Grado en Ingeniería Información

Estructura de Datos y Algoritmos

Sesión 12

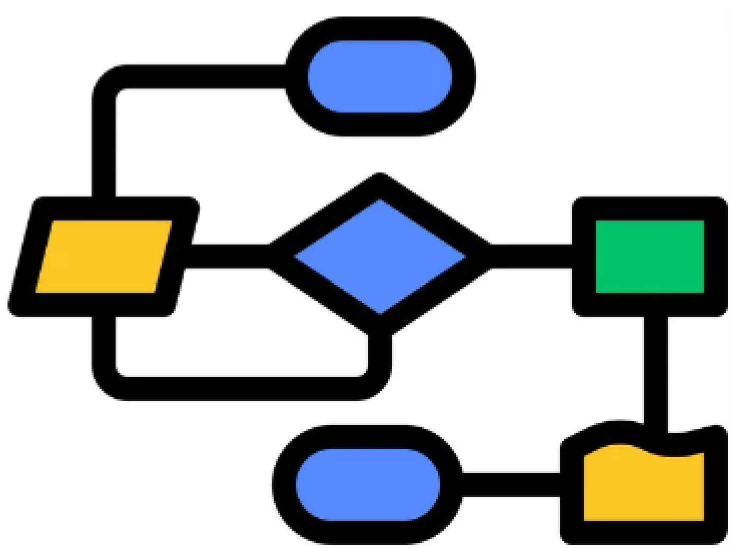
Curso 2023-2024

Marta N. Gómez

UNIVERSIDAD NEBRIJA

T4. Técnicas Algorítmicas

- Divide y Vencerás
- Algoritmos Voraces
- Programación Dinámica
- Backtracking

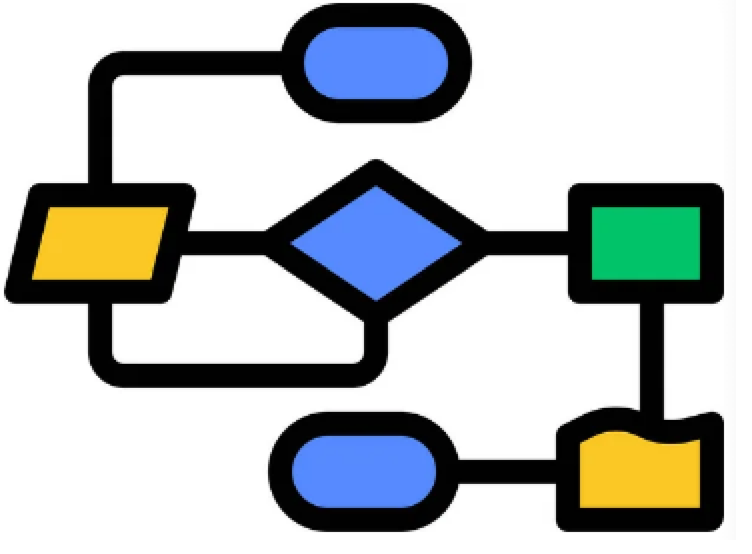


UNIVERSIDAD NEBRIJA

ÍNDICE

T4. Técnicas Algorítmicas

- **Divide y Vencerás**
- Algoritmos Voraces
- Programación Dinámica
- Backtracking



UNIVERSIDAD NEBRIJA

ÍNDICE

La técnica ***divide y vencerás*** consiste en:

- **Descomponer un problema** en un conjunto de **subproblemas del mismo tipo**, pero **más pequeños**.
- La **resolución** de los subproblemas se hace **aplicando la misma técnica**.
- La **combinación de las soluciones** permite resolver el **problema original**.

Normalmente, la resolución de los subproblemas se hace de ***forma recursiva***.

Proceso:

- Si el **problema (P)** tiene **solución directa** para los datos **(D), se resuelve.**
- En caso contrario los pasos son los siguientes:
  1. **Dividir el problema**: se plantea el problema de forma que se pueda descomponer en k **subproblemas de igual tipo** y de **menor tamaño**.
  2. **Resolver los subproblemas**: solución de forma **independiente**, directamente cuando son elementales (**caso base**) o de **forma recursiva**.
  3. **Combinar las soluciones**: construir la solución del problema original combinando las soluciones obtenidas anteriormente.

Esquema del algoritmo:

```
tip_Sol DivideyVenceras (tip_Pb pb) {
    if (esCasoBase(pb)) {
        return solucionCasoBase(pb);
    }
    else {
        dividirProblema(pb, subPb);
        for (i{0}; i < subPb.size(); i++) {
            Sol_subPb.at(i) = DivideyVenceras(subPb.at(i));
        }
        return combinarSoluciones(Sol_subPb);
    }
}
```

Consideraciones:

- El **número** de **subproblemas** (k) debe ser **pequeño**.
- Los **subproblemas** deben tener un **tamaño parecido** y que **no se solapen entre sí**.
- Las operaciones de ***dividirProblema*** y ***combinarSoluciones*** sean bastante **eficientes**.
- Hay que **evitar dividir el problema** cuando el tamaño del subproblema es suficientemente **pequeño**.

Eficiencia:

Si el problema $\boldsymbol{x}$ es de tamaño $\boldsymbol{n}$ y los subproblemas $\boldsymbol{x_1, x_2, \ldots, x_k}$ son de tamaño $\boldsymbol{n_1, n_2, \ldots, n_k}$, respectivamente, el **coste en tiempo** del diseño del **algoritmo recursivo de divide y vencerás** produce **la ecuación de recurrencia** de la forma:

$$\boldsymbol{T(n)} = \begin{cases} \boldsymbol{g(n)} & \boldsymbol{n \leq n_0} \\ \boldsymbol{\sum_{j=1}^{k} T(n_j) + f(n)} & \boldsymbol{n > n_0} \end{cases}$$

donde:

$\boldsymbol{T(n)}$: coste del algoritmo para el problema de tamaño $\boldsymbol{n}$

$\boldsymbol{n_0}$: tamaño umbral para no seguir dividiendo

$\boldsymbol{g(n)}$: coste del caso base

$\boldsymbol{f(n)}$: coste de descomponer el problema y combinar soluciones

Eficiencia:

Muchos algoritmos **divide y vencerás** responden a la **ecuación de recurrencia** de la forma:

$$T(n) = \begin{cases} c & si\ 0 \le n < b \\ kT(n/b) + f(n) & si\ n \ge b \end{cases}$$

donde:

k: número de subproblemas
n/b: tamaño de cada subproblema

Si se supone que la descomposición y la combinación no son muy costosas, su coste será polinómico: $f(n) \in \theta(n^i)$

Luego, la ecuación de recurrencia que hay que analizar será:

$$T(n) = kT(n/b) + \theta(n^i)$$

Eficiencia:

$$T(n) = kT(n/b) + \theta(n^i)$$

$k < b^i$	$T(n) \in \theta(n^i)$
$k = b^i$	$T(n) \in \theta(n^i log_b n)$
$k > b^i$	$T(n) \in \theta(n^{log_b k})$

Por tanto, el uso de la técnica de divide y vencerás <u>no garantiza la eficiencia del algoritmo</u>. Como se recoge en la tabla, el coste puede <u>empeorar</u>, <u>mantenerse</u> o <u>mejorar</u> respecto a la eficiencia de un algoritmo iterativo del mismo problema.

Algoritmo de búsqueda binaria:

Se compara el **dato a buscar** con el **elemento central del vector**:

- Si es el elemento buscado, se **finaliza**.
- Si no, se sigue **buscando en la mitad del vector** que determine la relación entre el valor del elemento central y el buscado, parte izquierda o derecha del vector.

El algoritmo finaliza cuando se **localiza el dato** buscado en el vector o **se termina el vector** porque no existe.

EJEMPLO: Divide y Vencerás

Búsqueda Binaria de un elemento de un Vector Ordenado

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15 > 7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15 > 12

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15 = 15

Análisis en el mejor caso

```
int busquedaBinaria (const string & f, char l)
{
    int i, ppio{0}, final;                        ← Ω(1)

    final = f.size()-1;                           ← Ω(1)

    while (ppio <= final)                         ← Ω(1)
    {
        i = (ppio+final)/2;                       ← Ω(1)

        if (l == f.at(i)) {                       ← Ω(1)
            return i;
        }
        else if (l < f.at(i)) {  // se busca en la mitad izquierda
            final = i-1;
        }
        else {                   // se busca en la mitad derecha
            ppio = i+1;
        }
    }

    return -1;
}
```

$\Omega(1)$

Luego: $T(n)$ es $\Omega(1)$

Análisis en el peor caso

```
int busquedaBinaria (const string & f, char l)
{
    int i, ppio{0}, final;                              ← O(1)

    final = f.size()-1;                                 ← O(1)

    while (ppio <= final)                               ← O(log n)
    {
        i = (ppio+final)/2;                             ← O(1)

        if (l == f.at(i)) {
            return i;                                   ← O(1)
        }
        else if (l < f.at(i)) {  // se busca en la mitad izquierda
            final = i-1;                                ← O(1)
        }
        else {                   // se busca en la mitad derecha
            ppio = i+1;                                 ← O(1)
        }
    }

    return -1;                                          ← O(1)
}
```

$O(1)$

$O(\log n)$

$O(\log n)$

Luego: *T(n)* es ***O(log n)***

Análisis en el peor caso

Hay que determinar el número de veces que se hace el bucle:

- **Cada iteración** del bucle **reduce**, aproximadamente, a la **mitad el número de elementos donde se busca** (tamaño del vector).
- Después de **k iteraciones** el número de elementos sobre el que se busca será, a lo sumo: $n/2^k$.
- La **última iteración** se produce cuando el **número de elementos es 1**, es decir: $1 = n/2^k$

Análisis en el peor caso

Tomamos logaritmos para resolver: $1 = \boldsymbol{n/2^k}$

$$\frac{n}{2^k} = 1 \Leftrightarrow log_2\frac{n}{2^k} = log_2 1 = 0$$

$$0 = log_2\frac{n}{2^k} = log_2 n - log_2 2^k = log_2 n - k$$

$$k = log_2 n$$

Luego, el **número de iteraciones (*k*) está acotado superiormente por $\boldsymbol{log_2 n}$.**

EJEMPLO: Divide y Vencerás

Búsqueda Binaria de un elemento de un Vector Ordenado

```
int busquedaBinariaREC (const string &f, char l, int ppio, int final)
{
    int i;

    if (ppio > final) return -1;
    else {
        i = (ppio+final)/2;
        if (l == f.at(i)) {
            return i;
        }
        else if (l < f.at(i)) {
            // se busca en la mitad izquierda
            return busquedaBinariaREC(f, l, ppio, i-1);
        }
        else {
            // se busca en la mitad derecha
            return busquedaBinariaREC(f, l, i+1, final);
        }
    }
}
```

Algoritmo de búsqueda binaria recursivo:

El problema se descompone en un **subproblema de tamaño n/2** y el **coste de la descomposición** y la **combinación de soluciones es constante**:

$$T(n) = T(n/2) + O(1) =$$

$$T(n/4) + O(1) + O(1) = \ \dots = T\left(n/\, 2^{\log n}\right) + \log n \, O(1) =$$

$$T(n/n) + \log n \, O(1) = 1 + \log n \, O(1) \in O(\log n)$$

Luego: $T(n)$ es $O(\log n)$

Adivinar un número natural positivo:

Se trata de **pensar un número natural positivo** y que lo **adivinen** simplemente **preguntando si es menor o igual que otros números**.

Solución - Adivinar un número natural positivo:

Se basa en el algoritmo de la **búsqueda binaria**, pero **sin vector**.

El problema **siempre tiene solución**.

El algoritmo **devuelve el número que está buscando** como resultado.

Utiliza una función "***esMenorIgual***" que devuelve ***true*** si y solo si el número que hay que adivinar **es menor o igual** que el que recibe dicha función como parámetro.

EJEMPLO 1: Divide y Vencerás

Adivinar un número natural positivo

```
int busquedaBinariaJuego (int ini, int fin) {
    if (ini == fin){
        return fin;    ← O(1)
    }
    else {
        int mitad = (ini + fin)/2;
        if (esMenorIgual(mitad)){
            return busquedaBinariaJuego (ini, mitad);
        }
        else {
            return busquedaBinariaJuego(mitad+1, fin);
        }
    }
}
```

$O(\log n)$

EJEMPLO 1: Divide y Vencerás

Adivinar un número natural positivo

```
int adivinarNumero (){
    // Primero hay que encontrar la cota inferior y superior
    int cInf{1}, cSup{1};

    while (!esMenorIgual(cSup)){
        cInf = cSup + 1;
        cSup = 2 * cSup;
    }

    return busquedaBinariaJuego(cInf, cSup);
}
```

$O(\log n)$

$O(\log n)$

$O(\log n)$

Ordenación rápida (Quicksort)

- El algoritmo consiste en **dividir el vector** que se desea ordenar en **dos bloques.** En el primero se sitúan todos los elementos que son **menores** que un valor que se toma como referencia (**pivote**) y en segundo bloque el resto.
- Este procedimiento se repite **dividiendo a su vez cada uno de estos bloques** y **repitiendo la operación** anteriormente descrita.
- La **condición de parada** se da cuando el bloque que se desea ordenar está formado por un **único elemento** (<u>bloque ordenado</u>).
- El **resultado se obtiene de la combinación de todos los resultados parciales**.

EJEMPLO: Divide y Vencerás

Ordenación rápida (Quicksort)

1era. ejecución

Pivote = (0+5)/2=2

50	60	20	30	40	**10**

10	**60**	**20**	30	40	**50**

10	**20**	**60**	30	40	50

2da. ejecución

Pivote = (2+5)/2=3

10	**20**

60	30	40	50

30	**60**	40	50

3era. ejecución

Pivote = (3+5)/2=4

30

60	40	50

40	**60**	50

4ta. ejecución

Pivote = (4+5)/2=4

40

60	**50**

50	**60**

EJEMPLO: Divide y Vencerás
Ordenación rápida (Quicksort)

```
void ordenQuickSort(vector <int> & v, int izq, int der)
{
    int i{izq}, d{der}, pivote,aux;

    pivote = v.at((i+d)/2);

    while (i < d)
    {
       while (v.at(i) < pivote) { i++;  }

       while (pivote < v.at(d)) { d--;  }

       if (i <= d)  // intercambio de elementos
       {   aux = v.at(i);
           v.at(i) = v.at(d);
           v.at(d) = aux;
           i++;  d--; // se ajustan las posiciones
       }
    }

    if (izq < d) {  ordenQuickSort(v, izq, d);   }
    if (i < der) {  ordenQuickSort(v, i, der);   }
}
```

Análisis del mejor caso: Se produce cuando el pivote divide al vector en dos partes iguales y el orden de los elementos es aleatorio:

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

T(n/2) es el **coste de ordenar una de las mitades** y ***n*** el **número de comparaciones realizadas**.

Si desarrollamos la recurrencia se obtiene:

$$T(n) = n\, logn - n - 1$$

Que es de orden $T(n) \in \Omega(n\, logn)$

Análisis del mejor caso: Supongamos que el tamaño del vector es una potencia de 2: $n = 2^k \rightarrow log\, n = k$

Recorrido	Comparaciones
1º	(n-1) comparaciones * 1 vector 2 vectores de tamaño n/2
2º	(n/2) comparaciones * 2 vector = n 4 vectores de tamaño n/4
3º	(n/4) comparaciones * 4 vector = n 8 vectores de tamaño n/8
…	…
k-ésimo	$(n/2^{k-1})$ comparaciones * 2^{k-1} vector = n 2^k vectores de tamaño $n/2^k$

$$n + n + \cdots + n = kn = n\, log n$$

Luego es de orden $T(n) \in \Omega(n\, logn)$

Análisis del peor caso: Si se elige como pivote el **primer elemento del vector** y además se considera que el vector esta **ordenado** decrecientemente entonces, el **bucle se ejecutará** en total: $(n-1)+(n-2)+(n-3)+\cdots+1$

Cada miembro de este sumando proviene de cada una de las sucesivas **ordenaciones recursivas**. Este sumatorio da lugar a la siguiente expresión:

$$(n-1)+(n-2)+(n-3)+\cdots+1=$$

$$\sum_{i=1}^{n-1}(n-i)=\frac{[(n-1)+1](n-1)}{2}=\frac{n(n-1)}{2}$$

Que es de orden cuadrático $T(n)\in O(n^2)$

Emparejar tornillos y tuercas:

Se tienen **dos cajones**, uno con ***n tornillos*** de **varios tamaños**, y otro con sus correspondientes ***n tuercas***. Se necesita **emparejar cada tornillo con su tuerca**, pero en la habitación no se dispone de luz y, por tanto, **no es posible realizar la comparación visual**. La única posibilidad es tratar de **enroscar una determinada tuerca con un tornillo para determinar si es <u>grande</u>, <u>pequeña</u> o <u>se ajusta bien</u> al tornillo**.

Solución - Emparejar tornillos y tuercas:

Si el número de tuercas es **$n \leq 1$, caso base**.

Si **$n > 1$**:

- Se toma **un tornillo cualquiera** y, comparando con él, se divide el conjunto de **tuercas en las menores**, **las mayores y las que enroscan (igual tamaño)**.
- Utilizando **una tuerca de las que enroscan**, se divide el conjunto de **tornillos en menores, mayores y los que enroscan (igual tamaño)**.
- Se hacen las **llamadas recursivas con los conjuntos de tuercas y tornillos menores y mayores**.

Importante: la partición solo es correcta en el caso de que **todos los tornillos y tuercas sean de tamaños diferentes**, porque de otro modo puede que los tamaños de los conjuntos no coincidan

EJEMPLO 2: Divide y Vencerás

Emparejar tornillos y tuercas

```
// Telem será el tamaño de cada tornillo o tuerca
// Los índices izq y der son valores de los límites de cada partición
void emparejarTornilloTuerca (vector <Telem> &tornillos,
                              vector <Telem> &tuercas, int izq, int der)
{   int i, d, k, l;

    if (izq < der)
    {
       particionVector(tuercas, izq, der, tornillos.at(izq), i, d);
       particionVector(tornillos, izq, der, tuercas.at(i), k, l);

       // Los índice i == k y d == l
       emparejarTornilloTuerca(tornillos, tuercas, izq, i-1);    }
       emparejarTornilloTuerca(tornillos, tuercas, d+1, der);    }
}
```

EJEMPLO 2: Divide y Vencerás
Emparejar tornillos y tuercas

vector	<	=	?	>	
	izq	i	k	d	der

Los índices ***i***, ***k*** se mueven **hacia la derecha**, mientras el índice ***d*** se mueve **hacia la izquierda**.

En cada paso, el algoritmo compara ***V.at(k)*** con el elemento **pivote**:

- Si ***V.at(k) es menor***, se **intercambia con V.at(i) para que se agrupe con los menores**. Además, con el intercambio, en V.at(k) se coloca un elemento igual al pivote. Luego, se intercambian los índices i y k.
- Si **V.at(k) es igual**, no hay cambios.
- Si **V.at(k) es mayor**, se **intercambia con V.at(d) para que se agrupe con los mayores**. Ahora, con el intercambio, en V.at(k) se coloca un elemento desconocido. Luego, solo se puede avanzar, decrementando el índice ***d***.

El bucle termina cuando k > d, es decir, ya no hay elementos mal colocados.

EJEMPLO 2: Divide y Vencerás

Emparejar tornillos y tuercas

```
// Telem será un tornillo o una tuerca concreta
// Los índices i y j son valores de salida para definir los límites de la partición
void particionVector (vector <Telem> &V, int izq, int der, Telem pivote, int &i, int &d) {
   int k{izq};
   Telem aux;
   i = izq; d = der;

    while (k <= d)
    {  // posible intercambio de elementos y ajuste de posiciones
       if (V.at(k) < pivote) {
           aux = V.at(k);
           V.at(k) = V.at(i);
           V.at(i) = aux;
           i++;  k++;
       } else if (V.at(k) == pivote) {
           k++;
       } if (V.at(k) > pivote) {
           aux = V.at(k);
           V.at(k) = V.at(d);
           V.at(d) = aux;
           d--;
       }
    }
}
```