Tema 3. Complejidad de algoritmos recursivos

1. INTRODUCCIÓN 1
CLASIFICACIÓN DE FUNCIONES RECURSIVAS 1
DISEÑO DE FUNCIONES RECURSIVAS 2
2. VENTAJAS E INCONVENIENTES DE LA RECURSIVIDAD 4
3. CÁLCULO DE LA COMPLEJIDAD 7
4. PROBLEMAS 14

Tema 3. Complejidad de algoritmos recursivos

1. Introducción

Recursividad: técnica con la que un problema se resuelve sustituyéndolo por otro problema de la misma forma pero más simple.

Ejemplo: Definición de factorial para n >= 0.

0! = 1

n! = n * (n-1)! si n>0

Es una herramienta muy potente, útil y sencilla.

Ciertos problemas se adaptan de manera natural a soluciones recursivas.

Clasificación de funciones recursivas

- Según desde dónde se haga la llamada recursiva:
- **Recursividad directa**: la función se llama a sí misma.
- **Recursividad indirecta**: la función A llama a la función B, y ésta llama a A.

- Según el número de llamadas recursivas generadas en tiempo de ejecución:
- Función recursiva **lineal** o **simple**: se genera una única llamada interna.
- Función recursiva **no lineal** o **múltiple**: se generan dos o más llamadas internas.

- Según el punto donde se realice la llamada recursiva, las funciones recursivas pueden ser:
  - **Final**: *(Tail recursion):* La llamada recursiva es la última instrucción que se produce dentro de la función.
  - **No final**: *(Nontail recursive function)*: Se hace alguna operación al volver de la llamada recursiva.

Las funciones recursivas finales suelen ser más eficientes (en la constante multiplicativa en cuanto al tiempo, y sobre todo en cuanto al espacio de memoria) que las no finales. (Algunos compiladores pueden optimizar automáticamente estas funciones pasándolas a iterativas.)

Ejemplo de función recursiva final: algoritmo de Euclides para calcular el máximo común divisor de dos números enteros positivos.

$$\text{mcd}(a, b) = \text{mcd}(b, a) = \begin{cases} \text{mcd}(a-b, b) & \text{si } a > b \\ \text{mcd}(a, b-a) & \text{si } a < b \\ a & \text{si } a = b \end{cases}$$

Diseño de funciones recursivas

- El problema original se puede transformar en otro problema similar 'más simple'.
- Tenemos alguna manera directa de solucionar 'problemas triviales'.

Ejemplo: cálculo del factorial de un número entero no negativo.

Para que un módulo recursivo sea correcto:

1. **Análisis por casos del problema:** Existe al menos una **condición de terminación** en la cual no es necesaria una llamada recursiva. Son los **casos triviales** que se solucionan directamente.

   Si n = 0 ó n = 1, el factorial es 1.

2. **Convergencia de las llamadas recursivas**: Cada llamada recursiva se realiza con un '**dato más pequeño**', de forma que se llegue a la condición de terminación.

   Factorial (n) = n * Factorial (n-1)

   En la llamada, n va haciéndose más pequeño, y en algún momento llegará a 0 ó 1.

3. Si las llamadas recursivas funcionan bien, el módulo completo funciona bien: **Principio de inducción.**

   Factorial(0) = 1

   Factorial(1) = 1

   Para n > 1, si suponemos correcto el cálculo del factorial de (n –1),

   *Factorial (n) = n * Factorial (n-1)*

Ejemplo: las torres de Hanoi.

Ejemplo *(no correcto)*: escribir en orden inverso los elementos de una lista.

```
void EscribirRevés (Lista L) {

   DatosLista x;

   if (!L.VaciaLista()) {
        L.Resto(L);
        EscribirRevés(L);
        L.Primero(x);
        cout << x << endl;
   }
}
```

2. Ventajas e inconvenientes de la recursividad

- Solución de problemas de manera natural, sencilla, comprensible y elegante.

Ejemplo: dado un número entero no negativo, escribir su codificación en binario.

```
void pasarbinario (int N){

    if (N < 2) {
            cout << N;
    }
    else {
             pasarbinario(N/2);
             cout << N%2;
    }
}
```

- Facilidad para comprobar y verificar que la solución es correcta. **Inducción matemática.**

```
boolean Pertenece (Lista L, DatosLista x) {

     DatosLista y;

     if (!L.VaciaLista()) {
          L.Primero(y);
          if Iguales(x, y) {
               return(True);
          }
          else {
               L.Resto(L);
               return( Pertenece(L, x));
          }
     }
     else {
               return(False);
     }
}
```

- En general, las soluciones recursivas son más **ineficientes** en tiempo y espacio que las versiones iterativas, debido a las llamadas a subprogramas, la creación de variables dinámicamente en la pila, la duplicación de variables, etc.

- Algunas soluciones recursivas repiten cálculos innecesariamente.

Ejemplo: cálculo del término n-ésimo de la sucesión de Fibonacci.

$Fib(n) = 1$ si $n = 0$ ó $n = 1$

$Fib(n) = Fib(n-2) + Fib(n-1)$ si $n > 1$

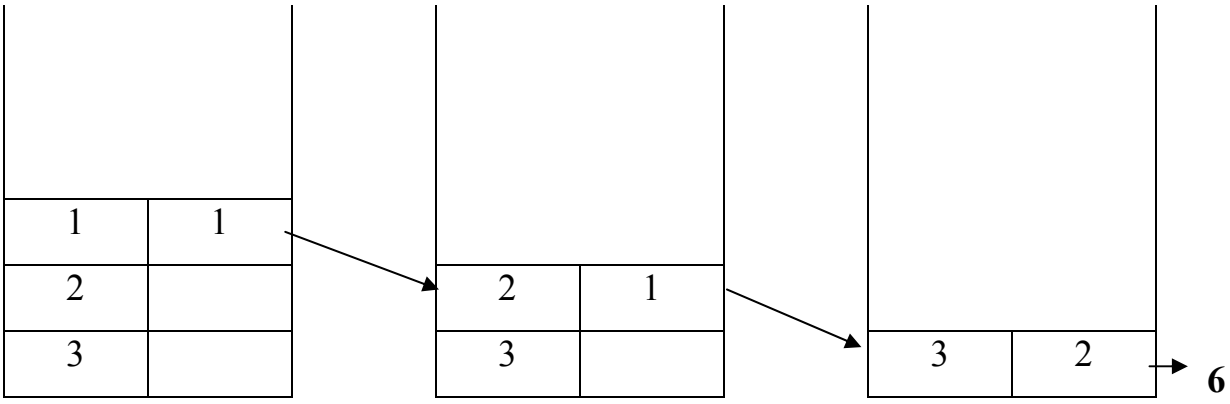
En general, cualquier función recursiva se puede transformar en una función iterativa.

- Ventaja de la función iterativa: más eficiente en tiempo y espacio.
- Desventaja de la función iterativa: en algunos casos, muy complicada; además, suelen necesitarse estructuras de datos auxiliares

Si la eficiencia es un parámetro crítico, y la función se va a ejecutar frecuentemente, conviene escribir una solución iterativa.

La recursividad se puede **simular con el uso de pilas** para transformar un programa recursivo en iterativo. Las pilas se usan para almacenar los valores de los parámetros del subprograma, los valores de las variables locales, y los resultados de la función.

Ejemplo: cálculo del factorial de 3



3. Cálculo de la complejidad

En un algoritmo recursivo, la función T(n) que establece su tiempo de ejecución viene dada por una ecuación E(n) de recurrencia, donde en la expresión aparece la propia función T.

T(n) = E(n), y en E(n) aparece la propia función T.

Para resolver ese tipo de ecuaciones hay que encontrar una expresión no recursiva de T. (En algunos casos no es tarea fácil.)

Ejemplo: Cálculo del factorial de un entero no negativo.

```
int Fact (int n) {

     if (n <= 1)
          return(1);
     else
          return( n * Fact(n-1));
}
```

Sea T(n) el tiempo de ejecución en el caso peor. Se escribe una ecuación de recurrencia que acota a T(n) **superiormente** como:

$$T(n) = \begin{cases} c_1, & si\ n \le 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

c_1: tiempo de ejecución del caso trivial n <=1.

Si n>1, el tiempo requerido por Fact puede dividirse en dos partes:

- T(n-1): La llamada recursiva a Fact con n-1.
- c2: El tiempo de evaluar la condición (n>1) y la multiplicación de n*Fact(n-1)

Para resolver las ecuaciones de recurrencia, se suelen emplear tres métodos:

1. Suponer una solución f(n), y usar la recurrencia para demostrar que T(n) = f(n). La prueba se hace generalmente por inducción sobre n.

2. **Sustituir las recurrencias por su igualdad hasta llegar a cierta $T(n_0)$ que sea conocida.**

3. Usar soluciones generales para ciertas ecuaciones de recurrencia conocidas: Resolución de la **ecuación característica.**

El método que vamos a usar es el segundo. Vamos a calcular siempre una **cota superior**.

Usando el segundo método en el caso del factorial:

$$
\begin{aligned}
T(n) &= T(n-1) + c_2 \\
&= (T(n-2) + c_2) + c_2 \qquad = T(n-2) + 2*c_2 = \\
&= (T(n-3) + c_2) + 2*c_2 \qquad = T(n-3) + 3*c_2 = \\
&\ldots \\
&= T(n-k) + k*c_2
\end{aligned}
$$

Cuando k = n–1, tenemos que $T(n) = T(1) + c_2*(n-1)$, y es O(n).

Ejemplo: Caso 1

```
int Recursiva1 (int n) {

 if (n <= 1)
   return(1);
 else
   return(Recursiva1(n-1)+ Recursiva1(n-1));
}
```

$$T(n) = \begin{cases} 1, & si\ n \le 1 \\ 2 \cdot T(n-1) + 1, & si\ n > 1 \end{cases}$$

T(n) = 2*T(n-1) + 1 =

= 2*(2*T(n-2) + 1) + 1 = 2^2*T(n-2) + (2^2-1) =

....

= 2^k*T(n-k) + (2^k-1)

Para k = n-1, T(n) = 2^{n-1}*T(1) + 2^{n-1}-1 , y por tanto T(n) es O(2^n).

Ejemplo: Caso 2

```
int Recursiva2 (int n) {

 if (n <= 1)
   return (1);
 else
   return(2 * Recursiva2(n-1));
}
```

En este caso, como en el factorial, T(n) es O(n).

Ejemplo: Caso 3

```
int Recursiva3 (int n) {

 if (n <= 1)
   return (1);
 else
   return (2 * Recursiva3(n /2));
}
```

$$T(n) = \begin{cases} 1, & si\ n \leq 1 \\ T(n/2) + 1, & si\ n > 1 \end{cases}$$

T(n) = T(n/2) + 1 =
= $T(n/2^2)$ + 2 =
....
= $T(n/2^k)$ + k

Como la ecuación $n/2^k = 1$ se resuelve para $k = \log_2 n$, tenemos que
$T(n) = T(1) + \log_2 n = 1 + \log_2 n$, y por tanto T(n) es O(log n).

Ejemplo: Caso 4

```
int Recursiva4 (int n, int x) {
  int i;
  if (n <= 1)
      return (1);
  else {
         for (i=1;i<=n;i++) {
             x = x + 1;
         }
         return( Recursiva4(n-1, x));
   }
}
```

$$T(n) = \begin{cases} 1, & si\ n \le 1 \\ T(n-1) + n, & si\ n > 1 \end{cases}$$

T(n) = T(n-1) + n = (T(n-2) + (n-1)) + n =

= ((T(n-3)+(n-2))+(n-1)+n = ... = T(n-k) + $\sum_{i=0}^{k-1}(n-i)$

Si k = n-1, T(n) = T(1) + $\sum_{i=0}^{n-2}(n-i)$ = 1 + ($\sum_{i=0}^{n-2} n$ - $\sum_{i=0}^{n-2} i$) =

= 1 + n (n-1) – (n-2)(n-1)/2. Por tanto, T(n) es $O(n^2)$.

Otra forma:

T(n) = T(n-1) + n = T(n-2) + n + (n-1) = ... = T(n-k) + $\sum_{i=n-k+1}^{n} i$

Si k = n, T(n) = T(0) + $\sum_{i=1}^{n} i$ Por tanto, T(n) es $O(n^2)$

Aplicación: Búsqueda binaria

```
boolean Búsqueda (Vector A, int iz, int de, TipoElemento x)
{    int mitad;
     if  (iz > de)
          return (False);
     else {
               mitad = (iz + de) / 2;
               if (A[mitad] == x)
                    return(True);
               else
                  if (A[mitad] > x)
                    return (Búsqueda(A, iz, mitad-1, x));
                  else
                    return (Búsqueda(A, mitad+1,de,x));
     }
}
```

La complejidad de este algoritmo es O(log n).

Realmente, la función de búsqueda no tiene por qué tener los parámetros Iz y De. La función original sería:

```
boolean  Buscar ( Vector A,  TipoElemento x) {
     return(Búsqueda(A, 0, N-1, x));
}
```

Aquí se dice que la función Búsqueda está **sumergida** en Buscar, o que hemos aplicado una **inmersión** de Búsqueda en Buscar. Se definen inmersiones para que, con los nuevos parámetros, se pueda hacer el diseño recursivo de la función **inmersora** (la función más general, Búsqueda).

¿Cuál es la complejidad de esta versión de la búsqueda binaria?

```
boolean Búsqueda (Vector A, int iz, int de, TipoElemento x)
{    int mitad;
     if  ((a[iz]>x) || (A[de]<x) || (iz > de))
          return (False);
     else {
          mitad = (iz + de) / 2;
          if (A[mitad] == x)
               return(True);
          else
               return ( Búsqueda(A, iz, mitad-1, x) ||
                        Búsqueda(A, mitad+1,de,x));
     }
}
```

Aplicación: Torres de Hanoi

Aplicación: Ordenación por mezcla (mergesort)

```
void Mezclar(vector A, indice iz, indice de, indice mitad) {
        vector B; indice k, j, i, h;

        j = mitad+1; h = iz; i = iz;
        while ((h <= mitad) && (j <= de)) {
                if (A[h] <= A[j]) {
                        B[i] = A[h];  h = h + 1;
                }
                else  {
                        B[i] = A[j];  j = j + 1;
                }
                i = i + 1;
        }
        if (h > mitad)
                for (k = j; k<=de; k++) {
                        B[i] = A[k];  i = i + 1;
                }
        else    for (k = h; k <= mitad; k++) {
                                B[i] = A[k];  i = i + 1 ;
                }
        for (k = iz; k<= de; k++)
                        A[k] = B[k];
}

void Ordena(vector A, indice iz, indice de) {
        indice mitad;

        if (iz < de) {
                mitad = (iz + de) % 2;
                ordena(A, iz, mitad);
                ordena(A, mitad+1, de);
                mezclar(A, iz, de, mitad);
        }
}

void  MergeSort (vector A, indice n) {
        Ordena(A, 0, n-1)
}
```

4. Problemas

- Calcular la complejidad de la función siguiente:

```
int Suma (Lista L, int n) {

     Lista L1, L2;
     int x, i;

     if (n = = 1) {
             L.Primero(x);
             return(x);
     }
     else {
            L1.Crear();
            L2.Crear();
            for (i=1; i<= n; i++) {
               L.Primero(x);
               L.Resto(L);
               if (i <= (n/2))
                    L1.Insertar(x);
               else
                    L2.Insertar(x);
            }
            return( Suma(L1, (n/2)) + Suma(L2,(n/2));
      }
}
```

- Calcular la complejidad de la función siguiente:

```
int F (int n) {
     int x, i;

     if (n <= 1)
         return(1);
     else {
             for(i=1;i<=n;i++) {
                 x = 1;
                 while (x < n) {
                     x = x*2;
                 }
             return(F(n/2)+F(n/2);
     }
}
```

1.- Diseñar una función recursiva que devuelva la suma de los valores de una pila de enteros.

2.- Función recursiva que liste en orden inverso los elementos de una pila.

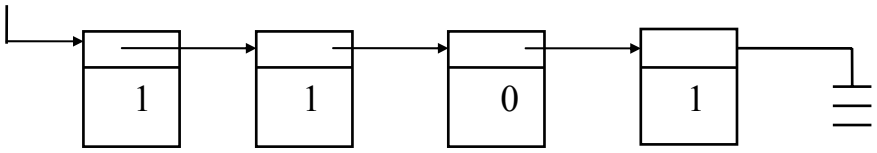
3.- Función recursiva que liste en orden inverso los elementos de una cola.

4.- Decimos que una pila P es sombrero de otra pila Q, si todos los elementos de P están en Q, en el mismo orden y en las posiciones más cercanas a la cima de la pila (incluida esta). Por definición la pila vacía es cima de cualquier otra. Diseñar una función que dadas 2 pilas nos diga si una es sombrero de otra.

5.- Implementar la función *Borrar (L: Lista; X: DatosLista)* de forma recursiva. Esta función eliminará de la lista el elemento tantas veces como éste aparezca.

6.- Diseñar una función que sume 1 a un nº en binario que viene almacenado en una lista, donde el primer elemento de la lista será el bit de menor peso del número.

Ejemplo:



Número: 1011 + 1 = 1100

7.- Dada una lista de colas, donde da igual el tipo de los elementos, diseñar una función que liste todos los elementos de la estructura.

8.- Dada una matriz como la indicada en la figura y las coordenadas de una posición, diseñar una función que elimine la figura que contiene a esa posición.

Ejemplo:

	1	2	3	4	5
1	#	#			
2	#			#	
3				#	
4			#	#	
5					

→ función (3, 4) →

	1	2	3	4	5
1	#	#			
2	#				
3					
4					
5					