L3 MI / PROG
TP#3: Exponentiation rapide

2023-W38

Exercice 1 : Exponentiation rapide

Le but de cet exercice est d'implémenter deux algorithmes d'exponentiation rapide dans le corps $\mathbb{K} := \mathbb{Z}/536870909\mathbb{Z}$ des entiers modulo le nombre premier $536870909 = 2^{29} - 3$.

Pour tout l'exercice, il est conseillé de compiler votre programme avec l'option d'optimisation -O2 ou -O3.

On fournit les fonctions suivantes permettant de calculer la somme et le produit de deux nombres dans $\mathbb{K}$:

```
// In: 0 <= a,b < 536870909
// Out: 0 <= x < 536870909 t.q. x ~ a + b [536870909]
uint64_t add293(uint64_t a, uint64_t b)
{
    return ((a + b) % 536870909);
}

// In: 0 <= a,b < 536870909
// Out: 0 <= x < 536870909 t.q. x ~ a * b [536870909]
uint64_t mul293(uint64_t a, uint64_t b)
{
    return ((a * b) % 536870909);
}
```

Q.1 :

1. Expliquez pourquoi remplacer 536870909 ci-dessus par $1099511627689 = 2^{40} - 87$ permet de correctement calculer l'addition dans $\mathbb{K}' := \mathbb{Z}/1099511627689\mathbb{Z}$, mais ne permet pas de calculer correctement le produit.
2. Donnez un critère nécessaire et suffisant sur la valeur du module pour que les deux fonctions ci-dessus soient correctes.

Q.2 :

1. Écrivez une fonction :

```
// In: 0 <= a < 536870909
// In: 0 <= n < 2**64
// Out: 0 <= x < 536870909 t.q. x ~ a^n [536870909]
uint64_t slow_exp(uint64_t a, uint64_t n);
```

calculant naïvement a^n dans $\mathbb{K}$ grâce à $n-1$ produits (quand $n > 0$).

2. Écrivez une fonction de test vérifiant notamment les équivalences suivantes modulo 536870909 :
   — $2023^{37} \equiv 218334479$
   — $2023^{38} \equiv 382763819$
   — $200023^{37} \equiv 334175149$
   — $200023^{38} \equiv 140174291$
   — $2023^{10^9} \equiv 478131823$
3. Mesurez le temps d'exécution de votre fonction (par exemple en utilisant la commande `time` dans un shell UNIX [1]) pour une dizaine de valeurs de n allant de 10^4 à 10^9. Le coût de cette fonction est il exponentiel ou linéaire en fonction de la *taille* de l'argument n ?

Q.3 :

1. Écrivez une fonction :

```
// In: 0 <= a < 536870909
// In: 0 <= n < 2**64
// Out: 0 <= x < 536870909 t.q. x ~ a^n [536870909]
uint64_t fast_exp_rec(uint64_t a, uint64_t n);
```

   calculant a^n dans $\mathbb{K}$ efficacement et récursivement en exploitant les égalités :

$$a^n = \begin{cases} (a^{n \div 2})^2 & \text{si } n \text{ est pair} \\ a(a^{n \div 2})^2 & \text{si } n \text{ est impair} \end{cases},$$

   où $a \div b$ désigne le quotient dans la division euclidienne de a par b.
2. Écrivez une fonction de test vérifiant les mêmes équivalences que dans la question précédente, ainsi que $2023^{10^{18}} \equiv 10222641$.
3. Mesurez le temps d'exécution de votre fonction comme dans la question précédente (pour des valeurs de n allant jusqu'à 10^{18}). Le coût de cette fonction est il exponentiel ou linéaire en fonction de la *taille* de l'argument n ?

Q.4⋆ :

1. Écrivez une fonction :

```
// In: 0 <= a < 536870909
// In: 0 <= n < 2**64
// Out: 0 <= x < 536870909 t.q. x ~ a^n [536870909]
uint64_t fast_exp_iter(uint64_t a, uint64_t n);
```

   calculant a^n dans $\mathbb{K}$ efficacement et itérativement en exploitant le fait que si l'on écrit $n_{63}, \ldots, n_0 \in \{0, 1\}$ les chiffres de n en base 2 (c'est à dire que $n = \sum_{i=0}^{63} n_i 2^i$), on a dans $\mathbb{Z}$ l'égalité $a^n = \prod_{i=0}^{63} a^{n_i 2^i}$ (avec la convention qu'un produit vide vaut 1). On exploitera également le fait que la suite des puissances $a, a^2, a^4, \ldots a^{2^{63}}$ peut se calculer par mises au carré successives de a.

1. Si l'exécution est trop rapide pour obtenir de cette façon un résultat exploitable, il est suffisant dans le cadre de cet exercice d'en calculer une approximation en prenant la moyenne d'un grand nombre de répétitions.

2. Écrivez une fonction de test vérifiant les mêmes équivalences que dans la question précédente.
3. Mesurez le temps d'exécution de votre fonction comme dans la question précédente. Le coût de cette fonction est il exponentiel ou linéaire en fonction de la *taille* de l'argument n ?

Q.5 :

1. Dans le cadre général du développement logiciel, quel peut être l'intérêt de d'abord implémenter une fonction inefficace comme `slow_exp` avant des fonctions plus rapides de spécifications identiques ?
2. Quels autres tests de vos fonctions (plus complets que ceux du sujet) pourriez-vous envisager, exploitant notamment le fait que vous disposez de plusieurs fonctions de spécifications identiques ?

Q.6 : Un corollaire du petit théorème de Fermat est que pour tout $a \in [\![1, p-1]\!]$ avec p premier, on a l'équivalence $a^{p-1} \equiv 1 \mod p$.

1. Comment ce théorème peut-il être utilisé pour diminuer dans certains cas le coût de vos fonctions d'exponentiation en réduisant une de ses entrées modulo un nombre bien choisi ?
2. Utilisez ce théorème et l'une de vos fonctions d'exponentiation ci-dessus pour écrire une fonction :

   ```
   // In: 0 < a < 536870909
   // Out: 0 < x < 536870909 t.q. x * a ~ 1 [536870909]
   uint64_t inv293(uint64_t a);
   ```

   calculant l'inverse d'un élément non nul de $\mathbb{K}$.
3. Écrivez une fonction de test qui prend en entrée un entier $a \in [\![1, 536870908]\!]$, calcule son inverse modulo 536870909 et vérifie que le produit avec a dans $\mathbb{K}$ vaut bien 1.