Millores d'eficència en iteració
Programació 2
Facultat d'Informàtica d'Informàtica, UPC

Professorat de PRO2

Tardor 2022

- Col·laboracions (en ordre alfabètic): Juan Luis Esteban, Ricard Gavaldà, Conrado Martínez, Fernando Orejas
- Aquestes transparències **no** substitueixen els apunts de l'assignatura, els complementen

Part I

Cost dels algorismes

- Temps: nombre d'operacions (respecte a la mida de les dades)
  - en llistes, vectors, arbres: nombre de elements
- Espai: memòria utilitzada (respecte a la mida de les dades)
- Cas pitjor, cas millor, cas mitjà (inserció vs. selecció)

Exemples de cost

- Constant: obtenir mida d'un vector, pila, cua, llista (amb `size()`)

Exemples de cost

- Constant: obtenir mida d'un vector, pila, cua, llista (amb `size()`)
- Constant: obtenir element d'un vector, cim de una pila, primer d'una cua, arrel d'un arbre, element d'una llista mitjançant iterador.

Exemples de cost

- Constant: obtenir mida d'un vector, pila, cua, llista (amb `size()`)
- Constant: obtenir element d'un vector, cim de una pila, primer d'una cua, arrel d'un arbre, element d'una llista mitjançant iterador.
- Logarítmic: cerca dicotòmica.

Exemples de cost

- Constant: obtenir mida d'un vector, pila, cua, llista (amb `size()`)
- Constant: obtenir element d'un vector, cim de una pila, primer d'una cua, arrel d'un arbre, element d'una llista mitjançant iterador.
- Logarítmic: cerca dicotòmica.
- Linial: suma dels elements d'una pila, un vector, etc.; cerques simples; assignació o còpia de vectors, llistes, etc. de tipus simples.

Exemples de cost

- Constant: obtenir mida d'un vector, pila, cua, llista (amb `size()`)
- Constant: obtenir element d'un vector, cim de una pila, primer d'una cua, arrel d'un arbre, element d'una llista mitjançant iterador.
- Logarítmic: cerca dicotòmica.
- Linial: suma dels elements d'una pila, un vector, etc.; cerques simples; assignació o còpia de vectors, llistes, etc. de tipus simples.
- Entre linial i quadràtic: Mergesort, Heapsort. Quicksort.

Exemples de cost

- Constant: obtenir mida d'un vector, pila, cua, llista (amb `size()`)
- Constant: obtenir element d'un vector, cim de una pila, primer d'una cua, arrel d'un arbre, element d'una llista mitjançant iterador.
- Logarítmic: cerca dicotòmica.
- Linial: suma dels elements d'una pila, un vector, etc.; cerques simples; assignació o còpia de vectors, llistes, etc. de tipus simples.
- Entre linial i quadràtic: Mergesort, Heapsort. Quicksort.
- Quadràtic: algorismes simples d'ordenació de vectors (inserció, selecció), veure si una estructura seqüencial té tots els seus elements diferents (sense ordenar-los).

Exemples de cost

- Constant: obtenir mida d'un vector, pila, cua, llista (amb `size()`)
- Constant: obtenir element d'un vector, cim de una pila, primer d'una cua, arrel d'un arbre, element d'una llista mitjançant iterador.
- Logarítmic: cerca dicotòmica.
- Linial: suma dels elements d'una pila, un vector, etc.; cerques simples; assignació o còpia de vectors, llistes, etc. de tipus simples.
- Entre linial i quadràtic: Mergesort, Heapsort. Quicksort.
- Quadràtic: algorismes simples d'ordenació de vectors (inserció, selecció), veure si una estructura seqüencial té tots els seus elements diferents (sense ordenar-los).
- Exponencial. General combinacions, permutacions, etc

Part I

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
- Deduir els seus valors inicials a partir de l'invariant
- Actualitzar els seus valors a cada iteració a partir de l'invariant

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)` té cost proporcional a $k \rightarrow$ cost total proporcional $(n-k) \cdot k$

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 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: 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 nf = 0;
    while (i < v.size()) {
      // Inv: 0 ≤ i ≤ v.size()
      // i nf = nombre d'elements frontissa a v[0..i − 1]
       if (v[i] == suma(v,i+1,v.size()-1) - suma(v,0,i-1)) ++nf;
       ++i;
    }
    return nf;
}
```

Exemple: element frontissa

```
int frontisses(const vector<double>& v) {
    int i = 0;
    int nf = 0;
    while (i < v.size()) {
      // Inv: 0 ≤ i ≤ v.size()
      // i nf = nombre d'elements frontissa a v[0..i − 1]
       if (v[i] == suma(v,i+1,v.size()-1) - suma(v,0,i-1)) ++nf;
       ++i;
    }
    return nf;
}
```

Com que `suma(v,a,b)` té cost proporcional a $b - a$, el cost d'aquesta funció és proporcional a $(v.\texttt{size}())^2$ - 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; int nf = 0;
  while (i < v.size()) {
    // Inv: 0 ≤ i ≤ v.size() i
    //      nf = nombre de frontisses a v[0..i − 1]
    //      i sumaant és la suma de v[0..i − 1],
    //      i sumapost és la suma de v[i + 1..v.size() − 1]
    if (v[i] == sumapost-sumaant) ++nf;
    sumaant += v[i];
    if (i < v.size()-1) sumapost -= v[i+1];
    ++i;
  }
  return nf;
}
```

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; int nf = 0;
  while (i < v.size()) {
    // Inv: 0 ≤ i ≤ v.size() i
    //      nf = nombre de frontisses a v[0..i − 1]
    //      i sumaant és la suma de v[0..i − 1],
    //      i sumapost és la suma de v[i + 1..v.size() − 1]
    if (v[i] == sumapost-sumaant) ++nf;
    sumaant += v[i];
    if (i < v.size()-1) sumapost -= v[i+1];
    ++i;
  }
  return nf;
}
```

Cost lineal - de l'ordre de $v.\mathtt{size}()$

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; int nf = 0;
  while (i < v.size()) {
    // Inv: 0 ≤ i ≤ v.size() i
    //      nf = nombre de frontisses a v[0..i − 1]
    //      i sumaant és la suma de v[0..i − 1],
    //      i sumapost és la suma de v[i + 1..v.size() − 1]
    if (v[i] == sumapost-sumaant) ++nf;
    sumaant += v[i];
    if (i < v.size()-1) sumapost -= v[i+1];
    ++i;
  }
  return nf;
}
```

Cost lineal - de l'ordre de $v.\texttt{size}()$

Lleugera m

Exemple: element dominador més avançat d'una llista

- Diem que un element d'una llista de nombres 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.

Exemple: element dominador més avançat d'una llista

- Diem que un element d'una llista de nombres 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`.

Exemple: element dominador més avançat d'una llista

- Diem que un element d'una llista de nombres 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.

Exemple: element dominador més avançat d'una llista

```
// Pre: L no és buida i està formada per naturals
// Post: El resultat és el dominador més avancat de L
int dom(const list<int> & L);
```

Element dominador més avançat d'una llista

```
iint dom(const list<int> & L) {
   list<int>::const_iterator it= L.begin();
        int s=(*it);
        ++it;
        /* Inv: s és l'element dominador més avançat de L[:it) */
        while (it != L.end()){
                if (suma(L,it)<=(*it)) s=(*it);
                ++it;
        }
        return s;
}
```

Element dominador més avançat d'una llista

```
iint dom(const list<int> & L) {
   list<int>::const_iterator it= L.begin();
        int s=(*it);
        ++it;
        /* Inv: s és l'element dominador més avançat de L[: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.

Element dominador més avançat d'una llista

```
iint dom(const list<int> & L) {
   list<int>::const_iterator it= L.begin();
        int s=(*it);
        ++it;
        /* Inv: s és l'element dominador més avançat de L[: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.

Element dominador més avançat d'una llista

No cal fer cada vegada totes les sumes dels elements anteriors als visitats.

Element dominador més avançat d'una llista

No cal fer cada vegada totes les sumes dels elements anteriors als visitats.
Introduïm una variable nova que contingui la suma necessària a cada moment.

Element dominador més avançat d'una llista

No cal 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.

Element dominador més avançat d'una llista

```
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[:it)
  // aux és la suma dels elements d'L[:it)
  while (it != L.end()){
    if (aux<=(*it)) s=(*it);
      aux+=(*it);
      ++it;
    }
  return s;
}
```

Element dominador més avançat d'una llista

```
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[:it)
  // aux és la suma dels elements d'L[: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 a la mida d'L.

Exemple: 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 mitjana dels anteriors.

Exemple: 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 mitjana dels anteriors.
- Donat un vector no buit d'enters, volem saber si és mitjanament ordenat

Exemple: 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 mitjana dels anteriors.
- Donat un vector no buit d'enters, volem saber si és mitjanament ordenat
- Apliquem un esquema de cerca.

Exemple: vector mitjanament ordenat

```
bool mitjord(const vector<int> &v)
/* Pre: v.size()> 0 */
/* Post: El resultat indica si v está mitjanament ordenat */
```

Exemple: vector mitjanament ordenat

```
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;
}
```

Exemple: vector mitjanament ordenat

```
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;
}
```

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`.

Exemple: vector mitjanament ordenat

```
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;
}
```

Exemple: vector mitjanament ordenat

```
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 a la mida d'v.

Exemple complet de justificació

Inicialització: Comencem per `i=1` perque el primer element no afecta a la propietat d'estar mitjanament ordenat. Així es satisfà `1<= i <=v.size()` i també `v[0..i-1] està mitjanament ordenat` donat que un vector amb un element `v[0]` està mitjanament ordenat (per definició). Si posem `b` a cert l'antecedent de `si not b llavors v[0..i] no està mitjanament ordenat` és fals i qualsevol implicació que tingui com a premisa fals sempre es compleix (per definició). Com `s` és la suma de `v[0..i-1]` i `i=1 s` ha de ser `v[0]`.

Exemple complet de justificació

Condició de sortida: Es pot sortir del bucle per dues raons:

- Si `i` arriba a ser `v.size()` vol dir, per l'invariant, que hem explorat tot el vector `v` i que està mitjanament ordenat. Per complir la postcondició `b` ha de ser cert, el que és correcte donat que `b` comença sent cert i no ho hem modificat
- Si sortim quan `i<v.size()` s'ha de complir que `b` es fals. Per que això passi, el invariant ens diu que hem trobat una posició `i`, que és una posició vàlida del vector perque és menor que `i<v.size()`, tal que `v[0..i]` no està mitjanament ordenat. En aquest cas `b` té el valor correcte i compleix la postcondició.

Exemple complet de justificació

Cos del bucle: Per l'invariant sabem que `v[0..i-1] està mitjanament ordenat` i per la condició d'entrada sabem que `i<v.size()` marca una posició vàlida del vector i que `b` és cert. Ara hem de tractar `v[i]`.

Exemple complet de justificació

Cos del bucle (cont): Si `v[i]` és més gran que la mitjana de `v[0..i-1]` que es pot calcular a partir de `s/i` sabem ara que `v[0..i]` está mitajanament ordenat, `b` ha de continuar sent cert, i actualitzem `s` sumant-li `v[i]` amb la qual cosa `s` conté la suma de `v[0..i]`. Si li sumen 1 a `i` es torna a complir l'invariant perque torna a ser cert `v[0..i-1] està mitjanament ordenat`. També `b` és cert, per tant `si not b llavors v[0..i] no està mitjanament ordenat` es cert com abans perque l'antecedent és fals i per tant la implicació és certa. Torna a ser cert que `s= suma v[0..i-1]`. A més si abans `i` era menor que `v.size()` ara `i <=v.size()`.

Exemple complet de justificació

Cos del bucle (cont): Si `v[i]` no és més gran que la mitjana de `v[0..i]` que es calcula amb `s/i`, donant el valor fals a `b` indiquem que `v[0..i]` no està mitjanament ordenat. La resta de les propietats de l'invariant no canvien doncs no es modifica `i`.

Exemple complet de justificació

Cos del bucle (cont): Si `v[i]` no és més gran que la mitjana de `v[0..i]` que es calcula amb `s/i`, donant el valor fals a `b` indiquem que `v[0..i]` no està mitjanament ordenat. La resta de les propietats de l'invariant no canvien doncs no es modifica `i`.

Finalització Si triem com a funció de fita `v.size()-i` veiem que decreix en cada iteració...

Exemple complet de justificació

Cos del bucle (cont): Si `v[i]` no és més gran que la mitjana de `v[0..i]` que es calcula amb `s/i`, donant el valor fals a `b` indiquem que `v[0..i]` no està mitjanament ordenat. La resta de les propietats de l'invariant no canvien doncs no es modifica `i`.

Finalització Si triem com a funció de fita `v.size()-i` veiem que decreix en cada iteració... excepte si es compleix `(v[i]<double(s)/i)` perque no modifiquem l'`i`. Però és veu fàcilment que sortim del bucle.

Exemple complet de justificació

Cos del bucle (cont): Si `v[i]` no és més gran que la mitjana de `v[0..i]` que es calcula amb `s/i`, donant el valor fals a `b` indiquem que `v[0..i]` no està mitjanament ordenat. La resta de les propietats de l'invariant no canvien doncs no es modifica `i`.

Finalització Si triem com a funció de fita `v.size()-i` veiem que decreix en cada iteració... excepte si es compleix `(v[i]<double(s)/i)` perque no modifiquem l'`i`. Però és veu fàcilment que sortim del bucle.
Podem triar com a funció de fita `v.size()-i+b` (on cert equival a 1 i fals a 0) i és fàcil de veure que aquesta funció decreix en cada iteració.

Part I

Funció exponencial

Sèrie de Taylor de l'exponencial

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

```
// Pre: x > 0 i n ≥ 0
// Post: el valor retornat es la suma dels n primers termes
//       de l'expansio en sèrie de Taylor de e^x, és a dir,
//       retorna sum_{k=0}^{n-1} x^k/k!
double exponencial(double x, int n);
```

Implementació

```
double exponencial(double x, int n) {
    double e = 0;
    int i = 0;
    while (i < n) {
        e += potencia(x,i)/factorial(i);
        ++i;
    }
    return e;
}
```

L'invariant és:

$$0 \leq i \leq n \wedge e = \sum_{k=0}^{i-1} \frac{x^k}{k!}$$

Implementació

Especificació de les dues funcions auxiliars:

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

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

Millora d'eficiència

Càlculs repetits tant al factorial com a la potència

Mantenir variable $p = \texttt{potencia}(x, i)$

Millora d'eficiència

Càlculs repetits tant al factorial com a la potència

Mantenir variable $p = \texttt{potencia}(x, i)$ = $x * \texttt{potencia}(x, i - 1)$

Mantenir variable $f = \texttt{factorial}(i)$

Millora d'eficiència

Càlculs repetits tant al factorial com a la potència

Mantenir variable $p = \texttt{potencia}(x, i)$ = $x * \texttt{potencia}(x, i-1)$

Mantenir variable $f = \texttt{factorial}(i)$ = $i * \texttt{factorial}(i-1)$

Millora d'eficiència

Càlculs repetits tant al factorial com a la potència

Mantenir variable $p = \texttt{potencia}(x, i)$ = $x * \texttt{potencia}(x, i-1)$

Mantenir variable $f = \texttt{factorial}(i)$ = $i * \texttt{factorial}(i-1)$

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

$$t_i = \frac{x^i}{i!}$$

Millora alternativa

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

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

Millora alternativa

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

$$t_i = \frac{x^i}{i!} = \frac{x^{i-1} * x}{(i-1)! * i} = t_{i-1}\frac{x}{i}, \qquad i > 0$$

Millora alternativa

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

$$t_i = \frac{x^i}{i!} = \frac{x^{i-1} * x}{(i-1)! * i} = t_{i-1}\frac{x}{i}, \qquad i > 0$$

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

Implementació

```
double exponencial(double x, int n) {
    double e = 0;
    double t = 1; // t = t_0 = x^0/0!
    int i = 0;
    // Inv: 0 ≤ i ≤ n, t = x^i/i!
    //      i e = Σ_{k=0}^{i-1} x^k/k!
    while (i < n) {
        e += t;
        ++i;
        t = t*x/i;
    }
    return e;
}
```

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
// Post: e conté la suma dels n primers termes de
//       l'expansio en sèrie de Taylor de cos(x)
double cosinus(double x, int n);
```

Millora

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

$$t_i = \frac{x^{2i}}{2i!}(-1)^i$$

Millora

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

$$t_i = \frac{x^{2i}}{2i!}(-1)^i$$

$$\frac{x^{2i-2}}{(2i-2)!}(-1)^{i-1} * \frac{x * x}{2i * 2(i-1)}(-1)$$

Millora

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

$$t_i = \frac{x^{2i}}{2i!}(-1)^i$$

$$\frac{x^{2i-2}}{(2i-2)!}(-1)^{i-1} * \frac{x*x}{2i*2(i-1)}(-1)$$

$$t_{i-1}\frac{x*x}{2i*2(i-1)}(-1), \qquad i > 0$$

Millora

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

$$t_i = \frac{x^{2i}}{2i!}(-1)^i$$

$$\frac{x^{2i-2}}{(2i-2)!}(-1)^{i-1} * \frac{x*x}{2i*2(i-1)}(-1)$$

$$t_{i-1}\frac{x*x}{2i*2(i-1)}(-1), \qquad i > 0$$

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

Implementació

```
double cos(double x, int n) {
    double e = 0;
    double t = 1; // t = t_0 = x^0/0!
    int i = 0;
    // Inv: 0 ≤ i ≤ n, t = (-1)^i x^{2i}/(2i)!
    //      i e = Σ_{k=0}^{i-1} (-1)^k x^{2k}/(2k)!
    while (i < n) {
        e += t;
        ++i;
        t = -t*x*x/(2i*2(i-1));
    }
    return e;
}
```