GUIA DE RECURSIVIDAD V2.2

Prof. Rhadamés Carmona
Última revisión: 28 de Noviembre de 2013

Recursividad

Consiste en la definición de algo (una propiedad, una operación, un procedimiento o una función) en términos de sí mismo.

Funciones como el Factorial y Fibonacci se pueden expresar en forma recursiva.

Fact(0) = 1
Fact(n) = n*Fact(n-1)

Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1)+Fib(n-2)

Algoritmos Recursivos

Un algoritmo se dice que es **recursivo** cuando *tiene la capacidad de llamarse o invocarse a sí mismo o de llamar a otro algoritmo desde el cual se puede volver a invocar al primero.* En cualquier caso, se garantiza que el número de llamadas recursivas que se realizan es *finito* (es decir, en algún momento DEBE finalizar la ejecución del algoritmo). Para ello, cada llamada recursiva resuelve un problema de menor tamaño, uno tras otro, hasta llegar a un problema cuya resolución sea directa o conocida.

La técnica que se aplica para definir la solución recursiva de problemas se conoce como *divide y vencerás* y se trata de resolver un problema mediante su descomposición en varios subproblemas similares al original (o del mismo tipo) pero con datos más pequeños. Si un problema puede subdividirse en subproblemas similares, entonces el mismo proceso puede ser aplicado a cada uno de los subproblemas hasta que se pueda encontrar una solución directa o conocida. Finalmente se combinan las soluciones de los subproblemas para producir la solución final del problema original.

Elementos de un algoritmo recursivo

• ***Los casos base:*** Son los casos del problema que se resuelven con un segmento de código sin recursividad. Normalmente corresponden a instancias del problema simples y fáciles de implementar cuya solución es conocida. Ejemplo, Factorial de 0.

• ***Casos recursivos:*** son casos que se resuelven mediante invocaciones a sí mismo, y por lo general, reduciendo el problema de dimensión para que se aproxime cada vez más a un caso base.

• ***Los parámetros:*** Toda acción o función recursiva tiene en general al menos un parámetro, normalmente pasado por valor, que debe cambiar en cada llamada recursiva hasta aproximarse o converger al caso base. El resultado puede ser retornado, o bien dejado en un parámetro por referencia.

Ejemplos:

```
Function Factorial(Integer n) : Integer
      If (n <= 0) then
            Return 1;
      Else
```

```
            Return n * Factorial(n-1);
      EndIf
EndFunction
```

```
// esta es la versión que requiere log2(n) pasos
Function Potencia(Real x, Integer y) : Real
      Real aux;
      If (y == 0) then
            Return 1;
      Else
            aux = Potencia(x, y div 2);
            aux = aux * aux;
            If (y mod 2 ≠ 0) then
                  aux = x * aux;
            EndIf
            Return aux;
      EndIf
EndFunction
```

```
// esta es la versión fuerza bruta que requiere n pasos
Function Potencia(Real x, Integer y) : Integer
      If (y == 0) then
            Return 1;
      Else
            Return x*Potencia(x,y-1);
      EndIf
EndFunction
```

Cual algoritmo de potencia creen ustedes que es más eficiente??. Como otro ejemplo tenemos el máximo común divisor, el cual tiene dos implementaciones; una basada en restas sucesivas, y el otro en el operador **MOD**.

```
// hipótesis: n,m>0. Devuelve el máximo entero que divide n y m
Function MCD(Integer n,m) : Integer
        Integer r;
        Select
        n=m: return n;
        n>m: return MCD(n-m, m);
        n<m: return MCD(n, m-n);
        EndSelect
EndFunction
```

```
// Algoritmo de Euclides
Function MCD(Integer n,m) : Integer
        Select
        m=0: return n;
        m>0: return MCD(y, x mod y);
        EndSelect
EndFunction
```

Ejecución

Cuando se invoca a una función o acción, se almacena el ambiente local en el tope de la pila del programa. En este ambiente se tienen los parámetros y demás variables locales. La función o acción podrá acceder a estas variables, así como el resto de su ambiente de referenciación (atributos del objeto al cual la función o acción pertenece, variables globales, y demás variables alcanzables según las reglas de alcance).

Si una función recursiva contiene variables locales y/o parámetros, se creará un nivel en la pila diferente por cada llamada. Los nombres de las variables locales serán los mismos, pero en cada nivel recursivo son un grupo nuevo de variables, que tienen posiciones de memoria distintas (están en distintos ambientes de referenciación). Las variables locales que pueden acceder a la función o acción recursiva son las definidas en su propio ambiente.

Así como se apilan las variables, se guarda la posición de la última instrucción ejecutada, de manera tal que, cuando la función termina, se desapila todo el ambiente local, y se devuelve el control al punto en donde se hizo la invocación. Los resultados pueden ser retornados, o dejados en variables pasadas por referencia, en atributos, o en variables globales.

Como ejercicio, hacer una traza del Factorial o la Potencia.

Tipos de algoritmos recursivos

Se puede clasificar en algoritmos de recursión directa (simple o lineal, múltiple y anidada), y algoritmos de recursión indirecta. Los algoritmos de recursión directa se llaman a sí mismos en su definición. Los algoritmos de recursión indirecta invocan a otro algoritmo que llama al primero.

a) Recursividad lineal: Aquella en cuya definición sólo aparece una llamada recursiva. Se puede transformar con facilidad en algoritmos iterativos. Como ejemplos tenemos, Factorial, Potencia, Búsqueda Binaria. La recursión lineal puede subdividirse en recursión lineal final, y recursión lineal no final. Se llama final cuando la llamada recursiva es la última operación que se efectúa, devolviendo el resultado que arroja la llamada recursiva. Ejemplo, el algoritmo de máximo común divisor o MCD. Se llama "no final" cuando requiere de combinar el resultado de la llamada recursiva antes de retornar. Ejemplo, factorial.

Típicamente los algoritmos de recursividad lineal no final tienen el siguiente esquema:

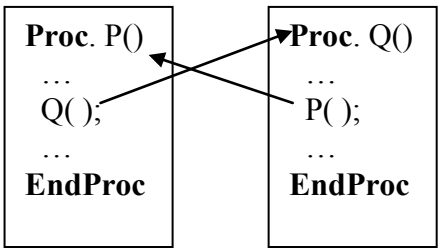
```
Function F(<tipo argumento> x) : <tipo_retorno>
        If Condicion(x) Then
                Return Algun_Computo(x); // solución conocida
        Else
                Aux = F(small(x)):  // llamada recursiva con datos más pequeños
                Return Combinar(x, Aux);
        EndIf
EndFunction
```

b) Recursividad múltiple: Se da cuando hay más de una llamada a sí misma dentro del cuerpo de la función, resultando más difícil de representar de forma iterativa. Ejemplo, Fibonacci en su forma recursiva original, torres de Hanoi, MergeSort, QuickSort.

c) Recursividad anidada: En algunos de los argumentos de la llamada recursiva hay una nueva llamada a sí misma. Ejemplo: Función de Ackermann, que es usada como benckmark para compiladores que optimizan código recursivo.

A(0 , n) = n +1 **(caso base)**
A(m , 0) = A(m – 1 , 1)
A(m , n) = A(m – 1 , A(m , n-1))

d) Recursividad cruzada o indirecta: Son algoritmos donde una función/acción provoca una llamada a sí misma de forma indirecta, a través de otras funciones/acciones.



Ejemplos de problemas divide y conquista, resueltos con recursión

Los problemas divide y conquista por lo general particionan el problema en varios subproblemas similares más pequeños, y tienen un costo adicional para combinar las soluciones parciales.

El máximo de N números puede calcularse en dos etapas. Primero se halla el máximo de los N/2 primeros números, luego se halla el máximo de los N/2 siguientes, y finalmente el costo de combinar ambas soluciones es simplemente obtener el mayor de ambos resultados parciales. Así, un problema de tamaño N se divide en dos problemas de tamaño N/2 con un costo adicional de combinar ambas soluciones parciales.

El binary search reduce en cada paso un problema de tamaño n a tamaño n/2. El MergeSort particiona un conjunto en dos, con un costo de combinación igual a la mezcla ordenada de ambos conjuntos. El QuickSort particiona el conjunto en 3, con un costo adicional de particionamiento y de concatenación.

Búsqueda Binaria:

```
// retorna la posición donde encontró el elemento, o -1 si no lo encuentra
// se invoca de esta manera: pos = BSearch(A, x, 1, N);
function BSearch(Ref Array A[1..N] of Integer; Integer x, Li, Ls):
Integer
   Interger c;
   If (Ls < Li)
      Return -1;       // significa no encontrado
   Else
      c = Li + (Ls-Li) div 2;
      Select
      A[c]==x: return c;       // posición del elemento
      A[c]>x : return BSearch(A, x, Li, c-1);
      A[x]<x : return BSearch(A, x, c+1, Ls);
      EndSelect
   EndIf
EndFunction
```

Ordenamiento por mezclas:

```
Procedure MergeSort(Ref Array A[1..N] of Integer, Integer Li, Ls)
      If (Ls>Li) Then
```

```
            Integer c = Li + (Ls-Li) div 2;
            MergeSort(A, Li, c);
            MergeSort(A, c + 1, Ls);
            // mezcla ordenada de A[Li..c] con A[c+1..Ls], y lo deja en A[Li..Ls]
            Mezclar(A, Li, c, Ls);
      EndIf
EndProc

Procedure Mezclar(Ref Array A[1..N] of Integer, Integer Li, c, Ls)
      Array Aux[1..Ls-Li+1] of Integer;
      Integer i,j,k;
      j = Li;     // índice del primer subarreglo
      k = c+1;    // índice del Segundo subarreglo
      For i=1 To Ls-Li+1 Do
            Select
            j == c+1:   // sub arreglo A[Li..c] ya tratado
                        Aux[i] = A[k]; k = k+1;
            k == Ls+1:  // sub arreglo A[Li+c+1..Ls] ya tratado
                        Aux[i] = A[j]; j=j+1;
            j<= c and k <= Ls: // no se han terminado los subarreglos
                        // tomamos el más pequeño
                        If (A[j] > A[k]) Then
                              Aux[i] = A[k]; k=k+1;
                        Else
                              Aux[i] = A[j]; j=j+1;
                        EndIf
            EndSelect
      EndFor
      // copiando Aux en el subarreglo A[Li..Ls]
      For i=Li To Ls Do
            A[i] = Aux[i-Li+1];
      EndFor
EndProcedure
```

Ordenamiento Rápido:

```
Procedure QuickSort(Ref Array A[1..*] of Integer, Integer N)
       If (N > 1) Then
              Integer S1, S2, S3;
              Array L1,L2,L3[1..N] of Integer;
              x = A[1];              //el 1ro. u otro elemento
              Menores(L1, A, x, S1); //L1 = { y en A : y < x }
              Iguales(L2, A, x, S2); //L2 = { y en A : y = x }
              Mayores(L3, A, x, S3); //L3 = { y en A : y > x }
              // Si es el tamaño del arreglo Li
              QuickSort(L1, S1);
              QuickSort(L3, S3);
              Concatenar(A,L1,L2,L3,N,S1,S2,S3);  // A=L1+L2+L3
      EndIf
EndProcedure

Procedure Concatenar(Ref Array Result[1..*] of Integer,
      Array X,Y,Z[1..*] of Integer, Integer Sr, Sx, Sy, Sz)

      // concatena los arreglos X,Y,Z de tamaños Sx,Sy y Sz resp.
      // dejando el resultado en el arreglo Result de tamaño Sr

      Integer I;
      For I=1 To Sx Do
            Result[I] = X[I];
      EndFor
```

```
      For I=1 To Sy Do
            Result[Sx+I] = Y[I];
      EndFor
      For I=1 To Sz Do
            Result[Sx+Sy+I] = Z[I];
      EndFor

EndProcedure
```

Nota: esta implementación de QuickSort es comúnmente mejorada evitando utilizar arreglos auxiliares. Para mayor información, visitar wikipedia.

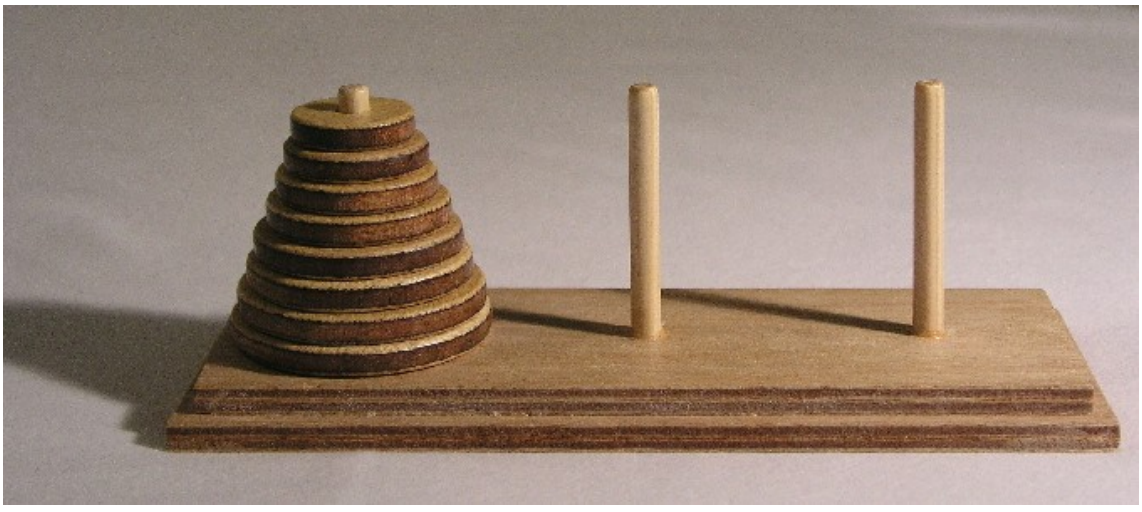
Ejemplo, en C++ utilizando plantillas:

```
template <class T>
void quicksort(vector<T>& v, int iz, int der)
{
  if (iz < der)
  {
    int iz_aux(iz), der_aux(der);
    T aux = v[iz]; // pivote
    while (iz_aux != der_aux)
    {
      while (v[iz_aux] < aux && der_aux > iz_aux)  ++iz_aux;
      while (aux < v[der_aux] && iz_aux < der_aux) --der_aux;
      swap(v[iz_aux], v[der_aux]);
    }
    swap(v[iz], v[der_aux]);

    quicksort(v, iz, iz_aux - 1);
    quicksort(v, iz_aux + 1, der);
  }
}


template <class T>
void ordena(vector<T>& v)
{
    quicksort(v, 0, v.size() - 1);
}
```

Torres de Hanoi:

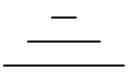


Es éste un clásico de los juegos de estrategia. Se parte de tres estacas, en la primera de las cuales hay n discos de diámetros diferentes ensartados formando una torre. Se trata de llevar los n discos a la tercera estaca, conservando la formación de torre. Los movimientos válidos consisten en llevar el disco superior de una estaca a cualquier otra (libre o con otros discos), de modo que no quede encima de un disco de diámetro menor.

Se puede demostrar que la cantidad mínima de movimientos es 2^n -1.

```
Procedure Hanoi(Integer n; Estaca origen, destino, auxiliar)
  If(n <= 1) Then
    moverdisco(origen,destino);
  Else
    Hanoi (n-1,origen,auxiliar,destino);
    moverdisco(origen,destino);
    Hanoi (n-1,auxiliar,destino,origen);
  EndIf
EndProc
```

Hagamos la corrida del algoritmo sólo en el primer nivel de recursión para entender el funcionamiento.



Origen Destino Auxiliar

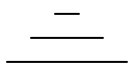
Hanoi (n-1,origen,auxiliar,destino): esto debe haber colocado los n-1 discos recursivamente del origen a la pila auxiliar, usando el destino como pila intermedia

Origen Destino Auxiliar

moverdisco(origen,destino): sencillamente, queda un disco en el origen, y se mueve directo al destino. Ya este disco quedó en la posición donde debe estar, y no será movido. Además, no importa que otro disco se le ponga encima, pues este es el más grande entre todos los discos restantes.

Origen Destino Auxiliar

Hanoi (n-1,auxiliar,destino,origen): finalmente, deben moverse recursivamente los discos de la pila auxiliar, al destino, para obtener el resultado final.



Origen Destino Auxiliar

A la final, la solución es divide y conquista. Mover n discos del origen al destino equivale a mover n-1 discos a la pila auxiliar, luego mover el disco restante a la pila destino, y finalmente mover los n-1 discos de la pila auxiliar a la pila destino.