2 Recursividad

2.1 Concepto de recursividad

Se dice que una función es recursiva cuando dicha función se define en términos de la misma función. Es importante recordar que no todas la funciones pueden llamarse a si mismas, deben estar diseñadas especialmente para comportarse de manera recursiva, de otro modo dichas funciones podrían conducir a bucles infinitos, o a que el programa termine inadecuadamente.

Por ejemplo, podríamos pensar en la creación de un algoritmo que produzca el resultado de un numero factorial, la definición iterativa de dicho algoritmo seria la siguiente:

```
int factorial (int n)
{
      prod = 1;
      for ( x = n; x > 0; x-- )
            prod *= x;
      return prod;
}
```

La definición iterativa es sencilla, pero no resulta tan intuitiva, en cambio, uno podría analizar mas detalladamente el problema y generar una definición recursiva para una función factorial que podría resultar mas intuitiva y sencilla de implementar:

```
n! = n * (n-1)!
!0 = 1
```

Podemos observar en dicha definición que el factorial de un numero es el producto de dicho numero por el factorial del numero entero menor próximo y así consecutivamente, además podemos ver que el factorial de el numero cero es 1, lo que nos indica un caso de base o un caso de parada que ya no requiere mas llamadas consecutivas. A partir de este caso base se realiza un retroceso en las llamadas que obtiene como resultado el producto del caso base por cada uno de los valores n del numero en las llamadas.

```
1     3! = 3 * 2!
2                 2! = 2 * 1!
3                             1! = 1 * 0!
4                                         0! = 1
3´                            1! = 1 * 1
2´                2! = 2 * 1
1´    3! = 3 * 2
→     6
```

Podemos observar que cada caso es reducido a un caso simple hasta que se alcanza el caso base de 0!, el cual es definido directamente como 1. En la línea 4 obtenemos un valor que esta definido directamente y no se realiza el factorial de ningún otro numero. De esta manera podemos realizar un retro seguimiento (***backtrack***) de la línea 4 a la 1, regresando el valor calculado en cada línea evaluando el resultado de las líneas previas. Analizando lo anterior concluimos que en una definición recursiva las instancias complejas de un proceso se define en términos de instancias mas simples, estando éstas definidas en forma explicita, facilitando el diseño de la función.

2.1.1 Propiedades de los algoritmos recursivos

La idea básica de la recursividad es definir una función en términos de si misma y reducirla hasta alcanzar un caso único (*caso base*) en el que los términos no involucran a la función, y así permita

detener las llamadas recursivas y recolectar el valor en cada retroceso auxiliándose de una pila para conservar los valores indispensables. En el caso del factorial los números n fueron almacenados en una pila en cada llamada.

Sin una salida no recursiva, la función recursiva no podría ser calculada. Alguna instancia del la definición recursiva debe eventualmente reducirse a la manipulación de uno o mas casos simples no recursivos.

2.2 Funcionamiento interno de la recursividad

Cuando se llama a una función recursiva, se crea un nuevo juego de variables locales, de este modo, si la función hace una llamada a si misma, se guardan sus variables y parámetros en una pila de datos, y la nueva instancia de la función trabajará con su propia copia de las variables locales, cuando esta segunda instancia de la función retorna, recupera las variables y los parámetros de la pila y continua la ejecución en el punto en que había sido llamada.
La mayoría de los lenguajes de alto nivel como C, C++ o Java permiten la construcción de subrutinas, funciones que pueden ser llamadas a si mismas (recursivas) de manera natural. A continuación se muestra una función recursiva que calcula el factorial de un numero n:

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

Veamos paso a paso que es lo que ocurre cuando realizamos una llamada a dicha función con un valor de n igual a 3, factorial(3):

```
llamada 1   n = 3       n != 0      n * factorial(2)
llamada 2   n = 2       n != 0      n * factorial(1)
llamada 3   n = 1       n != 0      n * factorial(0)
llamada 4   n = 0       n == 0      return 1 → 1(regresa a 3´)
llamada 3´  n = 1                   return n * 1 → 1 (regresa a 2´)
llamada 2´  n = 2                   return n * 1 → 2 (regresa a 1´)
llamada 1´  n = 3                   return n * 2 → 6 (termina)
```

Podemos observar que en cada llamada de función el valor de n es conservado en cada llamada a función, en realidad el valor de n es colocado en una pila y posteriormente es recuperado cada vez que regresa a la llamada de cada función factorial.

2.2.1 Ventajas y desventajas de la recursividad

Las funciones recursivas son mecanismo muy eficientes de programación pues facilitan el diseño de las mismas, sin embargo la recursividad no es en todos los casos un buen modo de resolver problemas, ya que existen casos en los que un algoritmo iterativo resolvería de manera bastante adecuada un problema determinado.

La recursividad consume recursos adicionales de memoria y tiempo de ejecución, y se debe aplicar a funciones que realmente obtengan beneficio directo. Para el caso de la función factorial parece ser una buena medida el empleo de recursividad, sin embargo analizaremos ahora el caso de una función que genera una secuencia de números a la que se le conoce como serie de fibonacci.

La serie de fibonacci es una secuencia de números enteros en donde cada elemento de la secuencia es la suma de los dos elementos que la preceden:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
```

El calculo de un numero para la secuencia de fibonacci puede ser definida de la siguiente manera recursiva:

```
fib(n) = n si n == 0 o n == 1
fib(n) = fib(n-2) + fib(n-1) si n >= 2
```

Para calcular *fib(4)*, por ejemplo, podríamos aplicar la siguiente definición recursiva para obtenerla:

```
fib(4)=     fib(2)            +     fib(3)
            fib(0)+ fib(1)    +     fib(1)+ fib(2)
            0     + 1         +     1     + fib(0) +  fib(1)
            0     + 1         +     1     + 0      +  1     = 3
```

Podemos observar que dentro de cada definición recursiva es necesario realizar nuevamente un llamado recursivo doble, y esto claramente es un inconveniente innecesario, pues es superfluo ocupar demasiados recursos computacionales para poder realizar el calculo, en este caso un algoritmo iterativo realizaría la misma operación, pero de manera mas eficiente.

```
int fibonacci( int n ) {
      if (n <= 1)
            return n;
      anterior = 0;
      actual = 1;
      for (i = 2; i <= n; i ++) {
            x = anterior
            anterior = actual;
            actual = x + anterior
      }
      return actual;
}
```

Es claro darse cuenta que un algoritmo iterativo para el caso de la serie de fibonacci resulta un método menos costoso por la cantidad de recursos computacionales que se necesitan emplear.

2.2.2 Ejemplos de algoritmos recursivos

A continuación se muestran algunos ejemplos de problemas cotidianos que pueden ser fácilmente replanteados en términos de recursividad, en primer lugar trataremos algunos ejemplos de operaciones recursivas con vectores y al final un ejemplo de diseño de permutaciones recursivas.

Búsqueda Binaria : Dentro de los métodos de búsqueda de elementos, la búsqueda binaria es uno de los mas apropiados cuando hablamos de datos que ya se encuentran ordenados. Se basa en el fundamento de divide y vencerás para localizar el elemento deseado. Es un proceso muy similar al que una persona utiliza para buscar información en un entorno ordenado, por ejemplo en una agenda, si uno abre la agenda en la parte central, por ejemplo en la M, y la letra a buscar es la J, pues únicamente se desplaza al inicio de la agenda que es la dirección correcta.

El algoritmo de búsqueda binaria comienza en el elemento central de la lista. Si el elemento central no es buscado, entonces se repite la búsqueda, centrándose en la primera o segunda mitad, dependiendo de que el elemento buscado sea más pequeño o más grande que el valor central.

La idea para implantar este mismo mecanismo en un proceso recursivo nos lleva a la necesidad de pensar una función que sea capas de recibir el arreglo en el que estamos buscando, el dato a buscar y los limites de cada sub-arreglo que vamos generando tras el proceso de búsqueda.

El resultado es el siguiente algoritmo recursivo, una función llamada *binsearch* que es capas de recibir como argumentos los datos antes mencionados, tras cada llamada se va seleccionando un sub-arreglo dentro del arreglo, donde se localizará el elemento:

```
int binsearch(int x, int* array, int bajo, int alto)
{
      int mid = 0;
   if ( bajo > alto )
      return -1;
   mid = (int)(bajo + alto)/2; // rendea al menor
   if ( x == array[mid])
      return mid;
   else if ( x < array[mid] )
      binsearch(x, array, bajo, mid-1);
   else
      binsearch(x, array, mid+1, alto);
}
```

Suma de elementos de un arreglo: La idea de este algoritmo es realizar la suma de todos los elementos de un arreglo de manera recursiva, para ello es necesario encontrar el primer elemento del arreglo y a partir de él de manera recursiva hacemos la recuperación de cada casilla del arreglo de manera que vamos sumando cada elemento del arreglo hasta llegar al ultimo elemento del arreglo representado por *n*.

```
int suma(int* array, int n)
{
   if (n == 0)
      return array[0];
   else
      return array[n] + suma(array, n-1);
}
```

Máximo elemento en un arreglo: La idea básica en este algoritmo es encontrar el número de mayor valor dentro de un arreglo de elementos, para lograrlo recorreremos todo el arreglo de fin a inicio de manera recursiva y en cada retroceso compararemos el valor actual de la llamada con el resultado anterior a ella:

```
int maximo(int* array, int n)
{
   if (n == 0)
      return array[0];
   else {
      int y = maximo(array, n-1);
      return ( array[n] > y ? array [n]: y );
   }
}
```

Permutaciones: El siguiente algoritmo resulta muy interesante pues se trata de construir permutaciones de elementos de manera recursiva. Las permutaciones de un conjunto son las diferentes maneras de colocar sus elementos, usando todos ellos y sin repetir ninguno.

Por ejemplo para A, B, C, tenemos: { ABC, ACB, BAC, BCA, CAB, CBA } como el conjunto de permutaciones posibles.

```
/* Prototipo de función */
void Permutaciones(char *, int l=0);

int main(int argc, char *argv[]) {
   char palabra[] = "ABCDE";

   Permutaciones(palabra);

   cin.get();
   return 0;
}

void Permutaciones(char * cad, int l) {
   char c;    /* variable auxiliar para intercambio */
   int i, j;  /* variables para bucles */
   int n = strlen(cad);

   for(i = 0; i < n-l; i++) {
      if(n-l > 2) Permutaciones(cad, l+1);
      else cout << cad << ", ";
      /* Intercambio de posiciones */
      c = cad[l];
      cad[l] = cad[l+i+1];
      cad[l+i+1] = c;
      if(l+i == n-1) {
         for(j = l; j < n; j++) cad[j] = cad[j+1];
         cad[n] = 0;
      }
   }
}
```

La mayoría de los procesos recursivos bien diseñados suelen ser muy eficientes y en contadas ocasiones fallan, pero es importante recordar no todos los procesos recursivos son eficientes por naturaleza propia.

Bibliografía

Y. Langsam, M. J. Augenstein, A. Tenenbaum. *Data Structures using C and C++*. Prentice Hall, Second edition. ISBN 0-13-036997-7.