Le problème de l'exponentiation rapide

Lundi 05 Mars 2018

Michael FRANÇOIS

francois@esiea.fr

https://francois.esiea.fr

Exponentiation (modulaire)

Exponentiation

• Soit G un ensemble muni d'une loi de composition interne notée $\circ$ et supposée associative ; c'est-à-dire :

- pour tout triplet (g_1, g_2, g_3) appartenant à l'ensemble G on a : $(g_1 \circ g_2) \circ g_3 = g_1 \circ (g_2 \circ g_3)$

• Si on utilise la notation multiplicative pour la loi $\circ$, l'opération d'exponentiation dans $(G, \circ)$ pour un entier $e > 0$, consiste à calculer la e-ième puissance d'un élément $g \in G$:

$$g^e = \underbrace{g \circ g \circ \cdots \circ g \circ g}_{\text{e termes}}$$

• L'élément g^e est défini à partir du moment où l'opération $\circ$ est associative.

Exponentiation modulaire

• Une opération fréquente en théorie des nombres consiste à élever un nombre à une puissance modulo un autre nombre, cette opération est appelée **exponentiation modulaire**.

• L'objectif est de trouver un moyen permettant de calculer efficacement :

$$g^e \mod n,$$

où g et e sont deux entiers positifs ou nuls et n est un entier strictement positif.

• L'**exponentiation modulaire** est une opération fondamentale pour de nombreuses fonctions de test de primalité, de génération pseudo-aléatoire ou même pour le cryptosystème à clé publique **RSA**.

• Elle est beaucoup utilisée en cryptographie (chiffrement, signature, etc.) et notamment dans les cartes à puces bancaires, cartes vitales, etc.

• Il faut préciser de suite que l'égalité :

$$g^n = e^{n*\log(g)},$$

qui n'a d'ailleurs pas de sens si g est une matrice par exemple, ne donne pas un algorithme pour le calcul de g^n lorsque g est un entier, on aura tout au plus une valeur approchée.

• Nous nous intéressons ici au calcul exact, et aussi rapide que possible, de g^n .

Exemples :

- $g = 3,\ e = 7,\ n = 17$
  - $g^e = g * g * g * g * g * g * g = 2187$
  - $g^e \mod n = 11$
- $g = 567,\ e = 123456789,\ n = 10^{100}$
  - $g^e \mod n =$ 328696614137630343867766087400493782711726661161493381315251517672036950022969768525804505113611624 7

Algorithme naïf

– Entrée : g, e, n où g, e sont deux entiers positifs ou nuls et $n > 0$.

– Sortie : $g^e \mod n$

• La méthode la plus naïve, consiste à faire d'abord $e - 1$ multiplications de g dans $\mathbb{Z}$ puis réduire le tout modulo n.

- complexité proportionnelle à e.

• On utilise l'arithmétique sur les grands entiers pour réduire à la fin $\Rightarrow$ il faut absolument penser a réduire au fur et à mesure des calculs :

$$g^e \mod n = g * g^{e-1} \mod n$$

Algorithme : version naïve de l'exponentiation modulaire
Données : $\mathtt{g} \in \mathbb{Z}_n, \mathtt{e} \in \mathbb{Z}_n, \mathtt{n} \in \mathbb{N}^*$;
Sortie : $\mathtt{g}^{\mathtt{e}} \mod \mathtt{n}$;
Début

```
si e=0 alors
    Retourner 1;
fin
si e=1 alors
    Retourner g mod n;
fin
Temp = 1;
pour i=0 jusqu'à e-1 inclus faire
    Temp = Temp * g mod n /*réduction au fur et à mesure*/
fin
Retourner Temp;
```

Code C :

```c
#include <stdio.h>
#include <stdlib.h>

int Expo_Modulaire(int g, int e, int n)
{
  int i, Temp;
  if (e==0) {return 1;}
  if (e==1) {return g % n;}
  Temp=1;
  for (i=0; i<e; i++)
  {
    Temp = Temp * g % n;
  }
  return Temp;
}

int main(int argc, char** argv)
{
  int g, e, n;
  g = abs(atoi(argv[1]));
  e = abs(atoi(argv[2]));
  n = abs(atoi(argv[3]));
  printf("%d^%d mod %d = %d\n", g, e, n, Expo_Modulaire(g, e, n));
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 3 7 17
3^7 mod 17 = 11
$ ./EXEC 5 6 97
5^6 mod 97 = 8
$ ./EXEC 78 1 9
78^1 mod 9 = 6
```

Exemple de Code C pour l'exponentiation simple : (**Fonction itérative**)

```c
#include <stdio.h>
#include <stdlib.h>

float Exponentiation_Iter(float g, int e)
{
  float EXP;
  int i;
  if (e==0) {return 1;}
  if (e==1) {return g;}
  EXP=g;
  for (i=2; i<=e; i++)
  {EXP = EXP * g;}
  return EXP;
}

int main(int argc, char** argv)
{
  float g; int e;
  g = atof(argv[1]);
  e = atoi(argv[2]);
  printf("%.2f^%d = %.2f\n", g, e, Exponentiation_Iter(g, e));
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 4 1
4.00^1 = 4.00
$ ./EXEC 5 6
5.00^6 = 15625.00
$ ./EXEC 7.2 8
7.20^8 = 7222040.00
$ ./EXEC 10 0
10.00^0 = 1.00
```

Exemple de Code C pour l'exponentiation simple : (**Fonction récursive**)

```c
#include <stdio.h>
#include <stdlib.h>

float Exponentiation_Rec(float g, int e)
{
      ?????

}

int main(int argc, char** argv)
{
  float g; int e;
  g = atof(argv[1]);
  e = atoi(argv[2]);
  printf("%.2f^%d = %.2f\n", g, e, Exponentiation_Rec(g, e));
  return 0;
}
```

Exemple de Code C pour l'exponentiation simple : (**Fonction récursive**)

```c
#include <stdio.h>
#include <stdlib.h>

float Exponentiation_Rec(float g, int e)
{
  if (e==0) {return 1;}
  if (e==1) {return g;}
  return g*Exponentiation_Rec(g, e-1);
}

int main(int argc, char** argv)
{
  float g; int e;
  g = atof(argv[1]);
  e = atoi(argv[2]);
  printf("%.2f^%d = %.2f\n", g, e, Exponentiation_Rec(g, e));
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 7 1
7.00^1 = 7.00
$ ./EXEC 8 5
8.00^5 = 32768.00
$ ./EXEC 1006 0
1006.00^0 = 1.00
$ ./EXEC 3 12
3.00^12 = 531441.00
$ ./EXEC 124 56
124.00^56 = inf
```

Exemple de Code C pour l'exponentiation simple : (**Fonction récursive Terminale**)

```c
#include <stdio.h>
#include <stdlib.h>

float Exponentiation_Rec_Term(float g, int e, float acc)
{

    ?????

}

int main(int argc, char** argv)
{
  float g; int e;
  g = atof(argv[1]);
  e = atoi(argv[2]);
  printf("%.2f^%d = %.2f\n", g, e, Exponentiation_Rec_Term(g, e, 1));
  return 0;
}
```

Exemple de Code C pour l'exponentiation simple : (**Fonction récursive Terminale**)

```c
#include <stdio.h>
#include <stdlib.h>

float Exponentiation_Rec_Term(float g, int e, float acc)
{
  if (e==0) {return acc;}
  if (e==1) {return g*acc;}
  return Exponentiation_Rec_Term(g, e-1, g*acc);
}

int main(int argc, char** argv)
{
  float g; int e;
  g = atof(argv[1]);
  e = atoi(argv[2]);
  printf("%.2f^%d = %.2f\n", g, e, Exponentiation_Rec_Term(g, e, 1));
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC 8 0
8.00^0 = 1.00
$ ./EXEC 8 1
8.00^1 = 8.00
$ ./EXEC 8 10
8.00^10 = 1073741824.00
$
```

Exponentiation rapide : principes de base

Exponentiation rapide : principes de base

• Tout algorithme efficace de calcul de $g^e \mod n$ doit donc résoudre deux problèmes :

- diminuer le coût d'un produit $x \circ y$, où x et y sont connus ;
- diminuer le nombre d'opérations sur des éléments de G.

• Diminuer le coût d'une opération demande en général d'utiliser (ou de trouver) un algorithme adapté et qui soit plus rapide que l'algorithme standard, ce qui n'est pas toujours évident.

- Par exemple si $G = \mathbb{Z}_n = \{0, \ldots, n-1\}$ avec n très grand, on peut penser à utiliser l'algorithme de multiplication de grands entiers le mieux adapté (qui dépend du nombre de chiffres de n). Nous nous intéressons ici essentiellement au problème de **diminuer le nombre d'opérations** sur des éléments de G.

Rappels

1. $g^{a+b} = g^a * g^b$
2. $g^{a*b} = (g^a)^b$
3. Si $e = \sum_{i=0}^{i=k} d_i$ alors :

$$g^e = g^{\sum_{i=0}^{i=k} d_i} = \prod_{i=0}^{i=k} g^{d_i}$$

4. Si $e = \sum_{i=0}^{i=k} a_i * b_i$ alors :

$$g^e = g^{\sum_{i=0}^{i=k} a_i * b_i} = \prod_{i=0}^{i=k} g^{a_i * b_i} = \prod_{i=0}^{i=k} (g^{a_i})^{b_i} = \prod_{i=0}^{i=k} (g^{b_i})^{a_i}$$

Conséquence 1 :

Par exemple si $e = 2^p$, on peut écrire que :

$$e = 2^P = 2^{1+p-1} = 2 * 2^{p-1} = 2^{p-1} + 2^{p-1}$$

donc à partir de la 2) on peut déduire :

$$g^e = g^{2*2^{p-1}} = (g^2)^{2^{p-1}} = \left(g^{2^{p-1}}\right)^2 = \left(g^{2^{p-1}}\right) * \left(g^{2^{p-1}}\right)$$

Ainsi, calculer par exemple x^8 ne nécessiterait que 3 multiplications :

$$x^2 = x * x, \quad x^4 = x^2 * x^2, \quad x^8 = x^4 * x^4$$

Conséquence 2 :

• Il est facile de conclure que si $e = 2^p$, nous sommes passés de $e - 1$ multiplications à $p = \log_2(e)$, ce qui fait une complexité logarithmique (*i.e.* $O \log e$).

• Pour $e \neq 2^k$, on prend l'écriture de e en base 2, soit :

$$e = (d_m \cdots d_1 d_0)_2$$

avec la convention $d_m \neq 0$.
On obtient ainsi :

$$g^e = \prod_{i=0}^{m} g^{2^i * d_i} = \prod_{i=0}^{m} \left(g^{2^i}\right)^{d_i}$$

Exemple : $g = 5$, $e = 12$ (en binaire $\Longrightarrow 1100$), alors on a :

$$5^{12} = \left(5^{2^3}\right)^1 * \left(5^{2^2}\right)^1 * \left(5^{2^1}\right)^0 * \left(5^{2^0}\right)^0$$

$$= 5^8 * 5^4 * 1 * 1$$

$$= 390625 * 625$$

$$= 244140625$$

Conséquence 3 :

- Ce qui donne alors directement l'algorithme :
  - on calcule $y_1 = g^2$, $y_2 = y_1^2 = g^4$, $\ldots$, $y_m = g^{2^m} = y_{m-1}^2$;
  - on définit l'ensemble $I = \{i | d_m \neq 0\}$;
  - on calcule $g^e = \prod_{i \in I} y_i$
- Ceci donne un premier algorithme, qu'on appelle l'algorithme d'exponentiation indienne (car il était connu en Inde 200 ans avant J.-C.).

Conséquence 4 :

• Cet algorithme se nomme en anglais $right - to - left$ binary exponentiation algorithm : il correspond donc à un parcours droite-gauche des bits de e. On peut s'en convaincre sur un exemple général : soit en base 2, $e = abcd$, on a donc :

$$e = a * 2^3 + b * 2^2 + c * 2^1 + d * 2^0 = a * 2^3 + b * 2^2 + c * 2 + d,$$

ce qui permet d'écrire :

$$g^e = ((g^a)^2 * g^b)^2 * g^c)^2 * g^d = (g^a)^{2^3} * (g^b)^{2^2} * (g^c)^{2^1} * g^d$$

$$= \left(g^{2^3}\right)^a * \left(g^{2^2}\right)^b * \left(g^{2^1}\right)^c * g^d$$

• Mais comme $a, b, c, d \in \{0, 1\}$, on a en fait une succession d'élévations au carré et éventuellement de produits.

Algorithme : Exponentiation "Diviser Pour Régner"
Données : $g \in \mathbb{Z}_n, e \in \mathbb{N}^*, n \in \mathbb{N}^*$;
Sortie : `Expo_Rap_Rec(g, e, n)` $= g^e \mod n$;
Début

```
si e=1 alors
    Retourner g mod n;
fin
si e est pair alors
    Retourner Expo_Rap_Rec(g^2, e/2, n) mod n;
sinon
    Retourner g*Expo_Rap_Rec(g^2, (e-1)/2, n) mod n;
fin
```

Question : Est-ce un algorithme Gauche-Droite ou Droite-Gauche ?

Code C :

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

unsigned int Expo_Rap_Rec(unsigned int g, unsigned int e, unsigned int n)
{
  if (e == 1) return g % n;
  if (e % 2 == 0) //test de parité
  {
    return Expo_Rap_Rec(pow(g, 2), e/2, n) % n;
  }
  else
  {
    return g*Expo_Rap_Rec(pow(g, 2), (e-1)/2, n) % n;
  }
}

int main(int argc, char** argv)
{
  unsigned int g, e, n;
  g = abs(atoi(argv[1]));
  e = abs(atoi(argv[2]));
  n = abs(atoi(argv[3]));
  printf("%u^%u mod %u = %u\n", g, e, n, Expo_Rap_Rec(g, e, n));
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall -lm
$ ./EXEC 2 3 5
2^3 mod 5 = 3
$ ./EXEC 3 9 97
3^9 mod 97 = 89
$ ./EXEC 5 8 11
5^8 mod 11 = 4
$ ./EXEC 15 9 679
15^9 mod 679 = 148
$
```

Algorithme : Exponentiation "Version Droite-Gauche"
Données : $\mathtt{g} \in \mathbb{Z}_n, \mathtt{e} \in \mathbb{Z}_n, \mathtt{n} \in \mathbb{N}^*$;
Sortie : Puiss-Mod(g, e, n) $= \mathtt{g}^{\mathtt{e}} \mod \mathtt{n}$;
Début
  Calculer $e_{bin} = (d_{k-1}, d_{k-2}, \cdots, d_1, d_0)$;
  $\mathtt{E_{mod}} = 1$; $\mathtt{Exp} = \mathtt{g}$;
  pour *pour i=0 jusqu'à k-1* **faire**
    si $d_i = 1$ **alors**
      $\mathtt{E_{mod}} = \mathtt{E_{mod}}\mathtt{Exp} \mod \mathtt{n}$
    fin
    $\mathtt{Exp} = \mathtt{Exp}^2 \mod \mathtt{n}$
  fin

Remarque : la variable Exp rassemble les puissances au carré successives de g tandis que la variable E_{mod} greffe Exp dans le résultat final quand sa puissance $d_i \neq 0$.

Exercice : faire la version Gauche-Droite.

Expressions Bien Parenthésées (EBP)

Expressions Bien Parenthésées (EBP)

• On s'intéresse aux chaînes de caractères dans lesquelles les parenthèses ouvrantes et fermantes sont bien distribuées :

- (ab(cd)) est bien parenthésée
- (a(bc)d) est bien parenthésée
- (ab))(cd n'est pas bien parenthésée

• En ignorant les caractères qui ne sont pas de parenthèses, on se ramène à étudier les chaînes constituées exclusivement de parenthèses.

• Une EBP (ici) c'est une chaîne de caractères (parenthèses) qui peuvent apparaître dans une expression arithmétique correcte, on dit que ce sont des parenthèses bien équilibrées, ou une chaîne de parenthèses bien équilibrée, ... Pour simplifier : EBP.

• On peut démontrer que le nombre d'expressions bien parenthésées contenant exactement n parenthèses ouvrantes est égal au nombre de Catalan $C(n)$:

- Le début de la suite $C(n)$ est
  $C(0) = 1, C(1) = 1, C(2) = 2, C(3) = 5, C(4) = 14$ etc.
- Pour tout n entier naturel :

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!}$$

• Les nombres de Catalan satisfont aussi la relation de récurrence :

$$C(0) = 1 \quad \text{et} \quad C(n+1) = \sum_{i=0}^{n} C(i)C(n-i) \quad \text{pour} \quad n \geq 0$$

- C(1) = C(0)C(0)
- C(2) = C(0)C(1) + C(1)C(0)
- C(3) = C(0)C(2) + C(1)C(1) + C(2)C(0)
- ...

Exercice : (À faire sur place)
Quel est le nombre de EBP pour n=3 ?
Donner toutes les EBP pour n=3.

Exercice : donner toutes les EBP pour n=3.

Solution :

1. ((( )))
2. (( ) ( ))
3. (( )) ( )
4. ( ) (( ))
5. ( ) ( ) ( )

Exercice : (à faire chez vous)
Écrire une fonction de "parsing", récursive ou itérative qui prenne en argument une chaîne de caractères composée de parenthèses et qui teste si cette chaîne est une EBP.

[Wiki] : *L'analyse syntaxique consiste à mettre en évidence la structure d'un texte, généralement une phrase écrite dans une langue naturelle, mais on utilise également cette terminologie pour l'analyse d'un programme informatique. L'analyseur syntaxique (parser, en anglais) est le programme informatique qui réalise cette tâche.*

Suggestions :

1. Input : S une chaîne de caractères.
2. Output : `Vrai` si S est une EBP, `Faux` sinon.
3. On parcourt la chaîne suivant une stratégie Gauche-Droite, on utilise un simple compteur qui est :
   - initialisé à 0
   - incrémenté ($+1$) si on rencontre une parenthèse ouvrante
   - décrémenté (-1) si on rencontre une parenthèse fermante
4. À quelle(s) condition(s) sur les valeurs successives de ce compteur la chaîne est bien une EBP ?
5. Utiliser ces indications pour écrire une fonction qui implémente une solution pour ce problème.

Bibliographie

- R. ERRA, "Cours 5 d'informatique", 1A-S2 2013-2014 ESIEA.
- Cormen, Leiserson, Rivest, Stein "Algorithmique", 3ème éd. DUNOD.

- http://fr.wikipedia.org/wiki/Exponentiation_modulaire