Exponentiation rapide

Concepts : Analyse de coût, diviser pour régner,
Méthodes : Décomposition du coût, "master theorem"

Présentation

Étant donné une opération $\star$ sur des objets (par exemple des entiers, des réels,...), l'objectif est de calculer $x^n = x \star x \star x \star \cdots \star x$ en un nombre minimal d'opérations . On suppose que l''opération $\star$ est associative, c'est à dire que pour tout triplet (x, y, z) on a $x \star (y \star z) = (x \star y) \star z$. Un premier algorithme naïf (3 donne un coût en nombre d'opérations $\star$ en $\mathcal{O}(n)$.

```
PUISSANCE-NAIF(x,n)
  Données : Un objet x et un entier n positif
  Résultat : La valeur de x^n

  y = x
  for i = 2 à n faire y = y ⋆ x
  retourner y
```

Algorithme 1: Algorithme naïf du calcul de la puissance

La preuve de cet algorithme est laissée en exercice ainsi que le calcul de sa complexité.

Algorithme

Pour améliorer l'algorithme naïf on utilise la propriété d'associativité de l'opération $\star$. C'est à dire que $x^{2n} = (x^n)^2$, on calcule d'abord x^n puis par une seule opération on déduit $x^2 n$.

```
PUISSANCE-DIV(x,n)
  Données : Un objet x et un entier n positif
  Résultat : La valeur de x^n

  if n = 1                                    // cas de base
    retourner x
  else                                        // récursion
    if n pair
1     z =PUISSANCE-DIV(x,n/2)
      retourner z ⋆ z
    else                                      // n est impair ⩾ 3
2     z =PUISSANCE-DIV(x,(n − 1)/2)
      retourner z ⋆ z ⋆ x
```

Algorithme 2: Algorithme naïf du calcul de la puissance

Preuve

La preuve se fait en 2 parties, la correction partielle liée à l'appel récursif et la terminaison qui garantit le résultat.

Exponentiation

1. **Correction partielle** Supposons que l'appel récursif ligne 1 (resp 2) fournisse le résultat correct. Dans ce cas la valeur retournée par la fonction PUISSANCE-DIV sera correcte, car la valeur de z vaut $x^{\frac{n}{2}}$ et la valeur retournée $z \star z$ est égale à $x^{\frac{n}{2}} \star x^{\frac{n}{2}} = x^n$, (respectivement z vaut $x^{\frac{n-1}{2}}$ et $x^{\frac{n-1}{2}} \star x^{\frac{n-1}{2}} \star x = x^n$).
2. **Terminaison** L'algorithme se termine car à chaque appel récursif de la fonction PUISSANCE-DIV le deuxième argument ($n/2$ ou $(n-1)/2$ est un entier strictement décroissant minoré par 1. Donc la séquence des appels récursifs se termine toujours avec un appel où $n = 1$. Dans ce cas la valeur retournée est x c'est à dire x^1 et en utilisant la correction partielle PUISSANCE-DIV(x,n) retourne bien la valeur de x^n.

Complexité

Pour évaluer le coût de cet algorithme en nombre d'appels à des opérations "primitives" $\star$ on décompose les appels récursifs et on compte le nombre d'appels récursifs. Notons $R(n)$ le nombre d'appels récursifs, la structure de l'algorithme montre que [1] $R(n) \leqslant R(\frac{n}{2})$. On en déduit que $R(n) \leqslant \log_2(n)$. Comme un appel récursif fait 0 (cas de base), ou 1 (cas n pair) ou 2 (cas n impair) opérations $\star$, on en déduit que le nombre d'opérations $\star$ $C(n)$ est majoré par $2\log_2(n)$. Comme l'algorithme fait toujours au moins $\log_2 n$ appels, $C(n)$ est minoré par $\log_2(n)$. Ce qui conduit à

$$C(n) = \Theta(\log(n)).$$

Exercices

1. Montrer que le meilleur cas pour PUISSANCE-DIV(x,n) est $\log_2(n)$ lorsque n est une puissance de 2.
2. Donner le pire cas pour PUISSANCE-DIV(x,n)
3. Faire le lien entre la séquence des appels récursifs et la décomposition binaire de n.
4. Écrire un programme itératif de calcul de la puissance.

Méthode des facteurs

```
PUISSANCE-FACT(x,n)
   Données : Un objet x et un entier n positif
   Résultat : La valeur de x^n
   if n = 1                                         // cas de base
      retourner x
   else                                             // récursion
      if n est premier
1        z =PUISSANCE-FACT(x,n − 1)
         retourner z ⋆ z
      else                                          // n est composé n = p × q
2        y =PUISSANCE-FACT(x,p)
         z =PUISSANCE-FACT(y,q)
         retourner z
```

Algorithme 3: Algorithme de calcul de la puissance basée sur une décomposition en facteurs

1. faire la preuve par récurrence

Exponentiation

Exercices

1. Écrire la preuve de cet algorithme.
2. Trouver une valeur de n pour laquelle PUISSANCE-DIV(x,n) a un coût meilleur que PUISSANCE-FACT(x,n).
3. Trouver une valeur de n pour laquelle PUISSANCE-FACT(x,n) a un coût meilleur que PUISSANCE-DIV(x,n).

Optimalité

Comme le nombre de façons de calculer x^n est fini. On peut construire l'*arbre des puissances*

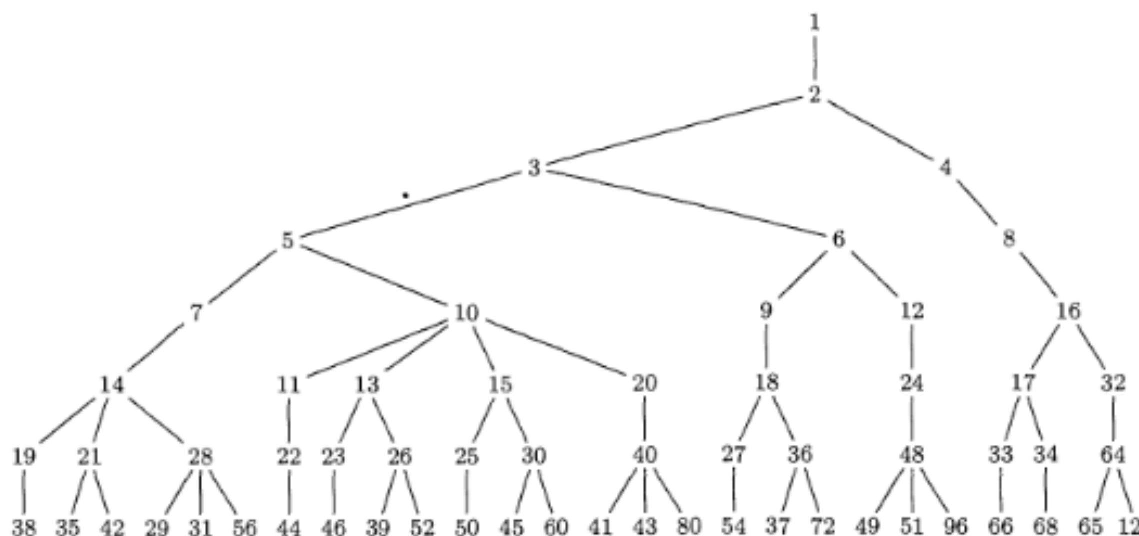

Fig. 14. The "power tree."

D.E. Knuth The Art of Computer Programming : Semi-Numerical Algorithms (vol 2) p464
Pour calculer par exemple x^{29} on calcule successivement x^2, x^3, x^5, x^7, x^{14}, x^{28} et x^{29}.

Complexité du problème

Théorème

> Le coût du problème du calcul de la puissance est en $\Theta(\log n)$.

En effet, montrons par récurrence la propriété *le coût minimal d'un algorithme de calcul de x^n est minoré par* $\log_2 n$.

Cette propriété est vraie pour $n = 1$ et $n = 2$. Supposons la propriété vraie pour tout $1 \leqslant i \leqslant n-1$ et considérons l'algorithme optimal qui calcule x^n (il existe car il existe un nombre fini de calculer x^n). Cet algorithme effectue uniquement des opérations $\star$. La dernière exécution fait le produit $x^k \star x^{n-k}$ pour une valeur de k que l'on peut choisir supérieur à $\frac{n}{2}$ (entre k et $n-k$ l'un des 2 est plus grand que $\frac{n}{2}$). Par suite le calcul de x^k demande au moins $\log_2 k$ opérations, c'est à dire au moins $\log_2 \frac{n}{2} = \log_2 n - 1$. Donc le nombre d'opérations est minoré par $(\log_2 n - 1) + 1 = \log_2 n$ cqfd.

Moralité

Quelle moralité tirer du théorème ci-dessus ?

Exponentiation

Références

L'analyse détaillée du problème de l'exponentiation est tirée de l'ouvrage de D.E. Knuth (chap 4.6.3) [1] avec des détails historiques.

The same idea applies, in general, to any value of n, in the following way: Write n in the binary number system (suppressing zeros at the left). Then replace each "1" by the pair of letters SX, replace each "0" by S, and cross off the "SX" that now appears at the left. The result is a rule for computing x^n, if "S" is interpreted as the operation of *squaring*, and if "X" is interpreted as the operation of *multiplying by* x. For example, if $n = 23$, its binary representation is 10111; so we form the sequence SX S SX SX SX and remove the leading SX to obtain the rule SSXSXSX. This rule states that we should "square, square, multiply by x, square, multiply by x, square, and multiply by x"; in other words, we should successively compute x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, x^{23}.

This binary method is easily justified by a consideration of the sequence of exponents in the calculation: If we reinterpret "S" as the operation of multiplying by 2 and "X" as the operation of adding 1, and if we start with 1 instead of x, the rule will lead to a computation of n because of the properties of the binary number system. The method is quite ancient; it appeared before 200 B.C. in Piṅgala's Hindu classic *Chandaḥ-sûtra* [see B. Datta and A. N. Singh, *History of Hindu Mathematics* **2** (Lahore: Motilal Banarsi Das, 1935), 76]. There seem to be no other references to this method outside of India during the next 1000 years, but a clear discussion of how to compute 2^n efficiently for arbitrary n was given by al-Uqlīdisī of Damascus in A.D. 952; see *The Arithmetic of al-Uqlīdisī* by A. S. Saidan (Dordrecht: D. Reidel, 1975), 341–342, where the general ideas are illustrated for $n = 51$. See also al-Bīrūnī's *Chronology of Ancient Nations*, edited and translated by E. Sachau (London: 1879), 132–136; this eleventh-century Arabic work had great influence.

D.E. Knuth The Art of Computer Programming : Semi-Numerical Algorithms (vol 2) p461

Références

[1] Donald E.Knuth. *The Art of Computer Programming Volume 2 : Seminumerical Algorithms*. 2002.