Departamento de Informática
Universidad de Valladolid
Campus de Segovia

TEMA 5: COMPLEJIDAD ALGORÍTMICA

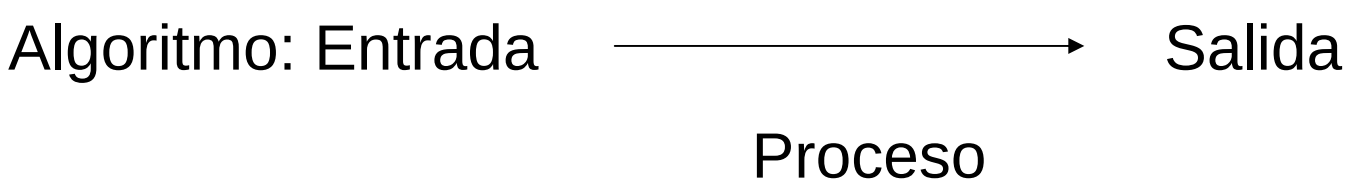
COMPLEJIDAD ALGORÍTMICA

- Conceptos básicos.
- Medidas de comportamiento asintótico.
- Reglas prácticas para hallar el coste
- Útiles matemáticos
- Complejidad de algoritmos de búsqueda y ordenación

DEFINICIÓN DE ALGORITMO

- Un algoritmo implica la descripción precisa de los pasos a seguir para alcanzar la solución de un problema dado.

- Por pasos se entiende el conjunto de acciones u operaciones que se efectúan sobre ciertos objetos.



CARACTERÍSTICAS DE UN ALGORITMO

- Un algoritmo debe poseer las siguientes características:
  - Precisión: Un algoritmo debe expresarse sin ambigüedad.
  - Determinismo: Todo algoritmo debe responder del mismo modo antes las mismas condiciones.
  - Finito: La descripción de un algoritmo debe ser finita.

CUALIDADES DE UN ALGORITMO

- Un algoritmo debe ser además:
  - **General**: Es deseable que un algoritmo sea capaz de resolver una clase de problemas lo más amplia posible.
  - **Eficiente**: Un algoritmo es eficiente cuantos menos recursos en tiempo, espacio (de memoria) y procesadores consume.
- Por lo general es difícil encontrar un algoritmo que reúna ambas por lo que se debe alcanzar un compromiso que satisfaga lo mejor posible los requisitos del problema.

COMPLEJIDAD ALGORITMICA.

- La complejidad algorítmica representa la cantidad de **recursos** (temporales) que necesita un algoritmo para resolver un problema y por tanto permite determinar la **eficiencia** de dicho algoritmo.

- Los criterios que se van a emplear para evaluar la complejidad algorítmica **no proporcionan medidas absolutas** sino medidas relativas al tamaño del problema.

EL TIEMPO EMPLEADO POR EL ALGORITMO SE MIDE EN PASOS

- La medida del tiempo tiene que ser independiente:
  - de la máquina
  - del lenguaje de programación
  - del compilador
  - de cualquier otro elemento hardware o software que influya en el análisis.

- Para conseguir esta independencia una posible medida abstracta puede consistir en determinar cuantos pasos se efectúan al ejecutarse el algoritmo.

COMPLEJIDAD ALGORITMICA. CONCEPTOS BÁSICOS

- El tiempo empleado por el algoritmo se mide en pasos.
- El coste depende del tamaño de los datos.
- A la hora de evaluar el coste se debe de tener en consideración tres posibles casos:
  - El coste esperado o promedio
  - El coste mejor
  - El coste peor
- Si el tamaño de los datos es grande lo que importa es el comportamiento asintótico de la eficiencia.

EL COSTE EN TIEMPO DEPENDE DEL TAMAÑO DE LOS DATOS

- El tiempo requerido por una algoritmo es función del tamaño de los datos.
- Por esta razón la complejidad temporal se expresa de la siguiente forma:

$$T(n)$$

- Dependiendo del problema, el tamaño del dato representa cosas diferentes:
  - el número en sí
  - el número de dígitos o elementos que lo compone.
- Otra característica importante es que no todos los datos, dentro de un problema, poseen la misma importancia de cara a la complejidad algorítmica.

EL COSTE EN TIEMPO DEPENDE DEL TAMAÑO DE LOS DATOS

- Ejemplo 1: Algoritmo que determina la paridad de un número restando 2 sucesivamente mientras el resultado sea mayor que 1 para finalmente comprobar el resultado.
  - El problema tendrá n DIV 2 restas (depende de n).

- Ejemplo 2: Algoritmo de suma lenta

```
while b>0 do begin
    a:=a+1;
    b:=b-1;
end;
```

  - En este caso T=T(b).

EL COSTE ESPERADO, EL MEJOR Y EL PEOR

- Otra característica es que la complejidad algorítmica no sólo depende del tamaño sino del propio dato en sí.

EL COSTE ESPERADO, EL MEJOR Y EL PEOR

```
type
tintervalo=0..N;
tvector=array[1..N] of integer
FUNCTION Busquedasecord(v:tvector;elem:telem):tintervalo
var
    i:tintervalo;
begin
    i:=0;
    repeat
        i:=i+1;
    until (v[i]>=elem) or (i=N);
    if v[i]=elem then
        Busquedasecord:=i
    else
        Busquedasecord:=0
End;
```

En este algoritmo se pueda dar las siguientes situaciones:
- **Caso mejor**: el elemento este en la primera posición.
- **Caso peor**: Se tenga que recorrer todo el vector.
- **Caso promedio o esperado**: Puesto que todas la posiciones son equiprobables el tiempo será n/2 pasos.

EL COSTE ESPERADO, EL MEJOR Y EL PEOR. NOTACIÓN

- $T_{max}(n)$: Representa la complejidad temporal en el peor de los casos.
- $T_{min}(n)$: Representa la complejidad en el mejor de los casos posibles.
- $T_{med}(n)$: Expresa la complejidad temporal en el caso promedio. Para su cálculo se suponen que todas las entradas son equiprobables.

EL COSTE ESPERADO, EL MEJOR Y EL PEOR. EJEMPLO

- Cálculo de T_{max}, T_{min}, y T_{med} para el algoritmo de búsqueda secuencial ordenada:

- Nomenclatura del tiempo constante empleado por las siguientes operaciones:
  - suma:‘s’
  - comparación: ‘c’
  - asignación: ‘a’

EL COSTE ESPERADO, EL MEJOR Y EL PEOR. EJEMPLO

T_{min}: Este tiempo se calcula cuando v[1]>=elem.

$$T_{min}=3a+3c+s=constante$$

T_{max}:Este tiempo se calcula cuando v[n]<=elem

$$T_{max}=a+n(s+2c+a)+c+a=n(s+2c+a)+2a+c$$

$$T_{max}=K_1 n+K_2$$

EL COSTE ESPERADO, EL MEJOR Y EL PEOR. EJEMPLO

T_{med}:Este tiempo se calcula considerando cualquier entrada equiprobable. Si $T(j)=jK_1+K_2$

Entonces:

$$T_{med}(n) = \sum_{j=1}^{n} T(j)\,P \qquad donde \qquad P = 1/n$$

$$T_{med}(n) = \sum_{j=1}^{n} (jk_1 + k_2)\frac{1}{n} =$$

$$= \sum_{j=1}^{n} \frac{k_1}{n} j + \sum_{j=1}^{n} \frac{k_2}{n} = \frac{k_1}{n}\frac{(n+1)n}{2} + k_2$$

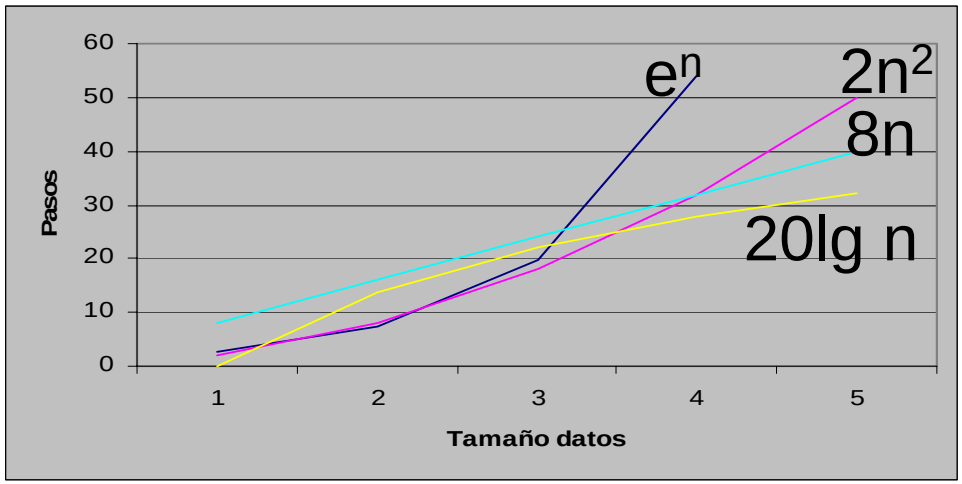
$$T_{med}(n) = \frac{k_1(n+1)}{2} + k_2 = \frac{k_1 n}{2} + \frac{k_1}{2} + k_2 = c_1 n + c_2$$

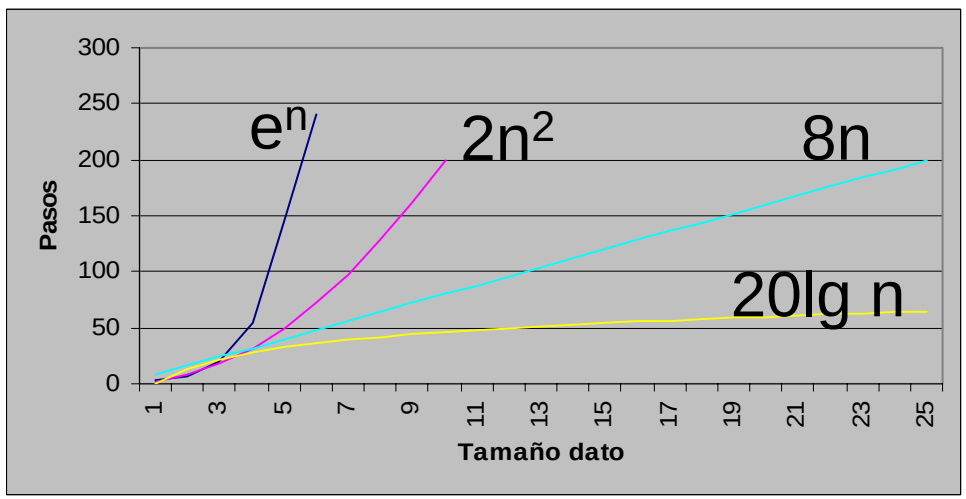
LO IMPORTANTE ES EL COMPORTAMIENTO ASINTÓTICO

- Tiempos empleados para el cálculo de algoritmos con distintos ordenes, considerando que el computador en cuestión ejecuta 1 Millón de operaciones por segundo (1MHz).

n \ T(n)	log n	n	n log n	n^2	n^3	2^n	n!
10	$3.3\ 10^{-6}$	10^{-5}	$3.3\ 10^{-5}$	10^{-4}	0.001	0.001	3.63
50	$5.6\ 10^{-6}$	$5\ 10^{-5}$	$2.8\ 10^{-4}$	0.0025	0.125	intratable	intratable
100	$6.6\ 10^{-6}$	10^{-4}	$6.6\ 10^{-4}$	0.01	1	intratable	intratable
10^3	10^{-5}	0.001	0.01	1	1000	intratable	intratable
10^4	$1.3\ 10^{-5}$	0.01	0.13	100	10^6	intratable	intratable
10^5	$1.6\ 10^{-5}$	0.1	1.6	10^4	intratable	intratable	intratable
10^6	$2\ 10^{-5}$	1	19.9	10^6	intratable	intratable	intratable

MEDIDAS DEL COMPORTAMIENTO ASINTÓTICO EJEMPLO





MEDIDAS DEL COMPORTAMIENTO ASINTÓTICO

- El orden de la función T(n) expresa el comportamiento dominante para los datos de gran tamaño.

- Para poder determinar y expresar correctamente el comportamiento asintótico es conveniente disponer de una adecuada notación. A continuación se presentan las notaciones:
  - O mayúscula
  - Ω Mayúscula
  - Θ mayúscula

NOTACIÓN ‘O’ MAYÚSCULA

- Definición: Sean $f,g:Z^+ \longrightarrow R^+$, se dice que $f \in O(g)$ o que f es del orden de g si existen constantes $n_o \in Z^+$ y $\lambda \in R^+$ tales que:

$$f(n) \leq \lambda\, g(n) \quad \text{para todo } n \geq n_o$$

- Esto significa que f no crece más deprisa que g. De esta forma acotamos superiormente el comportamiento asintótico de la función salvo constantes.
- Para el algoritmo de Búsqueda secuencial ordenada:

$T_{max}(n)=k_1 n+k_2 \in O(n)$ ya que

$k_1 n+k_2 \leq \lambda n$ para todo $n \geq k_2/(\lambda-k_1)$

- Todas las funciones de tiempo constante son O(1).

PROPIEDADES DE LAS NOTACIÓN. ESCALABILIDAD

- $O(\log_a n)=O(\log_b n)$

- Por esta razón no es necesario especificar la base del logaritmo: O(log n).

PROPIEDADES DE LAS NOTACIÓN. REGLA DE LA SUMA

- **Regla de la suma**: Si $f_1 \in O(g_1)$ y $f_2 \in O(g_2)$ entonces $f_1+f_2 \in O(max(g_1,g_2))$.

- La generalización de esta regla junto con la propiedad de la escalabilidad se expresa de la siguiente forma:

  - Si $f_i \in O(f)$ para todo i=1....k entonces:
    $c_1f_1+.....+c_kf_k \in O(f)$.
  - De donde cualquier polinomio $p_k(n) \in O(n^k)$

PROPIEDADES DE LAS NOTACIÓN. REGLA DEL SUMATORIO

- **Regla del sumatorio**: Si $f \in O(g)$ y la función g es creciente entonces:

$$\sum_{i=1}^{n} f(i) \in O\left(\int_{1}^{n+1} g(x)\,dx \right)$$

  - Si f(i)=i. $$\sum_{i=1}^{n} i = \frac{(n+1)n}{2} = \frac{n^2}{2} + \frac{n}{2}$$

$$\int_{1}^{n+1} x\,dx = \frac{x^2}{2}\Big|_{1}^{n+1} = \frac{(n+1)^2}{2} - \frac{1}{2} = \frac{n^2}{2} + n$$

  - De donde cualquier polinomio $p_k(n) \in O(n^k)$

CONSECUENCIA ÚTIL DE LA REGLA DEL SUMATORIO

$$\sum_{i=1}^{n} i^{K} \in O\left(n^{k+1}\right)$$

$$\int_{1}^{n+1} x^{k}\,dx = \left.\frac{x^{k+1}}{k+1}\right|_{1}^{n+1} = \frac{(n+1)^{k+1}}{k+1} - \frac{1}{k+1} \approx$$

$$k_1 (n+1)^{k+1} + k_2 \in O\left(n^{k+1}\right)$$

JERARQUÍA DE ÓRDENES DE FRECUENTE APARICIÓN

- Los comportamientos asintóticos de más frecuente aparición se pueden ordenar de menor a mayor crecimiento de la siguiente forma:

$$\mathbf{1 << \log n << n << n \log n << n^2 << n^3 << \ldots << 2^n << n!}$$

REGLAS PRÁCTICAS PARA HALLAR EL COSTE DE UN ALGORITMO

- Lo que se presenta a continuación son reglas generales para el cálculo de la complejidad temporal en el peor de los casos.

- Estas reglas deberán tener en consideración los costes de:
  - Las instrucciones simples
  - La composición de instrucciones
  - Las instrucciones de selección
  - De los bucles
  - Los subprogramas

INSTRUCCIONES SIMPLES

- Se considera que se ejecuta en tiempo constante:
  - La evaluación de las expresiones aritméticas siempre que los datos sean de tamaño constante así como las comparaciones de datos simples.
  - Las operaciones de asignación, lectura y escritura de datos simples.
  - Las operaciones de acceso a una componente de un array, a un campo de un registro y a la siguiente posición de un registro de un archivo.
- Todas estas operaciones se consideran de $\Theta(1)$.

COMPOSICIÓN DE INSTRUCCIONES

- Si suponemos que las instrucciones I_1 e I_2 poseen complejidades temporales, en el peor de los casos de $T_1(n)$ y $T_2(n)$ respectivamente entonces el coste de la composición de ambas instrucciones será:

$$T_{I1,I2}(n)=T_1(n)+T_2(n)$$

- Que aplicando la regla de la suma es el máximo de ambos.

INSTRUCCIONES DE SELECCIÓN

- Para la instrucción condicional:
  - if <condición> then I_1 else I_2

$$T_{selección}(n)=T_{condición}(n)+max(T_1(n),T_2(n))$$

  - Case <expresión> of
    caso1: I_1;
    caso2: I_2;
    .....
    .....
    cason: I_n;
    end; {case}

$$T_{selección}(n)=T_{expresión}(n)+max(T_1(n),....,T_n(n))$$

INSTRUCCIONES DE ITERACIÓN: BUCLES

- El caso más sencillo es el bucle for:
  - for i:=1 to m do I

$$T_{bucle}(n) = m + \sum_{j=1}^{m} T_{I_j}(n)$$

  - Donde m representa las ‘m’ veces que se incrementa i y la comprobación de si está entre los extremos inferior y superior.

  - Si el campo interior consume un tiempo constante entonces la ecuación anterior se escribe de la siguiente forma:

$$T_{bucle}(n) = m\,(1 + T_I(n))$$

- Los bucles while y repeat no poseen una norma general

SUBPROGRAMAS

- El coste de un subprograma se deriva de las reglas anteriores. Si el subprograma es recursivo entonces hay que distinguir entre los casos base y recurrentes.

SUBPROGRAMAS

```
FUNCTION Fac(n:integer):integer;
Begin
   if n=0 then
                Fac:=1
   else
                Fac:=n*Fac(n-1)
End;{Fac}
```

- El caso base posee un coste constante: $T_{fac}(0)=1$
- Para los casos recurrentes:

  $T_{fac}(n)=1+T_{fac}(n-1)=$

  $T_{fac}(n)=1+1+T_{fac}(n-2)=$

  .....

  $T_{fac}(n)=1+1+....+1+T_{fac}(0) = n+1 \in O(n)$

ÚTILES MATEMÁTICOS

- Para el cálculo de la complejidad es útil saber como calcular términos generales de sucesiones en las que los términos se definen en función de los términos anteriores.

  - Fórmulas con sumatorios
  - Sucesiones de recurrencia lineales de primer orden

FORMULAS CON SUMATORIOS

– *Sucesión aritmética* $x_n = x_{n-1} + r$

$$\sum_{i=1}^{n} x_n = \frac{(x_1 + x_n)n}{2}$$

$$1 + x + x^2 + \ldots + x^{n-1} = \frac{1 - x^n}{1 - x}$$

$$\sum_{i=1}^{\infty} x^i = 1 + x + x^2 + \ldots = \frac{1}{1 - x} \quad si \quad |x| < 1$$

$$\sum_{i=1}^{\infty} \frac{x^i}{i!} = e^x$$

$$\sum_{i=1}^{\infty} (-1)^i \frac{x^i}{i} = \log x$$

SUCESIONES DE RECURRENCIA LINEALES DE PRIMER ORDEN

$$x_n = c\, x_{n-1} = c^n\, x_0$$

$$x_n = b_n\, x_{n-1} \quad para \quad un \quad n \geq 1$$

$$x_n = b_1\, b_2\, ...\, b_n\, x_0$$

$$x_n = b_n\, x_{n-1} + c_n$$

realizando el cambio :

$$x_n = b_1\, b_2\, ...\, b_n\, y_n$$

en la recurrencia de x_{n+1}

$$x_{n+1} = b_{n+1}\, x_n + c_{n+1}$$

$$x_n = b^n \left(x_0 + c\, \frac{\left(1 - b^n\right)}{(1-b)b^n} \right)$$

ALGORITMOS DE BÚSQUEDA Y ORDENACIÓN

- Algoritmos de búsqueda en arrays
  - Secuencial
  - Secuencial ordenada
  - Binaria
- Ordenación de vectores
  - Selección directa
  - Inserción directa
  - Intercambio directo
  - Ordenación rápida (Quick Sort)
  - Ordenación por mezcla (Merge Sort)

ALGORITMOS DE BÚSQUEDA EN ARRAYS

- Surgen de la necesidad de conocer tanto si un dato se encuentra o no dentro de una colección como de la posición que ocupa.

- Búsqueda(vector,elemento):
  - $i \in \{1,....,n\}$ si existe tal elemento
  - 0 en otro caso

- Estructura de datos:

```
const
    N=100;
type
    tIntervalo=0..N;
    tvector=array[1..N] of tElem {tipo ordinal}
```

ALGORITMOS DE BÚSQUEDA EN ARRAYS

- Algoritmos de búsqueda:
  - Búsqueda secuencial
  - Búsqueda secuencial ordenada
  - Búsqueda binaria

BÚSQUEDA SECUENCIAL

- La búsqueda secuencial consiste en comparar secuencialmente el elemento deseado con los valores contenidos en las posiciones 1,....,n.

- El proceso termina cuando o bien encontramos el elemento o bien se alcanza el final del vector.

- Un primer nivel de diseño:

```
ind:=0
buscar elemento en vector
si vector[ind]=elemento entonces
    busquedasecuencial:=ind
sino
    busquedasecuencial:=0
```

BÚSQUEDA SECUENCIAL,
IMPLEMENTACIÓN EN PASCAL

```
FUNCTION Busquedasec(v:tvector ; elem:telem):tIntervalo;
{Dev. 0 si el elemento no está en ‘v’ o i si v[ì]=elem}

VAR
    i:tIntervalo;
BEGIN
    i:=0;
    repeat
        i:=i+1;
    until (v[i]=elem) or (i=N);
    if v[i]=elem then
                busquedasec:=i
        else
                busquedasec:=0
END; {busquedasec}
```

El número de veces que se ejecuta
El algoritmo dependerá del bucle.
En el peor de los casos será de orden O(n).

BÚSQUEDA SECUENCIAL ORDENADA

- El algoritmo anterior puede ser mejorado si el vector ‘v’ esta ordenado (i.e. Creciente).

- De esta forma si durante la búsqueda se alcanza una componente con mayor valor que ‘elem’, podremos asegurar que no se encuentra dentro de la colección.

BÚSQUEDA SECUENCIAL ORDENADA,
IMPLEMENTACIÓN EN PASCAL

```
FUNCTION Busquedasecord(v:tvector ; elem:telem):tIntervalo;
{Dev. 0 si el elemento no está en ‘v’ o i si v[ì]=elem}

VAR
    i:tIntervalo;
BEGIN
    i:=0;
    repeat
        i:=i+1;
    until (v[i]≥elem) or (i=N);
    if v[i]=elem then
                busquedasecord:=i
        else
                busquedasecord:=0
END; {Busquedasecord}
```

*Este algoritmo en el peor de los casos es de orden O(n).

BÚSQUEDA BINARIA

- El hecho de que el vector este ordenado se puede, también, aprovechar para conseguir una mayor eficiencia planteando el siguiente algoritmo.
  - Comparar ‘elem’ con el elemento central del vector. Si este es el elemento buscado se finaliza. Si no se sigue buscando en la mitad del vector que determine la relación entre el valor del elemento central y el buscado.
  - Este algoritmo finaliza cuando se localiza el elemento o se termina el vector.
- Debido a que el vector es dividido sucesivamente en dos se denomina búsqueda binaria.

BÚSQUEDA BINARIA,
IMPLEMENTACIÓN EN PASCAL

```
FUNCTION Busquedabinaria(v:tvector ; elem:telem):tIntervalo;
{Prec. V esta ordenado crecientemente}
{Dev. 0 si el elemento no está en 'v' o i si v[ì]=elem}
VAR
    einf,esup,posmed:tIntervalo;
    encontrado:boolean;
BEGIN
    einf:=1; esup:=N; encontrado:=false;
    while not encontrado and (esup≥einf) do begin
            posmed:=(esup+einf) DIV 2;
            if  elem=v[posmed] then
                        encontrado:=true
                else if elem>v[posmed] then
                                    einf:=postmed+1
                        else
                                    esup:=postmed-1
    end {while}
```

BÚSQUEDA BINARIA,
IMPLEMENTACIÓN EN PASCAL

```
    if encontrado then
            busquedabinaria:=posmed
      else
            busquedabinaria:=0
END; {Busquedabinaria}
```

BÚSQUEDA BINARIA

- La complejidad algorítmica de la búsqueda binaria es:

  Teniendo en cuenta que:

  $$2^k \leq N \leq 2^{k+1}$$

  entonces en el peor de los casos:

  $$T(n) \approx O[\log N]$$

  $2^k \leq N$ de donde

  $\log 2^k \leq \log N$

  $k \leq \log N$

ORDENACIÓN DE VECTORES

- Como se ha presentado con anterioridad, disponer de una colección de datos ordenados facilita la operación de búsqueda. A continuación se presentan algunos de los algoritmos de ordenación más frecuentes:

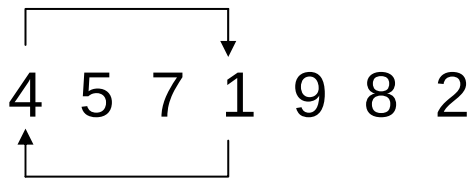
  - Algoritmos cuadráticos:
    - Selección directa
    - Inserción directa
    - Intercambio directo
  - Algoritmos avanzados de búsqueda:
    - Algoritmo rápido (Quick Sort)
    - Algoritmo de mezcla (Merge Sort)

SELECCIÓN DIRECTA

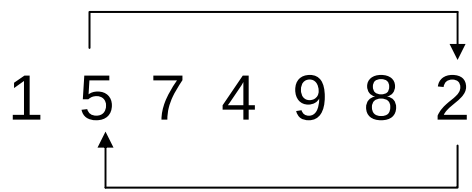
- Procedimiento: Recorre el vector y selecciona en cada uno de los recorridos el menor elemento para situarlo en su lugar correspondiente.

SELECCIÓN DIRECTA

- 1. Se sitúa en v[1] el menor elemento de entre v[1],...,v[n]. Para ello se intercambian los valores de v[1] y v[i] {v[i]=min(v)}



- 2. Se sitúa en v[2] el menor elemento de entre v[2],...,v[n]. Para ello se intercambian los valores de v[2] y v[i] {v[i]=min(v)}



- (n-1). Se sitúa en v[n-1] el menor elemento de entre v[n-1] y v[n]. Para ello se intercambian los valores de v[n-1] y v[n].

1 2 4 5 7 8 9

SELECCIÓN DIRECTA,
IMPLEMENTACIÓN EN PASCAL

```
PROCEDURE  Selecciondirecta( var v:tvector);
{Efecto. Se ordena 'v' ascendentemente}
VAR
    i, j,posmenor:tIntervalo;
    valmenor,aux:telem;
BEGIN
    for i:=1 to N-1 do begin
          valmenor:=v[i]
          posmenor:=i
          for j:=i+1 to N do
                if  v[j]<valmenor then begin
                      valmenor:=v[j];
                      posmenor:=j
                end; {if }
```

Busca el menor elemento de entre i+1,....,N

SELECCIÓN DIRECTA,
IMPLEMENTACIÓN EN PASCAL

```
        if posmenor<>i then begin
                aux:=v[i];
                v[i]:=v[posmenor];
                v[posmenor]:=aux
         end {if}
    end{for i}
END; {Selecciondirecta}
```


Intercambio si posmenor es diferente de i

INSERCIÓN DIRECTA

- Procedimiento: Recorre el vector ‘v’ insertando el elemento v[i] en su lugar correcto entre los ya ordenados.

INSERCIÓN DIRECTA

- 1. Se considera v[1] como primer elemento.
- 2. Se inserta v[2] en su posición correspondiente en relación a v[1] y v[2].
- 3. Se inserta v[3] en su posición correspondiente en relación a v[1], v[2] y v[3].
- i. Se inserta v[i] en su posición correspondiente en relación con v[1],...,v[i].
- n. Se repite la operación con el último elemento del vector.

INSERCIÓN DIRECTA,
IMPLEMENTACIÓN EN PASCAL

```
PROCEDURE  Inserciondirecta( var v:tvector);
{Efecto. Se ordena 'v' ascendentemente}
VAR
    i, j:tIntervalo;
    aux:telem;
BEGIN
    for i:=2 to N do begin
        aux:=v[i]
        j:=i-1
        while (j≥1) and (v[j]>aux) do begin
            v[j+1]:=v[j];
            j:=j-1
        end; {while}
        v[j+1]:=aux
    end {for}
END; {inserciondirecta}
```

Desplazamiento del menor valor

Colocación del valor en su lugar correspondiente

INTERCAMBIO DIRECTO

- Procedimiento: Recorre el vector ‘v’ buscando el menor elemento desde la última posición hasta la actual y lo inserta en dicha posición.

INTERCAMBIO DIRECTO

- 1. Situar el elemento menor en la primera posición.
  - Para ello se compara el último elemento con el penúltimo intercambiando los valores si están en orden decreciente. Se repite esta operación hasta llegar a la primera posición. (el elemento menor se encuentra en la primera posición).

- 2. Situar el segundo menor elemento en la segunda posición.
  - Para ello se procede como antes finalizando al alcanzar al segunda posición.

- 3. Repetir este proceso con la tercera, cuarta,...,y con la enésima posición intercambiando si procede en cada caso.

INTERCAMBIO DIRECTO,
IMPLEMENTACIÓN EN PASCAL

```
PROCEDURE  Intercambiodirecto( var v:tvector);
{Efecto. Se ordena 'v' ascendentemente}
VAR
    i, j:tIntervalo;
    aux:telem;
BEGIN
    for i:=1 to N-1 do
        for j:=N downto i+1
        {se busca el menor desde atrás y se sitúa en v_i}
            if v[j-1]>v[j] then begin
                aux:=v[j];
                v[j]:=v[j-1];
                v[j-1]:=aux;
            end; {if}
END; {intercambiodirecto}
```

ALGORITMOS DE BÚSQUEDA AVANZADA

- Ordenación rápida (Quick sort)
- Ordenación por mezcla (Merge sort)

ORDENACIÓN RÁPIDA

- El algoritmo consiste en dividir el vector que se desea ordenar en dos bloques. En el primero se sitúan todos los elementos que son menores que un cierto valor que se toma como referencia (pivote), mientras que en segundo irían el resto.

- Este procedimiento se repite dividiendo a su vez cada uno de estos bloques y repitiendo la operación anteriormente descrita.

- La condición de parada se da cuando el bloque que se desea ordenar está formado por un único elemento (bloque ordenado).

- El esquema seguido por este algoritmo es el de ‘divide y venceras’.

ORDENACIÓN RÁPIDA, PSEUDOCÓDIGO

Si v es de tamaño 1 entonces
    el vector v ya está ordenado
sino
    dividir v en dos bloques A y B
    con todos los elementos de A menores que los de B
fin {si}
Ordenar A y B usando Quick Sort
Devolver v ya ordenado.

- Donde dividir v en dos bloques se puede refinar:

Elegir un elemento como pivote de v
para cada elemento de v hacer:
    si el elemento es < pivote colocarlo en A
    en otro caso situarlo en B.

ORDENACIÓN RÁPIDA,
IMPLEMENTACIÓN EN PASCAL

```
PROCEDURE  Quicksort( var v:tvector);
{Efecto. Se ordena ‘v’ ascendentemente}
    PROCEDURE Sortdesdehasta(var v:tvector;izq,der:tintervalo);
    {Efecto. Se ordena ‘v[izq..der]’ ascendentemente}
     {siguiente página}

BEGIN {Quicksort}
    Sortdesdehasta(v,1,n);
END; {Quicksort}

PROCEDURE Sortdesdehasta(var v:tvector;izq,der:tintervalo);
    {Efecto. Se ordena ‘v[izq..der]’ ascendentemente}
    VAR
            i, j:tIntervalo;
            p,aux:telem;
    BEGIN
.....
```

ORDENACIÓN RÁPIDA,
IMPLEMENTACIÓN EN PASCAL

```
BEGIN
        i:=izq; j:=der; p:=v[(izq+der) DIV 2];
        while i<j do begin      {se reorganizan los vectores}
                while v[i]<p do
                        i:=i+1;
                while p<v[j] do
                        j:=j-1;
                if  i≤ j then begin {intercambio de elementos}
                        aux:=v[i];
                        v[i]:=v[j];
                        v[j]:=aux;
                        i:=i+1; j:=j-1; {ajuste posiciones}
                end; {if}
        end; {while}
if izq<j then sortdesdehasta(v, izq, j);
if i<der then sortdesdehasta(v, i, der);
END; {Sortdesdehasta}
```

ORDENACIÓN RÁPIDA,
TRAZA DE UN EJEMPLO

V=[0,5,15,9,11]

1. Sortdesdehasta(v,1,5) p=v[3]=15

	i=1	[0,5,15,9,11]	0<15
	i=2	[0,5,15,9,11]	5<15
	i=3	[0,5,15,9,11]	15 not <15
i=3	j=5	[0,5,15,9,11]	11 not >15
i=4	j=4	[0,5,11,9,15]	intercambio. Salida bucle

1.1.sortdesdehasta(v,1,4) p=v[2]=5

	i=1	[0,5,11,9]	0<5
	i=2	[0,5,11,9]	5 not< 5
i=2	j=4	[0,5,11,9]	9>5
i=2	j=3	[0,5,11,9]	11>5
i=2	j=3	[0,5,11,9]	5 not>5
i=3	j=1	[0,5,11,9]	intercambio. Salida bucle

ORDENACIÓN RÁPIDA,
TRAZA DE UN EJEMPLO

1.1.1.sortdesdehasta(v,3,4) p=v[3]=11

i=3 [11,9] 11 not <11

i=4 [11,9] 9 not> 11

i=4 j=3 [9,11] intercambio. Salida bucle

V=[0,5,15,9,11]

1.2 Sortdesdehasta(v,4,5) p=v[4]=11

i=4 [11,15] 11 not <11

i=4 j=5 [11,15] 15>11

i=4 j=4 [11,15] 11 not >11

i=5 j=3 [11,15] intercambio. Salida bucle

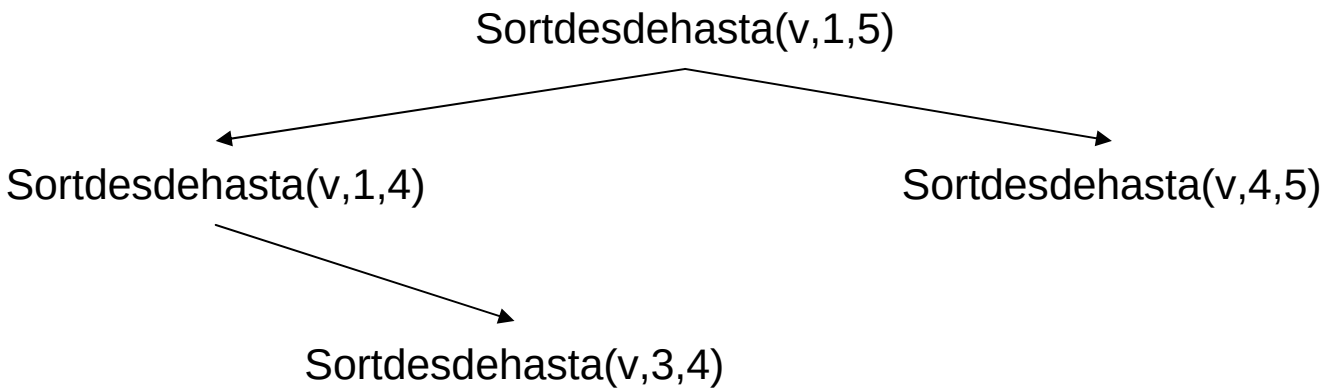
izq=4 not <j=3

i=5 not <der=5

termina recursión

vector ordenado: [0,5,9,11,15]

ORDENACIÓN RÁPIDA,
TRAZA DE UN EJEMPLO



ORDENACIÓN RÁPIDA,
CÁLCULO DE SU COMPLEJIDAD ALGORÍTMICA

- El peor caso: Si se elige como pivote el primer elemento del vector y además se considera que el vector esta ordenado decrecientemente entonces, el bucle '... Para cada elemento...' se ejecutará en total:

  (n-1)+(n-2)+(n-3)+....+1

  Cada miembro de este sumando proviene de cada una de las sucesivas ordenaciones recursivas.

  Este sumatorio da lugar a la siguiente expresión:

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

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

- Como se puede apreciar si la elección del pivote es tal que los subvectores son de igual tamaño y además los vectores están ordenados de forma aleatoria entonces el algoritmo posee una eficiencia del tipo O(n log n).

Recorrido del vector

Numero de divisiones

ORDENACIÓN POR MEZCLA

- En este caso se sigue un esquema similar a la ordenación rápida, el esquema de divide y venceras. Sin embargo el mayor esfuerzo no se realiza dividiendo el vector y reorganizando cada división, sino que tiene lugar cuando finalmente se construye el vector ordenado a partir de la mezcla de los subvectores que se generan.

- La idea consiste en dividir el vector v en dos subvectores para posteriormente mezclar ordenadamente las soluciones obtenidas al ordenar A y B, aplicando nuevamente el algoritmo ‘Merge Sort’ a ambos subvectores.

ORDENACIÓN POR MEZCLA, PSEUDOCÓDIGO

Si v es de tamaño 1 entonces
    el vector v ya está ordenado
sino
    dividir v en dos subvectores A y B
fin {si}

Ordenar A y B usando Mergesort
Mezclar las ordenaciones de A y B para generar el vector ordenado.

- Donde dividir v en dos subvectores se puede refinar:
  Asignar a A el subvector $[v_1,.....,v_{nDIV2}]$
  Asignar a B el subvector $[v_{nDIV2+1},.....v_n]$
- Mientras que mezclar las ordenaciones de A y B consiste en ir entremezclando adecuadamente las componentes ya ordenadas de A y B.

ORDENACIÓN POR MEZCLA,
IMPLEMENTACIÓN EN PASCAL

```
PROCEDURE  Mergesort( var v:tvector);
{Efecto. Se ordena 'v' ascendentemente}
    PROCEDURE Mergedesdehasta(var v:tvector; izq, der: tintervalo);
    {Efecto. Se ordena 'v[izq..der]' ascendentemente}
    VAR
    centro :tIntervalo

          PROCEDURE Merge(var v:tvector; iz, ce, de:tintervalo; var w:tvector);

   BEGIN {Mergedesdehasta}
          centro:=(izq+der)DIV2;
          if izq<centro then
                Mergedesdehasta(v,izq,centro);
          if centro<der then
                Mergedesdehasta(v,centro+1,der);
          Merge(v,izq,centro,der,v)
    END; {Mergedesdehasta}
```

ORDENACIÓN POR MEZCLA,
IMPLEMENTACIÓN EN PASCAL

```
BEGIN {Mergesort}
    Mergedesdehasta(v,1,n)
END; {Mergesort}
```

ORDENACIÓN POR MEZCLA,
IMPLEMENTACIÓN EN PASCAL

```
PROCEDURE Merge(v:tvector; iz, ce, de:tintervalo; var w:tvector);
{Efecto.w: mezcla ordenada de los subvec. v[iz..ce],v[ce+1..de] }
    VAR
        i, j,k:tIntervalo;
    BEGIN
        i:=iz; j:=ce+1; k:=iz;  {k recorre w}
        while( i ≤ ce) and (j ≤de) do begin
            if v[i]<v[j] then begin
                w[k]:=v[i];
                i:=i+1
            end; {if}
            else begin
                w[k]:=v[j];
                j:=j+1;
            end; {else}
            k:=k+1
        end; {while}
```

ORDENACIÓN POR MEZCLA,
IMPLEMENTACIÓN EN PASCAL

```
        for k:=j to de do
                w[k]:=v[k]
                for k:=i to ce do
                        w[k+de-ce]:=v[k]
END; {Merge}
```

ORDENACIÓN POR MEZCLA,
TRAZA DE UN EJEMPLO

V=[8,5,7,3]

1. Mergedesdehasta(v,1,4)centro=2

    1.1. Izq(1) <centro(2) Mergedesdehasta(v,1,2) centro=1

        1.1.1. izq(1) not< centro(1)

        1.1.2. centro(1)< der(2)

            1.1.2.1 izq(2) not< centro(2)

            1.1.2.2. centro(2) not< der(2)

            1.1.2.3. Merge(v,2,2,2,v) trivial

    Merge(v1,1,2,v)-------- w1[5,8]

ORDENACIÓN POR MEZCLA,
TRAZA DE UN EJEMPLO

1.2. Centro(2) < der(4) Mergedesdehasta(v,3,4) centro=3

1.2.1. izq(3) not< centro(3)

1.2.2. centro(3) < der(4)

1.2.2.1. izq(4) not< centro(4)

1.2.2.2. centro(4) not< der(4)

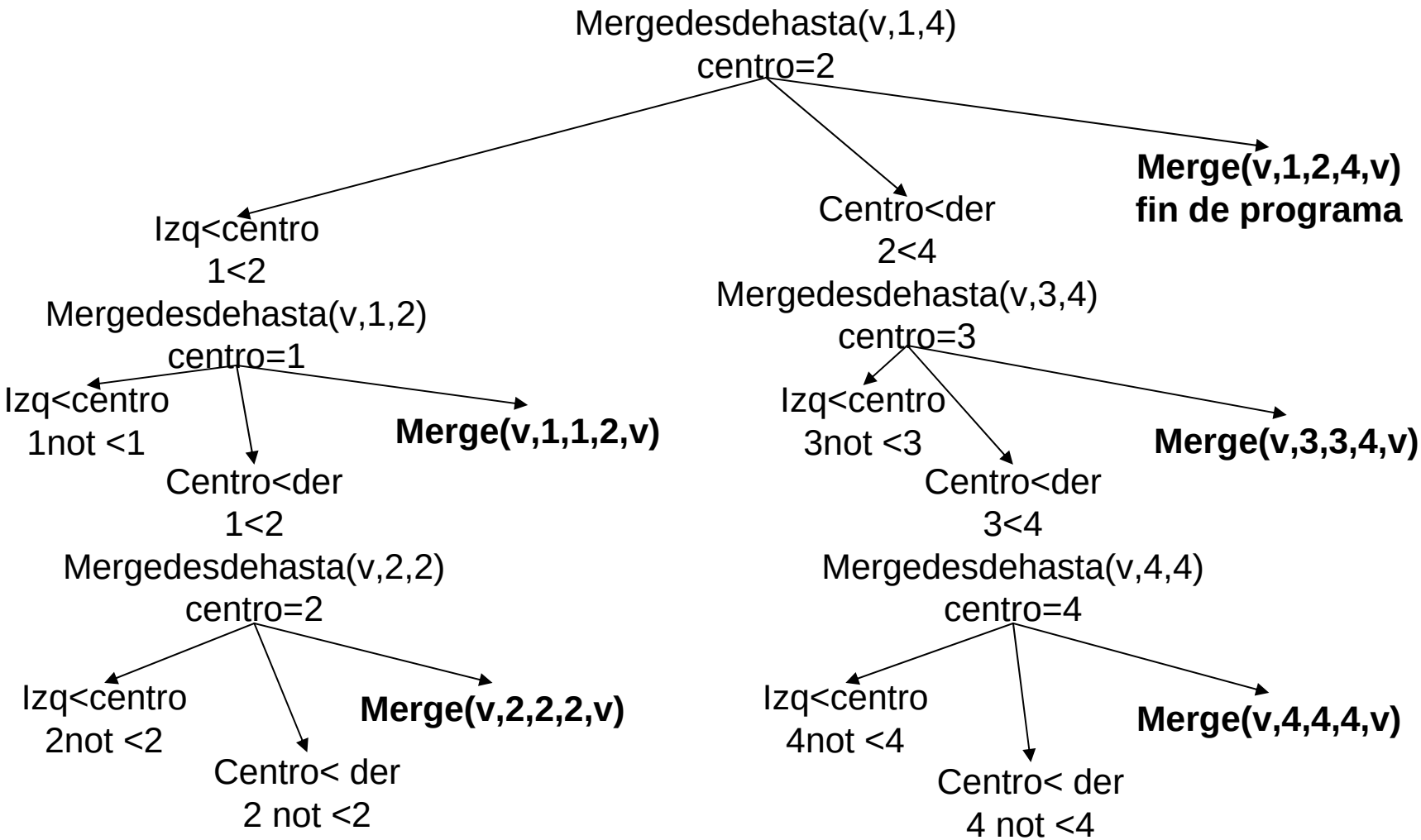
1.1.2.3. Merge(v,4,4,4,v) trivial

Merge(v3,3,4,v)-------- w2[3,7]

1.3. Merge(v,1,2,4,v)----------w[3,5,7,8]

Mezcla ordenada de w1 y w2

ORDENACIÓN POR MEZCLA,
TRAZA DE UN EJEMPLO



ORDENACIÓN POR MEZCLA,
CÁLCULO DE SU COMPLEJIDAD ALGORÍTMICA

- El peor caso: Puesto que la mezcla se ejecuta en un tiempo proporcional a n (la longitud del vector a ordenar) el coste en tiempo vendrá dado por la siguiente recurrencia:

$$T(n) = \begin{cases} k_1 & si \quad n = 1 \\ 2T\left(\frac{n}{2}\right) + k_2 n + k_3 & si \quad n > 1 \end{cases}$$

- Esta ecuación se resuelve mediante sustituciones sucesivas cuando n es una potencia de 2 (j tal que $n=2^j$).

$$\begin{aligned} T(n) &= 2\left(2T(n/4) + k_2(n/2) + k_3\right) + k_2 n + k_3 \\ &= 4T(n/4) + 2k_2 n + k_4 = \\ &= 4\left(2T(n/8) + k_2(n/4) + k_3\right) + 2k_2 n + k_4 = \\ &= 8T(n/8) + 3k_2 n + k_5 = \\ &= \ldots\ldots\ldots \quad \ldots\ldots\ldots \quad \ldots\ldots\ldots \quad \ldots\ldots\ldots \quad \ldots\ldots\ldots \quad . = \\ &= 2^j T(1) + jk_2 n + k_j \end{aligned}$$

- De donde sustituyendo $j \leq \log n$ lo que da una función de orden O(n logn) (una vez aplicadas las reglas de la suma y de la escalabilidad).