RECURSIVIDAD

A. GARRIDO

Departamento de Ciencias de la Computación e I.A. ETS Ingeniería Informática
Universidad de Granada. 18071 Granada. Spain. Email: A.Garrido@decsai.ugr.es

Resumen

En este documento se pretende hacer una breve descripción de la programación de funciones recursivas en C. En este documento se presentan las posibilidades de la programación recursiva junto con un conjunto de reglas y de técnicas que faciliten al lector la comprensión así como la creación de nuevas funciones recursivas. El objetivo fundamental es que sirva como un repaso de algunos conceptos de recursividad para facilitar el estudio de otros temas tales como las estructuras de datos que requieren que el programador sepa aplicar la programación recursiva.

1 Introducción a la recursividad.

El concepto de recursividad no es nuevo. En el campo de la matemática podemos encontrar muchos ejemplos relacionados con él. En primer lugar muchas definiciones matemáticas se realizan en términos de sí mismas. Considérese el clásico ejemplo de la función factorial:

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

donde la definición se realiza por un lado para un caso que denominamos base ($n = 0$) y por otro para un caso general ($n > 0$) cuya formulación es claramente recursiva.

Consideremos por otro lado la conocida demostración por inducción que generalmente sólo necesita realizar una simple demostraci´n para un caso base y una segunda para un caso general de tamaño n (con la ventaja de saberlo demostrado para cualquier caso menor que n) de forma que queda demostrado para todos los casos.

De una manera similar, cuando programamos una función recursiva podemos decir que está resuelta para un tamaño menor (véase más adelante). Se podría afirmar que la gran virtud de la programación recursiva radica en que para resolver un problema, el usar esta técnica implica que ya "tenemos resuelto" el problema (de menor tamaño como ya veremos). De esta forma, la recursividad constituye una de las herramientas más potentes en programación.

Básicamente, un problema podrá resolverse de forma recursiva si es posible expresar una solución al problema (algoritmo) en términos de él mismo, es decir, para obtener la solución será necesario resolver este mismo problema sobre un conjunto de datos o entrada de menor tamaño.

En principio podemos decir que los problemas recursivos que generalmente son más fáciles de resolver son aquellos que de forma natural se formulan recursivamente. De esta forma, estos problemas ofrecen directamente una idea de cómo debe ser el algoritmo recursivo. Considérese por ejemplo el caso del factorial, que por definición nos dice que hay un caso base ($n = 0$) y un caso recursivo ($(n-1)!$) que junto con una multiplicación por n nos da el resultado.

Por otro lado, hay problemas que no se expresan de forma recursiva y por tanto es necesario formular un algoritmo que, resolviendo el problema, se exprese en términos de sí mismo. Por tanto, este tipo de problemas requiere un esfuerzo adicional que está especialmente indicado para los que la solución no recursiva es especialmente compleja o problemas sobre los que aplicando algún tipo de técnica de diseño de algoritmos nos llevan a soluciones recursivas (por ejemplo *divide y vencerás*, ver Brassard[2]) que son simples de construir y no requieren un nivel alto de recursos. Un ejemplo es el conocido algoritmo de búsqueda binaria o dicotómica. El problema consiste en localizar un elemento dentro de un vector de elementos ordenados. Es claro que el algoritmo se podría resolver sin ningún problema de forma iterativa pero si aplicamos la técnica de *divide y vencerás* podemos llegar al citado algoritmo. Éste consiste en

la división del vector en dos partes y la selección del subvector (donde debe estar el elemento a buscar según la ordenación) para continuar buscando (el mismo problema sobre el subvector).

2 Diseño de funciones recursivas.

El primer paso para obtener la implementación de la solución de un problema por medio de una función recursiva es la *identificación del algoritmo recursivo*, es decir, tenemos que identifical cuál es la "idea" recursiva. Más concretamente: los casos base, casos generales y la solución en términos de ellos.

1. **Los casos base**. Son los casos del problema que se resuelve con un segmento de código sin recursividad. Normalmente corresponden a instancias del problema simples y fáciles de implementar para los cuales es innecesario expresar la solución en términos de un subproblema de la misma naturaleza, es decir, de forma recursiva. Obviamente el número y forma de los casos base son hasta cierto punto arbitrarios, pues depende del programador y su habilidad el hecho de identificar el mayor conjunto de casos en términos de simplicidad y eficiencia. En general no es difícil proponer y resolver un conjunto de casos base ya que por un lado corresponden a instancias simples y por otro en caso de problemas muy complejos siempre es posible proponer varios casos para asegurar que el conjunto es suficientemente amplio como para asegurar el fin de la recursión. Por supuesto, la solución será mejor cuanto más simple y eficiente resulte el conjunto de casos seleccionados.

   Una regla básica a tener en cuenta con respecto al diseño de una función recursiva es básicamente que *siempre debe existir al menos un caso base*. Es obvio que las llamadas recursivas para la solución de un problema no pueden darse de forma indefinida sino que debe llegar un caso en el que no es necesario expresar la solución como un problema de menor tamaño, es decir, tienen que existir casos base. Por ejemplo, si consideramos una definición de fatorial del tipo $n! = n(n-1)!$, necesitamos añadir el caso $0! = 1$ para indicar una condición de parada en la aplicación de la ecuación recursiva. Por supuesto, esta regla es bastante obvia pues básicamente estamos afirmando que la recursividad debe parar en algún momento.

2. **Los casos generales.** Cuando el tamaño del problema es suficientemente grande o complejo la solución se expresa de forma recursiva, es decir, es necesario resolver el mismo problema, aunque para una entrada de menor tamaño. Es justo aquí onde se explota la potencia de la recursividad ya que un problema que inicialmente puede ser complejo se expresa como:

   (a) Solución de uno o más subproblemas (de igual naturaleza pero menor tamaño).

   (b) Un conjunto de pasos adicionales. Estos pasos junto con las soluciones a los subproblemas componen la solución al problema general que queremos resolver.

   Si consideramos los subproblemas como algo resuelto (de hecho generalmente sólo requieren de llamadas recursivas) el problema inicial queda simplificado a resolver sólo el conjunto de pasos adicionales. Por ejemplo, para el caso del factorial es necesario calcular el factorial de $n-1$ (llamada recursiva) y adicionalmente realizar una multiplicación (mucho más simple que el problema inicial). Obviamente esto nos llevará a la solución si sabemos resolver el subproblema, lo cual está garantizado pues en última instancia se reduce hasta uno de los casos base.

   Por tanto una regla básica a tener en cuenta con respecto al diseño de los casos generales de una función recursiva es que *los casos generales siempre deben avanzar hacia un caso base*. Por ejemplo, cuando diseñamos la función del cálculo de factorial, el caso general hace referencia al subproblema $(n-1)!$ que como es obvio está más cerca de alcanzar el caso base $n = 0$.

2.1 Implementación de funciones recursivas.

Es este apartado vamos a ver algunos consejos y reglas prácticas para implementar funciones recursivas. En primer lugar, se plantean las posibles consideraciones a la hora de especificar la cabecera de la función recursiva (sintaxis y semántica). En este sentido, una regla básica a tener en cuenta es que *la cabecera debe ser válida tanto para llamar al problema original como a uno de sus subproblemas.*

Supongamos que queremos resolver el problema de dado un vector de 100 enteros ordenados devolver la posición de un elemento a buscar x. Inicialmente este problema puede ser resuelto de la siguiente forma:

```
int buscar (int v[], int x)
{
  register int i;

  for (i=0;i<100;i++)
    if (v[i]==x) return i;

  return -1;
}
```

En cambio, si nuestra intención es resolver este mismo problema de forma recursiva mediante el algoritmo de búsqueda binaria, la cabecera de la función debe de ser modificada ya que es imposible expresar la llamada a un subproblema sin añadir algún parámetro adicional. Dado que un subproblema consiste en continuar la búsqueda en una parte del vector, cuando se llama recursivamente a solucionarlo será necesario especificar el subconjunto de elementos del vector donde buscar. Así una cabecera válida podría ser:

```
int buscar (int v[], int i, int j, int x)
```

que localiza el elemento x en las posiciones comprendidas entre i y j (ambas inclusive) del vector v.

En segundo lugar, el cuerpo de la función debe de implementar los casos base y generales. Ambos están muy relacionados de manera que tenemos que tener en cuenta cómo programamos los casos generales para poder definir los casos base y al revés.

Para la programación de los casos generales, una regla básica que puede ser de gran utilidad es *suponer que las llamadas recursivas a subproblemas funcionan*. Es poco recomendable intentar mentalmente entender el funcionamiento y el progreso de las distintas llamadas recursivas cuando se están programando. Generalmente el programador debe asumir que las llamadas van a funcionar, es decir, tenemos que asumir que la llamada a la función responde a las especificaciones que hayamos determinado para la función sin más consideraciones referentes al funcionamiento interno.

Para personas que se intentan iniciar en la programación de funciones recursivas, puede ser de utilidad considerar que *la función que se pretende resolver ya está programada para problemas de menor tamaño* (la llamada recursiva funciona). El objetivo de esta suposición es conseguir que el programador no intente imaginar el funcionamiento de sucesivas llamadas recursivas. Por ejemplo, supongamos que queremos programar la función factorial para un valor n. Podemos considerar que disponemos de una función *Resuelto* que implementa el factorial para valores menores que n y además no conocemos su implementación. Sólo conocemos que su especificación es idéntica a la función factorial aunque sólo se puede usar para valores menores que n. Así su implementación sería

```
int factorial (int n)
{

  if (n==0)
    return 1;
  else
    ...                                                    /* Caso general */
}
```

a falta de construir el caso general. Ahora bien, sabemos que tenemos disponible una función *ya implementada y que funciona* que se llama *Resuelto* con la misma especificación (sintaxis *int Resuelto (int n)* y semántica idénticos, es decir hace exactamente lo mismo pero con otro nombre). Obviamente esta función no se puede usar para el tamaño original n de la función *factorial* pero si para cualquier caso más pequeño. Con este planteamiento, la implementación del caso general de la función *factorial* es muy simple pues ya tenemos resuelto el caso de cualquier factorial más pequeño. Si tenemos que calcular el factorial de n sólo necesitamos realizar una multiplicación de éste valor por el factorial de *n-1* que precisamente ya lo tenemos en la función *Resuelto* quedando

```
int factorial (int n)
{

  if (n==0)
```

```
    return 1;
  else
    return n*Resuelto(n-1);
}
```

Por supuesto, la solución definitiva se obtiene cambiando el nombre *Resuelto* por el de la misma función

```
int factorial (int n)
{

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

Es importante destacar que el hecho de haber usado otra función *Resuelto* en la programación del factorial tiene como objetivo el alejar al lector de pensar que estamos programando una función recursiva sino que estamos programando una función iterativa con la gran ventaja de que la mayor dificultad del problema (resolver el subproblema) está ya resuelto en una función auxiliar. Nótese que incluso la función *Resuelto* no tendría que ser recursiva, es decir, podríamos pensar que implementa el factorial de forma iterativa. Por tanto, pierde todo el sentido el considerar el funcionamiento y relación interna de las distintas llamadas recursivas.

2.1.1 Un ejemplo: La búsqueda binaria.

En primer lugar es interesante recordar *la idea o algoritmo recursivo* que resuelve este problema de búsqueda: La búsqueda de un elemento en un vector ordenado se puede realizar comparando con el elemento central, si es menor se realiza una búsqueda del elemento en el subvector mitad izquierda y si es mayor en la mitad derecha. Es decir, la solución se puede exponer en términos del mismo problema.

La cabecera de la función a implementar ya se ha presentado antes (página 3). Nótese como en la misma idea que presentamos en el párrafo anterior ya hacemos referencia a que la búsqueda se realiza en una parte del vector original. De ahí que la cabecera tenga que permitir la referenciar la parte del vector donde buscar.

Inicialmente, podemos considerar que un caso muy simple del problema y al que podemos llegar a través de llamadas recursivas es el caso de que busquemos el elemento en un segmento del vector de tamaño 1, es decir, que $i == j$. Por tanto, por ahora fijamos éste como el único caso base.

Por otro lado, tenemos que resolver el caso general. Éste consiste en realizar búsquedas en los subvectores correspondientes. Insistiendo de nuevo en la misma idea de obviar el hecho de que tratamos con una función recursiva, supongamos que las llamadar recursivas funcionan considerando que tenemos una función *Resuelto* con la misma cabecera

```
int Resuelto (int v[], int i, int j, int x)
```

Esta función resuelve el problema de la búsqueda por algún método que no nos interesa. Sólo es necesario tener en cuenta que funciona sólo en caso de un subproblema. Si no, la solución podría ser

```
int buscar (int v[], int i, int j, int x)
{
  return Resuelto (v,i,j,x);
}
```

que obviamente nos implicaría que no avanzamos hacia el caso base (*recursión infinita*). Ahora el problema puede paracer más sencillo (¡aunque no lo es!) al considerar que ya tenemos una función que nos resuelve el mismo problema. El caso general se puede formular ahora como la comparación con el elemento central, si es menor, la función *Resuelto* nos da la solución sobre la parte izquierda y si es mayor sobre la parte derecha (esta función es aplicable pues tenemos una mitad del vector que está más cerca del caso base).

Una primera solución al problema de búsqueda es

```
int buscar (int v[], int i, int j, int x)
{
  int centro;

  if (i==j)
    return (v[i]==x)?i:-1;
  else {
    centro=(i+j)/2;
    if (v[centro]==x)
      return centro;
    else if (v[centro]>x)
            return Resuelto (v,i,centro,x);
          else return Resuelto (v,centro,j,x);
}
```

Nótese como nos ha surgido un segundo caso base: que el elemento central sea la solución del problema. En este caso no son necesarias llamadas recursivas y obviamente se resuelve de una manera muy sencilla.

Por supuesto, la versión que buscamos se obtiene cambiando el nombre *Resuelto* por el de la función. En cualquier caso es recomendable comprobar que nuestra función sigue las distintas reglas expuestar a lo largo de esta breve introducción a la recursividad. Concretamente siempre es interesante asegurarnos que los casos base son alcanzables (otra manera de decir que el caso general avanza hacia el caso base). Por suspuesto, si el vector tiene muchos elementos, el cortar por la mitad implica un avance, pero ?'Qué ocurre en los casos límite, es decir, cuando estamos cerca de llamar a los casos base?. Es bien sabido en ingeniería del software que el estudio de los casos límite son una prueba interesante para intentar localizar errores de programacíon (ver Pressman[6]). Es en estos casos donde sí se recomienda hacer una traza del funcionamiento de la recursivida aunque obviamente es mucho más simple pues sólo consideramos uno o, como mucho, dos pasos anteriores al caso base[1].

Supongamos que estamos en el paso anterior a la llamada del caso base,es decir, con dos posiciones del vector tal como muestra el caso *(a)* de la figura 1.

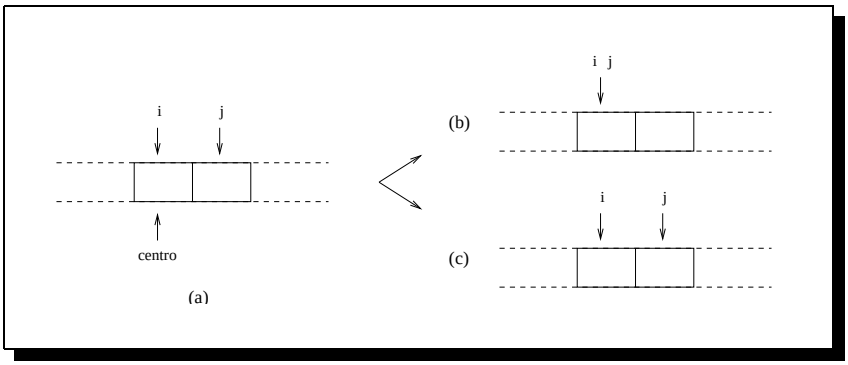

Figura 1: Efecto de la llamada recursiva cuando el problema se limita a dos elementos.

Si elemento es menor que el central se produce la situación del caso *(b)* y si es mayor la del caso *c*. Como podemos observar, la de este último no es válida pues no se ha avanzado hacia el caso base, es decir, la recursividad fallaría.

Una forma muy simple de mejorar el problema que tenemos sería incluir el caso de dos elementos como un nuevo caso base pues si comprobamos el comportamiento cuando hay más de dos seguro que no se daría ningún problema. Sin embargo notemos que cuando se hace una llamada recursiva ya se ha comprobado el caso del elemento central, por tanto parece más interesante solucionar el problema eliminando el elemento central de las llamadas recursivas. De esta manera garantizamos el avance hacia el caso base pues al menos eliminamos un elemento (el central). El nuevo algoritmo quedaría

```
int buscar (int v[], int i, int j, int x)
{
```

[1]Si se desea tener información más detallada acerca de la traza y funcionamiento interno de la recursividad se puede consultar Cortijo y otros[3]

```
  int centro;

  if (i==j)
    return (v[i]==x)?i:-1;
  else {
    centro=(i+j)/2;
    if (v[centro]==x)
      return centro;
    else if (v[centro]>x)
            return buscar (v,i,centro-1,x);
          else return buscar (v,centro+1,j,x);
}
```

De nuevo supongamos que estamos en el paso anterior a la llamada del caso base,es decir, con dos posiciones del vector tal como muestra el caso *(a)* de la figura 2.

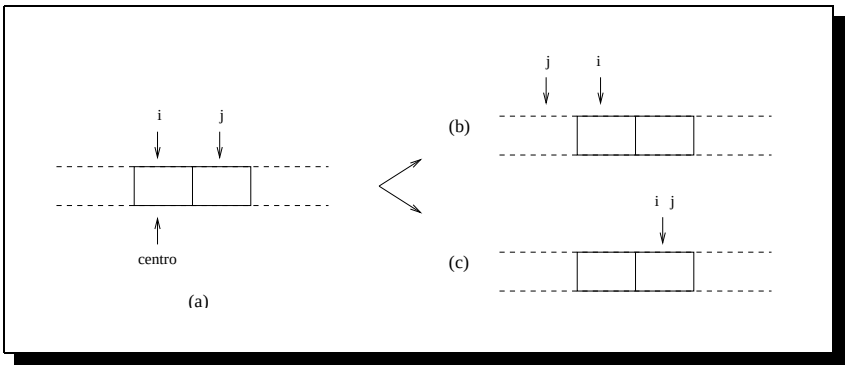

Figura 2: Efecto de la llamada recursiva cuando el problema se limita a dos elementos eliminando el elemento central en la llamada.

Ahora se produce un error en caso de que el elemento sea menor, es decir, en el caso *b*. Podemos solucionar el problema haciendo que en la primera llamada recursiva no se reste uno a la variable centro o añadiendo el caso base $i > j$ a la función. Si añadimos este caso, podemos comprobar que el caso $i == j$ no es necesario ya que también se reduce en la siguiente llamada al caso $i > j$. Por tanto podemos proponer la siguiente función como definitiva

```
int buscar (int v[], int i, int j, int x)
{
  int centro;

  if (i>j)
    return -1;
  else {
    centro=(i+j)/2;
    if (v[centro]==x)
      return centro;
    else if (v[centro]>x)
            return buscar (v,i,centro-1,x);
          else return buscar (v,centro+1,j,x);
}
```

2.1.2 Más búsqueda binaria.

Veamos otra forma de programar la búsqueda binaria que nos lleve a nuevos errores de manera que se pueda mostrar cómo sin necesidad de realizar la traza de las distintas llamadas recursivas podemos construir la función. Proponemos ahora la cabecera

```
int buscar (int *v, int n, int x)
```

que es válida para resolver nuestro problema. Los parámetros indican un vector, el tamaño del vector y el elemento a buscar. Una primera solución desarrollada en la misma línea que las versiones anteriores podría ser la siguiente

```
int buscar (int *v, int n, int x)
{
  int centro;

  if (n<=0)
    return -1;
  else {
    centro= n/2;
    if (v[centro]==x) return centro;
    else if (v[centro]>x) return buscar(v,centro,x);
         else return buscar(v+centro+1,n-centro-1,x);
  }
}
```

sobre la que podemos comprobar que los casos base están bien definidos y son alcanzables.

Si probamos esta función con el ejemplo que se muestra en la figura 3 el resultado obtenido no es correcto. Veamos sobre el código el comportamiento de la función.

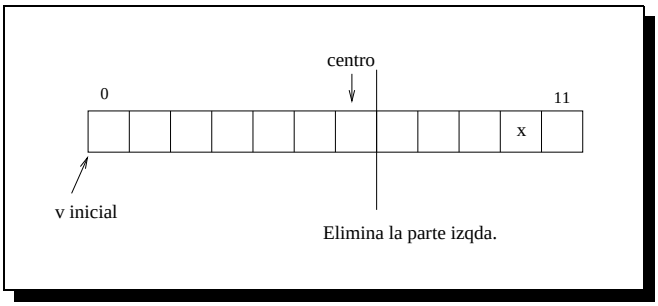

Figura 3: Representación de una llamada recursiva con el elemento a buscar en la mitad derecha.

El tamaño del vector es 12, por tanto se calcula el centro en la posición 6. Según el ejemplo, el elemento x está en la posición 10, por tanto la función llamará recursivamente en la mitad derecha del vector (última línea de la función). Nuestra cuestión es: supongamos que *la llamada a la función funciona*, ¿Qué devuelve la llamada inicial?.

Si la llamada recursiva funciona debería devolver el valor 3 pues es la posición del elemento x en el vector *v+centro+1*. Por tanto, la llamada inicial devolvería ese valor que, como vemos, no es correcto pues no tiene en cuenta los elementos que quedan a la izquierda, es decir, cuando el elemento se encuentra en la parte derecha la llamada inicial debería devolver la posición del elemento en el subvector de la derecha, sumándole las casillas a la izquierda (*centro+1*). Si modificamos la función obtenemos

```
int buscar (int *v, int n, int x)
{
  int centro;

  if (n<=0)
    return -1;
  else {
    centro= n/2;
    if (v[centro]==x) return centro;
    else if (v[centro]>x) return buscar(v,centro,x);
         else return centro+1+buscar(v+centro+1,n-centro-1,x);
  }
}
```

Como vemos, hemos revisado y localizado el error sin necesidad de realizar una traza de las distintas llamadas sino que hemos supuesto que la llamada recursiva funciona y sobre esa base se ha podido

localizar el error. Por tanto, si queremos revisar una función para comprobar que es correcta también es posible sacar partido del hecho de suponer que las llamadas recursivas funcionan, es decir, considerar que el problema está realmente **resuelto para tamaños menores que el original**, y evitar así el trazar mentalmente el comportamiento de éstas.

Precisamente, consideremos el caso de la figura 3 con la versión incorrecta de la función, es decir, sin la suma de los elementos a la izquierda. Efectivamente, hemos podido localizar el error considerando que teníamos resuelto el problema para vectores más pequeños, en concreto, hemos supuesto que devolvía el valor 3 y por tanto se podía resolver el problema con la mencionada suma. ¿Qué hubiera ocurrido si intentamos trazar las llamadas recursivas?. Si estudiamos bien el problema, nos damos cuenta de que la función devuelve el resultado de buscar el elemento en el subvector de la derecha, pero esta llamada a su vez, realiza otra llamada recursiva con un subvector que comienza precisamente en la posición 10, donde se encuentra el elemento. Por tanto, se devolverá el valor cero, que a su vez se devolverá a la primera llamada. Como resultado, la traza nos indica que el valor incorrecto obtenido no es 3, sino el valor cero. Un programador que intentara razonar con este valor, podría tener más dificultades para encontrar una solución, ya que si considera que devuelve un cero, no es tan trivial ver que una simple suma nos resuelve el problema.

Ahora bien, es interesante que el lector se interrogue acerca de este comportamiento. Dedíquese un momento a considerar la pregunta ¿Si devuelve cero, por qué va a funcionar con la suma de *centro+1* elementos? De nuevo insistimos en el razonamiento que debemos de considerar. En este caso, volvemos a intentar razonar sobre las llamadas recursivas, y es posible que nos cree confusión sobre la solución a nuestro problema. Por supuesto, un programador experimentado no tendrá ninguna duda sobre el problema, ya que la traza de las distintas llamadas nos demuestra que la solución efectivamente es correcta. La solución es la suma de 7 elementos más la solución devuelta de buscar el elemento en el subvector que comienza en la posición 7. Esta llamada no devuelve cero, ya que la función correcta devolverá este valor, pero de nuevo sumado con la parte izquierda del vector, en este caso, 3 elementos. De ahí que el valor 7 se sume al valor correcto 3, resultando finalmente como solución al problema el valor buscado 10. Como se puede observar, en el momento de construir una función recursiva, es conveniente considerar que la función para problemas más pequeños ya está resuelta con una función (la misma, pues es recursiva) que es correcta.

Por último, para terminar con este ejemplo tenemos que dar un detalle para completar la función ya que falla en caso de hacer una llamada en la parte derecha y no encontrar el elemento. Hemos considerado que funciona, y por tanto devolverá la posición del elemento en el subvector, pero sólo cuando realmente exista. Nos hemos olvidado de lo que ocurre cuando no existe. En este caso devolvería el valor -1 que tiene que ser pasado como resultado definitivo sin sumar la corrección de $centro + 1$. Por ejemplo, si consideramos el vector de la figura 3 sin que exista el elemento buscado, la llamada recursiva, si funciona, devolverá un valor -1 que sumado a $centro + 1$ (7) obtiene como posición del elemento 6.

La solución final de la función se obtiene modificando la última línea de manera que se llama a la búsqueda a la derecha, si devuelve un valor $r = -1$ el resultado final es éste y si no el resultado sería $r + centro + 1$. La solución final por tanto es

```
int buscar (int *v, int n, int x)
{
  int centro,r;

  if (n<=0)
    return -1;
  else {
    centro= n/2;
    if (v[centro]==x) return centro;
    else if (v[centro]>x) return buscar(v,centro,x);
          else {
            r= buscar(v+centro+1,n-centro-1,x);
            return (r==-1)-1:r+centro+1;
          }
}
```

Nótese la introducción de una variable local para evitar que se resuelva dos veces el subproblema de búsqueda en el subvector derecho.

3 Recursivo vs iterativo.

En muchos casos, la recursividad requiere más recursos (tiempo y memoria) para resolver el problema que una versión iterativa, por lo que, independientemente de la naturaleza del problema, la recursividad tiene su gran aplicación para problemas complejos cuya solución recursiva es más fácil de obtener, más estructurada y sencilla de mantener. Así por ejemplo, los problemas de cálculo de factorial o de búsqueda binaria son fáciles de programar iterativamente por lo generalmente no se utilizan las versiones recursivas:

- En el caso del factorial, no necesitamos más que un bucle para calcular el producto mientras que en la versión recursiva el número de llamadas que se anidarían es igual que el valor de entrada de manera que usar la versión recursiva es más un error que algo poco recomendable.
- Para la búsqueda binaria el máximo número de llamadas anidadas (produndidad de recursión) es del orden del logaritmo del tamaño del vector por lo que este factor es menos importante. A pesar de ello, y considerando que la versión iterativa es muy simple, también para este caso podemos rechazar el uso de la recursividad.

Incluso para casos más complejos pero correpondientes a funciones que se llaman muchas veces en un programa puede ser interesante eliminar la recursividad, ya sea de una manera sencilla como es el caso de la recursión de cola o de forma más compleja mediante la introducción de estructuras adicionales como *pilas* definidas por el programador (ver Aho y otros[1], Peña Marí[5]).

Además, cuando se resuelven problemas de manera recursiva hay que tener especial cuidado ya que el que la función tenga varias llamadas recursivas puede hacer que el número de subproblemas a resolver crezca de forma exponencial haciendo incluso que el resultado obtenido sea mucho más ineficiente. Por tanto, podemos plantear una última regla en la programación recursiva: *No resolver varias veces el mismo subproblema en llamadas recursivas distintas.* Para ilustrar esta idea véase más adelante el ejemplo de la sucesión de Fibonacci.

A pesar de estas consideraciones, existen multitud de problemas que se ven radicalmente simplificados mediante el uso de la recursividad y que cuya diferencia en recursos necesarios con respecto a la solución iterativa puede ser obviada a efectos prácticos.

4 Otros ejemplos.

El objetivo de esta sección es fijar los conceptos desarrollados mostrando algunos ejemplos prácticos. Por supuesto, el lector puede encontrar muchas más información en la literatura. Véanse por ejemplo otros algoritmos en Brassard[2], Cortijo y otros[3], Peña Marí[5], Sedgewick[7] o ejemplos más específicos sobre estructuras de datos más complejas en Fdez-Valdivia[4] y Weiss[8].

4.1 Sucesión de Fibonacci.

La sucesión de Fibonacci viene dada por

$$F(n) = \begin{cases} 1 & \text{si } n = 0 \text{ o } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases} \quad (2)$$

Como vemos es muy similar al ejemplo del cálculo del factorial. El obtener una solución no debiera ser ningún problema para el lector

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

El interés en esta función no es tanto la dificultad de implementarla sino la discusión sobre la conveniencia de una implementación recursiva. Como indicabamos anteriormente, era importante tener cuidado

de no resolver varias veces el mismo subproblema. En este caso podemos observar que para cada n hay que calcular *F(n-1)* y en segundo lugar *F(n-2)* que ya está incluido en el primero ya que si $n-1>1$

$$F(n-1) = F(n-2) + F(n-3) \tag{3}$$

por tanto el resultado será mucho más ineficiente al estar duplicando trabajo.

4.2 Torres de Hanoi.

El problema de las torres de Hanoi consiste en que inicialmente tenemos un conjunto de tres torres. La primera de ellas tiene apiladas n fichas de mayor a menor tamaño. El problema es pasar las n fichas de la primera a la tercera torre teniendo en cuenta que:

1. En cada paso sólo se puede mover una ficha.
2. Una ficha de un tamaño no puede apilarse sobre otra de menor tamaño.

La situación inicial en el caso de tener 4 fichas se muestra en la figura 4. El problema consiste en pasar las fichas de la torre 1 a la torre 3 teniendo en cuenta las dos restricciones anteriores.

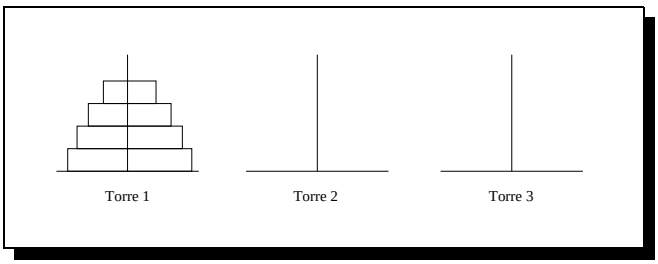

Figura 4: Situación inicial de las torres de Hanoi con 4 fichas.

Inicialmente el problema no parece trivial pues no es sencillo proponer un algoritmo iterativo que resuelva el problema. Ahora bien, si descomponemos el problema para que su solución se establezca en términos de la solución de un subproblema la situación es mucho más simple.

La "idea"recursiva que debemos considerar es que el problema de pasar 4 fichas de la torre 1 a la torre 3 se puede descomponer en pasar 3 fichas de la torre 1 a la torre 2, pasar la ficha más grande de la 1 a la 3 y despues volver a pasar las 3 fichas de la torre 2 a la torre 3. Como vemos, para resolver el problema de tama´ño 4 hemos de resolver dos subproblemas de tamaño 3.

En primer lugar tenemos que diseñar la cabecera de la función. Obviamente no basta con pasar como parámetro el número de fichas sino que además tendremos que especificar la torre inicial y final pues estos valores son diferentes en las llamadas a subproblemas. Por tanto la función tendrá la siguiente cabecera:

void Hanoi (**int** m, **int** i, **int** j)

que como vemos refleja los parámetros que queremos introducir y es válida para expresar una llamada a un subproblema (sólo tendremos que cambiar el número de fichas a uno menos y los número de las torres para usar la torre adicional). Es decir, vamos a diseñar una función *Hanoi* que tome como parámetros un entero (m que indica el número de fichas a pasar y dos enteros que indiquen la torre inicial y final (i,j). Como resultado la función escribirá en la salida estándar una línea por cada movimiento indicando la torre inicial y final de la que parte y a la que llega la ficha respectivamente.

En segundo lugar, es necesario determinar cuáles serán los casos base y casos generales. El caso base corresponde a un valor del número de fichas a pasar que sea bajo y por tanto sea un problema con fácil solución sin necesidad de hacer uso de la recursividad. Un buen ejemplo es hacer que el caso base corresponda al problema de traspasar una sola ficha de la torre i a la torre j. En esta caso, la solución es muy sencilla pues sólo es necesario escribir un mensaje diciendo que el movimiento es pasar la ficha de la torre i a la torre j.

El caso general corresponderá a tener más de una ficha. En este caso tenemos que resolver dos subproblemas de tamaño *m-1*. Insistamos de nuevo en la idea de que al programar no debemos de pensar en el funcionamiento interno de las llamadas recursivas sino que debemos de asumir que las llamadas

funcionan. Para ello, recurramos a suponer que existe una función *Resuelto (int m, int i, int j)* que escribe en la salida estándar todos los pasos que hay que dar para pasar m fichas de la torre i a la torre j.

Recordemos que el caso general consiste en escribir todos los pasos para llevar *m-1* fichas de la torre i a la torre intermedia, pasar una ficha de la torre i a la torre j y escribir todos los pasos para llevar *m-1* fichas de la torre intermedia a la torre j. La primera parte la resuelve la función *Resuelto*, la segunda es sólo escribir una línea con ese movimiento y que la tercera también la resuelve la función *Resuelto*. Nótese como la mayor dificultad del problema ya está resuelta en esta función ficticia que luego no será más que una llamada recursiva. Si unimos la solución del caso base y general podemos escribir

```
/* Resuelve el problema de pasar la torre "i" (1,2 ó 3) */
/* de una altura de "m" a la torre "j" haciendo uso de la */
/* torre 6-i-j                                            */

void Hanoi (int m, int i, int j)
{
  if (m==1)
    printf (``Ficha de %d a %d\n'',i,j);
  else {
    Resuelto (m-1,i,6-i-j);
    printf (``Ficha de %d a %d\n'',i,j);
    Resuelto (m-1,6-i-j,j);
  }
}
```

donde como vemos la torre adicional se obtiene como $6 - i - j$ como era de esperar. Efectivamente, el caso general expresa la "idea"recursiva que antes se indicaba. Si cambiamos la notación para referenciar las llamadas recursivas en lugar de la función ficticia obtenemos

```
void Hanoi (int m, int i, int j)
{
  if (m==1)
    printf (``Ficha de %d a %d\n'',i,j);
  else {
    Hanoi (m-1,i,6-i-j);
    printf (``Ficha de %d a %d\n'',i,j);
    Hanoi (m-1,6-i-j,j);
  }
}
```

Las distintas reglas que comentábamos a lo largo de este documento son válidas en este ejemplo. El caso general avanza hacia casos base y además estos son alcanzables. Si trazamos el comportamiento cuando el número de fichas es 2 el resultado de las distintas operaciones es correcto. Por lo tanto esta función es válida para resolver nuestro problema.

Por último, es interesante ver que el caso base podría haberse determinado como $m = 0$ obteniéndose una versión más simple de la función

```
void Hanoi (int m, int i, int j)
{
  if (m>0) {
    Hanoi (m-1,i,6-i-j);
    printf (``Ficha de %d a %d\n'',i,j);
    Hanoi (m-1,6-i-j,j);
  }
}
```

El lector puede comprobar que la función es válida y que obtiene la salida correcta al llamar al caso de $m = 1$.

4.3 Ordenación por selección.

Los algoritmos de ordenación pueden ser fácilmente considerados como algoritmos recursivos ya que en la mayoría de los casos se pueden expresar en términos de la ordenación de algún subvector del

problema original. Como caso ilustrativo consideremos la ordenación por selección aunque obviamente en la práctica deberemos usar un algoritmo de selección iterativo que será más rápido y menos costoso en recursos que el presentado aquí[2].

La "idea"recursiva que subyace en este algoritmo es que la ordenación de un vector de tamaño n es la selección del elemento más pequeño(m), el intercambio de éste con el elemento de la primera posición (A) y finalmente la ordenación del subvector de tamaño *n-1* comprendido entre los elementos 2 y n, ambos incluidos (ver figura 5).

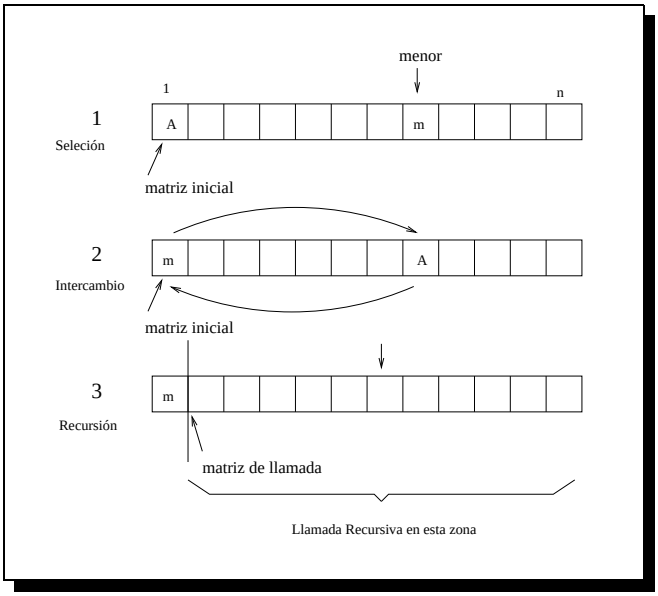

Figura 5: Algoritmo de ordenación por selección recursivo.

La cabecera de la función tendrá como parámetros un vector (un puntero al primer elemento) y el número de elementos del vector. Para especificar un subproblema de tamaño $n-1$ será necesario cambiar la matriz de llamada mediante la suma de 1 especificando a su vez un tamaño menor (ver figura 5) y código *C* más adelante).

En segundo lugar, es necesario determinar cuáles serán los casos base y casos generales. El caso base corresponde a un valor del número de elementos pequeño de forma que sea simple de solucionar sin necesidad de recursividad. En este caso podemos determinar $n = 1$ como el caso más simple. La solución para este tamaño es aún más simple: no hacer nada, puesto que el vector está ya ordenado. El caso general corresponderá a $n > 1$, es decir a tener 2 o más elementos. En este caso tendremos que llevar a cabo los tres pasos anteriormente expuestos:

1. Selección del menor valor. Mediante un bucle que recorre el vector seleccionamos el más pequeño de sus elementos (notemos la posición *minimo*.
2. Intercambio del elemento. El elemento en posición *minimo* se intercambia con el de posición 0.
3. Recursión. De forma recursiva se resuelve el problema de igual naturaleza pero de menor tamaño $(n-1)$ sobre el vector que comprende los $n-1$ elementos desde el de posición 1.

En lenguaje *C* se traduce en lo siguiente:

```
void SeleccionRecursivo (int matriz[], int n)
{
  register int i;
  int minimo;

  if (n>1) {
    minimo=0;
```

[2]Nótese que la profundidad de recursión para este algoritmo llega a ser del orden del número de elementos del vector

```
    for (i=1;i<n;i++)
      if (matriz[i]<matriz[minimo])
        minimo=i;
    i= matriz[0];
    matriz[0]= matriz[minimo];
    matriz[minimo]=i;
    SeleccionRecursivo (matriz+1,n-1);
  }
}
```

que claramente muestra que para casos límite tales como $n = 1$ o $n = 2$ tiene un funcionamiento correcto. Para un caso más general también obtiene el resultado deseado siempre que supongamos que la llamada recursiva funciona.

4.4 Ordenación por mezcla.

Por último, veamos un ejemplo de algoritmo de ordenación en el que puede estar más justificado el uso de la recursividad ya que su solución es bastante simple y muy clara con respecto a una posible solución iterativa.

El algoritmo lo podemos considerar una solución a la aplicación directa de la técnica de *divide y vencerás* (ver Brassard[2])a la ordenación de vectores. Básicamente la idea consiste en dividir el vector en dos partes, realizar la ordenación de esas dos partes y finalmente componerlas en una solución global. Es obvia la posible aplicación de la recursividad ya que cada una de esas dos partes se convierte en el mismo problema aunque de menor tamaño.

La composición de la solución global se realiza mediante mezcla ya que si tenemos un vector en el que cada una de las dos mitades que lo componen están ordenadas y pretendemos unirlas en una sola secuencia ordenada, la solución consistirá en ir recorriendo ambos subvectores desde los dos elementos más pequeños (desde el primer elemento al último) e ir añadiendo el más pequeño de los dos a la solución final. Para poder aplicar este algoritmo usaremos una matriz que llamaremos *auxiliar* en la que dispondremos los elementos ordenados de los dos subvectores y desde la que iremos añadiendo a la matriz principal donde quedarán los elementos ordenados.

En este algoritmo vamos a proponer una cabecera de función a la que se pasa la matriz a ordenar y los dos valores que delimitan los elementos a ordenar dentro de ella tal como hacíamos anteriormente para el algoritmo de búsqueda binaria. Por tanto sería

```
void OrdenMezcla (int matriz[], int izqda, int drcha)
```

Sin embargo, ya que necesitamos una matriz auxiliar para almacenar las mitades ordenadas pasaremos esa memoria como un parámetro adicional en lugar de buscarla en cada una de las llamadas recursivas. De esta forma, al algoritmo pasamos una matriz a ordenar y otra matriz de igual tamaño donde puede almacenar temporalmente los resultados intermedios quedándonos la siguiente cabecera

```
void OrdenMezcla (int matriz[], int izqda, int drcha, int auxiliar[])
```

Por otro lado, es necesario determinar los casos base y general. Al igual que en el apartado anterior consideramos el caso de ordenar un sólo elemento como caso base donde no hay que realizar ninguna operación. El caso general tendrá que implementar:

1. Ordenación de las dos mitades.
2. Mezcla de las dos mitades.

La primera de estas etapas corresponde a resolver el mismo problema pero de menor tamaño y la segunda no es más que una operación de mucho menor complejidad en la que únicamente tenemos que ir reescribiendo los elementos resultantes de forma ordenada. Sin más discusión, la solución definitiva es:

```
void OrdenMezcla (int matriz[], int izqda, int drcha, int auxiliar[])
{
  int i,j,k,centro;
```

```
  if (drcha-izqda>0) {
    centro=(izqda+drcha)/2;
    OrdenMezcla (matriz,izqda,centro,auxiliar);
    OrdenMezcla (matriz,centro+1,drcha,auxiliar);
    for (i=centro; i>=izqda; i--) auxiliar[i]=matriz[i];
    for (j=centro+1; j<=drcha; j++) auxiliar[drcha+centro+1-j]= matriz[j];
    i=izqda;
    j=drcha;
    for (k=izqda; k<=drcha; k++)
    if (auxiliar[i]<auxiliar[j]) {
      matriz[k]=auxiliar[i];
      i=i+1;
    }
    else {
      matriz[k]=auxiliar[j];
      j=j-1;
    }
  }
}
```

Dejamos como ejercicio para el lector el justificar:

1. No usamos una matriz *auxiliar* declarada como variable local con un tamaño suficientemente grande como para abarcar todos los elementos de la matriz a ordenar.
2. Los elementos de las dos mitades se disponen en el vector auxiliar de manera ascendente en el primer caso y descendente en el segundo.

Para más detalles sobre este algoritmo se puede consultar por ejemplo Brassard[2] y Sedgewick[7].

Referencias

[1] Aho, Hopcroft y Ullman. *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.

[2] Brassard y Bratley. *Fundamentos de algoritmia*. Prentice Hall, 1997.

[3] Cortijo, Cubero Talavera y Pons Capote. *Metodología de la programación*. Proyecto Sur. 1993.

[4] Fdez-Valdivia, Garrido y Gcia-Silvente. *Estructuras de datos. Un enfoque práctico usando C*. Los autores. 1998.

[5] Peña Marí, *Diseño de Programas, Formalismo y Abstracción*. Prentice Hall, 1997.

[6] Pressman, *Ingeniería del software. Un enfoque práctico*. MacGraw Hill, 1993.

[7] Sedgewick, *Algorithms*, Addison Wesley, 1989.

[8] Weiss, *Data Structures and algorithm analysis in C++*. Benjamin Cummings, 1992.