HEC 922, 2001/2002 **TP 10**

Les 8 et 9 janvier 2001

Calcul efficace de x^n

Laure Danthony
`http://www.ens-lyon.fr/~ldanthon/`

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 [1].

Dans un premier temps, on va coder les versions récursives et itératives de deux algorithmes de base dans le cas où x est un entier, en évaluant dans chaque cas le coût de chaque procédure. Dans une seconde partie, on réalisera le même travail dans le cas où x est une matrice de taille arbitrairement fixée. Evidemment, lors du code des algorithmes, on s'attachera à tester ses fonctions et ses procédures sur de nombreux exemples *représentatifs*. Enfin, dans une troisième partie (et si on a le temps), on regardera quelques algorithmes anecdotiques que l'on ne codera pas.

On fournit dans le répertoire habituel les fichiers `puiss_squelette.pas` pour la première partie et `puiss_mat_squelette.pas` pour la deuxième partie.

1 Exponentiation d'entiers

1.1 La méthode élémentaire

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

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

▶Exo :

1. Coder la version récursive `puiss_naive_rec` de cet algorithme.
2. Coder la version itérative `puiss_naive_iter` de cet algorithme.
3. Donner en fonction de n le nombre de multiplications que réalise cet algorithme

[1] Ce blabla ainsi que les questions posées sont en partie tirées du problème 2 année 2001 des Mines MP/MP* option info

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. Coder la version récursive `puiss_dicho_rec` de cet algorithme. On utilisera judicieusement la fonction `pair(n:integer)` [2].
2. 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 $res * puiss^k = x^n$ ". Quand s'arrête-t-on ?
3. Coder alors la version itérative `puiss_dicho_iter` de cet algorithme.
4. Donner en fonction de n le nombre de multiplications que réalise cet algorithme. On trouvera l'encadrement $\leq 2 \log n$.

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 [3].

▶ 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. Dans le fichier fourni `puiss_mat.pas` on fournit le squelette des programmes à remplir, ainsi que des fonctions de saisie et de visualisation des matrices. On utilisera judicieusement le copier-coller à partir du travail déjà effectué sur les entiers.

▶Exo :

1. Pourquoi ne peut-on plus ici utiliser de fonctions ?
2. Coder la procédure `mult(A,B:mat_int;var C:mat_int)` qui réalise l'opération matricielle $C \leftarrow A * B$.
3. Coder la procédure `mat_id(var Id:mat_int)` qui fait en sorte que `Id` devienne la matrice identité d'ordre `LONG`.
4. Coder la procédure `puiss_naive(A:mat_int;n:integer;var R:mat_int)` qui réalise l'opération matricielle $R \leftarrow A^n$ par la méthode naïve.
5. Coder la procédure `puiss_dicho(A:mat_int;n:integer;var R:mat_int)` 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.

[2] Au fait, que fait cette fonction ?
[3] Au fait, pourquoi ?

3 Vers un nombre de multiplications minimal

Avertissement Ce que l'on raconte ici est anecdotique et n'est donc évidemment pas exigeable. On va montrer que l'algorithme d'exponentiation rapide n'est pas optimal, et qu'il existe des façons de faire moins d'opérations dans la plupart des cas. Malheureusement, ce ne sont que des méthodes théoriques, car elles sont difficilement implémentables dans la plupart des cas.

3.1 La méthode des facteurs

L'algorithme dichotomique réduit sensiblement le nombre de multiplications par rapport à l'algorithme élémentaire. Mais il n'est pas optimal. C'est ce qu'on va montrer ici.

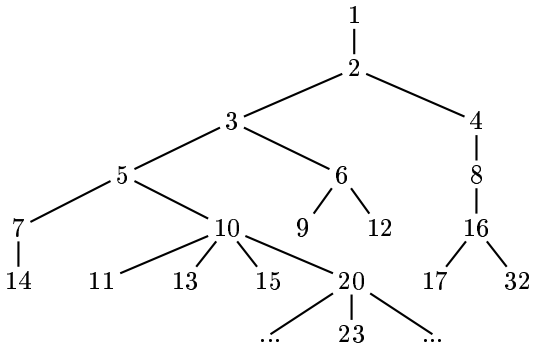
▶ En écrivant $15 = 3 * 5$ et en appliquant l'algorithme d'exponentiation rapide sur chacun des "petits bouts", montrer que cette méthode, dite "méthode des facteurs" est meilleure que la méthode d'exponentiation rapide pure pour calculer x^{15} (on trouvera 5 multiplications contre 6).

REMARQUE 1 On n'utilise pas cet algorithme en pratique car il suppose que l'on sait factoriser n, ce qui est très difficile à réaliser et qui "coûte très cher".

3.2 La méthode de l'arbre

On construit un arbre de la façon suivante :
- On crée un arbre de père 1 et de fils 2.
- On remplit chaque niveau de gauche à droite, en plaçant uniquement des nombres qui n'ont pas été encore placés, de façon croissante.
- Pour placer les fils d'un nœud, on effectue les sommes deux à deux avec chacun de ses ancêtres (au sens large, *i.e.* avec lui même, son père, *etc.*jusqu'à la racine de l'arbre, qui est 1).
- On s'arrête lorsque l'on a placé le chiffre n.

Pour $n = 23$, cela donne :



On en déduit un calcul de x^{23} : on calcule successivement x^2, $x^2 * x = x^3$, $x^2 * x^3 = x^5$, $(x^5)^2 = x^{10}$, $(x^{10})^2 = x^{20}$ et $x^{20} * x^3 = x^{23}$ en 6 opérations, alors que la méthode des facteurs et la méthode d'exponentiation rapide donnent 7 opérations. Cette méthode est expérimentalement meilleure que les autres, dans 90% des cas. Mais là aussi, elle est difficile à implémenter.