Exponentiation rapide

L'objectif de ce chapitre est de découvrir l'exponentiation rapide. Cet algorithme est particulièrement utilisé en cryptographie symétrique, notamment pour échanger rapidement des clés de façon sécurisée. Nous allons donc dans un premier temps étudier une nouvelle stratégie de résolution de problème, puis nous étudierons plus particulièrement le protocole d'échange de clés de Diffie Hellman. Enfin nous aborderons l'exponentiation rapide afin de réaliser cet échange de clé avec des grands entiers.

Pré-requis : afin de comprendre ce chapitre, il est nécessaire de connaitre les complexités de référence et savoir évaluer la complexité d'un algorithme.

1 Préambule : parcours gauche-droite ou droite-gauche ?

La première partie de ce TD consiste à vérifier si un algorithme ne peut pas se dérouler plus rapidement suivant le « bout » par lequel on commence à le traiter. Nous allons étudier le problème du nombre miroir. Rappelons dans un premier temps la problématique.

On considère un entier $n = \overline{n_k n_{k-1} \dots n_1 n_0}^{(10)}$. Le miroir de n est l'entier $M(n) = \overline{n_0 n_1 \dots n_{k-1} n_k}^{(10)}$. Par exemple, le nombre miroir de 761 et 167.

La construction de ce nombre miroir peut se réaliser de deux façons différentes. La première consiste à lire l'entier n **de gauche à droite** pour construire l'entier $M(n)$ à partir des unités. En reprenant l'exemple précédent, on construit successivement les entiers 7, puis 67 et enfin 167 pour obtenir le miroir de 761. Inversement, un parcours **droite-gauche** permet de partir des unités de 761 pour construire son miroir. On construit ainsi les entiers 1, 16 et enfin 167.

TD **Exercice 1.**

2 Échanges de clés avec le protocole Diffie-Hellman

La cryptologie est une science permettant de communiquer façon confidentielle entre deux parties. Cette communication repose sur l'utilisation de clés, permettant le chiffrement et le déchiffrement du message. Il existe une branche de la cryptographie dite symétrique, dans laquelle les parties voulant communiquer doivent posséder la même clé secrète (un exemple de tel chiffrement est le chiffre de César, ou de Vigenère). Le chiffrement et le déchiffrement sont alors réalisés à l'aide de la même clé.

Le problème est le suivant : comment se mettre d'accord sur la clé secrète. L'échange de clés de Diffie-Hellman résout ce problème. En supposant que Po le kakapo veuille

envoyer un message secret [1] de façon confidentielle à Jo l'ibijau, les deux ibijaux créent leur secret commun à l'aide de l'algorithme suivant.

- Soient g un entier et n un premier, tous deux publics.
- Po possède la valeur secrète a (appelée *clé privée*) et Jo possède la valeur secrète b.
- Po calcule $g^a \mod n$ et l'envoie à Jo.
- Jo calcule $g^b \mod n$ et l'envoie à Po.
- Po calcule $(g^b)^a \mod n$.
- Jo calcule $(g^e)^b \mod n$.

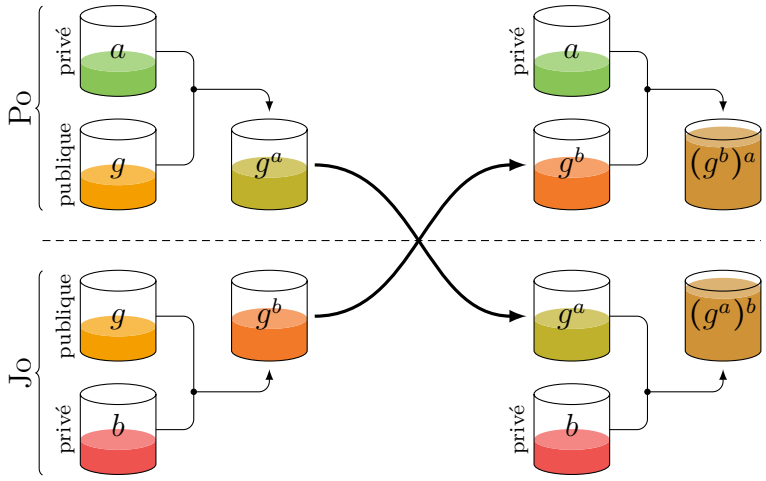

FIGURE 1 – Illustration de l'échange de clés Diffie-Hellman

Les deux ibijaux possèdent alors la même valeur $g^{ba} \mod n$ qui est la clé qui sera utilisée pour le chiffrement symétrique, comme illustré dans la figure 1.

L'objectif de cet atelier est donc de réaliser un véritable échange de clés utilisant Diffie-Hellman. En pratique, les valeurs g, n, a et b sont des entiers bien trop grands pour être stockés sur une variable de type `int`. En effet, pour assurer une sécurité suffisante et donc éviter de retrouver les clés privées, les exposants sont stockés sur plus de 160 bits, ce qui est beaucoup, vraiment, beaucoup plus grand qu'un entier sur 32 bits. Cela signifie également que le calcul g^a nécessite environ $a = 2^{160}$ multiplications, alors qu'il est actuellement impossible de réaliser plus de 2^{80} opérations sans attendre la fin de l'univers. Et Po et Jo ne veulent pas attendre la fin de l'univers pour améliorer l'égalité ibijau-kakapo.

2.1 Définition du problème

> La problématique est donc de calculer la valeur $g^e \mod n$, avec g, e et n des entiers, avec une complexité inférieure à la complexité linéaire traditionnelle.

Pour cela, nous allons découvrir au travers de ce TD l'*exponentiation rapide*. L'objectif final est de trouver l'algorithme le plus efficace pour réaliser cette exponentiation.

1. Les deux lascars préparent un coup d'état contre les institutions scientifiques classant les oiseaux par compétences.

Notons que nous ne savons pas encore manipuler des très grands entiers, donc nous nous contenterons de calculer des puissances sur 32 bits.

TD **Exercice 2.**

2.2 État de l'art

Dans un premier temps, il convient d'étudier l'algorithme actuel ainsi que quelques propriétés de la fonction puissance afin de trouver une optimisation efficace.

Ainsi, pour la suite de ce TP, et lorsque ce n'est pas explicitement précisé, nous nous contenterons de rédiger des algorithmes pour calculer g^e sans l'opération de modulo. Les programmes C en revanche réaliseront les calculs modulo n.

TD **Exercice 3.**

2.3 Propriétés mathématiques de l'exponentiation

Il faut donc trouver une solution pour réaliser cette exponentiation plus rapidement. Pour cela rappelons les quelques propriétés mathématiques connues de la mise à la puissance.

TD **Exercice 4.**

L'exercice précédent vient de vous faire prendre conscience qu'il semble possible de réaliser un calcul de complexité bien inférieure à la complexité linéaire évoquée auparavant.

TD **Exercice 5.**

2.4 Exponentiation tabulaire

TD **Exercice 6.**

Il semble intéressant de ré-écrire g^e en décomposant l'exposant sous la forme d'une somme de puissances de 2. Tout ceci doit naturellement vous faire penser à l'écriture à base b d'un entier. L'idée serait d'écrire e en base 2 afin de voir quelles puissances de 2 y sont présentes. Un calcul préalable des $g^{(2^k)}$ permettrait de limiter le nombre total de calculs. Ces puissances étant stockées dans un tableau, le nom exponentiation tabulaire est trouvé [2] ! L'exponentition tabulaire est détaillée dans l'algorithme 1.

Évaluons par exemple la complexité du calcul de g^{20} à l'aide de la méthode naïve et à l'aide de l'exponentiation tabulaire. L'exponentiation naïve réalise 19 produits. La complexité est donc bien linéaire en la valeur de l'exposant, ici 19 opérations. Concernant l'exponentiation tabulaire, d'un côté, on calcule les puissances de 2 successives de m et on les stocke dans un tableau.

Rang	0	1	2	3	4
Valeur	g	$g^2 = g \times g$	$g^4 = g^2 \times g^2$	$g^8 = g^4 \times g^4$	$g^{16} = g^8 \times g^8$

D'un autre côté, on sait, à l'aide de la décomposition binaire, que 20 s'écrit $\overline{10100}^{(2)}$. Cela signifie que $20 = 16 + 4$. Ainsi, $g^{20} = g^{16+4} = g^{16}g^4$. Comptons maintenant le nombre de

2. Vous trouverez peu d'informations sur le net à ce sujet puisque ce nom est créé de toutes pièces par votre enseignant.

produits réalisés. Il a fallu 4 produits pour créer le tableau, puis 1 de plus pour calculer $g^{20} = g^{16}g^4$, soit 5 produits en tout. Belle amélioration !

Algorithme 1 – Exponentiation tabulaire
Entrée. Entiers positifs g, e et n
Sortie. Entiers positifs g^e mod n

```
Entier res ← 1
Tableau d'entiers tab de taille N=⌈log2(e)⌉
Tableau binaire b ← DécompositionBinaire(e)
tab[0] ← g
Pour cpt allant de 1 à N faire
  tab[cpt] ← tab[cpt-1] × tab[cpt-1]
Fin Pour
Pour cpt allant de 0 à N-1 faire
  Si b[cpt] = 1 alors
    res ← res × tab[cpt]
  Fin Si
Fin Pour
Renvoyer res
```

TD **Exercices 7 et 8.**

3 Exponentiation rapide

Vous avez vu dans la section précédente qu'il était intéressant de stocker des puissances successives de $g^{(2^k)}$, avec $k \in \mathbb{N}$. Or l'algorithme peut être légèrement modifié pour éviter de stocker les puissances et les calculer au fur et à mesure.

Exemple 1. Prenons l'exemple de g^{20}. L'algorithme d'exponentiation tabulaire nous demandait de calculer au préalable g^2, g^4, g^8 et g^{16}. Mais il est aussi possible de les calculer au fur et à mesure. Rappelons que $20 = \overline{10100}^{(2)}$. Le tableau suivant indique l'évolution des variables au sein de l'algorithme d'exponentiation rapide.

Variable	Valeurs				
cpt	0	1	2	3	4
tmp	g	g^2	g^4	g^8	g^{16}
b[cpt]	0	0	1	0	1
res	1	1	g^4	g^4	g^{20}

La variable tmp joue donc le même rôle que le tableau tab de l'exponentiation tabulaire. La variable b[cpt] indique la valeur du bit n° cpt de la valeur 20. Le stockage des puissances successives $g^{(2^k)}$ est donc inutile. En effet, $g^{(2^{k+1})} = g^{(2.2^k)} = \left(g^{(2^k)}\right)^2$.

TD **Exercice 9.**

Notons qu'au cours de cet algorithme, on calcule à chaque étape la valeur $g^{(2^k)}$. Cela revient à mettre au carré la valeur précédente à chaque étape. Le calcul du résultat final nécessite uniquement une multiplication lorsque le bit de l'exposant en cours est à 1. Ce procédé de mise au carré systématique puis de multiplication occasionnelle a donné le

nom à cette méthode d'exponentiation rapide : *square and multiply*.

Il reste encore une étape à optimiser avant de vraiment pouvoir parler d'exponentiation rapide. L'algorithme *square and multiply* présenté précédemment nécessite de connaitre la décomposition binaire de l'exposant pour fonctionner. Or, on remarque que le parcours des bits est effectué de la droite vers la gauche. Il est donc relativement simple de fusionner l'algorithme de décomposition binaire et l'algorithme d'exponentiation. Le tout est résumé dans l'algorithme ci-dessous.

Algorithme 2 – Square and multiply
Entrée. Entiers positifs g, e
Sortie. Entiers positifs g^e

```
Entier res ← 1
Tant que e>0 faire
    Si e est impair alors
        res ← res × g
    Fin Si
    g ← g²
    e ← ⌊e/2⌋
Fin Tant que
Renvoyer res
```

TD **Exercice 10.**

4 Synthèse

Vous venez de rédiger et d'implémenter trois algorithmes vous permettant de calculer $g^e \mod n$. Il s'agit des algorithmes d'exponentiation naïve, tabulaire et rapide. Résumons maintenant leurs caractéristiques.

- **Expo. naïve** : très simple à implémenter : calcule $g \times g \times \cdots \times g$ avec e produits. La complexité est donc linéaire en e. Programmation récursive terminale simple à mettre en œvre.
- **Expo. tabulaire** : stocke les puissances successives de $g^{(2^k)}$ afin de diminuer le nombre total de produits à réaliser. Complexité logarithmique en e mais besoin d'espace de stockage et besoin de la décomposition binaire de l'exposant.
- **Expo. rapide** : calcul des $g^{(2^k)}$ interne à l'algorithme, décalage de l'exposant à chaque étape pour regarder la valeur des bits au fil de l'algorithme. Complexité logarithmique.

Conservez bien vos résultats et programmes pour le second semestre. Un challenge vous sera proposé afin de découvrir ce que Po et Jo se sont échangé.

5 Pour aller plus loin

5.1 Nombre de palindromes

La première partie de cet atelier consistait à calculer le miroir d'un nombre. Notons que la définition du miroir impliquait une écriture en base 10 des entiers. Que se passe-t-il si le miroir est calculé en base 2 ? Par exemple le miroir en base 10 de 7 est $M_{10}(7) = 7$, et le miroir binaire de $7 = \overline{111}^{(2)}$ est $M_2(7) = \overline{111}^{(2)} = 7$. Le miroir à base 10 de l'entier

7 est donc identique à son miroir en base 2. Mais l'entier 7 est particulier puisque c'est un palindrome en base 10 comme en base 2. Notons que ce n'est pas le cas pour 4 par exemple : $M_{10}(4) = 4 \neq M_2(4) = 1$.

TD **Exercice 11.**

5.2 Exponentiation modulaire, version arithmétique

Ce problème que vous voyez en informatique doit certainement vous rappeler les exercices d'arithmétique dans lesquels il était demandé de calculer a^b modulo n, avec b un grand nombre. Cet exercice va donc être l'occasion de revoir le cours d'algèbre afin de le coder.

L'algorithme

Dans un premier temps, il vous est demandé d'écrire l'algorithme correspondant au calcul de $a^b \mod n$. Nous vous rappelons simplement que cet algorithme consiste à :

- Chercher k, une puissance *sympa* de a modulo n.
- Réaliser la division euclidienne de b par k.
- Évaluer $a^b \mod n$ en utilisant k et la division euclidienne précédemment calculée.

La complexité

Maintenant que cet algorithme est créé, vous pouvez calculer sa complexité. Comme tous les calculs de ce genre, vous devez dans un premier temps identifier vos entrées et le N en question. Par la suite, identifiez le nombre de boucles réalisées en fonction de ce N, les calculs constants etc. et donnez la notation de Landau associée. Cet algorithme est-il plus intéressant que l'exponentiation rapide précédemment codée ?

Le code

Dernier exercice, vous vous doutez de ce que l'on va vous demander ... Il s'agit bien évidemment de coder cet algorithme. N'oubliez pas de prévoir un tableau permettant de stocker les puissances successives de a modulo n qui vous serviront à nouveau dans votre calcul final.