Millores d'eficiència
en programes recursius i iteratius

Assignatura PRO2

Novembre 2013

Índex

Capítol 7

Eficiència de programes

Entenem per *eficiència* (o *cost*) d'un algorisme la mesura dels recursos que necessita per funcionar. Usem aquest concepte per comparar algorismes que resolen el mateix problema o també per establir com de bo és un algorisme en si mateix. Es tracta d'un criteri totalment objectiu per determinar la qualitat dels nostres algorismes o programes.

Els recursos habituals a mesurar són el temps d'execució i l'espai de memòria. En general, els dos criteris estan relacionats: un consum més gran d'espai pot ajudar a obtenir algorismes més ràpids. Per això, a la majoria de situacions s'ha d'arribar a un compromís entre ambdós.

En aquests apunts ens centrarem a l'eficiència en temps. Si volem un estudi "a priori", és a dir, sense haver d'executar els nostres programes per conèixer quan triguen i, a més, volem fer mesures justes hem de fixar algunes "normes":

- no es consideren factors externs al programa com ara, la màquina on es fan les execucions, el sistema operatiu o el llenguatge de programació.
- el temps d'execució es mesura en termes del nombre d'instruccions de l'algorisme en relació amb la mida de les dades (dimensió en el cas dels vectors, valor en el cas dels números, nombre de files en el cas de les matrius, ...)
- per cada algorisme mirarem d'identificar el cas pitjor i només calcular el cost d'aquest, ja que en moltes ocasions hi ha una gran diferència entre els diferents casos d'un mateix algorisme

Alguns exemples de cost en temps

- Obtenir la mida d'un vector, pila, cua o llista (amb `size()`); obtenir el valor d'una posició d'un vector, el cim d'una pila, el primer d'una cua, l'arrel d'un arbre o l'associat a un iterador a una llista. El cost d'aquestes operacions és independent de la mida de les respectives estructures (al segon grup, però, s'ha de tenir en compte la mida dels elements: no és el mateix obtenir un enter que una paraula, per exemple). Diem que aquestes operacions tenen cost *constant*. En general, totes les operacions que hem vist a les especificacions de piles, cues, llistes o arbres són de cost constant (no ho són, obviament, l'assignació o còpia,

la lectura o la escriptura; tampoc el dimensionament de vectors). Per veure el cost d'altres operacions, com ara, `sort` o `merge` consulteu la web de referència de la STL. L'assignació de tipus simples i els operadors aritmètics i lògics també es consideren de cost constant.

- Suma dels elements d'una pila, un vector, etc.; cerques simples; assignació o còpia de vectors, llistes, etc. de tipus simples. El tractament de cada element visitat és un nombre constant d'instruccions de cost constant, per tant el nombre total d'aquestes es directament proporcional a la mida de l'estructura. En aquest cas parlem d'algorismes *lineals*.

- Algorismes simples d'ordenació de vectors (inserció, bombolla, ...), veure si una estructura seqüencial té tots els seus elements diferents (sense ordenar-los). Cada element de l'estructura dóna lloc a un nombre d'instruccions de cost constant que és funció del nombre d'elements anteriors o següents. El nombre total d'aquestes es directament proporcional al quadrat de la mida de l'estructura. En aquest cas parlem d'algorismes *quadràtics*.

- Algorismes sofisticats d'ordenació de vectors (mergesort, heapsort). El nombre d'instruccions de cost constant és proporcional a la mida del vector *multiplicada* pel seu logaritme en base 2. En aquest aspecte, l'algorisme quicksort és especial: si considerem el seu cost per al cas mitjà pertany a aquest grup, però pel seu cas pitjor és quadràtic.

- Cerca dicotòmica, algorismes ràpids per l'arrel quadrada. El nombre d'instruccions de cost constant és proporcional al logaritme en base 2 de la mida de les dades (vector, número). En aquest cas parlem d'algorismes *logarítmics*.

- Generació de combinacions, permutacions, etc., de n elements. Algorismes de cerca exhaustiva de candidats de mida n per complir una propietat. El nombre d'instruccions de cost constant és proporcional a una constant més gran que 1 elevada a la n-èsima potència. En aquest cas parlem d'algorismes *exponencials*.

Encara que nosaltres obtindrem el cost dels exemples que mostrarem, l'objectiu d'aquesta assignatura no és aprendre a calcular costos d'algorismes en general. Si ho és, però, ser capaços de detectar si els algorismes que dissenyem són innecessàriament ineficients i per tant hem de trobar alternatives més bones.

En una aproximació molt bàsica, podem dir que per obtenir algorismes eficients hi ha dos camins: encertar amb una bona idea que ens permeti desenvolupar l'algorisme directament, o bé, proporcionar un algorisme correcte sense fixar-nos gaire a l'eficiència i després mirar de trobar millores que ens portin a una solució eficient, a base de eliminar càlculs innecessaris. A la resta dels apunts mostrarem algunes técniques per a la segona opció.

Capítol 8

Millores d'eficiència d'algorismes recursius

Partirem d'algorismes recursius correctes i naturals, i mirem d'identificar i estalviar els càlculs innecessaris. L'eina bàsica per fer això és la immersió. En ocasions, afegirem noves dades amb la intenció de moure informació entre una crida recursiva i la següent, de manera que aquesta se'n pugui aprofitar i no hagi de tornar a calcular-la. D'això se'n diu *immersió d'eficiència per dades*. D'altres ocasions el que fem és obtenir informació procedent de la següent crida recursiva per estaviar-nos instruccions a l'actual. D'això se'n diu *immersió d'eficiència per resultats*. També pot haver-hi combinacions de les dues.

8.1 Números de Fibonacci

La successió de Fibonacci és molt coneguda. Es defineix així

$$\begin{aligned} &\text{fib}(0) = 0 \\ &\text{fib}(1) = 1 \\ &\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), \text{ si } n > 1 \end{aligned}$$

D'aquesta definició es deriva una funció recursiva molt simple per calcular l'n-èssim terme de la successió. Definim el resultat com a `int`, però donada la gran rapidesa del creixement de fib, si volem calcular termes superiors al 45é haurem de fer servir `double`.

```
int fib (int n){
/* Pre: n és un enter >=0 */
/* Post: el resultat és el terme n-èssim de la successió de Fibonnaci */
  int res;
  if (n<2) res = n;
  else res=fib(n-1)+fib(n-2);
  return res;
}
```

Es veu clarament que per valors n suficientment grans la repetició de càlculs innecessaris és enorme. Per exemple, si $n = 20$, cridem la funció dues vegades: $\text{fib}(19)$ i $\text{fib}(18)$. En executar

la crida amb 19, obtenim dues noves crides fib(18) i fib(17). La primera ja s'ha fet abans i no s'hauria de repetir. Si continuem desplegant les crides, ens en trobem tres amb 17, cinc amb 16, vuit amb 15, etc.

El que podem fer és una immersió per resultats que retorni dos valors consecutius de la successió. D'aquesta manera, a cada crida aprofitarem els elements retornats des de la següent per obtenir-ne els nous.

```
pair<int,int> fib_ef (int n) {
/* Pre: n és un enter >0 */
/* Post: el primer component del resultat és fib(n); el segon és fib(n-1) */
  pair<int, int>  y;
  if (n==1)  {y.first=1; y.second=0;}
  else {
    y = fib_ef(n-1);
    /* HI: y.first és fib(n-1); y.second és fib(n-2) */
    int aux;
    aux=y.first;
    y.first=y.first+y.second;
    y.second=aux;
  }
  return y;
}
```

Noteu que per obtenir fib(0) s'ha de cridar aquesta funció amb $n = 1$, o bé calcular-ho a part, ja que la crida amb $n = 0$ no és vàlida. Al segon cas, la funció completa quedaria així, amb la crida a la immersió.

```
int fib (int n)
/* Pre: n>=0 */
/* Post: el resultat és el terme n-èssim de la successió de Fibonnaci */
{
  int res;
  pair<int, int> res2;
  if (n==0) res=0;
  else {
    res2=fib_ef(n);
    res=res2.first;
  }
  return res;
}
```

Fixeu-vos també que si definim la immersió passant per referència els dos valors obtinguts de la successió, encara tenim una immersió per resultats, donat que aquests paràmetres només s'aprofiten *després* de la crida. Completeu aquesta versió com a exercici.

Per mesurar l'eficiència de les dues versions, fixem-nos que, per calcular fib(n), la segona produeix n crides i a cada crida s'aplica un nombre constant d'operacions, llavors és lineal. El

càlcul de la primera és més complicat, però es pot demostrar per inducció que el nombre de crides que produeix és aproximadament l'$(n+1)$-èssim número de Fibonacci, cadascuna d'elles, això sí, amb un nombre constant d'instruccions a cada crida. Com que se sap que $\text{fib}(n)$ és proporcional a certa constant més gran que 1 elevada a n, ens queda un algorisme exponencial.

Mirem ara d'obtenir una solució alternativa amb immersió d'eficiència per dades. La idea ha de ser anar fent càlculs (útils) i passar-los com a paràmetres a les següentes crides recursives. En aquest cas, podem mantenir dos valors consecutius de *fib* i actualitzar-los fins arribar al que volem. Per saber quan hem de parar necessitem un paràmetre més.

```
int fib_ef_param(int n, int i, int f1, int f2)
/* Pre: 0<i<=n, f1=fib(i-1), f2=fib(i) */
/* Post: el resultat és el terme n-èssim de la successió de Fibonnaci */
{
  int f;
  if (i==n) f=f2;
  else f=fib_ef_param(n,i+1,f2,f1+f2); {HI: Post}
  return f;
}
```

La clau per entendre per què aquesta funció és correcta és que qualsevol crida recursiva amb qualsevol tripleta `<i,f1,f2>` que compleixi la precondició ens porta a la postcondició, ja que el que tenim de fet es un cas de recursivitat final amb postcondició constant. Com a exercici, obteniu la versió iterativa que es deriva d'aquesta.

Amb aquesta solució, $\text{fib}(0)$ s'ha de calcular a part. La funció completa quedaria així, amb la crida a la immersió.

```
int fib (int n)
/* Pre: n>=0 */
/* Post: el resultat és el terme n-èssim de la successió de Fibonnaci */
{
  int res;
  if (n==0) res=0;
  else res=fib_ef_param (n,1,0,1);
  return res;
}
```

Quant a eficiència, novament aquesta versió requereix n crides per calcular $\text{fib}(n)$ i a cada crida s'aplica un nombre constant d'operacions, llavors és lineal.

8.2 Element dominador més alt d'una pila de naturals

Diem que un element d'una pila de números és *dominador* si és mes gran o igual que la suma dels elements més antics (visualment, els que se situen més abaix) que ell. Donada una pila no buida de naturals, volem obtenir el seu element dominador més recent o, equivalentment, més alt.

Noteu que, pel fet que la pila sigui de naturals, com a mínim tindrà un dominador, que serà el seu element més baix (és a dir, el més antic), si apliquem com a conveni que la suma de zero números és 0. Amb tota aquesta informació obtenim un algorisme molt bàsic per al nostre objectiu.

```
int suma(stack<int> p)
/* Pre: p=P */
/* Post: el resultat és la suma dels elements de P */
{
  int s=0;
  /* Inv: s és la suma dels elements de la part tractada de P,
     p és la part no tractada de P */
  while (not p.empty()){
    s+=p.top();
    p.pop();
  }
  return s;
}

int dom(stack<int> & p)
/* Pre: p=P, p no és buida i està formada per naturals */
/* Post: El resultat és el dominador més alt de P */
{
  int res=p.top();
  p.pop();
  if (not p.empty()){
    if (res<suma(p)) res=dom(p);
    /* HI: res és el dominador més alt de P sense el cim */
  }
  return res;
}
```

Comproveu que sota les condicions del segon `if`, es compleix que `HI` $\Rightarrow$ `Post`, ja que s'ha descartat que el cim original sigui dominador.

El problema és que aquest algorisme calcula moltes sumes repetides. A la primera crida se sumen els elements de tota la pila excepte el cim. A la segona, es tornen a sumar els mateixos elements, tret del segon cim, i així successivament. El nombre total de sumes és quadràtic respecte a la mida de la pila original.

Per obtenir una algorisme més eficient, mirem de fer cada suma una sola vegada. Per fer això, apliquem una immersió de resultats que retorni no només l'element dominador sinò també la suma de la pila. Així, en una crida només haurem de sumar un element a la suma retornada per l'anterior.

```
pair<int,int> dom_ef(stack<int>& p)
/* Pre: p=P, p no és buida i està formada per naturals */
/* Post: El resultat conté el dominador més alt de P i la suma */
```

```
{
  pair<int,int> res;
  res.first=res.second=p.top();
  p.pop();
  if (not p.empty()){
    pair<int,int> aux = dom_ef(p);
    /* HI: aux conté el dominador més alt de P menys el cim i la suma */
    if (res.first<aux.second) res.first=aux.first;
    res.second+=aux.second;
  }
  return res;
}
```

Clarament, el procés genera tantes crides com l'altura de la pila i a cada crida s'efectua un nombre constant d'instruccions bàsiques, per tant l'algorisme és lineal.

També es pot aconseguir un algorisme lineal amb una immersió per dades, si abans de començar el càlcul disponem de la suma de la pila. El nombre d'instruccions seguirà sent proporcional a l'altura de la pila, però el resultat no és tan satisfactori, ja que s'ha de començar per obtenir la suma total de la pila.

```
int dom_ef_param(stack<int>& p, int& s)
/* Pre:  p=P, p no és buida i està formada per naturals;
         s es la suma de P menys el cim */
/* Post: El resultat és el dominador més alt de P */
{
  int res;
  res=p.top();
  if (res<s) {
    p.pop();
    if (not p.empty()){
      s-=p.top();
      res = dom_ef_param(p,s);
      /* HI: res és el dominador més alt de P menys el cim */
    }
  }
  return res;
}
```

Com a les anteriors immersions per dades, el punt crític és garantir que els paràmetres de la crida recursiva compleixen la precondició. Per això, si traiem el cim de `P`, hem de restar a `s` el següent cim.

La funció completa queda

```
int dom_ef2(stack<int> & p)
/* Pre: p=P, p no és buida i està formada per naturals */
```

```
/* Post: El resultat és el dominador més alt de P */
{
  int s=suma(p)-p.top();
  res=dom_ef_param(p,s);
  return res;
}
```

8.3 Arbre binari balancejat

Un arbre binari es diu *balancejat* si és buit o bé si els seus dos fills són balancejats i el valor absolut de la diferència de les seves altures no és més gran que 1. A partir d'aquesta definició, derivem un algorisme molt senzill però, com veurem, molt ineficient.

```
int altura(Arbre<int>& a)
/* Pre: a=A */
/* Post: El resultat és la longitud del camí més llarg de l'arrel a una fulla d'A */
{
  int alt;
  if (a.es_buit()) alt= 0;
  else{
    Arbre<int> a1;
    Arbre<int> a2;
    a.fills(a1,a2);
    int y=altura(a1);
    int z=altura(a2);
    /* HI: y és la longitud del camí més llarg de l'arrel a una fulla del f.e. d'A,
      z és la longitud del camí més llarg de l'arrel a una fulla del f.d. d'A */
    alt= max(y,z)+1;
  }
  return alt;
}

bool balancejat(Arbre<int>& a)
/* Pre: a=A */
/* Post: el resultat indica si A és balancejat */
{
  bool bal;
  if (a.es_buit()) bal=true;
  else{
    Arbre<int> a1;
    Arbre<int> a2;
    a.fills(a1,a2);
    Arbre<int> b1=a1;
    Arbre<int> b2=a2;
    int y=altura(b1);
```

```
    int z=altura(b2);
    if (abs(y-z)>1) bal=false;
    else {
      bal=balancejat(a1);
      /* HI1: bal indica si el fill esq d'A és balancejat */
      if (bal) bal=balancejat(a2);
               /* HI2: bal indica si el fill dret d'A és balancejat */
    }
  }
  return bal;
}
```

Noteu que mentre no arribem al cas directe, a cada crida a `balancejat` es crida la funció `altura` dues vegades i, al pitjor dels casos, també es crida la funció `balancejat` dues vegades. Si tenim un arbre original `a` suficientment gran i desenvolupem la seqüència de crides a `altura` i a `balancejat`, veurem que l'altura del fill esquerre del fill esquerre d'`a` es calcula dues vegades (una com a part de l'altura del fill esquerre d'`a` i altra com a part de la crida a `balancejat` del fill esquerre d'`a`. El mateix passa amb els restants tres fills dels fills d'`a`. A la següent crida a `balancejat`, comprovarem que per a cada un dels corresponents fills l'altura es calcula tres vegades, a la següent quatre vegades, etc. Es pot demostrar per inducció que, pel cas pitjor, aquest procés produeix un nombre quadràtic de crides i per tant d'instruccions totals.

Per estalviar els càlculs innecessaris farem una immersió per resultats, de forma que la mateixa funció ens retorni el booleà que indica si l'arbre és balancejat i la seva altura. D'aquesta manera, aprofitem l'altura dels subarbres per saber si l'arbre original és balancejat.

```
pair<bool,int> ebalancejat(Arbre<int>& a)
/* Pre: a=A */
/* Post: el resultat conté si l'arbre A es balancejat i l'altura */
{
  pair<bool,int> res;
  if (a.es_buit()) res=make_pair(true,0);
  else{
    Arbre<int> a1;
    Arbre<int> a2;
    a.fills(a1,a2);
    pair<bool,int> ba1=ebalancejat(a1);
    /* HI1: el resultat conté si el fill esq d'A es balancejat i l'altura */
    pair<bool,int> ba2=ebalancejat(a2);
    /* HI2: el resultat conté si el fill dret d'A es balancejat i l'altura */
    res= make_pair(abs(ba1.second-ba2.second)<=1 and ba1.first and ba2.first,
                   max(ba1.second,ba2.second)+1);
  }
  return res;
}
```

Ara és senzill concloure que el nombre de crides és menor o igual que la mida de l'arbre i com que a cada crida s'executa un nombre constant d'instruccions bàsiques, l'algorisme és lineal.

Capítol 9

Millores d'eficiència d'algorismes iteratius

Com abans, partirem d'algorismes iteratius correctes i naturals, i mirem d'identificar i estalviar els càlculs innecessaris. Aquest cop, l'estratègia consistirà en fer servir el resultat de cada volta del bucle corresponent per minimitzar el cost del càlcul de la següent. Per fer això, introduirem variables locals. És molt important que el valor d'aquestes variables consti a l'invariant de la nova versió; així sabrem exactament com inicialitzar-les i actualitzar-les.

9.1 Desenvolupament de Taylor de la funció e^x

Recordem que el desenvolupament en sèrie de Taylor d'una funció $f(x)$ al punt a es la suma

$$f(x) = f(a) + \frac{f'(a)(x-a)}{1!} + \frac{f''(a)(x-a)^2}{2!} + \cdots + \frac{f^n(a)(x-a)^n}{n!} + \cdots$$

Volem un programa que calculi de manera eficient l'exponencial e^x, fent servir el seu desenvolupament de Taylor al punt zero fins al terme n-èssim, és a dir,

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}$$

La primera idea és iterar la suma terme a terme sense aprofitar els termes anteriors. Fem això amb les funcions auxiliars `pot` i `fact`.

```
double pot (double a, int b)
/* Pre: a>0; b>=0 */
/* Post: el resultat és a^b */
{
  double p=1;
  while (b>0) {
    p*=a;
    --b;
  }
  return p;
```

```
}

int fact(int n)
/* Pre: n>=0 */
/* Post: El resultat és n! */
{
  int f = 1;
  while (n>0){
    f*=n;
    --n;
  }
  return f;
}

double taylor_exp(double x, int n)
/* Pre: n>=0 */
/* Post: el resultat és la suma fins n de la sèrie de Taylor per a e^x */
{
  if (x==0) return 1;
  // x!=0
  double t=1;
  int i=1;
  // Inv: t és la suma fins i-1 de la sèrie de Taylor per a e^x
  while (i<=n) {
    t+=pot(x,i)/fact(i);
    ++i;
  }
  return t;
}
```

Podem veure que cada crida a `taylor_exp(x,n)` genera un nombre de productes quadràtic respecte a `n`.

Clarament, no és necessari fer cada vegada tots els productes de la potència i el factorial. Introduïm dues variables noves que continguin els valors corresponents. Només hem d'inicialitzar-les i actualitzar-les a cada volta del `while`.

```
double taylor_exp_ef1(double x, int n)
/* Pre: n>=0 */
/* Post: retorna la suma fins n de la sèrie de Taylor per a e^x */
{
  if (x==0) return 1;
  double t=1;
  int f=1;
  double p=1;
  int i=1;
  // Inv: t és la suma fins i-1 de la sèrie de Taylor per a e^x
```

```
  //      p = x^(i-1); f=(i-1)!;
  while (i<=n){
    p*=x; //p=x^i
    f*=i; //f=i!
    t+=p/f;
    ++i;
  }
  return t;
}
```

Amb això, el nombre de productes (i, de fet, d'operacions en total) passa a ser lineal respecte a `n`. Fins i tot, podem estalviar una de les variables auxiliars, mantenint-ne només una que contingui el nou terme de cada iteració. A més, d'aquesta manera baixem la probabilitat d'obtenir números massa grans per poder-se representar amb `double`.

```
double taylor_exp_ef2(double x, int n)
/* Pre: n>=0 */
/* Post: retorna la suma fins n de la sèrie de Taylor per a e^x */
{
  if (x==0) return 1;
  double t=1;
  double pf=1;
  int i=1;
  // Inv: t és la suma fins i-1 de la sèrie de Taylor per a e^x
  //      pf = x^(i-1)/(i-1)!;
  while (i<=n){
    pf*=x;  //pf = x^i/(i-1)!
    pf/=i; //pf=x^i/i!
    t+=pf;
    ++i;
  }
  return t;
}
```

9.2 Element dominador més avançat d'una llista de naturals

Diem que un element d'una llista de números és *dominador* si és mes gran o igual que la suma dels elements anteriors (els que se situen més a prop del seu `begin`) que ell. Donada una llista no buida de naturals, volem obtenir el seu element dominador més avançat, és a dir, el més allunyat del `begin`. Noteu que, pel fet que la llista sigui de naturals, com a mínim tindrà un dominador, que serà el `begin`), si apliquem com a conveni que la suma de zero números és 0.

Amb tota aquesta informació obtenim un algorisme molt bàsic per al nostre objectiu, amb l'ajuda d'una operació auxiliar que calcula la suma dels anteriors a un element donat y que es crida per a cada element de la llista.

```
int suma(const list<int>& L, list<int>::const_iterator it)
/* Pre: cert */
/* Post: El resultat és la suma dels elements d'L des del begin fins
   a l'anterior al referenciat per it */
{
  int s=0;
  for(list<int>::const_iterator it2= L.begin();it2 != it; ++it2)
    s+=(*it2);
  return s;
}

int dom(const list<int>& L)
/* Pre: L no és buida  i està formada per naturals */
/* Post: El resultat és el dominador més avancat de L */
{
  list<int>::const_iterator it= L.begin();
  int s=(*it);
  ++it;
  // Inv: s és l'element dominador més avançat de L fins l'anterior a it
  while (it != L.end()){
    if (suma(L,it)<=(*it)) s=(*it);
    ++it;
  }
  return s;
}
```

Podem veure que cada crida a `dom(L)` genera un nombre de sumes quadràtic respecte al nombre d'elements d'`L`.

Clarament, no és necessari fer cada vegada totes les sumes dels elements anteriors als visitats. Introduïm una variable nova que contingui la suma necessària a cada moment. Només hem d'inicialitzar-la i actualitzar-la a cada volta del `while`.

```
int dom_ef(const list<int>& L)
/* Pre: L no és buida i està formada per naturals */
/* Post: El resultat és el dominador més avancat de L */
{
  list<int>::const_iterator it= L.begin();
  int s,aux;
  s=aux=(*it);
  ++it;
  // Inv: s és l'element dominador més avançat d'L fins l'anterior a it;
  //      aux és la suma dels elements d'L fins l'anterior a it
  while (it != L.end()){
    if (aux<=(*it)) s=(*it);
    aux+=(*it);
    ++it;
```

```
  }
  return s;
}
```

Amb això, el nombre de sumes (i, de fet, d'operacions en total) passa a ser lineal respecte al nombre d'elements d'`L`.

9.3 Vector mitjanament ordenat

Diem que un vector d'enters és mitjanament ordenat si cada element, tret del primer, és més gran o igual que la mitja dels anteriors. Donat un vector no buit d'enters, volem saber si és mitjanament ordenat.

Apliquem un esquema de cerca. El primer algorisme que proposem fa servir una operació auxiliar que calcula la suma dels anteriors a un element donat y que es crida per a cada element del vector, donant lloc a un cost quadràtic.

```
int suma(const vector<int> &v, int i)
/* Pre; i>=0 */
/* Post: El resultat és la suma de v[0..i-1] */
{
  int s=0;
  for (int j=0; j<i;++j)
    s+=v[j];
  return s;
}

bool mitjord(const vector<int> &v)
/* Pre: v.size()> 0 */
/* Post: El resultat indica si v está mitjanament ordenat */
{
  int i=1;
  bool b=true;
  //Inv: v[0..i-1] está  mitjanament ordenat;
  //     si not b llavors v[0..i] no está mitjanament ordenat
  while (i<v.size() and b)
    if (v[i]<double(suma(v,i))/i) b=false;
    else ++i;
  return b;
}
```

Com als exemples anteriors, evitem fer cada vegada totes les sumes dels elements anteriors als visitats. Introduïm una variable nova que contingui la suma necessària a cada moment. Només hem d'inicialitzar-la i actualitzar-la a cada volta del `while`.

```
bool mitjord_ef(const vector<int> &v){
/* Pre: v.size()> 0 */
/* Post: El resultat indica si v está mitjanament ordenat */
  int i=1;
  bool b=true;
  int s=v[0];
  //Inv: v[0..i-1] està mitjanament ordenat;
  //     si not b llavors v[0..i] no està mitjanament ordenat;
  //     s= suma v[0..i-1]; 1<=i; i<=v.size();
  while (i<v.size() and b)
    if (v[i]<double(s)/i) b=false;
    else {
      s+=v[i];
      ++i;
    }
  return b;
}
```

Amb això, el nombre de sumes (i, de fet, d'operacions en total) passa a ser lineal respecte al nombre d'elements d'`v`.

Ineficiència per còpies innecessàries d'objectes

Retomem l'exemple de recorregut en amplada d'un arbre binari, vist al capítol "Estructures de dades arborescents", per analitzar el seu cost amb les eines d'aquest capítol. Encara que fem servir la versió iterativa, tot el que explicarem també es pot aplicar a la versió recursiva.

```
void nivells(Arbre<int> &a, list<int> & l)
/* Pre: a=A, l és buida */
/* Post: l conté els nodes d'A en ordre creixent respecte al nivell
         al qual es troben, i els de cada nivell en ordre d'esquerra a dreta */
{
  if(not a.es_buit()){
    queue <Arbre<int> > c;
    int node;
    c.push(a);
    while(not c.empty()){
      a=c.front();
      node=a.arrel();
      l.insert(l.end(),node);
      Arbre<int> a1;
      Arbre<int> a2;
      a.fills(a1,a2);
      if (not a1.es_buit()) c.push(a1);
      if (not a2.es_buit()) c.push(a2);
      c.pop();
    }
  }
}
```

Al seu moment vam dir que aquest algorisme no és gaire eficient perquè cada vegada que cridem l'operació `front` de la cua es fa una còpia de l'arbre obtingut, ja que l'assignació de la classe `Arbre` té aquest efecte. De la mateixa manera, cada vegada que afegim un arbre a la cua amb l'operació `push` també se'n fa una còpia.

Podem mesurar el cost del recorregut de la següent manera. A cada volta es visita un element de l'arbre i s'executa un nombre constant d'operacions. Si el cost de cada una d'aquestes operacions fos també constant, l'algorisme complet seria lineal. Com ja hem dit, això no és així perque

la còpia d'arbres no és constant, sinó lineal. Per tant, per obtenir el cost real de l'algorisme hem de tenir en compte aquestes còpies. Com que diferents arbres poden donar lloc a diferents descomposicions en subarbres, no és senzill sumar progressivament el cost de les diferents còpies; en comptes d'això, calcularem el nombre total de vegades que els nodes de l'arbre són copiats al llarg del tot el procés.

Sabem que tots els subarbres no buits de l'arbre original van passant per la cua, és a dir, són paràmetres de l'operació `push` i són retornats per l'operació `front`. El nombre de subarbres no buits coincideix amb el nombre d'elements de l'arbre, però resulta que cada element apareix a tants subarbres com a la profunditat a la que es troba més 1 (l'arrel de l'arbre només apareix a un subarbre, les següents arrels apareixen a dos subarbres, etc.) Per tant, cada element es copiat a tants `push` i `front` com la seva profunditat més 1. Si sumem aquesta quantitat al llarg de tots els elements de l'arbre, tindrem el cost de l'algorisme.

El problema, ara, és que diferents arbres poden donar lloc a diferents sumes. Recordem que hem dit al principi que ens conformem amb el cost al cas pitjor, per tant una bona estratègia serà trobar una fita superior pel cost i després veure que hi ha casos on s'assoleix la fita i aquests determinaran el cost al cas pitjor.

Donat que la profunditat d'un node mai supera la mida de l'arbre, si sumem aquestes profunditats al llarg de tots els elements de l'arbre, el resultat és *com a molt* quadràtic. Si a és un arbre de mida n podem afitar la suma de profunditats així

$$\sum_{x\in a} 1 + \text{prof d'}x \text{ en } a = n + \sum_{x\in a} \text{prof d'}x \text{ en } a \leq n + \sum_{x\in a} n = n + n^2$$

A més, fixeu-vos, que per a qualsevol mida hi ha arbres, com per exemple els arbres "lineals" en què tots els nodes (menys l'últim) tenen exactament un subarbre buit. La suma en qüestió per a un arbre d'aquests de mida n seria $1+2+3+\cdots+n = \frac{n(n+1)}{2}$. Per tant el cost al cas pitjor és quadràtic.

Podríem haver aplicat l'estratègia en sentit invers: primer determinar que per a tot n hi ha arbres de mida n amb una suma de profunditats $\frac{n(n+1)}{2}$, que permetria afirmar que el cost és *com a mínim* quadràtic, i després provar que no es pot superar aquest cost amb cap arbre de mida n.

Un càlcul similar s'aplica a la versió ineficient de l'algorisme per saber si un arbre és balancejat, vist més amunt.

Per obtenir una versió lineal hauríem d'aconseguir que totes les operacions del bucle fossin constants. Això es pot fer amb una cua de *punters* a arbre, o bé programant el recorregut dins de la classe `Arbre`, coses que encara no estem en condicions de fer, però que sí podrem intentar al capítol següent.