INF 111, L1, 2004/2005

Feuille 10

Novembre 2004

Méthodes de calcul de x^n

Laure Gonnord
`http://www-verimag.imag.fr/~gonnord/`

Généralités, Objectifs

Le problème du calcul efficace de x^n pour n entier positif devient crucial lorsque n est particulièrement grand ou bien lorsque x est d'un type de données où le coût d'une multiplication est algorithmiquement coûteux (exemple : une matrice ou un polynôme de degré élevé). Ce qui intéresse l'algorithmicien dans la calcul de x^n est le nombre de multiplications par x ou par des puissances déjà calculées de x. On va ici essayer de minimiser ce nombre de multiplications.

Dans un premier temps, on va coder deux algorithmes de base dans le cas où x est un entier, en évaluant dans chaque cas le coût de chaque fonction. Dans une seconde partie, on réalisera le même travail dans le cas où x est une matrice de taille arbitrairement fixée.

1 Exponentiation d'entiers

1.1 La méthode élémentaire

L'algorithme le plus simple pour calculer x^n est basé sur les équations suivantes :

$$\begin{cases} x^0 = 1 \\ x^{k+1} = x * x^k & \text{si } k > 0 \end{cases}$$

►Exo :

1. Écrire la fonction `puiss_naive` qui prend en argument deux entiers x et k et qui calcule x^k par cette méthode.
2. Donner en fonction de n le nombre de multiplications que réalise cet algorithme

1.2 L'exponentiation rapide

Cette méthode est basée sur l'exemple suivant :

$$x^{23} = x * (x^2)^{11} = (x * x^2) * (x^4)^5 = (x * x^2 * x^4) * (x^8)^2 = x * x^2 * x^4 * x^{16}$$

►Exo :

1. En nommant les différentes parties de l'égalité : `res`, `puiss` et `k` judicieusement, obtenir l'invariant de boucle suivant : «A chaque tour de boucle, on a $res * puiss^k = x^n$». Comment obtenir res, $puiss$, k à la boucle suivante ? Quand s'arrête-t-on ?
2. Ecrire la fonction `puiss_dicho` qui prend en argument les entiers x et k et qui calcule x^k avec cet algorithme.
3. Donner en fonction de n le nombre de multiplications $C(n)$ que réalise cet algorithme. On trouvera $C(n) \leq 2 \log n$. On commencera par établir $C(n) \leq 2 + C(n/2)$ puis on posera $k = 2^n$.

1.3 Quelques tests

Sur les exemples numériques, on ne "voit" pas la différence de temps d'exécution des deux algorithmes, vu que l'on ne peut pas demander des résultats sur des entiers très grands [1].

▶ Donner une solution facile qui modifie très peu les programmes écrits et qui permet de "voir" s'exécuter les algorithmes sur des temps notoirement différents.

2 Exponentiation de matrices

On va dans cette section coder les deux algorithmes précédents lorsque x est une matrice. Si on voulait faire ce travail en TP, on utiliserait des tableaux en deux dimensions, et il faudrait en premier lieu écrire les fonctions de saisie et de visualisation des matrices. Ici, on suppose que les matrices sont données et sont toutes de dimension N.

▶Exo :

1. Pourquoi ne peut-on plus ici utiliser de fonctions ?
2. Écrire la procédure `mult(int A[N][N], int B[N][N], int C[N][N])` qui réalise l'opération matricielle $C \leftarrow A * B$.
3. Écrire la procédure `mat_id(int Id[N][N])` qui fait en sorte que `Id` devienne la matrice identité d'ordre `N`.
4. Écrire la procédure `puiss_naive(int A[N][N], int n, int R[N][N])` qui réalise l'opération matricielle $R \leftarrow A^n$ par la méthode naïve.
5. Coder la procédure `puiss_dicho(int A[N][N], int n, int R[N][N])` qui réalise l'opération matricielle $R \leftarrow A^n$ par la méthode d'exponentiation rapide.
6. En s'inspirant de la section 1.3, faire en sorte d'obtenir des temps d'exécution notoirement différents pour les deux méthodes.

[1] Au fait, pourquoi ?