Dotze algorismes fonamentals
Programació 1

Departament de Llenguatges i Sistemes Informàtics
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

10 d'abril de 2013

1. Producte ràpid

Enunciat: Doneu un algorisme que calculi el producte de dos naturals de manera ràpida, fent només sumes, restes, multiplicacions per 2, i divisions per 2.

Idea: Observem que, en qualsevol cas, és cert que

$$x * y = x * (y - 1) + x,$$

però que quan y és parell, a més tenim que

$$x * y = (2 * x) * (y/2).$$

Solució recursiva:

```
// Pre: y >= 0
// Post: retorna x*y
int prod(int x, int y) {
    if (y == 0) return 0;
    if (y%2 == 0) return prod(x*2,y/2);
    return prod(x,y - 1) + x;
}
```

Solució iterativa:

```
// Pre: y >= 0
// Post: retorna x*y
int prod(int x, int y) {
   int p = 0;
   // Inv: X*Y = p + x*y on X, Y son els valors inicials de x, y
   while (y != 0) {
       if (y%2 == 0) {
             x = x*2;
             y = y/2;
       }
       else {
             p = p + x;
             y = y - 1;
       }
   }
   return p;
}
```

Observació: La mateixa idea es pot fer servir per al càlcul de potències (`x` elevat a `y` quan y no és negatiu). Naturalment, si totes les variables són `int`'s, només podreu provar el programa amb valors relativament petits abans que es produeixi sobreeiximent.

2. Cerca dicotòmica

Enunciat: Feu una funció

```
int posicio(double x, const vector<double>& v, int esq, int dre);
```

que retorni la posició on es troba `x` dins del subvector `v[esq..dre]`. Si `x` no apareix a `v[esq..dre]` o `esq > dre`, cal retornar `-1`.
Podem suposar que

- El vector v està ordenat creixentment.
- Inicialment `(0 <= esq) and (dre < v.size())`.

Nota: En canvi, `esq <= dre` pot no ser cert ni al principi: Penseu quina crida faríeu per ordenar una taula de mida 0.

Solució recursiva:

```
// Pre: (0 <= esq) and (dre < v.size()) and (v esta ordenat creixentment)
// Post: retorna i tal que, o be esq<=i<=dre i v[i]==x, o be i==-1 i x no es a v[esq..dre]
int posicio(double x, const vector<double>& v, int esq, int dre) {
    if (esq > dre) return -1;
    int pos = (esq + dre)/2;                // posicio central de v[esq..dre]
    if (x < v[pos]) return posicio(x, v, esq, pos - 1);
    if (x > v[pos]) return posicio(x, v, pos + 1, dre);
    return pos;
}
```

Solució iterativa:

```
// Pre: (0 <= esq) and (dre < v.size()) and (v esta ordenat creixentment)
// Post: retorna i tal que, o be esq<=i<=dre i v[i]==x, o be i==-1 i x no es a v[esq..dre]
int posicio(double x, const vector<double>& v, int esq, int dre) {
    int pos;
    bool trobat = false;
       // Inv: (0 <= esq), (dre < v.size()),
       // si x es a v[E..D], llavors es a v[esq..dre]
       // on E i D son els valors inicials d'esq i dre,
       // i a mes trobat indica que hem trobat x a pos
    while (not trobat and esq <= dre) {
        pos = (esq + dre)/2;                // posicio central de v[esq..dre]
        if (x < v[pos]) dre = pos - 1;
        else if (x > v[pos]) esq = pos + 1;
        else trobat = true;
    }
    if (trobat) return pos;
    else return -1;
}
```

3. Ordenació per inserció

Enunciat: Feu un procediment

```
void ordena_per_insercio(vector<double>& v);
```

que ordeni `v` de petit a gran utilitzant l'algorisme d'ordenació per inserció.

Solució:

```
// Pre: cap
// Post: v conte els elements inicials i esta ordenat creixentment
void ordena_per_insercio(vector<double>& v) {
    // Inv: v[0..i-1] esta ordenat creixentment
    for (int i = 1; i < v.size(); ++i) {
        double x = v[i];
        int j = i;
        while (j > 0 and v[j - 1] > x) {
            v[j] = v[j - 1];
            --j;
        }
        v[j] = x;
    }
}
```

4. Ordenació per selecció

Enunciat: Feu un procediment

```
void ordena_per_seleccio(vector<double>& v, int m);
```

que ordeni `v[0..m]` de petit a gran utilitzant l'algorisme d'ordenació per selecció. La resta de `v` no s'ha de modificar. Sabem que $0 \leq$ `m` $<$ `v.size()`.

Solució:

```
// Pre: cap
// Post: els valors d'a i b estan intercanviats respecte als inicials
void intercanvia(double& a, double& b) {
    double c = a;
    a = b;
    b = c;
}

// Pre: 0 <= m < v.size()
// Post: retorna i tal que 0<=i<=m i v[i] es maxim en v[0..m]
int posicio_maxim(const vector<double>& v, int m) {
    int pos = 0;
    // Inv: 0 <= pos < i, i v[pos] >= v[j] per a tot j en [0..i-1]
    for (int i = 1; i <= m; ++i) {
        if (v[i] > v[pos]) pos = i;
    }
    return pos;
}

// Pre: 0 <= m < v.size()
// Post: v[0..m] conte els elements inicials, pero ordenats
//       creixentment i la resta de v no ha estat modificada
// Versio recursiva
void ordena_per_seleccio(vector<double>& v, int m) {
    if (m > 0) {
        int k = posicio_maxim(v, m);
        intercanvia(v[k], v[m]);
        ordena_per_seleccio(v, m - 1);
    }
}
```

Solució alternativa:

Una versió iterativa de `ordena_per_seleccio` es:

```
// Pre: 0 <= m < v.size()
// Post: v[0..m] conte els elements inicials, pero ordenats
//       creixentment i la resta de v no ha estat modificada
// Versio iterativa
void ordena_per_seleccio_iter(vector<double>& v, int m) {
    // Inv: v[i+1..m] esta ordenat i els elements de v[i+1..m]
    // son tots mes grans que els de v[0..i]
```

```
    for (int i = m; i > 0; --i) {
        int k = posicio_maxim(v, i);
        intercanvia(v[k], v[i]);
    }
}
```

5. Ordenació per fusió

Enunciat: Feu un procediment

```
void ordena_per_fusio(vector<double>& v, int e, int d);
```

que ordeni `v[e..d]` de petit a gran utilitzant l'algorisme d'ordenació per fusió. La resta de `v` no s'ha de modificar. Sabem que $0 \le \texttt{e} \le \texttt{d} < \texttt{v.size()}$.

Solució:

```
// Pre: 0<=e<=m<=d<v.size() i v[e..m] i v[m+1..d], per separat, son ordenats creixentment
// Post: els elements de v[e..d] son els inicials, pero ordenats creixentment
//       i la resta de v no ha canviat
void fusiona(vector<double>& v, int e, int m, int d) {
    int n = d - e + 1;
    vector<double> aux(n);
    int i = e;
    int j = m + 1;
    int k = 0;
    // Inv: aux[0..k-1] es la fusio de v[e..i-1] i v[m+1..j-1]
    while (i <= m and j <= d) {
        if (v[i] <= v[j]) {
            aux[k] = v[i];
            ++i;
        }
        else {
            aux[k] = v[j];
            ++j;
        }
        ++k;
    }
    while (i <= m) {
        aux[k] = v[i];
        ++k;
        ++i;
    }
    while (j <= d) {
        aux[k] = v[j];
        ++k;
        ++j;
    }
    for (k = 0; k < n; ++k) v[k + e] = aux[k];
}


// Pre: 0<=e<=d<v.size()
// Post: els elements de v[e..d] son els inicials, pero ordenats creixentment
void ordena_per_fusio(vector<double>& v, int e, int d) {
    if (e < d) {
        int m = (e + d)/2;
        ordena_per_fusio(v, e, m);
```

```
        ordena_per_fusio(v, m + 1, d);
        fusiona(v, e, m, d);
    }
}
```

Observació: Aquest és dels pocs algorismes vistos a l'assignatura per als quals no és fàcil donar una versió sense recursivitat. Veureu com es faria iterativament a assignatures posteriors.

6. Cerca d'un string dins d'un altre

Enunciat:

Feu una funció que, donats dos strings x i y, retorni la posició on comença la primera ocurrència de l'string x dins de y. Si x no és substring de y, la funció ha de retornar -1. Per exemple, la funció ha de retornar 10 si x és `TATC` i y és `CGTAGATCTATATCGCTAACGGACTAACT` i ha de retornar -1 si x és `TATGC` i y és `CGTAGATCTATATCGCTAACGGACTAACT` .

Solució:

```
// Pre: cap
// Post: o be 0<=i<y.size() i x==y[i..i+x.size()-1] i x no apareix abans a y,
//       o be i==-1 i x no apareix enlloc dins de y
int posicio (const string& x, const string& y) {
   int i = 0;
   bool trobat = (x.size() == 0);
   int n = int(y.size()) - int(x.size());
   // Inv: no hi ha cap ocurrencia de x en y que
   // comenci en posicions 0..i-1
   while (not trobat and i <= n) {
       int j = 0;
       bool encaixa = true;
       // Inv: ..., i a mes x[0..j-1] == y[i..i+j-1]
       while (encaixa and j < x.size()) {
           encaixa = (x[j] == y[i+j]);
           ++j;
       }
       if (encaixa) trobat = true;
       else ++i;
   }
   if (trobat) return i;
   else {
       i = -1;
       return i;
   }
}
```

Solució alternativa:

Una solució que, segons els gustos, pot ser més elegant:

```
// Pre: cap
// Post: o be 0<=i<y.size() i x==y[i..i+x.size()-1],
//       o be i==-1 i x no apareix enlloc dins de y
int posicio (const string& x, const string& y) {
    int i,j;
    i=j=0;
    int n = int(y.size()) - int(x.size());
    // Inv: no hi ha cap ocurrencia de x en y
    // que comenci en posicions 0..i-1,
    // i a mes x[0..j-1] == y[i..i+j-1]
    while (j < x.size() and i <= n) {
```

```
        if (x[j] == y[i+j]) ++j;
        else {
             j=0;
             ++i;
        }
    }
    if (j == x.size()) return i;
    else {
        i = -1;
        return i;
    }
}
```

Observació: Hi ha altres algorismes que poden arribar a ser molts més eficients que aquest bàsic, al menys en alguns casos. Els podreu trobar en assignatures posteriors.

7. Producte de polinomis

Enunciat: Doneu una funció

```
Poli producte(const Poli& p, const Poli& q);
```

que calcula el producte de dos polinomis donats. El polinomi es representa pel seu grau i un vector: el coeficient de grau i apareix en la i-èssima posició del vector:

```
typedef vector<double> Coef;

struct Poli {
    int grau;
    Coef coefs;
};
```

Solució:

```
// Pre: 0<=p.grau<p.coefs.size(), 0<=q.grau<q.coefs.size()
// Post: retorna el polinomi p*q, amb el mateix conveni de representacio
Poli producte(const Poli& p, const Poli& q) {
     int n = p.grau + q.grau;
     Poli r;
     r.grau = n;
     Coef c(n+1,0);
     r.coefs = c;
     for (int i = 0; i <= p.grau; ++i) {
         for (int j = 0; j <= q.grau; ++j) {
             r.coefs[i+j] = r.coefs[i+j] + p.coefs[i]*q.coefs[j];
         }
     }
     return r;
}
```

8. Avaluació d'un polinomi en un punt

Enunciat: Doneu un algorisme que avalua un polinomi donat en un punt donat. El polinomi es representa pel seu grau i un vector: el coeficient de grau i apareix en la i-èssima posició del vector:

```
typedef vector<double> Coef;

struct Poli {
    int grau;
    Coef coefs;
};
```

Feu servir la regla de Horner, que es basa en el fet que

$$c_n x^n + \ldots c_1 x^1 + c_0 = (c_n x^{n-1} + \cdots + c_1) \cdot x + c_0$$

Solució:

```
// Pre: 0<=poli.grau<poli.coefs.size()
// Post: retorna el valor del polinomi poli en x
//       es a dir, suma(poli.coefs[i]*x^i,i=0..poli.grau)
double horner(const Poli& poli, double x) {
    double aval = 0.;
    // Inv: aval = poli[poli.grau...i+1](x)
    for (int i = poli.grau; i >= 0; --i) {
        aval = aval*x + poli.coefs[i];
    }
    return aval;
}
```

Solució alternativa:

Una alternativa a la funció `horner` és la següent funció `avalpoli`. Té l'inconvenient que fa el doble d'operacions producte que la solució basada en la regla de Horner.

```
// Pre: 0<=poli.grau<poli.coefs.size()
// Post: retorna el valor del polinomi poli en x
//       es a dir, suma(poli.coefs[i]*x^i,i=0..poli.grau)
double avalpoli(const Poli& poli, double x) {
    double aval = 0.;
    double pot = 1.;
    // Inv: aval = poli[i-1...0](x) and pot = x^i
    for (int i = 0; i <= poli.grau; ++i) {
        aval = aval + pot*poli.coefs[i];
        pot = pot*x;
    }
    return aval;
}
```

9. Suma de vectors dispersos

Enunciat: Quan un vector d'enters conté molts elements 0, és convenient per estalviar memòria i temps representar-lo en forma comprimida, també anomenada *dispersa*. En aquesta representació, guardem un parell (i, x) si la component i-èssima del vector és x i $x \neq 0$. A més, per afavorir les operacions amb el vector, guardem aquests parells ordenats per i.

Per exemple, el vector $[0, 0, 0, 8, 0, 6, 0, 0, 0, -4, 0]$ es representaria com $[(3, 8), (5, 6), (9, -4)]$.

Si definim un parell així,

```
struct Parell {
        int valor;        // Qualsevol valor
        int pos;          // Ha de ser mes gran o igual que zero
};
```

doneu una funció

```
vector<Parell> suma_dispersa(const vector<Parell>& v1, const vector<Parell>& v2)
```

que sumi dos vectors donats en forma dispersa i en retorni el resultat en forma dispersa, amb el nombre mínim de components.

Solució:

```
// Pre: v1, v2 son la representacio comprimida de dos vectors
// Post: retorna el vector representacio comprimida de v1+v2, de mida
//       exactament igual que el nombre de components no 0 de v1+v2
vector<Parell> suma_dispersa(const vector<Parell>& v1, const vector<Parell>& v2) {
   int n1 = v1.size();
   int n2 = v2.size();
   vector<Parell> v3(n1 + n2);
   int i,j,k;
   i = j = k = 0;
   // Inv: v3[0..k-1] conté la representació comprimida de v1[0..i-1]+v2[0..j-1]
   while (i < n1 and j < n2) {
       if (v1[i].pos < v2[j].pos) {
          v3[k] = v1[i];
          ++i; ++k;
       } else if (v1[i].pos > v2[j].pos) {
          v3[k] = v2[j];
          ++j; ++k;
       } else if (v1[i].valor + v2[j].valor != 0) {
          v3[k].pos = v1[i].pos;
          v3[k].valor = v1[i].valor + v2[j].valor;
          ++i; ++j; ++k;
       } else {
          ++i; ++j;
       }
   }
   // copiem a v3 la part restant que resta de v1, si v2 s'ha acabat
   while (i < n1) {
       v3[k] = v1[i];
```

```
        ++i; ++k;
    }
    // copiem a v3 la part restant que resta de v2, si v1 s'ha acabat
    while (j < n2) {
        v3[k] = v2[j];
        ++j; ++k;
    }
    // creem un vector de la mida justa i hi copiem la part emprada de v3
    vector<Parell> resultat(k);
    for (int m = 0; m < k; ++m) resultat[m] = v3[m];
    return resultat;
}
```

10. Producte de matrius

Enunciat: Dues matrius a i b són compatibles per al producte si hi ha naturals no nuls m, k i n tals que a és $m \times k$ i b és $k \times n$. En aquest cas, $a * b$ és una matriu $m \times n$.

Feu una funció que donades dues matrius compatibles per al producte retorni la seva matriu producte.

Solució:

```
// Suposem que una Matriu es defineix aixi
typedef vector<int> Fila;
typedef vector<Fila> Matriu;


// Pre: a i b son compatibles per al producte
// Post: p = a*b
void producte(const Matriu& a, const Matriu& b, Matriu& p) {
    p = Matriu(a.size(),Fila(b[0].size(),0));
    for (int i = 0; i < a.size(); ++i) {
        for (int j = 0; j < b[0].size(); ++j) {
            for (int k = 0; k < b.size(); ++k) {
                p[i][j] = p[i][j] + a[i][k]*b[k][j];
            }
        }
    }
}
```

11. Avaluació de sèries: aproximació del número π

Enunciat: Els primers dígits del número irracional π són $3.14159265358979323846\ldots$. Es coneixen moltes expressions per a pi π en forma de sumes infinites de números racionals. Per exemple,

$$\frac{\pi}{2} = 1 + \frac{1}{3} + \frac{1 \cdot 2}{3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7} + \ldots$$

Doneu un programa que, donat $n > 0$, calculi aproximacions a π sumant els n primers termes d'aquesta sèrie.

Solució:

```
#include <iostream>
using namespace std;

int main(){
    int n;
    double migpi = 0.0;
    double terme = 1.0;
    cin >> n;
    // Inv: migpi es la suma dels i-1 primers termes de la serie,
    // i terme es l'i-essim terme de la serie
    for (int i = 1; i <= n; ++i) {
        migpi = migpi + terme;
        terme = terme*i/(2.0*i + 1.0);
     }
    cout << 2*migpi << endl;
}
```

Observació: Fixeu-vos que, en general, no és el mateix sumar els primers n termes d'una sèrie per a π que obtenir els n primers dígits de π.

En aquest cas particular és fàcil fitar l'error d'aproximació quan deixem de sumar termes: Si l'aritmètica fos exacta, a cada iteració se satisfaria que `migpi` $\leq \pi/2 \leq$ `migpi` $+$ `terme`. Per tant, si volem d dígits correctes n'hi ha prou sumar fins que `2*terme` $< 10^{-d}$.

12. Cerca d'una arrel d'una funció continua

Enunciat: Suposeu que

```
double f(double x);
```

és una funció continua. Sabem, pel Teorema de Bolzano, que si $f(a)$ i $f(b)$ tenen signes diferents, llavors f té una arrel en l'interval $[a, b]$. Feu servir aquesta propietat per donar una funció

```
double arrel(double a, double b, double epsilon);
```

que retorna un double `c` en l'interval `[a,b]` tal que hi ha una arrel de `f` en `[c,c+epsilon]`.

Solució

```
// Pre: f es continua, epsilon > 0, a < b i f(a)*f(b) < 0
// Post: retorna c tal que c es a [a,b] i f te una arrel a [c,c + epsilon]
double arrel(double a, double b, double epsilon) {
    if (b - a <= epsilon) return a;
    double c = (a + b)/2;
    if (f(a)*f(c) <= 0) return arrel(a,c,epsilon);
    return arrel(c,b,epsilon);
}
```

Solució alternativa: Una versió iterativa de la funció `arrel` és

```
// Pre: f es continua, epsilon > 0, a < b i f(a)*f(b) < 0
// Post: retorna c tal que c es a [a,b] i f te una arrel a [c,c + epsilon]
double arrel(double a, double b, double epsilon) {
    // Inv: f(a)*f(b) <= 0
    while (b - a > epsilon) {
        double c = (a + b)/2;
        if (f(a)*f(c) <= 0) b = c;
        else a = c;
    }
    return a;
}
```

Observació 1: Sovint l'avaluació de `f` és costosa i convé minimitzar el nombre de crides a `f`. Es poden optimitzar les versions anteriors mantenint els valors ja calculats de `f(a)`, `f(b)` i `f(c)` en paràmetres addicionals (versió recursiva) o en variables auxiliars (versió iterativa). Es deixa com a exercici.

Observació 2: Una altra possiblitat és triar `c` no com el punt mig d'`a` i `b`, sinó com el punt on `f` seria 0 si fos una recta que passés per $(a, f(a))$ i $(b, f(b))$. Quan `f` efectivament s'assembla molt a una recta, aquesta versió pot fer moltes menys iteracions. S'anomena *cerca per interpolació* i es deixa com a exercici. Cal vigilar el cas en què $f(c) = 0$ durant el càlcul.