Tema 9. Recursividad

`http://aulavirtual.uji.es`

José M. Badía, Begoña Martínez, Antonio Morales y José M. Sanchiz

{badia, bmartine, morales, sanchiz}@icc.uji.es

Estructuras de datos y de la información

Universitat Jaume I

Índice

Bibliografía

- (Nyhoff'99), Capítulo 10 y apartados 13.3 y 13.4.
- (Main y Savitch'01), capítulos 9 y 13.
- *LISP. Introducción al cálculo simbólico*. D.S. Touretzky. Díaz de Santos, 1986. capítulo 8.
- Apuntes de "Metodología y tecnología de la programación".

Objetivos

- Aprender a interpretar algoritmos recursivos.
- Aprender a resolver problemas sencillos de forma recursiva.
- Aprender a calcular el coste de algoritmos recursivos.
- Conocer los principales algoritmos recursivos de ordenación.

1 Introducción

Definición

Un algoritmo es recursivo cuando se llama a sí mismo, directa o indirectamente.

Ejemplo

```
int sumatorio (int n) {
  int s;
  if (n == 1)  return 1;
  else {
     s = sumatorio(n - 1);
     s = n + s;
     return s;
  }
}
```

1 Introducción (II)

Consideraciones

- La solución del problema se expresa en función de la solución de un subproblema del mismo tipo de menor tamaño o más simple.
- El proceso se repite varias veces de modo recursivo.
- La recursión se para cuando encontramos un subproblema con solución trivial: condición de parada.
- El algoritmo recursivo consta de dos fases que se desarrollan en "direcciones" contrarias:
  - Aplicación de la recursión
  - Retorno de la recursión

1 Introducción (III)

Estructura de un algoritmo recursivo

Todo algoritmo recursivo consta de dos partes:

1. Resolución del caso base.
   - Se aplica cuando se cumple la condición de parada.
   - Se resuelve de modo trivial.
2. Realización de la/s llamada/s recursiva/s.
   - Se llama a la función en un caso más simple.
   - En ocasiones se realiza alguna operación adicional antes o después.

Recursión infinita

Utilidad de la recursividad

- **Ventajas**
  - Da lugar a soluciones simples de problemas complejos.
  - Da lugar a algoritmos simples.
- **Inconvenientes**
  - Los algoritmos recursivos suelen ser difíciles de entender.
  - Son más costosos en tiempo y espacio que sus versiones iterativas.

2 Tipos de recursividad

- **Directa vs. Indirecta**
  - **Directa:** El algoritmo recursivo se llama a sí mismo.
  - **Indirecta:** El algoritmo recursivo llama a otro que provoca eventualmente una llamada al original.
- **Final vs. No final**
  - **Final:** Al finalizar la llamada recursiva, no queda ninguna acción por ejecutar salvo retornar.
  - **No final:** Después de la llamada recursiva quedan acciones por realizar.

2 Tipos de recursividad (II)

Sumatorio con recursividad indirecta

```
int suma2 (int n);

int suma1 (int n) {
  if (n == 1)
    return 1;
  else
    return suma2(n);
}

int suma2 (int n) {
  return (n + suma1(n - 1));
}
```

2 Tipos de recursividad (III)

Sumatorio con recursividad final

```
void sumaf (int n, int & res) {
   if (n == 1)
      res = res + n;
   else{
      res = res + n;
      sumaf(n-1, res);
   }
}
...
int n=10, res=0;
sumaf(n, res);
```

3 Implementación recursiva

- Intentar plantear la solución del problema en función de la solución de un subproblema del mismo tipo.
  - **Factorial:** $n! = n * (n-1)!$
  - **Sumatorio:** $\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$
  - **Máximo:** $\text{máx}_{i:1\ldots n}\{v[i]\} = maximo(v[n], \text{máx}_{i:1\ldots n-1}\{v[i]\})$
- Encontrar uno o varios casos en los que la solución sea trivial: casos base $\equiv$ condición/es de parada.
  - **Factorial:** $0! = 1$ y $1! = 1$
  - **Sumatorio:** $\sum_{i=1}^{1} i = 1$
  - **Máximo:** $\text{máx}_{i:1\ldots 1}\{v[i]\} = v[1]$

4 Coste de algoritmos recursivos

- El coste de un algoritmo recursivo puede expresarse como una ecuación recursiva.
  La función $t(n)$ representará el número total de pasos básicos.

  Ejemplo: Cálculo del sumatorio: $t(n) = \begin{cases} 1 & \text{si } n = 1 \\ 1 + t(n-1) & \text{si } n > 1 \end{cases}$

- Obtenemos una ecuación no recursiva *desplegando* la ecuación anterior.

$$\begin{aligned} t(n) &= 1 + t(n-1) \\ &= 1 + 1 + t(n-2) \\ &\dots \\ &= \overbrace{1 + 1 + \ldots + 1}^{k \text{ veces}} + t(n-k) \quad (k = n-1) \\ &= \overbrace{1 + 1 + \ldots + 1}^{n \text{ veces}} = n \in O(n) \end{aligned}$$

4 Coste de algoritmos recursivos (II)

Búsqueda dicotómica recursiva

```
int buscadicr (float v[], int desde, int hasta, float buscado) {
  int centro;
  if (desde > hasta) return -1;
  else {
    centro = (desde + hasta)/2;
    if (v[centro] == buscado)
      return centro;
    else if (v[centro] > buscado)
      return buscadicr(v, desde, centro-1, buscado);
    else
      return buscadicr(v, centro+1, hasta, buscado);
  }
}
```

4 Coste de algoritmos recursivos (III)

Búsqueda dicotómica recursiva. Coste en el caso peor

$$t(n) = \begin{cases} 1 & \text{si } n = 1 \\ 1 + t(\frac{n}{2}) & \text{si } n > 1 \end{cases}$$

$$\begin{aligned} t(n) &= 1 + t(n/2) \\ &= 1 + 1 + t(n/4) \\ &\dots \\ &= \overbrace{1 + 1 + \ldots + 1}^{k \text{ veces}} + t(n/2^k) \quad (k = \log_2 n) \\ &= \overbrace{1 + 1 + \ldots + 1}^{1+\log_2 n \text{ veces}} = 1 + \log_2 n \in O(\log_2 n) \end{aligned}$$

4 Coste de algoritmos recursivos (IV)

Números de Fibonacci

F_1	F_2	F_3	F_4	F_5	F_6	F_7	$\ldots$	F_i
1	1	2	3	5	8	13	$\ldots$	$F_{i-1}+F_{i-2}$

$$F_n = \begin{cases} 1 & \text{si } n = 1 \text{ ó } n = 2 \\ F_{n-1} + F_{n-2} & \text{si } n > 2 \end{cases}$$

```
int fibonacci (int n) {
  if ( (n == 1) || (n == 2) )  return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

4 Coste de algoritmos recursivos (V)

Fibonacci: $t(n) = \begin{cases} 1 & \text{si } n = 1 \text{ ó } n = 2 \\ 1 + t(n-1) + t(n-2) & \text{si } n > 2 \end{cases}$

$$
\begin{aligned}
t(n) &= 1 + t(n-1) + t(n-2) \leq 1 + 2 * t(n-1) \\
&= 1 + 2 * (1 + t(n-2) + t(n-3)) \leq 1 + 2 * (1 + 2 * t(n-2)) \\
&= 1 + 2 + 2^2 * t(n-2) \\
&\ldots \\
&= 1 + 2 + 2^2 + \ldots + 2^k * t(n-k) \quad (k = n-2) \\
&= \sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1 \in O(2^n) : \textit{exponencial}!!!
\end{aligned}
$$

5 Recursión frente a Iteración

Todo algoritmo recursivo tiene una versión iterativa.

```
int fact_ite(int n) {
  int res = 1;
  for (int i=1; i<=n;i++)
    res = res*i;
  return res;
}

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

- Inconveniente:
  - Suele ser más largo y complicado.
- Ventaja:
  - Tiene un coste espacial y temporal inferior.

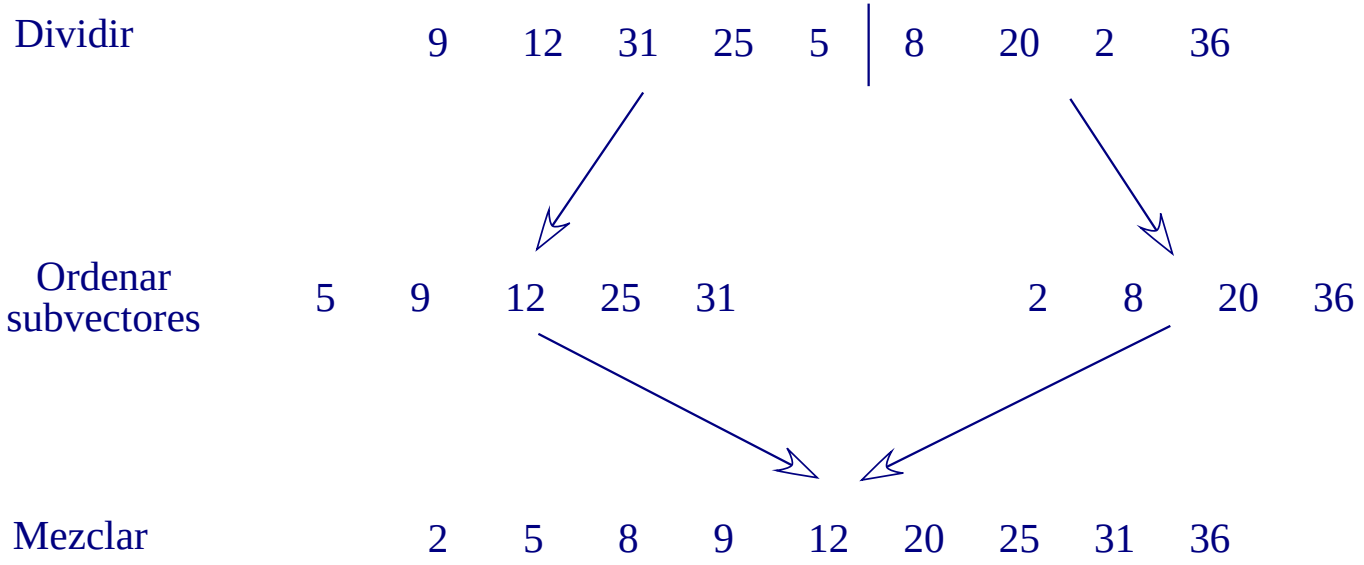
6 Ordenación recursiva

- Aplican un esquema divide y vencerás
  1. Dividir los elementos a ordenar en dos grupos de dimensión similar.
  2. Ordenar cada uno de los grupos. Posiblemente aplicando recursivamente el mismo esquema de división.
  3. Combinar los dos grupos ordenados en uno mayor.
- Tienen un coste medio $\in O(n \log_2 n)$
  - Heapsort
  - Mergesort
  - Quicksort

Idea general

1. Dividir el vector en dos partes iguales o casi iguales.
2. Ordenar cada subvector usando, por ejemplo, mergesort.
3. Mezclar los subvectores ordenados en uno solo.

Ejemplo. Una etapa



Algoritmo mergesort

```
void mergesort (int v[], int n) {
  if ( n > 1 ) {
    // División y Ordenación recursiva de los subvectores
    mergesort(v, n/2);
    mergesort(v+n/2, n-n/2);
    // Mezcla de los subvectores ordenados
    merge(v, n/2, v+n/2, n-n/2);
  }
}
```

Algoritmo de mezcla

Mientras queden elementos por recorrer en ambos subvectores
  Comparar los dos elementos actuales de ambos.
  Copiar el elemento menor en el vector auxiliar.
  Mover el índice del auxiliar y del vector con el elemento menor.
finmientras
Copiar los elementos restantes del vector no vacío en el auxiliar.
Copiar los elementos del auxiliar en el original.

Algoritmo merge

```
void merge(int v[], int nv, int w[], int nw) {
  int i=0, j=0, k=0, *vaux = new int[nv+nw];
  while ( (i < nv) && (j < nw) )
    if (v[i] < w[j])
      vaux[k++] = v[i++];
    else
      vaux[k++] = w[j++];
  while (i < nv) vaux[k++] = v[i++];
  while (j < nw) vaux[k++] = w[j++];
  for (k=0; k < nv+nw; k++)  v[k] = vaux[k];
  delete [] vaux;
}
```

6.1 Mergesort (V)

Coste de mergesort: $t(n) = \begin{cases} 1 & \text{si } n = 1 \\ 1 + t(n/2) + t(n/2) + coste\ de\ merge & \text{si } n > 1 \end{cases}$

Coste de merge: $\in O(n)$

$$
\begin{aligned}
t(n) &= 1 + t(n/2) + t(n/2) + O(n) = 1 + 2 * t(n/2) + O(n) \\
&= 1 + 2 * (1 + t(n/4) + t(n/4) + O(n/2)) + O(n) \\
&= 1 + 2 + 2^2 * t(n/2^2) + 2 * O(n) \\
&= 1 + 2 + 2^2 * (1 + t(n/2^3) + t(n/2^3) + O(n/2^2)) + 2 * O(n) = \ \ldots \\
&= \sum_{i=0}^{k-1} 2^i + 2^k * t(n/2^k) + k * O(n) \\
&= 2^k - 1 + 2^k * t(n/2^k) + k * O(n) \quad (k = \log_2 n) \\
&= (n-1) + n + O(n \log_2 n) \in O(n \log_2 n)
\end{aligned}
$$

Idea general

1. Elegir un elemento del vector como `pivote`.
2. Dividir el vector original en un subvector `v` con los elementos menores que el pivote y otro `w` con los elementos mayores.
3. Ordenar ambos subvectores, por ejemplo, utilizando quicksort.
4. Devolver el vector resultante de concatenar: `v`, `pivote`, `w`.

Algoritmo quicksort

```
void quicksort (int v[], int primero, int ultimo) {
  int pos; // posición definitiva del pivote
  if ( primero < ultimo ) { // vector con más de un elemento
    // Elección del pivote y División en dos subvectores
    pos = Dividir(v, primero, ultimo);
    // Ordenación de los subvectores
    quicksort(v,  primero, pos-1);
    quicksort(v,  pos+1, ultimo);
  }
}
```

Algoritmo de división

Elegir el elemento pivote (p.e. el primero del vector).
Recorrer el vector desde la izquierda y la derecha a la vez
  Buscar desde la derecha un elemento menor o igual que el pivote.
  Buscar desde la izquierda un elemento mayor que el pivote.
  Intercambiar ambos elementos.
Hasta que coincidan o se hayan cruzado ambas posiciones.
Intercambiar el pivote y el elemento en el que finalizó
el recorrido desde la derecha.

Algoritmo de división

```
int Dividir(int v[], int primero, int ultimo) {
  int pivote=v[primero], izq=primero, der=ultimo;
  while ( izq < der ) {
    while (v[der] > pivote)
      der--;
    while ( (izq < der) && (v[izq] <= pivote) )
      izq++;
    if (izq < der)
      Intercambiar(v[izq], v[der]);
  }
  Intercambiar(v[primero], v[der]);
  return der;  //posición en que ha quedado el pivote
}
```

6.2 Quicksort (V)

Quicksort: $t(n) = \begin{cases} 1 & \text{si } n = 0 \text{ ó } n = 1 \\ 1 + t(n') + t(n'') + coste\ de\ dividir & \text{si } n > 1 \end{cases}$

Coste de dividir: $\in O(n)$

Caso peor: vector ordenado: ($n' = 0$ y $n'' = n-1$) o ($n' = n-1$ y $n'' = 0$)

$$\begin{aligned}
t(n) &= 1 + t(0) + t(n-1) + O(n) \\
&= 1 + 1 + (1 + t(0) + t(n-2) + O(n-1)) + O(n) \\
&= 4 + t(n-2) + O(n-1) + O(n) \\
&= 4 + (1 + t(0) + t(n-3) + O(n-2)) + O(n-1) + O(n) = \ldots \\
&= 2 * k + t(n-k) + O(n-k+1) + \ldots + O(n) \quad (k = n-1) \\
&= 2 * (n-1) + t(n-(n-1)) + \sum_{i=2}^{n} O(i) \\
&= 2 * (n-1) + 1 + O(n^2/2) \in O(n^2) : cuadrático!!
\end{aligned}$$

Caso mejor: en cada etapa $n' \approx n'' \approx n/2$

$$
\begin{aligned}
t(n) &= 1 + t(n/2) + t(n/2) + O(n) = 1 + 2 * t(n/2) + O(n) \\
&= 1 + 2 * (1 + t(n/4) + t(n/4) + O(n/2)) + O(n) \\
&= 1 + 2 + 2^2 * t(n/2^2) + 2 * O(n) \\
&\dots \\
&= \sum_{i=0}^{k-1} 2^i + 2^k * t(n/2^k) + k * O(n) \qquad (k = \log_2 n) \\
&= (n-1) + n + O(n \log_2 n) \in O(n \log_2 n)
\end{aligned}
$$

Selección del pivote:

- Mediana de los elementos del vector.
- Mediana de tres elementos aleatorios del vector.