UNIVERSIDAD COMPLUTENSE MADRID

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
Facultad de Informática

Temario

- 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

Contenidos

1. Introducción a los algoritmos recursivos
2. Divide y vencerás
3. Búsqueda en un vector
4. Subvector de suma máxima
5. Ordenación
6. Ordenación: mergesort
7. Ordenación: quicksort
8. Otros problemas clásicos
9. Bibliografía

Introducción a los algoritmos recursivos

Ejemplo: Función para calcular el factorial

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

- Se ve intuitivamente que se generan n llamadas recursivas, análogo a lo que sería iterar en un bucle n veces. Por tanto, $T(n) \in \mathcal{O}(n)$.
- Si analizamos la recurrencia tenemos esto:

$$T(n) = \begin{cases} c_1 & \text{si } n = 0 \\ T(n-1) + c_2 & \text{si } n \neq 0 \end{cases} \in \mathcal{O}(n)$$

Ejemplo: Factorial devolviendo resultado por parámetro

- Aunque resulte más incómodo, podríamos devolver el resultado por parámetro en lugar de por return:

```
void factParam(int n, int& res){
    if (n == 0) res = 1;
    else {
        int resAux;
        factParam(n-1, resAux);
        res = resAux*n;
    }
}
```

- El coste sería igual al de la versión anterior (salvo constantes).

Ejemplo: Factorial con acumulador

- Si analizamos el comportamiento, podemos imaginar un primer bucle que va decreciendo el parámetro, y luego otro que va construyendo el resultado.
- Podemos escribir una versión en la que se vaya acumulando el resultado según va decreciendo el parámetro:

```
int factAc(int n, int ac){ // Func. recursiva auxiliar
    if (n == 0) return ac;
    return factAc(n-1, ac*n);
}

int fact(int n){ // Función principal
    return factAc(n, 1);
}
```

- El coste asintótico sería igual ($T(n) \in \mathcal{O}(n)$), pero el algoritmo sería algo más eficiente.

Ejemplo: Factorial con acumulador

- A este tipo de recursión se le conoce como **recursión final** (observa que no hay código detrás de la llamada recursiva).
- Cuando hay código detrás de la(s) llamada(s) se dice que es **recursión no-final**.
- Podemos escribir una versión (void) en la que el acumulador se use también como argumento de salida:

```
void factAc(int n, int& ac){
    if (n == 0) return ;
    else {
        ac = n*ac;
        factAc(n-1, ac);
    }
}

int fact(int n){ // Función principal
    int res = 1; // Necesaria la inicialización!
    factAc(n, res);
    return res;
}
```

Ejemplo: Suma de elementos de un vector

- Cambiamos de ejemplo. Consideremos una función para calcular la suma de los elementos de un vector.
- Una versión iterativa evidente podría ser así:

```
template <typename T>
int suma(const vector<T>& v){
    int r = 0;
    for (T e : v) r += e;
    return r;
}
```

- Se recorre el vector entero una vez. Por lo tanto el coste sería $\mathcal{O}(n)$ siendo $n = v.size()$.

Ejemplo: Suma de elementos de un vector

- Probemos con recursión. Para ello necesitamos poder tratar con fragmentos más pequeños del vector. Usaremos dos parámetros enteros, `ini` y `fin`. En la llamada inicial $ini = 0$ y $fin = v.size()$.
- Si no tendríamos que construir subvectores $\Rightarrow$ Muy ineficiente!

```
template <typename T>
int sumaRec(const vector<int>& v, int ini, int fin){
    int n = fin-ini;
    if (n == 0) return 0;
    return v[ini] + sumaRec(v, ini+1, fin);
}
```

- Esencialmente el algoritmo hace lo mismo que la versión iterativa, por tanto debería ser también $\mathcal{O}(n)$ (siendo $n = fin - ini$).
- Analizando la recurrencia obtenemos lo mismo:

$$T(n) = \begin{cases} c_1 & \text{si } n = 0 \\ T(n-1) + c_2 & \text{si } n \neq 0 \end{cases} \in \mathcal{O}(n)$$

Ejemplo: Suma de elementos de un vector

- Probemos ahora con una versión final con acumulador + parámetro de salida (se podría tb con acumulador + return, ver slide 7):

```
template <typename T>
void sumaRecAc(const vector<T>& v, int ini, int fin, int& acu){
    int n = fin - ini;
    if (n == 0) return;
    else {
        acu += v[ini];
        sumaRecAc(v, ini+1, fin, acu);
    }
}

int suma(const vector<T>& v){
    int r = 0; // Necesaria la inicialización!
    sumaRecAc(v, 0, v.size(), r);
    return r;
}
```

- El coste sería el mismo, aunque de nuevo se puede decir que esta versión es algo más rápida.

Ejemplo: Suma de elementos de un vector

- El vector se podría reducir también por el final, o por ambos extremos. Se deja como ejercicio.
- Otra forma de reducir el vector es dividiéndolo en dos partes:

```
template <typename T>
int sumaRecDiv(const vector<T>& v, int ini, int fin) {
  int n = fin - ini;
  if (n == 0) return 0;
  if (n == 1) return v[ini]; // Ojo, necesario!
  int mit = (ini+fin)/2;
  return sumaRecDiv(v,ini,mit) + sumaRecDiv(v,mit,fin);
}
```

- Tendríamos la siguiente recurrencia. Lógicamente el coste sigue siendo lineal (hay que recorrer el vector entero!).

$$T(n)^1 = \begin{cases} c_1 & \text{si } n < 2 \\ 2T(\frac{n}{2}) + c_2 & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n)$$

[1] Consideramos $n = fin - ini$

Divide y vencerás

Métodos algorítmicos

- 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

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 se resuelve de manera directa.
3. **Combina** las soluciones a los subproblemas para **construir la solución al problema original**.

Divide y vencerás: esquema

```
fun divide-y-vencerás(x : problema) dev y : solución
  if pequeño(x) then
    y := metodo-directo(x)
  else
    // Descomponer x en k >= 1 problemas más pequeños
    {x1, ..., xk} := descomponer(x)

    // Resolver recursivamente cada subproblema
    for j in [1..k]
      yj := divide-y-vencerás(xj)

    // Combina los yj para obtener la solucion de x
    y := combina(y1, ..., yk)
}
```

Suma de un vector

- Nuestra última implementación sumaRectDiv sería un algoritmo divide y vencerás!
- Aunque en este caso no ha servido de nada... No hemos mejorado el coste $\mathcal{O}(n)$ de la versión iterativa 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)

```
template <typename T>
bool member(const vector<T>& v, int ini, int fin,
            const T& e) {
  int n = fin - ini;
  if (n < 2) {
    return (n == 0) ? false : (v[ini] == e);
  } else {
    int mit = (ini + fin)/2;
    bool estaIzq = member(v, ini, mit, e);
    bool estaDcha = member(v, mit, fin, e);
    return estaIzq || estaDcha;
    // Se puede optimizar para que no busque en la dcha
    // si ya lo ha encontrado en la izqda.
  }
}
```

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

[2] Consideramos $n = fin - ini$

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 $\text{v}[\,\text{ini} \,..\, \text{fin}\,)$ y consulto su valor, estaré en una de estas dos situaciones:
  1. Si $\text{e} < \text{v[mit]}$, sabemos que **no puede aparecer** en el subvector $\text{v}[\text{mit} \,..\, \text{fin}\,)$.
  2. Si $\text{e} >= \text{v[mit]}$, sabemos que **si está debe aparecer** en el subvector $\text{v}[\text{mit} \,..\, \text{fin}\,)$.
- 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 a $\mathcal{O}(log\ n)$. Ahora si!
- El algoritmo resultante se llama **búsqueda binaria**.

Búsqueda en un vector ordenado: Búsqueda binaria

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

```
template <typename T>
bool memberOrd(const vector<T>& v, int ini, int fin,
               const T& e) {
  int n = fin - ini;
  if (n < 2) {
    return (n == 0) ? false : (v[ini] == e);
  } else {
    int mit = (ini + fin) / 2;
    if (e < v[mit])
      return memberOrd(v, ini, mit, e);
    else
      return memberOrd(v, mit, fin, e);
  }
}
```

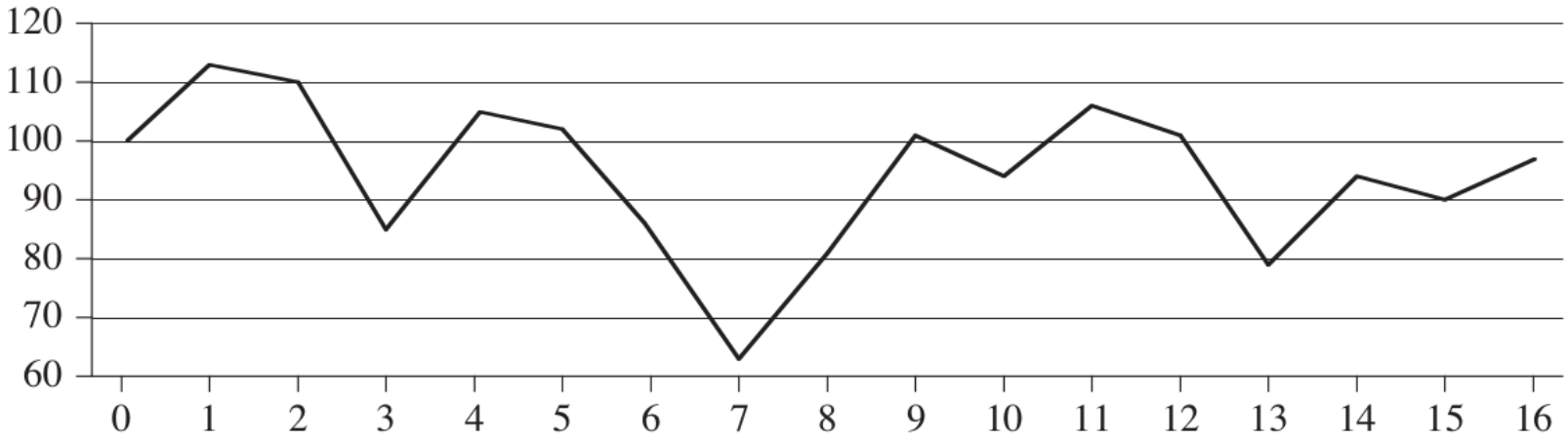
$$T(n)^3 = \begin{cases} c_1 & \text{si } n < 2 \\ T(\frac{n}{2}) + c_2 & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(log\ n)$$

[3] Consideramos $n = fin - ini$

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

[4]

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

[4] Imagen 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 $s_{ij} = \sum_{k=i}^{j-1} v[i]$, 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 $s_{ij} = v[i] + v[i+1] + \ldots + v[j-1]$ y quedarse con la mayor.

```
fun suma_max(v : vector<T>[0..n)) dev y : tupla
  suma_max := −∞
  max_i, max_j := −1
  for i in [0..n]
    for j in [i..n]
      s := suma(v, i, j) ∈ O(j − i)
      if s > suma_max then
        suma_max := s, max_i := i, max_j := j
  y := <suma_max, max_i, max_j>
```

Si realizamos el cálculo del coste de este algoritmo tendremos que está en $\mathcal{O}(n^3)$: 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 suma(v, i, j) repite mucho trabajo: suma(v,i,j+1) = suma(v,i,j) + v[j]. Si modificamos un poco el algoritmo tenemos que:

```
fun suma_max(v : vector<T>[0..n)) dev y : tupla
  suma_max := 0 //Siempre será cota inferior
  max_i := 0, max_j := 0 //Rango vacío, suma 0
  for i in [0..n-1]
    s := 0 //suma de v[i.. i)
    for j in [i+1..n] //Evitamos rangos vacíos
      s := s + v[j-1]
      if s > suma_max then
      suma_max := s, max_i := i, max_j := j
  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** $\rightarrow$ ¡ya lo tenemos!
2. El subvector de suma máxima **está completamente en la segunda mitad** $\rightarrow$ ¡ya lo tenemos!
3. El subvector de suma máxima **atraviesa la mitad del vector** $\rightarrow$ 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 en $\mathcal{O}(n)$ (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*:

```
def max_crossing(v : vector<T>, ini,fin,mit : nat)
                 dev y : tupla
  // Suponemos ini < mit < fin, luego fin - ini >= 2
  max_left := mit-1, left_sum := v[mit-1], sum := v[mit-1],
  for i in [mit-2..ini] // Hacia atrás
    sum := sum + v[i]
    if sum > left_sum then
      left_sum := sum, max_left := i

  max_right := mit+1, right_sum := v[mit], sum := v[mit],
  for j in [mit+1..fin) // Hacia delante
    sum := sum + v[j]
    if sum > right_sum then
      right_sum := sum, max_right := j+1
      // Si v[j] se ha sumado, max_right debe ser j+1

  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

```
fun max_subvector(v : vector<T>, ini,fin : nat) dev y : tupla
  n := fin - ini
  if n = 0 then
    y := <0, ini, ini>
  else if n = 1 then
    if v[ini] > 0 then
      y := <v[ini], ini, fin>
    else
      y := <0, ini, ini>
  else
    mit := (ini + fin) / 2
    l := max_subvector(v, ini, mit)
    r := max_subvector(v, mit, fin)
    cross := max_crossing(v, ini, fin, mit)
    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 $\mathcal{O}(1)$.

Subvector de suma máxima

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

$$T(n)^5 = \begin{cases} c_1 & \text{si } n < 2 \\ 2T(\frac{n}{2}) + c_2 n & \text{si } n \geq 2 \end{cases} \quad \in \mathcal{O}(n.log\ n)$$

- 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 $\mathcal{O}(n)$.

[5] Consideramos $n = fin - ini$

Ordenación

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**].

Ordenación

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

```
proc mergesort(v : vector<T>, ini, fin: nat)
// Ordena el vector v[ini .. fin)
  if ini < fin - 1 then // longitud > 1
    mid := (ini + fin) / 2
    mergesort(v, ini, mid)
    mergesort(v, mid, fin)
    merge(v, ini, mid, fin)
  // Si longitud <= 1 no hay que hacer nada
  // 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 $\in \mathcal{O}(n)$[6] tenemos la recurrencia:

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

[6] Lo verificaremos más adelante

Mezcla ordenada

```
1  proc merge(v : vector<T>, p,q,r : nat) {
2  // Suponemos p<=q<=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 ∨ (i < nl ∧ vl[i] <= vr[j]) then
16      // El vector vr está agotado o
17      // vr y vl no agotados y vl[i] <= vr[j]
18        v[k] := vl[i]
19        i := i + 1
20      else
21        v[k] := vr[j]
22        j := j + 1
```

Mezcla ordenada

- El procedimiento 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 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 $\in \mathcal{O}(n.log\ n)$, 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:

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

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 creamos 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 $\mathcal{O}(n.log\ n)$ 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.

quicksort

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

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

partition

```
// Problema de la bandera holandesa – Edsger Dijkstra
// Reordena v[ini .. fin ) en 3 segmentos contiguos
fun partition(v : vector<T>, pivote : T,
              ini,fin : nat) dev i,j: nat
  i := ini, j := ini, k = fin
  // MENORES → [ini..i); IGUALES → [i..j)
  // MAYORES → [k..fin); SIN PROCESAR → [j..k)
  while j < k
    if v[j] < pivote then
      swap(v, i, j) // Intercambio de elementos
      i := i+1, j := j+1
    else if v[j] > pivote then
      swap(v, j, k-1) // Intercambio de elementos
      k := k-1
    else
      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 $\rightarrow i = ini$, $j = fin$

$$T_{mejor}(n) \begin{cases} c_1 & \text{si } n < 2 \\ 2\,T_{mejor}(0) + c_2 n & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n)$$

- 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 $\rightarrow j = i + 1$

$$T_{peor}(n) \begin{cases} c_1 & \text{si } n < 2 \\ T_{peor}(n-1) + c_2 n & \text{si } n \geq 2 \end{cases} \in \mathcal{O}(n^2)$$

- 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[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

Problemas clásicos resueltos mediante *divide y vencerás*:

- 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]
- Envolvente convexa (*convex hull*) de conjunto de puntos $\in \mathcal{O}(n.log\ n)$
  [Joseph O'Rourke. *Computational Geometry in C (Second Edition)*. Cambridge University Press, 1998]
- Transformada Rápida de Fourier (FFT) $\in \mathcal{O}(n.log\ n)$
  [J. W. Cooley and John Tukey, 1965]

Bibliografía

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`