Fundamentos de Análise e Deseño de Algoritmos e Estruturas de Dados

---

TEMA 1: ANÁLISE DE ALGORITMOS.

1.1.- Recursividade.

■ Exemplo 1: a sucesión de Fibonacci. A función de Fibonacci, da que se obtên os termos da sucesión homónima, asigna un número inteiro a cada inteiro non negativo do seguinte xeito:

$Fib : Z \rightarrow Z$
$0 \rightarrow 1$
$1 \rightarrow 1$
$n \rightarrow Fib\ (n-1) + Fib\ (n-2)$ onde $n>1$

A primeira prática da Fundamentos de Análise e Deseño de Algoritmos e Estruturas de Dados consiste en primeiro lugar en realizar en Sun Pascal três algoritmos distintos que implementan a función de Fibonacci, para comprobar como a elección de un algoritmo ou outro, que fan o mesmo, determina tempos de espera de valores extremadamente diferentes.

As ordens dos algoritmos son as seguintes:

$fibo1\ (n) = O(\phi^n)$

$\phi = (1+\sqrt{5})/2 = 1.618$

$fibo2\ (n) = O(n)$
$fibo3\ (n) = O(\log n)$

[“O’s grandes”. Ver aptdo 1.2.1]

A medición de tempos pode-se abordar mediante a estratéxia empírica ou a posteriori -como no exemplo 1- ou mediante a estratéxia teórica ou a priori -a que estudamos nesta asignatura-.

A estratéxia teórica é unha análise formal, mentres que a empírica consiste en que o programa se executa na máquina só sobre uns exemplares de proba, e a partir dos valores obtidos supón-se que para todos, en xeral, hai unha correlación. *Esta técnica non permite comprobar a corrección do algoritmo.*

As medicións que se obtên amosan que a granularidade do mecanismo clock de medición de tempos en Sun Pascal non é o fino abondo.

Unha solución que se pode aportar, que é a aplicada na primeira prática, é calcular os tempos para execucións repetidas dos algoritmos e dividir o tempo total entre o número de veces que se executou. Por exemplo, se se calcula fibo2 (10) vinte veces e se tarda t2 ms, pode-se afirmar -sen certeza absoluta pero con unha maior aproximación que a primeira técnica- que fibo2 (10) tarda aproximadamente t2/20 ms.

Cando sexa posíbel debe-se penalizar equitativamente os distintos algoritmos que implementan un mesmo conceito. No exemplo 1, isto consistiria en calcular o mesmo número de veces as três implementacións de Fibonacci: por exemplo 20 veces fibo1 (10), 20 veces fibo2 (10) e 20 veces fibo3 (10). Na tábua de resultados obviamente porian-se os valores correspondentes a unha vez (unha execución), é dicer os cocientes t1/20, t2/20 e t3/20.

Debe-se deseñar unha estratéxia de medición que se soe acompañar dos resultados; non hai necesidade de ocultar a técnica usada à hora de apresentar os valores obtidos.

A ideia da ***verificación de algoritmos*** (apartado 1.3.3.) consiste en que a partir dos valores das tábuas comproba-se que as ordens de tempos correspondentes se cumpren. Isto aplicaremo-lo à primeira prática.

Adiantamos agora o fundamental da verificación de algoritmos para empezar a usar esta técnica.

Como verificar que un programa tarda un tempo que é da orden de F(n), o que se denota O(F(n))?

1º) Sexa T(n) o tempo de execución medido.
2º) Calculamos os valores T(n)/F(n) e estudamos a sua converxência cando n → ∞ (sendo n o parámetro de entrada)
-se converxen cara unha constante positiva, entón verifica-se F(n).
-se converxen cara cero entón F(n) foi sobreestimada.
-se diverxen (non hai un comportamento normal) F(n) foi subestimada.

■ Exemplo 2: Un problema de selección: Determinar o k-ésimo maior número de un conxunto de n números.

Hai duas solucións que son as mais sinxelas ainda que non as mais eficientes:

Algoritmo 1: Ler todos os números e pô-los nun vector de tamaño n; despois ordenar este vector -de maior a menor- por un método (burbulla, inserción), e por último tomar o k-ésimo elemento.

Algoritmo 2: Ler os k primeiros números e pô-los nun vector de tamaño k; despois ordenar este vector de maior a menor. A continuación van-se lendo os elementos restantes do k+1 até o n, se o número é menor que o k-ésimo elemento do vector, ignórase; se o número é maior que o k-ésimo elemento do vector, inserta-se na posición ordenada do vector desprazando os elementos menores, e saindo o menor de todos do vector. Ao final a solución é o k-ésimo elemento -o derradeiro- do vector.

Exemplo do algoritmo 2:
Elementos: 9 2 3 5 1 6 4 8 7 (n = 9)
Determinar o quinto maior elemento (k = 5)
Vector: 9 2 3 5 1
Ordenación, vector: 9 5 3 2 1
A referência é o k-ésimo elemento (o quinto) que é 1
Percórren-se 6 4 8 7 comparando-os co 1
6 > 1 → 9 6 5 3 2
A referência é agora o 2
4 > 2 → 9 6 5 4 3
8 > 3 → 9 8 6 5 4
7 > 4 → 9 8 7 6 5
A solución é o k-ésimo elemento, o quinto.

Un caso particular é cando k = n/2, trata-se do problema de calcular a mediana de un conxunto de valores. Neste caso os dous algoritmos son igual de malos.

Os problemas de tempo vên dados pola cantidade ou tamaño dos dados de entrada, como vimos de ver neste exemplo.

1.2.- Análise da eficiência dos algoritmos.

Primero hai que asegurar-se de que o algoritmo sexa correcto, despois hai que mirar se o algoritmo é eficiente. Isto fai-se determinando a cantidade de recursos necesários (como por exemplo o tempo de execución).

1.2.1.- Notacións asintóticas.

Def.- **T(n) = O (f(n))** ["f(n) cota superior de T(n)"]
se existen constantes c, n_0 / **T(n)** ≤ **f(n)·c** cando n ≥ n_0
(taxa de crescimento de T(n) £ taxa de crescimento de f(n))

Def.- **T(n) = Ω(g(n))** ["g(n) cota inferior de T(n)"]
se existen constantes c, n_0 / **T(n)** ≥ **g(n)·c** cando n ≥ n_0
(taxa de crescimento de T(n) ³ taxa de crescimento de g(n))

Def.- **T(n) = θ (h(n))**
⇔ T(n) = O (h(n)) ∧ T(n) = Ω (h(n))
(taxa de crescimento de T(n) = taxa de crescimento de h(n))

Def.- **T(n) = o (p(n))**
⇔ T(n) = O (p(n)) ∧ T(n) ≠ θ (p(n))
(taxa de crescimento de T(n) < taxa de crescimento de p(n))

*Nota: Advertir a diferenza entre "O" e "o": "O" permite que as taxas se igualen mentres que "o" non.

O "o grande"
Ω "omega"
θ "theta"
o "o pequena"

$T(n)=O(f(n)) \Leftrightarrow f(n)=\Omega(T(n))$

Exemplo: $n^2 = O(n^3)$ o cuadrado medra menos rápido que o cubo
$n^3 = \Omega(n^2)$ o cubo medra mais rápido que o cuadrado

Obxectivo de estas notacións asintóticas: ***estabelecer unha orden relativa entre as funcións*** comparando a sua taxa de crescimento relativa.

Exemplo: $T(n) = 1000 \cdot n$
$f(n) = n^2$
podemos dicer que $T(n)=O(f(n))$?

n	$T(n) = 1000 \cdot n$	$f(n) = n^2$	$T(n) \leq c \cdot f(n)$
1	1000	1	si para c = 1000
2	1000	4	si para c = 250
...	...	...	(c non constante)
1000	1000000	1000000	si para c = 1
1001	1001000	1002001	si para c = 1
1002	1002000	1004004	si para c = 1
...	...	...	(c si constante)

$n_0 = 1000$
$c = 1$

Observamos que a partir de $n_0 = 1000$ e ignorando os factores constantes, $f(n) = n^2$ é unha cota superior para $T(n) = 1000 \cdot n$

$T(n) = O(f(n))$
$1000 \cdot n = O(n^2)$

interpreta-se como $1000 \cdot n \leq n^2$
"n^2 é ao menos tan grande como $1000 \cdot n$"

☠☞ é incorrecto **$T(n) \leq O(f(n))$** porque existe unha relación de comparación implícita, pero non debe expresar-se directamente.

Exemplo:

$$\left.\begin{array}{l} n^2 = O(2n^2) \\ n^2 = \Omega(2n^2) \end{array}\right\} \Rightarrow n^2 = \theta(2n^2)$$

as taxas de crescimento de n^2 e $2n^2$ son iguais

Dado que
$2n^2 = O(n^2)$
$2n^2 = O(n^3)$
$2n^2 = O(n^4)$

o que hai que entender é que, como estamos dando unha série de cotas superiores para $2n^2$, o mellor será sempre dar a menor de elas; por iso é preferíbel dicer $2n^2 = O(n^2)$ que $2n^2 = O(n^3)$ ou que calquer $2n^2 = O(n^x)$ onde $x > 2$

De calquer xeito, à hora da verdade no cálculo de cotas usarán-se resultados e non estas definicións.

REGRAS

Empregando as seguintes três regras poderá-se ordenar por taxa de crescimento à maioria das funcións:

(1) $T_1(n) = O(f(n)) \wedge T_2(n) = O(g(n))$ entón:

a) $T_1(n)+T_2(n) = O(f(n)+g(n))$ [=max(O(f(n)),O(g(n)))]

b) $T_1(n)\cdot T_2(n) = O(f(n)\cdot g(n))$

☞ en xeral non é certo que $T_1(n)-T_2(n) = O(f(n)-g(n))$
nen que $T_1(n)/T_2(n) = O(f(n)/g(n))$

(2) $T(x)$ = polinómio de grau $n \Rightarrow T(x) = \theta(x^n)$

"a taxa de crescimento de un polinómio é igual que a taxa de crescimento do seu termo de maior exponente"

(3) $\log^k n = O(n)\ \forall$ k constante

"os logaritmos medran mui lentamente"

demostracións

(1a)

$T1 = O(f(n)) \Leftrightarrow \exists\ n1, c1\ /\ \forall\ n \geq n1,\ T1(n) \leq c1\cdot f(n)$
$T2 = O(g(n)) \Leftrightarrow \exists\ n2, c2\ /\ \forall\ n \geq n2,\ T2(n) \leq c2\cdot g(n)$

sexan
$n3 = \max(n1, n2)$
$c3 = \max(c1, c2)$

daquela
$\exists\ n3, c3\ /\ \forall\ n \geq n3,\ T1(n) \leq c3\cdot f(n)$
$\exists\ n3, c3\ /\ \forall\ n \geq n3,\ T2(n) \leq c3\cdot g(n)$

entón
$\exists\ n3, c3\ /\ \forall\ n \geq n3,\ T1(n)+T2(n) \leq c3\cdot f(n)+c3\cdot g(n) = c3(f(n)+g(n))$*
$\Leftrightarrow T1(n)+T2(n)=O(f(n)+g(n))$

asi que
$T1 = O(f(n)) \wedge\ T2 = O(g(n)) \Rightarrow T1(n)+T2(n)=O(f(n)+g(n))$ c.q.d.

*tamén podemos obter a outra expresión continuando
$\exists\ n3, c3\ /\ \forall\ n \geq n3,\ T1(n)+T2(n) \leq c3\cdot f(n)+c3\cdot g(n) = c3(f(n)+g(n)) \leq c3\cdot 2\cdot[\max(f(n),g(n))]$

sexa
$c = c3\cdot 2$

entón
$\exists\ n3, c\ /\ \forall\ n \geq n3,\ T1(n)+T2(n) \leq c\cdot[\max(f(n),g(n))]$
$\Leftrightarrow T1(n)+T2(n)=O(\max(f(n),g(n)((($
$\Leftrightarrow T1(n)+T2(n)=\max(O(f(n)),O(g(n)))$

asi que
$T1 = O(f(n)) \wedge\ T2 = O(g(n)) \Rightarrow T1(n)+T2(n)=\max(O(f(n)),O(g(n)))$ c.q.d.

(1b)

$T1 = O(f(n)) \Leftrightarrow \exists\ n1, c1\ /\ \forall\ n \geq n1,\ T1(n) \leq c1\cdot f(n)$
$T2 = O(g(n)) \Leftrightarrow \exists\ n2, c2\ /\ \forall\ n \geq n2,\ T2(n) \leq c2\cdot g(n)$

sexan
$n3 = \max(n1, n2)$
$c3 = \max(c1, c2)$

daquela
$\exists\ n3, c3\ /\ \forall\ n \geq n3,\ T1(n) \leq c3\cdot f(n)$
$\exists\ n3, c3\ /\ \forall\ n \geq n3,\ T2(n) \leq c3\cdot g(n)$

entón
$\exists\ n3, c3\ /\ \forall\ n \geq n3,\ T1(n)\cdot T2(n) \leq c3\cdot f(n)\cdot g(n)$

por definición
$\Leftrightarrow T1(n)\cdot T2(n)=O(f(n)\cdot g(n))$

asi que
$T1 = O(f(n)) \wedge T2 = O(g(n)) \Rightarrow T1(n)\cdot T2(n)=O(f(n)\cdot g(n))$ c.q.d.

(2)

$T(x)=dx^n+fx^{n-1}+...+kx+e$
por (1a) $T(x)=\max(O(dx^n),O(fx^{n-1}),...,O(kx),O(e))=O(dx^n)=O(x^n)$

Taxas características.-

-constante O(1)

-logarítmica O(log n)

-logarítmica cuadrada O($\log^2 n$)

-lineal O(n)

-cuadrática O(n^2)

-cúbica O(n^3)

-exponencial O(2^n)

*Nota: *Non se deben incluir constantes ou termos de orden menor nunha O grande*. É por iso que:

-non debe dicer-se O($2n^2$) senón O(n^2) $2n^2>n^2$

-non debe dicer-se O($n+n^2$) senón O(n^2) $n+n^2>n^2$

-non debe dicer-se O(15) senón O(1) $15>1$

-non debe dicer-se O($\log_2 n$) senón O(log n) $\log_2 n>\log_{10} n$

axustamos (reducimos) cota superior

Comparativa de tempos: Algoritmo Vs Máquina.-

-de arriba a abaixo: de mellor a pior algoritmo
-de esquerda a direita: de pior a mellor máquina

(tempos de execución en segundos para n = 1000)

	1,000 pasos/sg	2,000 pasos/sg	4,000 pasos/sg	8,000 pasos/sg
$\log_2 n$	.01	.005	.003	.001
n	1	.5	.25	.125
$n \cdot \log n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1,000	2,500	250	125
n^3	1,000,000	500,000	250,000	125,000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

A conclusión obvia é que é muito mais útil escoller algoritmos mais eficientes que cambiar de máquina a unha mais potente.

Teorema I

Hipótese

$\forall$ **c > 0, a > 1**
$\forall$ **f(n) monótona crescente: $n_1 \geq n_2 \Rightarrow f(n_1) \geq f(n_2)$**

Entón

$\mathbf{(f(n))^c = O(a^{f(n)})}$

"unha función exponencial medra mais rápido que unha función polinómica"

Casos particulares importantes:

$n^c = O(a^n)$

$(\log_a n)^c = O\left(a^{\log_a n}\right) = O(n)$ $\quad \forall\ c > 0,\ a > 1$

Exemplo:

$5n \cdot \log_2 n - 10 = \theta(n \cdot \log n)$

(medran exactamente da mesma maneira)

para isto, hai que probar tanto O como Ω:

O: achando $c,\ n_0$

Ω: achando $c',\ n'_0$

$c \neq c',\ n_0 \neq n'_0$

Teorema II

Poden-se determinar as taxas de crescimento relativas de f(n) e g(n) mediante

$$\lim_{n\to\infty} f(n)/g(n)$$

se é necesário, pode-se usar L´Hôpital que di que:

$$\lim_{n\to\infty} f(n) = \infty$$
$$\lim_{n\to\infty} g(n) = \infty$$

entón:

$$\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} f'(n)/g'(n)$$

do seguinte xeito:

$\lim_{n\to\infty} f(n)/g(n) = 0 \Rightarrow f(n) = o(g(n))$ $\quad n/\log_2 n = o(n)$

$T.C.f(n) < T.C.g(n)$

$\lim_{n\to\infty} f(n)/g(n) = c \neq 0 \Rightarrow f(n) = \theta(g(n))$

$T.C.f(n) = T.C.g(n)$

$\lim_{n\to\infty} f(n)/g(n) = \infty \Rightarrow g(n) = o(f(n))$

$T.C.g(n) < T.C.f(n)$

Exemplo: cal de estas duas funcións medra mais ràpidamente?

$n \cdot \log n$
$n^{1.5}$

Recordamos que non se deben incluir constantes ou termos de orden menor nas expresións de taxas de crescimento, entón calcular cal medra mais rápido de $(n \cdot \log n, n^{1.5}) \Leftrightarrow$ [%n] cal medra mais rápido de $(\log n, n^{0.5}) \Leftrightarrow$ [↑2] cal medra mais rápido de $(\log^2 n, n)$. Como n medra mais rápido que calquer potência de un logaritmo entón $n^{1.5}$ medra mais rápido que $n \cdot \log n$.

1.2.2.- Modelo de computación.

Manexamos as seguintes hipóteses:

(1) O ordenador é secuencial (PC normal, un só procesador).

(2) As instruccións sinxelas (+, ·, comparación, asignación) consomen o mesmo tempo (noción de operación ou instrucción elemental: aquela que o seu tempo de execución está acotado superiormente por unha constante que só depende da implementación dada -da linguaxen, da máquina, etc-). Só o número de estas operacións será relevante e diremos que consomen o mesmo tempo (unha unidade de tempo) sen pararnos a medir que é esta unidade (dependerá da implementación).

(3) Imos manexar inteiros de tamaño fixo (32 bits) e o ordenador non dispón de instruccións refinadas complexas como a inversión de matríces.

(4) Dispomos de memória infinita.

1.3.- Cálculo dos tempos de execución.

Hai factores xerais e considerados (estes últimos: algoritmo e entrada).

1.3.1.- Análise dos casos pior e médio.

T médio (n) ≤ T pior (n)

Para un algoritmo dado podemos ter un tempo médio en función do tamaño da entrada e un tempo para o pior dos casos.

***** Ordenación por inserción:**

A ordenación por inserción consiste en ir considerando porcións crescentes do vector, que xá están ordenadas.

```
T = [ 3 1 4 1 5 9 2 6 5 3 ]
      3
      1 3
      1 3 4
      1 1 3 4
      1 1 3 4 5
      1 1 3 4 5 9
      1 1 2 3 4 5 9
      1 1 2 3 4 5 6 9
      1 1 2 3 4 5 5 6 9
      1 1 2 3 3 4 5 5 6 9
```

```
procedimento ordenación por inserción
   (var A: arranxo; n: integer);
   {A[1..n]}
   var
        i, j: integer; tmp: dado;
   begin
        for i:=2 to n do begin
             tmp := A[i];
        j := i-1;
        while (j>0)
        and (A[j]>tmp) do begin
             A[j+1] := A[j];
             j := j-1;
        end;
        A[j+1] := tmp;
   end;
```

arranxo	i	j	k
3 1 4 1 5 9 2 6 5 3	1	1	0
1 3 4 1 5 9 2 6 5 3	2	1	2
1 3 4 1 5 9 2 6 5 3	3	3	2
1 1 3 4 5 9 2 6 5 3	4	1	2
1 1 3 4 5 9 2 6 5 3	5	5	2
1 1 3 4 5 9 2 6 5 3	6	6	2
1 1 2 3 4 5 9 6 5 3	7	3	4
1 1 2 3 4 5 6 9 5 3	8	7	8
1 1 2 3 4 5 5 6 9 3	9	6	7
1 1 2 3 3 4 5 5 6 9	10	4	5

(O anterior é o contido do arquivo inserc.dat producido polo seguinte código compilado baixo Turbo-Pascal 6.0 de Borland:)

```
program traza_insercion ;

const
    card = 10 ;

type
   arranxo = array [ 1 .. card ] of integer ;

var
  T : arranxo ;
  saída : text ;

procedure amosa ( T : arranxo ; i , j , k : integer ) ;
var
  ind : integer ;
begin
    for ind := 1 to card do begin
       write ( saída ,  ' ' , T [ ind ] )
    end
    ; writeln ( saída , '        ' , i : 3
           , ' ' , j : 3 , ' ' , k : 3 )
end ;

procedure inicializa ( var T : arranxo ) ;
begin
    { T = [ 3 9 4 1 5 9 2 6 5 3 ] }
    T[1] := 3 ; T[2] := 1 ; T[3] := 4 ; T[4] := 1 ; T[5] := 5 ;
    T[6] := 9 ; T[7] := 2 ; T[8] := 6 ; T[9] := 5 ; T[10] := 3 ;
end ;

procedure cabeceira ;
begin
    writeln ( saída ,
    ' arranxo                    i   j   k' ) ;
    writeln ( saída ,
    '------------------------------ --  --  --' )
end ;

begin
    assign ( saída , 'inserc.dat' )
    ; rewrite ( saída )
    ; inicializa ( T )
    ; cabeceira
    ; ord_inserc ( T )
    ; close ( saída )
end .
```

Observamos o comportamento do algoritmo de ordenación por inserción nos casos de arranxos ordenados crescente e decrescentemente:

```
 arranxo                 i    j    k
---------------------- --   --   --
 1 2 3 4 5 6             1    1    0
 1 2 3 4 5 6             2    2    0
 1 2 3 4 5 6             3    3    0
 1 2 3 4 5 6             4    4    0
 1 2 3 4 5 6             5    5    0
 1 2 3 4 5 6             6    6    0

 arranxo                 i    j    k
---------------------- --   --   --
 6 5 4 3 2 1             1    1    0
 5 6 4 3 2 1             2    1    2
 4 5 6 3 2 1             3    1    2
```

```
 3 4 5 6 2 1           4    1    2
 2 3 4 5 6 1           5    1    2
 1 2 3 4 5 6           6    1    2
```

Observa-se que na ordenación por inserción o pior caso é o de ter o arranxo orixinariamente ordenado descendentemente.

***** Ordenación por selección:**

```
procedimento ordenación por selección ( T [ 1 .. n ] )
   para i ← 1 até n-1 facer
        minj ← i
        minx ← T [ i ]
        para j ← i+1 até n facer
             se T [ j ] < minx entón
                  minj ← j
                  minx ← T [ j ]
             finse
        finpara
        T [ minj ] ← T [ i ]
        T [ i ] ← minx
   finpara
finprocedimento
```

Para cada posición mira-se todo o vector (à direita) buscando o menor (minx) dos elementos inferiores ao de referência, e substituen-se mutuamente ambos elementos:

```
 arranxo                      i    j    minx    minj
------------------------------ --   --    ----    ----
 3 9 4 1 5 9 2 6 5 3          1    0       3       1
 3 9 4 1 5 9 2 6 5 3          1    4       1       4
 1 9 4 3 5 9 2 6 5 3          1   10       1       4
 1 9 4 3 5 9 2 6 5 3          2   10       9       2
 1 9 4 3 5 9 2 6 5 3          2    3       4       3
 1 9 4 3 5 9 2 6 5 3          2    4       3       4
 1 9 4 3 5 9 2 6 5 3          2    7       2       7
 1 2 4 3 5 9 9 6 5 3          2   10       2       7
 1 2 4 3 5 9 9 6 5 3          3   10       4       3
 1 2 4 3 5 9 9 6 5 3          3    4       3       4
 1 2 3 4 5 9 9 6 5 3          3   10       3       4
 1 2 3 4 5 9 9 6 5 3          4   10       4       4
 1 2 3 4 5 9 9 6 5 3          4   10       3      10
 1 2 3 3 5 9 9 6 5 4          4   10       3      10
 1 2 3 3 5 9 9 6 5 4          5   10       5       5
 1 2 3 3 5 9 9 6 5 4          5   10       4      10
 1 2 3 3 4 9 9 6 5 5          5   10       4      10
 1 2 3 3 4 9 9 6 5 5          6   10       9       6
 1 2 3 3 4 9 9 6 5 5          6    8       6       8
 1 2 3 3 4 9 9 6 5 5          6    9       5       9
 1 2 3 3 4 5 9 6 9 5          6   10       5       9
 1 2 3 3 4 5 9 6 9 5          7   10       9       7
 1 2 3 3 4 5 9 6 9 5          7    8       6       8
 1 2 3 3 4 5 9 6 9 5          7   10       5      10
 1 2 3 3 4 5 5 6 9 9          7   10       5      10
 1 2 3 3 4 5 5 6 9 9          8   10       6       8
 1 2 3 3 4 5 5 6 9 9          8   10       6       8
```

```
 1 2 3 3 4 5 5 6 9 9            9   10        9        9
 1 2 3 3 4 5 5 6 9 9            9   10        9        9
```

(O anterior é o contido do arquivo selecc.dat producido polo seguinte código compilado baixo Turbo-Pascal 6.0 de Borland:)

```
program traza_seleccion ;

const
    card = 10 ;

type
   arranxo = array [ 1 .. card ] of integer ;

var
  T : arranxo ;
  saída : text ;

procedure amosa ( T : arranxo ; i , j , minx , minj : integer ) ;
var
  ind : integer ;
begin
    for ind := 1 to card do begin
       write ( saída ,  ' ' , T [ ind ] )
    end
    ; writeln ( saída , '        ' , i : 3
           , ' ' , j : 3 , '    ' , minx : 3 , '    ' , minj : 3 )
end ;

procedure ord_selecc ( var T : arranxo ) ;
var
  i , j : integer ;
  minj , minx : integer ;
begin
    j := 0 ;
    for i := 1 to card -1 do begin
                    minj := i ;
                    minx := T [ i ] ;
      amosa ( T , i , j , minx , minj ) ;
           for j := i+1 to card do begin
                              if T [ j ] < minx then begin
                                        minj := j ;
                                        minx := T [ j ] ;
             amosa ( T , i , j , minx , minj ) ;
           end ;
                    end ;
      T [ minj ] := T [ i ] ;
           T [ i ] := minx ;
      amosa ( T , i , j , minx , minj ) ;
       end ;
end ;

procedure inicializa ( var T : arranxo ) ;
begin
    { T = [ 3 9 4 1 5 9 2 6 5 3 ] }
    T[1] := 3 ; T[2] := 9 ; T[3] := 4 ; T[4] := 1 ; T[5] := 5 ;
    T[6] := 9 ; T[7] := 2 ; T[8] := 6 ; T[9] := 5 ; T[10] := 3 ;
end ;

procedure cabeceira ;
begin
    writeln ( saída ,
    ' arranxo                    i    j    minx    minj' ) ;
    writeln ( saída ,
    '------------------------------ --   --    ----    ----' )
end ;

begin
    assign ( saída , 'selecc.dat' )
    ; rewrite ( saída )
    ; inicializa ( T )
```

```
    ; cabeceira
    ; ord_selecc ( T )
    ; close ( saída )
end .
```

Observamos o comportamento do algoritmo de ordenación por selección nos casos de arranxos ordenados crescente e decrescentemente:

arranxo	i	j	minx	minj
1 2 3 4 5 6	1	0	1	1
1 2 3 4 5 6	1	6	1	1
1 2 3 4 5 6	2	6	2	2
1 2 3 4 5 6	2	6	2	2
1 2 3 4 5 6	3	6	3	3
1 2 3 4 5 6	3	6	3	3
1 2 3 4 5 6	4	6	4	4
1 2 3 4 5 6	4	6	4	4
1 2 3 4 5 6	5	6	5	5
1 2 3 4 5 6	5	6	5	5

arranxo	i	j	minx	minj
6 5 4 3 2 1	1	0	6	1
6 5 4 3 2 1	1	2	5	2
6 5 4 3 2 1	1	3	4	3
6 5 4 3 2 1	1	4	3	4
6 5 4 3 2 1	1	5	2	5
6 5 4 3 2 1	1	6	1	6
1 5 4 3 2 6	1	6	1	6
1 5 4 3 2 6	2	6	5	2
1 5 4 3 2 6	2	3	4	3
1 5 4 3 2 6	2	4	3	4
1 5 4 3 2 6	2	5	2	5
1 2 4 3 5 6	2	6	2	5
1 2 4 3 5 6	3	6	4	3
1 2 4 3 5 6	3	4	3	4
1 2 3 4 5 6	3	6	3	4
1 2 3 4 5 6	4	6	4	4
1 2 3 4 5 6	4	6	4	4
1 2 3 4 5 6	5	6	5	5
1 2 3 4 5 6	5	6	5	5

Desenrolamos o seguinte programa baixo TurboPascal 6.0 de Borland para comparar as ordenacións por inserción e por selección:

```
program compara_ordenacions ;

uses
   dos , crt ;

const
    card = 15000 ;

type
   dado = integer ;
   arranxo = array [ 1 .. card ] of dado ;
   tipo_modo = ( descendente , ascendente , desordenado ) ;

var
  T : arranxo ;
  F : file of arranxo ;
  h1 , m1 , s1 , c1 ,
  h0 , m0 , s0 , c0 ,
```

```
  dh , dm , ds , dc : word ;
  modo : tipo_modo ;

{--------------------------------------------------------------------------}
procedure quicksort_descendente ( var a : arranxo ; Lo , Hi : integer ) ;

procedure sort ( l , r : integer ) ;
var
 i , j : integer ;
 x , e : dado ;
begin
 i := l ; j := r ; x := a [ ( l + r ) div 2 ] ;
 repeat
   while a [ i ] > x do i := i + 1 ;
   while x > a [ j ] do j := j - 1 ;
   if i <= j then
   begin
     e := a [ i ] ; a [ i ] := a [ j ] ; a [ j ] := e ;
     i := i + 1 ; j := j - 1 ;
   end ;
 until i > j ;
 if l < j then sort ( l , j ) ;
 if i < r then sort ( i , r ) ;
end ;

begin { quicksort_descendente } ;
 sort ( Lo , Hi ) ;
end ;

{--------------------------------------------------------------------------}
procedure quicksort_ascendente ( var a : arranxo ; Lo , Hi : integer ) ;

procedure sort ( l , r : integer ) ;
var
 i , j : integer ;
 x , e : dado ;
begin
 i := l ; j := r ; x := a [ ( l + r ) div 2 ] ;
 repeat
   while a [ i ] < x do i := i + 1 ;
   while x < a [ j ] do j := j - 1 ;
   if i <= j then
   begin
     e := a [ i ] ; a [ i ] := a [ j ] ; a [ j ] := e ;
     i := i + 1 ; j := j - 1 ;
   end ;
 until i > j ;
 if l < j then sort ( l , j ) ;
 if i < r then sort ( i , r ) ;
end ;

begin { quicksort_ascendente } ;
 sort ( Lo , Hi ) ;
end ;

{--------------------------------------------------------------------------}
procedure por_insercion ( var T : arranxo ) ;
var
  i , j , k : integer ;
  aux : integer ;
begin
    k := 0 ;
    for i := 1 to card do begin
       j := 1 ;
       while T [ j ] < T [ i ] do
            j := j + 1 ;
       aux := T [ i ] ;
       for k := i downto j + 1 do
          T [ k ] := T [ k-1 ] ;
       T [ j ] := aux ;
       { monitorizacion } write ( i ) ; gotoxy ( 1 , wherey ) ;
```

```
    end
end ;

{-------------------------------------------------------------------------}
procedure por_seleccion ( var T : arranxo ) ;
var
  i , j : integer ;
  minj , minx : integer ;
begin
    j := 0 ;
    for i := 1 to card -1 do begin
                    minj := i ;
                    minx := T [ i ] ;
              for j := i+1 to card do begin
                              if T [ j ] < minx then begin
                                          minj := j ;
                                          minx := T [ j ] ;
                    end ;
                     end ;
             T [ minj ] := T [ i ] ;
              T [ i ] := minx ;
             { monitorizacion } write ( i ) ; gotoxy ( 1 , wherey ) ;
       end ;
end ;

{-------------------------------------------------------------------------}
procedure inicializa ( var T : arranxo ; modo : tipo_modo ) ;
var
  i : integer ;
  aux : integer ;
begin
    for i := 1 to card do
       T [ i ] := random ( 1000 ) ;
    case modo of
        ascendente : begin
                        writeln ( 'Arranxo inicialmente ordenado ' ,
                        'ascendentemente.' ) ;
                        quicksort_ascendente ( T , 1 , card ) ;
                    end ;
        descendente : begin
                         writeln ( 'Arranxo inicialmente ordenado ' ,
                         'descendentemente.' ) ;
                         quicksort_descendente ( T , 1 , card ) ;
                     end ;
        desordenado : writeln ( 'Arranxo inicialmente desordenado.' ) ;
    end ;
end ;

{-------------------------------------------------------------------------}
procedure diferenza_tempos
( h1 , m1 , s1 , c1 , h0 , m0 , s0 , c0 : word ;
var dh , dm , ds , dc : word ) ;
begin
    if c1 < c0
    then begin
        c1 := 100 + c1 ;
        s0 := s0 + 1 ;
    end ;
    if s1 < s0
    then begin
        s1 := s1 + 60 ;
        m0 := m0 + 1 ;
    end ;
    if m1 < m0
    then begin
        m1 := m1 + 60 ;
        h0 := h0 + 1 ;
    end ;
    if h1 < h0
    then begin
        h1 := h1 + 24 ;
```

```
    end ;
    dh := h1 - h0 ;
    dm := m1 - m0 ;
    ds := s1 - s0 ;
    dc := c1 - c0 ;
end ;

{=============================================================================}
begin
  clrscr ;
  randomize ;

  for modo := descendente to desordenado do begin

    inicializa ( T , modo ) ;

    assign ( F , 'borrame.dat' ) ;
    rewrite ( F ) ;
    write ( F , T ) ;
    close ( F ) ;

    writeln ( '>> Ordenacion por insercion: ' ) ;
    gettime ( h0 , m0 , s0 , c0 ) ;
    {********************}
    por_insercion ( T ) ;
    {********************}
    gettime ( h1 , m1 , s1 , c1 ) ;
    diferenza_tempos ( h1 , m1 , s1 , c1 ,
                       h0 , m0 , s0 , c0 ,
                       dh , dm , ds , dc ) ;
    writeln ( dh , ':' , dm , ':' , ds , ':' , dc ) ;

    reset ( F ) ;
    read ( F , T ) ;
    close ( F ) ;

    writeln ( '>> Ordenacion por seleccion: ' ) ;
    gettime ( h0 , m0 , s0 , c0 ) ;
    {********************}
    por_seleccion ( T ) ;
    {********************}
    gettime ( h1 , m1 , s1 , c1 ) ;
    diferenza_tempos ( h1 , m1 , s1 , c1 ,
                       h0 , m0 , s0 , c0 ,
                       dh , dm , ds , dc ) ;
    writeln ( dh , ':' , dm , ':' , ds , ':' , dc ) ;

  end ;

  erase ( F ) ;

end .
```

Exemplo posto na clase: Para n=5000 por inserción de vector ascendente leva-lle 20 centésimas. Para n=5000 por inserción de vector descendente leva-lle 3.5 minutos.

Observa-se que tanto para a ordenación por selección como para a ordenación por inserción o pior caso é o de ter o arranxo orixinariamente ordenado decrescentemente.

Como se observa en cada un dos cuadros da coluna da direita na tábua, o tempo de execución da ordenación por selección non depende muito do estado orixinal do vector porque a comparación `T [ j ] < minx` (bucle interno) executa-se o mesmo número de veces independentemente de como sexa o vector. A fluctuación de tempos de execución entre o pior e o mellor dos casos é do ±15%.

Exactamente o contrário acontece coa ordenación por inserción. A ordenación por inserción de un vector ordenado ascendentemente é mui rápida porque a comparación do while será sempre falsa o que fai que o bucle interno sexa asimilábel a unha operación sinxela como é unha comparación. Como mais adiante veremos, isto significa que no mellor dos casos estaremos nunha orden O(n).

En câmbio na ordenación por selección de un vector ordenado descendentemente, o bucle interno executa-se i-1 veces para todo i. Dicimos que o bucle interno se executa n veces e posto que o bucle exterior se executa n veces (é dicer, a orden de n-1 veces), a orden en conxunto é $O(n^2)$.

No pior dos casos ambos algoritmos son $O(n^2)$, pero ao dicer isto estamos pasando por alto que a inserción é mais rápida que a selección.

Isto deixa ao descoberto o inconveniente de tomar en consideración o pior dos casos, xá que se consideran iguais a dous casos que estrictamente non o son. Sen embargo pensar no pior dos casos ofrece unha certa garantia, unha fiabilidade, como para os sistemas de tempo real.

Outra razón para usar o método do pior caso posíbel é que se quixésemos usar o método do tempo médio, para cada vector de entrada de n elementos haberia n! disposicións posíbeis dos seus elementos. Haberia que calcular o tempo de proceso de cada unha de esas disposicións porque son todas equiprobábeis, e despois calcular o tempo médio. Ainda se o fixésemos asi, chegaríamos à conclusión de que o tempo médio para a inserción é da orden $O(n^2)$, porque non estariamos vendo a manifesta melloria que para n mui grandes presenta a inserción respeito da selección. Ou sexa, tanto traballo para chegar à mesma conclusión que co método do pior caso posíbel.

En canto ao algoritmo mais rápido de ordenación, para o pior dos casos con n tamaño do vector ten-se tamén que é $O(n^2)$. Sen embargo, o tempo promédio para n é O(n log n) o cal representa unha evidente melloria.

1.3.2.- Cálculo de O (o grande).

O cálculo de O (o grande) implica a simplificación de cálculos e permitirá desbotar algoritmos ineficientes, en comparación con outros, de xeito rápido.

O cálculo de O de paso darános unha garantia de terminación do algoritmo, o cal tamén é importante, posto que estamos calculando unha cota superior do mesmo.

Exemplo: algoritmo que calcula o sumatório de cubos, desde o do un consecutivamente até o de certo natural.

$$\sum_{i=1}^{n} i^3$$

```
función sumatório ( n )
{1}     suma_parcial ← 0 ;
{2}     para i ← 1 até n facer
{3}          suma_parcial ← suma_parcial + i*i*i
        finpara
{4}     devolver suma_parcial
finfunción
```

Numeradas con guarismos entre chaves aparecen as sentenzas con afectación ao tempo do bucle. Para cada unha de elas vemos agora o tempo de execución:

{1} 1 unidade (é unha simples asignación)
{4} 1 unidade (é unha simples devolución)
{3} 1 produto + 1 produto + 1 suma + 1 asignación
= 4 unidades x (as veces que se repiten, n)
= 4n
{2} n asignacións i ← i+1 que conducen ao bucle
+ 1 asignacións i ← i+1 que sai do bucle
+ n comparacións que conducen ao bucle
+ 1 comparación que sai do bucle
= 2n+2

Da suma total temos que a orden de este algoritmo é O(6n+4)=O(n)

Regras para calcular O (o grande)

(0) se hai chamadas a subprogramas, hai que avaliá-las antes.

(1) Composición iterativa (bucle): $T \approx T_{i.i.}*n$
O tempo de execución do bucle é aproximábel ao tempo de execución das instruccións internas multiplicado polo número de iteracións. Acerca das instruccións internas, é importante sinalar que non sempre é desbotábel o $T_{i.i.}$ como neste último exemplo, que nos daba unha constante.

(2) Bucles aniñados: A técnica consiste en ir desde dentro cara fóra. Ao final o tempo total do bucle aniñado aproxima-se a $T \approx T_{i+i}*$(prod.tam.ciclos)
O tempo do bucle aniñado é igual ao tempo das instruccións mais internas de todas multiplicado polo produto do tamaño dos ciclos.

Un exemplo son os algoritmos de ordenación que viamos anteriormente.

Outro exemplo:

```
procedimento pepe
   para i ← 1 até n facer
        para j ← 1 até n facer
             k ← k +1
        finpara
   finpara
finprocedimento
```

É $O(n^2)$. Antes viamos que non era necesário que fose até n, senón que pode-se ir até n (escoller sempre o pior dos casos).

(3) Composición secuencial: Suman-se os tempos de execución pero en realidade o que acontece é que é o máximo o que conta.

Exemplo:

```
procedimento pepe2
   para i ← 1 até n facer
        a [ i ] ← 0
   finpara ;
   pepe
finprocedimento
```

Temos que o bucle "para" é O(n) e que "pepe" é $O(n^2)$, entón o tempo do algoritmo conxunto pepe2 é $O(n+n^2) = O(n^2)$

(4) Composición alternativa (if-then-else):

$T_{si} \approx T_{condición} + máximo\ (\ T_{entón}\ ,\ T_{senón}\ )$

(5) En canto à recursividade, hai que tratá-la como un ciclo sempre que sexa posíbel. Hai que resolver unha relación de recorrência.

- Exemplo de recorrência: análise do algoritmo fibo1.

```
función fibo1 ( n )
   se n < 2
   entón devolver 2
   senón devolver fibo1 ( n -1 ) + fibo1 ( n -2 )
   finse
finfunción
```

T(n) é o tempo de execución de fibo1 (n).
T(0) = T(1) = O(1) xá que hai unha comparación e unha devolución o cal suma unha constante e a orden de unha constante é 1.
Para n > 2, T(n) = T(n-1)+T(n-2)+3 o "3" é porque hai unha comparación, unha suma e unha devolución.

É sinxelo demostrar por indución que $T(n) \geq fib(n)$, e xá demostramos que $fib(n) < (5/3)^n$ tamén por indución. Igualmente demóstra-se que $fib(n) \geq (3/2)^n$ entón sabemos que T(n) medra exponencialmente.

- Exemplo: Suma de subsecuência máxima: "dados n inteiros a_1, a_2, ..., a_n, atopar $\sum(k=i,j)\ a_k$ que sexa máximo". Subsecuência quer dicer que os números estexan consecutivos.

Por exemplo: [-2,11,-4,13,-5,-2] a subsecuência máxima é 11,-4,13 que suma 20.

Imos ver catro algoritmos que dan a subsecuência máxima.

Algoritmo 1: $O(n^3)$

```
función SumaSubMax1 ( a [ 1 .. n ] )
{1}  SumaMax ← 0 ; MellorI ← 0 ; MellorJ ← 0 ;
{2}  para i ← 1 até n facer
{3}      para j ← i até n facer
{4}           EstaSuma ← 0 ;
{5}           para k ← i até j facer
{6}                EstaSuma ← EstaSuma + a [ k ]
              finpara ;
{7}           se EstaSuma > SumaMax entón
{8}                SumaMax ← EstaSuma ;
{9}                MellorI ← i ;
{10}               MellorJ ← j
              finse
         finpara
     finpara ;
{11} devolver SumaMax
finfunción
```

Análise do algoritmo 1:

{2} i : tamaño n
{3} j : tamaño n - i + 1 , n no pior caso
{5} k : tamaño j - i + 1 , n no pior caso
{6} é o mais interno de três bucles de orden n : $O(n^3)$
{7} a {10} está dentro de dos bucles de orden n : $O(n^2)$
e como prima a orden maior, por {6}, o algoritmo é $O(n^3)$

{4} 1 {5} [j-i+1, n pior caso] para ≡ 2n+2 {6} n·(1+1) {7} 1 {8}{9}{10} 3
{3} [n-i+1, n pior caso]·n +2n+2 ≡ para
{2} ·n +2n+2
{11} 1 {1} 3

$O(((4n+7)\cdot n+2n+2)\cdot n+2n+2+1+3)=O(4n^3+9n^2+4n+6)=O(n^3)$

Algoritmo 2: elimina un bucle for ⇒ tempo de execución $O(n^2)$

```
función SumaSubMax2 ( a [ 1 .. n ] )
{1}      SumaMax ← 0 ; MellorI ← 0 ; MellorJ ← 0 ;
{2}      para i ← 1 até n facer
{3}          EstaSuma ← 0 ;
{4}          para j ← i até n facer
{5}               EstaSuma ← EstaSuma + a [ j ] ;
{6}               se EstaSuma > SumaMax entón
{7}                    SumaMax ← EstaSuma ;
{8}                    MellorI ← i ;
{9}                    MellorJ ← j
                  finse
             finpara
         finpara ;
{10}     devolver SumaMax
finfunción
```

Algoritmo 3: O(n log n)

```
función SumaSubMax3 ( a [ 1 .. n ] )
   devolver SumaSubMax ( a , 1 , n )
finfunción

función SumaSubMax ( a [ 1 .. n ] , esq , dir )
{1}      se esq = dir entón
{2}          se a [ esq ] > 0 entón
{3}               devolver a [ esq ]
             senón
```

```
{4}              devolver 0
                 finse
           senón
{5}              Centro ← ( esq + dir ) div 2 ;
{6}              SumaMaxEsq ← SumaSubMax ( a , esq , Centro ) ;
{7}              SumaMaxDir ← SumaSubMax ( a , Centro + 1 , dir ) ;

{8}              SumaMaxEsqBeira ← 0 ; SumaBeiraEsq ← 0 ;
{9}              para i ← Centro até esq paso -1 facer
{10}                  SumaBeiraEsq ← SumaBeiraEsq + a [ i ] ;
{11}                  se SumaBeiraEsq > SumaMaxEsqBeira entón
{12}                       SumaMaxEsqBeira ← SumaBeiraEsq
                      finse
                 finpara

{13}        SumaMaxDirBeira ← 0 ; SumaBeiraDir ← 0 ;
{14}        para i ← Centro + 1 até dir facer
{15}             SumaBeiraDir ← SumaBeiraDir + a [ i ] ;
{16}             se SumaBeiraDir > SumaMaxDirBeira entón
{17}                  SumaMaxDirBeira ← SumaBeiraDir
                 finse
           finpara
{18}       devolver max ( SumaMaxEsq , SumaMaxDir ,
                          SumaMaxEsqBeira , SumaMaxDirBeira )
           finse
finfunción
```

Análise do algoritmo 3:

Neste algoritmo usa-se eficientemente o conceito de “divide e vencerás”. “Dividir” consiste en partir en duas metades o conxunto obxecto de tratamento, e resolver cada unha das duas metades mediante a recursividade, tendo unha solución para unha metade e outra para a outra metade. “Vencer” consiste en unir as duas solucións --facendo posìbelmente un traballo adicional-- para dar coa solución ao problema orixinal.

Se partimos en duas partes o vector no que buscamos a subsecuência máxima, ésta pode atopar-se totalmente na primeira metade, na segunda, ou ben sobre o límite que separa a ambas.

Exemplo: subsecuência máxima en [4 -3 5 -2|-1 2 6 -2]

```
                          ^----^       ^-^
                             6          8
                          ^--------------^
                                 11
```

Subsecuência máxima de metade direita incluíndo a beira esquerda: 7
Subsecuência máxima de metade esquerda incluíndo a beira direita: 4

No algoritmo 3 debe-se pasar o arranxo a tratar por referência para evitar criar cópias do orixinal. A función SumaSubMax3 funciona como a interface para a primeira chamada ao algoritmo SumaSubMax que é o que fai o traballo.

{1} a {4} dan resposta ao caso base: se só hai un elemento no vector que se pasa como parámetro, devolve-se o mesmo se é positivo, senón devolve-se 0.

T(n):

T(1) = O(1) é o caso base, de {1} a {4}
os ciclos {9} a {12} e {14} a {17} son n/2 para cada un no pior dos casos, co que se percorren n elementos. Como as operacións que se fan son sinxelas, son de orden 1, entón ten-se O(n).

{1} a {5} ⎫
{8} inicialización antes do ciclo ⎪ O(1)<O(n) entón
{13} inicialización antes do ciclo ⎬ ignora-se todo isto à
{18} devolución ⎭ esquerda da chave

{6} e {7} consomen un tempo de T(n/2) cada un. {chamadas recursivas}

Entón temos T(1)=1 e T(n)=2·T(n/2)+n

A "solución intuitiva" consiste en ir calculando distintos valores para n:

T(2)= 4= 2·2
T(4)= 12= 4·3
T(8)= 32= 8·4
T(16)= 80= 16·5
...
T(n)= n·(k+1) onde $n=2^k$ $\log_2 n = k$
= n·(log n +1)
= n·log n +n
= O(n·log n +n)
= O(n·log n)

A "solución rigurosa" parte de que T(1)=1 e T(n)=2·T(n/2)+n

1. Divide-se por n:
T(n)/n=(2/n)·T(n/2)+(n/n)
T(n)/n=T(n/2)/(n/2)+1
2. Para todo n potência de 2:
T(n/2)/(n/2)=T(n/4)/(n/4)+1
T(n/4)/(n/4)=T(n/8)/(n/8)+1
...
T(2)/2=T(1)/1+1

Co que temos definidas log n ecuacións (dividir entre 2):

T(n)/n=T(n/2)/(n/2)+1
T(n/2)/(n/2)=T(n/4)/(n/4)+1
T(n/4)/(n/4)=T(n/8)/(n/8)+1
...
T(2)/2=T(1)/1+1

3. Sumamos todas as ecuacións e tachamos dous a dous os termos que son iguais à esquerda e à direita do "=". Resulta:

$T(n)/n=T(1)/1+1\cdot(\log n) \Rightarrow T(n)=n+n\cdot(\log n)=O(n\cdot\log n)$
↑
T(1)=1

Supuxemos que n é par e que é potência de dous, o cal nos val para este exemplo.

Que acontece se no algoritmo 3 **non** se pasa o vector por referência, é dicer se se pasa sen "var"?

Xeran-se R(n) cópias do vector, onde R(n) é o número de chamadas recursivas. Para o caso base non hai chamada recursiva: R(1)=0.

R(n)=1 {6} {chamada recursiva para a primeira metade}
+1 {7} {chamada recursiva para a segunda metade}
+R(n/2) {chamadas recursivas que implica 6}
+R(n/2) {chamadas recursivas que implica 7}
=2·R(n/2)+2

Aplicamos un método similar ao anterior para calcular R(n) en función de n:
R(n)=2·R(n/2)+2
R(n/2)=2·R(n/4)+2
R(n/4)=2·R(n/8)+2
...
R(8)=2·R(4)+2
R(4)=2·R(2)+2
R(2)=2·R(1)+2
e sabemos que R(1)=0 co que:
R(2)=2
⇒ R(4)=2·2+2=6
⇒ R(8)=2·6+2=14
⇒ R(16)=2·14+2=30
⇒ ...
⇒ R(n)=2·(n-2)+2=2n-4+2
⇒ R(n)=2n-2 ⇒ $T(n)=O(n^2)$

É dicer que ao non pôr a palabra "var" está-se convertendo o algoritmo 3 nun método tan ineficiente como o algoritmo 2.

Algoritmo 4: O(n)

```
función SumaSubMax4 ( a [ 1 .. n ] )
   i ← 1 ;
   EstaSuma ← 0 ;
   SumaMax ← 0 ;
   MellorI ← i ;
   MellorJ ← j ; {0}
   para j ← 1 até n facer
        EstaSuma ← EstaSuma + a [ j ] ;
        se EstaSuma > SumaMax entón
             SumaMax ← EstaSuma ;
             MellorI ← i ;
             MellorJ ← j ;
        senón
             se EstaSuma < 0 entón
                  i ← j + 1 ;
                  EstaSuma ← 0
        finse {1}
   finpara ;
   devolver SumaMax
finfunción
```

Este algoritmo é de orden O(n) porque dentro do bucle "para" fai-se un traballo constante.

A estes algoritmos con acceso secuencial unha soa vez aos elementos do vector chama-se-lles algoritmos en liña.

En todo momento o algoritmo pode dar unha solución parcial relativa ao subarranxo que xá leva procesado, o cal non acontecia cos três algoritmos vistos previamente.

Un algoritmo en liña que ocupa un espazo constante e que resolve o problema nun tempo lineal é a solución óptima.

Exemplo: [4 -3 5 -2 -1 2 6 -2]

	i	j	EstaSuma	SumaMax	MellorI	MellorJ
{0}	1		0	0	0	0
{1}		1	4	4	1	1
{1}		2	1			
{1}		3	6	6	1	3
{1}		4	4			
{1}		5	3			
{1}		6	5			
{1}		7	11	11	1	7
{1}		8	9			

Algoritmos no tempo de execución:

Algúns algoritmos de “divide e vencerás” executan-se en O(n·log n)

Un algoritmo é O(log n) se usa un tempo constante en dividir o problema en partes, normalmente en metades. Neste caso supomos que xá se leu a entrada, para ler unha entrada de tamaño n necesitamos O(n).

Exemplo: algoritmo de busca binária. Dados un inteiro x e n inteiros a_1, a_2, ..., a_n, ordenados e en memória, atopar i tal que $x=a_i$ ou devolver 0 se non se atopa.

Empezamos mirando no elemento que ocupa a posición central do algoritmo (a mediana). Se é x xá rematamos. Se é maior que x, miramos na metade esquerda, se é menor que x miramos na metade direita.

```
función busca_binária ( a [ 1 .. n ] )
{
utilizamos un centinela, situamos a x o elemento buscado en a[0] sendo o
conxunto ordenado onde buscar x o de índices 1 a n
}
   a [ 0 ] ← x ;
   baixo ← 1 ;
   alto ← n ;
   repetir
        metade ← ( baixo + alto ) div 2 ;
        se baixo > alto { os dous índices cruzaron-se }
        entón
             metade ← 0
        senón
             se a [ metade ] < x
             entón
                  baixo ← metade + 1
             senón
                  se a [ metade ] > x
                  entón
                       alto ← metade - 1
                  finse
             finse
        finse
```

até a [metade] = x

Exemplo: [1 4 5 8] x=7

iteración	alto	baixo	metade
0	1	4	
1	3		2
2	4		3
3	4	3	4
4			0

Análise da eficiência: consiste en dar un conxunto de técnicas para estimar e comparar teòricamente os tempos de execución antes de implementar o algoritmo, isto ben en función do **tamaño** da entrada (exemplo 2: Selección) ou ben en función do **valor** da entrada (exemplo 1: Fibonacci). Tamén existen técnicas híbridas que combinan ambos.

O obxectivo da análise da eficiência é mellorar a velocidade dos programas, detectar colos de botella (partes do programa que son as causantes da lentitude do mesmo). Conta-se con que os colos de botella se producen nas operacións de entrada/saída; o que se quer evitar é que os colos de botella se produzan nos algoritmos.

Exemplo: algoritmo de Euclides, que permite calcular o máximo común divisor de m e n onde m e n son naturais con $m \geq n$

```
función maximo_común_divisor ( m , n )
      mentres n > 0 facer
            resto ← m mod n  ⎫
            m ← n            ⎬  O(1)
            n ← resto        ⎭
      finmentres
función
```

iteración	m	n	resto
0	1989	1590	
1	1590	399	399
2	399	393	393
3	393	6	6
4	6	3	3
5	3	0	0

Despois de duas iteracións o resto é como máximo a metade do valor orixinal: $resto \leq metade$

número de iteracións: 2 log n = O (log n)

1.3.3.- Verificación empírica da análise:

A ideia é ver que acontece ao duplicar n:

T*2 en O(n)
 *4 en $O(n^2)$
 *8 en $O(n^3)$
T+k en O(log n)
T*2+k en $O(n \cdot \log n)$

(k = cte) Poden aparecer problemas cando n non é grande abondo, ou cando aparecen coeficientes grandes en termos de orden menor.

T(n) tempo medido, F(n) tempo teórico.

Para verificar que un programa é O(F(n)): Sexa T(n) o tempo de execución para n; calculamos T(n)/F(n) e estudamos a converxência:

>0: verifica-se F(n).

=0: F(n) está sobreestimada (podemos baixar un poco a cota superior).

diverxen: F(n) está subestimada (hai que tomar unha cota superior).

<fin do tema 1>