Anàlisi de l'eficiència dels algorismes iteratius i recursius

Toni Sellarès
Dep. Informàtica i Matemàtica Aplicada
Universitat de Girona

1 Introducció

Calcular l'*eficiència*, *cost* o *complexitat* d'un algorisme consisteix a mesurar la quantitat de recursos necessaris per a la seva execució. Entre dos algorisme que resolen el mateix problema és més eficient aquell que a l'executar-se (sobre les mateixes dades) utilitza menys recursos.

2 Anàlisi d'algorismes

Com a mesura de l'eficiència d'un algorisme se solen elegir factors relacionats amb la quantitat real de recursos que s'utilizten durant la seva execució, fonamentalment el temps i la quantitat de memòria. Nosaltres ens centrarem en la eficiència en temps.

Per a mesurar el temps d'execució d'un algorisme no podem programar-lo i executar-lo en un computador, i després utilitzar les unitats de mesura del temps habituals (microsegons, segons, ...) ja que el resultat de la mesura dependria fortament del computador i el compilador que s'han fet servir.

Per a obtenir l'eficiència en temps d'un algorisme ens basarem en l'*anàlisi* de l'algorisme, és a dir, en la consideració del temps d'execució una per una de totes les instruccions que conformen l'algorisme. Per això és necessari definir un *model de computació*. Nosaltres adoptarem el model teòric anomenat *random access machine* (RAM).

Una RAM representa un computador amb un processador que pot emmagatzemar en memòria nombres reals de precisió arbitrària i que és capaç d'executar les següents *operacions simples* en una unitat de temps:

- operations aritmètiques: $+, -, *, /, \sqrt{}, \log, \exp, \sin$;
- comparacions: $<, <=, >, >=, <>$;
- lectura o escriptura de qualsevol registre de memòria quan es coneix la seva adreça.

El temps d'execució d'un algorisme depèn de diversos factors, però principalment dels tres següents:

1. l'*implementació*: computador i compilador utilitzats per executar l'algorisme.

2. la *grandària* n de les dades: nombre de bits necessaris per a emmagatzemar un natural, nombre d'elements d'un vector, nombre de segments d'un conjunt.

3. el *contingut* de les dades d'entrada: vector a ordenar ja ordenat, segments a intersecar verticals.

Els factors de tipus 1 i 2 influeixen en el temps d'execució d'un algorisme de forma substancialment diferent. Els factors relacionats amb la implementació fan que el temps d'execució varii com a màxim en un factor constant: un computador pot ser cinc vegades més ràpid que un altre o un compilador generar codi tres vegades més lent. En canvi l'increment en una certa quantitat de la grandària de les dades d'entrada d'un algorisme pot originar que el temps d'execució de l'algorisme s'incrementi en una quantitat molt més gran: el temps d'execució d'alguns algorismes d'ordenació pot quedar elevat al quadrat al duplicar el nombre de dades a ordenar.

L'únic factor que condiderarem important, en la majoria de casos, per a mesurar l'eficiència d'un algorisme serà el factor 2: la grandària n de les dades. Formalment la grandària de les dades és el nombre de bits que es necessiten per a representar les dades d'entrada. No obstant, per a fer més clars els anàlisis, normalment farem servir qualsevol nombre natural positiu que mesuri la magnitud de les dades d'entrada. En general serà fàcil determinar quin valor caldrà prendre com a grandària. Quan l'entrada sigui un vector serà la seva longitud; si és una matriu quadrada serà la seva dimensió al quadrat; si és un conjunt de segments serà el nombre total de segments; etc. En el cas d'un algorisme que té per entrada un nombre natural o enter N, depenent del problema, podrem agafar com a grandària el propi valor de N o la longitud $\log_2 N$ de N. Cal especificar clarament quina de les dues possibilitats s'elegeix perquè es diferencien en una quantitat exponencial.

3 Càlcul exacte del temps d'execució

Definirem el temps d'execució $t(n)$ d'un algorisme sobre unes dades de grandària n com el total d'instruccions simples que ha d'executar.

Com a exemple estudiarem el temps d'execució d'una funció Suma1 que calcula la suma dels primers N nombres naturals positius:

```
1.  funcio Suma1 (N: nat) retorna s: nat
2.  {P ≡ N > 0}
3   var i: nat fvar
4.  s := 0; i := 1;
5.  mentre i ≤ N fer
6.          s := s + i;
7.          i := i + 1
8.  fmentre
9.  {Q ≡ s = Σ α ∈ {1..N} · α}
10. retorna s
11. ffuncio
```

Si com a grandària de les dades prendrem $n = N$, el temps d'execució $t_1(n)$ de la funció Suma1 serà:

$$t_1(n) = \underbrace{2}_{(4)} + \underbrace{(n+1)}_{(5)} + \underbrace{n(2+2)}_{(6-7)} + \underbrace{1}_{(10)} = 5n + 4$$

El temps necessari per a calcular la suma dels primers N nombres naturals positius es pot millorar si es fa servir una funció Suma2 que tingui en compte l'expressió de la suma dels elements d'una progressió aritmètica:

1. **funcio** Suma2 (N: nat) retorna s: nat
2. $\{P \equiv N > 0\}$
3. $s := (1 + N) * N/2$
4. $\{Q \equiv s = \Sigma\alpha \in \{1..N\} \cdot \alpha\}$
5. **retorna** s
6. **ffuncio**

Si tornem a agafar $n = N$ com a grandària de les dades, el temps d'execució $t_2(n)$ de la funció Suma2 serà:

$$t_2(n) = \underbrace{4}_{(3)} + \underbrace{1}_{(5)} = 5$$

4 Ordres de complexitat

Per a evitar que el temps d'execució d'un algorisme depengui de factors del tipus 1 (implementació), acceptarem l'anomenat *principi d'invariància* que afirma que els temps d'execució de dues implementacions diferents d'un mateix algorisme com a màxim difereixen en una constant multiplicativa positiva.

La dependència del temps d'execució $t(n)$ d'un algorisme respecte del factor 3 (contingut de les dades) l'eliminarem considerant el cost en el *cas pitjor*: calcularem el temps màxim necessari per a executar l'algorisme sobre qualsevol possible conjunt de dades de grandària n.

Ens interessarem en el *cost asimptòtic* dels temps d'execució $t(n)$ d'un algorisme, és a dir en el càlcul del temps d'execució en el cas que la gràndaria n de les dades creix ilimitadament.

L'acceptació del principi d'invariància i l'interès per l'estudi del cost asimptòtic ens portaran de forma natural a l'introducció del concepte d'*ordre de complexitat* del temps d'execució $t(n)$ d'un algorisme: només tindrem en compte la rapidesa de creixement del temps d'execució $t(n)$ amb la grandària n de les dades d'entrada.

5 La notació O majúscula

Sigui $f(n)$ una funció definida sobre els nombres naturals positius. Direm que el temps $t(n)$ d'execució (en el cas pitjor) d'un algorisme és inferior o igual que l'ordre

de complexitat de $f(n)$, i ho indicarem $t(n) \in O(f(n))$, si a partir d'una certa grandària de les dades la funció $f(n)$ és una fita superior, llevat d'un factor constant positiu, del temps d'execució $t(n)$. Formalment, si existeixen $n_0 \in N$ i $c \in R, c > 0$ de manera que per a tot $n \geq n_0$ és $t(n) \leq c \cdot f(n)$.

Així doncs, $O(f(n))$ és el conjunt format per totes les funcions que "al crèixer n creixen com a màxim tan ràpidament com $f(n)$".

Exemples:

a) $5 \in O(1)$.

b) $5n + 4 \in O(n)$.

c) $5n + 4 \in O(n^2)$.

Quan $t(n) \in O(f(n))$ llavors $t(n)$ té ordre de complexitat menor o igual que $O(f(n))$ i $O(t(n)) \subseteq O(f(n)$. Si $t(n) \in O(f(n))$ però $f(n) \notin O(t(n))$, llavors $t(n)$ té ordre de complexitat menor que $O(f(n))$ i $O(t(n)) \subset O(f(n)$. Quan $t(n) \in O(f(n))$ i $f(n) \in O(t(n))$, llavors $t(n)$ i $f(n)$ tenen el mateix ordre de complexitat i per tant $O(t(n)) = O(f(n))$. En el darrer cas direm que $t(n)$ és d'ordre *exacte* o *ajustat* $O(f(n))$.

Exemples:

a) $O(5n + 4) \subseteq O(n)$.

b) $O(5n + 4) = O(n)$ (l'ordre exacte de $5n + 4$ és $O(n)$).

c) $O(5n + 4) \subset O(n^2)$.

Observis que quan es treballa amb ordres de complexitat:

- només es té en compte el terme de $t(n)$ que creix amb n d'una manera més ràpida;
- els factors constants no es tenen en condideració;
- pot ser que l'ordre no estigui ben ajustat: $5n + 4 \in O(n^2)$.

Convé observar que quan es treballa amb ordres de complexitat s'ha d'anar en compte amb el factor constant "amagat" en la notació. Suposem que comparem dos algorismes que resolen un mateix problema, un de baix ordre de complexitat però factor constant amagat alt i un altre d'ordre de complexitat pitjor però de constant amagada molt inferior. És possible que l'ordre de complexitat més alt del segon, comparat amb l'elevada constant amagada del primer, faci que el segon sigui realment més ineficient només a partir d'una grandària de les dades exageradament gran, de manera que a la pràctica sigui més eficient l'algorisme pitjor.

També hi ha notacions per expressar *fites inferiors* de la complexitat d'un problema (Ω) i notacions d'optimitat per als casos en els quals coincideixen la fita inferior d'un problema amb la fita superior en el cas pitjor d'un algorisme que resol el problema (Θ).

6 Càlcul pràctic de complexitats

A la pràctica, per a comparar l'ordre de complexitat de dues funcions es fa servir el *criteri del límit* que estableix que:

1. $$\lim_{n\to+\infty} \frac{t(n)}{f(n)} = 0 \Longrightarrow O(t(n)) \subset O(f(n)).$$

2. $$\lim_{n\to+\infty} \frac{t(n)}{f(n)} = k \neq 0 \Longrightarrow O(t(n)) = O(f(n)).$$

Exemples:

a) $$\lim_{n\to+\infty} \frac{n}{n\log n} = \lim_{n\to+\infty} \frac{1}{\log n} = 0 \Longrightarrow O(n) \subset O(n\log n).$$

b) Si $p < q \in N$, llavors

$$\lim_{n\to+\infty} \frac{n^p}{n^q} = 0 \Longrightarrow O(n^p) \subset O(n^q).$$

c) Si $a > 0$, de $\log_a n = \frac{\log n}{\log a}$ es dedueix

$$\lim_{n\to+\infty} \frac{\log n}{\log_a n} = \log a \Longrightarrow O(\log_a n) = O(\log n).$$

Fent servir el criteri del límit es demostra fàcilment la següent jerarquia (incompleta) d'ordres de complexitat:

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n\log n) \subset O(n^2) \subset O(n^2\log n) \subset$$
$$\subset O(n^3) \subset O(n^d) \subset O(2^n) \subset O(3^n) \subset O(d^n) \subset O(n!) \subset O(n^n)$$

Donarem a continuació una taula amb el nom que reben els ordres de complexitat més habituals:

Ordre	Nom
$O(1)$	contant
$O(\log n)$	logarítmic
$O(n)$	lineal
$O(n\log n)$	quasilineal
$O(n^2)$	quadràtic
$O(n^3)$	cúbic
$O(n^k)$	polinòmic ($k \geq 2$)
$O(k^n)$	exponencial ($k \geq 2$)

7 Problemes tractables i intractables

A la taula que presentem tot seguit es pot veure com creixen les funcions $f(n)$, que defineixen els ordres de complexitat més habituals, quan creix n. Per a fer la taula més entenedora suposarem que el temps d'execució d'una intrucció simple és d'un μs. Farem servir les següents abreviatures: μs = microsegon = una milionèsima de segon; ms = mil·lisegon = una mil·lèsima de segon; s= segon, min=minut, sgl=segle.

$f(n)$	$n = 2$	$n = 64$	$n = 1024$	$n = 65536$
1	1μs	1μs	1μs	1μs
$\log_2 n$	1μs	6μs	10μs	16μs
n	2μs	64μs	1.02 ms	65.5 ms
$n \log_2 n$	2μs	384μs	10.2 ms	1.05 s
n^2	4μs	4.1 ms	1.04 s	71.6 min
n^3	8μs	0.26 s	17.9 min	8.9 anys
2^n	4μs	5849 sgl	$2^{972.5}$ sgl	2^{65484} sgl

El temps d'execució dels algorismes de complexitat constant és independent de la grandària de les dades. Els algorismes de complexitat logarítmica tenen un comportament excepcional: doblar la grandària de les dades quasi no afecta el temps d'execució. Pels algorismes de complexitat lineal i quasilineal, doblar la grandària de les dades comporta doblar aproximadament el temps d'execució. Els algorismes de complexitat quadràtica i cúbica tenen un comportament clarament inferior als d'ordre lineal i quasilineal. No és possible tractar problemes amb grandària massa elevada amb algorismes de complexitat polinòmica.

Dels problemes que es poden resoldre amb algorismes d'ordre inferior o igual a la complexitat polinòmica en direm problemes *tractables*. Els problemes que no es poden resoldre amb un algorisme de complexitat inferior a la polinòmica direm que són *intractables*.

Les consideracions anteriors ens porten a la conclusió que és una bona inversió dedicar esforços a trobar algorismes amb bons ordres de complexitat. Contrariament, no és rentable invertir temps en estalviar una variable auxiliar o a millorar un cas particular perquè no canvien la complexitat d'un algorisme, sinó que tan sols disminueix lleugerament la constant multiplicativa.

8 Ordre de complexitat d'alguns sumatoris

A l'analitzar l'ordre de complexitat d'algorismes iterartius moltes vegades cal calcular sumatoris. A continuació donem l'ordre de complexitat dels sumatoris que apareixen amb més frequència. La k i la n dels sumatoris representaran, respectivament, una constant i una variable.

$$\sum_{i=1}^{k} 1 = k \in O(1)$$

$$\sum_{i=1}^{n} k = kn \in O(n)$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in O(n^2)$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1}{6} \in O(n^3)$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4} \in O(n^4)$$

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

$$\sum_{i=0}^{n} k^i = \frac{k^{n+1}-1}{k-1} \in O(k^n);\ k \neq 1$$

$$\sum_{i=1}^{n} ik^i \in O(k^n)$$

$$\sum_{i=1}^{n} ik^{n-i} \in O(k^n)$$

$\sum_{i=1}^{n} \log_2 i \in O(n \log n)$. D'on es dedueix que $\log_2 n! \in O(n \log n)$.

$$\sum_{i=1}^{n} i^k \log_2 i \in O(n^{k+1} \log n)$$

9 Operacions entre ordres de complexitat

Entre ordres de complexitat hi ha definides les operacions:

Suma: $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$;

Producte: $O(f(n)) * O(g(n)) = O(f(n) * g(n))$.

La suma de dos ordres de complexitat es generalitza a la suma d'un nombre fix k d'ordres de complexitat mitjançant:

$$O(f_1(n)) + \cdots + O(f_k(n)) = O(f_1(n) + \cdots + f_k(n)) = O(\max\{f_1(n), \cdots, f_k(n)\}).$$

No obstant, si el nombre d'ordres de complexitat $k(n)$ que s'està sumant varia amb la grandària n de les dades, llavors cal tenir en compte que el nombre de sumands del sumatori resultant també depèn de n, i per tant haurem de calcular l'ordre de complexitat d'aquest sumatori:

$$O(f_1(n)) + \cdots + O(f_{k(n)}(n)) = O\left(\sum_{i=1}^{k(n)} f_i(n)\right).$$

Exemples:

$$O(1) + \cdots + O(k) = O(\max\{1, \cdots, k\}) = O(1).$$

$$O(1) + \cdots + O(n) = O\left(\sum_{i=1}^{n} i\right) = O(n^2).$$

10 Càlcul pràctic de l'eficiència d'algorismes iteratius

Les declaracions de funcions, procediments, variables i tipus no consumeixen temps d'execució perquè només es tenen en compte quan l'algorisme es compila, però no quan s'executa.

Quan estudiem l'ordre de complexitat en el temps $O(t(n))$ d'un algorisme S per unes dades d'entrada de grandària n en el cas pitjor, per abús de notació, posarem $O(S) = O(t(n))$.

Tot seguit donarem l'ordre de complexitat de les instruccions bàsiques:

- Assignació simple:
$$O(x := E) = O(1)$$
- Condició simple:
$$O(B) = O(1)$$
- Retorn del valor d'una variable simple:
$$O(\textbf{retorna } x) = O(1)$$
- Sequencial:
$$O(S_1; S_2) = O(S_1) + O(S_2) = \max\{O(S_1), O(S_2)\}$$
- Condicional:

  $O(\textbf{cas } B_1 \to S_1 \cdots B_n \to S_n \textbf{ fcas}) = \max\{O(B_1), \cdots, O(B_n), O(S_1), \cdots, O(S_n)\}$
- Iteració:
$$O(\textbf{mentre } B \textbf{ fer } S \textbf{ fmentre}) = \sum_{i=1}^{It(n)} O(\max\{(B)_i, (S)_i\})$$

  on $It(n)$ és el total d'iteracions i $O(B)_i$ i $O(S)_i$ són, respectivament, la complexitat de B i S a la iteració nombre i.

  En el cas particular en que totes les iteracions tenen la mateixa complexitat $O(\max\{B, S\})$ obtindrem:

$$O(\textbf{mentre } B \textbf{ fer } S \textbf{ fmentre}) = It(n) * O(\max\{B, S\}).$$

11 Exemples de càlcul de l'eficiència

1) Retornem als exemples de les funcions Suma1 i Suma2 vistos a la Secció 3.

Per a la funció Suma1 tenim que $It(n) = n$, d'on:

$$O(Suma1) = O(Inic) + O(\textbf{mentre} \cdots \textbf{fmentre}) + O(\textbf{retorna}) =$$
$$= O(1) + n * O(1) + O(1) = O(\max\{1, n, 1\}) = O(n).$$

$$O(Suma2) = O(s := (1 + N) * N/2) = O(1).$$

2) Volem estudiar l'ordre de complexitat de la funció Pot1 que permet calcular la potència B^N:

funcio Pot1 (B: real, N: nat) retorna p: real
$\{P \equiv \textbf{cert}\}$
var i: nat **fvar**
$p := 1; i := 1$
mentre $i \leq N$ **fer**
    $p := p * B$
    $i := i + 1$
fmentre
$\{Q \equiv p = \Pi\, \alpha \in \{1..N\} \cdot B\}$
retorna p
ffuncio

Com a grandària de les dades prendrem $n = N$. El bucle fa $It(n) = n$ iteracions. D'on:

$$O(Pot1) = O(Inic) + O(\textbf{mentre} \cdots \textbf{fmentre}) + O(\textbf{retorna}) =$$
$$= O(1) + n * O(1) + O(1) = O(\max\{1, n, 1\}) = O(n).$$

La eficiència de la funció Pot1 pot millorar-se amb la funció:

funcio Pot2 (B: real, N: nat) retorna p: real
$\{P \equiv \textbf{cert}\}$
var b: real, i: nat **fvar**
$p := 1; b := B; i := N$
mentre $i > 0$ **fer**
    si N senar **llavors** p:=p*b **fsi**
    $b := b * b$
    $i := i$ div 2
fmentre
$\{Q \equiv p = \Pi\, \alpha \in \{1..N\} \cdot B\}$
retorna p
ffuncio

Tornarem a prendre $n = N$ com a grandària de les dades. La principal dificultat la trobarem a l'hora de calcular el nombre d'iteracions $It(n)$ del bucle. Inicialment $i = N$, després de la primera iteració $i = N$ div 2, a continuació $i = N$ div 2^2 i així successivament fins que $i = 0$. El nombre d'iteracions $It(n)$ serà el nombre d'elements de la sequència:

$$N, N \text{ div } 2, N \text{ div } 2^2, \cdots, N \text{ div } 2^k, \cdots, 1$$

i per tant $It(n) = 1 + \lfloor \log_2 N \rfloor \in O(\log N) = O(\log n)$.

Finalment:

$$O(Pot) = O(Inic) + O(\textbf{mentre} \cdots \textbf{fmentre}) + O(\textbf{retorna}) =$$
$$= O(1) + O(\log n) * O(1) + O(1) = O(\max\{1, \log n, 1\}) = O(\log n).$$

3) Tot seguit estudiarem la complexitat de la funció Bin que transforma el nombre natural N en el nombre natural m de manera que la representació de m en base deu és igual que la representació de N en base dos.

funcio Bin (N: nat) retorna m: nat
$\{P \equiv \textbf{cert}\}$
var k, i: nat **fvar**
$m := 0; k := 1; i := N$
mentre $i > 0$ **fer**
    $m := m + (i \bmod 2) * k$
    $k := 10 * k$
    $i := i$ div 2
fmentre
$\{Q \equiv m = \Sigma\, \alpha \in N \cdot [(N \text{ div } 2^\alpha) \bmod 2] * 10^\alpha\}$
retorna m
ffuncio

En aquest cas agafarem com a gràndaria de les dades d'entrada la longitud de N, $n = \log_2 N$. Amb raonaments semblants als de l'exemple de la funció Pot2, es pot comprovar que $It(n) = 1 + \lfloor \log_2 N \rfloor \in O(n)$ i per tant:

$$O(Pot) = O(Inic) + O(\textbf{mentre} \cdots \textbf{fmentre}) + O(\textbf{retorna}) =$$
$$= O(1) + O(n) * O(1) + O(1) = O(\max\{1, n, 1\}) = O(n).$$

4) A continuació estudiarem la complexitat de la funció Sumzero que fa un recorregut per a determinar si existeixen quatre elements consecutius d'un vector que sumin 0.

Suposarem definit el tipus: vector = taula[1..N] d'enters.

funcio Sumzero (V: vector) retorna b: boolea
$\{P \equiv \mathbf{cert}\}$
var i, j: nat, s: ent **fvar**
$b := fals; i := 0$
mentre $i \leq N - 4$ **fer**
s:=0; j:=1
mentre $j \leq 4$ **fer**
$s := s + V[i + j]$
$j := j + 1$
fmentre
$b := b \vee (s = 0)$
$i := i + 1$
fmentre
$\{Q \equiv b = \exists\, \alpha \in [0..N - 4] \cdot (\Sigma\beta \in [1..4] \cdot V[\alpha + \beta] = 0)\}$
retorna b
ffuncio

Com a gràndaria de les dades d'entrada prendrem $n = N$. Cal observar que cada vegada que s'executa el bucle interior fa exactament 4 iteracions, i que totes les instruccions que es fan dins del bucle són de complexitat constant. Per tant tindrem:

$$O(\mathbf{mentre} \cdots \mathbf{interior} \cdots \mathbf{fmentre}) = O(1).$$

El nombre d'iteracions del bucle exterior és $It(n) = n - 3$ i totes les instruccions del bucle són de complexitat constant. Així finalment tindrem:

$$O(Sumzero) = O(Inic) + O(\mathbf{mentre} \cdots \mathbf{exterior} \cdots \mathbf{fmentre}) + O(\mathbf{retorna}) =$$
$$= O(1) + O(n) * O(1) + O(1) = O(\max\{1, n, 1\}) = O(n).$$

5) Estudiarem l'ordre de complexitat de l'algorisme d'ordenació dels elements d'un vector pel mètode de selecció. Farem servir el tipus vector definit a l'exemple 4) i els predicats esta_ordenat(v) i esta_permutat(V, v) que estableixen, respectivament, que el elements del vector v estan ordenats creixement i que els elements del vector v són una permutació dels elements del vector V.

funcio Selec (V: vector) retorna v: vector
$\{P \equiv \mathbf{cert}\}$
var $i, j, pmin$: nat **fvar**
$v := V; i := 1$
mentre $i < N$ **fer**
pmin:=i; j:=i+1
mentre $j < N$ **fer**
Si $v[j] < v[pmin]$ **llavors** $pmin := j$ **fsi**
$j := j + 1$
fmentre
$< v[i], v[pmin] >:=< v[pmin], v[i] >$
$i := i + 1$
fmentre
$\{Q \equiv \text{esta_ordenat}(v) \wedge \text{esta_permutat}(V, v)\}$
retorna v
ffuncio

Com a gràndaria de les dades d'entrada prendrem $n = N$. Convé observar que serà $O(v := V) = O(n)$ i $O(\textbf{retornav}) = O(n)$. Durant l'iteració i del bucle exterior el bucle interior itera exactament $n - i$ vegades i les operacions que s'executen al seu interior són de complexitat constant. Per tant tindrem:

$$O((\textbf{mentre} \cdots \textbf{interior} \cdots \textbf{fmentre})_i) = O(n - i).$$

El nombre d'iteracions del bucle exterior és $It(n) = n - 1$ i, com ja hem vist, la complexitat de les operacions de l'iteració i és $O(n - i)$. Així doncs la complexitat del bucle exterior serà:

$$O(\textbf{mentre} \cdots \textbf{exterior} \cdots \textbf{fmentre}) = \sum_{i=1}^{n-1} O(n-i) = O\left(\sum_{i=1}^{n-1} n - i\right) =$$

$$= O\left(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i\right) = O\left((n-1)n - \frac{n(n-1)}{2}\right) = O\left(\frac{n(n-1)}{2}\right) = O(n^2).$$

Finalment tindrem:

$$O(Selec) = O(Inic) + O(\textbf{mentre} \cdots \textbf{exterior} \cdots \textbf{fmentre}) + O(\textbf{retorna}) =$$

$$= O(n) + O(n^2) + O(n) = O(\max\{n, n^2\}) = O(n^2).$$

12 Eficiència d'algorismes recursius

En tots els casos un arbre de crides pot ajudar a trobar l'eficiència d'un algorisme recursiu.

Les fórmules que donem a continuació només són aplicables quan la nova crida de l'algorisme recursiu es fa sobre unes dades que tenen una grandària n' que s'obté de la grandària n de les dades anteriors mitjançant substracció o divisió.

Siguin:

- n: grandària de les dades de la crida anterior;
- a: nombre de crides recursives que fa l'algorisme;
- $n' = n - b$ o $n' = n/b$: grandària de les noves dades;
- cn^k: cost de les instruccions que no són crides recursives.

Són vàlides les següents fórmules:

REDUCCIÓ PER SUBSTRACCIÓ

Si

$$T(n) = \begin{cases} cn^k & \text{si} \quad 0 \leq n < b \\ aT(n-b) + cn^k & \text{si} \quad n \geq b \end{cases}$$

llavors

$$T(n) \in \begin{cases} O(n^k) & \text{si} \quad a < 1 \\ O(n^{k+1}) & \text{si} \quad a = 1 \\ O(a^{n/b}) & \text{si} \quad a > 1 \end{cases}$$

REDUCCIÓ PER DIVISIÓ

Si

$$T(n) = \begin{cases} cn^k & \text{si} \quad 1 \leq n < b \\ aT(n/b) + cn^k & \text{si} \quad n \geq b \end{cases}$$

llavors

$$T(n) \in \begin{cases} O(n^k) & \text{si} \quad a < b^k \\ O(n^k \log_b n) & \text{si} \quad a = b^k \\ O(n^{\log_b a}) & \text{si} \quad a > b^k \end{cases}$$

Bibliografia

Aho A.V., Hopcroft J.E and Ullman J.D., "*The design and analysis of computer algorithms*", Adison-Wesley, 1974.

Abellanas, M. y Lodares, D., "*Análisis de algoritmos y teoría de grafos*", RA-MA, 1984.

Brassard G. and Bratley P., "*Algorítmica. Concepción y análisis*", Ed. Masson, 1990.

Brassard G. and Bratley P., "*Fundamentos de Algoritmia*", Prentice-Hall", 1997.

Cormen T.H., Leiserson C.E. and Rivest R.L., "*Introduction to Algorithms*", MIT Press, 1994.

Galve, J., González, J.C., Sánchez, A. y Velázquez, J.A., "*Algorítmica. Diseño y Análisis de Algoritmos Funcionales e Imperativos*", RA-MA, 1993.