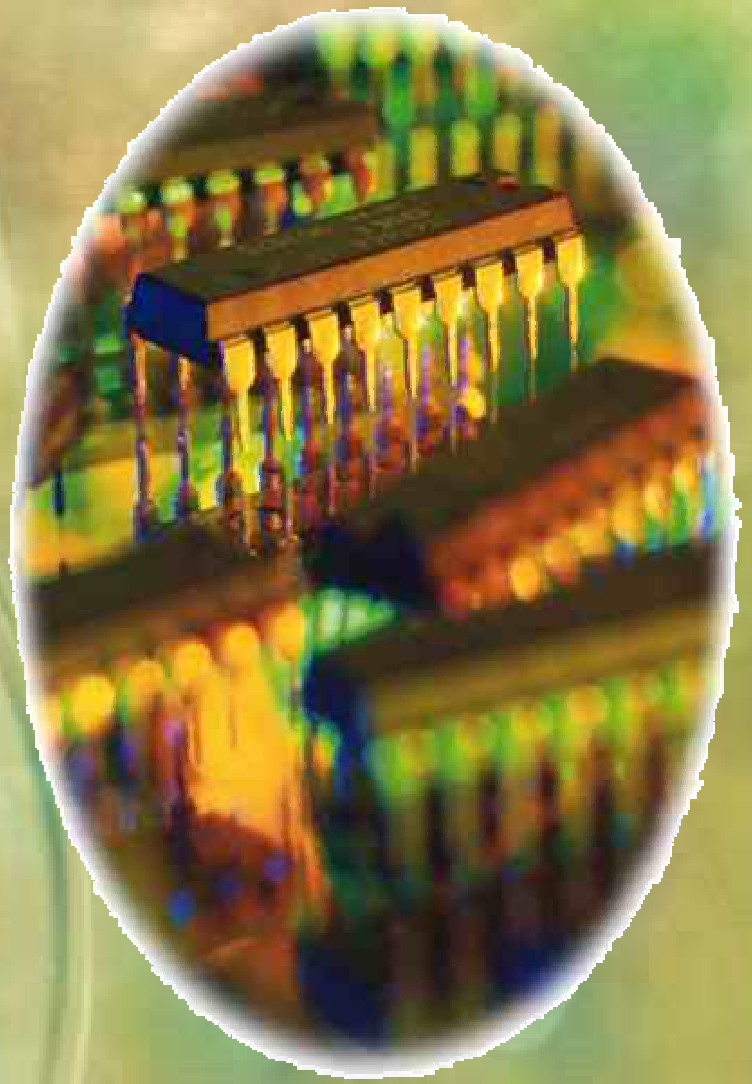
TEMA

2

Recursividad



Dept. Ciencias de la Computación e I.A.

Universidad de Granada

Índice

- Introducción a la recursión
- Recursividad frente a iteración
- Eficiencia de los algoritmos recursivos
- Ejemplos y ejercicios

Definición

Una función es recursiva cuando el concepto que se introduce está definido en base a si mismo

Ejemplo: Factorial de un número

n!= 1, si n=0

n!= n(n-1)!, si n≠0

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

Ejercicios: Define funciones recursivas para calcular la sumatoria de los n primeros naturales y la potencia de entero con exponente natural

Verificación de las funciones recursivas

a. **Debe haber, al menos, una posible llamada a la función que produce un resultado sin provocar una nueva llamada recursiva. A esto se le llama *caso base* de la recursión**

Ejemplo: Factorial de un número

No es correcto

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

Verificación de las funciones recursivas

b. **Todas las llamadas recursivas se refieren a un caso que es más cercano al caso base**

Ejemplo: Factorial de un número

n!=(n+1)!/(n+1)

No es correcto (se verifica **a**)

```
int factorial(int n){
 if (n==0)
      return 1;
 else
      return factorial(n+1)/(n+1);
}
```

Verificación de las funciones recursivas

c. **El caso base debe acabar alcanzándose**.

Ejemplo: Factorial de un número

$$n!=n(n-1)(n-2)!$$

No es correcto si n impar (se verifica **a** y **b**)

```
int factorial(int n){
 if (n==0)
      return 1;
 else
      return n*(n-1)*factorial(n-2);
}
```

Funciones recursivas con caso base múltiple

Sucesión de Fibonacci

fib(1)=0,

fib(2)=1,

fib(n)=fib(n-1)+fib(n-2), para n≠1 y n≠2

Ejercicio: Implementar la función

```
int fib(int n){
 if (n==0)
         cout<<“Error. No está definido”;
 else if (n==1)
         return 0;
         else if (n==2)
                  return 1;
              else
                  return fib(n-1)+fib(n-2);
}
```

No es correcto.
Para evitar errores utilizar FILTRO al usar la función.

Recursividad ante iteración

- Cualquier problema resuelto de forma recursiva puede ser resuelto de forma iterativa
  - La forma recursiva suele usar sentencias de selección (if)
  - La forma iterativa usa sentencias de iteración (for, while, do-while)
- En forma iterativa es necesario introducir variable locales
- La forma recursiva suele ser menos eficiente (en tiempo) que la iterativa pero mas simple

Ejemplos

Resolución del factorial de forma iterativa

```
int factorial(int n){
 int i, resultado;
 resultado=1;
 for (i=2;i<=n;i++)
     resultado=resultado*i;
 return resultado;
}
```

Ejercicios: Implementa de forma iterativa la recurrencia de Fibonacci

Ejemplos

Resolución de los números de Fibonacci de forma iterativa

```
int fib(int n){
    int resultado, f1, f2, i;
    if (n==0)
            cout<<“Error. No está definido”;
    else if (n==1)
            resultado = 0;
          else{
                        f1 = 0;
                        resultado = 1;
                        for (i=1; i<=n-2; i++){
                                    f2 = f1;
                                    f1 = resultado;
                                    resultado = f1 + f2;
                        }
           }
return resultado;
}
```

Eficiencia de los algoritmos recursivos

Los agoritmos recursivos suelen ser menos eficientes en tiempo que los iterativos.

Motivos:

1. Cada llamada requiere tiempo para
   - Hacer la llamada a la función
   - Crear las variables locales y copiar los parámetros por valor
   - Ejecutar las instrucciones en parte ejecutiva de la función
   - Destruir las variables locales y parámetros por valor
   - Salir de la función
2. Posible sobrecarga debido a la solución (ver Fibonacci)

Ejercicio 1

Realiza un subprograma que implemente la búsqueda lineal en forma recursiva

```
int busqueda_lineal(int a[], int elemento, int primero, int tam){
    if (primero > tam-1)
        return (-1);
    else if (a[primero] == elemento)
                return primero;
        else
    return busqueda_lineal(a,elemento,primero+1,tam);
}
```

Ejercicio 2

Realiza un subprograma que implemente la búsqueda binaria en forma recursiva

```
int busqueda_binaria(int a[], int elemento, int primero, int ult){
 int central;
 If (primero > ult-1)
    return (-1);
 else{
    central = (primero + ult)/2;
    if (a[central] == elemento)
          return central;
    else if (a[central]>elemento)
                  return busqueda_binaria(a,elemento,primero,central-1);
          else
                  return busqueda_binaria(a,elemento,central+1,ult);
 }
}
```

Ejercicio 3

Realiza un subprograma que implemente el método de ordenación por selección directa de forma recursiva

```
void OrdSelRecursivo(int a[], int primero, int ultimo) {
  int i, PosMenor, aux;
  // Busca el menor elemento
  if (primero + 1 < ultimo) {
    PosMenor = primero; // Supone que el menor es el primero
    for (i = primero + 1; i < ultimo; i++) // Encuentra el menor
      if (a[i] < a[PosMenor])
        PosMenor = i;
    // Lleva a cabo el intercambio
    aux = a[primero];
    a[primero] = a[PosMenor];
    a[PosMenor] = aux;
    // Llamada recursiva a la función con el subvector
    OrdSelRecursivo(a, primero + 1,ultimo);
  }
}
```

Quicksort I

El algoritmo *Quicksort* fue desarrollado en 1962 por C.A.R. Hoare, antes de que se implementaran los primeros lenguajes con capacidad para ejecutar funciones recursivas.

El algoritmo, aplicado a un (sub)vector *vec[ini..fin]* es:

1. **Elegir un elemento *p* cualquiera del vector, entre *ini* y *fin*. El elemento *p* se llama *pivote*. Usualmente se elige el elemento que está en el medio.**

2. **Particionar el vector en dos subvectores: *vec[ini..p]* y *vec[p+1..fin]*, intercambiando elementos de modo que todos los elementos que son menores que *p* estén a la izquierda y todos los mayores que *p* estén a la derecha.**

3. **Aplicar *Quicksort* al subvector *vec[ini..p]***

4. **Aplicar *Quicksort* al subvector *vec[p+1..fin]***

Quicksort II

La función que implementa el algoritmo Quicksort podría implementarse de la siguiente manera:

```
/*************************************************************************
 * Función recursiva que ordena un vector de enteros mediante Quicksort.
 * Si el vector no tiene mas de dos elementos no se hace nada. De lo
 * contrario, se particiona el vector y se aplica el método a los dos
 * subvectores que resultan de la partición.
 * Se reciben como parámetros el vector y el índice inicial y final.
 *************************************************************************/
void QuickSort(int a[], int ini, int fin) {
   int p;
   if (ini < fin) {
      p = ParticionarVector(a, ini, fin);
      QuickSort(a, ini, p);
      QuickSort(a, p + 1, fin);
   }
}
```

Quicksort III

```
/*********************************************************************
 * Función para particionar un vector en dos subvectores, intercambiando
 * elementos de modo que todos los menores que el pivote estén a la
 * izquierda y los mayores a la derecha.
 * Se reciben como parámetros el vector y el índice inicial y final.
 * Se retorna el índice para particionar el vector (posición del pivote).
 *********************************************************************/
int ParticionarVector(int a[], int ini, int fin)
{
   int pivote = a[(ini + fin) / 2];
   int izq = ini - 1;
   int der = fin + 1;
   int aux;
   while (izq < der) {
      do {// Busca el 1er. elemento de la izq. para intercambiar
         izq++;
      } while (a[izq] < pivote);
      do {// Busca el 1er. elemento de la derecha para intercambiar
         der--;
      } while (a[der] > pivote);
      if (izq < der) {// Lleva a cabo el intercambio
         aux = a[izq];
         a[izq] = a[der];
         a[der] = aux;
      }
   }
   return(der); // der es la posición donde queda el mayor en el último cambio
}
```