UNIVERSIDAD DE GRANADA

DECSAI

Algorítmica

Grado en Ingeniería Informática

Tema 2 – Algoritmos Divide y Vencerás

Manuel Pegalajar Cuéllar

manupc@ugr.es

Departamento de Ciencias de la Computación e Inteligencia Artificial

http://decsai.ugr.es

Objetivos del tema

- Plantearse la búsqueda de varias soluciones distintas para un mismo problema y evaluar la bondad de cada una de ellas.
- Tomar conciencia de la importancia del análisis de la eficiencia de un algoritmo como paso previo a su implementación en un lenguaje de programación.
- Saber realizar el análisis de eficiencia de un algoritmo, tanto a nivel teórico como empírico, y saber contrastar resultados experimentales con los teóricos.
- Comprender la técnica de resolución de un problema por división en problemas más pequeños.
- Conocer y saber aplicar los esquemas básicos de los algoritmos divide y vencerás.
- Conocer los criterios de aplicación de cada una de las distintas técnicas de diseño de algoritmos.

Estudia este tema en...

- G. Brassard, P. Bratley, “Fundamentos de Algoritmia”, Prentice Hall, 1997, pp. 247-290
- J.L. Verdegay: Lecciones de Algorítmica. Editorial Técnica AVICAM (2017).

Anotación sobre estas diapositivas:

El contenido de estas diapositivas es esquemático y representa un apoyo para las clases presenciales teóricas. No se considera un sustituto para apuntes de la asignatura.

Se recomienda al alumno completar estas diapositivas con notas/apuntes propios, tomados en clase y/o desde la bibliografía principal de la asignatura.

UNIVERSIDAD DE GRANADA

DECSAI

Algoritmos Divide y Vencerás

1. Introducción: DyV en términos de eficiencia.
2. La técnica Divide y Vencerás.
3. El problema del umbral.
4. Búsqueda Binaria.
5. Ordenación rápida.
6. El problema de selección.
7. Multiplicación rápida de enteros largos.
8. Multiplicación rápida de matrices.
9. La línea del horizonte.

¿Cuándo se puede aplicar Divide y Vencerás?

- Cuando el caso de un problema es “muy grande”, en ocasiones puede **dividirse en casos más pequeños** que sean más fáciles de resolver, con el fin de mejorar la eficiencia y resolver el problema más rápidamente.
- La solución al caso inicial (el “muy grande”) puede construirse después **combinando las soluciones de los casos más pequeños** en los que se ha dividido.
- Debe existir un **caso base** indivisible o, en su defecto, un algoritmo básico que resuelva el problema.
- ¿Es este método efectivo, en términos de eficiencia, para diseñar algoritmos? Lo veremos con un ejemplo: Ordenación por mergesort.

Algoritmo MergeSort

- **Existen métodos básicos** de ordenación: Burbuja, Inserción, Selección… Que son $O(n^2)$
- Podríamos pensar en ordenar un vector de tamaño **n** "por partes"; es decir, en lugar de resolver el problema de ordenación completo, dividir el vector en dos subvectores de tamaño **n/2**, que se ordenan de forma independiente.
- Combinaríamos los dos subvectores de tamaño **n/2** ordenados para generar el vector ordenado del tamaño inicial **n**.
- Podríamos aplicar este procedimiento recursivamente, hasta llegar a un caso base.Por ahora, supondremos caso base cuando n<=1 (vector de una componente como mucho, que ya está ordenado).

¿Sería esta estrategia más eficiente que ordenar el vector completo directamente mediante los métodos básicos tradicionales? Lo estudiaremos asintóticamente con el orden $O(\cdot)$.

Algoritmo MergeSort

```
1  void MergeSort(int *v, int ini, int fin) {
2
3      // Cuando ini>=fin es el caso base, donde v ya esta ordenado
4      // Solo dividimos vector cuando tiene más de 1 componente
5      if (ini < fin) {
6          int med = (ini + fin)/2; // Partimos vector en 2
7          MergeSort(v, ini, med); // Ordenamos parte izq
8          MergeSort(v, med + 1, fin); // Ordenamos parte dcha
9          combina(v, ini, med, fin); // Combinamos soluciones
10     }
11 }
12
```

Su eficiencia es $T(n)= 2*T(n/2) + T_{combina}(n)$

Algoritmo MergeSort – Función Combina

```
1  void combina(int *v, int ini, int med, int fin) {
2      int vtam= fin - ini + 1;
3      int *aux= new int[vtam];
4      int i = ini, j= med+1, k=0;
5
6       // Copiar el mínimo elem de los vectores
7       while (i <= med && j <= fin) {
8          if (v[i] < v[j]) {  aux[k]= v[i]; i++; }
9          else { aux[k]= v[j]; j++; }
10         k++;
11     }
12
13     // Si subvector 1 más grande que el 2, copiamos 1 en aux
14     while (i <= med) { aux[k] = v[i]; i++; k++; }
15     // Si subvector 2 más grande que el 1, copiamos 2 en aux
16     while (j <= fin) { aux[k] = v[j]; j++; k++; }
17
18     // Devolvemos aux en v
19     for (int n = 0; n < vtam; ++n) v[ini+n]= aux[n];
20     delete [] aux;
21 }
```

La eficiencia de Combina es : $T_{combina}(n) = n$

Algoritmo MergeSort – Eficiencia

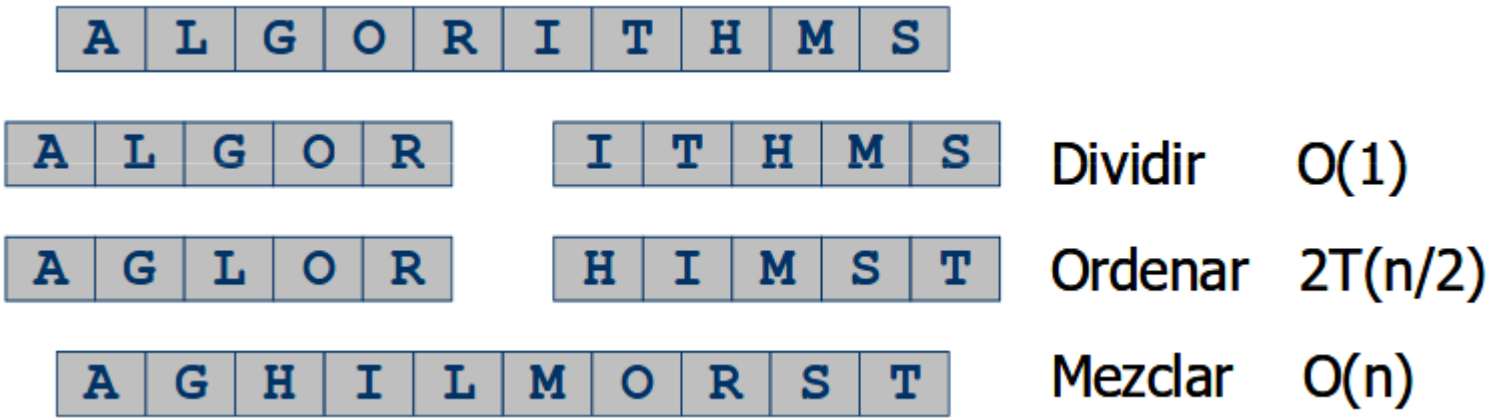
- **Eficiencia del algoritmo de ordenación MergeSort:**

$$T(n)= 2*T(n/2) + n$$

- Resolviendo la ecuación, **MergeSort es O(n*log(n))**.
- En comparación con los algoritmos básicos de **Inserción, Burbuja y Selección, que son O(n^2)**, DyV nos ha dado el método MergeSort con una mejor eficiencia.

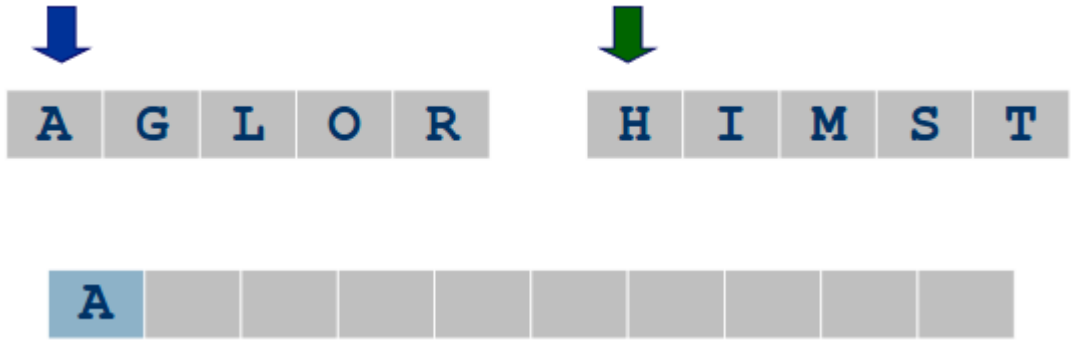
Algoritmo MergeSort – Cómo funciona

- **Divide el vector en 2 partes**
- **Sigue dividiendo (hasta un caso base)**
- **Resuelve cada parte por separado**
- **Combina las subsoluciones**
- **Devuelve la solución a la llamada recursiva anterior**



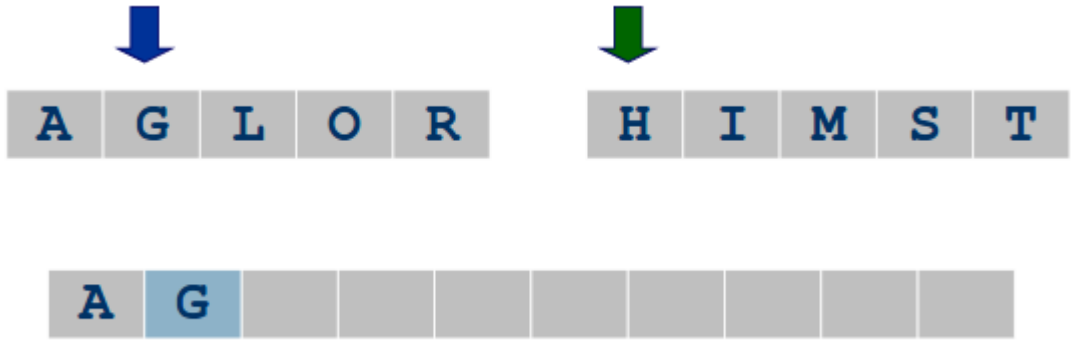
Algoritmo MergeSort – Cómo funciona la combinación

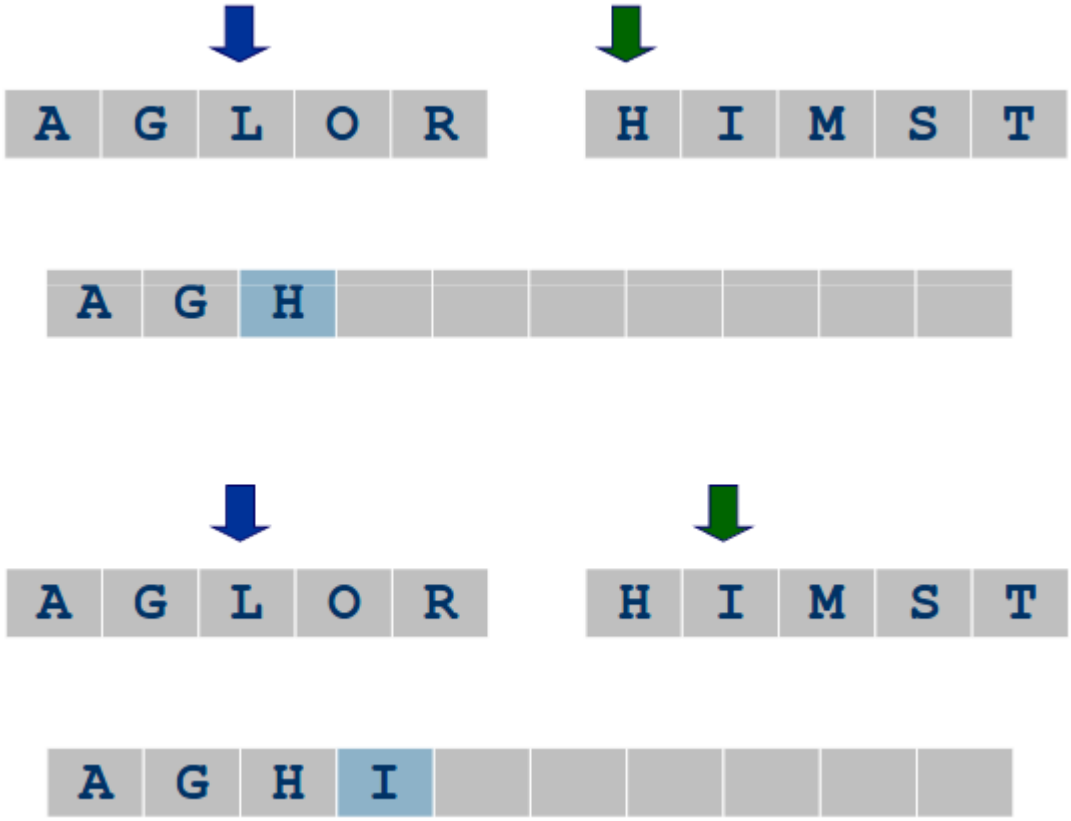
- Hay dos índices, uno al comienzo de cada subvector
- Se comprueba qué subvector tiene el elemento menor
- Se inserta ese elemento y se actualiza el índice afectado
- Al terminar un subvector, se copia el resto del otro subvector a la salida

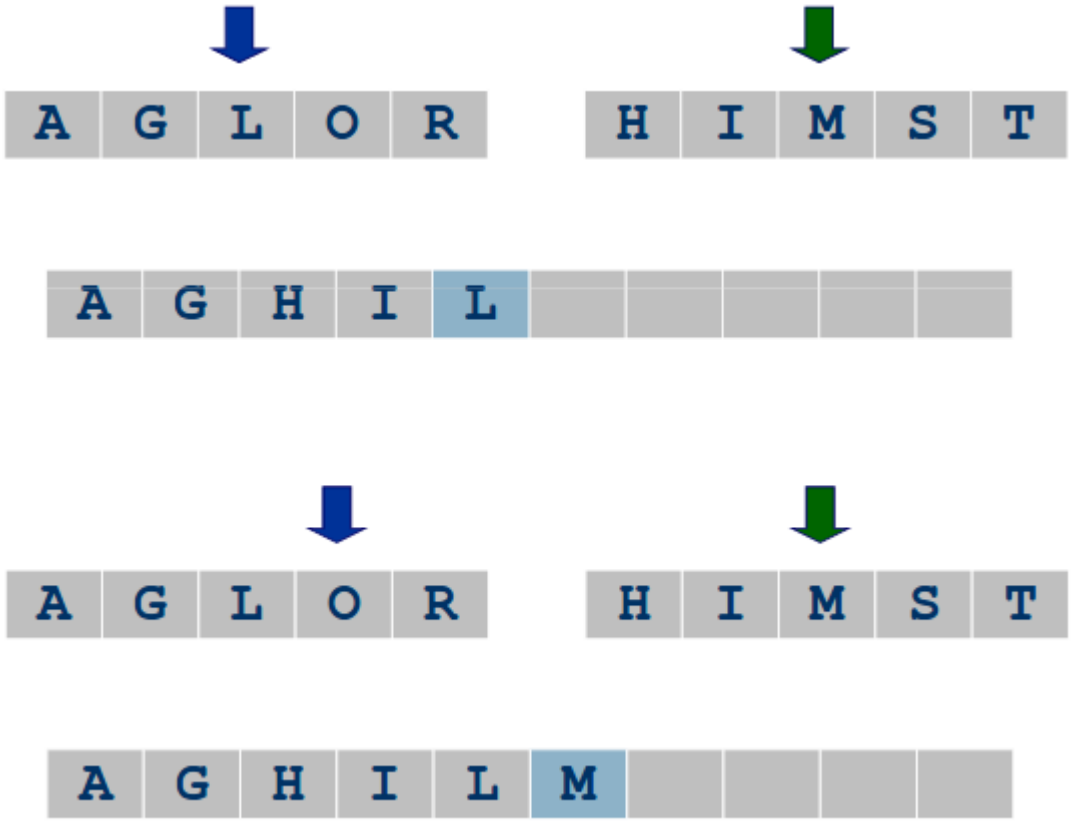


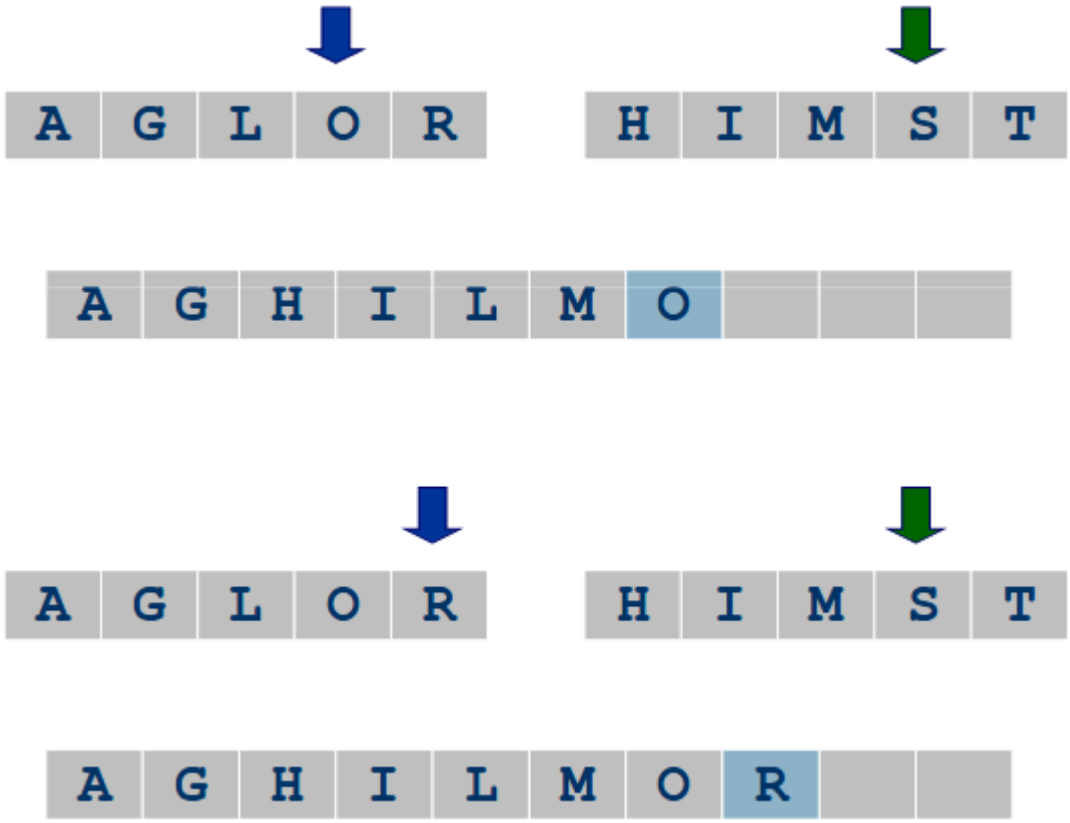
Algoritmo MergeSort – Cómo funciona la combinación

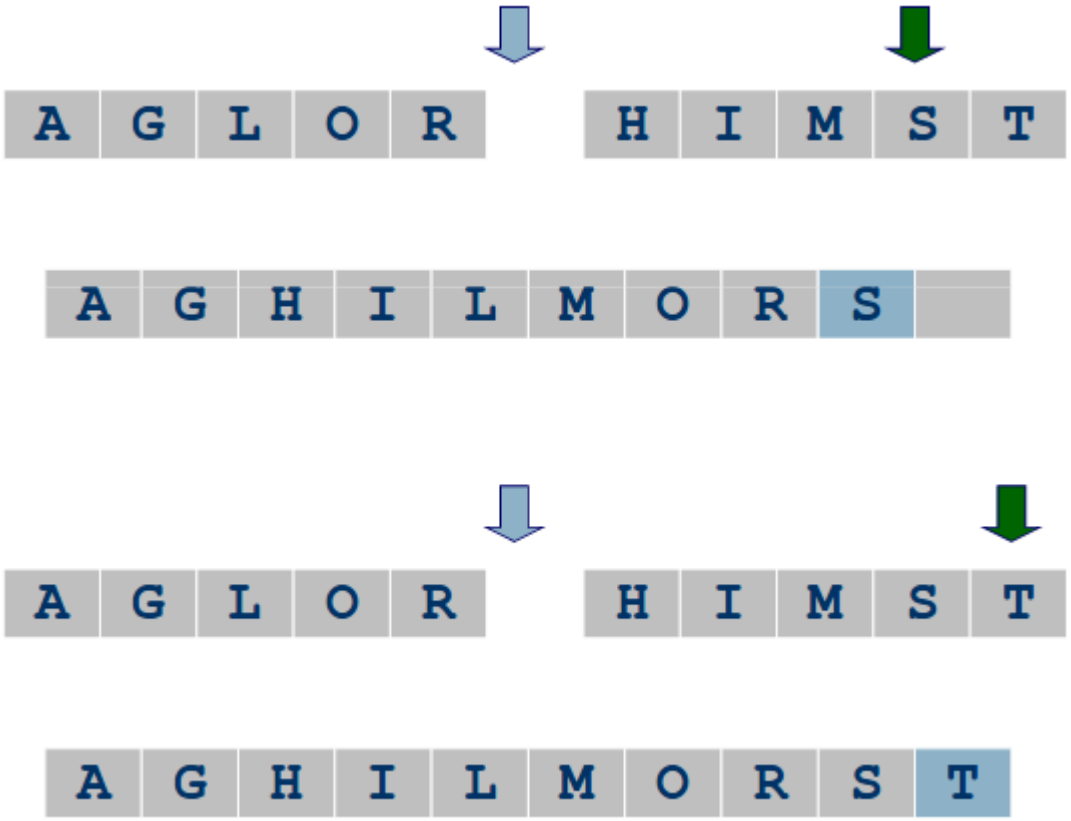
- **Hay dos índices, uno al comienzo de cada subvector**
- **Se comprueba qué subvector tiene el elemento menor**
- **Se inserta ese elemento y se actualiza el índice afectado**
- **Al terminar un subvector, se copia el resto del otro subvector a la salida**




A G L O R
H I M S T
A G H
A G L O R
H I M S T
A G H I


A G L O R
H I M S T
A G H I L
A G L O R
H I M S T
A G H I L M


A G L O R
H I M S T
A G H I L M O
A G L O R
H I M S T
A G H I L M O R


A G L O R H I M S T
A G H I L M O R S
A G L O R H I M S T
A G H I L M O R S T

UNIVERSIDAD DE GRANADA

DECSAI

Algoritmos Divide y Vencerás

1. Introducción: DyV en términos de eficiencia.
2. La técnica Divide y Vencerás.
3. El problema del umbral.
4. Búsqueda Binaria.
5. Ordenación rápida.
6. El problema de selección.
7. Multiplicación rápida de enteros largos.
8. Multiplicación rápida de matrices.
9. La línea del horizonte.

La idea general

La técnica Divide y Vencerás consiste en ir dividiendo de forma recursiva un problema “grande” en otros más pequeños, que se puedan resolver por separado.

Se dejará de dividir el problema recursivamente cuando se llegue a un caso base o cuando el problema sea ya indivisible.

Se resolverá cada subproblema por separado, y se combinarán las subsoluciones para dar lugar a la solución final del problema “grande” propuesto inicialmente.

Requisitos para poder aplicar DyV

- El caso del problema **debe poder dividirse** en uno o más casos equivalentes de tamaño menor, **independientes entre sí**, que puedan **resolverse por separado**.
- Las soluciones a los casos de tamaño menor **deben poder combinarse entre sí** para poder dar lugar a la solución del caso inicial.
- Debe existir, a priori, un **método básico** que resuelva el problema o, en su defecto, **un caso base indivisible** donde el problema esté resuelto.

Plantilla de algoritmo Divide y Vencerás

Función S= DyV(P, n)

Si "P es suficientemente pequeño", **entonces:**
Calcular Solucion= BASICO(P,n)

En otro caso, hacer:
Dividir P en k subcasos más pequeños P_1, P_2, P_3, ..., P_k, con tamaños lo más similares posible n_1, n_2, n_3, ..., n_k

Para cada subcaso generado i=1..k, **hacer:**
$SubSolucion_i$= DyV(P_i, n_i)

Calcular S= Combinar($SubSolucion_1$, $SubSolucion_2$, ..., $SubSolucion_k$)

Fin-En otro caso
Devolver S

P es el problema que se desea resolver, de tamaño **n**. **S** es la solución al problema.

Ejemplo: Problema de elevar un número real a un entero

- Sea un **número real *r*** y un **número entero *n***. Se desea diseñar un algoritmo que realice la operación de elevar ***r*** a ***n***: $\mathbf{r^n}$
- Existe un método básico que resuelve el problema (basta con multiplicar ***r*** un total de ***n*** veces:

$$\mathbf{r^n = r \cdot r \cdot \ldots \cdot r \text{ (n veces)}}$$

- La eficiencia de este método básico es **O(n)**

```
 6   double ElevarBasico(double r, int n) {
 7
 8       double aux= 1;
 9       for (int i= 0; i<n; i++)
10           aux*= r;
11
12       return aux;
13   }
```

Ejemplo: Problema de elevar un número real a un entero

- ¿Podríamos aplicar la estrategia DyV para mejorar la eficiencia y reducir el número de operaciones?
  - El problema **debe poder dividirse** en uno o más casos equivalentes de tamaño menor, **independientes entre sí**, que puedan **resolverse por separado**
- ***En este caso, podríamos dividir el problema en dos de tamaño parecido (n/2), independientes y que pueden resolverse por separado, sacando partido de las propiedades de las potencias, haciendo:***

$$r^n = \begin{cases} r^{\lfloor n/2 \rfloor} \cdot r^{\lfloor n/2 \rfloor} & n = \text{par} \\ r \cdot r^{\lfloor n/2 \rfloor} \cdot r^{\lfloor n/2 \rfloor} & n = \text{impar} \end{cases}$$

Ejemplo: Problema de elevar un número real a un entero

- ¿Podríamos aplicar la estrategia DyV para mejorar la eficiencia y reducir el número de operaciones?
  - Las soluciones a los casos de tamaño menor **deben poder combinarse entre sí** para poder dar lugar a la solución del caso inicial.
- ***De hecho, la combinación es la multiplicación de la subsolución de calcular $r^{n/2}$ por la subsolución de calcular $r^{n/2}$:***

$$r^n = \begin{cases} r^{\lfloor n/2 \rfloor} \cdot r^{\lfloor n/2 \rfloor} & n = \text{par} \\ r \cdot r^{\lfloor n/2 \rfloor} \cdot r^{\lfloor n/2 \rfloor} & n = \text{impar} \end{cases}$$

Ejemplo: Problema de elevar un número real a un entero

- ¿Podríamos aplicar la estrategia DyV para mejorar la eficiencia y reducir el número de operaciones?
  - Debe existir, a priori, un **método básico** que resuelva el problema o, en su defecto, **un caso base indivisible** donde el problema esté resuelto.
- ***En nuestro caso existen ambos: el método básico de las diapositivas anteriores, y los casos base $r^0=1$ y $r^1=r$:***

$$r^n = \begin{cases} r^{\lfloor n/2 \rfloor} \cdot r^{\lfloor n/2 \rfloor} & n = \text{par}; n > 1 \\ r \cdot r^{\lfloor n/2 \rfloor} \cdot r^{\lfloor n/2 \rfloor} & n = \text{impar}; n > 1 \\ 1 & n = 0 \\ r & n = 1 \end{cases}$$

Ejemplo: Problema de elevar un número real a un entero

- **Diseño de nuestro algoritmo:**
  - **Caso base:** Si n<=1 **->** Solución = r^n
  - **División del problema en subproblemas:** El problema original r^n se divide en 1 subproblema de tamaño n/2: Calcular $r^{n/2}$.
  - **Método de combinación:** Sea $\mathbf{s_1}$ la subsolución del primer subproblema. La solución **S** al problema general se calcula combinando la subsolución de la siguiente forma:
    - **$S = s_1 \cdot s_1$, en caso de que n sea par**
    - **$S = r \cdot s_1 \cdot s_1$, en caso de que n sea impar**

Ejemplo: Problema de elevar un número real a un entero

- **Diseño de nuestro algoritmo:**
  - **Adaptación de la plantilla DyV al problema:**

Función S= DyV(r, n)

Si r=0, **S=1;**
En otro caso, **Si r=1**, **S=r;**
En otro caso, hacer:

s1= DyV(r, floor(n/2)); // floor: Redondea al entero inferior
Si n es par,
S= s1*s1
En otro caso
S= r*s1*s1
Fin-en otro caso
Devolver S

Ejemplo: Problema de elevar un número real a un entero

- **Implementación de nuestro algoritmo:**

```
16  double ElevarDyV(double r, int n) {
17
18      if (n==0) return 1;
19      else if (n==1) return r;
20      else {
21          double subsolucion= ElevarDyV(r, n/2);
22          if (n%2 == 0)
23              return subsolucion*subsolucion;
24          else return r*subsolucion*subsolucion;
25      }
26  }
```

- La ecuación en recurrencias es **T(n)= T(n/2)+1**
- La eficiencia del algoritmo es **O(log(n))**

Algoritmos Divide y Vencerás

1. Introducción: DyV en términos de eficiencia.
2. La técnica Divide y Vencerás.
3. El problema del umbral.
4. Búsqueda Binaria.
5. Ordenación rápida.
6. El problema de selección.
7. Multiplicación rápida de enteros largos.
8. Multiplicación rápida de matrices.
9. La línea del horizonte.

Plantilla de algoritmo Divide y Vencerás

Función S= DyV(P, n)

Si "P es suficientemente pequeño", **entonces:**
Calcular Solucion= BASICO(P,n)

En otro caso, hacer:
Dividir P en k subcasos más pequeños P_1, P_2, P_3, …, P_k, con tamaños lo más similares posible n_1, n_2, n_3, …, n_k

Para cada subcaso generado i=1..k, **hacer:**
$SubSolucion_i$= DyV(P_i, n_i)

Calcular S= Combinar($SubSolucion_1$, $SubSolucion_2$, …, $SubSolucion_k$)

Fin-En otro caso
Devolver S-

¿Qué significa "P es suficientemente pequeño" ?

- Si el tamaño del caso inicial es muy grande, hacer muchas divisiones en una llamada recursiva puede ser muy costoso: Tan malo es tener un caso de problema enorme como millones y millones de problemas pequeñitos.

- ¿Cuándo dejar de hacer divisiones y resolver el problema por otro método menos costoso? **Solución:** Cuando *"P sea suficientemente pequeño"*

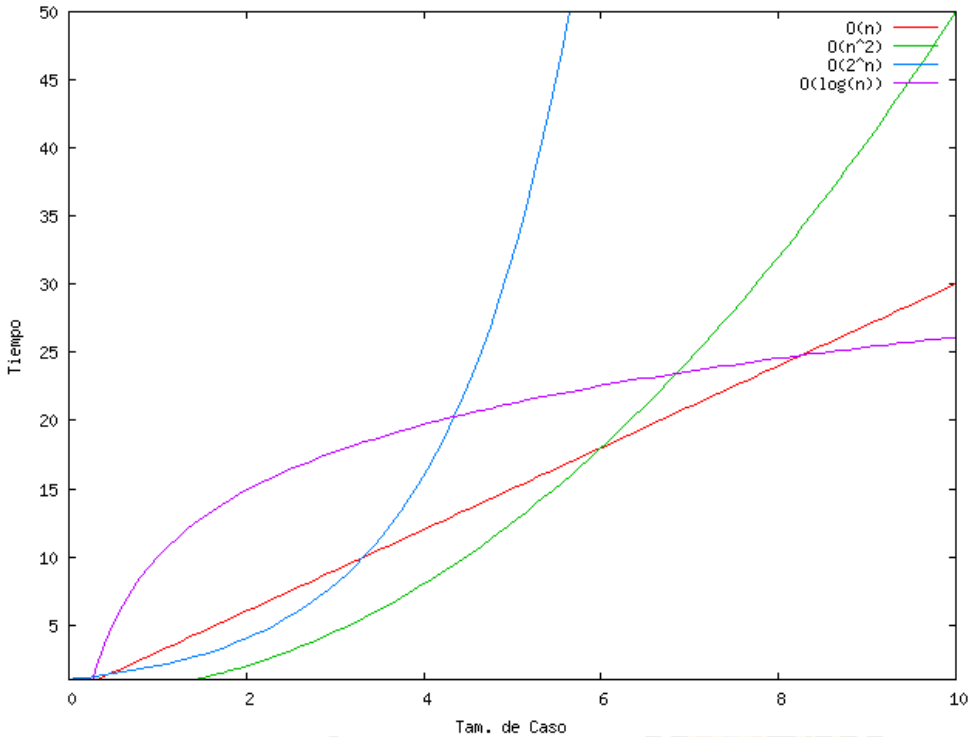
- ¿Cómo saber cuándo P es *suficientemente pequeño*? **Comparando los tiempos de ejecución del algoritmo DyV diseñado y el algoritmo básico.**

Ejemplo de caso extremo de selección de algoritmos. Volviendo al tema 1 (órdenes de eficiencia):

Para n<=6, el algoritmo $O(n^2)$ es mejor.

Para 6<n<=8, el algoritmo O(n) es mejor.

Para n>8, el algoritmo O(log(n)) es mejor.



Ejemplo: Mergesort y algoritmo base Inserción

- Nuestro algoritmo base para el problema de ordenación será el algoritmo de ordenación por inserción, con eficiencia $O(n^2)$.
- Para determinar cuándo el problema de ordenación es suficientemente pequeño como para dejar de aplicar DyV, se ha calculado empíricamente el tiempo de ejecución de los algoritmos básico y DyV:
  - Se ejecuta Inserción para problemas de diferente tamaño (ejemplo n=1000, 2000, 3000, 4000, 5000), y se mide su tiempo de ejecución.
  - Se calcula el valor de la constante oculta $T_1(n) <= K_1*n^2$
  - Se hace lo mismo para MergeSort: $T_2(n) <= K_2*n*\log_2(n)$
  - Se despeja n de la igualdad $K_1*n^2 = K_2*n*\log_2(n)$

Tiempos Inserción

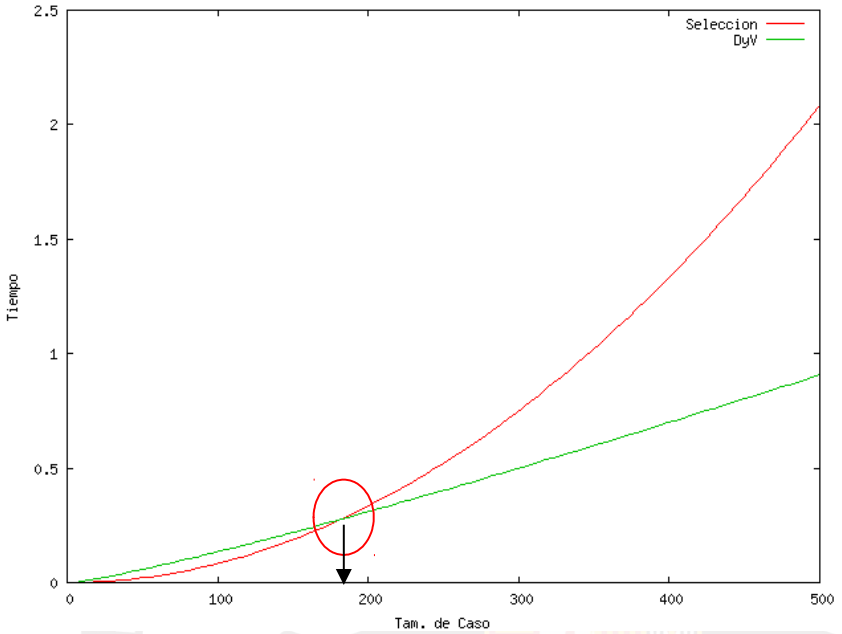
n	t
1000	15
2000	31
3000	63
4000	125
5000	172
6000	250
7000	344

Tiempos MergeSort

n	t
10000	16
20000	31
30000	31
40000	63
50000	62
60000	62
70000	78

$K_1 = 8{,}34 \cdot 10^{-6}$

$K_2 = 2{,}92 \cdot 10^{-4}$



¿En qué punto MergeSort comienza a ser mejor que Inserción?

$K_1 * n^2 = K_2 * n * \log_2(n)$

n se puede calcular gráficamente o de forma aproximada si no se dispone de métodos matemáticos para ello

Ejemplo: Mergesort y algoritmo base Inserción

- Implementación del algoritmo **MergeSort** tras resolver el problema del umbral:

```
 3  void MergeSort(int *v, int ini, int fin) {
 4
 5      if (fin-ini<190) {
 6
 7          Insercion(v, ini, fin);
 8
 9      } else {
10
11          int med= (ini+fin)/2;
12          Mergesort(v, ini, med);
13          Mergesort(v, med+1, fin);
14          Combina(v, ini, med, fin);
15      }
16  }
```

Algoritmos Divide y Vencerás

1. Introducción: DyV en términos de eficiencia.
2. La técnica Divide y Vencerás.
3. El problema del umbral.
4. Búsqueda Binaria.
5. Ordenación rápida.
6. El problema de selección.
7. Multiplicación rápida de enteros largos.
8. Multiplicación rápida de matrices.
9. La línea del horizonte.

Enunciado del problema

Sea **v** un vector de tamaño **n**. El problema consiste en buscan un elemento **x** dentro del vector, y devolver su posición.

Método básico

- Recorrer el vector comprobando el valor de cada componente, y devolver el índice de **x** cuando se encuentre.

```
37  int BuscarBasico(double *v, const int ini, const int fin,
38                   const double x) {
39
40      for (int i= ini; i<=fin; i++)
41          if (v[i] == x) return i;
42      return -1;
43  }
```

- Eficiencia: **O(n)**

Búsqueda binaria: Idea general

Sea **v** un vector de tamaño **n**, **ordenado ascendentemente**. El problema consiste en buscan un elemento **x** dentro del vector, y devolver su posición.

Si el vector está ordenado, podemos comprobar el valor de su elemento central. Si es mayor que el elemento **x** buscado, podemos descartar la mitad derecha del vector.

En otro caso, si es menor que el elemento **x** buscado, podemos descartar la mitad izquierda del vector.

Diseño: 1. División del problema en subproblemas

- Para buscar un elemento **x** en un vector **v** de tamaño **n**, el cual ya se encuentra previamente ordenado, podríamos dividir el vector en uno de tamaño la mitad (n/2), independiente y que puede resolverse por separado.
- Para dividir el vector, seleccionaremos su posición central.

Diseño: 2. Resolución de cada subproblema y combinación.

- Si el elemento central del vector contiene el elemento requerido, se devuelve su índice.
- Si **x** es menor que el elemento central, reduciremos la búsqueda a la primera mitad del vector y lo resolveremos recursivamente. Si es mayor, la búsqueda la reduciremos a la segunda mitad del vector.
- No hace falta combinar soluciones, puesto que el problema se reduce a un problema de tamaño más reducido.

Diseño: 3. Caso base

- Se parará la recursividad cuando el vector a ordenar tenga tamaño 0 (el elemento no se encuentra) o cuando se encuentre el elemento en el centro del vector.

Diseño: 4. Adaptación de la plantilla DyV

Función S= BB(V, ini, fin, x)

Si ini>fin **devolver** S="No hay solución";
En otro caso, hacer:
centro= (ini+fin)/2;
Si V[centro]=x **devolver** S=centro;
En otro caso, Si x<V[centro] **devolver** S=BB(V, ini, centro-1, x)
En otro caso, Si x>V[centro] **devolver** S=BB(V, centro+1, fin, x)
Fin-En otro caso

Implementación: Proyecto Code::Blocks *BusquedaBinaria*

```
47  int BuscarBinario(double *v, const int ini, const int fin,
48                    const double x) {
49      int centro;
50      if (ini>fin) return -1;
51
52      centro= (ini+fin)/2;
53      if (v[centro] == x) return centro;
54      if (v[centro]>x) return BuscarBinario(v, ini, centro-1, x);
55      return BuscarBinario(v, centro+1, fin, x);
56  }
```

- **Ecuación en recurrencias:**

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

- Eficiencia: **O(log2(n))**. **Se mejora con respecto al método básico.**

UNIVERSIDAD DE GRANADA

DECSAI

Algoritmos Divide y Vencerás

La idea general

El algoritmo QuickSort (Ordenación Rápida) es el mejor algoritmo DyV de ordenación, en caso promedio.

La idea básica de QuickSort, para ordenar un vector **v** de tamaño **n** es la siguiente:

- Determinar un elemento “pivote”, para dividir el vector en 2 partes con elementos <= pivote y > que pivote, respectivamente.
- Ordenar los dos subvectores generados.
- Combinar las dos soluciones para obtener **v** ordenado.

Diseño: 1. División del problema en subproblemas

- Para ordenar un vector **v** de tamaño **n**, podríamos dividir el vector en dos de tamaño parecido (n/2), independientes y que pueden resolverse por separado.
- Para dividir el vector, seleccionaremos un elemento al que llamaremos **pivote**, de modo que los elementos menores que el pivote queden a la izquierda del vector, y los mayores o iguales a la derecha.
- En un caso óptimo, el pivote dividirá al vector en dos mitades iguales. En el peor caso, dejará un subvector con **1 elemento** y otro subvector con **n-1 elementos**.

Diseño: 2. Resolución de cada subproblema y combinación.

- Se reordenarán los dos subvectores. Al llegar a un caso base de 1 elemento, el vector estará ordenado.
- La penúltima llamada recursiva (vector de 2 elementos) también hará que el vector resultante esté ordenado, porque hemos pivotado y el elemento mayor estará en la segunda posición, mientras que el elemento menor estará en la primera.
- En las llamadas recursivas anteriores, pasará lo mismo gracias a que se va pivotando cada vez que se ejecuta la función.
- Por tanto, no se requiere combinación adicional para QuickSort: Los métodos de pivotaje y las llamadas recursivas lo aplican inherentemente.

Diseño: 3. Caso base.

- Se parará cuando el vector a ordenar tenga tamaño 1.

Diseño: 4. Adaptación de la plantilla DyV

Función S= QS(V, ini, fin)

Si ini>=fin devolver S=vacío;

En otro caso, hacer:
[PosPivote, V]= Pivotar(V, ini, fin);
V= QS(V, ini, PosPivote);
V= QS(V, PosPivote+1, fin);
Fin-En otro caso
Devolver V

Diseño: 4. Método Pivotar

Función [PosPivote, V]= QS(V, ini, fin)

```
   Pivote= V[ini];
   i= ini+1; j= fin;
   Mientras (i<=j), Hacer:
      Intercambiar V[i] y V[j]
      Mientras (V[i]<Pivote Y i<=j) Hacer i= i+1;
      Mientras (V[j]>=Pivote Y i<=j) Hacer j= j-1;
   Fin-Mientras
   Si (ini<j)
      Intercambiar V[ini] y V[j];
Devolver [j, V]
```

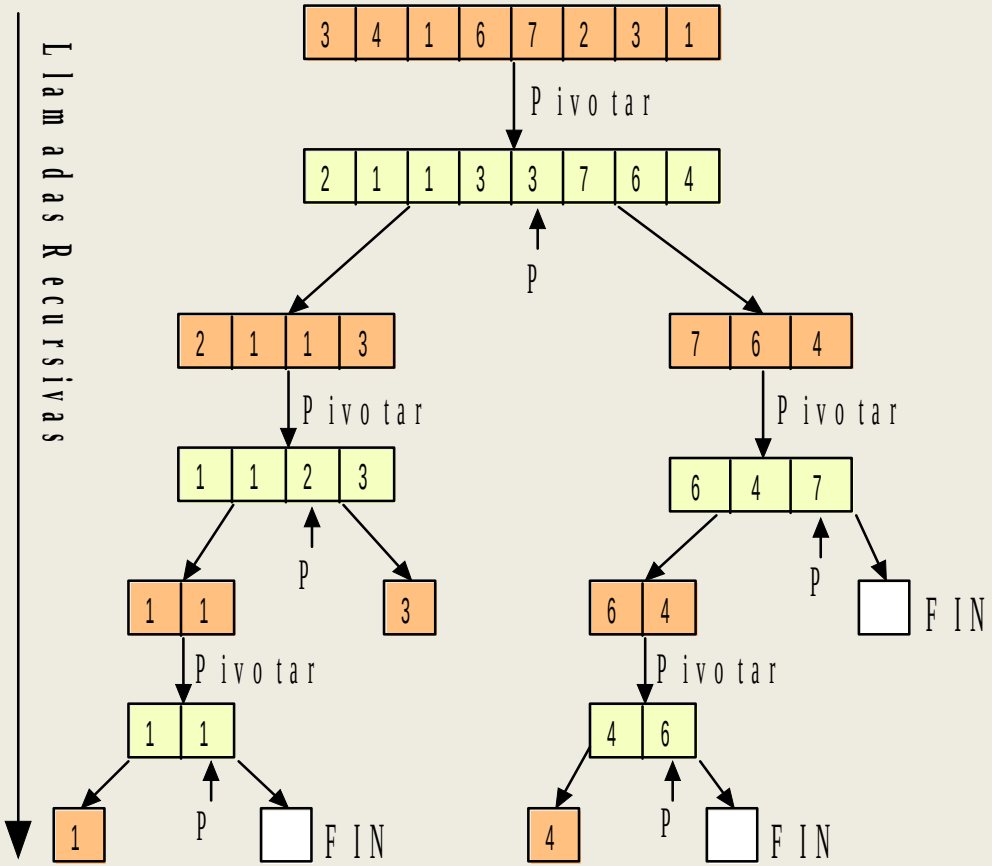
Implementación: Proyecto Code::Blocks *QuickSort*, Tema 2

```
void QuickSort(double *v, const int ini, const int fin) {
    if (ini<fin) {

        int posPivote= pivotar(v, ini, fin);
        QuickSort(v, ini, posPivote);
        QuickSort(v, posPivote+1, fin);
    }
}
```

```
 6  int pivotar(double *v, const int ini, const int fin) {
 7
 8      double pivote= v[ini], aux;
 9      int i= ini+1, j= fin;
10
11
12      while (i<=j) {
13          while (v[i]<pivote && i<=j) i++;
14          while (v[j]>=pivote && j>=i) j--;
15
16          if (i<j) {
17              aux= v[i]; v[i]= v[j]; v[j]= aux;
18          }
19      }
20
21
22      if (j>ini) {
23          v[ini]= v[j];
24          v[j]= pivote;
25      }
26      return j;
27  }
```

Ejemplo: QuickSort. Ordenar (3, 4, 1, 6, 7, 2, 3, 1)



Eficiencia de QuickSort

- **Eficiencia de QuickSort en el peor de los casos:**

  $T(n)= T(n-1) + n$ **-> Eficiencia $O(n^2)$**

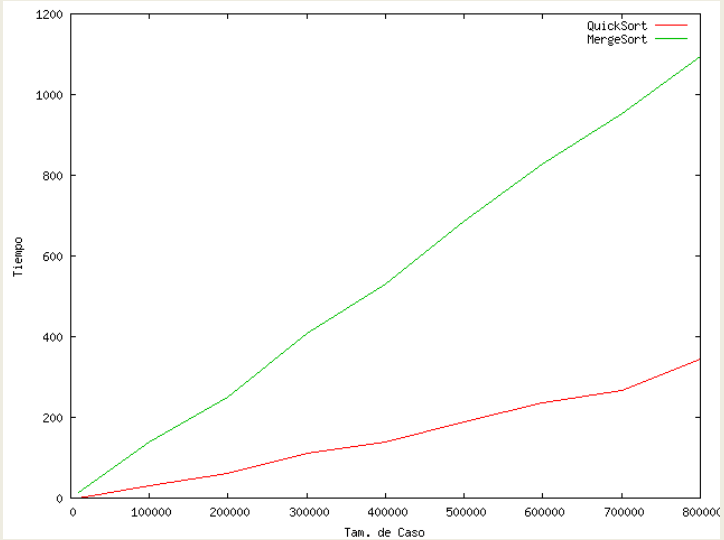
- **Eficiencia de QuickSort en el mejor de los casos:**

  $T(n)= 2 \cdot T(n/2) + n$ **-> Eficiencia $\Omega(n*\log(n))$**

- La realidad es que, en caso promedio, se comporta mejor que los demás algoritmos de ordenación, debido a que la probabilidad de que ocurra el peor caso es muy baja.

QuickSort vs MergeSort

Comparación de QuickSort y MergeSort:



UNIVERSIDAD DE GRANADA

Algoritmos Divide y Vencerás

1. Introducción: DyV en términos de eficiencia.
2. La técnica Divide y Vencerás.
3. El problema del umbral.
4. Búsqueda Binaria.
5. Ordenación rápida.
6. El problema de selección.
7. Multiplicación rápida de enteros largos.
8. Multiplicación rápida de matrices.
9. La línea del horizonte.

DECSAI

Definición del problema

Sea un vector **v** de **n** componentes. Se desea conocer qué elemento se situaría en la ***i-ésima posición*** del vector en caso de que este estuviese ordenado.

Ejemplo de utilidad: Cálculo de la mediana.

Método básico

Bastaría con ordenar el vector y devolver el elemento que hubiese en la i-ésima posición.

Complejidad del método: O(n·log(n)) , dado que la ordenación tiene esa eficiencia y es la operación más compleja que se realiza.

La idea general

- No necesitamos que el vector esté ordenado completamente. Basta con que el elemento de la i-ésima posición esté ordenado.

- Podemos reutilizar la **función pivotar de QuickSort** para resolver el problema dado que, al terminar su ejecución, en la j-ésima posición del vector se encuentra el pivote y todos los elementos a su izquierda son menores que él, por lo que se encuentra en su posición ordenada.

Diseño: 2. División del problema en subproblemas

- Para encontrar el elemento en la ***i-ésima*** posición del vector, colocaremos todos los elementos menores que un pivote a la izquierda del vector, y los elementos mayores o iguales en la parte derecha. La posición del pivote, **posPivote**, estará ordenada por tanto.
- Si la posición **i** que buscamos es la posición del pivote, hemos terminado.
- Si la posición **i** que buscamos es inferior a **posPivote**, entonces dividimos el problema en 1 subproblema (vector **v** desde la primera posición hasta **posPivote**).
- Si la posición **i** que buscamos es superior a **posPivote**, entonces dividimos el problema en 1 subproblema (vector **v** desde **pospivote+1** hasta **la última componente**).

Diseño: 3. Resolución de cada subproblema y combinación.

- Se volverá a calcular el pivote para el subvector generado, aplicando recursivamente el procedimiento hasta que se encuentre la i-ésima posición.
- Como la función de pivotar deja en **PosPivote** el elemento que estamos buscando, y sólo dividimos el problema en 1 subproblema, no es necesario realizar combinación ni operaciones adicionales.

Diseño: 4. Caso base.

- Se parará cuando **PosPivote** sea la componente **i** deseada.

Diseño: 5. Adaptación de la plantilla DyV

Función S= Seleccion(V, ini, fin, i)

Si ini==fin devolver S=V[ini];

En otro caso, hacer:
[PosPivote, V]= Pivotar(V, ini, fin);
Si (PosPivote == i) **devolver** V[PosPivote];
En otro caso,
Si (PosPivote>i) devolver Selección(V, ini, PosPivote-1, i);
En otro caso,
Si (PosPivote<i) devolver Selección(V, PosPivote+1, fin, i);
Fin-En otro caso

Implementación. Proyecto *Code::Blocks* Selección, Tema 2

```
30  double Seleccion(double *v, const int ini, const int fin, const int i) {
31
32      if (ini == fin) return v[ini];
33      int posPivote= pivotar(v, ini, fin);
34      if (posPivote == i) return v[posPivote];
35      if (posPivote > i) return Seleccion(v, ini, posPivote-1, i);
36      return Seleccion(v, posPivote+1, fin, i);
37  }
```

```
6   int pivotar(double *v, const int ini, const int fin) {
7
8       double pivote= v[ini], aux;
9       int i= ini+1, j= fin;
10
11      while (i<=j) {
12          while (v[i]<pivote && i<=j) i++;
13          while (v[j]>=pivote && j>=i) j--;
14
15          if (i<j) {
16              aux= v[i]; v[i]= v[j]; v[j]= aux;
17          }
18      }
19
20      if (j>ini) {
21          v[ini]= v[j]; v[j]= pivote;
22      }
23      return j;
24  }
```

Eficiencia de Seleccion

- En el peor de los casos, **Selección** divide al vector en dos partes, de tamaño 1 y **n-1** respectivamente. Por tanto, su eficiencia será:
  - T(n)= T(n-1)+n; **-> $O(n^2)$**
- En el mejor caso, **Selección** divide al vector en dos partes de igual tamaño **n/2**. Por tanto su eficiencia será:
  - T(n)= T(n/2)+n; **-> $\Omega(n)$**
- Al igual que ocurre con QuickSort, es muy improbable que se dé el peor caso, por lo que en promedio se comporta como **$\Omega(n)$**.

Algoritmos Divide y Vencerás

1. Introducción: DyV en términos de eficiencia.
2. La técnica Divide y Vencerás.
3. El problema del umbral.
4. Búsqueda Binaria.
5. Ordenación rápida.
6. El problema de selección.
7. Multiplicación rápida de enteros largos.
8. Multiplicación rápida de matrices.
9. La línea del horizonte.

Definición del problema

Sean dos números enteros positivos **A** y **B**, con **n** dígitos cada uno. Se desea calcular la multiplicación **C= A*B**.

Método básico

Existen multitud de métodos de multiplicación; por ejemplo, el método clásico:

```
   12583
 x   744
-------
   50332
  50332
 88081
-------
 9361752
```

Si un entero tiene menos dígitos que otro, se rellena con 0's por la izquierda.

Eficiencia: $\Theta(n^2)$. n^2 multiplicaciones + k*n sumas

La idea general

- Haciendo una descomposición de cada entero en sumas de coeficientes por potencias de 10, intentar dividir el problema en varios subproblemas más pequeños.

$$A= a_0 +10*a_1+100*a_2+\ldots+10^n*a_n$$

$$B= b_0 +10*b_1+100*b_2+\ldots+10^n*b_n$$

- Ejemplo:

$$1234= 4*10^0 + 3*10^1 + 2*10^2 + 1*10^3$$

Primer Diseño:

- ¿Podríamos dividir un número entero de **n** cifras en 2 enteros?

$$A= a_0 +10^{n/2}*a_1 \quad ; \quad B= b_0 +10^{n/2}*b_1$$

- Ejemplo: $1234= 34*10^0 + 12*10^2$
- ¿Cómo se multiplicarían?

$$A*B= (a_0 +10^{n/2}*a_1) * (b_0 +10^{n/2}*b_1)=$$

$$= a_0 b_0 + a_0*10^{n/2}*b_1 + b_0*10^{n/2}*a_1 + 10^{n/2}*a_1*10^{n/2}*b_1$$

$$= a_0b_0 + 10^{n/2} (a_0 b_1 + b_0 a_1)+ 10^n a_1b_1$$

- **Tenemos que realizar 4 multiplicaciones de números de tamaño n/2 y 3 sumas -> Eficiencia $O(n^2)$. No ahorramos nada.**

Segundo Diseño:

- Consideremos ahora las propiedades de la suma y el producto:

$$r= (a_0+a_1)*(b_0+b_1)= a_0b_0+a_0b_1+a_1b_0+a_1b_1$$

- Las multiplicaciones a_1b_1 y a_2b_2 debemos calcularlas forzosamente para la fórmula original, pero también podemos calcular r como el producto $r= (a_0+a_1)*(b_0+b_1)$, luego calcular a_1b_1 y a_2b_2, y calcular $(a_1b_0+a_0b_1)$ como la diferencia de ambos;

$$(a_1b_0+a_0b_1)= r- a_0b_0 – a_1b_1$$

- **Con esta estrategia, realizamos 3 multiplicaciones de números de tamaño n/2, 2 sumas y 2 restas -> Podemos reducir el número de operaciones.**

Diseño: 1. División del problema en subproblemas

- Dividiremos cada entero **A** y **B**, de tamaño **n,** en dos enteros de tamaño **n/2** cada uno:

$$A= a_0 +10^{n/2}*a_1 \quad ; \quad B= b_0 +10^{n/2}*b_1$$

- **Dividimos el problema en 3 subproblemas:**
  - **P1= $(a_0+a_1)*(b_0+b_1)$**
  - **P2= a_1*b_1**
  - **P3= a_0*b_0**

Diseño: 2. Resolución de cada subproblema y combinación.

$$\mathbf{P1= (a_0+a_1)*(b_0+b_1)\ \ ;\ \ P2= a_1*b_1\ \ ;\ \ P3= a_0*b_0}$$

- Los subproblemas **P1, P2, P3** son independientes y se resolverán por separado. Llamemos **S1, S2, S3** a las subsoluciones de cada uno de los subproblemas.
- Combinaremos las subsoluciones de la siguiente forma para llegar a la solución global **S**:

  Parcial= S1-S2-S3 (el cálculo de $\mathbf{(a_1b_0+a_0b_1)}$)

  $\mathbf{S= 10^n*S2+10^{n/2}*Parcial+S3}$

- Ejemplo: A= 1234 = $a_0 + 10^{n/2}*a_1$ == $34+12*10^2$

  B= 5678 = $b_0 + 10^{n/2}*b_1$ == $78+56*10^2$

S1= (12+34)*(56+78)= 6164 ; S2= 12*56=672; S3= 34*78= 2652

Parcial= 2840; S= 10^4*S2+10^2*Parcial+S3= 7006652

Diseño: 3. Caso base.

- Se parará cuando **n=1, multiplicación de números de 1 dígito.**

Diseño: 4. Adaptación de la plantilla DyV

Función S= Multiplica(A, B)

Si n =1, **entonces** S=A*B;

En otro caso, hacer:
Reescribir A=(a_0,a_1), B=(b_0, b_1)
S_3= Multiplica(a_0, b_0)
S_2= Multiplica(a_1, b_1)
S_1= Multiplica(a_0+a_1, b_0+b_1)
Parcial= S_1 -S_2-S_3;
S= $10^n S_2$+$10^{n/2}$Parcial+S_3
Fin-En otro caso
Devolver S

Eficiencia

- Reescribir los números $A=(a_0,a_1)$, $B=(b_0, b_1)$ es O(1)
- El algoritmo hace 3 llamadas recursivas para resolver el problema de tamaño n/2.
- Suponemos que las sumas y restas podrían ser implementadas como O(n).
- En este caso, la ecuación en recurrencias es:

$$T(n)= 3T(n/2)+n$$

- **Resolviendo la ecuación, obtenemos que la eficiencia del método DyV es $O(n^{\log_2(3)})$, mejor que el método básico.**

UNIVERSIDAD DE GRANADA

Algoritmos Divide y Vencerás

1. Introducción: DyV en términos de eficiencia.
2. La técnica Divide y Vencerás.
3. El problema del umbral.
4. Búsqueda Binaria.
5. Ordenación rápida.
6. El problema de selección.
7. Multiplicación rápida de enteros largos.
8. Multiplicación rápida de matrices.
9. La línea del horizonte.

DECSAI

Definición del problema

Sean dos matrices cuadradas **A** y **B** de tamaño **n*n**. Se desea calcular la multiplicación **C= A*B**.

Método básico

$$A_{n,n} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \quad A_{n,n} * B_{n,n} = C_{n,n} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix} \quad c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$B_{n,n} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix}$$

- El algoritmo clásico tiene que recorrer las **n*n** filas de C, calculando c_{ij} como la suma de **n** multiplicaciones.
- Eficiencia del algoritmo clásico: **O(n³)**

La idea general

- Haciendo una descomposición de cada matriz en submatrices que se puedan multiplicar entre sí, intentar dividir el problema en varios subproblemas más pequeños.
- Por ejemplo, en 4 submatrices de tamaños n/2 * n/2:

$$A_{n,n} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B_{n,n} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C_{n,n} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Primer Diseño:

$$A_{n,n} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B_{n,n} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C_{n,n} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

- Calcular las submatrices de **C** utilizando cálculos tradicionales:

$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$$
$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$
$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$$
$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$$

- Se calculan 8 multiplicaciones de tamaño n/2 * n/2 y 4 sumas (las sumas son $O((n/2)^2) = O(n^2)$)
- $T(n)= 8T(n/2)+4n^2$ -> **$O(n^3)$**
- **No obtenemos beneficio**

Segundo Diseño:

- Strassen consiguió descubrir en 1969 una forma de dividir el problema en otros subproblemas que, aunque abusando del número de sumas y restas de submatrices, reducía el número de multiplicaciones totales:

- **Se divide el problema de tamaño n en 7 subproblemas:**

$$M = (A_{11} + A_{22}) * (B_{11} + B_{22})$$
$$N = (A_{21} + A_{22}) * B_{11}$$
$$O = A_{11} * (B_{12} - B_{22})$$
$$P = A_{22} * (B_{21} - B_{11})$$
$$Q = (A_{11} + A_{12}) * B_{22}$$
$$R = (A_{21} - A_{11}) * (B_{11} + B_{12})$$
$$S = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

Diseño: 1. División del problema en subproblemas

- Dividiremos cada matriz **A** y **B** en 4 submatrices de tamaño equivalente:

$$A_{n,n} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B_{n,n} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C_{n,n} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

- Se divide el problema en 7 subproblemas:

$$M = (A_{11} + A_{22}) * (B_{11} + B_{22})$$
$$N = (A_{21} + A_{22}) * B_{11}$$
$$O = A_{11} * (B_{12} - B_{22})$$
$$P = A_{22} * (B_{21} - B_{11})$$
$$Q = (A_{11} + A_{12}) * B_{22}$$
$$R = (A_{21} - A_{11}) * (B_{11} + B_{12})$$
$$S = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

Diseño: 2. Resolución de cada subproblema y combinación.

- Los 7 subproblemas son independientes y se resolverán por separado. Llamemos **M, N, O, P, Q, R, S** a las subsoluciones de cada uno de los subproblemas.
- Combinaremos las subsoluciones de la siguiente forma para llegar a la solución global **C**:

$$C_{11} = M + P - Q + S$$
$$C_{12} = O + Q$$
$$C_{21} = N + P$$
$$C_{22} = M + O - N + R$$

Diseño: 2. Resolución de cada subproblema y combinación.

- Ejemplo:

$$A_{2,2}=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}\quad B_{2,2}=\begin{pmatrix}5 & 6\\ 7 & 8\end{pmatrix}\quad C_{n,n}=\begin{pmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{pmatrix}$$

$$M=(A_{11}+A_{22})*(B_{11}+B_{22})=(1+4)*(5+8)=65$$
$$N=(A_{21}+A_{22})*B_{11}=(3+4)*5=35$$
$$O=A_{11}*(B_{12}-B_{22})=1*(5-8)=-3$$
$$P=A_{22}*(B_{21}-B_{11})=4*(7-5)=8$$
$$Q=(A_{11}+A_{12})*B_{22}=(1+2)*8=24$$
$$R=(A_{21}-A_{11})*(B_{11}+B_{12})=(3-1)*(5+6)=22$$
$$S=(A_{12}-A_{22})*(B_{21}+B_{22})=(2-4)*(7+8)=-30$$

$$C_{11}=M+P-Q+S=65+8-24+-30=19$$
$$C_{12}=O+Q=-3+24=21$$
$$C_{21}=N+P=35+8=43$$
$$C_{22}=M+O-N+R=65+-3-35+22=49$$

Diseño: 3. Caso base.

- Se parará cuando **n=1, multiplicación de números.**

Diseño: 4. Adaptación de la plantilla DyV

Función C= MultiplicaMat(A, B)

Si n =1, **entonces** C=A*B;

En otro caso, hacer:
Dividir A=[A11 A12; A21, A22] y B= [B11 B12; B21,B22];
M= MultiplicaMat(A11+A22, B11+B22);
N= MultiplicaMat(A21+A22, B11);
O= MultiplicaMat(A11, B12-B22);
P= MultiplicaMat(A22, B21-B11);
Q= MultiplicaMat(A11+A12, B22);
R= MultiplicaMat(A21-A11, B11+B12);
S= MultiplicaMat(A12-A22, B21+B22);
...

Diseño: 4. Adaptación de la plantilla DyV *(continuación)*

```
...
   Calcular C11= M+P-Q+S
   Calcular C12= O+Q
   Calcular C21= N+P
   Calcular C22= M+O-N+R

   Combinar C= [C11 C12; C21 C22]
Fin-En otro caso
Devolver C
```

Eficiencia

- Se calculan 7 multiplicaciones y k sumas restas

$$T(n)= 7T(n/2)+kn^2$$

- **Resolviendo la ecuación, obtenemos que la eficiencia del método DyV es $O(n^{\log_2(7)})$, mejor que el método básico.**

Requisitos del método de Strassen

- Las matrices A y B deben ser cuadradas, y el número de filas y de columnas tiene que ser potencia de 2: 2^m
- Si estos requisitos no se cumplen, se pueden añadir y rellenar filas y columnas con el valor 0 (y los valores de la diagonal a valor 1) hasta que la matriz tenga el tamaño requerido.

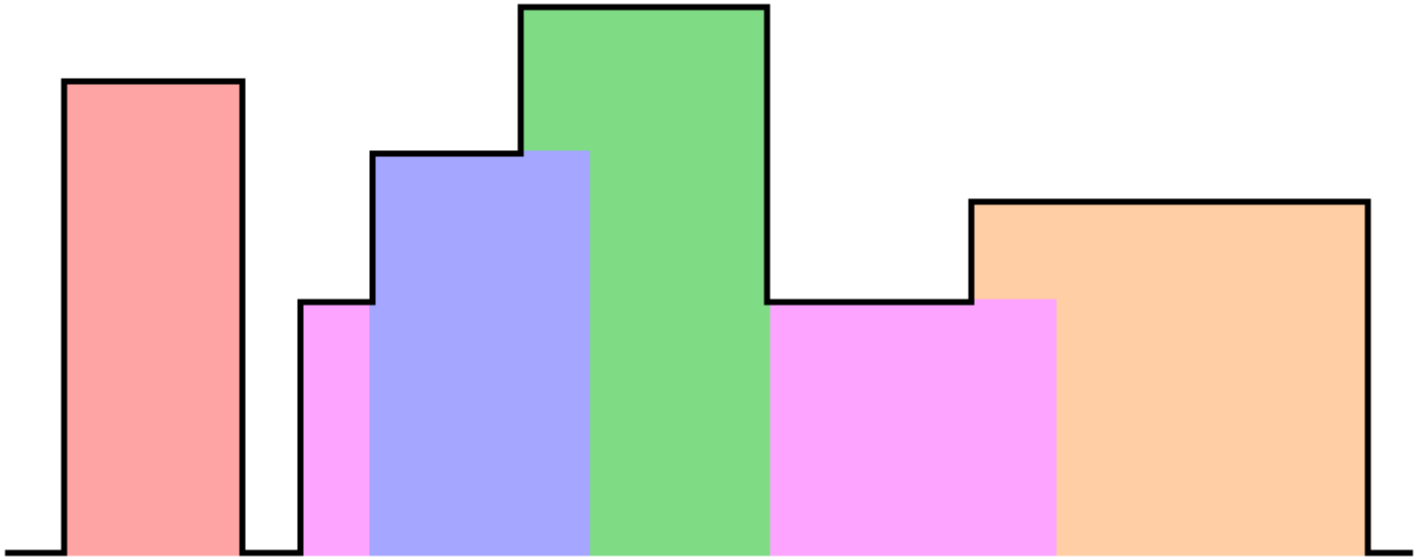
Algoritmos Divide y Vencerás

1. Introducción: DyV en términos de eficiencia.
2. La técnica Divide y Vencerás.
3. El problema del umbral.
4. Búsqueda Binaria.
5. Ordenación rápida.
6. El problema de selección.
7. Multiplicación rápida de enteros largos.
8. Multiplicación rápida de matrices.
9. La línea del horizonte.

Ejercicio de tema propuesto: Enunciado del problema

Sea **{r_i}** un conjunto de **n** rectángulos, de los que conocemos su posición izquierda **L(r_i)**, su posición derecha **D(r_i)** y su alto **A(r_i)**.

El problema consiste en dibujar la silueta sobre el conjunto de rectángulos, como si se tratase de la línea del horizonte de un conjunto de edificios.



Idea general para el método básico

- Podemos establecer una línea base **B** desde $\min_{1<=i<=n}\{\mathbf{L(r_i)}\}$ hasta $\max_{1<=i<=n}\{\mathbf{D(r_i)}\}$, inicialmente inicializada a una altura de 0.

- A continuación, recorrer cada rectángulo $\mathbf{r_i}$ y, para cada punto **p** desde $\mathbf{L(r_i)}$ **hasta** $\mathbf{D(r_i)}$, comprobar si **B(p)** tiene el valor máximo. En caso contrario, asignarlo.
- Eficiencia: $\mathbf{O(n^2)}$, suponiendo que hay **n** edificios cuyo ancho es también **n**.

Método básico

Función B= Skyline(R, n)

Entrada: R, conjunto de n rectángulos

Salida: B, línea de alturas de cada punto del skyline

B= línea desde min{ L(R) } hasta max{ D(R) }, inicializada a 0

Para cada rectángulo r en R, **hacer:**

Para cada punto p(r) desde L(r) hasta D(r), **hacer:**

Actualizar B(p) = max {B(p) , p(r) }

Fin-Para

Fin-Para

Devolver B

Ejercicio

1. Buscar información del problema *Skyline* en la bibliografía.
2. Si es posible, plantear el problema de la línea del horizonte desde el punto de vista de la técnica Divide y Vencerás.
3. Buscar una solución DyV que resuelva el problema.
4. Comparar la solución con el método básico, en términos de eficiencia.
5. Exponer la solución en pizarra.

Alternativa

Búsqueda de varios problemas no vistos en teoría ni prácticas (mínimo 3 ejercicios). Exponer los problemas en clase, junto con su diseño DyV.

UNIVERSIDAD DE GRANADA

DECSAI

Algorítmica

Grado en Ingeniería Informática

Tema 2 – Algoritmos Divide y Vencerás

Manuel Pegalajar Cuéllar

manupc@ugr.es

Departamento de Ciencias de la Computación e Inteligencia Artificial

http://decsai.ugr.es