



U N I V E R S I D A D
COMPLUTENSE
M A D R I D

Tema 2: Esquema algorítmico de divide y vencerás. Algoritmos de ordenación

Slides adaptadas a partir del original de Enrique Martín
Estructuras de Datos y Algoritmos (EDA)
Grado en Desarrollo de Videojuegos
Fac. Informática

Contenidos

- 1 Introducción
- 2 Divide y vencerás
- 3 Suma de un vector
- 4 Búsqueda en un vector
- 5 Subvector de suma máxima
- 6 Ordenación
- 7 Ordenación: mergesort
- 8 Ordenación: quicksort
- 9 Otros problemas clásicos
- 10 Bibliografía

Introducción

- Análisis de eficiencia
 - ① Análisis de eficiencia de los algoritmos ✓
- Algoritmos
 - ② **Esquema algorítmico de Divide y vencerás (*Divide-and-Conquer*) y algoritmos de ordenación**
 - ③ Esquema algorítmico de Vuelta atrás (*backtracking*)
- Estructuras de datos
 - ④ Diseño e implementación de Tipos abstractos de datos (TADs)
 - ⑤ Tipos de datos lineales
 - ⑥ Tipos de datos arborescentes
 - ⑦ Diccionarios
 - ⑧ Aplicaciones de TADs

- Los *esquemas algorítmicos* son estrategias conocidas para la resolución de problemas. Son *patrones* que se aplican en la resolución de problemas que presentan unas **características comunes**.
- En este curso veremos únicamente 2 métodos algorítmicos:
 - Divide y vencerás
 - Vuelta atrás
- En el 3^{er} curso veréis más en la asignatura *Métodos algorítmicos en resolución de problemas*:
 - Algoritmos voraces
 - Programación dinámica
 - Ramificación y acotación

Divide y vencerás

Divide y vencerás

El método de *divide y vencerás* se basa en 3 pasos:

- 1 **Divide** el problema original en un número de subproblemas que son **instancias más pequeñas del mismo problema**.
- 2 **Resuelve** cada uno de los subproblemas **recursivamente**. Cuando el tamaño del subproblema es *suficientemente* pequeño resuélvelo de manera directa.
- 3 **Combina** las soluciones a los subproblemas para **construir la solución al problema original**.

Divide y vencerás: esquema

```
1 fun divide-y-vencerás(x : problema) dev y : solución
2   if pequeño(x) then
3     y := metodo-directo(x)
4   else
5     // Descomponer x en k >= 1 problemas más pequeños
6     {x1, ..., xk} := descomponer(x)
7
8     // Resolver recursivamente cada subproblema
9     for j in [1..k]
10       yj := divide-y-vencerás(xj)
11
12     // Combina los yj para obtener la solución de x
13     y := combina(y1, ..., yk)
14 }
```


Suma de un vector

Suma de un vector

- Consideremos un vector $v[0..N)$ de enteros
- ¿Cómo calculamos la suma de todos sus elementos usando el esquema *divide y vencerás*?
- Tenemos varias maneras de dividir el problema en subproblemas más pequeños:
 - 1 Reducción por sustracción: generar un único subproblema de tamaño $N - k$ y sumar al resultado los k elementos no tratados
 - 2 Reducción por división: generar k subproblemas de tamaño $\frac{N}{k}$ y sumar sus resultados

Suma de un vector

Solución por sustracción con $k = 1$. Suponemos $0 \leq ini \leq fin \leq N$ y que el algoritmo calcula la suma del subvector $v[ini .. fin]$

```
1 template <typename T>
2 T sumaDV_1(const vector<T> &v, const int ini ,
3           const int fin) {
4     if (fin - ini == 0 ) // n = 0
5       return 0; // Elemento neutro de tipo T
6     else { // n > 0
7       T sum_p = sumaDV_1(v, ini+1, fin);
8       return sum_p + v[ini];
9     }
10 }
```

$$T(n)^1 = \begin{cases} 2 & \text{si } n = 0 \\ T(n-1) + 4 & \text{si } n > 0 \end{cases} \in \mathcal{O}(n)$$

¹Consideramos $n = fin - ini$

Suma de un vector

Solución por división con $k = 2$. Suponemos $0 \leq ini \leq fin \leq N$ y que el algoritmo calcula la suma del subvector $v[ini .. fin)$

```
1 template <typename T>
2 T sumaDV_div(const vector<T> &v, const int ini ,
3             const int fin) {
4     const int n = fin - ini;
5     if (n < 2) // n < 2
6         return (n == 0) ? 0: v[ini];
7     // expr. condicional -> (Cond) ? Valor_true: Valor_false
8     else { // n >= 2
9         int mit = (fin + ini) / 2;
10        T sum_p1 = sumaDV_div<T>(v, ini, mit);
11        T sum_p2 = sumaDV_div<T>(v, mit, fin);
12        return sum_p1 + sum_p2;
13    }
14 }
```

$$T(n)^2 = \begin{cases} 5 & \text{si } n < 2 \\ 2T(\frac{n}{2}) + 9 & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n)$$

²Consideramos $n = fin - ini$

- Hemos comprobado que en ambas soluciones el coste está en $\mathcal{O}(n)$. Es exactamente el mismo coste que nos habría salido si aplicamos el enfoque iterativo natural
- Si probásemos otras variantes de sustracción o división con otros valores de k obtendríamos el mismo coste
- Es *lógico*: para sumar los elementos de un vector de n elementos tendremos que recorrer los n elementos de una u otra forma

Búsqueda en un vector

Búsqueda en un vector no ordenado

- Consideremos un vector **no ordenado** $v[0..N)$ de elementos de tipo T , y un elemento e de tipo T
- ¿Cómo comprobamos si dicho elemento aparece en el vector usando el esquema *divide y vencerás*?
- Tenemos varias maneras de dividir el problema en subproblemas más pequeños, pero vamos a centrarnos en la reducción por **división** generando únicamente **2 subproblemas**: *el elemento estará en el vector completo si aparece en alguna de sus dos mitades*

Búsqueda en un vector no ordenado

Suponemos $0 \leq ini \leq fin \leq N$ y que la búsqueda se ciñe al subvector $v[ini .. fin)$

```
1 template <typename T>
2 bool member(const vector<T> &v, const int ini ,
3           const int fin , const T e) {
4     const int n = fin - ini;
5     if (n < 2) {
6       return (n == 0) ? false : (v[ini] == e);
7     } else {
8       const int mit = (ini + fin) / 2;
9       bool b1 = member<T>(v, ini , mit , e);
10      bool b2 = member<T>(v, mit , fin , e);
11      return b1 || b2;
12    }
13 }
```

$$T(n)^3 = \begin{cases} c_1 & \text{si } n < 2 \\ 2T(\frac{n}{2}) + c_2 & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n)$$

³Consideramos $n = fin - ini$

Búsqueda en un vector ordenado

- ¿Cambiaría algo si el vector fuese ordenado? ¡Claro!
- Si accedo a la posición media ($\text{mit} = (\text{ini} + \text{fin}) / 2$) del vector $v[\text{ini} .. \text{fin}]$ y consulto su valor, estaré en una de estas dos situaciones:
 - 1 Si $e < v[\text{mit}]$, sabemos que **no puede aparecer** en el subvector $v[\text{mit} .. \text{fin}]$
 - 2 Si $e \geq v[\text{mit}]$, sabemos que **si está debe aparecer** en el subvector $v[\text{mit} .. \text{fin}]$
- Gracias a esta propiedad, al desarrollar el algoritmo como *divide y vencerás* nos podremos ahorrar una llamada recursiva. Esto va a producir una mejora en el coste
- El algoritmo resultante se llama **búsqueda binaria**

Búsqueda en un vector ordenado

Suponemos $0 \leq ini \leq fin \leq N$ y que la búsqueda se ciñe al subvector $v[ini .. fin)$

```
1 template <typename T>
2 bool member_ord(const vector<T> &v, const int ini ,
3               const int fin , const T e) {
4     const int n = fin - ini;
5     if (n < 2) {
6         return (n == 0) ? false : (v[ini] == e);
7     } else {
8         const int mit = (ini + fin) / 2;
9         if (e < v[mit])
10            return member_ord<T>(v, ini, mit, e);
11        else
12            return member_ord<T>(v, mit, fin, e);
13    }
14 }
```

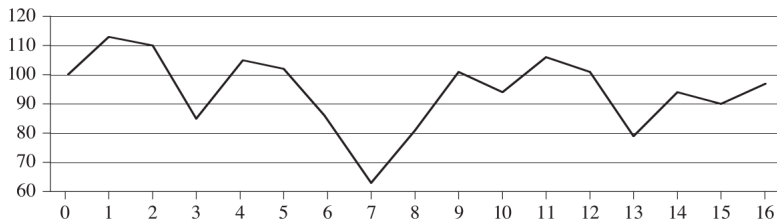
$$T(n)^4 = \begin{cases} c_1 & \text{si } n < 2 \\ T(\frac{n}{2}) + c_2 & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(\log n)$$

⁴Consideramos $n = fin - ini$

Subvector de suma máxima

Subvector de suma máxima

Imaginemos que conocemos cómo han evolucionado las acciones de una determinada empresa a lo largo de unos días:



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

5

¿Cómo descubriríamos cuáles fueron los mejores días para comprar y vender las acciones de tal manera que maximizásemos el beneficio?

⁵Imagen obtenida de Thomas H. Cormen *et al.*, 2009

Subvector de suma máxima

- Problemas como el anterior se pueden expresar como **encontrar el subvector de suma máxima** dentro de un vector
- En el caso de la bolsa nos centraríamos en el *vector de diferencias* de un día con el día anterior, que contiene números enteros.
- Querremos encontrar los índices i y j y la suma máxima $s_{ij} = \sum_{k=i}^{j-1} v[k]$, con $0 \leq i \leq j \leq n$
- Una vez encontrado el subvector de suma máxima $v[i .. j)$ podemos generar la solución al problema de la bolsa:
 - Compra las acciones el día $i - 1$
 - Vende las acciones el día j

Subvector de suma máxima

Existe una versión iterativa directa: detectar todos los posibles rangos $[i..j]$ con $i \leq j$, calcular la suma $s_{ij} = v[i] + v[i+1] + \dots + v[j-1]$ y quedarse con la mayor.

```
1 fun suma_max(v : vector<T>[0..n]) dev y : tupla
2   suma_max := -∞
3   max_i, max_j := -1
4   for i in [0..n]
5     for j in [i..n]
6       s := suma(v, i, j) ∈ O(j-i)
7       if s > suma_max then
8         suma_max := s, max_i := i, max_j := j
9   y := <suma_max, max_i, max_j>
```

Si realizamos el cálculo del coste de este algoritmo tendremos que está en $O(n^3)$: dos bucles anidados que dependen de n , más el coste de suma que también depende de la longitud del vector

Subvector de suma máxima

¿Se puede hacer mejor? **Sí**, si nos damos cuenta que $\text{suma}(v, i, j)$ repite mucho trabajo: $\text{suma}(v, i, j+1) = \text{suma}(v, i, j) + v[j]$. Si modificamos un poco el algoritmo tenemos que:

```
1 fun suma_max(v : vector<T>[0..n)) dev y : tupla
2   suma_max := 0 //Siempre será cota inferior
3   max_i := 0, max_j := 0 //Rango vacío, suma 0
4   for i in [0..n-1]
5     s := 0 //suma de v[i.. i)
6     for j in [i+1..n] //Evitamos rangos vacíos
7       s := s + v[j-1]
8       if s > suma_max then
9         suma_max := s, max_i := i, max_j := j
10  y := <suma_max, max_i, max_j>
```

Ahora tenemos un coste en $\mathcal{O}(n^2)$, que es una mejora considerable. ¿Se puede hacer mejor? Probemos con *divide y vencerás*

Subvector de suma máxima

Imaginemos que dividimos el vector en **dos mitades** y podemos obtener el rango de suma máxima para cada una de ellas. ¿Podríamos reconstruir la solución a partir de esa información?

- 1 El subvector de suma máxima **está completamente en la primera mitad** → ¡ya lo tenemos!
- 2 El subvector de suma máxima **está completamente en la segunda mitad** → ¡ya lo tenemos!
- 3 El subvector de suma máxima **atraviesa la mitad del vector** → esto habría que calcularlo...

El problema de encontrar el subvector de suma máxima *que atraviesa la mitad del vector* **no es una instancia del problema original**. No podemos resolverlo de manera recursiva con el mismo procedimiento, pero sí se puede resolver de manera directa $\in \mathcal{O}(n)$ [con $n = fin - ini$]

Subvector de suma máxima

Recorremos el vector desde mit hacia cada uno de los lados, quedándonos con el mejor rango encontrado. Finalmente los *concatenamos*:

```
1 def max_crossing(v : vector<T>, ini , fin , mit : nat)
2     dev y : tupla
3     // Suponemos ini < mit < fin, luego fin - ini >= 2
4     max_left := mit-1, left_sum := v[mit-1], sum := v[mit-1],
5     for i in [mit-2..ini] // Hacia atrás
6         sum := sum + v[i]
7         if sum > left_sum then
8             left_sum := sum, max_left := i
9
10    max_right := mit+1, right_sum := v[mit], sum := v[mit],
11    for j in [mit+1..fin) // Hacia delante
12        sum := sum + v[j]
13        if sum > right_sum then
14            right_sum := sum, max_right := j+1
15        // Si v[j] se ha sumado, max_right debe ser j+1
16
17    y := <left_sum + right_sum , max_left , max_right>
```

Subvector de suma máxima

Con estos 3 ingredientes ya sí que podemos resolver el problema de encontrar el subvector de suma máxima en $v[ini .. fin)$ usando el método de *divide y vencerás*:

- Si $fin - ini = 0 \rightarrow$ Rango vacío, suma máxima = 0
- Si $fin - ini = 1 \rightarrow$ Rango unitario, suma máxima = $\max(0, v[ini])$
- Si $fin - ini \geq 2 \rightarrow$ Sea la mitad $mit = (ini + fin) / 2$. El subvector de suma máxima será el máximo entre:
 - 1 El subvector de suma máxima en $v[ini .. mit)$
 - 2 El subvector de suma máxima en $v[mit .. fin)$
 - 3 El subvector de suma máxima que atraviesa la separación, es decir, incluye $v[mit-1]$ y $v[mit]$

Subvector de suma máxima

```
1 fun max_subvector(v : vector<T>, ini , fin : nat) dev y : tupla
2   len := fin - ini
3   if len = 0 then
4     y := <0, ini , ini>
5   else if len == 1 then
6     if v[ini] > 0 then
7       y := <v[ini], ini , fin>
8     else
9       y := <0, ini , ini>
10  else
11    mit := (ini + fin) / 2
12    l := max_subvector(v, ini , mit)
13    r := max_subvector(v, mit, fin)
14    cross := max_crossing(v, ini , fin , mit)
15    y := max_tuple(l , r , cross)
```

La función `max_tuple` recibe 3 tuplas y devuelve aquella que tenga un mayor valor en su primera componente. Por lo tanto su coste está en $\mathcal{O}(1)$

- El coste de esta función estaría definido por la siguiente recurrencia:

$$T(n)^6 = \begin{cases} c_1 & \text{si } n < 2 \\ 2T(\frac{n}{2}) + c_2n & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n \cdot \log n)$$

- Este coste mejora al de nuestro mejor algoritmo iterativo: $\mathcal{O}(n^2)$
- Sin embargo se puede mejorar aún más: cada llamada recursiva podría devolver los subvectores máximos que comienzan en sus extremos (ver Narciso Martí *at al.*, 2013). En este caso el coste puede bajar y estar en $\mathcal{O}(n)$

⁶Consideramos $n = fin - ini$

Ordenación

Hasta ahora hemos visto cómo se aplica el esquema de *divide y vencerás* a distintos problemas, pero hay un problema concreto donde este esquema es muy útil: **la ordenación**.

- Los métodos iterativos (inserción, selección, burbuja) ordenan en $\mathcal{O}(n^2)$. *¿Se puede hacer mejor?*
- **Sí**: el algoritmo *mergesort* ordena en $\mathcal{O}(n \log n)$ en el caso peor, y *quicksort* en $\mathcal{O}(n \log n)$ en el caso promedio. *¿Se puede hacer aún mejor?*
- **No**: cualquier algoritmo de ordenación *basado en comparaciones* debe realizar al menos $n \log n$ comparaciones [Thomas H. Cormen *et al.*, 2009. **Capítulo 8**]

El enfoque de *divide y vencerás* que hay detrás de *mergesort* y *quicksort* es el mismo:

- 1 Dividir el vector en dos *partes*
- 2 Ordenar recursivamente cada una de las partes
- 3 Combinar ambas partes ordenadas para obtener el vector ordenado completo

La diferencia radica en cómo realizar la partición y la posterior combinación:

- *mergesort* divide por la mitad en dos partes de igual tamaño $\mathcal{O}(1)$ y luego tiene que mezclarlas de manera ordenada $\mathcal{O}(n)$
- *quicksort* utiliza un pivote para recolocar los elementos del vector en 3 partes con coste $\mathcal{O}(n)$: los estrictamente menores, los iguales y los estrictamente mayores. Luego la combinación es innecesaria: $\mathcal{O}(1)$

Ordenación: mergesort

Ordenación: mergesort

```
1 proc mergesort(v : vector<T>, ini , fin : nat)
2 // Ordena el vector v[ini .. fin )
3   if ini < fin - 1 then // longitud > 1
4     mid := (ini + fin) / 2
5     mergesort(v, ini , mid)
6     mergesort(v, mid, fin)
7     merge(v, ini , mid, fin)
8   // Si longitud <= 1 no hay que hacer nada
9   // porque ya está ordenado
```

- mergesort no devuelve nada, sino que ordena el propio vector v
- El código de *mergesort* es muy sencillo, aunque aún queda detallar el procedimiento *merge()*
- Si $n = fin - ini$ y el coste de *merge* $\in \mathcal{O}(n)^7$ tenemos la recurrencia:

$$T(n) \begin{cases} c_1 & \text{si } n < 2 \\ 2T(\frac{n}{2}) + c_2n & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n \log n)$$

⁷Lo verificaremos más adelante

Mezcla ordenada

```
1 proc merge(v : vector<T>, p,q,r : nat) {
2   // Suponemos  $p \leq q \leq r$ ,  $v[p..q]$  y  $v[q..r]$  ordenados
3   // Genera una mezcla ordenada de ambas partes en  $v[p..r]$ 
4   nl := q - p, nr := r - q
5   vl : vector<T>[0..nl) // Memoria adicional
6   vr : vector<T>[0..nr) // Memoria adicional
7
8   for i in [0..nl) // Copia  $v[p..q]$  en  $vl[0..nl)$ 
9     vl[i] = v[p+i]
10  for j in [0..nr) // Copia  $v[q..r]$  en  $vr[0..nr)$ 
11    vr[j] = v[q+j]
12
13  i := 0, j := 0;
14  for k in [p..r) // Mezclamos vl y vr en v
15    if j >= nr  $\vee$  (i < nl  $\wedge$  vl[i] <= vr[j]) then
16      // El vector vr está agotado o
17      // vr y vl no agotados y  $vl[i] \leq vr[j]$ 
18      v[k] := vl[i]
19      i := i + 1
20    else
21      v[k] := vr[j]
22      j := j + 1
```

- El procedimiento $\text{merge}(v, p, q, r)$ contiene 3 bucles:
 - L8 con $nl = q - p$ iteraciones de coste constante
 - L10 con $nr = r - q$ iteraciones de coste constante
 - L14 con $r - p$ iteraciones de coste constante
- Si tomamos $n = r - p$ tenemos que:
 - los bucles L8 + L10 realizan $(q - p) + (r - q) = r - p = n$ iteraciones
 - el bucle L14 realiza n iteraciones
- En total, tenemos que $\text{merge}(v, p, q, r)$ realiza un número de operaciones proporcional a n , luego su coste $\in \mathcal{O}(n)$

Evaluación de mergesort

- Hemos visto que el coste de *mergesort* es óptimo: su coste $\in \mathcal{O}(n \cdot \log n)$, y esa es la cota inferior para cualquier algoritmo de ordenación. **¿Por qué seguimos buscando?**
- *mergesort* tiene una ligera desventaja, ya que necesita **espacio adicional** para la mezcla ordenada. Concretamente:
 - En L5 reserva $q - p$ elementos en memoria
 - En L6 reserva $r - q$ elementos en memoria
 - En total cada llamada a merge tiene un coste de n elementos de memoria. Si calculamos el **coste en memoria adicional** nos saldrá una recurrencia así:

$$T_{mem}(n) \begin{cases} 0 & \text{si } n < 2 \\ 2T_{mem}(\frac{n}{2}) + n & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n \cdot \log n)$$

Evaluación de mergesort

- En cada llamada a merge estamos creando 2 vectores, pero podríamos reutilizarlos de una llamada a otra
- Para ello, antes de empezar la ordenación podemos crear dos vectores vl y vr de tamaño $n/2$ y que todas las llamadas a merge los utilicen
- En ese caso el coste en memoria adicional baja de $\mathcal{O}(n \cdot \log n)$ a $\mathcal{O}(n)$.
¿Se puede mejorar?
- Para mejorarlo necesitamos un algoritmo **lineal** para la mezcla ordenada que no use memoria adicional. Existen, pero son complicados y hay que implementarlos con cuidado para no «fastidiar» el coste de la ordenación:
 - M.A. Kronrod. *Optimal ordering algorithm without operational field*. Soviet Math. Dokl., 10 (1969), pp. 744-746
 - Antonios Symvonis. *Optimal Stable Merging*. The Computer Journal, 38(8), 1995.

Ordenación: quicksort

Ordenación: quicksort

A diferencia de *mergesort*, *quicksort* realiza el «trabajo duro» para generar los subproblemas, mientras que la combinación de resultados es trivial.

Para ello:

- 1 Escoge un elemento del vector que usará como **pivote** para realizar el particionado
- 2 Usando ese pivote, reordena los elementos del vector para formar 3 partes **consecutivas**: los elementos menores que el pivote, los iguales al pivote, y los elementos mayores al pivote
- 3 Los elementos iguales al pivote **ya están en su lugar definitivo**, así que únicamente hay que ordenar recursivamente los menores y mayores
- 4 Una vez esas partes están ordenadas no hay que hacer nada: todo está en su sitio

```
1 proc quicksort(v : vector<T>, ini , fin : nat)
2   // Ordena v[ini .. fin )
3   if ini < fin - 1 then // longitud > 1
4     pivote := elige_pivote(v ,ini , fin)
5     i , j := partition(v , pivote , ini , fin)
6     // Parte en 3 trozos
7     quicksort(v , ini , i)
8     quicksort(v , j , fin)
```

- Suponemos un mecanismo para elegir al pivote $\text{elige_pivote} \in \mathcal{O}(1)$
- La función $\text{partition}(v, \text{pivote}, \text{ini}, \text{fin})$ recoloca los elementos de $v[\text{ini} .. \text{fin})$ y nos devuelve 2 índices i y j tales que
 - $\forall k \in [\text{ini}..i). v[k] < \text{pivote}$
 - $\forall k \in [i..j). v[k] = \text{pivote}$
 - $\forall k \in [j..fin). v[k] > \text{pivote}$
- Es importante que el coste de partition esté en $\mathcal{O}(n)$, con $n = \text{fin} - \text{ini}$

partition

```
1 // Problema de la bandera holandesa — Edsger Dijkstra
2 // Reordena v[ini .. fin ) en 3 segmentos contiguos
3 fun partition(v : vector<T>, pivote : T,
4             ini, fin : nat) dev i, j: nat
5   i := ini, j := ini, k = fin
6   // MENORES → [ini..i); IGUALES → [i..j)
7   // MAYORES → [k..fin); SIN PROCESAR → [j..k)
8   while j < k
9     if v[j] < pivote then
10      swap(v, i, j) // Intercambio de elementos
11      i := i+1, j := j+1
12     else if v[j] > pivote then
13      swap(v, j, k-1) // Intercambio de elementos
14      k := k-1
15     else
16      j := j+1
```

La diferencia $k - j$ decrece en cada iteración ($k - 1$ o $j + 1$). Si $n = fin - ini$ entonces el bucle realiza n iteraciones de coste constante \rightarrow **coste** $\in \mathcal{O}(n)$

Coste de quicksort

El coste de quicksort dependerá del tamaño de las partes generadas por partition , que a su vez depende del **pivote** elegido:

- Caso mejor: todos los elementos iguales al pivote $\rightarrow i = ini, j = fin$

$$T_{mejor}(n) \begin{cases} c_1 & \text{si } n < 2 \\ 2T_{mejor}(0) + c_2n & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n)$$

- Caso peor: todos los elementos mayores, o todos menores $\rightarrow i = ini \vee j = fin$. Además, no hay más elementos con el mismo valor que el pivote $\rightarrow j = i + 1$

$$T_{peor}(n) \begin{cases} c_1 & \text{si } n < 2 \\ T_{peor}(n-1) + c_2n & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n^2)$$

- Caso promedio $\in \mathcal{O}(n \log n)$ [Thomas H. Cormen *et al.*, 2009]

Coste en memoria de quicksort y pivotaje

Desde el punto de vista de coste en espacio, quicksort no necesita ninguna cantidad de memoria adicional. Las llamadas a quicksort y partition utilizan una cantidad constante de variables.

A la hora de elegir el pivote se pueden seguir distintas técnicas:

- Escoger el primer elemento del vector ($v[\text{ini}]$). Si el vector original está «bastante ordenado» tiende a generar el caso peor
- Escoger una posición al azar
- Seleccionar 3 posiciones al azar y quedarse con el valor *mediano*.

En todo caso la técnica de selección debe ser poco costosa: $\mathcal{O}(1)$

Estabilidad de la ordenación

- Un método de ordenación es estable si dos elementos con la misma clave aparecen en el mismo orden relativo tras la ordenación
- Si los vectores a ordenar contienen enteros no importa, pero sí cuando ordenamos estructuras complejas (p.ej. registros sanitarios en base a su apellido)

Algoritmo	¿Estable?
Bubble sort	✓
Insertion sort	✓
Selection sort	
Mergesort	✓
Quicksort	

Otros problemas clásicos

Otros problemas clásicos

En este tema hemos visto unos cuantos ejemplos de problemas que admiten una solución siguiendo el esquema de *divide y vencerás*. Sin embargo, nos hemos dejado algunos problemas clásicos sin tratar:

- Multiplicación de matrices cuadradas $N \times N$ con el algoritmo de Strassen $\in \mathcal{O}(n^{\log 7})$
[Thomas H. Cormen *et al.*, 2009]
- Encontrar el i -ésimo menor elemento de un vector $\in \mathcal{O}(n)$
[Gilles Brassard, Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996]
- Obtener la envolvente convexa (*convex hull*) de un conjunto de puntos $\in \mathcal{O}(n \cdot \log n)$
[Joseph O'Rourke. *Computational Geometry in C (Second Edition)*. Cambridge University Press, 1998]

Bibliografía

- Narciso Martí, Yolanda Ortega, Alberto Verdejo. *Estructuras de Datos y Métodos Algorítmicos: 213 Ejercicios resueltos (2ª Edición)*. Garceta, 2013. **Capítulo 11**.
http://cisne.sim.ucm.es/record=b3290150~S6*spi
También está disponible la versión de Pearson Prentice-Hall:
*http://cisne.sim.ucm.es/record=b2789524~S6*spi*
- Larry Nyhoff. *ADTs, data structures, and problem solving with C++ (Second Edition)*. Pearson/Prentice Hall, 2005. **Capítulo 13**.
http://cisne.sim.ucm.es/record=b3601644~S6*spi
- Thomas H. Cormen, Charles E. Leiserson, Ronarld L. Rivest, Clifford Stein. *Introduction to Algorithms (Third Edition)*. MIT Press, 2009. **Capítulos 2, 4 y 7**.
http://cisne.sim.ucm.es/record=b2541535~S6*spi