TP #7 — Calcul de logarithme discret

L'objectif de ce TP est d'implémenter l'algorithme du *baby-step/giant-step* pour le calcul du logarithme discret. Cette tâche est importante en calcul formel et en cryptographie, où son coût important (sous certaines hypothèses, notamment celle de ne pas disposer d'ordinateur quantique efficace) est exploité pour concevoir des systèmes efficaces et sûrs.

On définit le problème du calcul de logarithme discret de la façon *ad hoc* suivante : soit $x \in [\![1, 4\,294\,967\,290]\!]$ (sachant que $4\,294\,967\,291 = 2^{32} - 5$ est un nombre premier), on cherche $e \in [\![0, 4\,294\,967\,289]\!]$ (unique, garanti d'exister) tel que $2^e \equiv x \mod 4\,294\,967\,291$.
Par un léger abus de notation, on utilisera dans toute la suite $\mathbb{F}_{325}$ et $\mathbb{F}_{325}^\times$ pour désigner les intervalles $[\![0, 4\,294\,967\,290]\!]$ et $[\![1, 4\,294\,967\,290]\!]$ respectivement, ainsi que $\mathcal{F}_{325}$ pour désigner $[\![0, 4\,294\,967\,289]\!]$.

Préliminaires

On définit les variables globales en lecture seule suivantes :

```
const uint64_t mod325 = 4294967291; // 2^32 - 5 ; premier
const uint32_t base325 = 2; // base pour le logarithme
const uint32_t sqrt325 = 65536; // sqrt(2^32 - 5) arrondi
                                // vers le haut
```

Celles-ci ont une portée *fichier* et une durée de vie *statique* : elles peuvent être utilisées dans toute fonction du programme.

1. Écrivez une fonction C de signature :
   `uint32_t mul325(uint32_t x, uint32_t y)`
   qui pour $x, y \in \mathbb{F}_{325}$ renvoie l'unique $z \in \mathbb{F}_{325}$ tel que $xy \equiv z \mod 2^{32} - 5$.
   Prenez garde aux dépassements de capacité !
2. Écrivez (ou reprenez depuis un TP précédent) une fonction C de signature :
   `uint32_t pow325(uint32_t x, uint32_t e)`
   qui pour $x \in \mathbb{F}_{325}$ et e entier naturel quelconque renvoie l'unique $y \in \mathbb{F}_{325}$ tel que $y \equiv x^e \mod 2^{32} - 5$.
   Cette fonction devra avoir un coût au plus linéaire en la taille de ses arguments.
   Prenez garde aux dépassements de capacité !

Résolution naïve du problème

On peut simplement résoudre le problème du logarithme discret par recherche exhaustive : soit $x \in \mathbb{F}_{325}^\times$, on teste pour tout $e \in \mathcal{F}_{325}$ si $2^e \equiv x \mod 2^{32} - 5$

jusqu'à trouver une solution.

3. Écrivez une fonction C de signature :
   `uint32_t dlog325(uint32_t x)`
   qui utilise cette approche pour résoudre le problème.
4. Quel est son coût dans le pire cas, en fonction de la taille de ses entrées ? À supposer que l'on ne connaisse pas de meilleur algorithme pour le résoudre, justifiez le fait que ce problème puisse-t être qualifié de « coûteux ».
5. Testez, *via* une fonction de test dédiée (qui utilise par exemple de l'aléa pour générer ses cas de test).
   Commencez par ne pas faire trop de tests, et soyez patients.

Intermède : optimisations du compilateur

Les compilateurs C modernes (pour une interprétation généreuse de « moderne ») permettent d'*optimiser* les performances du programme compilé, les améliorant parfois de façon considérable. La façon la plus simple d'activer de telles optimisations est *via* un niveau d'optimisation global, renseigné par L'option `-O` (à ne pas confondre avec `-o`…). Le second niveau (« `-O2` ») est généralement un bon compromis, mais il existe également typiquement un niveau trois (et des optimisations qu'il faut activer séparément, par exemple `-funsafe-math-optimizations`, qui contrairement à ce que son nom laisse penser ne sont pas des optimisations « *fun & safe* » :(

6. Compilez votre programme avec l'option `-O2` et aucun *sanitizer*, et constatez la différence de temps d'exécution de vos tests par rapport à l'absence d'optimisation.

N.B. L'utilisation d'optimisation complique souvent les tâches de débuggage. Dans un contexte professionnel, les logiciels sont typiquement compilés avec des options différentes pour la phase de développement (par exemple pas d'optimisation, présence de symboles aidant à l'utilisation d'un débugueur, utilisation de *sanitizers*…) et la phase de déploiement (par exemple avec optimisation, désactivation des `asserts`, pas de *sanitizer*…). Dans le cadre des TPs, nous nous trouvons la plupart du temps en « phase de développement », mais il se peut comme ici que la performance soit aussi parfois importante.

Résolution par l'algorithme du *baby-step/giant-step*

On peut grandement améliorer l'efficacité de la résolution du problème du logarithme discret en utilisant l'algorithme du *baby-step/giant-step*. Celui-ci est un cas particulier d'algorithmes de type « rencontre au milieu » (en anglais : *meet-in-the-middle*, souvent écrit *MITM*), dont nous aurons l'occasion de reparler au

second semestre. Le principe de base de tels algorithmes est d'exprimer leur solution en fonction d'un élément commun (une *collision*) entre deux ensembles. Typiquement, si l'on cherche un élément $e \in \mathcal{S}$ dans un certain ensemble de taille N, une technique *MITM* visera à créer deux ensembles $\mathcal{S}_1$ et $\mathcal{S}_2$ de taille environ $\sqrt{N}$ tels qu'il existe $e' \in \mathcal{S}_1 \cap \mathcal{S}_2$ dont la connaissance permet de déduire e, et que cette intersection soit de petite taille. Si la création de $\mathcal{S}_1$, $\mathcal{S}_2$ et la recherche de toutes les collisions entre les deux ensembles se font pour un coût linéaire en leur taille, on obtient alors un algorithme de coût $O(\sqrt{N})$ au lieu de $O(N)$ pour une recherche exhaustive.

Dans le cas de la recherche de logarithme discret, cette technique se réalise de la façon suivante. On pose $N = 2^{32} - 5$, et $I = 65536 = \lceil \sqrt{2^{32} - 5} \rceil$, et soit $x \in \mathbb{F}^x_{325}$ l'élément dont on recherche le logarithme discret e, on définit alors :

— $\mathcal{S}_{\text{giant}} = \{2^{kI} \mid 0 \leq kI < N\}$ (qui ne dépend pas de x)
— $\mathcal{S}_{\text{baby}} = \{2^{e+s} \mid 0 \leq s < I\}$ (où 2^{e+s} peut se calculer rien qu'en connaissant x (mais pas e) comme $x \times 2^s$)

où toutes ces valeurs sont prises « modulo $2^{32} - 5$ ». Ces deux ensembles sont bien de taille environ $\sqrt{N}$, et la connaissance de $e' \in \mathcal{S}_{\text{giant}} \cap \mathcal{S}_{\text{baby}}$ permet de résoudre le problème. En effet, un tel e' s'écrit à la fois comme $2^{k'I}$ pour une certaine valeur connue $k'I$ et comme $2^{e+s'}$ pour une certaine valeur connue s' ; on retrouve alors e par une simple soustraction $k'I - s'$ (quand $k'I > 0$; sinon s' donne directement la réponse). Enfin, un tel e' est garantit d'exister puisque par construction de $\mathcal{S}_{\text{giant}}$ il est d'intersection non vide avec tout ensemble d'au moins I éléments d'exposants consécutifs.

Construction de $\mathcal{S}_{\text{giant}}$

On commence par implémenter la construction de $\mathcal{S}_{\text{giant}}$, qui ne dépend pas du logarithme discret recherché. Concrètement, on doit stocker deux choses pour chaque élément de cet ensemble : sa valeur $\in \mathbb{F}_{325}$ (pour pouvoir la comparer plus tard aux éléments de $\mathcal{S}_{\text{baby}}$) et son exposant (ou logarithme) $\in \mathcal{F}_{325}$ (pour pouvoir calculer e une fois une collision trouvée).
Pour cela on va utiliser le type :

```
struct u32_pair
{
    uint32_t exp;
    uint32_t val;
};
```

7. Écrivez une fonction C de signature :
   `void init_gs(struct u32_pair giant_steps[sqrt325])`
   qui initialise son argument en y écrivant l'élément d'exposant kI à la case k.

Tri de $\mathcal{S}_{\text{giant}}$

Pour pouvoir plus tard efficacement implémenter la recherche de collision entre $\mathcal{S}_{\text{giant}}$ et $\mathcal{S}_{\text{baby}}$, une solution simple consiste à trier $\mathcal{S}_{\text{giant}}$ suivant les valeurs `.val` de ses éléments, puis à effectuer une recherche par dichotomie.

8. En supposant que l'on souhaite calculer un unique logarithme discret, expliquez pourquoi utiliser un tri de coût quadratique en la longueur de son entrée annulerait ici tout l'intérêt de l'algorithme du *baby-step/giant-step*.

L'un des algorithmes de tri les plus adaptés à ce contexte est celui du *tri par base* (en anglais : *radix sort*). Celui-ci généralise le tri par comptage (déjà vu dans des sujets précédents) au cas où la valeur maximale des éléments (ici : $2^{32}-4$) est trop grande pour que ce dernier soit efficace. L'idée du tri par base est la suivante : pour trier un tableau de N éléments de valeur $\in [\![0, K-1]\!]$, on choisit une *base* B satisfaisant (idéalement) les conditions suivantes :

— $B \leq N$ et idéalement, plus petit que le nombre d'éléments pouvant être stockés dans le premier (ou second) niveau de cache ;
— $\log_B K$ est « petit ».

Ensuite, on effectue $\lceil \log_B K \rceil$ itérations sur le tableau, où la $i$ème itération trie les éléments *en considérant uniquement le* i*ème chiffre de leur écriture en base* B, en partant du chiffre de poids faible (c'est le plus simple). Pour que ceci soit correct, il est nécessaire que chacun de ces tris soit *stable*, c'est à dire préserve l'ordre relatif de deux éléments égaux (à l'étape considérée). La correction peut alors se prouver par l'invariant qu'à l'issue de la $i$ème itération, les éléments du tableaux sont triés « modulo » B^i (c'est à dire, en considérant uniquement leurs i chiffres de poids faible en base B).
On donne un exemple d'exécution jouet pour $K = 1000$ et $B = 10$:

— Tableau initial : [562, 053, 747, 484, 688, 909, 528, 606, 661, 017]
— Après itération 1 : [66**1**, 56**2**, 05**3**, 48**4**, 60**6**, 74**7**, 01**7**, 68**8**, 52**8**, 90**9**]
— Après itération 2 : [6**0**6, 9**0**9, 0**1**7, 5**2**8, 7**4**7, 0**5**3, 6**6**1, 5**6**2, 4**8**4, 6**8**8]
— Après itération 3 : [**0**17, **0**53, **4**84, **5**28, **5**62, **6**06, **6**61, **6**88, **7**47, **9**09]

On peut implémenter le tri par base en utilisant n'importe quel tri stable au sein de chaque itération, mais le plus courant est d'utiliser un tri par comptage. Celui-ci doit cependant être adapté au fait qu'il est utilisé en sous-routine d'un tri par base : on ne peut plus simplement « réécrire » le tableau avec le bon nombre d'éléments de chaque valeur. On esquisse une telle version :

— Phase 1 : on compte le nombre d'occurrences de chaque élément du tableau (comme pour n'importe quel tri par comptage), que l'on stocke dans un tableau d'occurrences *occ*.
— Phase 2 : pour *i* de *1* à la longueur de *occ* - 1, on ajoute successivement *occ[i-1]* à *occ[i]*. Après cette étape, *occ[i]* vaut un de plus que l'indice le

plus grand où l'on trouve (éventuellement) un élément de valeur i dans le tableau trié (ou de façon (presque) équivalente, le plus petit indice où l'on peut éventuellement trouver une valeur $i+1$).

— Phase 3 : on crée un nouveau tableau vierge, puis en parcourant le tableau initial *depuis la fin* (c'est important pour la stabilité) on y écrit chaque élément (dans son intégralité) au bon endroit.

9. Écrivez une fonction C de signature :

```
void sort_gs(struct u32_pair giant_steps[sqrt325])
```

qui trie son argument suivant les valeurs `.val` de ses éléments, en utilisant un tri par base avec base `sqrt325` (il suffira donc d'effectuer deux étapes de tri par comptage, qui pourront être écrites explicitement (sans boucle)). Cette fonction devra utiliser `malloc` pour la création des (ou du) « tableaux » temporaire.

10. Quel est le coût (asymptotique, dans le pire cas, à constantes près) du tri par base couplé à un tri par dénombrement comme ci-dessus, en fonction de N, K, B ?

11. Testez, *via* une fonction de test dédiée. (On pourra se contenter d'un test vérifiant que l'argument de `sort_gs` est trié après appel.)

Recherche efficace dans $\mathcal{S}_{\text{giant}}$

12. Écrivez une fonction C de signature :

```
struct u32_pair
bfind_gs
(struct u32_pair giant_steps[sqrt325], uint32_t x)
```

qui utilise une recherche dichotomique pour trouver et renvoyer un éventuel élément de `giant_steps` dont la valeur `.val` est égale à `x`. Si aucun tel élément n'existe, cette fonction renverra une `struct u32_pair` dont les deux champs sont à `0` (pourquoi cela n'est-il pas ambigu ?).

13. Testez *via* une fonction de test dédiée.

Un petit pas pour le bébé, un grand pas pour le géant

14. Écrivez une fonction C de signature :

```
uint32_t bsgs325(uint32_t x)
```

qui résout le problème de calcul du logarithme discret dans $\mathbb{F}_{325}^{\times}$ en utilisant l'algorithme de *baby-step/giant-step*.
Cette fonction devra construire (et manipuler) $\mathcal{S}_{\text{giant}}$ en utilisant les fonctions précédemment écrite, mais ne devra pas (explicitement) construire $\mathcal{S}_{\text{baby}}$.
Toutes les allocations de « tableaux » devront se faire avec `malloc`.

15. Testez, *via* une fonction de test dédiée.

16. Vérifiez (du mieux que vous le pouvez) que votre programme ne possède pas de fuite mémoire.

Compromis temps-mémoire

On peut exprimer l'algorithme du *baby-step/giant-step* (comme beaucoup d'algorithmes de type *MITM*) plus généralement comme un *compromis temps-mémoire* : soit M la quantité mémoire (à constante près) disponible pour résoudre un problème de calcul de logarithme discret dans un ensemble de taille N, le *baby-step/giant-step* tel que présenté ci-dessus ne peut être utilisé que si $\sqrt{N} = O(M)$.

17. Proposez (sans l'implémenter) une modification de l'algorithme qui fonctionne pour toute valeur de M.

Soit T le coût temporel (pire cas, asymptotique à constantes et facteurs logarithmiques près) de votre algorithme pour une certaine valeur de M.

18. Pourquoi a-t'on nécessairement $T \geq M$?
19. Donnez une expression de N en fonction de T et M (avec $M \leq T$).

Cette expression donne le *compromis temps-mémoire* réalisé par le *baby-step/giant-step*.

Raffinements

20. Expliquez comment adapter l'algorithme du *baby-step/giant-step* si l'on sait que le logarithme de x appartient à un certain intervalle R (de longueur possiblement très inférieure à N, le nombre total d'exposants).
21. De même si l'on sait qu'il a peu de chiffres non nuls dans son écriture en base deux. (Au besoin, faites l'hypothèse que vous savez comment partitionner en deux les chiffres de l'exposant de façon à ce que chaque partition contient la moitié de ses chiffres non nuls.)

Dans la vraie vie

On peut montrer que dans un certain sens l'algorithme du *baby-step/giant-step* (avec mémoire *a priori* illimitée) est optimal : il n'existe pas d'algorithme plus rapide pour calculer un logarithme discret. Ce résultat a cependant trois limitations :

— Il ne concerne que les algorithmes « génériques », qui fonctionnent *de la même façon* pour n'importe quel *groupe*. Par exemple, il existe des algorithmes plus efficaces pour la recherche de logarithme discret dans les groupes (additifs ou multiplicatifs) des entiers modulo un nombre premier (comme par exemple le groupe considéré dans ce TP).
— Comme on l'a vu, le coût mémoire du *baby-step/giant-step* peut être important (et limitera en premier son utilisation pratique). Il existe d'autres

algorithmes comme l'algorithme ρ (basé sur les mêmes idées) ou des kangourous (un peu différent) qui ont un coût temporel comparable et un coût en mémoire (très) faible.

— Il ne s'applique que pour des algorithmes exécutés sur des ordinateurs classiques, et pas par exemple sur des ordinateurs « quantiques ». Dans ce dernier cas, on connaît des algorithmes (beaucoup) plus efficaces. Mais on ne dispose pas pour l'instant d'ordinateur quantique suffisamment puissant pour faire tourner ces algorithmes dans la vraie vie.