Informatique

Exponentiation rapide

Ce TP recoupe l'un des exercice de programmation de la feuille "Fonctions et procédures".

0. Présentation

Nous supposons que *Maple* ne dispose pas de la fonction puissance (**^**).
On programme successivement deux algorithmes :

- l'algorithme naturel (mais naïf) ;
- l'algorithme dit *d'exponentiation rapide*, qui tire parti de l'écriture binaire d'un entier.

On analyse ensuite leur efficacité et leur précision.

I. Algorithme naïf

Il est simple (mais fastidieux) d'élever un nombre *x* à la puissance *n* ($n \in N$) en le multipliant par lui-même *n* -1 fois.

1. Écrire une procédure **expoBete(x,n)** faisant "naïvement" le travail décrit ci-dessus.

```
> expobete := proc(x,n)
  local e, q ;
  e := 1 ;            # contient les produits partiels
  q := n ;
  while q > 0         # tant qu'il reste à multiplier...
  do
    e := evalf(e * x) ;      # ... on multiplie e par x...
    q := q - 1        # ... et on décrémente q
  end ;
  e
  end :
```

2. Tester :

```
> expobete(2,10) ;
```

1024

3. Équiper la procédure précédente d'un compteur. Combien de multiplications pour élever 2 à la puissance 10 ?

```
> expobete2 := proc(x,n)
  local c, e, q ;     # "c" est le compteur
  e := 1 ; q := n ; c := 0 ;
  while q > 0
  do
    e := e * x ;
    c := c + 1 ;
```

```
    q := q - 1
  end ;
  (e,c)
  end :
> expobete2(2,10) ;
```

$$1024, 10$$

Rien de surprenant.

II. Algorithme d'exponentiation rapide

Toutefois, pour élever *x* à la puissance, p. ex., 16 (= 2^4), il est bien plus judicieux de procéder ainsi : multiplier *x* par lui-même (x^2), puis le résultat par lui-même (x^4), encore multiplié par lui-même (x^8) et enfin une dernière fois (x^{16}).
Si l'on cherchait x^{21}, il aurait suffi de multiplier x^{16} par x^4 (calculé en cours de route) et enfin par *x*.

1. Écrire une procédure **expoRapide(x,n)** tirant parti de ces remarques.

```
> exporapide := proc(x,n)
  local e, m, q ;
  e := 1 ;            # contient les produits partiels
  q := n ;
  m := x ;            # contient x, x^2, x^4, x^16, ...
  while q > 0         # tant qu'il reste à multiplier...
  do
    if type(q,odd)    # si k-ième chiffre binaire de n...
      then e := e * m #  est égal à 1...
    fi;               # ... e est multiplié par m = x^(2^k)...
    m := m * m ;      # ... puis m est élevé au carré...
    q := iquo(q,2)    # ... tandis que q est divisé par 2
  end ;
  e
  end :
```

2. Tester.

```
> exporapide(2,10) ;
```

$$1024$$

3. Équiper la procédure précédente d'un compteur. Combien de multiplications pour élever 2 à la puissance 10^5 ?

```
> exporapide2 := proc(x,n)
  local c, e, m, q ;
  e := 1 ; q := n ; m := x ; c := 0 ;
  while q > 0
  do
    if type(q,odd)
      then e := e * m ; c := c + 1 ;
    fi;
    m := m * m ; c := c + 1 ;
    q := iquo(q,2)
```

```
   end ;
   c ;
   end :
> exporapide2(2,10^5) ;
                                  23
```

Pour mémoire, l'algorithme naïf en aurait requis 10^5...

III.Analyse

On se propose de tester la *précision* des algorithmes précédents sur une valeur approchée de π^{10^7}.

1. Quelle est la valeur fournie par *Maple* ?

```
> evalf(Pi^(10^7)) ;
                        5.339596126 10^4971498
```

On la prendra comme référence.

2. Il sera prudent de placer un evalf judicieux dans la procédure "naïve" afin de limiter le nombre de calculs ! Comparer les précisions obtenues :

```
> expobete(evalf(Pi),10^7) ;
                        5.339596019 10^4971498
```

```
> exporapide(evalf(Pi),10^7) ;
                        5.340227235 10^4971498
```

Cette valeur, obtenue beaucoup plus rapidement que la précédente, est moins précise !
Le résultat dépend des machines, mais typiquement, la précision fournie par l'algorithme naïf est au moins égale à celle de l'algorithme rapide.

3. Essayons de l'expliquer. (Le raisonnement qui suit est heuristique, et n'a rien à voir avec *Maple*).

(a) Dans la procédure naïve, comment évolue l'erreur d_i à l'étape *i* ? En déduire l'ordre de grandeur de l'erreur finale d_n en fonction de l'erreur initiale d_0.

Elle est approximativement multipliée par *a* : $d_i = a\, d_{i-1}$. On fait ceci *n* fois, donc $d_n = a^n\, d_0$.

(b) Dans l'algorithme rapide :

- Combien fait-on d'élévations au carré au maximum en fonction de *n* ?

Leur nombre est approximativement $k = \log_2(n)$.

- Pour chacune d'elles, comment évolue l'erreur d_i?

La valeur élevée au carré étant notée x_i, $x_i = x_{i-1}^2$ donc si x_{i-1} comporte une erreur d_{i-1}, l'erreur dans x_i sera environ $2\, x_{i-1}\, d_{i-1}$, ou encore $2\, a^{2i-1}\, d_{i-1}$.

- En déduire l'expression de l'ordre de grandeur de l'erreur finale en fonction de l'erreur initiale. Commentaire ?

On aura donc à la fin des *k* élévations au carré une erreur :
$d_k = 2^k \, a^{2^k - 1 + 2^k - 2 + \ldots + 1} \, d_0$ soit environ $2^k \, a^{2^k - 1}$ ou encore $\frac{2^k \, a^n}{a}$,
supérieure à la précédente ! Comme quoi, "moins de calculs" n'est pas toujours synonyme de "moins d'erreur".