Millores d'eficiència en recursió i iteració

Ricard Gavaldà

Programació 2
Facultat d'Informàtica de Barcelona, UPC
Primavera 2019

Aquesta presentació no substitueix els apunts

Contingut

Eliminació de càlculs repetits

Eficiència per eliminació de càlculs repetits

Iteració:

- Afegim variables locals que recorden càlculs ja efectuats per a la propera iteració
- No apareixen en la Pre ni la Post. L'especificació no canvia
- Però apareixen a l'invariant. Cal dir què valen a cada iteració

Recursió:

- Les variables locals no serveixen. Es creen noves a cada crida
- Funció d'immersió d'eficiència, recursiva: Nous paràmetres d'entrada o de sortida
- S'han d'afegir a la Pre/Post!
- La funció desitjada no és recursiva, crida a la d'immersió

Immersions d'eficiència

Concepte d'immersió d'eficiència

- Font freqüent d'ineficiència: Repetir càlculs ja fets
- En programes iteratius: Guardar variables temporals que guarden resultats d'una iteració a la següent
- En programes recursius, una variable temporal és local *a cada crida recursiva*. No guarda resultats d'una crida a l'altra
- Immersió d'eficiència: Introducció de paràmetres o resultats addicionals per transmetre valors ja calculats en/a altres crides
- Pot haver de fer-se *a més* d'una immersió d'eficiència.

Exemple: suma dels k anteriors

```
// Pre: v.size() > k >= 0
// Post: retorna cert sii hi ha algun i entre k i
//    v.size()-1 tal que v[i] = v[i-k]+...+v[i-1]
bool kanteriors(const vector<double>& v, int k);
```

Exemple: suma dels *k* anteriors

```
bool kanteriors(const vector<double>& v, int k) {
    int i = k;
    while (i < v.size()) {
        // Inv: no hi ha cap j < i tal que
        //        v[j] = v[j-k]+...+v[j-1]
        if (v[i] == suma(v,i-k,i-1)) return true;
        ++i;
    }
    return false;
}
```

suma(v,i-k,i-1) cost k $\rightarrow$ cost total (n-k)$\cdot$k

Exemple: suma dels k anteriors

Millora: propagar la suma dels k anteriors

```
bool kanteriors(const vector<double>& v, int k) {
    double sum = 0;
    for (int j = 0; j < k; ++j) sum += v[j];
    int i = k;
    while (i < v.size()) {
       // Inv: no hi ha cap j < i tal que
       //      v[j] = v[j-k]+...+v[j-1]
       // i a mes sum = v[i-k]+...+v[i-1]
       if (v[i] == sum) return true;
       sum = sum - v[i-k] + v[i];
       ++i;
    }
    return false;
}
```

cost total proporcional a n, independent de k

Exemple: suma de *k* elements anteriors

```
// Pre: v.size() > k >= 0
// Post: retorna cert sii hi ha algun i entre k i
//    v.size()-1 tal que v[i] = v[i-k]+...+v[i-1]
bool kanteriors(const vector<double>& v, int k);
```

Primer, cal immersió d'especificació:

```
// Pre: v.size() >= m >= k >= 0
// Post: retorna cert sii hi ha algun i entre m i
//   v.size()-1 tal que v[i] = v[i-k]+...+v[i-1]
bool i_kanteriors(const vector<double>& v, int k, int m);
```

Exemple: suma de *k* elements anteriors

```
bool i_kanteriors(const vector<double>& v, int k, int m) {
    if (m == v.size())
        return false;
    else if (v[m] == suma(v,m-k,m-1))
        return true;
    else
        return i_kanteriors(v,k,m+1);
}
```

Problema: fem k sumes a cada crida

$\rightarrow$ cost total (n-k)·k

Exemple: suma de *k* elements anteriors

Immersió d'eficiència:

```
// Pre: v.size() >= m >= k >= 0 i a mes
//      sum = v[m-1]+...+v[m-k]
// Post: retorna cert sii hi ha algun i entre m i v.size()-1
//       tal que v[i] = v[i-k]+...+v[i-1]
bool ie_kanteriors(const vector<double>& v,
                         int k, int m, double sum);
```

Exemple: suma de k elements anteriors

```
// Pre: v.size() >= m >= k >= 0 i a mes
//      sum = v[m-1]+...+v[m-k]
bool ie_kanteriors(const vector<double>& v,
                         int k, int m, double sum) {
    if (m == v.size())
        return false;
    else if (v[m] == sum)
        return true;
    else return
        ie_kanteriors(v,k,m+1,sum+v[m]-v[m-k]);
}
// Post: retorna cert sii hi ha algun i entre m i v.size()-1
//       tal que v[i] = v[i-k]+...+v[i-1]
```

Crida inicial:

```
bool kanteriors(const vector<double>& v, int k) {
     return ie_kanteriors(v,k,k,suma(v,0,k-1));
}
```

Exemple: suma de k elements anteriors

Immersió d'eficiència alterativa: afegim la suma com a resultat, en comptes de com a paràmetre d'entrada

```
// Pre: v.size() >= m >= k >= 0
// Post: retorna cert sii hi ha algun i entre m i v.size()-1
//       tal que v[i] = v[i-k]+...+v[i-1], i a mes sum
//       conte la suma de v[m-k...m-1]
bool ie_kanteriors(const vector<double>& v,
                         int k, int m, double& sum);
```

Exercici: la implementació i la crida inicial.
Pista: cas base: m = v.size()

Exemple: element frontissa

Diem que en un vector un element és *frontissa* si és igual que la diferència entre els que el segueixen i els que el precedeixen

Exemples:
[1,3,11,6,5,4]
[2,1,1]
[1,2,1,0,4]

Exemple: element frontissa

```
// Pre: cert
// Post: retorna el nombre d'elements frontissa de v
int frontisses(const vector<double>& v);
```

Exemple: element frontissa

```
int frontisses(const vector<double>& v) {
   int i = 0;
   int n = 0;
   while (i < v.size()) {
      // Inv: 0 <= i <= v.size() i n = frontisses de v[0..i-1]
      if (v[i] == suma(v,i+1,v.size()-1) - suma(v,0,i-1)) ++n;
      ++i;
   }
   return n;
}
```

Suposem que suma(v,a,b) fa de l'ordre de b-a operacions
Llavors el cost és de l'ordre de v.size()^2 - quadràtic

Exemple: elements frontissa

Millora: reaprofitar sumes fetes

```
int frontisses(const vector<double>& v) {
    double sumapost = suma(v,1,v.size()-1);
    double sumaant = 0;
    int i = 0;
    while (i < v.size()) {
        // Inv: n = frontisses de v[0..i-1],
        // sumaant és la suma de v[0..i-1],
        // sumapost és la suma de v[i+1..v.size()-1]
        if (v[i] == sumapost-sumaant) ++n;
        sumaant += v[i];
        if (i < v.size()-1) sumapost -= v[i+1];
        ++i;
    }
    return n;
}
```

Cost lineal - de l'ordre de v.size()

Lleugera millora: mantenir directament sumapost-sumaant

Exemple: elements frontissa

```
// Pre: cert
// Post: retorna el nombre d'elements frontissa en v
int frontisses(const vector<double>& v);
```

Primer, cal immersió d'especificació:

```
// Pre: -1 <= i < v.size()
// Post: retorna el nombre d'elements frontissa en v[0..i]
int i_frontisses(const vector<double>& v, int i);
```

Exemple: elements frontissa

```
// Pre: -1 <= i < v.size()
// Post: retorna el nombre de frontisses en v[0..i]
int i_frontisses(const vector<double>& v, int i) {
    if (i == -1) return 0;
    else {
         int n = i_frontisses(v,i-1);
         if (v[i] == suma(v,i+1,v.size()-1)-suma(v,0,i-1)) ++n;
         return n;
    }
}
```

Problema: recàlcul de sumes - quadràtic

Exemple: elements frontissa

Immersió d'eficiència afegint paràmetres d'entrada:
Passem suma(posteriors), suma(anteriors) com a paràmetres

```
// Pre: -1 <= i < v.size(),
//      sumaant = suma(v[0..i-1]), sumapost = suma(v[i+1..v.size()-1]
// Post: retorna el nombre de frontisses en v[0..i]
int ie_frontisses(const vector<double>& v, int i,
                  double sumaant, double sumapost);
```

Crida inicial: `frontisses(v)` és

```
ie_frontisses(v,v.size()-1,suma(v,0,v.size()-2),0)
```

Implementació queda com a exercici
De l'ordre de v.size() operacions - lineal

Exemple: elements frontissa

Alternativa, immersió d'eficiència afegint resultats

```
// Pre: -1 <= i < v.size()
// Post: retorna el nombre de frontisses en v[0..i],
// sumaant = suma(v[0..i-1]), sumapost = suma(v[i+1..v.size()-1])
int ie_frontisses(const vector<double>& v, int i,
                  double& sumaant, double& sumapost));
```

Crida inicial:

```
int frontisses(const vector<double>& v) {
     double sa,sp;
     return ie_frontisses(v,v.size()-1,sa,sp);
}
```

Implementació queda com a exercici
De l'ordre de v.size() operacions - lineal

Arbre de mitjanes

Donat un arbre de doubles, construir-ne un altre de la mateixa forma que a cada node conté la mitjana dels valors del subarbre arrelat al node corresponent de l'original

```
// Pre: a = A
// Post: retorna l'arbre de mitjanes d'A
BinTree<double> arbre_mitjanes(const BinTree<double>& a);
```

Dificultat: la mitjana d'un arbre NO es pot calcular a partir de l'arrel i les mitjanes dels dos subarbres

Arbre de mitjanes: Solució ineficient

```
BinTree<double> arbre_mitjanes(const BinTree<double>& a) {
   if (a.empty()) {
       return BinTree<double>();
   } else {
       BinTree<double> b1, b2;
       double x = a.value();
       b1 = arbre_mitjanes(a.left());
       b2 = arbre_mitjanes(a.right());
       double s1 = sum(a.left()); double s2 = sum(a.right());
       int n1 = size(a.left()); int n2 = size(a.right());
       return BinTree<double>((x+s1+s2)/(1+n1+n2),b1,b2);
   }
}
```

Ineficient: Tres recorreguts de a.left() i a.right()
Cost quadràtic (penseu cas de arbre “linia”)

Arbre de mitjanes

Immersió d'eficiència: 1 recorregut que retorni a més suma i mida

```
// Pre: a = A
// Post: retorna l'arbre de mitjanes d'A, s conté
// la suma dels nodes d'A, i n conté el nombre de nodes d'A
BinTree<double> ie_arbre_mitjanes(const BinTree<double>& a,
                                  double& s, int& n);
```

Crida inicial:

```
BinTree<double> arbre_mitjanes(const BinTree<double>& a) {
    double s; int n;
    return ie_arbre_mitjanes(a,s,n);
}
```

Arbre de mitjanes

```
BinTree<double> ie_arbre_mitjanes(const BinTree<double>& a,
                                  double& s, int& n) {
   if (a.empty()) {
       s = 0; n = 0;
       return BinTree<double>();
   } else {
       BinTree<double> b1, b2;
       double s1, s2;
       int n1, n2;
       double x = a.value();
       b1 = arbre_mitjanes(a.left(),s1,n1);
       b2 = arbre_mitjanes(a.right(),s2,n2);
       s = s1 + s2;
       n = n1 + n2;
       return BinTree<double>((x+s)/(1+n),b1,b2);
   }
}
```

Cost lineal

Determinar si un arbre és equilibrat

Concepte important en estructures de dades avançades:

Un arbre és equilibrat si i només si

- els seus dos fills són equilibrats, i a més
- la diferència d'alçades dels subarbres fills no supera la unitat.

Implementació, I

```
// Pre: a = A
// Post: el valor retornat indica si A es un arbre equilibrat
bool equilibrat(const BinTree<int>& a);
```

Suposem que ja tenim implementada la funció

```
  // Pre: a = A
  // Post: el valor retornat es la longitud del camí més llarg
  //       de l'arrel a una fulla de l'arbre A
int alcaria(const BinTree<int>& a);
```

i que tarda temps proporcional a la mida de l'arbre

Implementació, II

```
// Pre: a = A
// Post: el valor retornat indica si A es un arbre equilibrat
bool equilibrat(const BinTree<int>& a) {
   if (a.es_buit()) return true;
   else {
      BinTree<int> a1, a2;
      bool b1 = equilibrat(a.left());
      bool b2 = equilibrat(a.right());
      int h1 = alcaria(a.left());
      int h2 = alcaria(a.right());
      return (abs(h1 - h2) <= 1) and b1 and b2;
   }
}
```

Pregunta: Fonts (plural) d'ineficiència?

Solució: immersió d'eficiència

Retornar més informació per evitar repetir càlculs:

```
   // Pre: a = A
   // Post: retorna un booleà que diu si A és equilibrat,
   // i si el booleà és true, h conté l'alçada d'A
bool i_equilibrat(const BinTree<int>& a, int& h);
```

Crida inicial:

```
bool equilibrat(const BinTree<int>& a) {
     int h;
     return ie_equilibrat(a,h);
}
```

Implementació de la funció d'immersió

```
bool ie_equilibrat(const BinTree<int>& a, int& h) {
   if (a.es_buit()) {
      h = 0;
      return true;
   }
   else {
      int h1, h2;
      if (not ie_equilibrat(a.left(),h1)) return false;
      if (not ie_equilibrat(a.right(),h2)) return false;
      return (abs(h1 - h2) <= 1);
   }
}
```

Clau de la justificació: Per hipòtesi d'inducció, si fem algun return false està ben retornat, i si no fem cap dels dos return false, h1 i h2 contenen les altures dels dos fills d'a, i el return que fem també és el resultat correcte

Eficiència: Consideracions generals

Concepte d'eficiència

(Informal; més rigorosament, a EDA)

Cost d'un algorisme: Funció de la mida de l'entrada

- En temps: "nombre d'operacions de l'ordre de ..."
- En memòria: "nombre de bytes de l'ordre de ..."

"de l'ordre de ..." = "amb una constant multiplicant"

Concepte d'eficiència

Exemple: programes que operen amb un vector de mida n, temps proporcional a:

- Cerca seqüencial: n
- Cerca dicotòmica: $\log_2 n$
- Ordenació per selecció o inserció: n^2
- Ordenació per barreja ("mergesort"): $n \log_2 n$
- Ordenació ràpida ("quicksort"): $n \log_2 n$

Diem "$O(f(n))$" per dir "$f(n)$ amb una constant multiplicant davant"

Més exemples d'eliminació de càlculs repetits

Funció exponencial

Sèrie de Taylor de l'exponencial

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^i}{i!} + \ldots \tag{1}$$

```
   // Pre: x > 0 i n >= 0
double exponencial(double x, int n);
   // Post: el valor retornat es la suma dels n primers termes
   //       de l'expansio en sèrie de Taylor de e^x
```

Implementació

```
   // Pre: x > 0 i n >= 0
double exponencial(double x, int n) {
   double e = 0;
   int i = 0;
   while (i < n) {
       e += potencia(x,i)/factorial(i);
       ++i;
   }
   return e;
}
   // Post: el valor retornat es la suma dels n primers termes
   //       de l'expansio en sèrie de Taylor de e^x

   // Inv: 0 <= i <= n,
   //      e conté la suma dels i primers termes de l'expansio
   //      en sèrie de Taylor de la funció exponencial
```

Implementació

Especificació de les dues funcions auxiliars:

```
double potencia(double x, int n);
   // Pre: x > 0 i n >= 0
   // Post: retorna x^n

int factorial(int n);
   // Pre: n >= 0
   // Post: retorna n!
```

Millora d'eficiència

Càlculs repetits tant al factorial com a la potència

Mantenir variable p = potencia(x,i) = x*potencia(x,i-1)

Mantenir variable f = factorial(i) = i*factorial(i-1)

Problema: x^i i $i!$ creixen molt ràpid
però $x^i/i!$ decreix

Millora alternativa

Sigui t_i el terme i-essim de la sèrie de Taylor

$$t_i = \frac{x^i}{i!} = \frac{x^{i-1} * x}{(i-1)! * i} = t_{i-1}\frac{x}{i}$$

$$t_0 = x^0/0! = 1$$

Implementació

```
double exponencial(double x, int n) {
   // Pre: x > 0; n >= 0
   double e = 0;
   double t = 1; // t conté x^0/(0!)
   int i = 0;
   // Inv: 0 <= i <= n
   //      t conté x^i/(i!);
   //      e conté la suma dels i primers termes de l’expansio
   //      en sèrie de Taylor de la funció exponencial
   while (i < n) {
       e += t;
       ++i;
       t = t*x/i;
   }
   return e;
   // Post: el valor retornat es la suma dels n primers termes
   //       de l’expansió en sèrie de Taylor de e^x
}
```

Exercici: Funció cosinus

Sèrie de Taylor del cosinus

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots \tag{2}$$

```
   // Pre: n >= 0
double cosinus(double x, int n);
   // Post: e conté la suma dels n primers termes de l'expansio en sèrie
   //       de Taylor del cosinus de x
}
```

Pista: $t_0 = \ldots \quad , \quad t_{i+1} = \ldots t_i \ldots$

Exemple: La successió de Fibonacci

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{si } n \geq 2 \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \ldots$$

Implementació recursiva

```
// Pre: n >= 0
// Post: retorna fib(n)
int fibonacci(int n) {
    if (n < 2) return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Quants cops cridem fibonacci(n-i) en executar fibonacci(n)?
Resposta: fib(i) cops (inducció!)

Cost temporal?

fib(n) creix com ϕ^n
(ϕ = raó àuria = solució de $(\phi = 1 + 1/\phi)$ = $\simeq 1.618033\ldots$)

Càlcul molt lent

Truc

Suposem que tenim el parell (f_{n-1}, f_{n-2})
Llavors, el parell (f_n, f_{n-1}) és

$$(f_n, f_{n-1}) = (f_{n-1} + f_{n-2}, f_{n-1})$$

Versió iterativa

```
// Pre: n >= 0
// Post: retorna fib(n)
int fibonacci(int n) {
   if (n <= 1) return n;
   else {
       int f1 = 1;
       int f2 = 0;
       int i = 2;
       // Inv: f1 = fib(i-1), f2 = fib(i-2), 2 <= i <= n
       while (i <= n) {
           int temp = f1;
           f1 = f1 + f2;
           f2 = temp;
           ++i;
       }
       return f1;
   }
}
```

Detecció de la repetició de càlculs en programes recursius

```
#include <utility>

// Pre: n > 0
// Post: retorna a "first" i "second" fib(n) i fib(n-1)
pair<int,int> i_fibonacci(int n);
```

Implementació funció d'immersió

```
// Pre: n > 0
// Post: retorna en "first" i "second"
//       els valors de fib(n) i fib(n-1)
pair<int,int> i_fibonacci(int n) {
   pair<int,int> p;
   if (n == 1) {
       p.first = 1;
       p.second = 0;
   } else {
       p = i_fibonacci(n - 1);
       // HI: p.first i p.second contenen fib(n-1) i fib(n-2)
       p = pair<int,int> (p.first + p.second, p.first);
   }
   return p;
}
```

Crida a la funció d'immersió

```
// Pre: n >= 0
// Post: retorna fib(n)
int fibonacci(int n) {
    if (n == 0) return 0;
    else return i_fibonacci(int n).first;
}
```

Alternativa

Funcions que retornen més d'un valor

$\rightarrow$ paràmetres per referència

```
// Pre: n > 0
// Post: retorna en f1 i f2 els valors de fib(n) i fib(n-1)
void i_fibonacci(int n, int& f1, int& f2);
```