicio...fin]` se resolverá en base al resultado de la comparación entre `v[mitad]` y `b` como sigue:

- si `v[mitad].compareTo(b) == 0` termina la Búsqueda y su resultado es `mitad`, la posición de la primera aparición de `b` en el sub`array` de Búsqueda;
- si `v[mitad]).compareTo(b)` $>$ `0`, como `v[inicio...fin]` está ordenado ascendentemente, la Búsqueda de `b` se debe realizar tan sólo en el sub`array` `v[inicio...mitad-1]`;
- si `v[mitad].compareTo(b)` $<$ `0` la Búsqueda de `b` se debe realizar tan sólo en el sub`array` `v[mitad+1...fin]`;

Nótese que los casos expuestos sólo son admisibles cuando la talla del **subarray** de Búsqueda es mayor que cero; obviamente, en un `array` vacío no se puede encontrar `b` por lo que el resultado de la Búsqueda será, sin más, `-1`. De hecho, este es el caso base de la Recursión, al que se llegará siempre que `b` no se encuentre en el **subarray** de Búsqueda; si es mayor que todas sus componentes no se encontrará al buscarlo en `v[mitad+1...fin]` y si es menor fallará la Búsqueda en `v[inicio...mitad-1]`.

Intuitivamente, el coste de la estrategia de Reducción Logarítmica que se acaba de presentar es mejor que el de la Búsqueda Ascendente **porque uno de los dos subproblemas en los que se divide el problema original ... ¡no es tal!** Formalmente, aplicando el **Teorema** 3 con `a`=1 y `c`=2 se tiene que $\texttt{T}_{\texttt{buscarBin}}{}^{\texttt{P}}\texttt{(x>0)} \in \Theta\texttt{(log}_2\texttt{x)}$, de donde para cualquier otra instancia $\texttt{T}_{\texttt{buscarBin}}\texttt{(x>0)} \in \texttt{O(log}_2\texttt{x)}$. Por tanto, siguiendo las etapas del diseño recursivo, cabe refinar esta estrategia hasta convertirla en un método Java como el siguiente:

```
public static <T extends Comparable<T>> int buscarBin(T v[], T b){
   return buscarBin(v, b, 0, v.length-1);
}
/** ∀ i: inicio≤i<fin: v[i]≤v[i+1] AND 0≤inicio≤v.length AND -1≤fin<v.length **/
private static <T extends Comparable<T>> int buscarBin(T v[], T b, int inicio, int fin){
   int resMetodo = -1;
   if ( inicio <= fin ){
         int mitad = (inicio + fin)/2; int comparacion = v[mitad].compareTo(b);
         if ( comparacion < 0 )      resMetodo = buscarBin(v, b, mitad+1, fin);
         else if ( comparacion > 0 ) resMetodo = buscarBin(v, b, inicio, mitad-1);
              else                   resMetodo = mitad;
   }
   return resMetodo;
}
 // buscarBin(v,b,inicio,fin) == -1 AND ∀i: inicio≤i≤fin: v[i] != b OR
// buscarBin(v,b,inicio,fin) == i: inicio≤i≤fin: v[i] = b
```

Ejercicios propuestos:

1. Compruébese la terminación y coherencia de las llamadas del método `buscarBin`.
2. Analícese para una talla dada qué tipo de presentaciones de las componentes del subarray `v[inicio...fin]` de `buscarBin` originan su Peor y Mejor de los Casos. Establézcanse entonces las Ecuaciones de Recurrencia para el Peor de los Casos y determínese el coste Asintótico *a-priori* de `buscarBin` en sus casos Mejor y Peor. En base al resultado obtenido, ¿qué se puede decir del comportamiento general de `buscarBin`?
3. Los métodos recursivos `buscar` y `buscarBin` se invocan en distintas ocasiones desde el `main` del programa `TestBuscarArrayOrd` que los contiene. Añádanse a dichos métodos las líneas de código necesarias para poder trazar su ejecución, i.e. para mostrar por pantalla la talla del problema que resuelven en cada llamada que origine su ejecución, los datos de tal llamada y sus resultados. Hecho esto, ejecutar el `main` de `TestBuscarArrayOrd` y contestar a las siguientes preguntas: en el Peor de los Casos de cada uno de estos métodos, ¿que tipo de Recursión presenta? ¿cuánto y cómo disminuye la talla en cada una de sus llamadas? ¿cuántas llamadas genera su invocación más alta?.

   **`TestBuscarArrayOrd` figura en la tabla de material didáctico de los Contenidos PoliformaT del tema.

Multiplicación de dos números `a` y `b` Naturales

La solución recursiva más sencilla a este problema se basa en una conocida igualdad matemática: si `a`$>$ `0` entonces `a*b` $=$ `((a-1) * b) + b`; sino `a * b` $=$ `0`. La transcripción de este resultado a Java es directa:

```
/** a >= 0 AND b >= 0 **/
static int multiplicar(int a, int b){
   if ( a == 0 ) return 0;
   return multiplicar(a - 1, b) + b;
}
```

Como es fácil deducir, la talla del problema que resuelve `multiplicar` es `x = a` y su coste es siempre lineal con dicha talla; aplicando el **Teorema** `1` con `a` $=$ `1` $=$ `c` $=$ `1` se obtiene el mismo resultado, i.e. $T_{multiplicar}(x) \in \Theta(x)$.

Una primera mejora del coste del método propuesto, nótese, consiste en que el parámetro que representa la talla sea siempre el menor de `a` y `b`:

```
static int multiplicarMinimo(int a, int b){
   if ( a < b ) return multiplicar(a, b);
   return multiplicar(b, a);
}
```

Pero esta mejora no supone una modificación de la estrategia recursiva que expresa `multiplicar`; para lograrla, visto que su tipo de Recursión es Lineal, que la talla del problema decrece una unidad en cada llamada y que la sobrecarga es constante, se puede intentar su Reducción Logarítmica. Para ello, basta recordar la correspondiente igualdad matemática para `a`$>$`0`:

$$\text{si } a \text{ es par, } a*b = (\tfrac{a}{2}*b)*2\text{; sino, } a*b = ((\tfrac{a}{2}*b)*2)+b$$

Transcribiéndola a Java, se obtendría el siguiente método:

```
/** 0 <= a <= b **/
static int multiplicarRL(int a, int b){
   if ( a == 0)     return 0;
   int resLlamada = multiplicarRL(a/2, b);
   if ( a%2 == 0 )  return resLlamada * 2;
   return resLlamada * 2 + b;
}
```

Las Ecuaciones de Recurrencia que expresan el coste de `multiplicarRL` son entonces, para una talla dada `x = a` y cualquier instancia, las siguientes:

$$T_{multiplicarRL}(x = 0) = k'; T_{multiplicarRL}(x > 0) = 1 * T_{multiplicarRL}(\frac{x}{2}) + k$$

Aplicando el **Teorema** 3 con a=1 y c=2 a la segunda ecuación, $T_{multiplicarRL}(x>0) \in \Theta(log_2 x)$; como era de esperar, la Reducción Logarítmica efectuada mejora el coste Asintótico, que pasa a ser logarítmico con la talla en lugar de lineal.

Cuestión: discútase si `multiplicarRL` tiene un coste Espacial menor que el de `multiplicar`.

El problema de la Exponenciación

Dados dos Naturales `a` e `b`, el problema de calcular a^b tiene talla `x = b` y puede resolverse aplicando cualquiera de las estrategias recursivas que definen las siguientes igualdades matemáticas:

1. si $b > 0$ entonces $a^b = (a^{b-1} * a)$; sino $a^b = 1$.
2. si $b > 0$ es par, $a^b = a^{\frac{b}{2}} * a^{\frac{b}{2}}$; sino, si $b > 0$ es impar, $a^b = a^{\frac{b}{2}} * a^{\frac{b}{2}} * a$

Es evidente que la transcripción Java de la primera desigualdad origina un método sencillo cuyo coste espacial y temporal es lineal con la talla `x = b`; en cuanto a la segunda, donde la talla del problema se divide siempre por dos, en base al criterio de Reducción Logarítmica su transcripción a método Java debería ser más eficiente, con un coste espacial y temporal logarítmico con `b`. Ahora bien, la siguiente transcripción directa a Java **no** lo es:

```
/** a > 0 AND b >= 0 */
static int potencia(int a, int b){
   int resMetodo = 1;
   if ( b > 0 ){
     int resLlamada1 = potencia(a, b/2);
     int resLlamada2 = potencia(a, b/2);
     resMetodo = resLlamada1 * resLlamada2;
     if ( b % 2 != 0)  resMetodo =  resMetodo * a;
   }
   return resMetodo;
}

// potencia(a, b) == a^b
```

Obsérvese que el tipo de Recursión que exhibe este método es Múltiple, y no Lineal como la del más sencillo. Lo peor es que ello se debe a que en su caso general se realiza dos veces la misma llamada `potencia(a, b/2)`, lo que supone una innecesaria repetición de cálculos y el consiguiente consumo extra de espacio para almacenarlos. Así, aunque la talla decrece geométricamente y la sobrecarga es constante, el coste de `potencia` es lineal con la talla; en efecto, si las Ecuaciones de Recurrencia que expresan la Complejidad de `potencia` para una instancia son

$T_{potencia}(x=0) = k'$; $T_{potencia}(x>0) = 2 * T_{potencia}(\frac{x}{2}) + k$

y se aplica el **Teorema** 3 con a=c=2 a la segunda de ellas se tiene que $T_{potencia}(x) \in \Theta(x)$, que coincide con la del método más sencillo.

Cuestión: suponiendo que se realiza la invocación `potencia(a, 8)` y utilizando tan sólo el Árbol de Llamadas que ésta genera, razónese por qué el método recursivo Múltiple y el Lineal consumen la misma cantidad de tiempo y espacio.

La pregunta lógica que se plantea al obtener la igualdad de costes presentada es por qué o dónde ha fallado la estrategia de Reducción Logarítmica aplicada. De la discusión realizada sobre la estructura recursiva de `potencia` es fácil deducir que para realizar una Reducción Logarítmica, además de que la talla decrezca geométricamente y que la sobrecarga sea constante, es imprescindible que el tipo de Recursión sea también Lineal. En el ejemplo de `potencia` esto se logra, simplemente, realizando una cuidadosa transcripción de la estrategia a Java:

```
static int potenciaRL(int a, int b){
   int resMetodo = 1;
   if ( b > 0 ){
      int resLlamada = potenciaRL(a, b/2);
      resMetodo = resLlamada * resLlamada;
      if ( b % 2 != 0)  resMetodo =  resMetodo * a;
   }
   return resMetodo;
}
```

Esto es, se realiza una única llamada `potenciaRL(a, b/2)` y se multiplica por sí mismo su resultado `resLlamada`. Tan sencilla solución evita la repetición innecesaria de cálculos que se realizaba en `potencia`, lo que conlleva una mejora sustancial del coste. Si se compara entonces `potenciaRL` con el método sencillo, la única diferencia que existe entre ellos es que la talla decrece más rápidamente en `potenciaRL`; por tanto, para el mismo número de llamadas, una, y la misma sobrecarga, constante, `potenciaRL` es más rápido; formalmente, basta con aplicar el **Teorema** 3 con $a = 1$ y $c = 2$ para resolver la Relación de Recurrencia del caso general $T_{potenciaRL}(x>0) = 1 * T_{potenciaRL}(\frac{x}{2}) + k$, de donde $T_{potenciaRL}(x>0) \in \Theta(\log_2 x)$, que también es su coste Espacial.

Ejercicios propuestos:

1. Dado un array `v` de componentes `Integer`, ordenado de forma creciente y sin elementos repetidos, se quiere determinar si existe alguna componente de `v` que represente el mismo valor que el de su posición en `v`; en el caso que no haya ninguna posición para la que se cumpla dicha condición el resultado será `-1`. Se pide:
   - diseñar un método recursivo de coste mínimo que resuelva este problema y analizar su coste Temporal;
   - si el `array v` tiene elementos repetidos, ¿cuál es el coste del método más eficiente que resuelve el mismo problema? ¿Por qué? ¿Y si `v` no está ordenado?

2. Sea `v` un `array` de `Integer` positivos que se ajustan al perfil de una curva cóncava, i.e. existe una única posición `k` de `v`, `0≤k<v.length`, tal que `∀j: 0≤ j<k: v[j]>v[j+1]`

AND ∀j: k<j<v.length: v[j-1]<v[j]. Se pide diseñar el método recursivo que más eficientemente calcule dicha posición k y analizar su coste.

3. Sea m una Matriz cuadrada de n x n componentes Comparable, tal que para cada una de sus filas se cumplen las dos condiciones siguientes: sus componentes están ordenadas de forma estrictamente creciente y todas ellas son menores que las de la fila siguiente. Aprovechando estas propiedades, se pide diseñar un método recursivo de mínimo coste que dada una componente x de m devuelva la posición de la Matriz (i.e. fila y columna) donde se encuentra. Analícese también su coste y discútase si sería el mismo que se obtendría si las componentes de m no cumplieran las propiedades expuestas.

2.2. Recursión Múltiple VS Recursión Lineal: la estrategia Divide y Vencerás (DyV) y su aplicación al problema de la Ordenación

La prevención asociada al uso de la Recursión Múltiple resulta más que natural. Por una parte, intuitivamente, la experiencia adquirida a través de ejemplos como fibonacci o potencia ha permitido comprobar que si el tipo de Recursión que presenta un método es Múltiple, i.e. se producen a > 1 llamadas recursivas en su caso general, entonces se debe exigir que los subproblemas que representan dichas llamadas sean, como mínimo, disjuntos o más exactamente, esencialmente sin superposiciones que obliguen a una costosa y superflua repetición de cálculos y definición de memoria para recordarlos. Por otra parte, ya más formalmente, si se comparan los resultados que da uno cualquiera de los cuatro teoremas de coste enunciados para a=1 y a>1 se advierte que siempre que a>1, Recursión Múltiple, el coste es mayor que cuando a=1, Recursión Lineal. Sin embargo esta afirmación no se mantiene si se combinan adecuadamente los factores talla y sobrecarga, como demuestra el siguiente ejemplo: la comparación de los resultados de los **Teoremas** 1 y 3 para una sobrecarga constante indica que si la Recursión es Múltiple, la talla decrece geométricamente y (c = a)≥ 2 se obtiene una cota igual a la que presenta el mismo problema con una reducción aritmética de la talla y a = 1; obviamente la Recursión Lineal es siempre más sencilla de implementar que la Múltiple, pero lo que se pretende destacar con el ejemplo es que una correcta combinación de los factores talla y sobrecarga puede romper el *a-priori* establecido peligro de la Recursión Múltiple e incluso llegar a darle la vuelta. A saber, para una sobrecarga lineal con la talla, los resultados del **Teorema** 2 con a=1 pueden ser peores que los del **Teorema** 4 con a≥c; por ejemplo como se demostrará a continuación, el conocido problema de la Ordenación genérica de las componentes de un array puede ser resuelto más eficientemente con Recursión Múltiple que con Lineal siempre que la talla decrezca geométricamente y la sobrecarga sea lineal con la talla.

Cuestión: en función de la discusión realizada, razónese cuál de los siguientes métodos resuelve con mayor eficiencia el problema de sumar las componentes de un array:

```
static int sumarV1(Integer v[], int inicio){
   if ( i > v.length ) return 0;
   return sumarV1(v, inicio+1, m) + v[inicio].intValue();
}
static int sumarV2(Integer v[], int inicio, int fin){
   if ( inicio == fin ) return v[inicio].intValue();
   int mitad = (inicio + fin) / 2;
   return sumarV2(v, inicio, mitad) + sumarV2(v, mitad+1, fin);
}
```

A partir de los resultados sobre Recursión Múltiple que se acaban de presentar se puede establecer una estrategia recursiva y eficiente para la resolución de problemas; tal estrategia se denomina **Divide y Vencerás** (DyV) y consta de los siguientes pasos:

1. **DIVIDIR** el problema original de talla `x` en `a` $>$ `1` subproblemas disjuntos y cuya talla divida la del original de la forma más equilibrada posible, i.e. la talla de cada uno de ellos $\frac{x}{c} \leq \frac{x}{a}$. Por ejemplo, si se divide un `array v` en dos a partir de su posición central, se obtienen dos **subarray** de talla la mitad de la de `v`;
2. **VENCER** o resolver los subproblemas definidos de forma recursiva excepto, por supuesto, los que originan el caso base, que se resolverán de forma trivial o directa;
3. **COMBINAR** las soluciones de los subproblemas adecuadamente para obtener la solución del problema original.

Un método que refleja esta estrategia podría ser el siguiente:

```
public static TipoResultado vencer( TipoDatos x ){
    TipoResultado resMetodo, resLlamada1, resLlamada2,..., resLlamadaK;
    if ( casoBase(x) ) resMetodo   = solucionBase(x);
    else{              int c = dividir(x);
                       resLlamada1 = vencer(x/c);
                       ...
                       resLlamadaK = vencer(x/c);
                       resMetodo   = combinar(x, resLlamada1, ..., resLlamadaK);
    }
    return resMetodo;
}
```

donde la Relación de Recurrencia que expresa la Complejidad de su caso general sería:

$$T_{vencer}(x>x_{base}) = a * T_{vencer}(\frac{x}{c}) + T_{dividir}(x) + T_{combinar}(x)$$

Como es fácil deducir, la resolución exacta de esta ecuación depende de los parámetros `a`, `c` y de la sobrecarga $T_{dividir}(x) + T_{combinar}(x)$. Por tanto, suponiendo que en cualquier caso $T_{dividir}(x) + T_{combinar}(x) \in O(x^g)$, i.e. que la sobrecarga es como mucho del orden de un polinomio de grado $g \geq 0$, y que para ser realistas nada bueno se puede obtener si $g \geq 3$, se pueden extraer las siguientes conclusiones sobre el coste de un método DyV:

1. Si la sobrecarga es constante, i.e. para $g=0$, $T_{vencer}(x>x_{base})$ viene dado por el **Teorema 3**: en función del número de llamadas `a` que se realicen y de lo equilibrada que sea la división de la talla original en cada una de ellas, `c`, $T_{vencer}(x>x_{base})$ variará entre $\Theta(\log_c x)$ para $a=1$ y $\Theta(x^{\log_c a})$ para $a>1$.

2. Si la sobrecarga es lineal con la talla del problema `x`, i.e. $g=1$, el **Teorema 4** permite obtener $T_{vencer}(x > x_{base})$: la elección de los valores de `a` y `c` determina un coste que varía entre $\Theta(x)$ para $a<c$ y $\Theta(x^{\log_c a})$ para $a>c$, pasando por $\Theta(x * \log_c x)$ si $a=c$.

   A grosso modo, lo que indican estos dos resultados es que al usar la estrategia DyV con una sobrecarga constante o lineal, un número de llamadas excesivo puede contrarrestar el efecto beneficioso de una distribución equilibrada de la talla del problema.

3. Para sobrecargas mayores que la lineal, mencionar tan sólo que se puede demostrar (véase la sección 7.5.3 del libro de Weiss, págs.195-196) que si $a > c^g$, $T_{vencer}(x) \in O(x^{\log_c a})$; si $a = c^g$, $T_{vencer}(x) \in O(x^g * \log x)$; si $a < c^g$, $T_{vencer}(x) \in O(x^g)$

Ejercicio propuesto: estudiar la resolución del problema de la subSecuencia de Suma Máxima que se plantea en las secciones `7.5.1` y `7.5.2` del libro de Weiss, págs.`188-194`.

Validación de un diseño DyV

Hasta el momento, para validar un método recursivo se ha demostrado su terminación y la coherencia de sus llamadas y se ha obviado la demostración de su corrección o prueba formal de que el método obtiene la solución correcta del problema que resuelve para cualquier valor de su talla; ahora bien, para métodos DyV esta prueba resulta sencilla y clarificadora, por lo que se incluirá como parte de la validación: si se demuestra por Inducción sobre la talla del problema que el método DyV diseñado es correcto se comprueba que la secuenciación de los métodos que implementan los pasos **DIVIDIR VENCER** y **COMBINAR** también lo es; por ejemplo, para demostrar por Inducción sobre la talla del problema que el método `vencer` es correcto se deberá comprobar que:

- **Base de la Inducción:** si `casoBase(x)`, `vencer` es correcto si lo es `solucionBase(x)`;
- **Prueba de Inducción:** si `!casoBase(x)`, suponiendo por **Hipótesis de Inducción** que los resultados de las (al menos dos) llamadas recursivas a `vencer(x/c)` son correctas, `vencer(x)` es correcto, suponiendo que `dividir` y `combinar` también lo son.

Mejora adicional del coste de ejecución de los métodos DyV

En diferentes puntos de este tema se ha subrayado que sólo hay que resolver recursivamente aquellos problemas lo suficientemente complejos como para merecerlo; cuando existen dos soluciones a un mismo problema, por ejemplo el de la Ordenación, una iterativa sencilla pero poco eficiente, como la de Inserción o Selección Directa, y otra recursiva DyV más eficiente pero también más sofisticada, para tallas suficientemente grandes del problema no cabe duda de cual hay que utilizar. Sin embargo, cuando la ejecución el método DyV alcanza tallas suficientemente pequeñas, digamos que menores que una cierta **talla umbral**, el consumo Espacial y Temporal extra que supone la ejecución de un método recursivo invalida la comparación establecida; por ejemplo, se puede determinar experimentalmente que para tallas del problema menores que `10`, aunque cualquier valor entre `5` y `20` produce resultados similares, el método iterativo `insercionDirecta` es más eficiente que el método recursivo DyV `quickSort`. Es por ello que, siempre que sea posible, tras calcular experimentalmente su talla umbral, el coste de ejecución de un método DyV se puede mejorar transformando su caso base como sigue:

```
public static TipoResultado vencer( TipoDatos x ){
   TipoResultado resMetodo, resLlamada1,..., resLlamadaK;
   if ( x < X_UMBRAL )
          resMetodo = solucionIterativa(x);
   else{  int c = dividir(x);
          resLlamada1 = vencer(x/c); ... resLlamadaK = vencer(x/c);
          resMetodo = combinar(x, resLlamada1, ..., resLlamadaK);
   }
   return resMetodo;
}
```

Cuestión: si se escoge como estrategia de Ordenación Divide y Vencerás, razónese por qué es mejor usar `insercionDirecta` que `seleccionDirecta` como `solucionIterativa(x)`.

Soluciones DyV al problema de la Ordenación genérica de un `array`

A modo de ejemplo, se plantea ahora la forma de utilizar la estrategia DyV para resolver eficientemente el problema de la Ordenación genérica de un `array`; ello servirá para describir más adelante dos métodos concretos que instancian dicho planteamiento, `mergeSort` y `quickSort`.

Como ya se sabe, la estrategia general para ordenar un `array v` es un Recorrido en el que la visita a cada componente `v[i]` del `array`, $0 \leq$`i`$<$`v.length`, consiste en ordenarla con respecto a las restantes; por tanto,

1. **antes de visitar** `v[i]` todas las componentes del **sub**`array v[0 ... i-1]` están ordenadas y sólo restan por ordenar las componentes del **sub**`array v[i ... v.length-1]`;
2. **tras visitar** `v[i]` se ha avanzado un paso hacia la terminación de la Ordenación de `v`, o se ha decrementado en uno la talla del problema; en concreto, `v` quedará ordenado cuando concluya la visita a `v[v.length-1]`, pues sólo restarán por ordenar las cero componentes de un **sub**`array` vacío que, obviamente, ya están ordenadas.

A partir de lo dicho, para completar la estrategia de Ordenación sólo resta determinar cómo ordenar la componente `v[i]` del sub`array v[i ... v.length-1]`, $0 \leq$`i`$\leq$`v.length`. Para ello, como se estudió en PRG, se pueden aplicar tres principios: el de Inserción, el de Selección y el de Intercambio; el primero de ellos, recuérdese, fue el que se usó en el Tema 3 para diseñar el siguiente método iterativo de Ordenación Ascendente:

```
public static <T extends Comparable<T>> void insercionDirecta(T v[]){
  for( int i = 1; i < v.length; i++ ){
    T aIns =  v[i]; int posIns = i;
    // Búsqueda del lugar de inserción ordenada; desplazar mientras NO se encuentre
    for (; posIns>0 && aIns.compareTo(v[posIns-1])<0 ; posIns--) v[posIns]=v[posIns-1];
    // Inserción en posIns
    v[posIns] = aIns;
  }
}
```

El código de este método refleja cómo ordenar `v[i]` por Inserción: se determina primero mediante una Búsqueda Descendente la posición `posIns` que debe ocupar `v[i]` en el sub`array` **ya** ordenado `v[0 ... i-1]`, esto es la posición de inserción ordenada de `v[i]` en `v`, y luego se inserta `v[i]` en tal posición; durante la Búsqueda, precisamente para que cuando concluya `v[i]` se pueda insertar directamente en `posIns`, las componentes de `v[0 ... i-1]` mayores o iguales que `v[i]` se desplazan una posición a la derecha. Nótese entonces que

- la primera componente de `v` que se ordena es `v[1]`, pues `v[0]` ya está ordenada con respecto a las cero componentes del **sub**`array` vacío que hay a su izquierda;
- `insercionDirecta` es un método natural de Ordenación, pues cuanto más ordenado está inicialmente `v` menor es su coste y viceversa: en el Peor de los Casos, cuando inicialmente `v` está ordenado **des**cendentemente, `posIns = 0` para cualquier componente de `v` y, por tanto, la Búsqueda de la posición de inserción ordenada de `v[i]` tiene un coste lineal con la talla `i` del sub`array v[0...i-1]`; en el Mejor de los casos, cuando inicialmente `v` ya está ordenado **as**cendentemente, `posIns = i` para cualquier componente de `v` y, por tanto, la Búsqueda de la posición de inserción ordenada de `v[i]` tiene un coste independiente de la talla de `v[0...i-1]`.

Enunciando recursivamente esta estrategia se tendría que, para ordenar ascendentemente las componentes de un subarray `v[i ... v.length-1]`, $0 \leq i \leq$ `v.length`:

- si el subarray está vacío o tiene una componente, i.e. `i == v.length ó i == 0`, no hay que hacer nada, pues obviamente ya está ordenado;
- si el subarray tiene al menos dos componentes, i.e. $2 \leq$ `i` $<$ `v.length`, primero se ordena ascendentemente por Inserción su `i`-ésima componente y luego, aplicando los pasos 1 ó 2, se ordenan sus restantes `v.length-i-1` componentes.

Traduciendo a Java esta estrategia se tendría:

```
/** 1 <= i <= v.length*/
public static <T extends Comparable<T>> void insercionDirectaR(T v[], int i){
  if ( i < v.length ){
    T aIns =  v[i]; int posIns = i ;
    for (; posIns>0 && aIns.compareTo(v[posIns-1])<0; posIns--) v[posIns]=v[posIns-1];
    v[posIns] = aIns;
    /** Ordenada por Inserción v[i], ordenar las restantes v.length-i-1 */
    insercionDirectaR(v, i+1);
  }
}
```

Nótese que este método difiere de su equivalente iterativo `insercionDirecta` en dos cosas: el `if` que evalúa el caso de la Recursión y la llamada recursiva `insercionDirectaR(a, i+1)` substituyen al bucle `for` que se utiliza para recorrer iterativamente `v`; es necesario realizar la invocación `insercionDirectaR(v, 1)` desde un método guía para ordenar el `array v`.

Por lo demás, como ambos métodos usan la misma estrategia para resolver el mismo problema, el coste Temporal de `insercionDirecta` se puede obtener a partir de las Ecuaciones de Recurrencia asociadas a `insercionDirectaR`: considerando que la talla del problema es el número de componentes del **sub**`array v` a ordenar, `x = v.length-i`,

$$T_{\texttt{insercionDirectaR}}(x \leq 1) = k'$$
$$T_{\texttt{insercionDirectaR}}(x > 1) = 1 * T_{\texttt{insercionDirectaR}}(x-1) + T_{\texttt{visitar}}(x)$$

Obviamente, para poder resolver estas ecuaciones hay que especificar antes $T_{\texttt{visitar}}(x)$, el coste de Ordenar por Inserción una componente de `v` con respecto a las `x-1` restantes. Como ya se ha señalado, este coste depende de cuán ordenadas estén inicialmente las componentes de `v`: en el Peor de los Casos, si inicialmente las componentes están ordenadas **des**cendentemente, $T_{\texttt{visitar}}{}^{P}(x) = k * x$; en el Mejor de los Casos, si las componentes ya están ordenadas **as**cendentemente, $T_{\texttt{visitar}}{}^{M}(x) = k*1$. Por tanto,

- $T_{\texttt{insercionDirectaR}}{}^{P}(x>1) = 1 * T_{\texttt{insercionDirectaR}}{}^{P}(x-1) + k * x$

  de donde, aplicando el **Teorema** 2 con $a=c=1$, $T_{\texttt{insercionDirectaR}}{}^{P}(x) \in \Theta(x^2)$.
- $T_{\texttt{insercionDirectaR}}{}^{M}(x>1) = 1*T_{\texttt{insercionDirectaR}}{}^{M}(x-1) + k$.

  de donde, aplicando el **Teorema** 1 con $a=c=1$, $T_{\texttt{insercionDirectaR}}{}^{M}(x) \in \Theta(x)$.

A partir del coste obtenido para sus instancias significativas ya se puede deducir que en general, para cualquier instancia, la Ordenación por Inserción Directa se realizará en un tiempo como máximo cuadrático con la talla, $O(x^2)$, y como mínimo lineal con ella, $\Omega(x)$.

Cuestión: usando el criterio de Selección para ordenar la componente `v[i]`, diséñese el método `seleccionDirectaR` y, tras analizar su coste, compárese con `insercionDirectaR`.

Si se desea utilizar una estrategia Divide y Vencerás para mejorar el coste de la estrategia de Ordenación que subyace en `insercionDirectaR`, i.e. Recursión Lineal con reducción aritmética de la talla y sobrecarga lineal en el Peor de los Casos, se deben tener en cuenta lo siguiente:

- No se pueden utilizar los resultados del **Teorema** `3` sino los del `4` pues, en la mayoría de las ocasiones, ordenar una componente de un `array` con respecto a las restantes requiere un tiempo lineal con la talla y no constante; la cota lineal del Peor de los Casos del criterio de Inserción no se mejora ni utilizando el criterio de Selección, que es lineal para cualquier instancia, ni el de Intercambio, que tiene el mismo comportamiento que el de Inserción pero provoca una Ordenación no natural.
- La posibilidad de coste lineal que proporciona el **Teorema** `4` con `a<c` queda excluida al considerar que requiere una división geométrica y equilibrada de la talla del problema, i.e. como mínimo `c=2`, y que la estrategia recursiva que se quiere mejorar es Lineal, `a=1`. En conjunto, las dos condiciones expresadas obligan a usar Recursión Múltiple con $a \geq c$; es más, la posibilidad de que $a > c$ también queda descartada, por ejemplo tres o más llamadas recursivas en las que la talla es la mitad de la original, pues su coste está acotado por $x^{\log_c a}$, igual o superior al coste de la estrategia que se quiere mejorar.

Por tanto, no queda más remedio que caracterizar cualquier posible estrategia DyV que se proponga para la Ordenación como aquella en la que, **en promedio**, hay que **VENCER** dos subproblemas de talla aproximadamente la mitad que la original, $a = c = 2$, y en el que se emplea como máximo un tiempo lineal para **DIVIDIR** la talla original de forma equilibrada y **COMBINAR** los subproblemas resueltos. Con estas características, el **Teorema** `4` garantiza un coste $\Theta(x * \log_c x)$, que para valores suficientemente grandes de la talla mejora bastante la cota cuadrática en promedio de cualquier estrategia de Ordenación recursiva Lineal.

2.2.1. Ordenación por Fusión o Merge: métodos `mergeSort` y `fusionDyV`

La **Ordenación por Fusión** o ***Merge*** que realiza el método denominado `mergeSort` es el ejemplo más claro y directo de aplicación de la estrategia Divide y Vencerás al problema de la Ordenación. En efecto, para ordenar los $x = der-izq+1 > 1$ Datos de un **subarray** `v[izq...der]`, $0 \leq izq \leq der < v.length$, la estrategia de Fusión realiza los siguientes pasos:

1. **DIVIDIR** por `2`, $c = 2$, el número de componentes a ordenar por Fusión; para ello basta calcular la posición central de `v[izq...der]`, `mitad = (der+izq)/2`, lo que permite descomponerlo en dos **subarray** `v[izq...mitad]` y `v[mitad+1...der]`, cada uno de los cuales tiene una talla aproximadamente igual a la mitad de la original;
2. **VENCER** u ordenar por Fusión los dos **subarray** en los que se ha descompuesto el original, también $a = 2$, siempre que sus tallas sean estrictamente mayores que uno; si alguno de ellos contiene tan sólo una componente evidentemente ya está ordenada;
3. **COMBINAR** en tiempo lineal los resultados de los dos subproblemas resueltos en el paso anterior para obtener la Ordenación de las componentes de `v[izq ... der]`; como las componentes de `v[izq...mitad]` y `v[mitad+1...der]` ya han sido ordenadas, se puede realizar su **Fusión** o **Mezcla Natural** en tiempo lineal.

La transcripción directa a Java de la estrategia presentada sería la siguiente:

```
private static <T extends Comparable<T>> void mergeSort(T v[], int izq, int der){
   if ( izq < der ){
       int mitad = (izq+der)/2;
       mergeSort(v, izq, mitad); mergeSort(v, mitad+1, der);
       fusionDyV(v, izq, mitad+1, der);
   }
}
```

donde `fusionDyV` es una modificación del método `fusion` que se presentó en el apartado anterior de este tema que permite **COMBINAR** en un tiempo lineal los resultados de los dos subproblemas en que se ha dividido la Ordenación en `mergeSort`; específicamente, los cambios a realizar en `fusion` son los siguientes:

- En lugar de dos `arrays a` y `b` de tipo `int` y tallas `a.length` y `b.length` respectivamente, se deben fusionar los dos **subarray** de tipo genérico `T extends Comparable` ya ordenados en los que se ha divido el original `v[izq...der]`, `v[izq...mitad]` y `v[mitad+1...der]`.
- Al variar los Datos del problema cambian el número de parámetros formales que los representan en la cabecera del método; así `v` es el único `array` que aparece junto con los índices donde empiezan y acaban sus dos **subarray** a fusionar, los que determinan sus tallas y permiten inicializar adecuadamente las variables locales a la fusión.

  Esta variación de Datos repercute también en el tipo del resultado del método; en lugar de un nuevo `array` el resultado es el propio `v[izq...der]` ya ordenado, por lo que la fusión se realiza en un `array` auxiliar que luego se debe copiar en `v[izq...der]`.

El siguiente método `fusionDyV` recoge todos estos cambios:

```
/** izqA es la 1er posición del array ordenado en la 1-er llamada;
 *  juega el papel de a[0] en fusion, de ahí su nombre
 *  izqB es la 1er posición del array ordenado en la segunda llamada;
 *  juega el papel de b[0] en fusion, de ahí su nombre. Además,
 *  izqB == derA + 1, por lo que derA no aparece como parámetro formal
 *  derB es la última posición del array ordenado en la segunda llamada;
 *  juega el papel de b[b.length] en fusion, de ahí su nombre
**/
private static <T extends Comparable<T>> void fusionDyV(T v[],int izqA,int izqB,int derB){
   T[] res = Arrays.copyOf(v, derB-izqA+1);
   int i = izqA, derA = izqB - 1, j = izqB, k = 0;
   while ( i <= derA && j <= derB ){
     if ( v[i].compareTo(v[j])<0 ) res[k] = v[i++]; else res[k] = v[j++];
     k++;
   }
   for ( int l = i; l <= derA; l++ ) res[k++] = v[l];
   for ( int l = j; l <= derB; l++ ) res[k++] = v[l];
   // res se copia en v[izq...der]
   for ( int l = izqA; l <= derB; l++ ) v[l] = res[l-izqA];
 }
```

Recordar finalmente que hay que realizar la invocación `mergeSort(v,0,v.length-1)` desde un método guía homónimo para ordenar un `array v`.

Ejercicio propuesto: sea el `array` de `Integer (3,41,52,26,38,5,7,9,49)`; realícese una traza del método `mergeSort` sobre él.

Validación de `mergeSort`

Como para cualquier otro método recursivo, la validación de `mergeSort` se compone de las siguientes pruebas:

1. **Terminación**: en su caso general la talla del problema `x = der-izq+1` decrece, pues en cada una de las dos llamadas que se producen se divide aproximadamente por 2: la talla de `mergeSort(v,izq,mitad)` es `mitad-izq+1` $= \frac{x}{2}$ y la de `mergeSort(v,mitad+1,der)` es `der-mitad` $= \frac{x-1}{2}$, cualquiera de ellas menor estricta que `x` puesto que `x`$\geq$2 si `izq`$<$`der`; por tanto, en base a las propiedades de la división entera, ambas pueden alcanzar en un tiempo finito el valor `1` del caso base, donde termina la Recursión.
2. **Coherencia de las llamadas recursivas**: en cualquiera de los casos establecidos, los valores de los parámetros formales `izq` y `der` siempre cumplen la Precondición ya que en cualquier llamada, al ser $0 \leq$ `izq` $\leq$ `mitad` $<$ `der` $<$ `v.length`, varían dentro del dominio de definición, i.e. $0 \leq$ `i` $\leq$ `j` $<$ `v.length`; nótese que en la llamada más alta `izq=0` y `der=v.length-1` y en cualquier otra bien se mantienen como en la precedente (el de `izq` en la primera y el de `der` en la segunda) o bien, cuando varían, `izq` lo hace en el intervalo `[mitad+1 ... v.length-1]` y `der` en `[0...mitad]`.
3. **Corrección**: probar por Inducción que `mergeSort` ordena `v[izq...der]` supone a su vez
   - **Base de la Inducción:** probar que si `x=1` entonces `mergeSort(v, izq, der)` ordena la única componente de `v[izq...der]`. Nótese que esta prueba es trivial porque un `array` de talla uno ya está ordenado y `mergeSort` no hace nada si `izq == der`.
   - **Prueba de Inducción:** probar que si `x`$>$`1` entonces `mergeSort(v, izq, der)` ordena el subarray `v[izq...der]`, suponiendo por **Hipótesis de Inducción** (HI) que la llamada a `mergeSort(v, izq, mitad)` ordena `v[izq...mitad]` y que la llamada a `mergeSort(v, mitad+1, der)` hace otro tanto con `v[mitad+1...der]`.

     También ahora la prueba es fácil: si por HI `v[izq...mitad]` y `v[mitad+1...der]` están ordenados entonces `fusionDyV` los mezcla en un único `array v[izq...der]` también ordenado y, por tanto, `mergeSort` en su caso general es correcto; nótese que es la corrección de `fusionDyV` la que garantiza la de `mergeSort`.

Coste Asintótico de `mergeSort`

Para comprobar que el coste Temporal de `mergeSort` es Θ`(x` $*$ `log`$_2$`x)` se deben obtener primero a partir de su código las siguientes Relaciones de Recurrencia:

$$T_{\texttt{mergeSort}}(\texttt{x=1}) = \texttt{k'}$$
$$T_{\texttt{mergeSort}}(\texttt{x>1}) = 2 * T_{\texttt{mergeSort}}(\tfrac{\texttt{x}}{2}) + T_{\texttt{fusionDyV}}(\texttt{x})$$

Recordando entonces que $T_{\texttt{fusionDyV}}(\texttt{x}) \in \Theta(\texttt{x})$ y aplicando el **Teorema** 4 con `a=c=2` se tiene que para cualquier instancia $T_{\texttt{mergeSort}}(\texttt{x}) \in \Theta(\texttt{x} * \log_2\texttt{x})$.

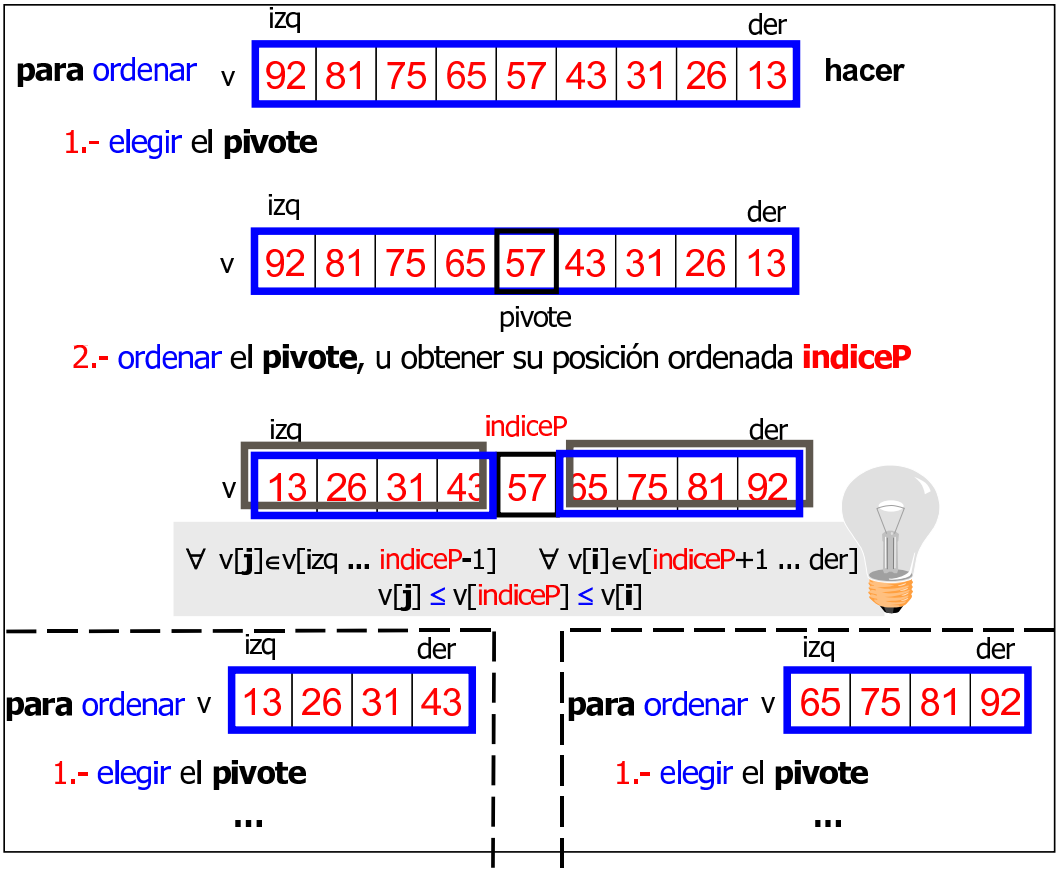
A pesar de este excelente coste Temporal, que puede ser mejorado significativamente como se propone en los ejercicios `1` y `2` que figuran a continuación, `mergeSort` presenta un serio inconveniente frente a otros métodos de Ordenación rápida *in-situ* que se estudiarán más adelante: requiere inevitablemente el uso de un `array` auxiliar para llevar a cabo la fusión de los subarray ya ordenados `v[izq...mitad]` y `v[mitad+1...der]` y, por ello, su coste Espacial de ejecución, que no el Asintótico, dobla al de éstos.

Ejercicios propuestos:

1. Realícense las modificaciones necesarias en el método `mergeSort` propuesto para que en cada llamada a `fusionDyV` se evite la copia del `array` auxiliar en el original.
2. Determínese experimentalmente el valor de la talla umbral para `mergeSort`, esto es el tamaño del `array v` a partir del cuál es más efectivo espacial y temporalmente ordenar por un método directo como `insercionDirecta`.
3. Una estrategia alternativa a la presentada para diseñar la Ordenación por Fusión sería la de dividir el `array` en tres partes, ordenarlas y después fusionarlas. Analícese su coste Temporal e indíquese si mejora el del método `mergeSort` propuesto.

2.2.2. Quick Sort: métodos `quickSort` y `particion`

Para evitar la fusión de sub`array`s ya ordenados que caracteriza a `mergeSort` se puede pensar en `DIVIDIR` el sub`array` a ordenar utilizando un procedimiento un poco más sofisticado: ordenar una dada de sus componentes, el denominado **pivote**, situando a su izquierda aquellas que sean menores (o iguales) que ella y situando a su derecha aquellas que sean mayores (o iguales); así, como ilustra la siguiente figura, la ordenación del pivote provoca una **Partición** del resto de las componentes del sub`array` en dos grupos disjuntos, que acto seguido pueden ser ordenados independientemente usando el mismo procedimiento.



La estrategia DyV esbozada recibe el nombre de ***Quick Sort*** y, como se puede deducir con ayuda del ejemplo de la figura anterior, sus dos características más relevantes son las siguientes:

1. para que la Partición del subarray a ordenar se produzca en un tiempo lineal, y con ello se cumpla la restricción de sobrecarga que exige la estrategia DyV, **el pivote se debe ordenar por Intercambio**: para ordenar la componente elegida como pivote en el ejemplo, `v[4] = 57`, como 92 es mayor que 57 y 13 menor, `intercambiar(92, 13)`; como 81 es mayor que 57 y 26 menor, `intercambiar(81,26)`; `...`; tras `intercambiar(65,43)` concluye la Partición porque el pivote es igual a sí mismo. Nótese entonces que, obviamente, el número de comparaciones realizadas es exactamente igual a la talla del subarray `v`.
2. **la Partición del subarray `v` es más o menos equilibrada en función** del pivote que se elija, o más precisamente **del valor que tenga el pivote con respecto a los que poseen las restantes componentes del subarray**: única y exclusivamente porque el pivote elegido en el ejemplo, `v[4] = 57`, es la mediana de las componentes a ordenar la Partición que se produce es la más equilibrada posible, o da lugar a dos subarray de talla aproximadamente igual a la mitad de la original; sin embargo, si se hubiera elegido como pivote la componente mínima o la máxima del subarray, `v[8] = 13` o `v[0] = 92`, la Partición habría sido completamente desequilibrada, un **subarray** vacío y otro con una componente menos que el original.

En resumen, y teniendo en cuenta que al **DIVIDIR** es posible generar **subarray** vacíos, una posible transcripción Java de la estrategia de Ordenación DyV *Quick Sort* sería la siguiente:

```
private static <T extends Comparable<T>> void quickSort(T v[], int izq, int der){
  if (izq < der){
    // DIVIDIR el problema en dos subproblemas disjuntos: ordenar por Intercambio
    // el pivote de forma que, EN PROMEDIO, se obtenga una Partición equilibrada
    int indiceP = particion(v, izq, der);

    // VENCER cada subproblema planteado: EN PROMEDIO,
    // ordenar por Quick Sort x/2 componentes menores(o iguales) que el pivote
    // ordenar por Quick Sort x/2 componentes mayores (o iguales) que el pivote
    quickSort(v, izq, indiceP-1); quickSort(v, indiceP+1, der);

    // ******** NO ES NECESARIO COMBINAR ********
    // v[izq ... der] YA está ordenado: se ha ordenado su pivote v[indiceP]
    // y se han ordenado v[izq ... indiceP-1] y v[indiceP+1 ... der]
  }
}
public static <T extends Comparable<T>> void quickSort(T v[]){
  quickSort(v, 0, v.length-1);
}
```

Cabe reseñar que en este código no se especifican detalles de diseño que pueden afectar significativamente a su coste, como la elección del pivote o el tratamiento de las componentes que son iguales al pivote o la forma precisa de realizar la Partición en tiempo lineal; sólo cuando se realice el análisis del coste de `quickSort` se estará en disposición de decidir cuál es la mejor manera de implementarlos, por lo que se ha decidido relegarlos al diseño del método `particion` que se realizará más adelante.

Ejercicio propuesto: recordando que la talla del problema puede llegar a valer cero y suponiendo que el método `particion` funciona correctamente, realizar la validación de `quickSort`.

Coste Asintótico de `quickSort`

Para tallas mayorores que uno la Complejidad del método `quickSort` depende únicamente del resultado del método `particion`, el `int indiceP` que representa el orden que el pivote elegido ocupa entre las componentes de `v` y, por tanto, determina la talla de las dos llamadas que se realizan tras obtenerlo, `quickSort(v,izq,indiceP-1)` y `quickSort(v,indiceP+1,der)`, o la talla de los respectivos sub`array vI` y `vD` que se ordenan con ellas.

Así, en el Mejor de los Casos `indiceP` $= \frac{(izq+der)}{2}$ en cada llamada que se realiza a `quickSort`, i.e. el pivote y la mediana de `v` coinciden y se logra una Partición completamente equilibrada en dos **subarray** de talla aproximada $\frac{x}{2}$. La Relación de Recurrencia para este caso,

$$T_{quickSort}^{M}(x>1) = 2 * T_{quickSort}^{M}(\tfrac{x}{2}) + k * x$$

se puede resolver utilizando el **Teorema** 4 con a=c=2, de donde $T_{quickSort}^{M}(x) \in \Theta(x * \log_2 x)$; por tanto, para cualquier instancia del problema $T_{quickSort}(x) \in \Omega(x * \log_2 x)$.

En el Peor de los Casos `indiceP = izq` o `indiceP = der` en cada llamada, i.e. el pivote elegido es el mínimo o el máximo de `v` y se origina una Partición completamente desequilibrada en dos **subarray**, uno vacío y otro que contiene todas las componentes de `v` menos el pivote: si `indiceP = izq`, `vI` es un `array` vacío y `vD = v[izq+1...der]` y si `indiceP = der`, `vD` es un `array` vacío y `vI = v[izq...der-1]`. A raíz de esta Partición, la llamada recursiva asociada al **subarray** vacío ya no genera más llamadas y es sólo la otra la que se considera para expresar la Complejidad de `quickSort` en el Peor de los Casos:

$$T_{quickSort}^{P}(x>1) = 1 * T_{quickSort}^{P}(x-1) + k * x$$

Aplicándole el **Teorema** 2 con a = c = 1, $T_{quickSort}^{P}(x) \in \Theta(x^2)$ y, para cualquier otra instancia, $T_{quickSort}(x) \in O(x^2)$.

Los resultados del análisis realizado permiten concluir que, cuando se diseñe `particion`, una buena implementación de la selección del pivote y del paso de Partición debiera minimizar la posibilidad de obtener una Partición desequilibrada, esto es con tallas de `vI` y `vD` desiguales, aún para instancias degeneradas como aquellas en las que `v` está inicialmente ordenado Ascendente o descendentemente o, peor aún, aquellas en las que todas sus componentes son iguales. Asimismo, vista la enorme diferencia que los separa, obligan a plantear sin paliativos el estudio de su coste Medio; recuérdese que ya al presentar la estrategia DyV para la Ordenación Rápida se ha indicado que se espera, **en promedio**, **DIVIDIR** la talla original por la mitad al ordenar cada componente del `array` en tiempo lineal, lo que supone que la cota $O(x * \log_2 x)$ indicada es la de su caso Medio. Aunque esta expectativa es suficiente para los propósitos de la asignatura, no constituye sin embargo una demostración formal, como la que se puede encontrar en las páginas de la 228 a la 230 del Capítulo 8 del libro de Weiss; en ella, se logra calcular el coste Medio de cada llamada recursiva a `quickSort` y, a partir de él, el coste medio de `quickSort` que es, efectivamente, $O(x * \log_2 x)$.

Diseño del método `particion`

A continuación se propone un esquema algorítmico del método `particion` que, en tiempo lineal, debe seleccionar una componente de `v[izq...der]` como pivote y ordenarla con respecto a las demás; no se considera en él la posibilidad de que existan componentes iguales al pivote:

```
T pivote = elegirPivote(v, izq, der);
/** ordenar el pivote: para toda componente de v[izq...der]
 * intercambiar aquella que situada a la izquierda del pivote,
 * desde izq, sea mayor que éste CON aquella que situada a la
 * derecha del pivote, desde der, sea menor que éste
**/
int i = izq; int j = der;
do{
   /** Búsqueda Ascendente de la primera componente de v[i...der] mayor que el pivote */
   while ( v[i].compareTo(pivote) < 0 ) {i++}
   /** Búsqueda Descendente de la primera componente de v[izq...j] menor que el pivote */
   while ( v[j].compareTo(pivote) > 0 ) {j--}
   /**  intercambiar v[i] con v[j] y seguir */
   if ( i <= j ) { intercambiar( v, i, j); i++;  j--; }
} while (  i <= j )
/** i > j AND (pivote = v[j] AND vI == v[izq...j-1] AND vD = v[i...der])
 *        OR  (pivote = v[i] AND vI == v[izq...j]   AND vD = v[i+1...der])
 *
 * Para ordenar el resto, ejecutar quickSort(v, izq, j); quickSort(v, i, der);
**/
```

En lo que sigue, en primer lugar se completará su diseño desarrollando aquellas implementaciones de la elección del pivote y el tratamiento de las componentes iguales a él que garanticen que al ordenar el pivote se consigue una división equilibrada de `v`; una vez se disponga de código ejecutable, se introducirán en él nuevas modificaciones para ahorrar intercambios y comparaciones innecesarias, lo que mejorará su coste real de ejecución, que no el Asintótico, a costa de su legibilidad.

Implementación eficaz de la elección del pivote

El estudio del Peor de los Casos indica que para una elección del pivote eficaz se requiere un método que, independientemente de la instancia, minimice la posibilidad de obtener una Partición desequilibrada. Así, queda excluido elegir el pivote como la primera (última) componente de `v`, pues si éste inicialmente ya está ordenado ascendentemente sería su mínimo (máximo) -viceversa para el orden inverso- o elegirlo al azar, pues para instancias aleatorias puede resultar ser el máximo o el mínimo de `v`.

Una elección más razonable es la de la componente central de `v`, $\texttt{v[}\frac{\texttt{izq+der}}{2}\texttt{]}$, perfecta para instancias ordenadas puesto que coincide con su mediana y que para instancias aleatorias tiene una probabilidad muy baja de forzar un coste cuadrático. Sin embargo, se le puede achacar como defecto que sea una estrategia tan pasiva como las anteriores, que no hace nada para seleccionar un buen pivote pero evita elegir uno malo.

En el método que se propone ahora, la **Mediana de Tres**, se intenta seleccionar un pivote mejor que el central partiendo del hecho de que, como indica el estudio del Mejor de los Casos, el mejor pivote de cualquier `array` es su componente mediana; como por desgracia el cálculo de la mediana de `v[izq...der]` aumenta significativamente la sobrecarga real de `quickSort`, para obtener una buena estimación de ésta en tiempo constante se calcula la mediana de sólo tres de sus componentes: la primera, la central y la última.

```
private static <T extends Comparable<T>> T medianaDeTres(T v[], int primera, int ultima){
     int central = (primera+ultima)/2;
     if ( v[central].compareTo(v[primera]) < 0 ) intercambiar(v, primera, central);
     if ( v[ultima].compareTo(v[primera])  < 0 ) intercambiar(v, primera, ultima);
     if ( v[ultima].compareTo(v[central])  < 0 )  intercambiar(v, central, ultima);
     /** v[central] es la Mediana de Tres, mayor (o igual) que v[primera] y
       * menor (o igual) que v[ultima] **/
     return v[central];
}
```

Utilizar `medianaDeTres` como implementación de `elegirPivote` supone realizar en el esquema de `particion`, además de un obvio cambio de nombre, una serie de pequeños ajustes: gracias a que la ejecución de `medianaDeTres(v, izq, der)` no sólo proporciona el pivote de la Partición en posición $\frac{izq+der}{2}$ sino que también ordena `v[izq]` y `v[der]` con respecto a éste, las Búsquedas de componentes a intercambiar pueden empezar en `i = izq+1` y `j = der-1`, en lugar de `izq` y `der` respectivamente, y tener como centinelas a las propias `v[izq]` y `v[der]`, en la Descendente sobre `j` y en la Ascendente sobre `i` respectivamente.

Tratamiento eficaz de las componentes iguales al pivote

Hasta el momento se ha eliminado del esquema de `particion` la posibilidad de que en el `array` existan componentes iguales al pivote; con ello se excluía también la posibilidad de ordenar con `quickSort` instancias con todas las componentes iguales, que aunque a primera vista pueda parecer hasta absurda -¿cuándo sino se puede encontrar un caso real en el que se quieran ordenar miles de componentes iguales?- no lo es tanto cuando se recuerda que `quickSort` es un método recursivo. Por ejemplo, si se quiere ordenar un `array` de `1.000.000` componentes, de las cuales `3.000` son iguales entre sí, no es de extrañar que un momento dado se produzca una llamada recursiva con sólo esas `3.000`; y para tal ocasión, resulta imprescindible asegurar que `quickSort` sigue siendo un método eficiente.

La discusión anterior muestra la necesidad de contemplar detenidamente cuál de las dos siguientes estrategias es la mejor cuando en una Búsqueda del bucle de Partición aparece una componente igual al pivote: parar la Búsqueda como cuando se encuentra una componente mal situada con respecto al pivote y, por tanto, intercambiarla con otra más tarde o bien continuar. Para responder adecuadamente, considérese de nuevo el caso en el que todas las componentes del `array` sean iguales. Parar la Búsqueda provocaría un intercambio por cada par de componentes ¡iguales! comparadas con el pivote, esto es $\frac{x}{2}$ intercambios por cada `x` comparaciones; aunque pudiera parecer un desperdicio de tiempo, sin embargo, no lo es: gracias a él, los índices `i` y `j` se cruzan justo en el centro del `array`, la posición del pivote, por lo que la Partición resultante es siempre equilibrada. Por el contrario, si la decisión que se toma al encontrar una componente igual al pivote es continuar la Búsqueda, aunque no se produce intercambio alguno, a cambio, `i` terminaría valiendo `der`, si se hubiera controlado el posible *overflow* y, por tanto, se obtendría una Partición completamente desequilibrada.

El análisis realizado es concluyente: como es mejor hacer intercambios innecesarios que particiones desiguales, hay que detener la Búsqueda del bucle de Partición en la que aparezca una componente igual al pivote. Nótese que el esquema de `particion` presentado ya contempla esta situación, por lo que no sufrirá modificación alguna para que trate eficazmente aquellos `array` que tengan componentes repetidas.

Algunas mejoras más del coste efectivo de `particion`

Como se indicó al principio de esta sección, una vez se dispone de código ejecutable para `particion` se le pueden introducir algunas modificaciones para que, a costa de la legibilidad, mejore su coste real de ejecución. Específicamente, el código que figura a continuación es el resultado de realizar las siguientes:

1. Eliminar los intercambios innecesarios que se producen para desplazar el pivote desde su posición inicial (la central del `array`) hasta su posición ordenada; para ello, basta con quitarlo de en medio durante el proceso de su ordenación, dejándolo por ejemplo en la posición `der-1`, y situarlo en su posición ordenada una vez ésta haya sido establecida.

   Cuestión: ¿cuál es la posición ordenada del pivote en la primer Partición del `array` de `Integer [10, 4, 8,2, 3, 1, 9]`? Calcúlese cuántos intercambios cuesta situar al pivote en ella cuando su posición inicial es la central del `array` y cuando es `der-1`.

2. Eliminar el `if ( i<=j )` que precede a cada intercambio del anterior bucle de `particion` para que éste se realice siempre dentro de los límites del `array`; para ello, basta con ejecutar una vez menos el bucle de Partición, sólo mientras `i<j`. Aunque así se puede producir a veces un último intercambio incorrecto basta con deshacerlo al acabar el bucle.
3. Compactar al máximo los dos bucles de Búsqueda modificando, sólo, su inicialización.

```
T pivote = medianaDeTres(v, izq, der);
/** ocultar el pivote en la posición der-1, pues para ordenarlo basta
 * buscar su posición ordenada y, tras encontrarla, situarlo en ella */
intercambiar(v, (izq+der)/2, der-1);
/** ordenar el pivote: buscar la posición ordenada del pivote */
int i = izq; int j = der - 1;
for (  ; i < j ; ){
   /** Búsqueda Ascendente de la primera componente
     * de v[i...der] mayor o igual que el pivote */
   while ( v[++i].compareTo(pivote) < 0 ){}
   /** Búsqueda Descendente de la primera componente
     * de v[izq...j] menor o igual que el pivote */
   while ( v[--j].compareTo(pivote) > 0 ){}
   /** intercambiar v[i] con v[j] **/
   intercambiar(v, i, j);
}
/** deshacer el último intercambio por si fuera incorrecto */
intercambiar(v, i, j);
/** restaurar el pivote: situarlo en su posición ordenada */
intercambiar(v, i, der-1);
/** izq <= i < der AND pivote = v[i] AND vI == v[izq...i-1] AND vD == v[i+1...der]
 * Para ordenar el resto, ejecutar quickSort(v, izq, i-1); quickSort(v, i+1, der); */
```

`quickSort` VS `mergeSort`

Para finalizar esta sección se debe señalar que `quickSort` (casi) siempre es más rápido que `mergeSort` gracias a lo compacto y sofisticado que resulta el código de `particion`, tan laboriosamente obtenido: aunque en promedio ambos métodos resuelven el problema de ordenar un `array` de igual manera, dos llamadas recursivas de talla la mitad que la original y una sobrecarga lineal, el proceso de Partición implementado es más eficiente que el de Fusión, aunque no siempre conduzca a divisiones de la talla equilibradas, y no requiere de un `array` auxiliar.

Ejercicios propuestos:

1. Realícese la traza de la ejecución de `quickSort(v, 0, 10)` sobre el `array` de `Integer` {`86, 66, 24, 78, 100, 106, 46, 53, 89, 69, 5`}, indicando para cada llamada recursiva el valor de los parámetros `izq` y `der`, el estado del `array` tras la llamada a `medianaDeTres` y el valor del pivote, los intercambios que se realizan en el bucle de Partición y el estado del `array` tras los intercambios y la restauración del pivote.
2. Ordénese el `array` {`8, 1, 4, 1, 5, 9, 2, 6, 5`} usando los métodos `insercionDirecta`, `mergeSort`, `quickSort` con pivote igual a la componente central del `array` y `quickSort` con pivote igual a la Mediana de Tres y talla umbral de tres.
3. Si todas las componentes del `array` son iguales ¿qué tiempos de ejecución tienen `mergeSort`, `quickSort` e `insercionDirecta`? ¿y si las componentes ya están ordenadas inicialmente?

2.3. Una solución eficiente al problema de la Selección: método `seleccionRapida`

Estrechamente relacionado con el de Ordenación, el denominado **problema de la Selección** consiste en encontrar el `k`-ésimo menor Dato de una Colección de talla dada, del que un caso particular importante es el cálculo de su mediana, o sea del $\frac{x}{2}$ -ésimo menor. Si la Colección se representa sobre un `array`, una solución al problema de coste Θ`(x * log x)` sería ordenarlo, por ejemplo con `quickSort`, y después acceder a su posición `k-1` (recuérdese que la primera posición de un `array` es la cero); alternativamente, se puede obtener el mismo resultado pero con coste Θ`(x * k)` tras ordenar con `seleccionDirecta` las `k-1` primeras componentes del `array`. Ahora bien, como la Selección es un problema más sencillo que la Ordenación, un pequeño cambio en `quickSort` es capaz de rebajar los costes anteriores hasta una cota promedio lineal con la talla. En efecto, la denominada estrategia de **Selección Rápida**, se basa en la idea de que ordenar el pivote de `v[izq...der]`, 0$\leq$`izq`$\leq$`der`$<$ `v.length`, i.e. situarlo en su posición ordenada `indiceP`, provoca una Partición en la que todas las componentes menores o iguales que el pivote se sitúan a su izquierda, en `vI == v[izq...indiceP-1]`, y todas las mayores o iguales a su derecha, en `vD == v[indiceP+1...der]`. Así, si después de la Partición `k-1 = indiceP`, la `k`-ésima menor componente ya ocupa la posición `k-1` de `v`; sino, cuando `k-1` $<$ `indiceP` la `k`-ésima menor componente `v` se debe buscar en `vI` y sino en `vD`. Naturalmente el proceso expuesto es válido para `array` de talla mayor estricta que uno; cuando sólo hay una componente ésta es la única que puede ser seleccionada, con lo que concluye el proceso.

El siguiente método Java implementa la estrategia de Selección Rápida expuesta para tallas mayores que una cierta talla umbral, que expresa indirectamente la constante `LIMITE`:

```
private static <T extends Comparable<T>> void seleccionRapida(T v[],int k,int izq,int der){
   if ( izq + LIMITE > der)  insercionDirecta(v, izq, der);
   else{
           int indiceP = particion(v, izq, der);
           if ( k-1 < indiceP )        seleccionRapida(v, k, izq, indiceP-1);
           else if ( k-1 > indiceP )   seleccionRapida(v, k, indiceP+1, der);
   }
}
public static <T extends Comparable<T>> void seleccionRapida(T v[], int k){
    seleccionRapida(v, k, 0, v.length-1);
}
```

Ejercicios propuestos:

1. Analícese de manera pormenorizada el coste Asintótico del método `seleccionRapida` para las instancias significativas del problema.
2. Realícese la traza de la ejecución de `seleccionRapida(v,4)` sobre el `array` de `Integer [86, 66, 24, 78, 100, 106, 46, 53, 89, 69, 5)]`, indicando para cada una de las llamadas recursivas los valores de los parámetros `izq` y `der`, el estado de `v` tras la llamada a `medianaDeTres` y el valor del pivote, los intercambios que se realizan en el bucle de Partición y el estado de `v` tras los intercambios y la restauración del pivote.