TD #2 — Exponentiation rapide, tableaux &c.

Exercice 1. *Exponentiation entière (rapide)*

En C, il n'existe pas d'opérateur d'exponentiation prédéfini pour les types entiers : le langage (ou sa bibliothèque standard) ne permet pas directement de calculer a^b pour a, b de type `unsigned`, ou `uint32_t`, ou `uint64_t`… Nous allons voir comment ce calcul peut néanmoins être effectué efficacement en général, en utilisant un algorithme d'*exponentiation rapide*. Cet algorithme est très général (*cf.* le prochain exercice) et est à connaître absolument. Nous le verrons (sous diverses formes) de nombreuses fois au cours de l'année.

1. Écrivez une fonction C de signature :
   `uint64_t sloexp(uint64_t base, uint64_t e)`
   qui calcule et renvoie $base^e$ à condition que ce nombre soit plus petit que `UINT64_MAX`, et renvoie un résultat arbitraire sinon. (Autrement dit, on ignore toute subtilité liée aux limites de représentation des types entiers de base.) Cette fonction devra utiliser l'algorithme naïf consistant à calculer $e-1$ produits.

L'algorithme utilisé dans cette fonction *sloexp* n'est pas très efficace ; nous allons voir comment effectuer le même calcul en utilisant environ $\log e$ produits seulement (soit exponentiellement mieux que précédemment) ! Pour cela, on exploite les deux observations suivantes :

— si l'on écrit $e_k, \ldots, e_0 \in \{0,1\}$ les chiffres de e *en base 2* (c'est à dire que $e = \sum_{i=0}^{k} e_i 2^i$), on a dans $\mathbb{N}$ l'égalité $x^e = \prod_{i=0}^{k} x^{e_i 2^i}$ (avec la convention qu'un produit vide vaut 1) ;
— la suite des puissances $x, x^2, x^4, \ldots x^{2^k}$ peut se calculer en seulement k produits par mises au carré successives de x, x^2, etc.

EXEMPLE. Soit $e = 10 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$, on a $x^{10} = x^{2^3} \times x^{2^1}$. On peut donc calculer l'exponentiation rapide de x à 10 en calculant successivement x^{2^1}, x^{2^2}, x^{2^3} (ce qui ne nécessite que 3 produits) et en « accumulant » les seules valeurs x^{2^1} et x^{2^3} dans une variable renvoyée à la fin de l'itération (ce qui ne nécessite qu'un produit), pour un total de $4 < 9$ produits.

Alternativement, ce même algorithme peut être vu comme exploitant le fait « récursif » que pour x, $e \in \mathbb{N}$, on a l'égalité $x^e = (x^{e \div 2})^2 \times x^{e \bmod 2}$ (ou de façon équivalente $x^e = (x^2)^{e \div 2} \times x^{e \bmod 2}$), où l'on note $e \div 2$ et $e \bmod 2$ respectivement le quotient et le reste de la division de e par 2.

EXEMPLE. Soit $e = 10$ on a $x^{10} = (x^5)^2$, et $x^5 = (x^2)^2 \times x$; l'exponentiation rapide de x à 10 peut donc se calculer en calculant x^2, $(x^2)^2$, multiplier cette dernière quantité par x (donnant x^5) et élever le résultat au carré.

2. Utilisez l'algorithme suggéré par les observations ci-dessus pour calculer 3^6.
3. Écrivez une fonction C de signature
   `uint64_t fastexp(uint64_t base, uint64_t e)`
   qui calcule et renvoie $base^e$ à condition que ce nombre soit plus petit que `UINT64_MAX`, et renvoie un résultat arbitraire sinon. Cette fonction devra utiliser l'algorithme d'exponentiation rapide déduit des observations ci-dessus.

   CONSEIL. On peut approcher l'écriture de cette fonction de multiples façons ; un raisonnement « global » consiste à directement traduire les deux observations ci-dessus, mais il est également possible de définir un *invariant* conservé entre chaque itération de boucle, et un *variant* décidant de sa terminaison ; il « suffit » alors d'écrire la boucle pour faire décroître le variant et satisfaire l'invariant, en utilisant pour cela la caractérisation « récursive ».
4. Montrez que le nombre d'opérations arithmétiques effectuées par votre fonction est inférieur à $C_1 \log e + C_0$, avec C_1 et C_2 des constantes entières que l'on ne cherchera pas à expliciter. Pour cela, on se contentera de montrer que le nombre d'itérations de la boucle de la fonction *fastexp* est majoré par $C_1 \log e$.

Exercice 2. *Double-and-add*

On souhaite développer un algorithme de multiplication des nombres entiers naturels utilisant un ensemble restreint d'opérations « élémentaires » les plus « simples » possibles, soit l'addition et la multiplication et la division par 2.

1. Quel peut être l'intérêt de se restreindre à cet ensemble d'opérations en général, et en particulier quand les nombres manipulés sont représentés en base deux ?
2. Écrivez une fonction C de signature
   ```
   uint64_t damul(uint64_t a, uint64_t b)
   ```
   qui calcule et renvoie $a \times b$ à condition que ce nombre soit plus petit que `UINT64_MAX`, et renvoie un résultat arbitraire sinon. Les seules opération arithmétiques que cette fonction peut utiliser sont la multiplication et la division par 2, et elle devra exploiter une approche par « exponentiation rapide ».

N.B. Le nom *double-and-add* de l'algorithme de cet exercice est celui sous lequel il est généralement connu dans le contexte de la cryptographie (sur courbes elliptiques), mais il en possède de nombreux autres (« multiplication des paysans russes », « multiplication égyptienne », « multiplication éthiopienne »…).

Exercice 3. *Factorielle*

1. Écrivez une fonction C de signature :
   ```
   void fact(size_t n, uint64_t f[n])
   ```
   qui modifie son argument f de sorte qu'à l'issue de son exécution l'on a $f[i] = i!$, à condition que $i!$ soit représentable par un type `uint64_t`.
   Cette fonction devra utiliser au plus n multiplications.
2. Quelle est la valeur maximale que peut prendre le paramètre n telle qu'aucun des calculs effectués par votre fonction ne résulte en un *wraparound* ou un dépassement de capacité ?

Exercice 4. *Miroir*

1. Écrivez une fonction C de signature :
   ```
   void rev(size_t an, unsigned a[an])
   ```
   qui modifie son argument a en son « miroir ». C'est à dire que si l'on désigne les an éléments de a par $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ (dans l'ordre des indices croissants) avant appel de la fonction, ceux-ci seront $\alpha_{n-1}, \alpha_{n-1}, \ldots, \alpha_1, \alpha_0$ après appel.

Exercice 5. *Recherche de collision*

1. Écrivez une fonction C de signature :
   ```
   unsigned col(size_t an, unsigned a[an], size_t bn, unsigned b[bn])
   ```
   qui recherche une *collision* entre les éléments contenus dans ses arguments a et b, c'est à dire un élément apparaissant à la fois dans les deux tableaux. Si (au moins) une collision est trouvée, *col* doit renvoyer l'élément formant cette collision, et elle renvoie `0` sinon.
2. Quel est un défaut des spécifications données pour *col* ci-dessus ?
3. Proposez une amélioration.
4. Écrivez une fonction *col2* (dont vous préciserez les spécifications) qui implémente votre amélioration.

On suppose maintenant que les entrées des tableaux a et b sont triées par ordre croissant.

5. Écrivez une fonction *col3* (dont vous préciserez les spécifications) qui recherche une collision entre les éléments contenus dans a et b, et en renvoie une si elle en trouve une. Cette fonction devra être « plus efficace » que *col2*.

Exercice 6. *Recherche de point fixe*

1. Écrivez une fonction C de signature :
   ```
   bool fp(size_t an, int a[an])
   ```
   qui prend en paramètre un tableau a d'éléments deux-à-deux distincts et renvoie `true` s'il existe un entier i tel que `a[i] == i`, et `false` sinon.
2. Comment pourrait-on modifier la signature de *fp* afin d'éviter d'éventuels problèmes à l'exécution ?

On suppose maintenant que les entrées de a sont triées par ordre croissant, en plus d'être deux-à-deux distinctes.

3. Comment pouvez-vous adapter les spécifications de *fp* dans le cas où elle ferait une telle hypothèse ?
4. Écrivez une nouvelle version « *fp2* » de *fp* faisant cette hypothèse ; cette version devra être « plus efficace » que *fp*.