© DLSI (Univ. Alicante)

TEMA 2
L'eficiència dels algoritmes

PROGRAMACIÓ I ESTRUCTURES DE DADES

L'eficiència dels algoritmes

- 1. Noció de complexitat
  - Complexitat temporal, magnitud del problema i pas
- 2. Cotes de complexitat
  - Cota superior, inferior i mitjana
- 3. Notació asimptòtica
  - O, Ω, Θ
- 4. Obtenció de cotes de complexitat

1. Noció de complexitat

DEFINICIÓ

- Càlcul de complexitat: determinació de dos paràmetres o **funcions** de cost:
  - **Complexitat espacial:** Quantitat de recursos espacials (memòria) que un algoritme consumeix o necessita per a la seua execució
  - **Complexitat temporal**: Quantitat de temps que un algoritme necessita per a la seua execució
- Possibilitat de fer
  - Valoracions
    - L'algoritme és: "bo", "el millor", "prohibitiu"
  - Comparacions
    - L'algoritme A és millor que el B

1. Noció de complexitat

COMPLEXITAT TEMPORAL

- Factors que afecten al temps d'execució:
  - Externs
    - La màquina en què s'ha d'executar
    - El compilador: variables i model de memòria
    - L'experiència del programador
  - Interns
    - El nombre d'instruccions associades a l'algoritme
- Temps d'execució : ***Temps*(*A*)** = $C + f(T)$
  - *C* és la contribució dels factors externs (constant)
  - *f(T)* és una **funció** que depén de *T* (talla o magnitud del problema)

1. Noció de complexitat

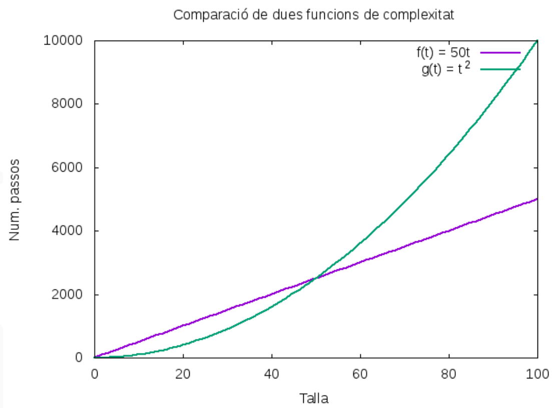
COMPLEXITAT TEMPORAL

- **Talla** o magnitud d'un problema:
  - Valor o conjunt de valors associats a l'**entrada** del problema que representa una mesura de la seua magnitud respecte d'altres entrades possibles
- **Pas** de programa:
  - Seqüència d'operacions amb contingut semàntic el cost de la qual és **independent** de la talla del problema
  - Unitat de mesura de la complexitat d'un algoritme
- Complexitat temporal:
  - **Funció** que expressa el **nombre de passos** de programa que un algoritme necessita executar per a qualsevol entrada possible **en funció de la talla**
  - No es tenen en compte els factors externs

1. Noció de complexitat

COMPLEXITAT TEMPORAL

- Complexitat temporal:
  - **Funció** que expressa el **nombre de passos** de programa que un algoritme necessita executar per a qualsevol entrada possible **en funció de la talla**



1. Noció de complexitat

COMPLEXITAT TEMPORAL. Exemples

```
int ejemplo1 (int n)
{
    n+ = n;
    return n;
}
```

Important: talla = n

ejemplo1: f (n) = 1 pas

Aquestes instruccions s'executen sempre el mateix número de vegades independentment del valor de n. (1 pas)

```
int ejemplo2 (int n)
{
    int i;
    for (i=0; i ≤ 2000; i++)
        n+ = n;
    return n;
}
```

ejemplo2: f (n) = 1 pas

1. Noció de complexitat

COMPLEXITAT TEMPORAL. Exemples

ejemplo3: f (n) = $\overset{1}{1} + \overset{3}{1} \cdot \overset{2}{(n + 1)}$ *passos*

```
int ejemplo3 (int n)
{   int i, j;
    j = 2;
    for (i=0; i ≤ 2000; i++)
        j=j*j;
    for (i=0; i ≤ n; i++)
    {
        j = j + j;
        j = j - 2;
    }
    return j;
}
```

1 Aquestes instruccions s'executen sempre el mateix número de vegades independentment del valor de n. (1 pas)

2 El bucle for es repetirà n+1 vegades

3 Aquestes dos instruccions dins del bucle for sempre es repeteixen el mateix número de vegades. Es consideren 1 pas.

Important: talla = n

1. Noció de complexitat

COMPLEXITAT TEMPORAL. Exemples

1 4 3 2

ejemplo4: $f(n) = 1 + 1 \cdot n \cdot (n + 1)$ *passos*

```
int ejemplo4 (int n)
{   int i, j,k;
    k = 1;

    for (i=0; i ≤ n; i++)
        for (j=1; j ≤ n; j++)
            k = k + k;

    return k;
}
```

1

S'executen sempre el mateix número de vegades independentment del valor de n. (1 pas)

2

El bucle for (i=0; i<=n; i++) es repeteix n+1 vegades

3

El bucle for (j=1; j<=n; j++) es repeteix n veces

4

La instrucció k=k+k és 1 pas

```
int ejemplo5 (int n)
{   int i, j,k;
    k = 1;
    for (i=0; i ≤ n; i++)
        for (j=i; j ≤ n; j++)
            k = k + k;
    return k;
}
```

ejemplo5: $f(n) = 1 + \Sigma_{i=0..n}(\Sigma_{j=i..n} 1)$ *passos*

1. Noció de complexitat

COMPLEXITAT TEMPORAL. Exemples

```
int ejemplo4 (int n)
{   int i, j,k;
    k = 1;
    for (i=0; i ≤ n; i++)
        for (j=1; j ≤ n; j++)
            k = k + k;
        return k;
}
```

ejemplo4: $f(n) = 1 + 1 \cdot n \cdot (n + 1)$ *passos*

1

S'executen sempre el mateix número de vegades independentment del valor de n. (1 pas)

```
int ejemplo5 (int n)
{   int i, j,k;
    k = 1;

    for (i=0; i ≤ n; i++)
        for (j=i; j ≤ n; j++)
            k = k + k;

    return k;
}
```

2

El bucle for (i=0; i<=n; i++) es repeteix n+1 vegades

3

El bucle for (j=i; j<=n; j++) es repeteix n-i +1 vegades

4

La instrucció k=k+k és 1 pas

ejemplo5: $f(n) = 1 + \Sigma_{i=0..n}(\Sigma_{j=i..n} 1)$ *passos*

1. Noció de complexitat

COMPLEXITAT TEMPORAL. Exemples

```
int ejemplo5 (int n)
{   int i, j,k;
    k = 1;

    for (i=0; i ≤ n; i++)
        for (j=i; j ≤ n; j++)
            k = k + k;

    return k;
}
```

ejemplo5: $f(n) = 1 + \Sigma_{i=0..n}(\Sigma_{j=i..n} 1)$ *passos*

1 3 2

2 $$\sum_{j=i}^{n} 1 = 1 \cdot (n - i + 1)$$

3 $$\sum_{i=0}^{n} 1 \cdot (n - i + 1) = \frac{(n+1+1)(n+1)}{2} = \frac{(n+2)(n+1)}{2}$$

Resolució de sumatoris:

$$\sum_{i=m}^{n} C = C \cdot (n - m + 1)$$

$$\sum_{i=1}^{n} i = \frac{(a_1 + a_n)n}{2} (S.P.A)$$

S.P.A
- a_1=n+1
- a_n=1
- nº termes=n+1

1. Noció de complexitat

COMPLEXITAT TEMPORAL. Exercicis

for(i = sum = 0; i < n; i++) sum += a[i]; **(talla = n)**

---

for(i = 0; i < n; i++) { **(talla = n)**
  for(j = 1, sum = a[0]; j <= i; j++) sum += a[j];
  cout << "La suma del subarray " << i << " es " << sum << endl; }

---

for(i = 4; i < n; i++) { **(talla = n)**
  for(j = i–3, sum = a[i-4]; j <= i; j++) sum += a[j];
  cout << "La suma del subarray " << i-4 << " es " << sum << endl; }

---

for(i = 0, length = 1; i < n-1; i++) { **(talla = n)**
  for(i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
  if(length < i2 – i1 + 1) length = i2 – i1 + 1; }

1. Noció de complexitat

CONCLUSIONS

- Només ens ocuparem de la complexitat temporal
- Normalment són objetius contraposats

    (complexitat temporal <--> complexitat espacial)
- Càlcul de la complexitat temporal:
    - *a priori*: comptant passos
    - *a posteriori*: generant instàncies per a distints valors i cronometrant el temps
- Es tracta d'obtindre la funció. Les unitats de mesura (pas, sg, msg ...) no són rellevants (tot es tradueix a un canvi d'escala)
- El nombre de passos que s'executen sempre és funció de la magnitud (o talla) del problema

2. Cotes de complexitat

INTRODUCCIÓ

- Donat un vector X de *n* nombres naturals i donat un nombre natural *z*:
    - Calcula l'index $i : X_i = z$
    - Calculeu el nombre de passos que realitza

El número de vegades que s'executa el bucle "mientras" depén de la grandària del vector i de la distribució interna dels elements

1 pas

```
funcion BUSCAR (var X:vector[N]; z: N): devuelve N
var i:natural  fvar;
comienzo
  i:=1;

  mientras (i ≤|X|) ∧ (Xi≠z) hacer
      i:=i+1;
  fmientras

  si i= |X|+1 entonces devuelve 0      (*No encontrado*)
             si_no devuelve i
fin
```

2. Cotes de complexitat

EL PROBLEMA

- No podem comptar el nombre de passos perquè depèn:
  - De la magnitud del problema |X|
  - De la instància del problema que es pretén resoldre (possible valor que puguen prendre les variables d'entrada)
- Exemple:

X	z	Nº PASSOS
(0, 1)	1	
(1, 2, 3)	1	
(2)	3	
(1, 0, 2, 4)	3	
(1, 0, 2, 4)	0	
(1, 0, 2, 4)	1	

2. Cotes de complexitat

LA SOLUCIÓ: cotes de complexitat

- Quan apareixen diferents casos per una mateixa talla genèrica *n*, s'introdueixen les cotes de complexitat:
  - **Cas pitjor**: cota superior de l'algoritme → $C_s(n)$
  - **Cas millor**: cota inferior de l'algoritme → $C_i(n)$
  - Terme mitjà: cota mitjana → $C_m(n)$
- Totes són funcions de la magnitud del problema (*n*)
- La cota mitjana és difícil d'avaluar *a priori*
  - És necessari conèixer la distribució de la probabilitat d'entrada
  - No és la mitjana de la inferior i de la superior (ni estan totes ni tenen la mateixa proporció)

2. Cotes de complexitat

EXERCICI: cotes superior i inferior

```
funcion BUSCAR (var X:vector[N]; z: N): devuelve N
var i:natural  fvar;
comienzo
  i:=1;

   mientras (i ≤|X|) ∧ (Xi≠z) hacer
      i:=i+1;
   fmientras

   si i= |X|+1 entonces devuelve 0    (*No encontrado*)
              si_no devuelve i
fin
```

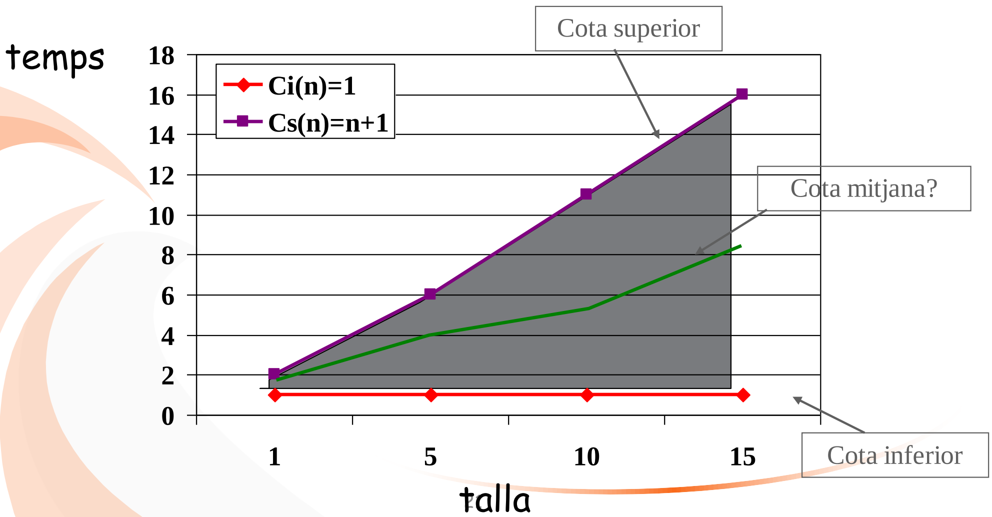
- Talla del problema: nombre d'elements de X: *n*
- Hi ha cas millor i pitjor?
  - Cas millor: l'element està el primer: $X_1 = z \rightarrow c_i(n) = 1$
  - Cas pitjor: l'element no està: $\forall i\ 1 \leq i \leq |X|,\ Xi \neq z \rightarrow c_s(n) = n+1$

$$1 + \sum_{i=1}^{n} 1 = 1 + (n - 1 + 1) = n + 1$$

2. Cotes de complexitat

EJERCICI: cotes superior e inferior

- Complexitat funció Buscar



2. Cotes de complexitat

CONCLUSIONS

- La **cota mitjana** no la calcularem. Només es parlarà de complexitat en el cas mitja quan la cota superior i la inferior coincideixen
- L'estudi de la complexitat es fa per a magnituds grans del problema per diversos motius:
  - Els resultats per a magnituds xicotetes o no són fiables o proporcionen poca informació sobre l'algoritme
  - És lògic invertir temps en el desenvolupament d'un bon algoritme només si es preveu que aquest farà un gran volum d'operacions
- La complexitat que resulta de magnituds grans de problema es denomina **complexitat asimptòtica** i la notació utilitzada és la notació asimptòtica

3. Notació asimptòtica

INTRODUCCIÓ

- Notació matemàtica utilitzada per a representar la complexitat espacial i temporal quan $n \rightarrow \infty$
- Es defineixen classes d'equivalència que engloben les funcions que "creixen de la mateixa forma"
- Es defineixen tres tipus de notació:
  - Notació O (big-omicron) $\Rightarrow$ cas pitjor
  - Notació Ω (omega) $\Rightarrow$ cas millor
  - Notació Θ (big-theta) $\Rightarrow$ cas mitjà

3. Notació asimptòtica

Teorema de l'escala de complexitat

$$O(1) \subset \underbrace{O(\lg\lg n)}_{\text{polilogarísmiques}} \subset \underbrace{O(\lg n) \subset O(\lg^{a>1} n)}_{\text{logarísmiques}} \subset \underbrace{O(\sqrt{n})}_{\text{sublineals}} \subset \underbrace{O(n)}_{\text{lineals}} \subset$$

$$\subset \underbrace{O(n \lg n) \subset O(n^2) \subset \cdots \subset O(n^{a>2})}_{\text{polinòmiques}} \subset \underbrace{O(2^n) \subset O(n!) \subset O(n^n)}_{\text{exponencials}}$$

- $f(n) + g(n) + t(n) \in O(Max(f(n), g(n), t(n)))$
- Exemples:
  - $10000000n + 1$ pertany a $O(n)$
  - $n^2 + \log n$ pertany a $O(n^2)$
  - $n^3 + 2^n + n\log n$ pertany a $O(2^n)$
- Vàlid per Notació Ω y Notació Θ

3. Notació asimptòtica

NOTACIÓ O: escala de complexitat

Complexitat	*n = 32*	*n = 64*
n^3	3 seg.	26 seg.
2^n	5 dies	$58 \cdot 10^6$ anys

- Temps de resposta per a dos valors de la talla i complexitats n^3 i 2^n.

(pas = 0,1 mseg.)

– Queda clara la necessitat del càlcul de complexitat

```
función POT_2 (n: natural): natural
    opción
        n = 1: devuelve 2
        n > 1: devuelve 2 * POT_2(n-1)
    fopción
ffunción
```

Cost lineal

1 seg.

```
función POT_2 (n: natural): natural
    opción
        n = 1: devuelve 2
        n > 1: devuelve POT_2(n-1)+POT_2(n-1)
    fopción
ffunción
```

Cost exponencial

miles d'anys

4. Obtenció de cotes de complexitat

INTRODUCCIÓ

- Etapes per a obtindre les cotes de complexitat:
  1. Determinació de la **TALLA** o magnitud (de la instància) del problema
  2. Determinació del **CAS MILLOR I MITJOR**: instàncies per a les quals l'algoritme tarda més o menys
     - No sempre hi ha millor i pitjor cas, ja que hi ha algoritmes que es comporten de la mateixa forma per a qualsevol instància de la mateixa grandària
  3. Obtenció de les cotes per a cada cas. Mètodes:
     - **compte de passos**
     - **relacions de recurrència** (funcions recursives)

4. Obtenció de cotes de complexitat

INTRODUCCIÓ

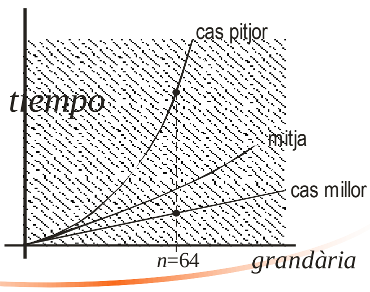
```
función FACTORIAL (n:natural): natural
```

- La talla és *n* i no hi ha cas millor ni pitjor

```
función BUSCA (v: vector[natural]; x:natural)
```

- La talla és $n=|v|$
- cas millor: instàncies on x està en $v[1]$
- cas pitjor: instàncies on x no està en v

- Es tracta de delimitar amb una regió el temps que tarda un algoritme en executar-se



4. Obtenció de cotes de complexitat

Exemples

Cálcul del màxim d'un vector

```
funcion  MÁXIMO  (var  v : vector[n]; n:entero) : entero
var  i, max : entero fvar
comienzo
   max:=v[1]
   para i:=2 hasta n hacer
       si v[i]>max  entonces  max:=v[i] fsi
   fpara
   devuelve max
fin
```

1 paso

- determinar la **talla** del problema: ***n****=grandària del vector*

- **Millor cas** $c_i = 1 + \sum_{i=2}^{n} 1 = 1 + (n-2+1) = n \in \Omega(n)$

La condició v[i]>max MAI es compleix

- **Pitjor cas** $c_s = 1 + \sum_{i=2}^{n} 2 = 1 + (n-2+1) \;=\; n \quad \in O(n)$

$\in \Theta(n)$

La condició v[i]>max SEMPRE es compleix

4. Obtenció de cotes de complexitat

Exemples

Búsqueda d'un element en un vector ordenat (Busca binària)

```
funcion BUSCA (var v:vector[N]; x,pri,ult: natural): natural
var m: natural fvar
comienzo
   repetir
      m:= (pri+ult)/2
      si v[m]>x  entonces ult:= m-1
                 sino pri:= m+1
      fsi
   hasta  (pri>ult) ∨ v[m]=x
   si v[m]=x  entonces devuelve m
              sino devuelve 0
   fsi
fin
```

4. Obtenció de cotes de complexitat

Exemples

- Determinar la **talla** del problema: ***n****=grandària del vector*
- ***Millor cas****: x està en la meitat del vector*
- ***Pitjor cas****: x no està en el vector*
- *Complexitats*
  - ***millor cas****: 1+1=2* $\in \Omega(1)$
  - ***pitjor cas***
    - ***1+u·1****, on u és el nombre de vegades que s'executa el bucle*

      1ª iteració: Talla=n

      2ª iteració: Talla=n/2

      3ª iteració: Talla=n/4

      . . . .

      k-èsima interació: $Talla=n/2^{(k-1)}$

      . . . .

      última iteració: Talla = 1 $\mathbf{(n/2^{(u-1)} = 1)}$
    - *És a dir, en l'última iteració nomes ens queda 1 element.*

    Aïllant u:

    $u=\log_2 n+1$

    $1+ u \cdot 1=1+(\log_2 n+1) \in O(\log_2 n)$

4. Obtenció de cotes de complexitat

Algoritmes d'ordenació

- Directes
  - Inserció directa
  - Inserció binària
  - Selecció directa
  - Intercanvi directe (bambolla)

4. Obtenció de cotes de complexitat

Algoritmes d'ordenació

INSERCIÓ DIRECTA

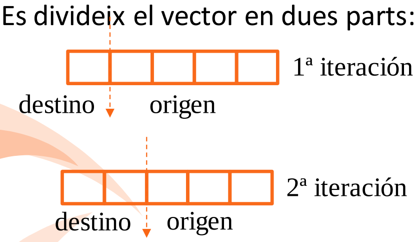
- Divideix lògicament el vector en dues parts: origen i destí
- Començament:
  - *destí* té el primer element del vector
  - *origen* té els *n-1* elements restants
- Es va prenent el primer element d'*origen* i s'insereix en *destí* en el lloc adequat, de manera que *desti* sempre està ordenat
- L'algoritme finalitza quan no queden elements en *origen*
- Característiques
  - cas millor: vector ordenat
  - cas pitjor: vector ordenat inversament

4. Obtenció de cotes de complexitat

Algoritmes d'ordenació

INSERCIÓ DIRECTA

•Es divideix el vector en dues parts: origen i destí



- En cada iteració, l'element a[i] del subvector "origen" s'insereix en la seua posició correcta del subvector "destí" a[1..i-1]

4. Obtenció de cotes de complexitat

Algoritmes d'ordenació

INSERCIÓ DIRECTA

```
funcion  INSERCION_DIRECTA  (var a:vector[natural];  n: natural)
var i,j: entero;  x:natural  fvar
comienzo
    para  i:=2  hasta  n  hacer
        x:=a[i];  j:=i-1
        mientras   (j>0) ∧ (a[j]>x)    hacer
            a[j+1]:=a[j]
            j:=j-1
        fmientras
        a[j+1]:=x
    fpara
fin
```


4. Obtenció de cotes de complexitat

Algoritmes d'ordenació

INSERCIÓ BINÀRIA

EL BUCLE mientras BUSCA EL LLOC DEL PRIMER ELEMENT D' **ORIGEN** EN **DESTÍ**.
EL COST D'AQUEST BUCLE **SEMPRE** ÉS **log2(i)** (CERCA BINÀRIA).
I LA SUMA PER A i=2..n DE log2(i) es pot aproximar com **n*log2(n)**

```
funcion  INSERCION_BINARIA  (var a:vector[natural];  n: natural)
var i, j, iz, de, m: entero;  x:natural  fvar
comienzo
    para  i:=2  hasta  n  hacer
        x:=a[i]; iz:=1; de:=i-1
        mientras  (iz≤de)  hacer
            m:= (iz+de)/2
            si a[m]>x  entonces de:= m-1
                       sino iz:=m+1
            fsi
        fmientras
        para  j:=i-1  hasta iz hacer  (*decreciente*)
            a[j+1]:=a[j]
        fpara
        a[iz]:=x
    fpara
fin
```

4. Obtenció de cotes de complexitat

Algoritmes d'ordenació

INSERCIÓ BINÀRIA

- És una millora de l'algoritme d'inserció directa
- Canvia en un punt:
  - Quan es busca la posició on s'ha d'inserir l'element en el subvector "destí", es fa de forma dicotòmica: es dividieix el vector "destí" en dues parts de manera successiva fins que es troba la posició correcta.
  - Quan es troba la posició, la resta d'elements es mouen cap a la dreta.

4. Obtenció de cotes de complexitat

Algoritmes d'ordenació

SELECCIÓ DIRECTA

EN CADA ITERACIÓ, ES SELECCIONA EL MÍNIM D'**ORIGEN** I S'INTERCANVIA PER L'ÚLTIM DE **DESTÍ**.

```
funcion  SELECCION_DIRECTA  (var a:vector[natural]; n:natural)
var i, j, posmin: entero;  min:natural  fvar
comienzo
    para  i:=1  hasta  n-1  hacer
        min:=a[i];  posmin:=i
        para  j:=i+1  hasta  n  hacer
            si  a[j]<min  entonces
                min:=a[j]; posmin:=j
            fsi
        fpara
        a[posmin]:=a[i];  a[i]:=min
    fpara
fin
```

Aquest algoritme busca el menor element del subvector a[i..n-1] y ho intercanvia per l'element que està en la posició i

4. Obtenció de cotes de complexitat

Algoritmes d'ordenació

INTERCANVI DIRECTE (bambolla)

Es fa un recorregut del vector de dreta a esquerra (**n-i posicions)** i s'intercanvia cada element per l'anterior si l'element actual es menor.

```
funcion INTERCAMBIO_DIRECTO  (var a:vector[natural]; n:natural )
var i,j:entero  fvar
comienzo
    para  i:=2  hasta  n  hacer
        para  j:=n  hasta  i  hacer
            si  a[j]<a[j-1]  entonces
                SWAP(a[j],a[j-1])
            fsi
        fpara
    fpara
fin
```