

Estrategias algorítmicas

Tema 3(I)

Algorítmica y Modelos de Computación

Curso 2017-18

Tema 3. Estrategias algorítmicas sobre estructuras de datos no lineales.

1. Introducción.
2. Algoritmos divide y vencerás.
3. Algoritmos voraces.
4. Programación dinámica.
5. Algoritmos vuelta atrás (Backtracking).
6. Ramificación y poda.

1. Introducción

- A través de los años, los científicos de la computación han identificado diversas técnicas generales que a menudo producen algoritmos eficientes para la resolución de muchas clases de problemas. Este capítulo presenta algunas de las técnicas más importantes como son:
 - dividir para conquistar,
 - técnicas ávidas,
 - programación dinámica y
 - el método de retroceso
- Se debe, sin embargo, destacar que hay algunos problemas, como los NP completos, para los cuales ni éstas ni otras técnicas conocidas producirán soluciones eficientes. Cuando se encuentra algún problema de este tipo, suele ser útil determinar si las entradas al problema tienen características especiales que se puedan explotar en la búsqueda de una solución, o si puede usarse alguna solución aproximada sencilla, en vez de la solución exacta, difícil de calcular.

2. Algoritmos “Divide y Vencerás”.

1. Introducción.
2. Método general.
 - 2.1. LA APROXIMACIÓN Divide y Vencerás DyV
 - 2.2. Esquema general.
 - 2.3. Esquema recursivo.
3. Análisis de tiempos de ejecución.
4. Ejemplos de aplicación.
 - 4.1. Ordenación por mezcla (Mergesort).
 - 4.1. Ordenación rápida (Quicksort).
 - 4.3. Búsqueda del k -ésimo menor elemento.

2. Algoritmos “Divide y Vencerás”. Introducción.

- ❑ La técnica **divide y vencerás** consiste en descomponer el problema en un conjunto de subproblemas más pequeños de igual tipo o similar. Después se resuelven estos subproblemas y se combinan las soluciones para obtener la solución para el problema original.
- ❑ El nombre divide y vencerás también se aplica a veces a algoritmos que reducen cada problema a un único subproblema, como la búsqueda binaria para encontrar un elemento en una lista ordenada o el cálculo del factorial. Estos algoritmos pueden ser implementados más eficientemente que los algoritmos generales de “divide y vencerás” usando una serie de recursiones mediante la conversión en simples bucles. Bajo esta amplia definición, sin embargo, cada algoritmo que usa recursión o bucles puede ser tomado como un algoritmo de “divide y vencerás”. Esta subclase simple de problemas e denominan **decrementa y vencerás** o algoritmos de **simplificación**.
- ❑ La corrección de un algoritmo de “divide y vencerás”, está habitualmente probada una inducción matemática, y su **coste computacional** se determina resolviendo **relaciones de recurrencia**.

2. Algoritmos “Divide y Vencerás”. Método general.

□ LA APROXIMACIÓN Divide y Vencerás DyV

Esquema DyV recursivo. En cada paso de la recursión:

DIVIDIR el problema original en varios subproblemas que se dividen la talla de forma equilibrada.

VENCER (resolver) los subproblemas de forma recursiva. Si éstos son de un tamaño suficientemente pequeño, resolverlos de forma directa.

COMBINAR las soluciones para obtener la solución del problema original.

- Esquema recursivo.
- Coste de la parte de *división* de problemas en subproblemas
+ Coste de la parte de *combinación* de resultados.
- Parte “difícilcil” o “dura” del problema: *dividir* (ejem.: Quicksort) o *combinar* (ejem.: Mergesort).
- **Eficiencia**: problema de la división equilibrada. El esquema es eficiente cuando los subproblemas tienen una talla lo mas parecida posible.

2. Algoritmos “Divide y Vencerás”. Método general.

- **Requisitos** para aplicar divide y vencerás:
 - Necesitamos un **método** (más o menos **directo**) de resolver los problemas de tamaño pequeño.
 - El problema original debe poder dividirse fácilmente en un conjunto de subproblemas, del **mismo tipo** que el problema original pero con una resolución **más sencilla** (menos costosa).
 - Los subproblemas deben ser **disjuntos**: la solución de un subproblema debe obtenerse independientemente de los otros.
 - Es necesario tener un método de **combinar** los resultados de los subproblemas.
- Normalmente los subproblemas deben ser de tamaños parecidos.
- Como mínimo necesitamos que haya dos subproblemas.
- Si sólo tenemos un subproblema entonces hablamos de técnicas de **reducción** (o **simplificación**).
 - **Ejemplo:** Cálculo del factorial.
 $\text{Fact}(n) := n * \text{Fact}(n-1)$

2. Algoritmos “Divide y Vencerás”. Método general.

□ Esquema general:

función DivideVencerás (p: problema)

Dividir (p, p_1, p_2, \dots, p_k) // p_i son subproblemas de p

para $i = 1, 2, \dots, k$

$s_i = \text{Resolver}(p_i)$

fpara

return Combinar (s_1, s_2, \dots, s_k)

ffunción

- Normalmente para **resolver** los subproblemas se utilizan llamadas **recursivas** al mismo algoritmo (aunque no necesariamente).

2. Algoritmos “Divide y Vencerás”. Método general. Esquema recursivo.

- **Esquema recursivo:** Normalmente para resolver los subproblemas se utilizan llamadas recursivas al mismo algoritmo.

```
función DivideVencerás (P: problema) return Solución
    si P suficientemente pequeño entonces
        return Solución = SoluciónDirecta(p)           // caso base
    sino
        Dividir (P, p1, p2, ..., pk) // descomponer P en k subproblemas
        para i = 1, 2, ..., k
            si = DivideVencerás (pi)           // Resolver (pi)
        return Solución = Combinar (s1, s2, ..., sk)
    fsi
ffunción
```

2. Algoritmos “Divide y Vencerás”. Método general. Esquema recursivo.

- Aplicación de divide y vencerás: encontrar la forma de definir las **funciones genéricas**.
 - **Pequeño**: determina cuándo el problema es pequeño para aplicar la resolución directa.
 - **SoluciónDirecta**: método alternativo de resolución para tamaños suficientemente pequeños.
 - **Dividir**: función para descomponer un problema grande en subproblemas. Obtiene el índice, entre dos dados, por el que dividir el problema
 - **Combinar**: combina dos resultados de subproblemas para obtener el resultado del problema.
- Según este esquema, si los subproblemas en los que se divide el problema inicial no son suficientemente pequeños para convenir resolverlos directamente, el proceso de dividir en subproblemas se realiza repetidamente

2. Algoritmos “Divide y Vencerás”. Análisis de tiempos de ejecución.

- **Dividir** el problema original de talla n en a subproblemas de tamaño n/b . **Coste** $D(n)$.
 - Mergesort: Calcula el índice medio del vector. Coste $\Theta(1)$.
- **Resolver** los a subproblemas de tamaño n/b . **Coste** $aT(n/b)$.
 - Mergesort: Resuelve recursivamente los dos subproblemas de tamaño $n/2$. Coste $2T(n/2)$.
- **Combinar**. **Coste** $C(n)$.
 - Mergesort: Merge (fusión) de un vector de n elementos. Coste $\Theta(n)$.
- En **general**, el diseño Divide y Vencerás produce algoritmos recursivos cuyo tiempo de ejecución se puede expresar mediante una ecuación en recurrencia del tipo:

$$T(n) = \begin{cases} c & \text{si } n \leq n_0 \\ aT(n/b) + D(n) + C(n) & \text{si } n > n_0 \end{cases}$$
$$T(n) = \begin{cases} c & \text{si } n \leq n_0 \\ aT(n/b) + \Theta(n^p) & \text{si } n > n_0 \end{cases}$$

- La **solución** a esta ecuación puede alcanzar distintas complejidades dependiendo de los valores que tomen a y b , como se indica a continuación.

2. Algoritmos “Divide y Vencerás”. Análisis de tiempos de ejecución.

- Si el algoritmo realiza a llamadas recursivas de tamaño n/b , con $a \geq 1$ y $b > 1$, y la combinación requiere $C(n) = d \cdot n^p \in O(n^p)$, entonces: $T(n) = aT(n/b) + d \cdot n^p$

■ Suponiendo $n = b^k \Rightarrow k = \log_b n \Rightarrow T(b^k) = a \cdot T(b^{k-1}) + d \cdot b^{pk}$

- Se puede deducir (teorema maestro) que:

$$T(n) \in \left\{ \begin{array}{ll} O(n^{\log_b a}) & \text{Si } a > b^p \\ O(n^p \cdot \log n) & \text{Si } a = b^p \\ O(n^p) & \text{Si } a < b^p \end{array} \right\}$$

- **Generalización** del teorema maestro:

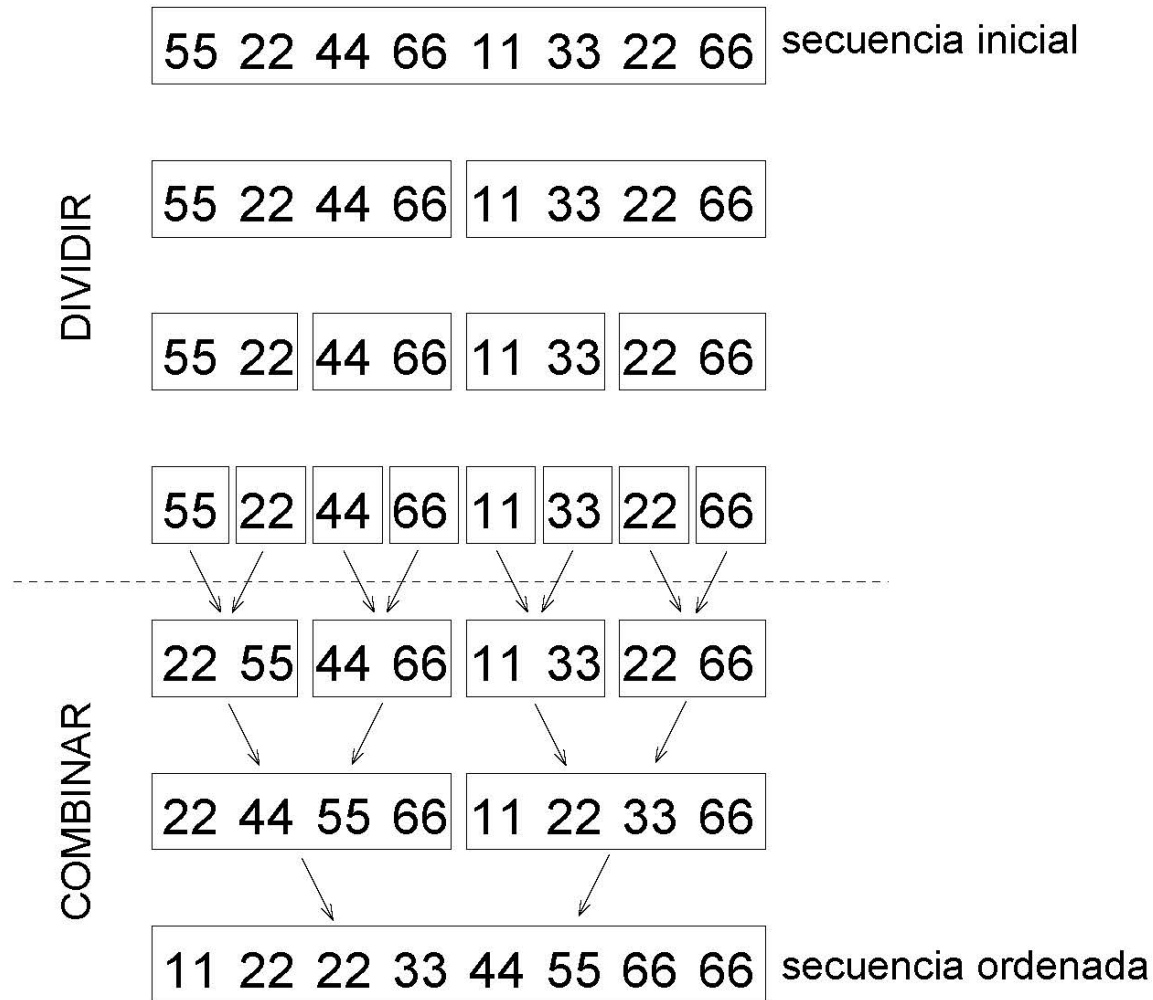
Solución a la ecuación $T(n) = aT(n/b) + \Theta(n^k \log^p n)$ con $a \geq 1, b > 1$ y $p \geq 0$

$$T(n) \in \left\{ \begin{array}{ll} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \cdot \log^{p+1} n) & \text{si } a = b^k \\ O(n^k \cdot \log^p n) & \text{si } a < b^k \end{array} \right\} \text{ Teorema maestro}$$

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. Ordenación por mezcla (mergesort).

- **Mergesort: Estrategia.** La **ordenación por mezcla (mergesort)** sigue exactamente el esquema de DyV:
- Para ordenar los elementos de un array de tamaño n :
 - *Dividir.* Divide la secuencia de n elementos a ordenar en dos subsecuencias de $n/2$ elementos cada una.
 - *Resolver.* Ordena las dos subsecuencias recursivamente utilizando Mergesort.
 - *Combinar.* Combina las dos subsecuencias ordenadas para generar la solución (mezclar). Se puede conseguir en $O(n)$.
 - *Pequeño.* La recursión finaliza cuando la secuencia a ordenar contiene un único elemento, en cuyo caso la secuencia ya está ordenada.

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Mergesort**: Ejemplo.



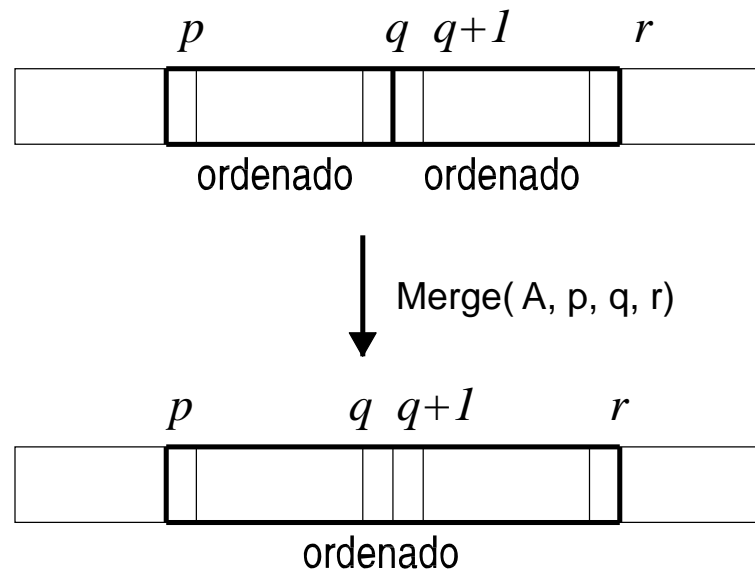
2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. Mergesort: Algoritmo.

```
MergeSort (A, p, r) /* Ordena un vector A desde p hasta r */  
  if  $p < r$  {  
    /* dividir en dos trozos de tamaño igual (o lo más parecido posible), es decir  $\lceil n/2 \rceil$  y  $\lfloor n/2 \rfloor$  */  
     $q = \lfloor (p+r)/2 \rfloor$ ;          /* Divide */  
    /* Resolver recursivamente los subproblemas */  
    MergeSort (A,p,q);        /* Resuelve */  
    MergeSort (A,q+1,r) ;    /* Resuelve */  
    /* Combinar: mezcla dos listas ordenadas en  $O(n)$  */  
    Merge (A,p,q,r);        /* Combina */  
  }
```

- Llamada inicial: **MergeSort(A,0,n-1)**, siendo n el número de elementos del vector A

2. Algoritmos “D y V”. Ejs de aplicación. Mergesort; **Merge**: Estrategia

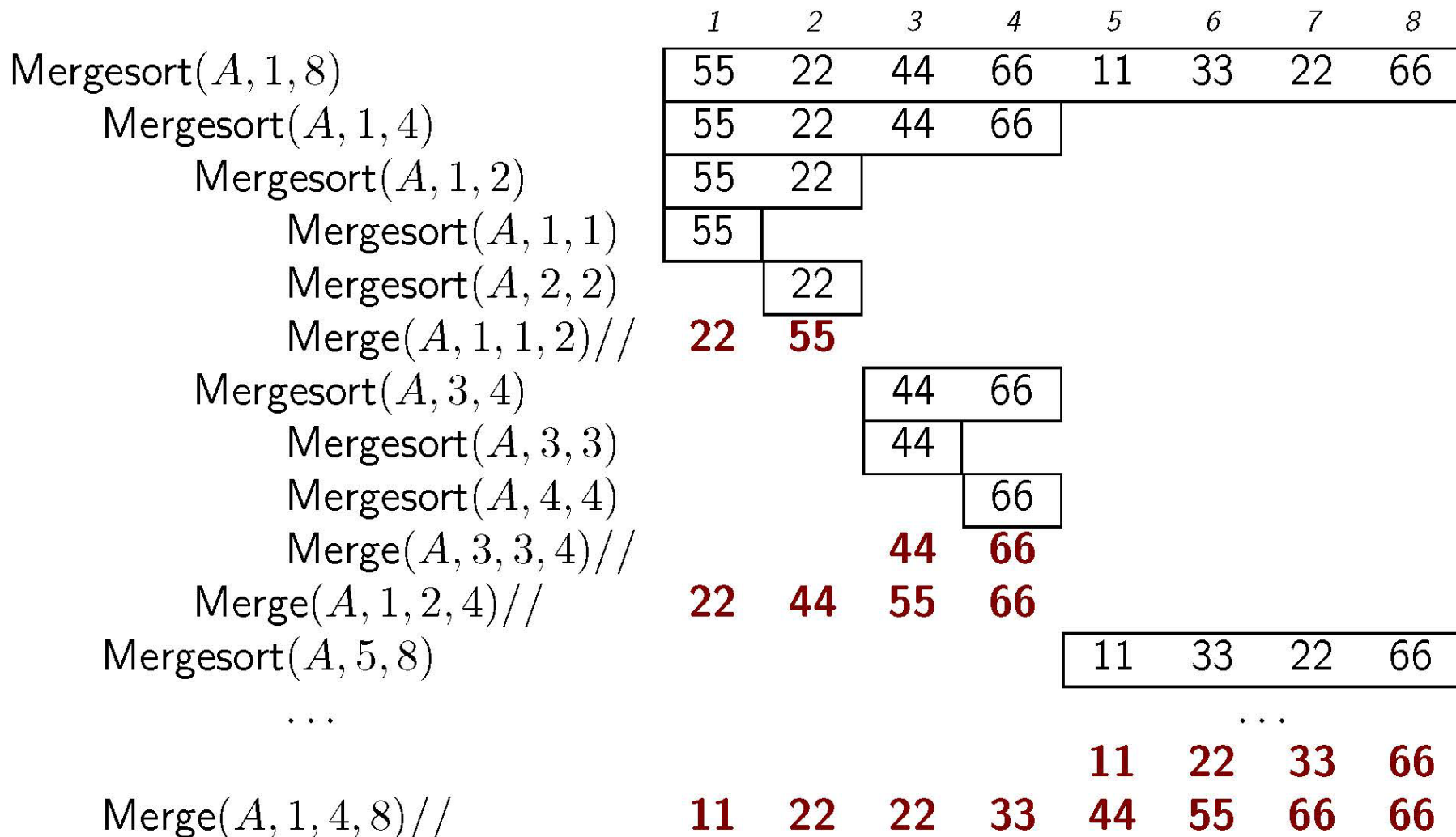
- **Problema:** Mezclar ordenadamente dos subsecuencias ya ordenadas.



2. Algoritmos “D y V”. Ejs de aplicación. Mergesort; **Merge: algoritmo.**

```
Merge( $A, p, q, r$ )
   $i = p; j = q + 1; k = 0;$ 
  while  $((i \leq q) \wedge (j \leq r))$  {
    if  $(A[i] \leq A[j])$  {
       $B[k] = A[i]; i = i + 1;$ 
    } else {
       $B[k] = A[j]; j = j + 1;$ 
    }
     $k = k + 1;$ 
  }
  while  $(i \leq q)$  /*...quedan todavia elementos de  $A[i..q]$ */ {
     $B[k] = A[i]; i = i + 1; k = k + 1;$ 
  }
  while  $(j \leq r)$  /*...quedan todavia elementos de  $A[j..r]$ */ {
     $B[k] = A[j]; j = j + 1; k = k + 1;$ 
  }
  for  $(k = p; k \leq r; k++)$  { /*...volcamos  $B$  en  $A$ */
     $A[k] = B[k - p];$ 
  } /*Coste:  $\Theta(n)$ */
```

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Mergesort: Traza**



2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Mergesort: Análisis.**

- $T(n)$ = tiempo de ejecución de Mergesort.
 - Si $n = 1$, el tiempo es constante.
 - Si $n > 1$ (supondremos por simplicidad que n es una potencia de 2):
- **Dividir:** Calcula el índice medio del vector: $\Theta(1)$.
- **Vencer:** Resuelve recursivamente los dos subproblemas, cada uno de ellos de talla $n/2$: **$2T(n/2)$** .
- **Combinar:** Merge (fusión) de un vector de n elementos: $\Theta(n)$.

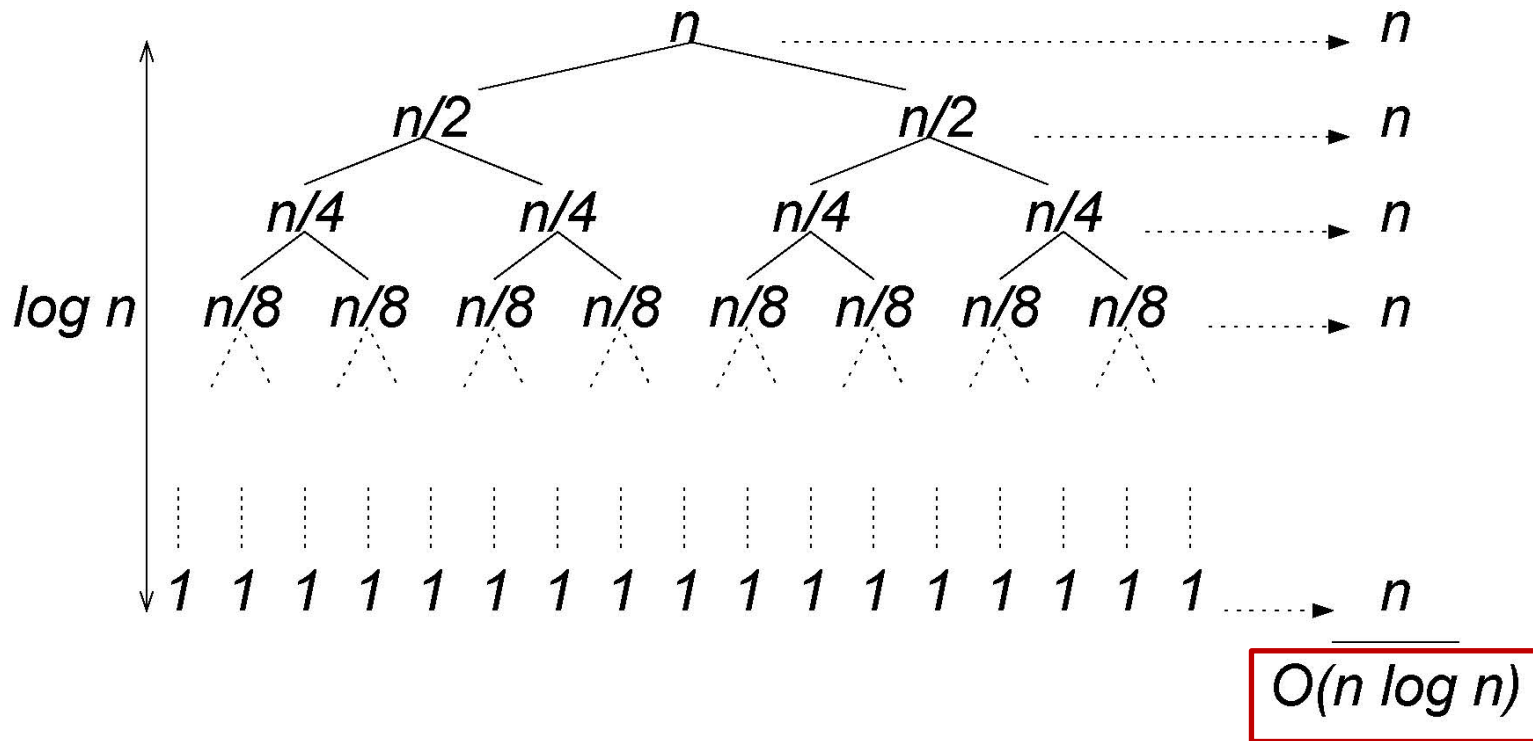
$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ 2T(n/2) + c_2n + c_3, & \text{si } n > 1 \end{cases}$$

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Mergesort: Análisis.**

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ 2T(n/2) + c_2n + c_3, & \text{si } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n/2) + c_2n + c_3 \\ &= 2(2T(n/2^2) + c_2n/2 + c_3) + c_2n + c_3 = \\ &= 2^2T(n/2^2) + 2c_2n + c_3(2 + 1) \\ &= 2^2(2T(n/2^3) + c_2n/2^2 + c_3) + 2c_2n + c_3(2 + 1) = \\ &= 2^3T(n/2^3) + 3c_2n + c_3(2^2 + 2 + 1) \\ &\vdots \{i \text{ iteraciones}\} \\ &= 2^iT(n/2^i) + ic_2n + c_3(2^{i-1} + \dots + 2^2 + 2 + 1) \\ T(n) &= \{n/2^i = 1, i = \log n\}; \sum_{j=0}^{i-1} 2^j = 2^i - 1 \\ &= nc_1 + c_2n \log n + c_3(n - 1) \in \boxed{\Theta(n \log n)} \end{aligned}$$

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Mergesort:** División de subproblemas **equilibrada**.



$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ 2T(n/2) + c_2n + c_3, & \text{si } n > 1 \end{cases}$$

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación.

Mergesort: División de subproblemas **NO equilibrada**

```
Mergesort_bad (A, p, r)
  if (p < r) {
    Mergesort_bad (A, p, p);
    Mergesort_bad (A, p + 1, r);
    Merge(A, p, p, r);
  }
```

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ T(1) + T(n-1) + c_2n + c_3, & \text{si } n > 1 \end{cases}$$

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ T(n-1) + c_2n + c_4, & \text{si } n > 1 \end{cases}$$

iiii **Coste: $\Theta(n^2)$** !!!!!

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Mergesort**: Posibles mejoras.

- ❑ Uno de los inconvenientes es que se necesita un espacio $2n$ debido al Merge. Existe un algoritmo de fusión que no utiliza un vector adicional en $O(n)$, pero con una constante muy alta que no lo hace apropiado.
- ❑ Otro inconveniente es la talla de la pila de recursión: la máxima profundidad de la pila será proporcional a $\log n$.
- ❑ Una mejora sobre el Mergesort consiste en no realizar llamadas recursivas cuando la talla del vector es pequeña, haciendo una ordenación por inserción o selección directa.
- ❑ Se pueden evitar movimientos redundantes de datos haciendo llamadas alternativamente sobre el vector original y el vector auxiliar.

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Mergesort**: Resumen.

- El tiempo que consume este algoritmo queda definido por la siguiente recurrencia

$$T(n) = \left\{ \begin{array}{ll} 1 & \text{si } n=1 \text{ (caso base)} \\ 2T(n/2) + \Theta(\text{Merge}) & \text{en otro caso} \end{array} \right\}$$

■ Para calcular la recurrencia necesitamos el algoritmo que realiza el Merge

- Analisis del algoritmo **Merge**: el algoritmo requiere de dos pasadas a los vectores a mergear, una para realizar el merge y otra para copiar del vector auxiliar al vector original, por tanto, $T(n) = 2n \Rightarrow T(n) = \Theta(n)$

- Por lo que la recurrencia del algoritmo de MergeSort es

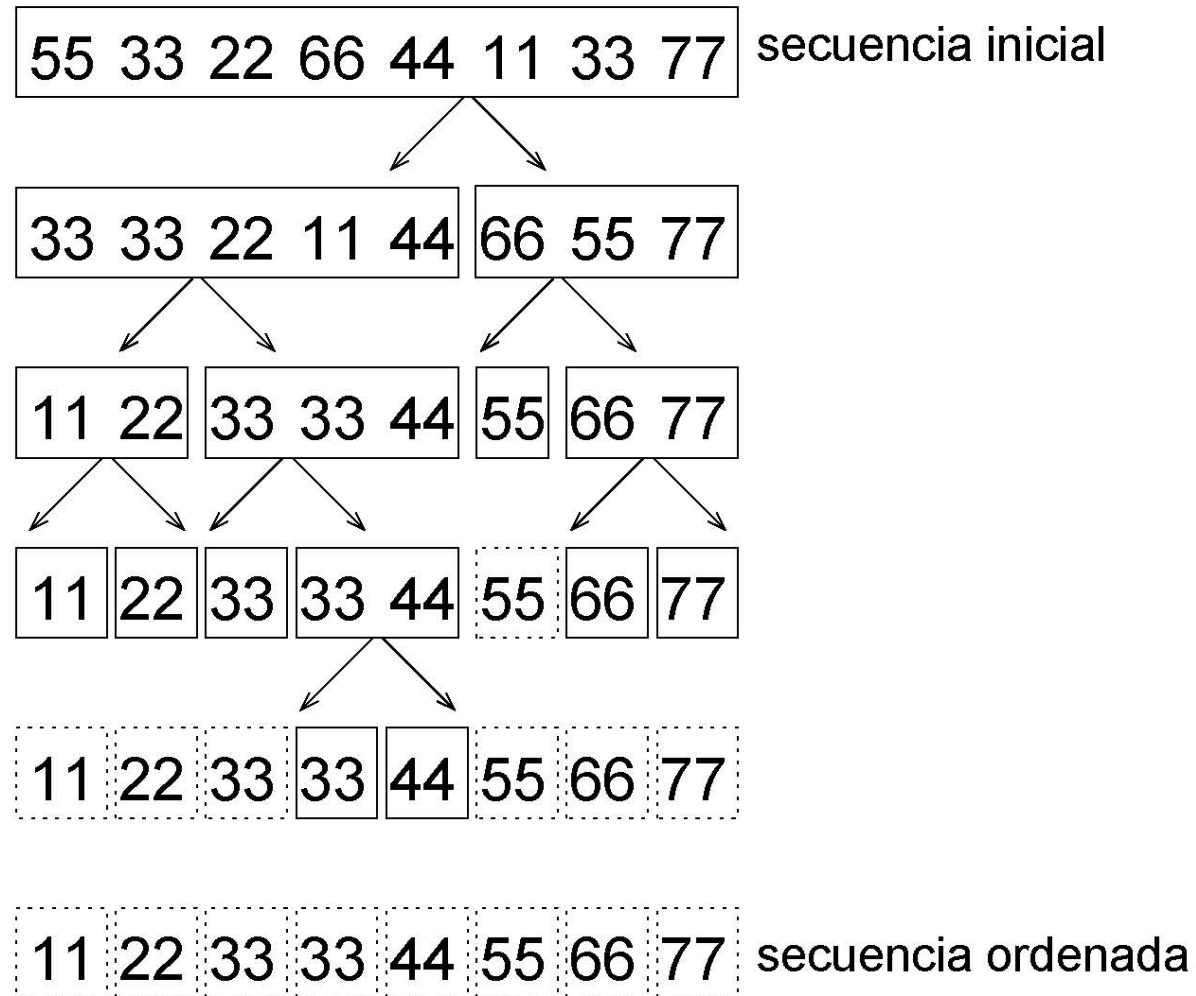
$$T(n) = \left\{ \begin{array}{ll} 1 & \text{si } n=1 \text{ (caso base)} \\ 2T(n/2) + n & \text{en otro caso} \end{array} \right\}$$

- Recurrencia conocida. Aplicando el teorema maestro: **MergeSort** $\in \Theta(n \log n)$
- $T(n) = \Theta(n \log n)$ Peor , mejor y caso medio son iguales ya que el tiempo que consume el algoritmo no depende de la disposición inicial de los elementos del vector

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. Ordenación por partición (Quicksort)

- Quicksort, al igual que Mergesort, se basa en la estrategia DyV. La diferencia estriba en que la parte costosa del algoritmo está en el paso de “**dividir**”. Los tres pasos de la recursión para ordenar un subvector $A[p..r]$ son:
 - **Dividir**: El vector $A[p..r]$ se particiona (reorganiza) usando un procedimiento **Partition** (*Pivote*) en dos subvectores $A[p..q]$ y $A[q+1..r]$, de forma que los elementos de $A[p..q]$ son menores o iguales que los de $A[q+1..r]$. El índice q se calcula también en el procedimiento de partición.
 - **Resolver** (Vencer): Los dos subvectores $A[p..q]$ y $A[q+1..r]$ se ordenan recursivamente utilizando Quicksort.
 - **Combinar**: Como los subvectores se ordenan en su lugar correspondiente, no hay que hacer nada para combinar las soluciones: el vector completo $A[p..r]$ ya está ordenado.
 - **Pequeño**: La recursión finaliza cuando la secuencia a ordenar contiene un único elemento, en cuyo caso la secuencia ya está ordenada.

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Quicksort**: Ejemplo



2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Quicksort**: Algoritmo.

QuickSort (A, p, r)

```
if ( $p < r$ ) {  
     $q = \text{Partition}(A, p, r)$   
    QuickSort ( $A, p, q$ )  
    QuickSort ( $A, q+1, r$ )  
}
```

- Llamada inicial: **QuickSort**($A, 0, n-1$), siendo n el número de elementos del vector A

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación.

Quicksort: Algoritmo **PARTITION**.

- El objetivo de la función de partición es reorganizar el vector $A[p..r]$ dejando en $A[p..q]$ elementos menores o iguales que en $A[q + 1..r]$, de forma que las particiones resulten lo más equilibradas posible.

Posibles soluciones:

- **Mejor** algoritmo posible: Elegir como pivote la mediana (deja ambas partes equilibradas) → algoritmo muy costoso.
- Elegir como pivote el primer elemento $A[p]$ → algoritmo **Partition**
- Otras opciones: pivote aleatorio, mediana de un subconjunto de elementos, . . .

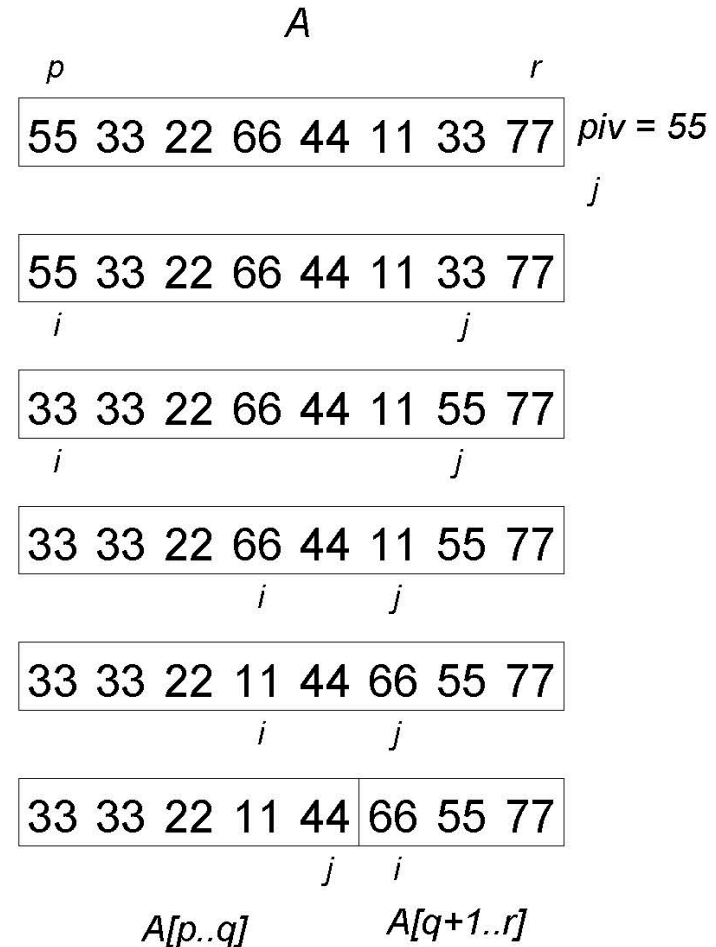
2. Algoritmos “Divide y Vencerás”. Ejs de aplicación.

Quicksort: Algoritmo **PARTITION**.

```

Partition( $A, p, r$ )
   $piv = A[p]; i = p - 1; j = r + 1;$ 
  do{
    do{
       $j = j - 1;$ 
    }while ( $A[j] > piv$ )
    do{
       $i = i + 1;$ 
    }while ( $A[i] < piv$ )
    if ( $i < j$ ) {
       $A[i] \longleftrightarrow A[j];$ 
    }
  }while ( $i < j$ )
  return  $j$ ;

```



Coste: $\Theta(n)$

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Quicksort: Traza.**

Quicksort($A, 1, 8$)

Partition($A, 1, 8$) $\rightarrow 5$

Quicksort($A, 1, 5$)

Partition($A, 1, 5$) $\rightarrow 2$

Quicksort($A, 1, 2$)

Partition($A, 1, 2$) $\rightarrow 1$

Quicksort($A, 1, 1$) //

Quicksort($A, 2, 2$) //

Quicksort($A, 3, 5$)

Partition($A, 3, 5$) $\rightarrow 3$

Quicksort($A, 3, 3$) //

Quicksort($A, 4, 5$)

Partition($A, 4, 5$) $\rightarrow 4$

Quicksort($A, 4, 4$) //

Quicksort($A, 5, 5$) //

Quicksort($A, 6, 8$)

...

1	2	3	4	5	6	7	8
55	33	22	66	44	11	33	77
33	33	22	11	44	66	55	77
33	33	22	11	44			
11	22	33	33	44			
11	22						
11	22						

11

22

33	33	44
33	33	44

33

33	44
33	44

33

44

66	55	77
----	----	----

...

11 22 33 33 44 55 66 77
55 66 77

2. Algoritmos “Divide y Vencerás”. Ejs de aplicación. **Quicksort: Análisis.**

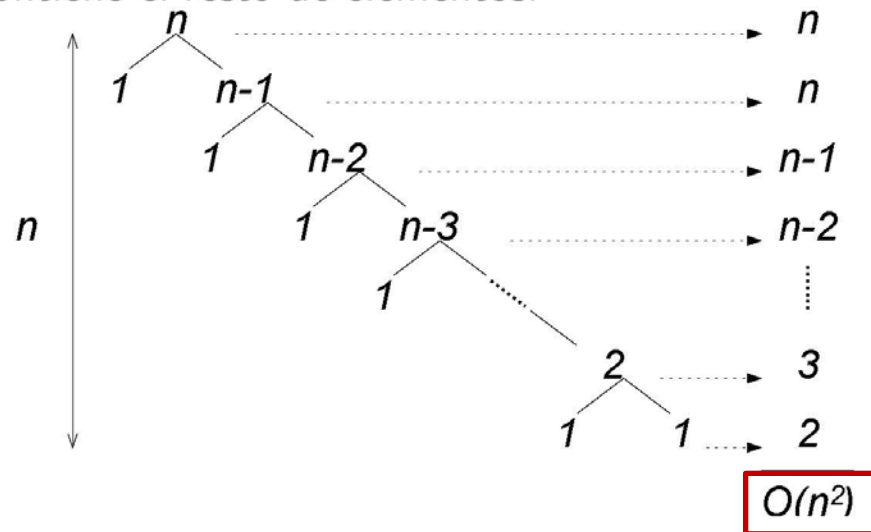
Sea n el número de elementos del vector. El coste de Partition es $O(n)$ y reorganiza el vector en dos subvectores de tamaño i y $(n - i)$. La relación de recurrencia es:

$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ T(i) + T(n - i) + k'n, & \text{si } n > 1 \end{cases}$$

2. Algoritmos “DyV”. Ejs de aplicación.

Quicksort: Análisis (**Peor** caso: Partición desequilibrada).

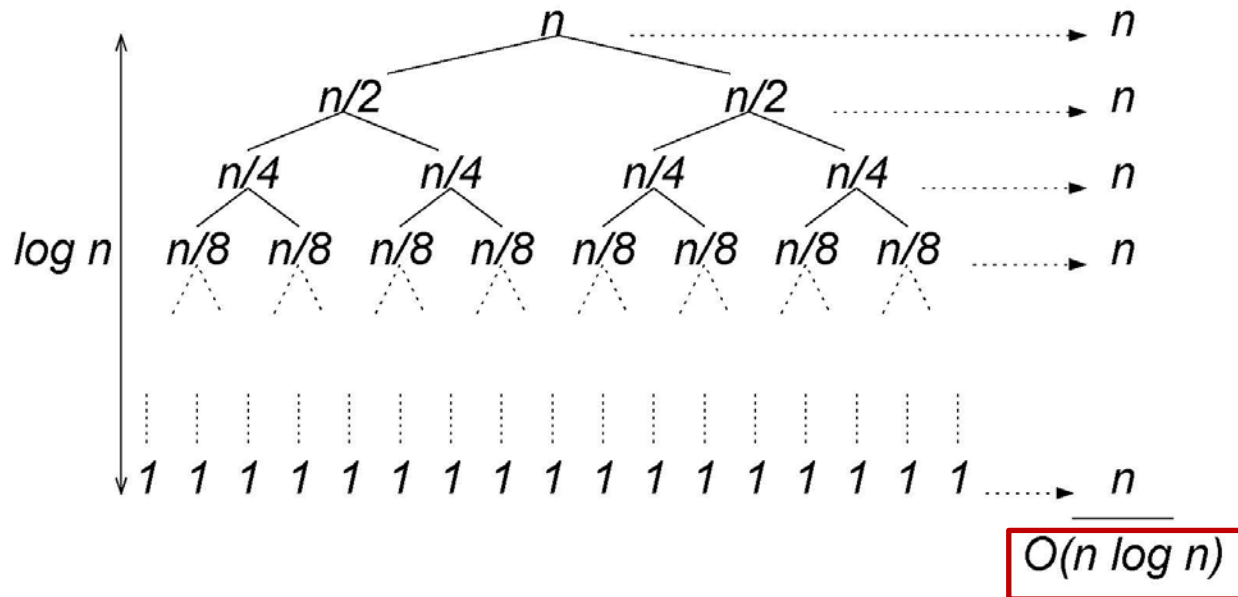
Peor caso: supongamos que todos los elementos del vector son diferentes y que están ordenados de menor a mayor. En este caso, el algoritmo Partition divide el vector original en dos subvectores: uno con un único elemento (el mínimo, que ha actuado como pivote), mientras que el otro contiene el resto de elementos.



$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ T(1) + T(n-1) + k'n, & \text{si } n > 1 \end{cases}$$





2. Algoritmos “DyV”. Ejs de aplicación. Quicksort: Análisis (**Mejor** caso: Partición equilibrada).

Mejor caso: será aquel en el que en cada llamada a Partition deja las dos partes equilibradas.



$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ 2T(n/2) + k'n, & \text{si } n > 1 \end{cases}$$

2. Algoritmos “DyV”. Ejs de aplicación. Quicksort: Análisis (**Caso promedio**).

Instancia (I)	Talla particiones		Prob. (p)
	1	(n-1)	1/n
	2	(n-2)	1/n
	3	(n-3)	1/n
⋮	⋮	⋮	⋮
	(n-1)	1	1/n

$$T(n) = k'n + \frac{1}{n}(T(1) + T(n-1))$$

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

$$+ \frac{1}{n}(T(3) + T(n-3))$$

⋮

$$+ \frac{1}{n}(T(n-1) + T(1))$$

$$= k'n + \frac{2}{n} \sum_{i=1}^{n-1} T(i)$$

$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ k'n + 2/n \sum_{i=1}^{n-1} T(i), & \text{si } n > 1 \end{cases}$$

2. Algoritmos “DyV”. Ejs de aplicación. Quicksort: Análisis (**Caso promedio**).

$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ k'n + 2/n \sum_{i=1}^{n-1} T(i), & \text{si } n > 1 \end{cases}$$

Se puede demostrar que, $\forall n \geq 2$, $T(n) \leq k''n \log n$, siendo $k'' = 2k' + k$. Por tanto, el coste promedio de Quicksort es $O(n \log n)$.

$$\boxed{n = 2} \quad T(2) = 2k' + \sum_{i=1}^1 T(i) = 2k' + k \leq (2k' + k)2 \log 2$$

$$\boxed{n > 2} \quad T(n) = k'n + 2/n \sum_{i=1}^{n-1} T(i) = k'n + 2/n T(1) + 2/n \sum_{i=2}^{n-1} T(i)$$

$$(H.I.) \leq k'n + 2k/n + 2/n \sum_{i=2}^{n-1} k''i \log i \leq k''n \log n \in \boxed{O(n \log n)}$$

2. Algoritmos “DyV”. Ejs de aplicación. Quicksort: **Posibles mejoras.**

□ **Posibles mejoras de Quicksort**

- No realizar llamadas recursivas cuando la talla del vector es pequeña, haciendo una ordenación por inserción o selección directa.
- Elegir el valor utilizado como pivote de forma aleatoria entre todos los valores del vector. Para ello, antes de realizar cada partición, se intercambiaría el valor del elemento utilizado como pivote con el valor del elemento seleccionado al azar.

2. Algoritmos “DyV”. Ej de aplicación: Búsqueda del k -ésimo menor elemento.

- **Problema:** Dado un vector de n elementos, el problema de la selección consiste en buscar el k -ésimo menor elemento.
- **Solución directa:** ordenar los n elementos y acceder al k -ésimo. Coste: $O(n \log n)$
- **Solución DyV:** Algoritmo más eficiente utilizando la idea del algoritmo Partition:
 - **Dividir (Partition):** El vector $A[p..r]$ se particiona (reorganiza) en dos subvectores $A[p..q]$ y $A[q+1..r]$, de forma que los elementos de $A[p..q]$ son menores o iguales que el pivote y los de $A[q+1..r]$ son mayores o iguales.
 - **Combinar:** Si $k \leq q$, entonces el k -ésimo menor estaría en $A[p..q]$. Si no, estará en $A[q+1..r]$.
 - **Resolver** (Vencer): Buscamos en el subvector correspondiente haciendo llamadas recursivas al algoritmo.
 - **Pequeño:** La recursión finaliza cuando el subvector contiene un único elemento, en cuyo caso habremos encontrado el k -ésimo menor.

2. Algoritmos “DyV”. Ej de aplicación: Búsqueda del k -ésimo menor elemento. Algoritmo.

```
Select ( $A, p, r, k$ )  
    if ( $p == r$ ) {  
        return  $A[p]$ ;  
    }  
     $q = \text{Partition}(A, p, r)$ ;  
     $i = q - p + 1$ ;    /* $i$  es el número de elementos en el primer subvector*/  
    if ( $k \leq i$ ) {  
        return  $\text{Select}(A, p, q, k)$ ;    /*...buscamos en  $A[p..q]$ */  
    } else {  
        return  $\text{Select}(A, q + 1, r, k - i)$ ;    /*...buscamos en  $A[q + 1..r]$ */  
    }
```

□ Llamada inicial: $\text{Select}(A, 1, n, k)$, siendo n el número de elementos del vector A .

2. Algoritmos “DyV”. Ej de aplicación: Búsqueda del k -ésimo menor elemento. Traza.

Instancia: $A = \{55, 33, 22, 66, 44, 11, 33, 77\}$; $k = 3$

Vector ordenado: $A' = \{11, 22, \mathbf{33}, 33, 44, 55, 66, 77\}$

	1	2	3	4	5	6	7	8
Select($A, 1, 8, 3$)	55	33	22	66	44	11	33	77
Partition($A, 1, 8$) $\rightarrow 5$	33	33	22	11	44	66	55	77

Select($A, 1, 5, 3$)	33	33	22	11	44
Partition($A, 1, 5$) $\rightarrow 2$	11	22	33	33	44

Select($A, 3, 5, 1$)	33	3	44
Partition($A, 3, 5$) $\rightarrow 3$	33	33	44

Select($A, 3, 3, 1$) **33**

2. Algoritmos “DyV”. Ej de aplicación: Búsqueda del k -ésimo menor elemento. Select Iterativo.

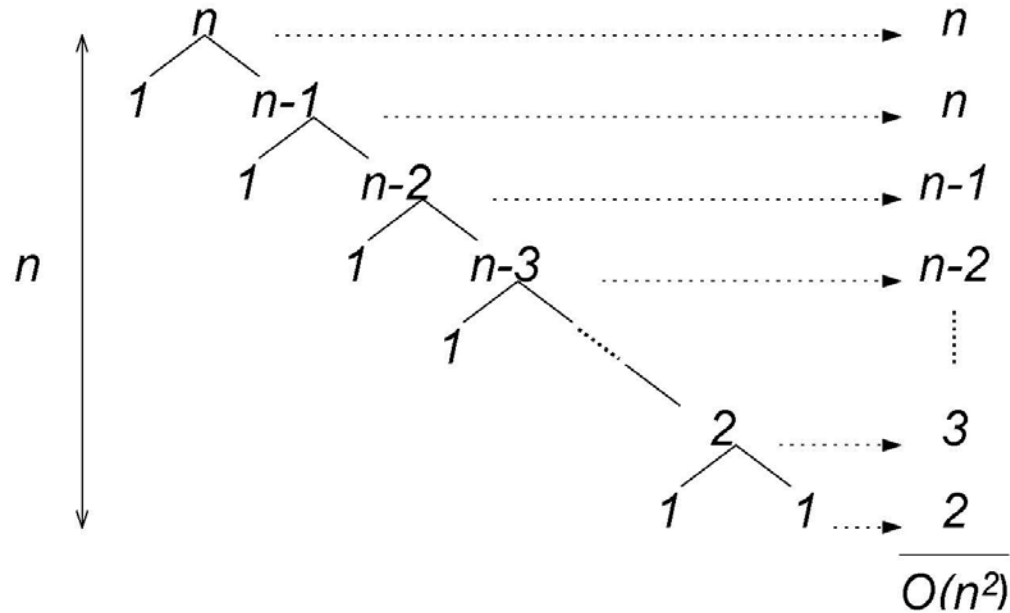
```
Select( $A, p, r, k$ )
  if ( $p == r$ ) {
    return  $A[p]$ ;
  }
   $q = \text{Partition}(A, p, r)$ ;
   $i = q - p + 1$ ;
  if ( $k \leq i$ ) {
    return  $\text{Select}(A, p, q, k)$ ;
  } else {
    return  $\text{Select}(A, q + 1, r, k - i)$ ;
  }
```

```
SelectIt( $A, p, r, k$ )
  while ( $p < r$ ) {
     $q = \text{Partition}(A, p, r)$ ;
    if ( $k \leq q$ ) {
       $r = q$ ; /*...buscamos en  $A[p..q]$ */
    } else {
       $p = q + 1$ ; /*...buscamos en  $A[q + 1..r]$ */
    }
  }
  return  $A[p]$ ;
```

- Realizar una traza de Select y SelectIt con
 $A = \{31, 23, 90, 0, 77, 52, 49, 87, 60, 15\}$ y $k = 7$.

2. Algoritmos “DyV”. Ej de aplicación: Búsqueda del k -ésimo menor elemento. Análisis (Peor caso: desequilibrado).

Peor caso: vector ordenado de forma no decreciente y buscamos el elemento mayor ($k = n$).
Coste: $O(n^2)$.



$$T(n) = \begin{cases} c, & \text{si } n \leq 1; \\ T(n-1) + c'n, & \text{si } n > 1 \end{cases}$$

2. Algoritmos “DyV”. Ej de aplicación: Búsqueda del k -ésimo menor elemento. **Análisis (Mejor caso).**

Mejor caso: el elemento a buscar es el menor ($k = 1$) o mayor ($k = n$) y éste actúa como pivote \rightarrow Una única llamada a Partition. Coste: $O(n)$.

0 7 3 5 9

↓ Select(A, 1, 5, 1)
↓ Partition(A, 1, 5)=1

0 | 7 3 5 9

↓ Select(A, 1, 1, 1)
↓ return "0"

9 0 7 3 5

↓ Select(A, 1, 5, 5)
↓ Partition(A, 1, 5)=4

5 0 7 3 | 9

↓ Select(A, 5, 5, 1)
↓ return "9"

2. Algoritmos “DyV”. Ej de aplicación: Búsqueda del k -ésimo menor elemento. **Análisis (Caso promedio).**

- Peor caso $\in O(n^2)$
- Mejor caso $\in O(n)$
- Promedio: con ciertas asunciones de aleatoriedad, el algoritmo Select tiene un coste $\in \Theta(n)$ (pág. 188, Cormen 90), (pág. 167, Horowitz 98).

Supongamos que en cada llamada a Partition, el problema se reduce a la mitad. Relación de recurrencia:

$$T(n) = \begin{cases} c, & \text{si } n \leq 1; \\ T(n/2) + c'n, & \text{si } n > 1 \end{cases}$$

2. Algoritmos “DyV”. Ej de aplicación: Búsqueda del k -ésimo menor elemento. **Análisis (Caso promedio).**

$$T(n) = \begin{cases} c, & \text{si } n \leq 1; \\ T(n/2) + c'n, & \text{si } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n/2) + c'n \\ &= (T(n/2^2) + c'n/2) + c'n = T(n/2^2) + (n + n/2)c' \\ &= (T(n/2^3) + c'n/2^2) + (n + n/2)c' = T(n/2^3) + n(1 + 1/2 + 1/2^2)c' \\ &\dots \\ &= T(n/2^i) + c'n \sum_{j=0}^{i-1} 1/2^j \\ &\quad // \text{Serie geométrica: } \sum_{j=0}^i x^j = \frac{x^{i+1} - 1}{x - 1}; // \sum_{j=0}^{i-1} 1/2^j = \frac{1/2^i - 1}{1/2 - 1} = \frac{2(2^i - 1)}{2^i} \\ &= T(n/2^i) + n \frac{2(2^i - 1)}{2^i} c' \quad \{n/2^i = 1, i = \log n\} \\ &= T(1) + n \frac{2(n - 1)}{n} c' = c_1 + 2(n - 1)c' \in \boxed{O(n)} \end{aligned}$$

2. Algoritmos “Divide y Vencerás”.Conclusiones

- **Idea básica Divide y Vencerás:** dado un problema, descomponerlo en partes, resolver las partes y juntar las soluciones.
- Idea muy sencilla e intuitiva, pero en los problemas reales de interés:
 - Pueden existir muchas formas de descomponer el problema en subproblemas \Rightarrow Quedarse con la mejor.
 - Puede que no exista ninguna forma viable, los subproblemas no son independientes \Rightarrow Descartar la técnica.
- Divide y vencerás requiere la existencia de un **método directo** de resolución:
 - Tamaños pequeños: solución directa.
 - Tamaños grandes: división y combinación.
 - Establecer el límite pequeño/grande.

3. Algoritmos voraces (“Greedy”).

1. Introducción.
2. Esquema
 - 2.2. Funcionamiento
 - 2.1. Elementos de los que consta la técnica
 - 2.3. Ejemplo: problema del cambio de moneda.
3. Análisis de tiempos de ejecución.
4. Ejemplos de aplicación.
 - 4.1. Selección de actividades.
 - 4.2. Problema de la mochila.
 - 4.3. Algoritmos de grafos que utilizan una estrategia voraz
 - 4.3.1. Problema del recubrimiento mínimo en un grafo
 - 4.3.1.1. Algoritmo de Kruskal.
 - 4.3.1.2. Algoritmo de Prim.
 - 4.3.2. Problema del camino mínimo.
 - 4.3.2.1. El algoritmo de Dijkstra.

3. Algoritmos voraces. Introducción. Ejemplos.

- Cajero automático: El desglose de una cantidad de dinero con el menor número posible de monedas y billetes.
- La mochila con fraccionamiento: La carga de un contenedor con diferentes materiales fraccionables que reportan beneficios distintos de modo que maximicemos el beneficio total sin exceder la cantidad máxima de carga.
- La asignación de un recurso único a una serie de actividades de las que se conoce el tiempo de inicio y finalización de modo que se atienda el mayor número de tareas.
- Código Huffman: Diseño de la codificación de un alfabeto de modo que se minimize la longitud esperada (en bits) de los mensajes.
- Algoritmos de grafos que utilizan una estrategia voraz:
 - Los algoritmos de Kruskal y de Prim para el problema del recubrimiento mínimo en un grafo no dirigido, ponderado y conexo.
 - El algoritmo de Dijkstra para el problema del camino más corto de un vértice a otro en un grafo ponderado.
- Problemas que admiten solución aproximada aunque no óptima con una estrategia voraz:
 - El coloreado de un grafo.
 - El problema del viajante de comercio.

3. Algoritmos voraces. Introducción.

- Es uno de los esquemas más simples y al mismo tiempo de los más utilizados.
- Típicamente se emplea para resolver **problemas de optimización**:
 - existe una entrada de tamaño **n** que son los **candidatos** a formar parte de la solución;
 - existe un subconjunto de esos **n** candidatos que satisface ciertas restricciones: se llama **solución factible** (prometedoras);
 - hay que obtener la solución factible que maximice o minimice una cierta **función objetivo**: se llama **solución óptima**.

3. Algoritmos voraces. Esquema. Funcionamiento.

- Un **algoritmo voraz** funciona por pasos:
 - Inicialmente partimos de una solución vacía.
 - En cada paso se escoge el siguiente elemento para añadir a la solución, entre los candidatos.
 - Una vez tomada esta decisión no se podrá deshacer.
 - El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución.
- **Ejemplo:** “algoritmo de comprar 2 kilos de manzanas en el mercado”.
Características básicas del algoritmo:
 - Inicialmente empezamos con una solución “vacía”, sin manzanas.
 - Función de **selección**: seleccionar la mejor manzana del montón o la que “parezca” que es la mejor.
 - Examinar la manzana detenidamente y decidir si se coge o no.
 - Si no se coge, se aparta del montón.
 - Si se coge, se mete a la bolsa (y ya no se saca).
 - Una vez que tenemos 2 kilos paramos.

3. Algoritmos voraces. Esquema. Elementos de los que consta la técnica

- Se puede generalizar el proceso intuitivo a un esquema algorítmico general.
- En cada paso tenemos los siguientes conjuntos de elementos:
 - **C:** Conjunto de elementos **candidatos, pendientes** de seleccionar (inicialmente todos).
 - **S:** Candidatos seleccionados para la **solución**.
 - **R:** Candidatos seleccionados pero **rechazados** después.
- Los candidatos depende de cada problema, el **qué** o **cuáles** sean.

3. Algoritmos voraces. Esquema. Elementos de los que consta la técnica

□ Esquema general de un algoritmo voraz:

```
función voraz( C: CjtoCandidatos  var S: CjtoSolución )  
    S  $\leftarrow \emptyset$   
    mientras (C  $\neq \emptyset$ ) Y NO solución(S) hacer  
        x  $\leftarrow$  seleccionar(C)  
        C  $\leftarrow$  C - {x}  
        si factible(S, x) entonces  
            insertar(S, x)  
        fin si  
    finmientras  
    si NO solución(S) entonces  
        devolver “No se puede encontrar solución”  
    sino  
        devolver S  
    fin si  
finfunción
```

3. Algoritmos voraces. Esquema. Elementos de los que consta la técnica

Funciones genéricas

- ❑ **solución (S).** Comprueba si un conjunto de candidatos es una solución (independientemente de que sea óptima o no).
- ❑ **seleccionar (C).** Devuelve el elemento más “prometedor” del conjunto de candidatos pendientes (no seleccionados ni rechazados).
- ❑ **factible (S, x).** Indica si a partir del conjunto **S** y añadiendo **x**, es posible construir una solución (posiblemente añadiendo otros elementos).
- ❑ **insertar (S, x).** Añade el elemento **x** al conjunto solución. Además, puede ser necesario hacer otras cosas.
- ❑ Función **objetivo (S).** Dada una solución devuelve el coste asociado a la misma (resultado del problema de optimización).

3. Algoritmos voraces. Ejemplo: problema del cambio de moneda (I).

□ Ejemplo: problema del cambio de moneda.

1. Disponemos de monedas de distintos valores: de 1, 2, 5, 10, 20 y 50 céntimos de euro, y de 1 y 2 euros. Supondremos una cantidad ilimitada de cada una.

□ Construir un algoritmo que dada una cantidad **P** devuelva esa cantidad con monedas de estos tipos, usando un número **mínimo** de monedas.

P. ej.: para devolver 3.89 €: 1 moneda de 2€, 1 moneda de 1€, 1 moneda de 50 c€, 1 moneda de 20 c€, 1 moneda de 10 c€, 1 moneda de 5 c€ y 2 monedas de 2 c€.

□ Podemos aplicar la técnica **voraz**: en cada paso añadir una moneda nueva a la solución actual, hasta que el valor llegue a **P**.

□ Para este sistema monetario encuentra siempre la solución óptima.

2. Puede no encontrar la solución óptima: supongamos que tenemos monedas de 100, 90 y 1. Queremos devolver 180.

□ Algoritmo voraz: 1 moneda de 100 y 80 monedas de 1: total 81 monedas.

□ Solución óptima: 2 monedas de 90: total 2 monedas.

➤ Puede haber solución y no encontrarla.

➤ Puede no haber solución y no lo detecta.

3. Algoritmos voraces. Ejemplo: problema del cambio de moneda(II)

- **Candidatos iniciales:** todos los tipos de monedas disponibles. Supondremos una cantidad ilimitada de cada tipo.
- **Solución:** conjunto de monedas que suman la cantidad **P**.
- Una solución será de la forma **($x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$)**, donde x_i es el número de monedas de tipo **i**. Suponemos que la moneda **i** vale **c_i** .
 - Ej anterior: $P = 3.89$; $C = \{1, 2, 5, 10, 20, 50, 100, 200\}$; $X = \{0, 2, 1, 1, 1, 1, 1, 1\}$
- **Formulación:** **Minimizar** $\sum_{i=1}^8 x_i$, sujeto a $\sum_{i=1}^8 x_i c_i = P$, $x_i \geq 0$
- **Funciones del esquema:**
 - **inicialización.** Inicialmente $x_i = 0$, para todo $i = 1..8$
 - **solución.** El valor actual será solución si $\sum x_i \cdot c_i = P$
 - **objetivo.** La función a minimizar es $\sum x_i$, el número de monedas resultante.
 - **seleccionar.** Elegir en cada paso la moneda de valor más alto posible, pero menor que el valor que queda por devolver.
 - **factible.** Valdrá siempre verdad.
- En lugar de seleccionar monedas de una en una, podemos usar la división entera y coger todas las monedas posibles de mayor valor.

3. Algoritmos voraces. Ejemplo: problema del cambio de moneda(III)

- **Implementación.** Usamos una variable local **actual** para acumular la cantidad devuelta hasta este punto. Da el cambio de P utilizando el menor número posible de monedas

```
funcion Devolver-cambio (int P): conjunto de monedas(X)
  const C={1,2,5,10,20,50,100,200} // C=monedas disponibles; conj. candidatos
  int X[1..N] ;                      // X= conjunto que contendrá la solución
  actual = 0                          // suma acumulada de la cantidad procesada
  para i = 1 hasta N
    X[i] = 0                          // inicialización(X)
  fpara
    mientras actual ≠ P               // no solución(X)
      j = el mayor elemento de C tal que C[j] ≤ (P-actual) // seleccionar(C)
      si j=0 entonces                 // Si no existe ese elemento => no factible(j)
        devolver "No existe solución";
      fsi
        X[j] = (P-actual) div C[j]    // insertar(C,X)
        actual = actual + C[j]*X[j]
      fmientras
        devolver X                    // objetivo(X)
ffuncion
```

- **Orden de complejidad** del algoritmo: podemos encontrar el siguiente elemento en un tiempo constante (ordenando las monedas): **O(n)**.

3. Algoritmos voraces. Análisis de tiempos de ejecución.

- El orden de complejidad depende:
 - del número de candidatos
 - de las funciones básicas a utilizar,
 - del número de elementos de la solución.
- Si tenemos **n**=número de elementos de C y **m**= número de elementos de una solución.
- Repetir, como máximo **n** veces y como mínimo **m**:
 - Comprobar si el valor actual es **solución**: **f(m)**.
 - Normalmente $O(1)$ ó $O(m)$.
 - **Selección** de un elemento entre los candidatos: **g(n)**.
 - Entre $O(1)$ y $O(n)$.
 - La función **factible** es parecida a **solución**, pero con una solución parcial **h(n)**.
 - La unión de un nuevo elemento a la solución (**insertar**) puede requerir otras operaciones de cálculo, **j(n, m)**.
- Tiempo de ejecución **genérico**:

$$t(n,m) \in O(n \cdot (f(m) + g(n) + h(m)) + m \cdot j(n, m))$$
- En la práctica los algoritmos voraces **suelen ser** bastante **rápidos**, encontrándose dentro de órdenes de complejidad **polinomiales**.

3. Algoritmos voraces. Ejemplos de aplicación. Selección de actividades.

☐ Selección de actividades.

- Se tienen n actividades (por ej., clases) que deben usar un recurso (por ej., un aula) que sólo puede ser usado por una actividad en cada instante (de manera exclusiva durante un intervalo de tiempo).
- Cada actividad está numerada. La actividad número i necesita el recurso durante el intervalo de tiempo $[c_i, f_i)$. (Naturalmente $c_i < f_i$). Cada actividad i :
 - ☐ c_i : instante de comienzo
 - ☐ f_i : instante de finalización
 - ☐ debe hacerse durante $[c_i, f_i)$.
- Dos actividades i, j se dicen **compatibles** si los intervalos $[c_i, f_i)$ y $[c_j, f_j)$. (no se superponen (i.e., $(c_i \geq f_j)$ ó $(c_j \geq f_i)$)).
- **El problema de selección de actividades** consiste en seleccionar un subconjunto de las actividades propuestas de manera que sean compatibles entre sí y que sea un subconjunto de cardinal máximo de entre los que forman subconjuntos compatibles.

3. Algoritmos voraces. Ejemplos de aplicación. **Selección de actividades.**

□ Selección de actividades.

- La parte fundamental de un algoritmo voraz es su **criterio de selección voraz** :
 - Seleccionar la actividad que comience más temprano.
 - Seleccionar la actividad de menos duración.
 - Seleccionar la actividad que termine más pronto.
 - Seleccionar la actividad que menos solapamientos tenga con el resto de actividades candidatas.

- Para que puedan formar una solución, debemos adjuntarles el *criterio de satisfacibilidad* :
 - que no se solape con las actividades ya seleccionadas.

3. Algoritmos voraces. Ejemplos de aplicación. **Selección de actividades.**

- Selección de actividades. **Demostración de corrección del criterio (1/2)**

- Demostración de que “*Seleccionar la actividad que termine más pronto*” (que no se solape con las actividades ya seleccionadas) nos lleva a una solución óptima.
 - Demostración por inducción sobre el número de actividades seleccionadas.
 - Consideremos las actividades numeradas de 1 a n .
 - Asumimos, además, que $f_1 \leq \dots \leq f_n$

- **Base de la inducción:** Sea $\text{cardinal}(S)=1$. Entonces $S = \{1\}$.
 - Si OPT es una solución optimal cualquiera y k es la “primera” actividad de OPT (o sea $f_k = \min \{f_j : j \in OPT\}$), pueden ocurrir dos casos: $k=1$ ó $k \neq 1$.
 - Si $k=1$, terminamos puesto que eso significaría que $S \subseteq OPT$ que es lo que queremos demostrar.
 - Si $k \neq 1$, obsérvese que $(OPT - \{k\}) \cup \{1\}$ es también una solución optimal ya que tiene el mismo numero de actividades que OPT , y además son actividades compatibles puesto que $f_1 \leq f_k$.

3. Algoritmos voraces. Ejemplos de aplicación. **Selección de actividades.**

- Selección de actividades. **Demostración de corrección del criterio (2/2)**
- **Hipótesis de inducción:** Sea $\text{cardinal}(S) = m - 1$ y sea S parte de una solución optimal OPT y construido según el criterio de selección indicado arriba.
 - Imaginemos OPT ordenado por tiempo de terminación de sus actividades.
 $OPT = \{a_1, \dots, a_{m-1}\} \cup \{b_m, \dots, b_n\}$
 - Por el criterio de selección es obvio que las actividades de S han de ser las primeras, es decir $S = \{a_1, \dots, a_{m-1}\}$.
 - Sea b_m la primera de $OPT - S$ (o sea, $f_{b_m} = \min\{f_j : j \in OPT - S\}$).
 - Sea i la actividad elegida por el criterio de selección tras haber seleccionado el conjunto S , formalmente $f_i = \min\{f_j : j \text{ es compatible con } S\}$.
 - $f_i \leq f_{b_m}$, puesto que todas las actividades de $OPT - S = \{b_m, \dots, b_n\}$ son compatibles con S
 - Entonces, $(OPT - \{b_m\}) \cup \{i\}$ es solución optimal pues tiene el mismo cardinal que OPT y la actividad i es compatible además con $(OPT - S) - \{b_m\}$.
 - La selección voraz m -ésima nos lleva a un conjunto de seleccionados incluido en una solución optimal.

3. Algoritmos voraces. Ejs de aplicación. **Selección de actividades: Algoritmo.**

□ **Algoritmo Selector_Actividades**

Los tiempos de comienzo y finalización respectivos de cada actividad $i = 1, \dots, n$ están en $c(1..n)$ y $f(1..n)$

```
función Selector_Actividades (c(1..n),f(1..n); var S(1..m))
// return conjunto_de_actividades (S(1..m),  $m \leq n$ )
// c(1..n) y f(1..n) ordenados según el criterio ( $i < j \Rightarrow f(i) \leq f(j)$ )
  S ← {1}
  z ← 1      // z representa a la última actividad seleccionada
  para i desde 2 hasta n hacer
    si c(i) ≥ f(z) entonces
      S ← S ∪ {i}
      z ← i
  fsi
  fpara
  return S
ffunción
```

Análisis: $\Theta(n)$

3. Algoritmos voraces. Ejs de aplicación. **Problema de la mochila.**

□ Problema de la mochila.

□ Tenemos:

- **n** objetos, cada uno con un peso (p_i) y un valor o beneficio (b_i)
- Una mochila en la que podemos meter objetos, con una **capacidad** de **peso máximo M**.

□ **Objetivo:** llenar la mochila con fragmentos de esos objetos, maximizando la suma de los beneficios (valores) transportados, y respetando la limitación de capacidad máxima **M**.

□ Se supondrá que los objetos **se pueden partir** en trozos ($0 \leq x_i \leq 1$).

➤ **Ejemplo:** $n = 3$; $M = 20$

$$p = (18, 15, 10)$$

$$b = (25, 24, 15)$$

- **Solución 1:** $S = (1, 2/15, 0)$
Beneficio total = $25 + 24 \cdot 2/15 = 28,2$
- **Solución 2:** $S = (0, 2/3, 1)$
Beneficio total = $15 + 24 \cdot 2/3 = 31$

3. Algoritmos voraces. Ejs de aplicación. **Problema de la mochila.**

☐ **Datos del problema:**

- **n**: número de objetos disponibles.
- **M**: capacidad de la mochila.
- **p** = (**p**₁, **p**₂, ..., **p**_n) pesos de los objetos.
- **b** = (**b**₁, **b**₂, ..., **b**_n) beneficios de los objetos.

☐ **Representación de la solución:**

- Una solución será de la forma **S** = (**x**₁, **x**₂, ..., **x**_n), con $0 \leq x_i \leq 1$, siendo cada **x**_i la fracción escogida del objeto **i**.

☐ **Formulación matemática:**

- **Maximizar** $\sum_{i=1}^n x_i b_i$ sujeto a la restricción $\sum_{i=1}^n x_i p_i \leq M$ y $0 \leq x_i \leq 1$

3. Algoritmos voraces. Ejs de aplicación. **Problema de la mochila.**

- ❑ El problema se ajusta bien a la idea de algoritmo voraz.
- ❑ **Diseño de la solución:**
 - **Candidatos:** Cada uno de los **n** objetos de partida.
 - Función **solución**: Tendremos una solución si hemos introducido en la mochila el peso máximo **M**, o si se han acabado los objetos.
 - Función **seleccionar**: Escoger el objeto más prometedor.
 - Función **factible**: Será siempre cierta (podemos añadir trozos de objetos).
 - **Añadir** a la solución: Añadir el objeto entero si cabe, o en otro caso la proporción del mismo que quede para completarla.
 - Función **objetivo**: Suma de los beneficios de cada candidato por la proporción seleccionada del mismo.
- ❑ Queda por definir la función de **selección**,
 - Qué **criterio** usar para seleccionar el **objeto más prometedor**

3. Algoritmos voraces. Ejs de aplicación.

Problema de la mochila. Algoritmo Mochila_Frag. (v.1.)

```
función Mochila_Frag (M: entero; b, p: [1..n]; var X: [1..n])
  para i:= 1, ..., n hacer
    X[i]:= 0                                /* la solución se construye en x */
  finpara
  pesoAct:= 0
  /* bucle voraz */
  mientras pesoAct < M hacer
    ➡ i:= el mejor objeto restante
    si pesoAct + p[i] ≤ M entonces
      X[i]:= 1
      pesoAct:= pesoAct + p[i]
    sino
      X[i]:= (M - pesoAct)/p[i]
      pesoAct:= M
    finsi
  finmientras
  devolver X
finfunción
```

3. Algoritmos voraces. Ejs de aplicación. **Problema de la mochila.**

- **Criterios de selección** para seleccionar el mejor objeto de los restantes:
 - Criterios incorrectos (no garantizan el óptimo):
 1. Seleccionar, del artículo de mayor beneficio (valor), el mayor fragmento que quepa en la mochila
 2. Seleccionar, del artículo de menor peso, el mayor fragmento que quepa en la mochila
 - Criterio de selección correcto:
 3. Seleccionar el mayor fragmento que quepa en la mochila del artículo que tenga el mayor cociente beneficio/peso (coste por unidad de peso)
- **Ejemplo 1:** $n = 4$; $M = 10$; $p = (10, 3, 3, 4)$; $b = (10, 9, 9, 9)$
 - Criterio 1: $S = (1, 0, 0, 0)$. Beneficio total = 10
 - Criterio 2 y 3: $S = (0, 1, 1, 1)$. Beneficio total = 27
- **Ejemplo 2:** $n = 4$; $M = 10$; $p = (10, 3, 3, 4)$; $b = (10, 1, 1, 1)$
 - Criterio 1 y 3: $S = (1, 0, 0, 0)$. Beneficio total = 10
 - Criterio 2: $S = (0, 1, 1, 1)$. Beneficio total = 3
- Los criterios 1 y 2 pueden dar soluciones no muy buenas.
- El criterio 3 garantiza siempre una solución óptima.
- Obsérvese que, entonces, una **solución óptimal** es de la forma siguiente
 $(1, \dots, 1, x_k, 0, \dots, 0)$ con $x_k \neq 1$, asumiendo que hemos ordenado los artículos de manera que
$$b_1 / p_1 \geq \dots \geq b_n / p_n$$
- No obstante, puede haber soluciones igualmente óptimas que no estén construidas así.

3. Algoritmos voraces. Ejs de aplicación. **Problema de la mochila.**

□ ¿Mejor objeto restante? → Ejemplar de problema: $n = 5$, $M = 100$

	1	2	3	4	5	
b_i	20	30	66	40	60	
p_i	10	20	30	40	50	$\sum_{i=1}^n x_i p_i > M$

□ **Selección:**

1. ¿Objeto más valioso? $\leftrightarrow b_i$ max
2. ¿Objeto más ligero? $\leftrightarrow p_i$ min
3. ¿Objeto más rentable? $\leftrightarrow b_i/p_i$ max

	1	2	3	4	5	
b_i	20	30	66	40	60	
p_i	10	20	30	40	50	
b_i/p_i	2,0	1,5	2,2	1,0	1,2	Objetivo $\sum_{i=1}^n x_i b_i$
x_i (b_i max)	0	0	1	0,5	1	146
x_i (p_i min)	1	1	1	1	0	156
x_i (b_i/p_i max)	1	1	1	0	0,8	164

3. Algoritmos voraces. Ejs de aplicación.

Problema de la mochila. Algoritmo Mochila_Frag (v.2.)

- Algoritmo Mochila_Frag. Una **solución consiste en ordenar los artículos según el criterio** comentado y poner, sucesivamente, todo lo que quepa de cada artículo, tomados precisamente en ese orden.

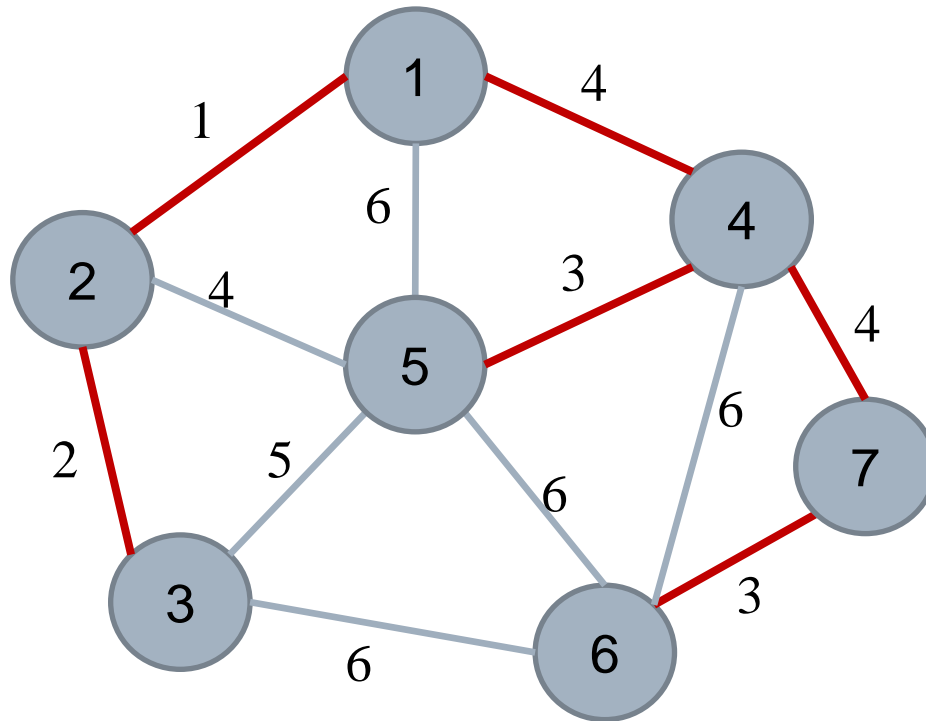
```
función Mochila_Frag (b (1..n), p(1..n), M) return X
  Ordenar_artículos_según_criterio() // por ej  $b_i/p_i$  max
  peso ← 0
  i ← 1 // representa el número del artículo a tratar
  mientras i ≤ n and peso + p(i) ≤ M hacer
    x(i) ← 1
    peso ← peso + p(i)
    i ← i + 1
  fmientras
  // si  $i < n \Rightarrow \text{peso} + p(i) > M$  y  $\forall k: 1..i - 1$  es  $x(k) = 1$ 
  si i ≤ n entonces
    x(i) ← (M - peso)/p(i)
    para k desde i+1 hasta n hacer
      x(k) ← 0
    fpara
  fsi
  return x (1..n)
ffunción
```

3. Algoritmos de grafos con estrategia voraz. Árboles de recubrimiento mínimo.

- ❑ **Árboles de recubrimiento de coste mínimo** (minimum spanning trees, MST, (a.r.m.))
- ❑ **Objetivo:** dado un grafo, obtener un nuevo grafo que sólo contenga las aristas imprescindibles para una optimización global de las conexiones entre todos los nodos
- ❑ **Aplicación:** problemas que tienen que ver con distribuciones geográficas
- ❑ **MSTs. Terminología:**
 - ❑ **árbol libre** (spanning tree): es un grafo no dirigido conexo acíclico
 - todo árbol libre con **n vértices** tiene **$n-1$ aristas**
 - si se añade una arista se introduce un **ciclo**
 - si se borra una arista quedan vértices no conectados
 - cualquier par de vértices está unido por un único camino simple
 - un árbol libre con un **vértice distinguido** es un **árbol con raíz**
 - ❑ **árbol de recubrimiento** de un grafo no dirigido y etiquetado no negativamente: es cualquier subgrafo que contenga todos los vértices y que sea un árbol libre
 - ❑ **árbol de recubrimiento de coste mínimo:** es un árbol de recubrimiento y no hay ningún otro árbol de recubrimiento cuya suma de aristas sea menor

3. Algoritmos de grafos con estrategia voraz. Árboles de recubrimiento mínimo.

□ Problema del **recubrimiento mínimo** en un grafo



□ **Problema:** Conectar todos los nodos con el menor coste total.

□ **Solución:** Algoritmos clásicos de Prim y Kruskal.

3. Algoritmos de grafos con estrategia voraz. Árboles de recubrimiento mínimo.

□ Problema del **recubrimiento mínimo** en un grafo

- Dado un grafo $G = (N; A)$ valorado, no dirigido y conexo, donde N es el conjunto de nodos y A el conjunto de aristas
- Cada arista tiene un peso (coste, longitud, distancia), una cantidad no negativa.
- El problema consiste en encontrar un conjunto de aristas que permita comunicar todos los nodos entre sí y tal que la suma de los pesos de estas aristas sea mínimo o, lo que es lo mismo, **su árbol de recubrimiento mínimo**.
- Un árbol de recubrimiento (spanning tree) de un grafo G es un subgrafo de G tal que:
 1. Contiene todos los vértices de G y un subconjunto de sus aristas
 2. Tiene estructura de árbol (es un grafo conexo sin ciclos)
- Este problema es isomorfo a muchos otros, bastante frecuentes. Por ejemplo: “Obtener el trazado de coste mínimo de una red de comunicaciones” (kilómetros de carretera o de fibra óptica).

3. Árboles de recubrimiento mínimo.

- Las **características** de este problema, de acuerdo con el método voraz, son:
 1. Los **candidatos** son las **aristas** del grafo G
 2. La **función solución** consiste en determinar si el conjunto de aristas seleccionadas es un árbol de recubrimiento de G
 3. Un conjunto de aristas es **factible** si no contiene ningún ciclo
 4. La **función de selección depende del algoritmo** concreto
 5. La **función objetivo** es la suma de los pesos de las aristas seleccionadas (esta suma debe ser minimizada)

3. Árboles de recubrimiento mínimo.

- El **esquema general** de un algoritmo voraz para obtener el **árbol de recubrimiento mínimo** de un grafo es el siguiente:

```
función MST ( G =(N,A) ) : árbol
    T = ∅
    mientras T no es solución hacer
        // T aún no determina un árbol de recubrimiento
        a = { seleccionar una arista según criterio de optimalidad local }
        si al añadir a a T no se crea un ciclo entonces
            T = T ∪ { a }
        fsi
        si T es un árbol de recubrimiento entonces
            devolver T
        fsi
    fmientras
ffunción
```

- La **función de selección** depende del algoritmo concreto
- Los algoritmos de **Kruskal** y **Prim** siguen este esquema general.

3. Árboles de recubrimiento mínimo. Algoritmo de Kruskal.

□ Algoritmo de Kruskal.

Partiendo del árbol vacío, se selecciona en cada paso la arista de menor etiqueta que no provoque ciclo sin requerir ninguna otra condición sobre sus extremos.

□ Esquema: $G = (N, A)$

1. Empezar con un grafo sin aristas: $T = (N, \emptyset)$
2. Seleccionar la arista de menor coste de A .
3. Si la arista seleccionada forma un **ciclo** en T , eliminarla. Si no, añadirla a T .
4. Repetir los dos pasos anteriores hasta tener **$n-1$** aristas.

3. Árboles de recubrimiento mínimo. Algoritmo de Kruskal.

□ **Implementación**, se necesita:

- Ordenar las aristas de **A**, de menor a mayor: $O(a \log a)$.
- Comprobar si una arista dada **(u, v)** provocará un ciclo.
 - Una arista **(u, v)** forma un ciclo si **u** y **v** están en el mismo componente conexo (conjunto).

□ **Funciones**:

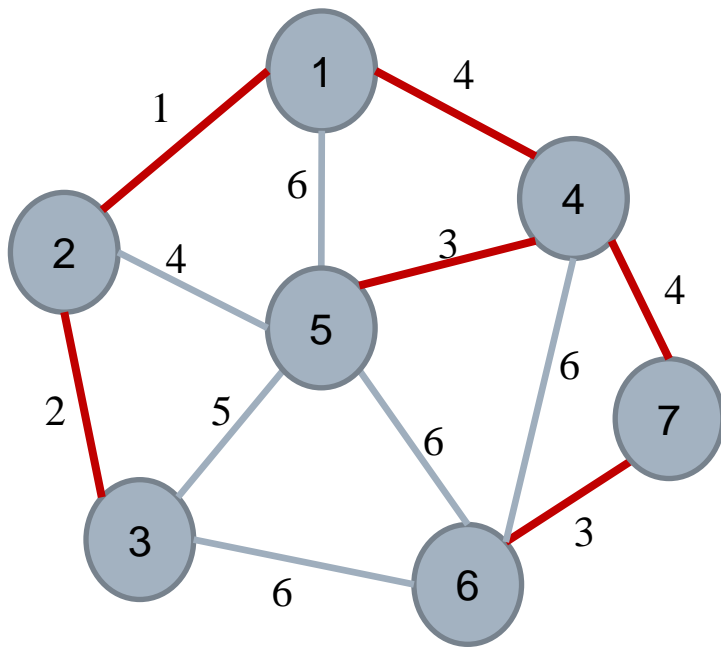
- **Inicialización**: crear una relación de equivalencia vacía (cada nodo es un componente conexo).
- **Seleccionar** las aristas **(u, v)** de menor a mayor.
- La arista forma **ciclo** si: **Buscar(u)=Buscar(v)**
- **Añadir** una arista **(u, v)**: **Fusionar(U, V)** (juntar dos componentes conexos en uno).

□ **Estructuras de datos**:

- “grafo”: aristas ordenadas
- componentes conexas: Conjuntos Disjuntos (buscar(x), fusionar(A,B)):

3. Árboles de recubrimiento mínimo. Algoritmo de Kruskal. Ejemplo.

arista	(1,2)	(2,3)	(4,5)	(6,7)	(1,4)	(2,5)	(4,7)	(3,5) ...
peso	1	2	3	3	4	4	4	5



paso	selección	componentes conexas(conjuntos)						
inicial	-	1	2	3	4	5	6	7
1	(1,2)	1,2		3	4	5	6	7
2	(2,3)	1,2,3			4	5	6	7
3	(4,5)	1,2,3			4,5		6	7
4	(6,7)	1,2,3			4,5		6,7	
5	(1,4)	1,2,3,4,5					6,7	
6	(2,5) (ciclo-> rechazada)	1,2,3,4,5					6,7	
7	(4,7)	1,2,3,4,5,6,7						

Solución $T = \{ (1,2), (2,3), (4,5), (6,7), (1,4), (4,7) \}$

3. Árboles de recubrimiento mínimo. Algoritmo de Kruskal.

```
función Kruskal ( G =(N,A) ) : árbol
  Ordenar A según longitudes crecientes;
  n := |N|;
  T := conjunto vacío;
  inicializar n conjuntos, cada uno con un nodo de N;
  /* bucle voraz */
  repetir
    a := (u,v) : arista más corta de A aún sin considerar;
    Conjunto U := Buscar (u);
    Conjunto V := Buscar (v);
    si Conjunto U <> Conjunto V entonces
      Fusionar (Conjunto U, Conjunto V);
      T := T U {a}
    fsi
  hasta |T| = n-1;
  devolver T
ffunción
```

3. Árboles de recubrimiento mínimo. Algoritmo de Kruskal. Análisis.

□ **Teorema:** Kruskal calcula el árbol expandido mínimo.

□ **Análisis:** $|N| = n$ y $|A| = m$

- ordenar A : $\Theta(m \log m) \equiv \Theta(m \log n) : n - 1 \leq m \leq n(n - 1)/2$
- inicializar n conjuntos disjuntos: $\Theta(n)$
- $2m$ buscar (peor caso) y $n - 1$ fusionar (siempre): $\Theta(m \log n)$
- resto: $\Theta(m)$ (peor caso)

$$\Rightarrow T(n) = \Theta(m \log n)$$

□ **Mejora:** utilizar un montículo de aristas en vez de ordenarlas

No cambia la complejidad del peor caso pero se obtienen mejores tiempos.

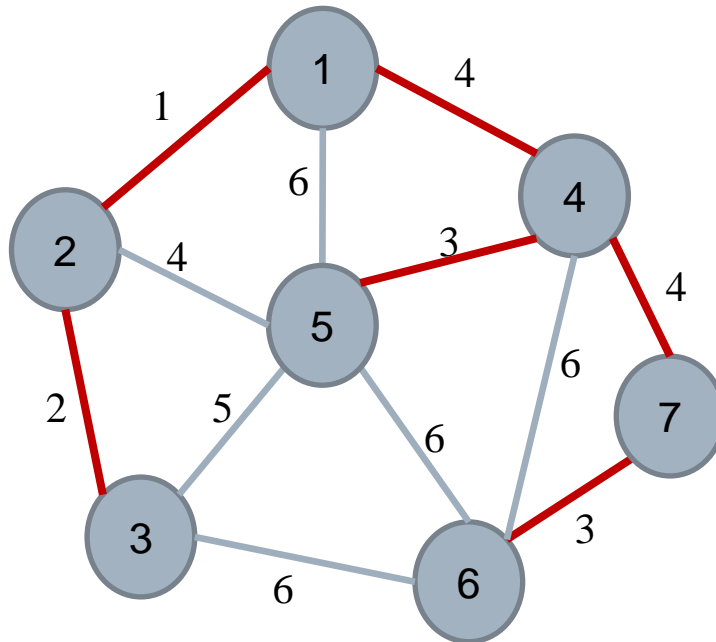
3. Árboles de recubrimiento mínimo. Algoritmo de Prim.

- Kruskal es un bosque que crece, en cambio Prim es un único árbol que va creciendo hasta alcanzar todos los nodos.
- **Esquema: $G = (N, A)$**
 1. **Inicialización:** $B = \{\text{nodo arbitrario}\} = \{1\}$, $T = \{\emptyset\}$
 2. **Selección:** **arista más corta** que parte de B :
de la arista seleccionada (u, v) , $u \in B$ y $v \in N - B$
se añade (u, v) a T y v a B
 1. **Repetir** los dos pasos anteriores hasta tener B los n nodos.
- Invariante: T define en todo momento un a.r.m. del subgrafo (B, A)
- Final: $B = N$ (Solución: T)
- **Algoritmo v.1:**

```
función Prim v.1 ( G =(N,A) ): árbol
    T := conjunto vacío;
    B := {un nodo de N};
    mientras B <> N hacer
        a := (u,v): arista más corta que parte de B
                (u pertenece a B y v no);
        T := T U {a};
        B := B U {v}
    fmientras
    devolver T
ffunción
```

3. Árboles de recubrimiento mínimo. Algoritmo de Prim. Ejemplo.

□ **Ejemplo:** (el mismo que para Kruskal):



L	1	2	3	4	5	6	7
1	∞	1	∞	4	6	∞	∞
2	1	∞	2	∞	4	∞	∞
3	∞	2	∞	∞	5	6	∞
4	4	∞	∞	∞	3	6	4
5	6	4	5	3	∞	6	∞
6	∞	∞	6	6	6	∞	3
7	∞	∞	∞	4	∞	3	∞

paso	selección	B
inicial	-	1
1	(1,2)	1,2
2	(2,3)	1,2,3
3	(1,4)	1,2,3,4
4	(4,5)	1,2,3,4,5
5	(4,7)	1,2,3,4,5,7
6	(7,6)	1,2,3,4,5,6,7=N

Solución $T = \{(1,2),(2,3),(1,4),(4,5),(4,7),(7,6)\}$

- Observación: No se producen rechazos.
- Teorema: Prim calcula el árbol expandido mínimo.
 - Demostración por inducción sobre $|T|$.

3. Árboles de recubrimiento mínimo. Algoritmo de Prim. Implementación.

□ Implementación:

■ Estructuras de datos:

- L: matriz de adyacencia $L[i, j] = \begin{cases} \text{distancia si } \exists (i, j) \\ \infty \text{ sino} \end{cases}$

⇒ matriz simétrica (desperdicio de memoria)

■ Funciones:

- Para cada nodo $i \in N - B$:

MásPróximo[i] : nodo $\in B$ más próximo

DistanciaMínima[i] : distancia de (i, MásPróximo [i])

- Para cada nodo $i \in B$:

DistanciaMínima [i] = -1

3. Árboles de recubrimiento mínimo. Algoritmo de Prim. Implementación. _Algoritmo v.2

```
función Prim v.2 ( L[1..n,1..n] ) : árbol
    DistanciaMínima[1] := -1;
    T := conjunto vacío;
    para i := 2 hasta n hacer
        MásPróximo [i] := 1;
        DistanciaMínima [i] := L[i,1]
    fpara
        /* bucle voraz */
        repetir n-1 veces:
            min := infinito;
            para j := 2 hasta n hacer
                si 0 <= DistanciaMínima [j] < min entonces
                    min := DistanciaMínima[j];
                    k := j
            fsi
            fpara
                T := T U {(MásPróximo[k], k)};           /* añadir arista a T */
                DistanciaMínima [k] := -1;                 /* añadir k a B */
                para j := 2 hasta n hacer
                    si L[j,k] < DistanciaMínima[j] entonces
                        DistanciaMínima [j] := L[j,k];
                        MásPróximo[j] := k
                fsi
            fpara
        finrepetir
    devolver T
finfunción
```

3. Algoritmos de grafos que utilizan una estrategia voraz

□ **Análisis:** $|N| = n$ y $|A| = m$

- Inicialización : $\Theta(n)$
- Bucle voraz: $n - 1$ iteraciones, y cada para anidado : $\Theta(n)$

$$\Rightarrow T(n) = \Theta(n^2)$$

□ **Comparación:**

	Prim $\Theta(n^2)$	Kruskal $\Theta(m \log n)$
Grafo denso(cuando tiene muchas aristas): $m \rightarrow n(n - 1)/2$	$\Theta(n^2)$	$\Theta(n^2 \log n)$
Grafo disperso(muy pocas aristas): $m \rightarrow n$	$\Theta(n^2)$	$\Theta(n \log n)$

- Tabla: Complejidad temporal de los algoritmos de Prim y Kruskal.
- (densidad del grafo= $d=2m/n(n-1)$)

3. Algoritmos de grafos que utilizan una estrategia voraz. Problema del camino mínimo. Algoritmo de Dijkstra.

□ **Problema del camino mínimo.** El algoritmo de **Dijkstra**.

Dado un grafo $G = (N, A)$ dirigido, con longitudes en las aristas ≥ 0 , con un nodo distinguido como origen de los caminos (el nodo 1), encontrar los caminos mínimos entre el nodo origen y los demás nodos de N .

□ **Técnica voraz:**

■ 2 conjuntos de nodos:

- S \equiv seleccionados: camino mínimo establecido
- C \equiv candidatos: los demás

■ invariante: $N = S \cup C$

1. **inicialmente**, $S = \{1\}$ al final S contiene todos los nodos del grafo
 \Rightarrow **final**: $S = N$: función solución

2. **función selección**: nodo de C con **menor distancia** conocida desde 1
 \rightarrow existe una información **provisional** sobre distancias mínimas

□ **Definición**: Un **camino** desde el origen a un nodo v es **especial** \Leftrightarrow todos sus nodos intermedios están en S .

\Rightarrow **D**: **vector con longitudes de caminos especiales mínimos** ;

Selección de $v \Leftrightarrow$ el camino especial mínimo 1.. v es también camino mínimo.

3. Al final, **D** contiene la solución, las longitudes de los caminos mínimos.

3. Problema del camino mínimo. Algoritmo de Dijkstra. Implementación.

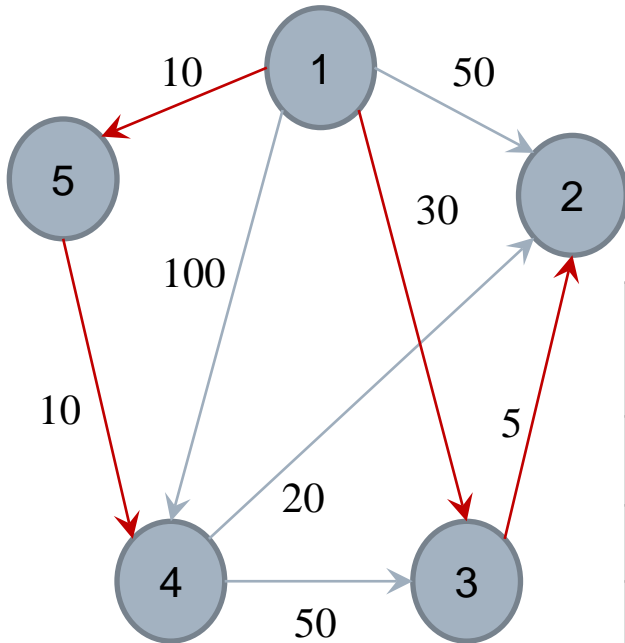
□ Implementación:

- $N = 1, 2, \dots, n$ (1 es el origen)
- Matriz L da la longitud de todas las aristas dirigidas, *Matriz de adyacencia, no simétrica*.

$$L \begin{cases} L[i, j] \geq 0 & \text{si } (i, j) \in A \\ L[i, j] = \infty & \text{en otro caso} \end{cases}$$

```
función Dijkstra ( L[1..n,1..n] ) : vector [2..n]
    vector D[2..n];
    vector S[1..n];
    /* inicialización */
    C := {2, 3, ..., n};
    S := {1};
    para i := 2 hasta n hacer D[i] := L[1,i]   fpara
    /* bucle voraz */
    repetir n-2 veces:
        v := nodo de C que minimiza D[v];
        C := C-{v}; S := S ∪ {v};
        para cada w en C hacer
            D[w] := min(D[w], D[v]+L[v,w]) /*si D[w]> D[v]+L[v,w] ⇒ D[w]:=D[v]+L[v,w] */
        fpara
    frepetir
    devolver D
ffunción
```

3. Algoritmo de Dijkstra. Ejemplo.



L	1	2	3	4	5
1	∞	50	30	100	10
2	∞	∞	∞	∞	∞
3	∞	5	∞	∞	∞
4	∞	20	50	∞	∞
5	∞	∞	∞	10	∞

paso	V	C	D (vector distancias)			
			2	3	4	5
inicial	-	{2,3,4,5}	50	30	100	10
1	5	{2,3,4}	50	30	20	10
2	4	{2,3}	40	30	20	10
3	3	{2}	35	30	20	10

Tabla: Evolución del conjunto C y de los caminos mínimos.

➤ **Observación:** 3 iteraciones = $n - 2$

Solución $D = \{35, 30, 20, 10\} \equiv$ (distancia desde el nodo 1 a cada nodo del grafo)

3. Problema del camino mínimo. Algoritmo de Dijkstra. **Análisis(I).**

□ **Teorema:** Dijkstra encuentra los caminos mínimos desde el origen hacia los demás nodos del grafo. Demostración por inducción.

□ **Análisis:** $|N| = n$, $|A| = m$, $L[1..n, 1..n]$

- Inicialización : $O(n)$

- ¿Selección de v ?

 - “*implementación rápida*”: recorrido sobre C

 - ≡ examinar $n - 1, n - 2, \dots, 2$ valores en D

- $\Sigma = \Theta(n^2)$

- Para anidado: $n - 2, n - 3, \dots, 1$ iteraciones

- $\Sigma = \Theta(n^2)$

$$\Rightarrow T(n) = \Theta(n^2)$$

□ **Mejora:** si el grafo es disperso ($m \ll n^2$), utilizar listas de adyacencia

→ ahorro en para anidado: recorrer lista y no fila o columna de L

3. Problema del camino mínimo. Algoritmo de Dijkstra. **Análisis(II).**

□ **Análisis** (Cont.):

□ ¿Cómo evitar $\Omega(n^2)$ en selección?

→ C: montículo min, ordenado según D[i]

⇒ inicialización en $\Theta(n)$

C := C - v en $O(\log n)$

Para anidado: modificar D[w] = $O(\log n)$ ≡ flotar

¿Nº de veces? Máximo 1 vez por arista (peor caso)

□ En total:

■ extraer la raíz n - 2 veces (siempre)

■ modificar un máximo de m veces un valor de D (peor caso)

⇒ $T(n) = \Theta((m+n)\log n)$

□ **Observación:** La “*implementación rápida*” es preferible si el grafo es denso

3. Algoritmos voraces. Conclusiones.

- Avance rápido se basa en una idea intuitiva:
 - Empezamos con una solución “vacía”, y la construimos paso a paso.
 - En cada paso se selecciona un candidato (el más prometedor) y se decide si se mete o no (función factible).
 - Una vez tomada una decisión no se vuelve a deshacer.
 - Acabamos cuando tenemos una solución o cuando no queden candidatos.
- **Primera cuestión:** Cuáles son los **candidatos** y cómo se **representa** una **solución** al problema.
- **Cuestión clave:** diseñar una función de selección adecuada.
 - Algunas pueden garantizar la solución óptima.
 - Otras pueden ser más heurísticas.
- **Función factible:** garantizar las restricciones del problema.
- En general los algoritmos voraces son la solución rápida a muchos problemas (a veces óptimas, otras no).
- Si podemos deshacer decisiones: **BACKTRACKING**