Tableau 1D : exemple de traitement et de complexités

Des algorithmes d'évaluation polynomiale

Plan et objectifs

L'évaluation polynomiale
- des additions et des multiplications répétées
- des coefficients à stocker dans un tableau 1D

Deux algorithmes d'évaluation
- évaluation classique
- évaluation de Horner

Analyse de la complexité des deux algorithmes
- l'évaluation classique peut-être quadratique
- l'évaluation de Horner est linéaire

Rappels : les vecteurs de nombres

Les vecteurs de nombres

nombres : entier, flottants, (booléens ou bits)
vecteurs : ensemble de ces valeurs
opérations arithmétiques ou logiques : +, -, x, /, ÷, %, AND, OR, XOR
une relation d'ordre : > ou >=

Rappels : parcourir tout le tableau

Exemples
- les calculs arithmétiques ou statistiques des valeurs stockées dans le tableau : max/min, somme/norme, moyenne/écart type, médiane, ...
- évaluation polynomiale : les coefficients sont stockés dans le tableau
- bientôt : le tri des valeurs, par ordre, par signe, ...

Mise en œuvre d'un parcours complet
- la boucle `pour` sur les indices du tableau
- écriture équivalente avec boucle `tant que` + initialisation indice + incrément indice + condition d'arrêt

Evaluation d'un polynôme

Données
un degré : n ou deg
un polynôme a défini par ses n+1 coefficients numériques
stockés dans un tableau de longueur n+1 : a=[a_0, a_1, ..., a_{n-1}, a_n]
une valeur d'évaluation : x

Sortie : on veut calculer
y = $a(x) = a_0 + a_1*x + a_2*x^2 + ... + a_{n-1}*x^{n-1} + a_n*x^n$

Traitement
algorithme classique : on calcule l'expression précédente
. $(...((a_0 + a_1*x) + a_2*x^2) + ...) + a_{n-1}*x^{n-1}) + a_n*x^{n})$
. on répète : le calcul de x^i, le produit avec a[i] et accumulation dans l'ordre croissant des puissances de x

- algorithme de Horner :
. y = $(((((a_n*x + a_{n-1}) *x + a_{n-2}) * x + ...+ (a_2*x + a_1) *x + a_0$
. écriture plus compliquée mais algorithme plus simple et plus efficace

Deux algorithmes d'évaluation polynomiale

Algorithme classique
Algorithme de Horner

Algorithme classique : étape 1

On veut calculer $x^i = x*x*x*$ ... *x pour i = 0, 1, ..., deg.

- si i=0, $x^0 = 1$ - si i=3, x^3=x*x*x ou $x^3=x^2*x$
- si i=1, x^1=x - pour i, x^i = x*x*...*x ou $x^i=x^{i-1} * x$
- si i=2, x^2=x*x - et ainsi jusqu'à i=deg.

Solution 1 : on re-calcule x^1, x^2, x^3 ... x^i à chaque fois

```
xi= 1.0
pour j de 1 à i faire
    xi = xi * x
fin pour // ici j==i et xi==x*x*...*x pour i=1, 2, 3 ....
```

Remarque : on ne calcule pas x^0, xi vaut directement 1.0 quand i==0.

Algorithme classique : étape 2

Ensuite, pour chaque i de 0 à deg :
on multiplie xi par a[i] et on accumule dans la valeur calculée à l'itération précédente, dans l'ordre croissant des puissances de x (xi)
Avant la première itération, la variable-résultat est initialisée à 0 car on effectue une accumulation dans cette variable

```
res = 0.0
pour i de 0 à deg faire // on note deg le degré du polynôme = taille du tableau +1
    // l'étape 1 donne xi pour chaque i
    res = res + a[i] * xi
fin pour // ici xi=x*x*...*x pour i=0,1, ...
```

En supposant que "l'étape 1 fait bien son travail", on vérifie que cette boucle calcule bien :
a[0] pour deg==0 a[0] + a[1]*x + a[2]*x^2 pour deg==2, ...
a[0] + a[1]*xpour deg==1 a[0] + a[1]*x + a[2]*x^2 + ... + a[deg]*x^{deg} pour deg

Algorithme classique

```
fonction EvalPoly(a: tableau de flottants, deg : entier, x: flottant) retourne flottant
// rôle : évalue le polynôme a de degre deg en x. Les coeff sont stockés dans le tableau a avec indice==degré
    res : flottant  = 0.0          // valeur de p(x) initialisée à 0
    xi : flottant = 1.0            // pour x*x*...*x = x^i
    i, j : entier                  // itérateurs
début
    pour i de 0 à deg faire
        xi = 1.0
        pour j de 1 à i faire      // cette boucle n'est pas effectuée si i==0
            xi = xi * x
        fin pour                   // ici on a xi==x*x*...*x==x**i
        res = res + a[i]*xi             // accumulation du terme de degré i (a[i]*xi)
    fin pour                       // pour i=0, 1, ..., deg
    retourne res
fin fonction
```


Remarques

Version 1 :
- deux boucles pour imbriquées
- imbriquée = l'indice de la boucle intérieure dépend de l'indice de la boucle extérieure
- ici : la valeur d'arrêt de la boucle intérieure est modifiée à chaque itération de la boucle extérieure
- ce genre d'imbrication est très classique mais piégeux au début
- attention aussi aux initialisations
  - ici : xi est remis à 1.0 à chaque nouvelle itération de la boucle extérieure

Version 2 :
- on peut éviter de recalculer "tout xi" à chaque fois, ce qui supprime la boucle pour intérieure
- à faire en exercice. Quelles conséquences ? (correction en fin de présentation)

Algorithme de Horner

On va calculer :

$y = (((((a_n*x + a_{n-1})*x + a_{n-2})*x + ... + (a_2*x + a_1)*x + a_0$

Principe :

On multiplie le résultat précédent par x, on ajoute a[i] et on accumule dans une variable résultat.

Cette accumulation s'effectue dans l'ordre des indices décroissants

Comme on accumule, la variable-résultat est initialisée avec précaution :

elle vaut soit 1.0 (on fait un produit), soit a_n au départ (attention aux calculs qui suivent)

Remarques :
- écriture plus compliquée mais algorithme plus simple et plus efficace
- on commence par les indices de degré élevé, puis on décrémente
  → boucle pour avec un pas négatif

```
pour i de n à 0 avec (pas= -1) faire  // n, n-1, n-2, ..., 2, 1, 0
    ....
fin pour
```


Algorithme de Horner

```
fonction Horner(a: tableau de flottants, deg : entier, x: flottant) retourne flottant
// rôle : évalue le polynôme a en x par l'algorithme de Horner.
// Les coeff de a sont stockés dans le tableau a avec indice==degré
    res : flottant              // valeur de p(x)
    i : entier                  // iterateur
début
    res = a[deg]                //coefficient de plus haut degré
    pour i de (deg – 1) à 0 avec (pas= -1) faire
        res = res * x + a[i]    //accumulation de Horner
    fin pour
    retourne res
fin fonction
```

Algorithme de Horner

On vérifie que sont successivement calculés

- a[0] si deg==0
- a[1]*x + a[0] si deg==1
- (a[2]*x + a[1]) *x + a[0] si deg==2
- ((a[3]*x +a[2]) *x + a[1]) *x + a[0] si deg==3
- ...
- (((a[n] *x +a[n-1]) *x + ...) * x + a[1]) *x + a[0] si deg==n

Remarque
- noter l'initialisation de res par le coefficient de plus haut degré

Application : appel

```
declare
// rôle : évalue le polynôme (x+1)^4 par l'algorithme de Horner.
    n : entier = 4                                           // degré
    pascal4[5] : tableau de flottants = [1., 4., 6., 4., 1.]  // coeff du poly pascal4
    x , y : flottant                                         // x et pascal4(x)
début
    afficher("entrer valeur d'évaluation");  lire(x)  // entrée x

    y = Horner(pascal4, 4, x)                         // calcul

    afficher("(x+1)**4 en ", x, " vaut ", y)          // sortie
fin
```


Analyse de la complexité

Comptons, comptons !

Complexité de l'évaluation polynomiale

Complexité en temps
- est une fonction du degré n
- est mesurée par le nombre d'opérations arithmétiques : +, *
- ces opérations apparaissent dans la/les boucles pour
- on se concentre sur ces boucles
- l'ordre de grandeur pour des grandes valeurs de n nous intéresse

Rappel utile
- 1+2+ ...+ n = ?
- somme des n premiers termes d'une suite arithmétique de raison 1 et de premier terme 1
- 1+2+ ...+ n = n(n+1)/2
- ordre de grandeur pour les grandes valeurs de n : $n(n+1)/2 = n^2/2 + \ldots$
- cette somme est croissante "comme x^2" :
  elle a une croissance quadratique

Complexité de l'évaluation classique

```
pour i de 0 à deg faire                  // n == deg
      xi = 1.0
      pour j de 1 à i faire              // cette boucle n'est pas effectuée si i==0
            xi = xi * x
      fin pour
      res = res + a[i]*xi
fin pour
```

Comptons !

- à chaque itération en i : 1 addition, 1 multiplication par $xi=x^i$
- calcul naïf de $x^i = x*x* ...* x$
  - → i multiplications **à chaque fois ...**
  - cad. 0 puis 1 puis 2 ... puis n
- total sur n+1 itérations en i :
  - n+1 additions
  - 1+2+ ... + n multiplications = n(n+1)/2
- total = $(n+1)(n+2)/2 = n^2/2 + ...$
- ordre de grandeur quadratique
  - évaluation quadratique en le nombre d'opérations arithmétiques
- deux boucles pour imbriquées → quadratique

Complexité de l'évaluation de Horner

```
res = a[deg]
pour i de (deg – 1) à 0 avec (pas= -1) faire    // si n==deg alors n itérations
      res = res * x + a[i]
fin pour
```

Comptons !

- à chaque itération : 1 addition et 1 multiplication
- total sur **n** itérations : n additions et n multiplications
- total évaluation : 2n opérations arithmétiques
- évaluation linéaire en le nombre d'opérations arithmétiques
  - . il a été montré que *cet algorithme est optimal* pour l'évaluation des polynômes à valeurs scalaire (1D) : Ostrowsky (54), Pan (66)
  - . optimal = il n'existe pas d'algorithme moins coûteux
- une seule boucle pour → linéaire

Conclusion

Sur les algorithmes d'évaluation polynomiale

L'évaluation polynomiale
- des additions et des multiplications répétées
- des coefficients à stocker dans un tableau 1D

Deux (trois) algorithmes d'évaluation
- évaluation classique : deux versions dont une naïve
- évaluation de Horner : plus simple en fait !

Analyse de la complexité des deux algorithmes
- l'évaluation de Horner est linéaire
- l'évaluation classique naïve est quadratique donc plus coûteuse

Morale
- deux boucles pour imbriquées : coût quadratique
- une seule boucle pour : coût linéaire

Exercices

Coder ces algorithmes :

- pour obtenir des jolis tracés d'évaluation de vos polynômes préférés
- pour mesurer leurs performances réelles
  . prendre des polynômes type $(x\text{-}1)^n$ sous forme développée (triangle de Pascal, coefficients binomiaux) pour faire varier facilement le degré des polynômes;
  . pour chaque degré n, évaluer en un nombre important de points et faire la moyenne
  . rassembler ces mesures dans des courbes et comparer avec les résultats théoriques

Supplément

On peut aussi écrire l'évaluation classique avec un algorithme linéaire

Algorithme classique : version 2

On peut éviter de recalculer "tout xi" à chaque fois, ce qui supprime la boucle pour intérieure

```
fonction EvalPolyEco(a: tableau de flottants, deg : entier, x: flottant) retourne flottant
// rôle : évalue le polynôme a de degre deg en x. Les coeff sont stockés dans le tableau a avec indice = degré
    res : flottant  = a[0]          // valeur de p(x) initialisée à a[0]
    xi : flottant = 1.0             // pour x*x*...*x = x^i
    i, j : entier                   // itérateurs
début
    pour i de 1 à deg faire         // boucle non effectuée si i==0
        xi = xi * x                 // mise à jour de xi
        res = res + a[i]*xi         //accumulation du terme de degré i
    fin pour
    retourne res
fin fonction
```


Algorithme classique : version 2

On peut éviter de recalculer "tout xi" à chaque fois, ce qui supprime la boucle pour intérieure

Quid de sa complexité ?