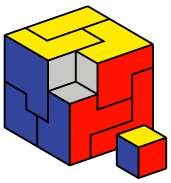
Prologin 2021



Concours national d'informatique

Épreuve écrite d'algorithmique
Online

21 février 2021

Crêpage de chignon

1 Préambule

Bienvenue à **Prologin**. Ce sujet est l'épreuve écrite d'algorithmique et constitue la première des trois parties de votre épreuve régionale. Sa durée est de 3 heures. Par la suite, une épreuve de programmation sur machine (3 heures 30).

Conseils

— Lisez bien tout le sujet avant de commencer.
— **Soignez la présentation** de votre copie.
— N'hésitez pas à poser des questions.
— Si vous avez fini en avance, relisez bien, ou préparez votre présentation pour l'entretien.
— N'oubliez pas de passer une bonne journée.

Remarques

— Le barème est donné à titre indicatif uniquement.
— Indiquez lisiblement vos nom et prénom, la ville où vous passez l'épreuve et la date en haut de votre copie.
— Tous les langages sont autorisés, veuillez néanmoins préciser celui que vous utilisez.
— Ce sont des humains qui lisent vos copies : laissez une marge, aérez votre code, ajoutez des commentaires (**seulement** lorsqu'ils sont nécessaires) et évitez au maximum les fautes d'orthographe, sinon ça va barder.
— Le barème récompense les algorithmes les plus efficaces : écrivez des fonctions qui trouvent la solution le plus rapidement possible.
— Si vous trouvez le sujet trop simple, relisez-le, réfléchissez bien, puis dites-le-nous, nous pouvons ajouter des questions plus difficiles.

2 Sujet

Préparer un concours d'informatique n'est vraiment pas de tout repos. Il faut trouver les fonds, gérer le stock de crêpes, écrire des sujets et **surtout** s'assurer que la bonne entente règne entre les organisateur·trices.
Cette année, Prologin a décidé de mettre en place une répartition des tâches "par crêpes" : chaque tâche vaut un certain nombre de crêpes, et la présidence de Prologin distribue les crêpes à la fin de la finale, lors d'un grand banquet.
Il se trouve que durant la finale, motivé·es par la salle de jeux vidéo, les organisateur·trices ont tendance à se refiler les tâches qu'ils ne veulent pas les un·es aux autres, et que ces échanges constants de crêpes compliquent énormément la répartition lors du banquet final [1].
Cette année, une nouvelle organisation est prévue pour gérer les échanges de crêpes entre les organisateur·trices, et Prologin a besoin de vous [2] pour la mettre en oeuvre.

2.1 Registre public et signature numérique

La solution trouvée par la présidence est un ordinateur posé sur le frigo [3] du local Prologin. N'importe qui peut l'utiliser et ajouter des échanges de crêpes. N'importe qui peut consulter l'ensemble des échanges de crêpes à un moment donné. Prologin a besoin de vous pour programmer l'application qui va gérer les échanges.
Une transaction est définie comme un triplet *(émetteur, receveur, montant)*. Le registre est un ensemble de transaction. Les transactions sont ajoutées au registre une par une, donc dans l'ordre chronologique.

Registre simple

Pour commencer, on propose que chacun·e écrive ses propres transactions dans le registre.
Un émetteur et un récepteur sont, dans une premier temps, une simple chaîne de caractère avec le nom de l'orga [4].

Question 1 (2 points)

> Décrire une structure de donnée pour le registre et les transactions. Attention, lisez bien les questions suivantes avant de répondre car vous ré-utiliserez votre structure de données par le suite.

Question 2 (1 point)

> Écrire une fonction qui prend en argument un émetteur, un receveur, un montant et un registre, et qui ajoute la transaction au registre.

De manière régulière, la présidence de Prologin attribue des crêpes [5] à des organisateur·trices en particulier pour services rendus. Ces attributions de crêpes seront représentées par des transactions *("bureau", orga, montant)*.

Question 3 (2 points)

> Écrire une fonction qui calcule pour un individu donné le montant total de crêpes qu'il ou elle possède. Ce montant peut être positif ou négatif (car la personne a pu donner plus de crêpe qu'il ou elle en a reçu).

Question 4 (1 point)

> Écrire une fonction qui calcule combien de crêpes au total la présidence devra fournir lors du banquet.

Signatures numériques

Des informateur·ices anonymes ont contacté·es la présidence de Prologin. Un sordide trafic de crêpes est à l'oeuvre au sein des organisateur·trices, basé sur la falsification du registre de transaction de crêpes. Il semblerait que certain·es organisateur·trices malhonnêtes écrivent des transactions à la place d'autres pour leur voler des crêpes !

1. D'où l'expression "se crêper le chignon", inventé par Joseph Marchand à la fin d'une finale Prologin
2. OUI, VOUS !
3. Rempli de crêpes, mais ça n'a rien à voir avec le sujet.
4. Les organisateur·trices peuvent avoir des noms très très longs
5. De manière très équitable, ce sont des gens responsables.

Pour éviter que cela se reproduise, on introduit le concept de signature numérique.
Une signature numérique est un mécanisme cryptographique permettant de vérifier l'authenticité d'un document (ou ici d'une transaction) grâce à une paire de clefs :

- **Une clef secrète**, que chaque organisateur·trice garde pour lui, qui permet de signer une transaction. La fonction de signature est donc une fonction *Signe* qui prend en argument une transaction (sous forme d'une chaîne de caractère), une clef secrète (un entier) et qui renvoie une signature de la transaction (sous forme d'une chaîne de caractère). Sans clef secrète, il est impossible de générer une signature authentique.
- **Une clef publique**, accessible à tous, qui permet de vérifier qu'une signature est authentique. La fonction de vérification est donc une fonction *Verifie* qui prend en argument une transaction, une signature et une clef publique (toujours un entier) et qui renvoie *VRAI* si la signature est authentique et *FAUX* sinon.

On pourra supposer que les clefs publiques sont stockées dans un dictionnaire avec le nom de l'organisateur·trice comme clef de dictionnaire.

Question 5 (5 points)

> En supposant l'existence de *Signe* et *Verifie* dans votre langage de programmation, implémentez 2 fonctions :
>
> - *nouvelle_transaction* prend en argument un émetteur, un receveur, un montant, une clef secrète et un registre, et ajoute la transaction au registre,
> - *verifie_registre* prend en argument un dictionnaire de clefs publiques et un registre et vérifie les transactions en supprimant celles non authentiques.

2.1.1 Ajout d'un identifiant

Même si toutes les transactions sont signées et donc infalsifiables, certains problèmes persistent.

Question 6 (3 points)

> À votre avis, pourquoi doit-on ajouter un identifiant unique à chaque transaction si on veut éviter des transactions frauduleuses ?

Question 7 (2 points)

> Faut-il signer la transaction puis ajouter l'identifiant, ou bien ajouter l'identifiant puis signer le tout ?

Question 8 (5 points)

> Un identifiant unique est trop difficile à générer (parcourir tous les identifiants déjà utilisés un par un prendrait trop de temps, et les générer dans l'ordre créerait une collision si deux personnes ajoutent une transaction au même moment). On décide donc de prendre à chaque fois un entier aléatoire entre 1 et N. Sachant qu'il y a m transactions en tout, quelle est la probabilité qu'il y ait au moins une collision ? Explicitez vos calculs.

Question 9 (3 points)

> Si $N = 10^6$, combien de transaction **environ** faudra-t-il réaliser pour avoir au moins une chance sur deux d'avoir une collision ? Et pour n'importe quel N (supposé grand devant le nombre de transaction) ? [6].

Question 10 (2 points)

> Désormais une transaction se compose d'un identifiant, un émetteur, un destinataire, un montant et une signature. On suppose que les algorithmes de la question 4 ont été modifiés pour inclure l'identifiant à chaque transaction. Écrire une fonction qui vérifie qu'il n'y a pas deux transactions avec le même identifiant.

Question 11 (1 point)

6. Ce résultat s'appelle le paradoxe des anniversaires, et est à l'origine d'une large classe d'attaque contre des systèmes cryptographiques dans la vraie vie véritable !

> Donner la complexité spatiale et temporelle de l'algorithme précédent en fonction du nombre m de transactions dans le registre.

2.2 Amélioration de la recherche de doublon d'identifiant

2.2.1 Arbres Binaires de Recherche

Un Arbre Binaire de Recherche (ABR) est un arbre binaire dont les noeuds sont étiquetés par des clefs que l'ont peut toujours comparer (ici des entiers), qui permet donc de représenter un ensemble (ici l'ensemble des identifiants déjà utilisés dans le registre). De plus, dans un ABR, l'étiquette d'un noeud est toujours plus grande que toutes les étiquettes des noeuds de son sous-arbre gauche, et toujours plus petite que toutes les étiquettes des noeuds de son sous-arbre droit.

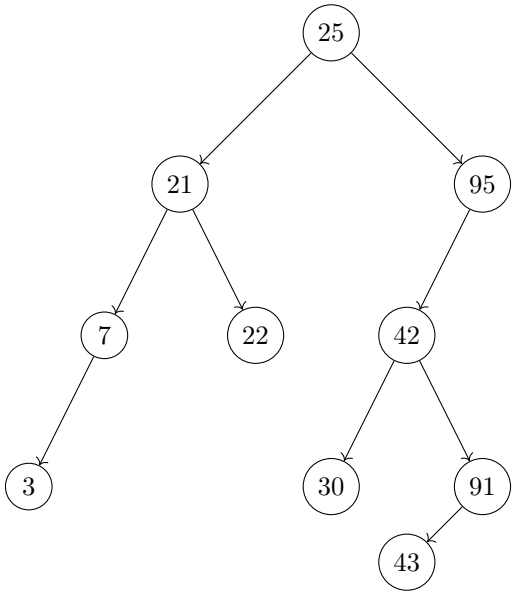


FIGURE 1 – Exemple d'un Arbre Binaire de Recherche

Question 12 (3 points)

> Proposer une structure de données dans votre langage de programmation permettant de représenter un Arbre Binaire de Recherche [7]. Lisez bien les questions suivantes, vous réutiliserez votre structure dans celles-ci.

Question 13 (2 points)

> Implémenter une fonction qui, étant donné un ABR et un élément x, renvoie VRAI si x est présent dans l'ABR, FAUX sinon.

Question 14 (3 points)

> Implémenter une fonction qui ajoute un élément dans un ABR si cet élément n'y est pas déjà, et qui renvoie une erreur dans le cas contraire.

Question 15 (1 point)

> Réécrire la fonction de la question 10 en utilisant ce qui précède.

Question 16 (5 points)

7. À partir de maintenant, ils serons appelés ABR.

> Même si ce n'est pas encore d'actualité, le bureau envisage de supprimer certaines transactions jugées abusives ou confidentielles [8]. En attendant une solution pour mettre à jour le registre, il vous demande déjà d'implémenter une fonction permettant de supprimer un élément de l'ABR.

Question 17 (2 points)

> Donner la complexité spatiale et temporelle des questions 13, 14 et 16 dans le cas d'un arbre complet (toutes les feuilles sont à la même profondeur).

Question 18 (2 points)

> Connaissez-vous une structure de données permettant d'avoir les mêmes complexités dans le pire des cas ?

2.3 La répartition

Un autre péril plane sur la répartition des crêpes. Certaines personne s'amusent à créer des transactions avec des crêpes qu'elles n'ont pas [9]. Il nous faut un moyen de vérifier que tous les échanges sont légaux pour faire la répartition.

Question 19 (2 points)

> Écrire une fonction prenant en entrée le registre et vérifiant, à tout instant, qu'aucun organisateur·trice n'est en négatif de crêpes (il est autorisé pour le bureau d'être en négatif de crêpes car ce sont eux qui les cuisineront lors du banquet, on suppose évidemment qu'aucun organisateur·trice ne s'appelle "bureau" [10]).

Question 20 (1 point)

> Modifier votre fonction précédente ou écrire une nouvelle qui renvoie une structure de données de votre choix représentant la quantité de crêpes que chacun·e doit recevoir de la présidence lors du banquet si personne n'a triché, "Erreur" sinon.

Il se trouve que la présidence a mieux à faire qu'à courir après tous les organisateur·trices pour leur donner des crêpes [11], la solution suivante a donc été trouvée par notre président : les crêpes seront données à certain·es organisateur·trices en particulier qui devrons faire la répartition eux·elles-même.

Question 21 (3 points)

> Écrire une fonction qui, étant donné la sortie de la fonction de la question 20 et une liste d'organisateur·trices associée au nombre de crêpes données par la présidence, propose une liste de transfert de crêpes entre organisateur·trices pour que tout le monde ait toutes les crêpes qui lui sont dues. Il n'est pas demandé de proposer une fonction particulièrement efficace.

La répartition des crêpes est particulièrement inefficace, on voudrait une repartion qui minimise le nombre de transferts entre les organisateur·trices.

Question 22 (8 points)

> Proposer une fonction qui, étant donné la sortie de la fonction de la question 20 et une liste d'organisateur·trice associée au nombre de crêpes données par la présidence propose une liste de transfert de crêpes entre organisateur·trices minimisant le nombre de personnes différentes un·e même organisateur·trice doit aller voir pour lui donner des crêpes. Argumenter sur son efficacité et ses limite : une fonction optimale n'est pas attendue, laissez parler votre créativité.

8. Les crêpes sont sources de bien des convoitises...
9. Nous soupçonnons que ce sont les mêmes personnes qui falsifiaient les registres précédemment, mais nous n'avons aucune preuve.
10. Il y en avait un, mais on a pas réussi à lui créer un mail...
11. Entre autres : organiser le concours Prologin, Girls Can Code !, envoyer des mails, répondre à des mails, forcer les organisateur·trices à travailler (:eyes :), faire des crêpes...

2.4 Et après le banquet ?

On souhaite désormais trouver un système pour vérifier que personne ne modifie le registre de crêpes (à son avantage) une fois la finale terminée [12]. De plus, on s'attend à ce qu'une fois la finale terminée, certaines paires de clefs privées et publiques se perdent (car elles ne sont plus vraiment utiles) et donc on ne peut plus compter sur le système de signatures numériques.
On aimerait que ce système prenne peu de mémoire (taille linéaire en nombre de transaction, indépendamment de leur contenu, notamment dû à la taille du nom des organisateur·trices [13]).
On considère dans cette partie qu'aucune transaction ne peut être ajoutée au registre. Une transaction pourrait cependant être modifiée (changement de montant, d'émetteur ou de destinataire) de façon illicite et c'est ce que l'on cherche à détecter. L'identifiant, lui, ne change pas et on suppose que les transactions sont triées par ordre croissant d'identifiants dans le registre.
On suppose [14] que le nombre de transactions est une puissance de deux : 2^n avec $n \in \mathbb{N}$. Les transactions peuvent donc être numérotées par $t_0, t_1, t_2, \ldots, t_{2^n-1}$.

2.4.1 Fonction de hachage

Une fonction de hachage est une fonction qui a une donnée arbitraire (ici une chaîne de caractères représentant une transaction) associe un entier codé sur k bits (un « *haché* ») - peu importe la taille de la donnée en entrée.
Une fonction de hachage H est dite cryptographique si, de plus, elle vérifie les trois conditions suivantes :

1. **résistance à la première préimage** : il est difficile de trouver un antécédent par la fonction de hachage.
2. **résistance à la seconde préimage** : pour un message donné, il est difficile de trouver un message différent qui génère le même haché.
3. **résistance aux collisions** : il est difficile de trouver deux messages qui donnent le même haché.

Question 23 (5 points)

> Expliquer comment utiliser une fonction de hachage pour que chaque participant rentre chez lui avec un témoin (de taille très inférieure à la taille du registre), qui pourra être utilisé à l'avenir pour vérifier que le registre n'a pas été modifié entre temps.

Question 24 (5 points)

> Dans cette question uniquement, on décide de ne considérer que les caractères alpha-numériques (plus l'espace) et de les coder sur des entiers. Le `0` est codé par 0, le `1` par 1 ... le `9` par 9, le `a` par 10, le `b` par 11, ..., le `z` par 35 et l'espace par 36. On choisit comme fonction de hachage la fonction qui à une chaîne de caractère associe la somme des codes des caractères cette dernière, modulo 2^k.
> La fonction de hachage précédente est-elle
>
> 1. résistante à la première préimage ?
> 2. résistante à la seconde préimage ?
> 3. résistante aux collisions ?

On suppose donc désormais avoir une fonction de hachage cryptographique H.

2.4.2 Arbre de Merkle

Un arbre de Merkle est un arbre binaire strict (les noeuds internes ont exactement deux fils) dont les noeuds sont étiquetés par des hachés (feuilles y compris).

- Les feuilles sont étiquetées par les hachés des donnés dont on veut garantir l'intégrité. Ici, ce sera les transactions $t_0, t_1, t_2, \ldots, t_{2^n-1}$ (dans l'ordre, de gauche à droite par exemple).
- Les noeuds internes sont étiquetés par le haché de la concaténation des hachés de ces deux fils (que l'on aura préalablement converti en chaîne de caractère).

Le nombre de transactions étant une puissance de deux, on imposera de plus l'arbre de Merkle complet (toutes les feuilles sont à la même profondeur).

12. En effet, certaines dettes de crêpes durent pendant des années.
13. Qui peuvent avoir des noms pesant plusieurs Giga quand écrits en ASCII.
14. Quitte à rajouter des transactions de montant nul.

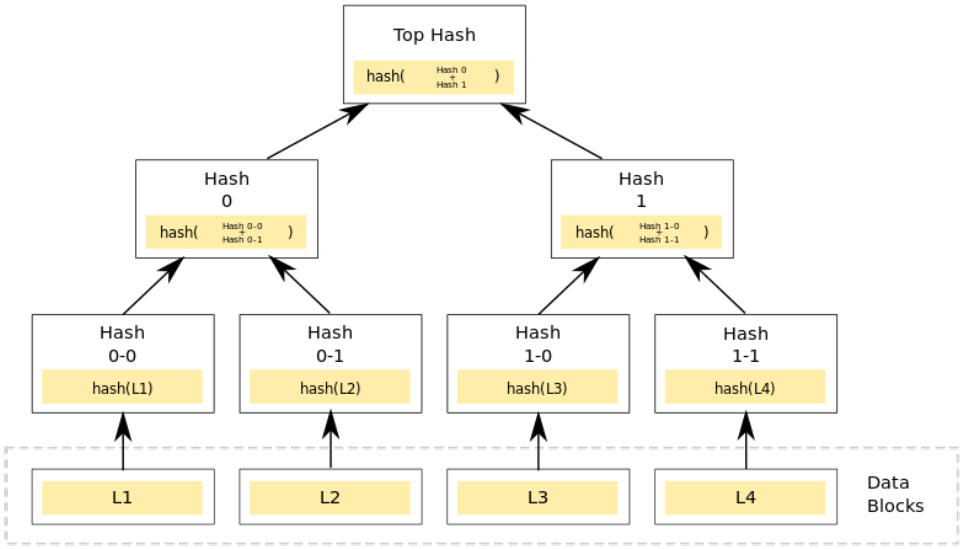


FIGURE 2 – Exemple d'un arbre de Merkle [15]

Question 25 (3 points)

> Proposer une structure de données permettant de représenter un arbre de Merkle. Lisez bien les questions suivantes car vous réutiliserez votre structure plus tard.

Question 26 (3 points)

> On suppose la fonction de hachage H mentionnée précédemment déjà implémentée dans votre langage de programmation. Implémenter une fonction qui prend en argument un registre et qui renvoie l'arbre de Merkle correspondant.

Question 27 (5 points)

> En supposant que chaque participant rentre chez lui avec l'arbre de Merkle original, et qu'un autre arbre de Merkle lui parvienne plus tard, écrire une fonction efficace qui vérifie qu'aucune transaction n'a été modifiée entre temps, et si c'est le cas, qui donne le numéro des transactions en question. Quelle est la complexité temporelle de votre fonction dans le cas où une seule transaction a été trafiquée ?

3 Bonus

Question bonus 28 (6 points)

> Maintenant on suppose que chaque participant ne garde que la racine de l'arbre de Merkle. Montrer qu'un organisateur·trice en possession de l'arbre complet peut prouver qu'une certaine transaction est licite et en voyant uniquement la transaction en question et $\log(m)$ hachés. Attention le participant doit ensuite pouvoir vérifier que cette transaction est licite en $O(m \log(m))$.

Question bonus 29 (6 points)

> Implémenter l'algorithme de vérification de la question précédente.

Question bonus 30 (8 points)

15. Illustration : `https://en.wikipedia.org/wiki/Merkle_tree`

> Proposer une fonction de hachage cryptographique, et prouver sa sécurité.

Question bonus 31 (2 points)

> Donner la meilleure recette de crêpe que vous connaissez.

Question bonus 32 (3 points)

> Avec la recette précédente, donner une estimation de la quantité de farine qu'il faudra pour nourrir l'ensemble des organisateur·trices et participant·es lors de la finale Prologin de cette année, sachant que jetSett sera présent.

Le sujet comporte 8 pages (sans compter la page de garde), 32 questions, et 6 questions bonus. Les questions normales sont notées sur 80 points, plus 1 point de présentation.