Tecnología de la Programación
Recursividad

2020

Juan Antonio Sánchez Laguna
Grado en Ingeniería Informática
Facultad de Informática
Universidad de Murcia

TABLA DE CONTENIDOS

Recursividad

Decimos que algo es recursivo cuando forma parte de su propia definición. Encontramos ejemplos de recursividad en la definición de acrónimos como el de GNU que significa GNU is Not Unix. En ilustraciones en cuyo interior aparece una réplica de sí misma, en los fractales y en la definición de funciones matemáticas y relaciones de recurrencia como Factorial o la conocida serie de Fibonacci.

La recursividad también aparece en las ciencias de la computación. Muchos lenguajes de programación permiten definir funciones y estructuras de datos recursivas. De hecho, algunos lenguajes, como Lisp o Haskell, no tiene sentencias de control iterativas, como `while` y `for`, y dependen exclusivamente de la recursividad para realizar cualquier secuencia repetitiva de acciones.

Funciones recursivas

Las funciones recursivas son aquellas que se llaman a sí mismas. La llamada puede ser directa, si aparece en su propio código fuente, o indirecta si se llama a otra función y ésta otra llama a la primera.

```
void saluda() {
  printf( "Hola\n" );
  saluda();
}
```

Fig. 1. Ejemplo de función recursiva

La función `saluda` es recursiva, pero, si llegara a ejecutarse, el programa entraría en un bucle infinito en el que nunca pararía de mostrar la palabra "Hola" y, en condiciones normales, esto no es un comportamiento deseable. Para que una función recursiva no produzca bucles infinitos es necesario evitar que la llamada recursiva, es decir, la llamada a la propia función, se ejecute siempre.

El problema se soluciona añadiendo parámetros a la función y usándolos para decidir cuándo hacer la llamada recursiva y cuándo no. Pero la decisión no puede responder a un criterio arbitrario. Así pues, para diseñar correctamente una función recursiva se deben tener en cuenta los siguientes criterios:

- Alguno de los parámetros de entrada se usa para determinar si el problema a resolver es simple o complejo.
- Para los problemas simples, también llamados casos base, existe una solución directa.
- Para los problemas complejos, o casos generales, se hacen una o más llamadas recursivas cuyos resultados se usan para obtener la solución.
- Los parámetros usados en cada una de las llamadas recursivas representan problemas más simples y tienden hacia alguno de los casos base definidos.

Si una función recursiva está bien diseñada terminará encontrando la solución a cualquier caso, pues se tratará de un caso base o bien la solución la encontrará a partir de los resultados de casos intermedios que la lleven a uno base.

Las funciones recursivas son importantes porque permiten implementar algoritmos recursivos, y hay muchos problemas cuya solución se puede expresar más clara y elegantemente de forma recursiva.

La función Factorial

Muchas construcciones matemáticas se pueden definir recursivamente con una relación de recurrencia. Para estos casos resulta muy sencillo escribir funciones recursivas bien diseñadas, pues las definiciones incluyen todo lo necesario: los casos simples, sus soluciones y la fórmula para obtener la solución a los problemas complejos a partir de otros más simples. La función Factorial, que aparece definida en la Fig. 2, es una de las más conocidas.

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

Fig. 2 Función Factorial

La Fig. 3 muestra el código en C de una función recursiva que calcula el Factorial de cualquier entero positivo.

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

Fig. 3. Función recursiva que calcula el Factorial de un entero positivo

Como se puede ver, este ejemplo cumple todas las condiciones para considerarse una función recursiva correctamente diseñada:

- En la primera línea del código se decide si el problema es simple o complejo en función del valor del parámetro `n`.
- Si es menor o igual a cero se trata del caso base y se resuelve directamente devolviendo un uno, según se indica en la propia definición de Factorial.
- Si el problema es complejo se obtiene su solución multiplicando `n` por el resultado de calcular `factorial( n-1 )`, que es un problema idéntico estructuralmente pero para un caso más simple.
- Además, al usarse `n-1` como parámetro de la llamada recursiva se consigue que converja hacia cero que es el caso base y, por tanto, que se termine la recursión.

Siendo muy escrupulosos, la condición que determina el caso base debería ser `(n==0)`. Sin embargo, al comparar con menor o igual que cero la función se protege ante errores en los datos de entrada cuando `n` sea negativa.

Por otro lado, aunque la función es correcta, el hecho de usar el tipo de datos `int` limita su eficacia. Los cálculos intermedios superan rápidamente el valor máximo para un entero positivo haciendo que para `n > 12` los resultados sean incorrectos.

Finalmente, la función `factorial` está escrita usando una construcción típica de C. Como la ejecución de `return` causa la terminación de la función, en lugar de una sentencia `if-else` se usa un solo `if`. Las sentencias que hay tras la del `if` sólo se ejecutan si la condición no se cumple. Pero, evidentemente, el código se puede escribir de muchas otras formas como muestra la Fig. 4.

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

```
int factorial( int n ) {
  if ( n > 0 ) {
    return n * factorial( n - 1 );
  } else return 1;
}
```

Fig. 4. Otras formas de escribir la función factorial

Diseño de la función Factorial

Para entender mejor el diseño recursivo de la función Factorial, se plantea aquí un enfoque alternativo en el que se empieza resolviendo cada problema con una función diferente. Por ejemplo, para calcular el factorial de 3 se podrían escribir las cuatro funciones mostradas en la Fig. 5.

```
int factorial_3() {
  return 3 * factorial_2();
}
```

```
int factorial_2 () {
  return 2 * factorial_1();
}
```

```
int factorial_1() {
  return 1 * factorial_0();
}
```

```
int factorial_0 () {
  return 1;
}
```

Fig. 5. Funciones para calcular el Factorial de tres

Cada función calcula el Factorial de un número concreto. Lo hace multiplicando dicho número por el Factorial de otro número concreto, de cuyo cálculo se encarga otra función. Sólo en el caso de `factorial_0` se tiene una solución directa que, por definición, es 1.

Las tres primeras funciones: `factorial_3`, `factorial_2`, y `factorial_1`, son idénticas en cuanto a estructura, por lo que, aplicando abstracción operacional, se pueden sustituir por una única función con un parámetro como la de la Fig. 6.

```
int factorial_N( int n ) {
  return n * factorial_N( n - 1 );
}
```

Fig. 6. Función recursiva que describe el caso general para el cálculo de Factorial

La función recursiva `factorial_N` describe el caso general para calcular el Factorial de cualquier número mayor que cero. Sin embargo, no es correcta puesto que nunca llega a terminar. La condición de terminación se encuentra en `factorial_0`, pues muestra claramente que cuando `n` vale cero no se necesita invocar a ninguna otra función. De hecho, también en `factorial_0` aparece la solución a al caso base de la recursión que, por definición, es 1.

Para añadir a `factorial_N` el comportamiento descrito en `factorial_0`, y conseguir así que el proceso recursivo se detenga por no ser necesario simplificar más el problema, se usa una sentencia `if` que permite distinguir el caso base del general. Así pues, finalmente se obtiene a la función ya mostrada en la Fig. 3.

Aunque no todas las funciones recursivas se diseñan igual, tratar de responder las siguientes preguntas puede facilitar el diseño:

¿Cómo se define el problema en términos de la solución a una versión más pequeña o simple del mismo problema?

¿Cómo se reduce o divide un problema complejo para obtener otro más simple?

¿Cuál es el problema más pequeño o simple y cuál es su solución?

¿Se llega al problema más simple con la forma de reducción o división elegida?

La pila del sistema

Como pasa con cualquier otra llamada a función, la ejecución de las funciones recursivas causa un cambio en el flujo del programa, pues la siguiente instrucción en ejecutarse será la primera de la función llamada. Al acabar la llamada, la ejecución debe seguir por donde iba antes de hacerla, y esto requiere guardar el estado del programa justo antes de hacer la invocación. Para ello se crea un "registro de activación" que es almacenado en la Pila del sistema. Los registros de activación son estructuras de datos que contienen los valores de los parámetros usados en la invocación, las variables locales, la dirección de retorno y el valor a devolver. Al terminar la llamada, el registro de activación se saca de la Pila y la ejecución continúa por la instrucción siguiente a la que produjo la llamada.

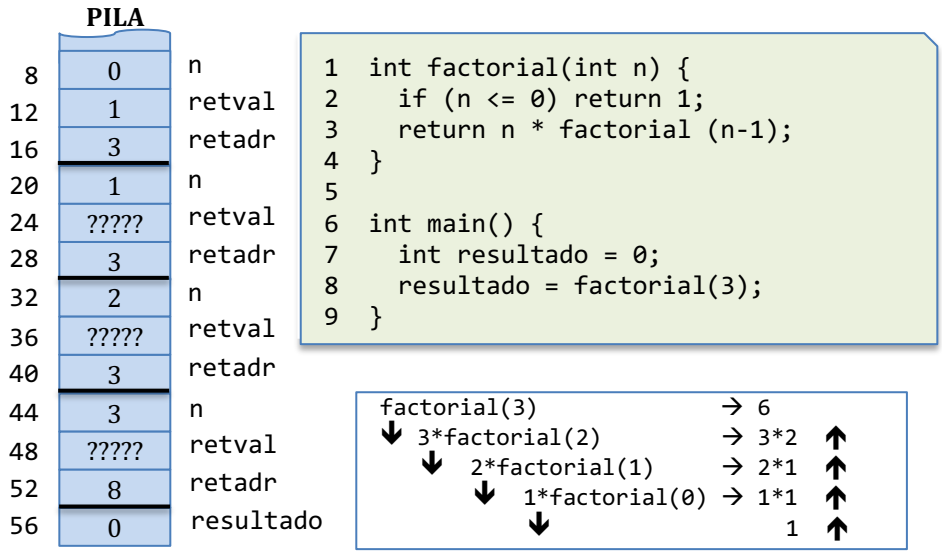

Fig. 7 Representación gráfica de la Pila del sistema a mitad de la llamada recursiva a factorial(3)

La Fig. 7 muestra el estado de la Pila del sistema tras ejecutar `factorial(3)` justo en el momento (línea 2 del código) en el que se va a devolver el valor de `factorial(0)`. Esta habrá sido la última llamada recursiva; la que detectó el caso base y terminó la recursión devolviendo un `1`. Este valor es el que va a usar `factorial(1)` para devolver `1*1` y así sucesivamente se irán sacando de la Pila del sistema los distintos registros de activación hasta que `factorial(3)` termine dejando en la variable `resultado` el valor `1*2*3 = 6`.

Aunque el código que se ejecuta sea el mismo, cada llamada recursiva es independiente de las demás. Es decir, cada una se ejecuta con sus propias variables locales que no interfieren con las de las demás llamadas. Por lo tanto, es posible, y conveniente, pasar información de una llamada a otra a través de los parámetros.

Como se puede ver, el proceso completo de ejecución tiene dos fases. En la primera, las llamadas van quedando a medio en espera de obtener resultados a problemas más simples que los que cada una trata de resolver. Esta fase acaba al alcanzarse el caso base. Y en la segunda fase, las llamadas van terminando, y la solución global se va componiendo poco a poco hasta que la primera de las llamadas termina con la solución al problema original.

Las funciones recursivas hacen un uso intensivo de la Pila. Por tanto, al evaluar su rendimiento también se debe tener en cuenta este aspecto.

Cálculo del Máximo Común Divisor

Otro ejemplo similar al de Factorial es el cálculo del máximo común denominador o mcd. En este caso el problema está representado por el valor de dos números distintos. La Fig. 8 muestra una relación de recurrencia que define cómo se calcula.

$$mcd(m,n) = \begin{cases} m & m = n \\ mcd(m-n,n) & m > n \\ mcd(m,n-m) & m < n \end{cases}$$

Fig. 8 Función para calcular el máximo común denominador

La Fig. 9 muestra una función recursiva que realiza el cálculo del mcd de dos números según la definición anterior. En este ejemplo la complejidad del problema a resolver depende de la diferencia de valor que haya entre los dos parámetros. El caso base o trivial se da cuando los dos valores son iguales y se soluciona devolviendo cualquiera de ellos y terminando. El caso general se da cuando no coinciden y se resuelve restándole el menor al mayor y aplicando recursivamente la misma función sobre la nueva pareja de valores.

```
int mcd ( int m, int n ) {
  if ( m == n ) return m;
  if ( m < n ) return mcd( m, n-m );
  return mcd( m-n, n );
}
```

Fig. 9. Función recursiva que calcula el máximo común divisor de dos enteros

Independientemente del valor inicial de `m` y `n` al restarle el menor al mayor, ambos números estarán más cerca el uno del otro. Por lo tanto, se garantiza que los parámetros usados en la llamada recursiva del caso general tienden hacia el caso base. Y aunque en el código aparecen dos llamadas recursivas, en cada ocasión sólo se hace una de las dos dependiendo de cuál de los dos valores es el mayor.

Función potencia

Para calcular la potencia n-ésima de un número basta multiplicar dicho número por sí mismo n veces. Pero la Fig. 10 muestra una relación de recurrencia que también define cómo realizar su cálculo de forma recursiva.

$$m^n = \begin{cases} 1 & \text{si } n = 0 \\ m \cdot m^{n-1} & \text{si } n > 0 \end{cases}$$

Fig. 10 Función para calcular el la función potencia

La Fig. 9 muestra una función recursiva que implementa este algoritmo. Aunque hay dos parámetros, la complejidad depende únicamente del segundo. El caso base o trivial se da cuando éste vale cero y se soluciona devolviendo 1. El caso general se resuelve multiplicando `m` por el resultado de calcular una potencia menor de la misma base: `potencia(m, n-1)`.

```
int potencia( int m, int n ) {
  if ( n == 0 ) return 1;
  return m * potencia( m, n-1 );
}
```

Fig. 11. Función recursiva que calcula la n-ésima potencia de m

La sucesión de Fibonacci

En algunos problemas es necesario hacer más de una llamada recursiva para poder resolver el caso general. Esto es lo que se conoce como recursividad múltiple. Por ejemplo, en la sucesión de Fibonacci cada número se define recursivamente a partir de los dos anteriores excepto los dos primeros que son 0 y 1 respectivamente.

$$f_n = f_{n-1} + f_{n-2}$$
$$f_0 = 0$$
$$f_1 = 1$$

Fig. 12 Sucesión de Fibonacci

A partir de esta definición resulta muy sencillo escribir una función recursiva como la de la Fig. 13 que calcule el n-ésimo número de la sucesión de Fibonacci.

```
int fibonacci ( int n ) {
  if ( n < 2 ) return n;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Fig. 13. Función recursiva que calcula el n-ésimo número de la sucesión de Fibonacci

Sin embargo, esta función es muy ineficiente porque repite muchos cálculos. Para entenderlo basta dibujar el árbol de recursión de una llamada concreta. El correspondiente a la llamada `fibonacci(6)` se puede ver en la Fig. 14.

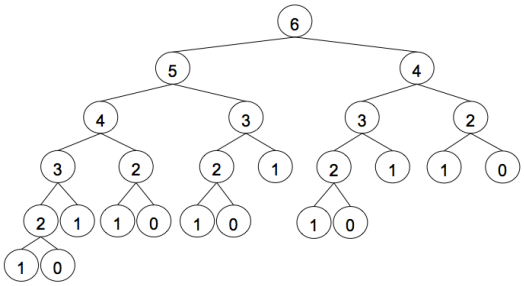

Fig. 14 Árbol de recursión de la llamada fibonacci(6)

En un árbol de recursión se muestran todas las llamadas recursivas que se realizan a partir de una dada. Cada llamada se representa como un círculo en el que aparecen los valores pasados como parámetros. En la parte superior se coloca el círculo correspondiente a la llamada inicial. Y cuando para resolver una llamada es necesario hacer más llamadas recursivas, los círculos que las representan se añaden en un nivel inferior conectados a la primera mediante líneas.

El árbol de la Fig. 14 muestra claramente que para calcular el sexto término hay que calcular el quinto y el cuarto; que para calcular el quinto también hay que repetir todos los cálculos para obtener el cuarto y así sucesivamente. Como puede verse hay muchos términos que se calculan varias veces haciendo que esta función sea muy ineficiente. De hecho, el número de llamadas crece de forma exponencial.

Un último aspecto a tener en cuenta cuando se escriben funciones recursivas múltiples es que el caso o casos base deben cubrir las necesidades de todas las llamadas recursivas. En este ejemplo, la segunda llamada recursiva requiere un valor de un elemento dos puestos anteriores. Si el caso base sólo se diera cuando `n` vale cero, en `fibonacci(1)` se llamaría a `fibonacci(-1)` y daría error.

Procedimientos recursivos

Funciones como `factorial`, `mcd` o `fibonacci` calculan un valor a partir del resultado de aplicar esas mismas funciones para un caso más simple. Pero hay ocasiones en las que el problema a resolver no tiene como solución un valor sino un efecto. En este caso se usan procedimientos: funciones que devuelven `void`.

Por ejemplo, el problema consistente en mostrar una cuenta atrás dado el valor inicial, se puede solucionar recursivamente del siguiente modo. Para resolver el caso general, es decir, contar hacia atrás desde n, primero se muestra el valor de n, y después, se resuelve el mismo problema para n-1. Si se continua así, al final se llega a cero. Por lo tanto, la solución recursiva se puede plantear del siguiente modo:

- **Caso base**: La cuenta atrás desde un valor menor que cero no requiere hacer nada.
- **Caso general**: La cuenta atrás desde n se hace mostrando el valor de n y haciendo **después** la cuenta atrás desde n-1.

```
void cuenta_atras( int n ) {
  if ( n < 0 ) return;
  printf( "%d ", n );
  cuenta_atras( n-1 );
}
```

Fig. 15. Función recursiva que hace una cuenta regresiva

En la Fig. 15, se ve claramente cómo primero se muestra el valor de `n` y después se hace la llamada recursiva con `n-1`. El caso base, que se da cuando `n` alcanza un valor negativo, permite detener la recursión usando para ello la sentencia `return`, pero, en este caso, sin devolver ningún valor.

Lo interesante de esta solución es que la contribución de la llamada recursiva actual a la solución del problema original se produce antes de tratar recursivamente el resto del problema. De este modo, cuando se alcanza el caso base, el problema original está completamente resuelto. Y sólo queda esperar que las llamadas recursivas terminen en el orden inverso al que se produjeron, pero sin hacer nada más. Esto sucede siempre que la llamada recursiva es la última instrucción del procedimiento, y se conoce como recursividad de cola o terminal.

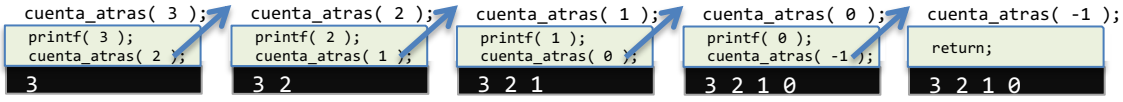

Fig. 16. Representación gráfica de la llamada recursiva a `cuenta_atras(3)`

La estrategia usada en este ejemplo imita directamente el comportamiento iterativo de soluciones implementadas con sentencias como `while` o `for`. El problema se resuelve poco a poco desde el principio. Cada llamada recursiva se encarga de una pequeña parte del problema directamente y deja la solución del resto del problema a la siguiente llamada recursiva.

Las variables locales que se usarían para resolver el problema con un bucle `while` o `for`, n en este caso, se convierten aquí en parámetros porque esta es la única forma razonable de compartir información entre distintas llamadas. Se podrían usar variables globales, pero esto es poco recomendable.

Como se verá después, pocas veces resulta interesante resolver recursivamente este tipo de problemas porque es más sencillo hacerlo de forma iterativa.

Por otro lado, para resolver el problema contrario, es decir, escribir una función recursiva que muestre los primeros n números naturales en orden, no se puede usar la misma estrategia. Es necesario construir la solución a partir del caso base, no al contrario.

Se empieza identificando el caso más simple que, en este ejemplo, es el consistente en mostrar cero números naturales. Este caso se resuelve no haciendo nada, pues no hay que mostrar ningún número. A partir de ahí, suponiendo que el problema ya está resuelto para un valor de n-1, es decir, que ya se ha mostrado la secuencia (1, 2, 3, …, n-1), lo que queda por hacer es mostrar un número más, en concreto el n. Así pues, la solución recursiva será:

- **Caso base**: La cuenta adelante de cero números no requiere hacer nada.
- **Caso general**: La cuenta adelante de n números se resuelve haciendo la cuenta adelante de n-1 números y mostrando **después** el valor de n.

```
void cuenta_adelante( int n ) {
  if ( n == 0 ) return;
  cuenta_adelante( n-1 );
  printf( "%d ", n );
}
```

Fig. 17. Función recursiva que hace una cuenta hacia delante

Lo más interesante de esta solución es que la llamada recursiva se hace antes de resolver la tarea asignada a la llamada actual. Esto concuerda con la idea de que la solución para el caso general dependa de que primero se resuelva algún caso más simple, que es la filosofía básica a la hora de construir funciones recursivas.

La diferencia entre esta implementación y la de la función `cuenta_atras` es mínima. Sólo cambia el orden de las dos últimas instrucciones, pero el resultado es totalmente diferente.

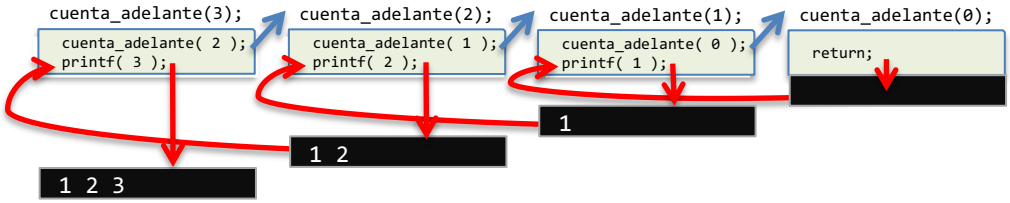

Fig. 18. Representación gráfica de la llamada recursiva a `cuenta_adelante(3)`

La estrategia usada en este segundo ejemplo es la inversa a la del primero. Cada llamada recursiva requiere de la solución a un problema más simple para después poder contribuir con su parte a la solución del problema original. Por tanto, hasta que no se ha resuelto la llamada recursiva para el caso n-1 no se puede construir la solución para el caso n.

Tratamiento recursivo de cadenas en C

Las cadenas de caracteres en C se pueden considerar una estructura de datos recursiva pues admiten la siguiente definición:

- **Caso base**: La cadena vacía (`"\0"`) es la más pequeña de las cadenas.
- **Caso general**: Cualquier carácter seguido de una cadena es una cadena.

Es decir, una cadena es, o bien la cadena vacía representada únicamente por el carácter elegido como marca de fin, o bien un carácter seguido de otra cadena.

Las cadenas en C se almacenan en arrays de caracteres, por tanto, las variables que las representan pueden indexarse usando el operador `[]`. Pero también pueden usarse variables de tipo puntero a carácter (`char *`) para recorrerlas gracias a la aritmética de punteros.

Si se tiene un puntero al inicio de una cadena y se incrementa el valor de dicho puntero en una unidad, éste apuntará al siguiente carácter de la cadena, que, por definición, se puede considerar una cadena que comience precisamente en ese carácter.

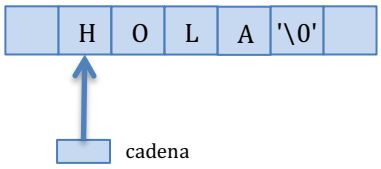

Fig. 19 Representación gráfica de un puntero a una cadena de caracteres en C

Por ejemplo, en la Fig. 19 aparece una cadena en memoria conteniendo el texto "HOLA" y una variable de tipo puntero a carácter, llamada `cadena`, que la representa porque apunta a la zona de memoria donde está la letra 'H'. Como `cadena` es un puntero, `cadena + 1` será otro puntero que apuntará a la letra 'O', es decir, `cadena + 1` representa la cadena "OLA".

La definición recursiva de cadena que se acaba de hacer permite escribir funciones recursivas que procesen cadenas. Y, aunque no sea lo más habitual, ni lo más eficiente, es interesante porque permite practicar la aritmética de punteros en C.

Por ejemplo, el problema consistente en calcular la longitud de una cadena se puede resolver recursivamente del siguiente modo:

- **Caso base**: La longitud de la cadena vacía es cero.
- **Caso general**: La longitud de una cadena no vacía es una unidad más que la longitud de la cadena que empieza en su segundo carácter.

```
int longitud( char * cadena ) {
  if ( cadena[0] == '\0' ) return 0;
  return 1 + longitud( cadena + 1 );
}
```

Fig. 20. Función recursiva que devuelve la longitud de una cadena de caracteres en C

La primera línea de la función `longitud`, mostrada en la Fig. 20, comprueba si se trata del caso base, es decir, si la primera letra de la cadena es la marca de fin, lo que implicaría que se trata de la cadena vacía. Y en ese caso se devuelve cero. En caso negativo se ejecuta la segunda instrucción, que termina devolviendo una unidad más de lo que devuelva la llamada recursiva a `longitud`, pero con otra cadena distinta: la que empieza en la segunda letra. Por lo tanto, en cada llamada recursiva el parámetro `cadena` apunta a una letra situada más cerca del final del array hasta que, finalmente, se alcanza la marca de fin. En ese momento la función termina devolviendo cero y, a medida que las distintas llamadas vayan terminando, se irán sumando unos hasta obtener la longitud de la cadena original.

Otro problema interesante es el consistente en mostrar una cadena letra a letra. La solución recursiva se puede definir del siguiente modo:

- **Caso base**: La cadena vacía no requiere hacer nada.
- **Caso general**: Para mostrar una cadena se imprime su primera letra y, después, se muestra el resto de cadena.

La estrategia usada aquí es similar a la usada en la función `cuenta_atras` (Fig. 15).

```
void muestra( char * cadena ) {
  if ( cadena[0] == '\0' ) return;
  printf( "%c ", cadena[0] );
  muestra ( cadena + 1 );
}
```

Fig. 21. Función recursiva que muestra una cadena letra a letra en orden natural

En este procedimiento se usa `return` en el caso base para terminar la ejecución de la función, no para devolver un valor. Pero lo más importante es que **antes de hacer la llamada recursiva ya se habrá mostrado la primera letra de la cadena actual**, es decir, el trabajo de la llamada actual ya está hecho y sólo queda tratar recursivamente el resto de la cadena para mostrar sus letras una a una.

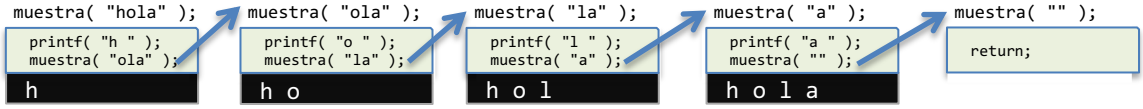

Fig. 22 Representación gráfica de la llamada recursiva a `muestra( "hola" )`

Para resolver recursivamente el problema contrario, es decir, mostrar la cadena al revés, se empieza por pensar que el caso más simple consiste en mostrar la cadena vacía. A partir de ahí, suponiendo que al terminar la llamada recursiva ya se habrá mostrado el resto de la cadena actual del revés, bastaría con mostrar después la primera letra de esa misma cadena para solucionar el problema.

- **Caso base**: La cadena vacía no requiere hacer nada.
- **Caso general**: La cadena invertida se construye mostrando su primera letra después de mostrar el resto de la cadena invertido.

```
void artseum( char * cadena ) {
  if ( cadena[0] == '\0' ) return;
  artseum( cadena + 1 );
  printf( "%c ", cadena[0] );
}
```

Fig. 23. Función recursiva que muestra una cadena letra a letra del revés

La función `artseum` (`muestra` al revés) es igual que `muestra`, pero con las dos últimas instrucciones intercambiadas. Lo que se consigue así es que la impresión se haga en la segunda fase de la recursión, es decir, cuando las distintas llamadas recursivas van terminando (flechas rojas en la Fig. 24) y, por tanto, el resultado será la impresión al revés pues la primera llamada que imprime es `artseum("a")`.

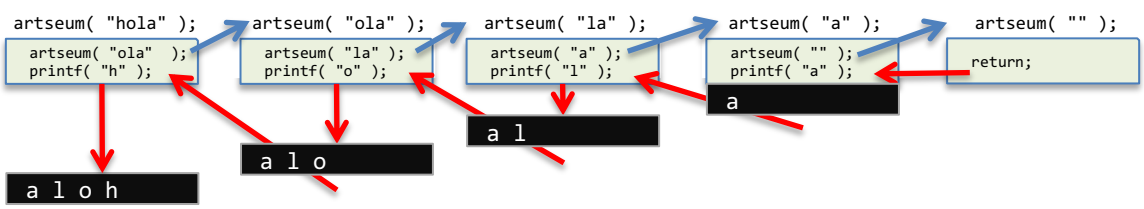

Fig. 24 Representación gráfica de la llamada recursiva a `artseum( "hola" )`

Finalmente, se muestra aquí otro ejemplo de tratamiento recursivo de cadenas en C, igualmente interesante, pero algo más complejo de implementar. Se trata del problema consistente en determinar si una cadena es igual que otra.

Para abordar este problema conviene empezar por definir recursivamente el proceso de construcción de dos cadenas iguales:

- **Caso base**: La cadena vacía sólo es igual a la cadena vacía.
- **Caso general**: Dos cadenas iguales a las que se les concatena por delante una misma letra también son iguales.

Es decir, dos cadenas son iguales si ambas son vacías, o si su primer carácter coincide y, además, las dos cadenas restantes son iguales.

Por tanto, se puede comparar el primer carácter de las dos cadenas y, en función del resultado, continuar estudiando el resto de la cadena o no. Si las cadenas son iguales la recursión acabará al llegar al final de ambas, y si son distintas la recursión acabará al encontrar las primeras dos letras diferentes. Las cadenas pueden tener distinta longitud y, en ese caso, en algún momento de la recursión una cadena será vacía y la otra no por lo que el carácter a comparar de la no vacía será distinto de la marca de fin. Por tanto, la recursión acabará devolviendo falso. La Fig. 25 muestra una función que implementa la solución.

```
int iguales( char * a, char * b ) {
  if ( *a == '\0' && *b == '\0' ) return 1; // Ambas son la cadena vacía
  return *a == *b && iguales( a+1, b+1 );
}
```

Fig. 25. Función recursiva que determina si dos cadenas de caracteres en C son iguales

El caso base consiste únicamente en comprobar si estamos ante dos cadenas vacías para acabar devolviendo 1. El caso general se resuelve con dos comprobaciones que deben cumplirse simultáneamente. La expresión booleana se ha escrito poniendo primero, es decir, a la izquierda, la comparación de los caracteres iniciales. Semánticamente sería lo mismo ponerlo al revés, pero hacerlo de esta forma es más eficiente porque C hace evaluación en cortocircuito de las expresiones lógicas. Por tanto, si la parte izquierda de una condición tipo "Y lógico" es falsa, la parte derecha no se evalúa ya que el resultado será falso de todos modos. Así pues, si los primeros caracteres de cada cadena son distintos no se llega a realizar la llamada recursiva y se evita todo el trabajo que eso podría implicar.

Por otro lado, en este ejemplo en lugar de acceder al primer carácter de cada cadena usando el acceso indexado con el operador [] se ha utilizado el operador de desreferencia (*) por ser una práctica muy habitual al escribir código en C.

Tratamiento recursivo de estructuras de datos enlazadas lineales

Las estructuras de datos enlazadas lineales sirven para representar secuencias de elementos y también se definen de forma recursiva. Consisten en una colección de nodos y cada uno está formado por un elemento de la secuencia y un puntero al nodo que contenga el siguiente elemento de la secuencia.

```
struct Nodo {
  int dato;
  struct Nodo * sig;
};
typedef struct Nodo * NodoPtr;
```

Fig. 26. Definición de una estructura enlazada lineal

La Fig. 26 muestra una forma de definir recursivamente dicha estructura en C. Las funciones recursivas permiten procesar simple y elegantemente estructuras de datos como esta. En general, la estrategia usada coincide con las operaciones similares realizadas para cadenas, pero en este caso, la forma de avanzar de un elemento al siguiente es a través del puntero `sig` de cada nodo.

En este apartado se van a estudiar algunas funciones recursivas que trabajan con estructuras enlazadas lineales (listas). Sin embargo, se va a suponer que éstas NO usan la estrategia del nodo cabecera. Para usar las funciones con listas que sí incluyan nodo cabecera, basta con invocar a las funciones pasando la dirección del primer nodo que guarda un elemento, es decir, el siguiente a la cabecera.

Por ejemplo, la longitud de una lista se calcula sumándole uno la longitud de la lista que empieza en el segundo nodo de la lista actual.

```
int longitud( NodoPtr lista ) {
  if ( lista == NULL ) return 0;
  return 1 + longitud( lista->sig );
}
```

Fig. 27. Función recursiva que calcula la longitud de una estructura enlazada lineal

La función recursiva que muestra todos los elementos de una lista en su orden natural, Fig. 28, sigue el modelo de las funciones `cuenta_atras` para enteros y `muestra` para cadenas. Es decir, procesa un elemento y deja el procesado del resto de la lista para la siguiente llamada recursiva: `muestra( lista->sig )`.

```
void muestra( NodoPtr lista ) {
  if ( lista == NULL ) return;
  printf( "%d ", lista->dato );
  muestra( lista->sig );
}
```

Fig. 28. Función recursiva que muestra una estructura enlazada lineal

Por su parte, para mostrar los elementos en el orden inverso, la función `artseum` mostrada en la Fig. 29 aplica la misma estrategia que la función de igual nombre para cadenas, y que la función `cuenta_adelante`. En cada llamada a la función, antes de mostrar el primer elemento de la lista actual, se deja que la llamada recursiva muestre el resto de lista del revés.

```
void artseum( NodoPtr lista ) {
  if ( lista == NULL ) return;
  artseum( lista );
  printf( "%d ", lista->dato );
}
```

Fig. 29. Función recursiva que muestra una estructura enlazada lineal del revés

La liberación de la memoria asociada a una estructura enlazada debe hacerse con cuidado. No se debe liberar un nodo cuyos enlaces deban ser usados posteriormente para acceder a nodos que aún no hayan sido liberados. Por lo tanto, una posible estrategia consistiría en liberar en primer lugar el último nodo, después el anterior y así hasta llegar al primero. Así pues, una solución recursiva que aplique esa estrategia se definiría así:

- **Caso base**: La lista vacía (NULL) no requiere liberación.
- **Caso general**: Una lista no vacía se libera liberando primero la lista que comienza en el segundo nodo y después el primer nodo.

En la función libera de la Fig. 30 la última instrucción es justamente la que libera el primer nodo de la estructura enlazada que se recibe como parámetro. Por tanto, en ese momento ya se habrá liberado el resto de la lista de forma recursiva.

```
void libera( NodoPtr lista ) {
  if ( lista == NULL ) return;
  libera( lista->sig );
  free( lista );
}
```

Fig. 30. Función recursiva que libera la memoria de una estructura enlazada lineal

La igualdad entre listas se resuelve de forma similar a lo que se hizo con cadenas:

- **Caso base**: Dos listas vacías son iguales.
- **Caso general**: Dos listas no vacías son iguales si sus primeros elementos lo son y las listas que comienzan a continuación también son iguales.

La implementación de la función `iguales` mostrada en la Fig. 31 es ligeramente distinta a la mostrada para cadenas. En una cadena la marca de fin es un dato más que puede ser consultado y comparado con otro dato normal, pero el final de una lista se alcanza cuando no hay más nodos, es decir, cuando la lista es la dirección NULL. Por tanto, no es seguro acceder al dato de la lista sin saber si esta es vacía o no. La solución elegida aquí consiste en determinar primero si alguna de las dos listas es vacía y usar las propias direcciones para decidir si ambas son vacías (iguales) terminando la función devolviendo uno, o si sólo una es vacía (distintas) terminando la función devolviendo cero. De este modo, sólo cuando las dos listas son NO vacías se procede a comparar primero el primer elemento de ambas listas y después, en caso de que coincidan, el resto de la lista.

```
int iguales( NodoPtr a, NodoPtr b ) {
  if ( a == NULL || b == NULL ) return a == b;
  return a->dato == b->dato && iguales( a->sig, b->sig );
}
```

Fig. 31. Función recursiva que compara dos estructuras enlazadas lineales

Veamos ahora cómo copiar una estructura enlazada de forma recursiva. La idea es sencilla, consiste en insertar a continuación de la copia del primer nodo, la copia recursiva del resto de la estructura enlazada.

- **Caso base**: La copia de una lista vacía es una lista vacía.
- **Caso general**: La copia de una lista no vacía consiste en la copia del primer nodo seguida de la copia del resto de la lista.

```
NodoPtr copia( NodoPtr lista ) {
  if ( lista == NULL ) return NULL;
  NodoPtr nuevo = malloc( sizeof( struct Nodo ) );
  nuevo->dato = lista->dato;
  nuevo->sig = copia( lista->sig );
  return nuevo;
}
```

Fig. 32. Función recursiva que devuelve una copia de una estructura enlazada lineal

En la función `copia` mostrada en la Fig. 32, tras crear un nuevo nodo y copiar en su campo datos el dato del primer nodo de la estructura enlazada recibida como parámetro, se guarda en su campo `sig` el resultado de la llamada recursiva cuyo objetivo es copiar el resto de la lista. El caso base que finaliza la recursión cuando se intenta copiar `NULL` se comprueba al principio del código. Finalmente se devuelve el nuevo nodo que, evidentemente, será el primero de la nueva estructura enlazada.

Finalmente vamos a ver cómo invertir una lista de forma recursiva. Lo que se busca aquí es modificar los enlaces de los nodos para que al final queden enlazados en el sentido contrario al original, pero sin tener que reservar ni eliminar memoria en ningún momento.

La inversión de una lista se resuelve con el siguiente esquema recursivo:

- **Caso base**: La inversión de una lista vacía es la misma lista vacía.
- **Caso base**: La inversión de una lista de un único elemento es la misma lista.
- **Caso general**: La inversión de una lista de más de un elemento se obtiene colocando el primer elemento al final de la inversión de la lista que empieza en el segundo elemento.

Como puede verse en el código de la Fig. 33, en primer lugar, aparecen los dos casos base combinados en una única sentencia condicional. A continuación, y haciendo uso de la propia llamada recursiva, se invierte la lista que comienza en el segundo nodo. Y después, teniendo en cuenta que el campo `sig` del primer nodo (llamémosle A) aún contiene la dirección del que antes era el segundo (llamémosle B), y que ahora B está en último lugar, la sentencia "`lista->sig->sig = lista`" coloca el nodo A a continuación de B pasando a ser A el último de la lista. Para terminar, se marca el nodo A como último poniendo en su campo `sig` el valor `NULL` y se devuelve la dirección del nuevo primer nodo de la lista invertida que tiene almacenada la variable `invertida`.

```
NodoPtr invierte( NodoPtr lista ) {
  if ( lista == NULL || lista->sig == NULL ) return lista;
  NodoPtr invertida = invierte( lista->sig );
  lista->sig->sig = lista;
  lista->sig = NULL;
  return invertida;
}
```

Fig. 33. Función recursiva que invierte una lista

Recursión frente a iteración

Las funciones recursivas que sólo incluyen una llamada recursiva en su código se denominan funciones recursivas lineales o simples y son muy fáciles de transformar en sus equivalentes iterativas. Factorial, potencia y mcd son lineales.

La Fig. 34 muestra una versión iterativa de la función Factorial. Consiste en un bucle que va acumulando el resultado de las distintas multiplicaciones. Como se puede comprobar, la condición que hace terminar el bucle es la misma que sirve para distinguir entre el caso base y el caso general en la versión recursiva.

```
int factorial ( int n ) {
  int f = 1;
  while ( n > 0 ) {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Fig. 34. Función iterativa que calcula el factorial de un entero positivo

Esta versión iterativa tiene un tiempo de ejecución de O(n), siendo n el valor del entero cuyo Factorial se desea calcular. La versión recursiva también tiene la misma complejidad. Y aunque determinar la complejidad de una función recursiva puede ser una tarea compleja, en este caso es sencillo calcular su tiempo de ejecución, pues resulta evidente que el número de llamadas recursivas efectuadas coincide con el valor de n. Por lo tanto, el número de llamadas, y con él el tiempo de ejecución, crecen de forma lineal con el valor de n.

Es decir, tanto la solución recursiva como la iterativa realizan aproximadamente el mismo número de operaciones. Pero la solución recursiva tiene un coste que no aparece en la versión iterativa. El tiempo usado para la gestión de los registros de activación no es despreciable. Además, la Pila del sistema tiene un tamaño máximo limitado y, en muchas ocasiones, el número de llamadas recursivas necesarias para resolver un problema es tan grande que puede llenar la Pila. Por su parte, la solución iterativa equivalente, ni dedica tanto tiempo a crear registros de activación, ni está limitada por el tamaño de la Pila del sistema.

Las funciones recursivas más fáciles de transformar en su equivalente iterativa son aquellas que sólo tienen una llamada recursiva y, además, la llamada recursiva es la última instrucción de la función. De estas funciones se dice que presentan recursividad de cola o terminal y se debe evitar su uso. Las funciones `mcd`, `cuenta_atras`, o `muestra` presentan recursividad de cola.

Sin embargo, la expresividad de la recursividad y la elegancia con la que se pueden resolver ciertos problemas gracias a ella, ha dado lugar a la existencia de lenguajes de programación funcionales, como Haskell o Lisp, que no incluyen sentencias de iteración y se basan exclusivamente en la recursión como mecanismo para llevar a cabo acciones repetitivas.

Al programar en estos lenguajes sí se recomienda hacer uso intensivo de la recursividad de cola para evitar los desbordamientos de pila. La razón es que sus propios compiladores/intérpretes eliminan la recursividad de cola de forma automática al generar el código en ensamblador asociado al código de alto nivel.

En muchos casos las soluciones recursivas son más fáciles de escribir que las iterativas, pero hay que ser cuidadoso pues pueden ser mucho más ineficientes. La sucesión de Fibonacci se define fácil y elegantemente de forma recursiva, pero es mucho más eficiente calcularla de forma iterativa. Aunque, como muestra la Fig. 35, el código no es tan elegante ni tan claro como el de la versión recursiva.

```
int fibonacci ( int n ) {
  if ( n < 2 ) return n;
  int a = 1;
  int b = 0;
  for ( int i = 0; i < n; i = i + 1 ) {
    int c = a + b;
    a = b;
    b = c;
  }
  return b;
}
```

Fig. 35. Función iterativa que calcula el n-ésimo número de la sucesión de Fibonacci

Que las funciones recursivas puedan transformarse en iterativas y viceversa no significa que siempre haya que descartar la recursividad como herramienta para resolver problemas. La clave es saber cuándo aplicarla.

- Si una función es recursiva lineal su árbol de recursión será lineal y la versión iterativa de la misma función será siempre más eficiente.
- Si una función es recursiva múltiple su árbol de recursión será ramificado no lineal. Pero si contiene nodos duplicados, o bien se aplican técnicas de optimización para evitar la repetición de cálculos, o bien se opta por una solución iterativa más eficiente.
- Si una función recursiva produce un árbol de recursión no lineal y sin nodos duplicados entonces su uso sí será adecuado.

Es decir, las funciones recursivas conviene usarlas cuando no son muy ineficientes y no hay una alternativa iterativa más simple. Además, la recursividad, como herramienta de diseño, puede facilitar la comprensión del problema y ayudar a plantear soluciones iterativas más eficientes.

También se usan funciones recursivas para procesar estructuras de datos que se definen recursivamente pues, en estos casos, normalmente, la solución recursiva es mucho más simple que la iterativa y tienen un coste computacional similar.

Las funciones recursivas son especialmente adecuadas cuando la solución a un problema consiste en dividirlo en dos o más problemas con la misma estructura que, además de ser más simples, usen conjuntos de datos de entrada disjuntos para evitar que se repitan nodos en el árbol de recursión. El algoritmo de Ordenación por Mezcla (Mergesort) y el de Ordenación Rápida (Quicksort) son ejemplos de este tipo de algoritmos.

El diseño de los algoritmos que solucionan este tipo de problemas se basa habitualmente en la estrategia denominada “Divide y Vencerás” y se estudia en cursos superiores. Pero la idea en la que se basan todos ellos es simple. Consiste en dividir un problema en otros más simples, pero iguales en cuanto a estructura, y solucionar estos aplicando la misma estrategia hasta llegar a uno tan simple que su solución sea directa. La solución al problema original se obtiene combinando las soluciones a los problemas más simples.

Búsqueda binaria

La búsqueda binaria es un algoritmo recursivo que sólo se puede aplicar sobre conjuntos de datos ordenados. Consiste en comparar la clave, es decir, el elemento buscado, con uno que divida al conjunto de búsqueda en dos mitades y descartar la mitad en la que seguro que no puede estar. El proceso se repite en la otra mitad hasta encontrarlo. La idea es la misma que se aplica normalmente al buscar a mano en un diccionario de papel.

Asumiendo que n elementos están ordenados de menor a mayor en una secuencia, y situando el menor de todos en el lado izquierdo y el mayor en el derecho, si la clave es mayor que el elemento central se puede descartar el subconjunto de la izquierda. Y si fuera menor se descartaría el derecho. El proceso se repite considerando como conjunto de datos el subconjunto no descartado. De este modo, tras cada comparación quedan la mitad de elementos, lo que permite afirmar que sólo son necesarias $\log_2(n)$ operaciones para reducir el conjunto de búsqueda original a un único elemento que, en el peor de los casos, no será el buscado concluyendo que la clave no se encuentra en el conjunto de búsqueda.

Si para representar la secuencia de datos del conjunto de búsqueda se usa un array, el elemento de la posición `i` será menor que el de la posición `j` para todo `i<j`. Así pues, sabiendo el número total de elementos, el elemento central, llamado pivote, estará en la posición `n/2`. Pero, tras descartar uno de los subconjuntos no conviene crear un nuevo array y copiar en él los datos del array original sobre los que se seguirá aplicando el algoritmo. Es mucho más eficiente acceder al mismo array en todas las llamadas recursivas y usar índices que determinen el comienzo y el final del subconjunto de datos del array a tratar en cada caso.

El conjunto inicial abarcaría desde 0 hasta `n-1` siendo `n` la longitud del array original. El pivote o elemento central, denominado `m`, se situaría en `(n-1)/2` por lo que en la primera llamada recursiva, habría dos subconjuntos posibles: por un lado, desde 0 hasta `m-1` y, por otro lado, desde `m+1` hasta `n-1`.

```
int bb_aux( int numeros[], int i, int j, int clave ) {
  if ( i > j ) return -1;
  int m = i + ( j - i ) / 2;
  if ( clave < numeros[m] ) return bb_aux( numeros, i, m - 1, clave );
  if ( numeros[m] < clave ) return bb_aux( numeros, m + 1, j, clave );
  return m;
}

int busqueda_binaria( int numeros[], int n, int clave ) {
  return bb_aux( numeros, 0, n - 1, clave );
}
```

Fig. 36. Búsqueda binaria recursiva

La función recursiva necesita tener cuatro parámetros: el array de datos, la clave, la posición del primer elemento del conjunto de búsqueda y la posición del último elemento del conjunto de búsqueda. Pero resulta poco natural obligar al usuario a indicar el comienzo y el final del array, lo lógico sería que se tuviera que indicar únicamente su tamaño.

Para que el interfaz de uso de la función no se vea afectado por las necesidades de diseño del algoritmo recursivo se pueden usar dos funciones: la recursiva y la que ofrece públicamente la funcionalidad.

La primera tiene todos los parámetros necesarios según el diseño recursivo. La segunda tiene los parámetros mínimos imprescindibles que espera un usuario. En este caso, la función `bb_aux` (Fig. 36) es la que implementa el algoritmo de la búsqueda binaria de forma recursiva y la función `busqueda_binaria`, que no es recursiva, es la que se ofrece al usuario final.

La primera línea de la función `bb_aux` determina si se trata de un caso base o no. Cuando `i > j` es porque los índices se han cruzado y no hay más datos que explorar, por lo que el elemento buscado no se encuentra en el array. Si `i` es igual a `j`, aún queda por explorar el elemento situado en esa posición. De esta forma tanto los arrays con longitud par como los de longitud impar se exploran correctamente.

La posición del pivote (`m`) se calcula haciendo `i + ( j – i ) / 2` que es algebraicamente equivalente a `( i + j ) / 2`. Pero esta forma de hacer el cálculo evita sobrepasar el valor máximo de un entero al hacer la suma y, por tanto, obtener un valor incorrecto. Así se pueden ordenar arrays de la máxima longitud.

Tras calcular la posición del pivote, se compara el elemento situado ahí con la clave buscada, y según el resultado se hace la llamada recursiva con un subconjunto o con el otro. Pero, en cualquier caso, sólo se hace una llamada recursiva.

Nótese que si la clave coincide con `m` no se hace ninguna llamada recursiva y la función termina devolviendo la posición del elemento encontrado. Este sería un segundo caso base de la función. Como ya se ha dicho, el otro caso base se da cuando se ha terminado de buscar en todo el array sin éxito.

El tiempo de ejecución de este algoritmo es de O(log n) pues ese es el número de cálculos de elementos centrales y comparaciones con la clave que se realiza en el peor de los casos. Lo que implica que es más eficiente que la búsqueda secuencial que es de O(n). Obviamente, los datos deben estar ordenados y el coste de ordenarlos no se está teniendo en cuenta al evaluar la eficiencia de la búsqueda.

Esta función, al igual que Factorial, también es recursiva lineal ya que sólo se hace una llamada recursiva dentro de su código. Por lo tanto, es posible escribir una versión iterativa, más eficiente, aunque algo menos elegante. En concreto, la versión iterativa necesitará usar menos la Pila que la recursiva pero será igual de eficiente en cuanto a tiempo de ejecución pues sigue siendo de O(log n). La Fig. 37 muestra el código de la versión iterativa del algoritmo de búsqueda binaria.

```
int busqueda_binaria( int numeros[], int n, int clave ) {
  int i = 0;
  int j = n-1;
  while ( i <= j ) {
    int m = i + ( j - i ) / 2;
    if ( numeros[m] < clave ) i = m+1;
    if ( numeros[m] > clave ) j = m-1;
    if ( numeros[m] == clave ) return m;
  }
  return -1;
}
```

Fig. 37. Búsqueda binaria iterativa

Máximo de un array

En el caso de la función factorial, la complejidad del problema depende del valor de los parámetros de entrada, que en ese ejemplo sólo es `n`. Pero no siempre es así, en algunas ocasiones, la complejidad de un problema viene determinada por el tamaño del conjunto de datos que se deben tratar. En estos casos se suele conocer una solución directa cuando la cantidad de datos es pequeña. Y los casos generales se resuelven dividiendo el conjunto de datos original en otros más pequeños y resolviendo recursivamente cada uno de estos problemas por separado para terminar resolviendo el problema original combinando las soluciones anteriores. Es decir, la misma idea en la que se basa la estrategia "Divide y Venceras".

Por ejemplo, para calcular el máximo de un array de enteros se puede definir la solución recursivamente del siguiente modo:

- **Caso base**: Si un array sólo tiene un elemento ese será el máximo
- **Caso general**: Si un array tiene más de un elemento el máximo será el mayor de los máximos de los dos arrays resultantes de dividir el original en dos mitades.

El caso base se daría cuando el array tuviera un entero y el caso general cuando el número de elementos fuera mayor que uno.

La dificultad de este problema radica en cómo representar los datos de entrada para poder dividir el array eficientemente. En general no conviene crear nuevos arrays y copiar en ellos los datos del array original, así que la mejor alternativa es la usada en la búsqueda binaria: tratar todo el tiempo con el mismo array pero usar índices que determinen el comienzo y el final de la parte del array a tratar.

Así pues, inicialmente se consideraría desde `0` hasta `n-1` siendo `n` la longitud del array original. Y en la primera llamada recursiva se trataría, por un lado, desde `0` hasta `n/2` y, por otro lado, desde el siguiente elemento hasta `n-1`.

El código que implementa el algoritmo aparece en la Fig. 38 y, como se puede ver, se ha dividido en tres funciones distintas. La función `mayor` permite simplificar la comparación de dos valores. La función `m_aux` es la que implementa el algoritmo recursivo y `maximo` la que ofrece el interfaz público al usuario.

En la función `m_aux` aparece en primer lugar el caso base que se da cuando el array tiene un solo elemento (índices de `i` y `j` iguales) y después, tras calcular el centro resuelve el caso general haciendo dos llamadas y calculando el mayor de ambas.

```
int mayor( int a, int b ) {
  if ( a > b ) return a;
  else return b;
}

int m_aux( int numeros[], int i, int j ) {
  if ( i == j ) return numeros[i];
  int m = i + (j - i) / 2;
  return mayor( m_aux( numeros, i, m ), m_aux( numeros, m + 1, j ) );
}

int maximo( int numeros[], int n ) {
  return m_aux( numeros, 0, n - 1 );
}
```

Fig. 38. Función recursiva que calcula el máximo de los elementos de un array

Ordenación rápida (Quicksort)

Uno de los algoritmos de ordenación más eficientes es el denominado Quicksort. Este algoritmo usa la técnica "Divide y vencerás" del siguiente modo. Primero se elige uno de los valores del array a ordenar, al que se le llama pivote, y se procede a organizar todos los elementos del array en relación al pivote. Es decir, recolocarlos de modo que los que sean menores que el pivote queden a su izquierda (posiciones menores del array) y que los que sean mayores queden a su derecha. En ese momento el pivote estará colocado en su posición definitiva y dividirá el conjunto original en dos subconjuntos de datos que habrá que ordenar.

En el mejor de los casos, cuando el pivote elegido coincida con uno de los valores que, en la ordenación final, ocupe un lugar central, los dos subconjuntos que quedan por ordenar tendrán aproximadamente la mitad de elementos que el original. Pero, en cualquier caso, el siguiente paso consiste en aplicar el mismo algoritmo a cada subconjunto.

El final de la recursión está garantizado ya que cuando el conjunto de datos sólo tiene un elemento, evidentemente está ordenado, y no hay que volver a aplicar el algoritmo recursivamente.

```
1       void q_aux( int a[], int primero, int ultimo) {
2         int i, j, central;
3         int pivote;
4         central = primero + (ultimo - primero) / 2;
5         pivote = a[central];
6         i = primero;
7         j = ultimo;
8         do {
9           while ( a[i] < pivote ) i++;
10          while ( a[j] > pivote ) j--;
11          if ( i<=j ) {
12            int tmp;
13            tmp = a[i];
14            a[i] = a[j];
15            a[j] = tmp;
16            i++;
17            j--;
18          }
19        } while ( i <= j );
20        if ( primero < j ) q_aux( a, primero, j );
21        if ( i < ultimo ) q_aux( a, i, ultimo );
22      }
23
24      int quicksort( int numeros[], int n ) {
25        return q_aux( numeros, 0, n - 1 );
26      }
```

Partición del conjunto en dos

Solución recursiva para cada parte

Fig. 39. Función recursiva que implementa el algoritmo de Ordenación Rápida

La Fig. 39 muestra el código en C de una función recursiva que implementa el algoritmo de Ordenación Rápida para números enteros. Como en el caso de la búsqueda binaria o el cálculo recursivo del máximo de un array, se ha dividido el código en dos funciones: una función auxiliar que es la que implementa el algoritmo recursivo y otra que ofrece el interfaz público de Quicksort.

En la función `q_aux` el primer parámetro es el array a ordenar, y junto a él aparecen dos parámetros extra que delimitarán el intervalo de datos del array que se desea ordenar: los parámetros `primero` y `ultimo`.

Por lo tanto, para ordenar el array completo, la primera llamada a la función debería tener `0` y `n-1` como segundo y tercer argumentos respectivamente. La función `quicksort` hace esa llamada con el array que recibe como parámetro.

La función `q_aux` tiene dos partes. La primera parte se encarga de dividir el intervalo original en dos. Para ello, primero selecciona el elemento situado en el centro del intervalo como pivote (líneas 4 y 5). Después, el bucle que comienza en la línea 8 organiza los elementos del intervalo colocando a la derecha del pivote aquellos elementos que sean mayores que él y a la izquierda los menores.

Antes de comenzar el bucle las variables `i` y `j` empiezan valiendo respectivamente lo mismo que `primero` y `ultimo`. Al terminar el bucle, `j` es menor que `i` y el pivote ocupa una posición correcta en relación al resto de elementos del intervalo. Sólo queda ordenar los dos intervalos en los que el pivote ha dividido el intervalo original. El primer intervalo va desde `primero` hasta `j` y el segundo desde `i` hasta `ultimo`, y el pivote queda situado justo entre `j` e `i`.

La segunda parte resuelve la ordenación de cada uno de los dos intervalos aplicando recursivamente el mismo algoritmo. Y, aunque no lo parezca, el caso base se encuentra justo ahí. Las comprobaciones que se hacen en las líneas 20 y 21, antes de hacer la llamada recursiva, son las que se encargan de parar el proceso recursivo cuando el conjunto a ordenar ya lo está por contener un único elemento.

El estudio de la complejidad temporal de este algoritmo se hace en cursos superiores. Pero, básicamente se puede ver que, si las particiones dividen cada intervalo en dos mitades, como en el caso de la búsqueda binaria, harán falta $\log_2 n$ divisiones hasta llegar a intervalos de un elemento. Si en cada caso se necesitan n operaciones para recolocar los elementos alrededor del pivote, se puede decir que el tiempo de ejecución será de $O(n \log n)$ por lo que es un algoritmo muy eficiente.