Rapport du Projet long - LU2IN006

runlin ZHOU 28717281 Group 6

haoran ZHANG 28717301 Group 6

Introduction :

L'objectif de ce projet est donc mettre en place une méthode de vote fiable, en utilisent un certain nombre de protocoles et de structures de données. Les protocoles sont pour garantir la validité de chaque vote et la sécurité des informations du electeur. Et les structures de données est désignées pour le stockage, la récupération des données et l'identification des électeurs.

Comme ce projet est divisé en 5 parties, je présenterai ensuite la fonction de fichier dans le dossier, puis j'analyserai le processus de notre mise en œuvre de la fonction ci-dessus en 5 parties.

Présentation du répertoire :

Le répertoire contient :

le Makefile et un grand nombre de fichiers .txt, chacun d'entre eux ayant sa propre utilité.

- *les fichiers .h et .c avec le code, qui sont contenant des descriptions de fonctions*
- *le Makefile, qui peut compiler un ensemble de fichers .h et .c*
- trois exécutables : mainPartie5, main et test,
  1. test : une série de tests sur les fonctions défini
  2. main : la réalisation du but de projet pour Partie 1-4
  3. mainPartie5 : la réalisation du but de Partie 5
- *un grand nombre de fichiers .txt, chacun d'entre eux ayant sa propre utilité*

—— *Par exemple :*

U*n fichier commande.txt qui contient une séquence d'instructions de tests (pour produire le courbe de vitesse), sortie_vitesse.txt qui stocker toutes les données qui sont obtenu sur l'exécutable main pour créer le schéma avec les commandes dans le ficher commande.txt. (Nous préciserons les fonctions des fichiers .txt plus tard.)*

Attention :

En raison de la limite de mots par ligne du fichier Txt, il a la possibilité de certaines des données de decalration.txt occupent deux lignes, ce qui peut provoquer des erreurs lors de la

lecture du fichier. Cela signifie que les données stockées dans un pointeur peuvent être perdues, ce qui peut conduire à un « segmentation fault ».

Si vous rencontrez ce problème, veuillez exécuter le fichier main ou test dans une autre fois.

Présentation par chaque partie :

Partie 1:

Introduction :

Cette partie est divisée en deux sections principales.

Dans l'exercice 1, nous complétons la définition des fonctions qui prennent le module de puissance (a^m mod n) et la génération aléatoire de nombres premiers.

Dans l'exercice 2, nous avons mis en œuvre la méthode de chiffrement RSA. Il est facile de noter que les fonctions de l'exercice 1 sont basiques pour réaliser le but dans l'exercice 2 et seront largement utilisées dans l'exercice 2.

Introduction des fonctions impoartants :

modpow() :

Cette fonction utiliser la suite suivante :

$$u_n(x) = \begin{cases} 1 & \text{si } n = 0 \\ (u_{\frac{n}{2}}(x))^2 & \text{si n est pair} \\ (u_{\frac{n-1}{2}}(x))^2 * x & \text{si n est impair} \end{cases}$$

Ce qui peut calculer la puissance de x avec la complexité en $o(log_2 n)$.

generate_key_value() :

La valeur de clé public et privée est passé par les pointeurs. Selon la définition de la méthode
*de RSA, on définit une fonction « methode_bezout » qui utilise **l'algorithme bezout.***

Le but est de calculer les coefficients de Bézout : $\boldsymbol{u_1} * a + \boldsymbol{u_2} * b = \boldsymbol{u_3} = pgcd(a, b)$

> **Entrées :** a et b

> **Sortie:**
> $u_1 * a + u_2 * b = u_3 = pgcd(a, b)$

> **Initialisation :**
> $(u_1, u_2, u_3) \leftarrow (1, 0, a)$
> $(v_1, v_2, v_3) \leftarrow (0, 1, b)$

> **Itération :**
> Tant que : $v_3 \neq 0$
> $q = \frac{u_3}{v_3}$
> $(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - q * (v_1, v_2, v_3)$
> $(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3)$
> $(v_1, v_2, v_3)) \leftarrow (t_1, t_2, t_3)$

Enfin, le nombre « s » est généré aléatoirement jusqu'à ce que pgcd (s, t) satisfasse la condition et finalement deux paires de nombres (s, n), (u, n) sont produites.

(ps. La valeur de u est passée par un pointeur dans la fonction « methode_bezout ».)

Fonction encrypt() et decrypt() :

Ces deux programmes sont utilisés respectivement pour crypter et décrypter des messages.
La fonction encrypt() convertit la chaîne de caractères en code ascii correspondant, puis la crypte avec la fonction modpow(), en utilisant la clé publique calculée précédemment comme argument.

La fonction decrypt() est l'opposé de en, mais avec la même structure. Les deux retournent un tableau dont la taille est égale à la taille du message transmis.

Reponse des questions :

Exe 1.1 :

La variable i augmente 1 à chaque tour de boucle, donc la condition du while n'est plus vérifier d'après $((p-1)-3)$ fois de boucle. Pour toutes les valeurs de p, la boucle est exécutable $p-4$ fois. Donc, la complexité de la fonction « is_prime_naive » est en $o(x)$.

Exe 1.3 :

La variable i augmente 1 à chaque tour de boucle, donc la condition du while est vérifier d'après n fois de boucle. Pour toutes les valeurs de n, la boucle est exécutable n fois. Donc, la complexité de la fonction « modpow_naive » est en $o(x)$.

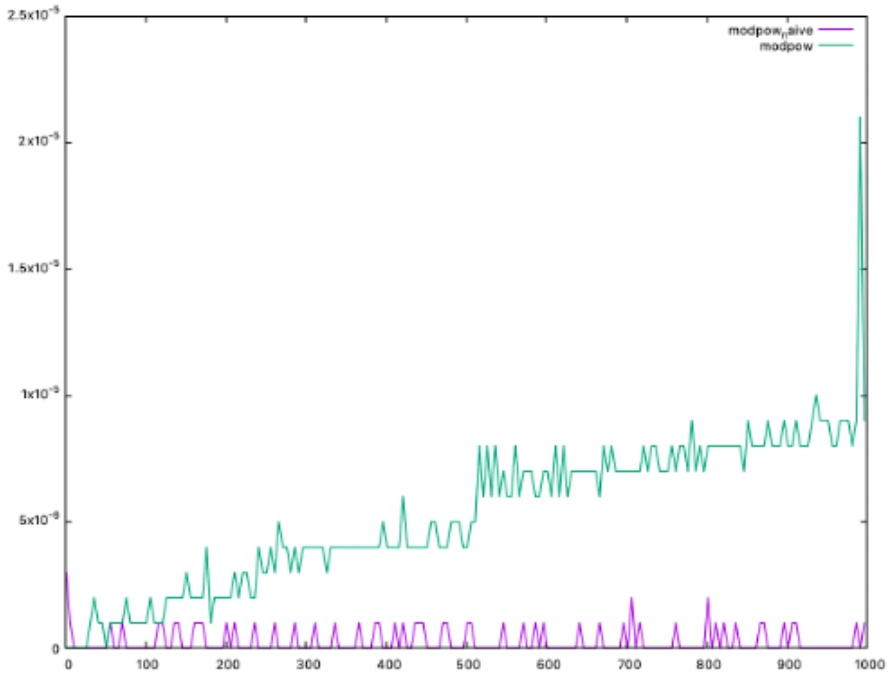
Exe 1.5 :

*En utilisent la fonction **clock()**, on peut générer la vitesse de calcul des deux fonctions - is_prime_naive() et is_prime_miller().*

Dans le exécutable main, on peut voir que les données sont stockées dans le ficher « sortie_vitesse.txt ». Ce fichier est composé de 3 colonnes représentant le nombre que nous

voulons déterminer s'il s'agit d'un nombre premier et la vitesse.

Sur terminal, la commande ***gnuplot -p < commande.txt*** *est nécessaire pour produire l'image de courbe qui est nommé par « courbes_vitesse.ps ».*



la ligne verte est is_prime_miller() et la ligne violette est is_prime_naive()

Il s'ensuit que is_prime_miller est plus fort que is_prime_naive en termes de vitesse de calcul et que la trajectoire de son image est similaire à la fonction $log_2 n$.

Exe 1.7 :

Selon des informations connues dans le titre :

> *Pour tout entier p non premier quelconque, au moins 3/4 des valeurs entre* 2 *et* $p-1$ *sont des témoins de Miller pour* p

Donc, la probabilité de que on ne peut pas trouver le témoin est $\frac{1}{4}$.

Dans cette fonction, un autre paramètre k joue également un rôle important. La valeur de k représente le nombre de fois de sélections aléatoires d'un nombre entier dans l'intervalle $[2, p-1]$, *chaque sélection aléatoire étant indépendante des autres et satisfaisant ainsi* ***la loi de Bernoulli.***

Donc, la probabilité d'erreur de cet algorithme est

$$p(k) = (\frac{1}{4})^n$$

Par conséquent, la borne supérieure est $\frac{1}{4}$, *quand* $k = 1$, *car p(k) est une fonction décroissance pour* $\forall p \in [0, \infty)$. *Cette conclusion peut également être tirée de la sortie dans*

l'exécutable main, où le pourcentage de déterminations correctes que t est premier augmente lorsque k augmente.

Interprétation des tests de performances :

Cette section décrit et analyse les résultats des tests de performance, contenus dans test.c.

Dans cette section, certaines des fonctions les plus simples utilisent directement assert() pour vérifier l'exactitude, par exemple : is_prime_naive().

Certains programmes plus complexes sont soumis à de multiples expériences par des boucles for afin de garantir leur exactitude. En revanche, dans le fichier main.c, ces fonctions sont vérifiées et raisonnées en utilisant différentes combinaisons d'arguments.

> 💡 ***Par exemple** is_prime_miller(): et ensuite, après une série de tests, on peut constater que la correction de l'algorithme du programme augmente de façon spectaculaire avec des valeurs croissantes de k.*

En même temps, afin de gagner du temps dans mian.c au moment de l'exécution, le traitement pré-expérimental a été effectué dans test.c. De cette façon, les données peuvent être présentées de manière plus intuitive dans mian.c, et l'efficacité du travail est également améliorée.

Enfin, les données de test ne sont pas très précises et pourraient être améliorées en modifiant le type de données, mais cela nécessiterait de corriger tous les types de données du fichier ensemble, ce qui n'en vaut pas la peine.

Partie 2 :

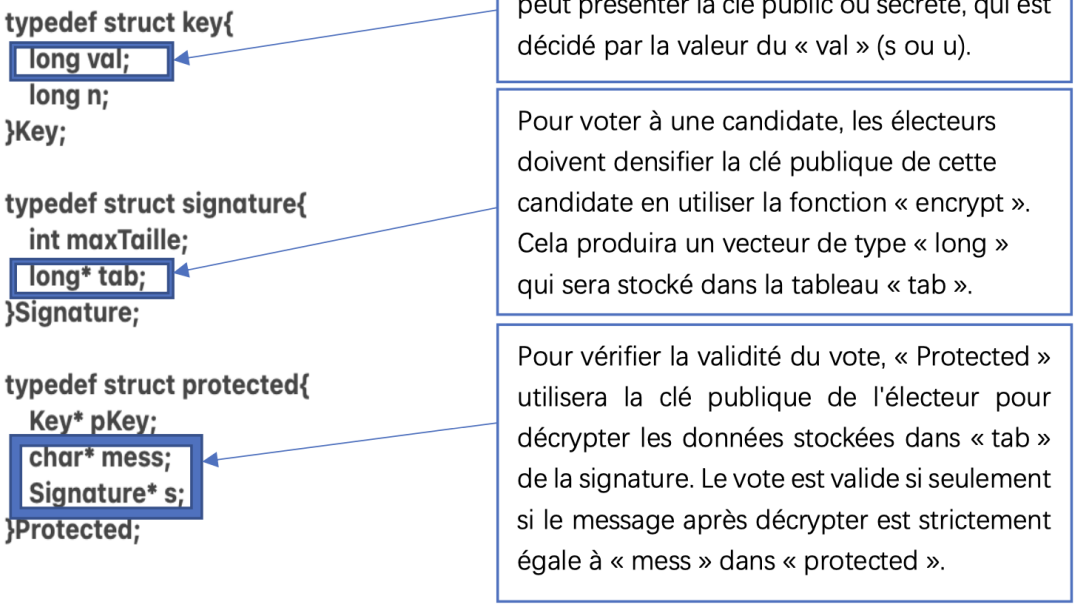
Introduction :

Cette partie sur la construction d'un système de vote simple, qui est associé à trois structures : Key, Signature, Protected.

- *Pour chaque électeur, ils ont chacun deux clés, une publique et une privée.*
- *S'ils veulent voter pour un candidat, ils émettent une « Signature », qui est obtenu en chiffrant la clé publique du candidat par la clé privée de l'électeur.*
- *Dans le même temps, nous devons créer un mécanisme de « Protected » pour garantir l'identité des électeurs () et des candidates (la clé puplique de cadidate).*

Introduction des fonctions importants :

trois structs generate :



Generate_random_data() :

Dans cette fonction, on va créer trois fichiers txt : keys.txt, candidates.txt, declarations.txt.

Les formats pour chaque fichier :

- *keys.txt -> (%ld,%ld) (%ld,%ld) qui sont les clé publique et secrète des électeurs*
- *candidates.txt -> (%ld,%ld) qui est la clé publique de candidate*
- *declaration.txt -> #%s#%s#..... qui est un vecteur encrypté par la clé secrète des électeurs et séparés par **« # »***

La première étape consiste à générer un tableau de int que nous avons choisis au hasard comme candidats, en utiliser la fonction « random_int_list ». Nous allons ensuite générer une paire de clé (avec la fonction « init_pair_keys ») et les stocker dans le fichier keys.txt avec le format qu'on définit précèdent, en répétant nv fois.

Puisque nous utilisons la variable « i » comme marqueur pour terminer la boucle, lorsque i se trouve dans le tableau (on le vérifier avec la fonction « is_in_liste ») que nous avons créé précédemment, nous stockons la clé publique de la personne dans le fichier « candidates.txt ».

Enfin, on va lire les données sur les deux fichiers : « keys.txt » et « candidates.txt » et les combinerons à « Protected » pour vérifier la validité du vote. Les votes éligibles seront ensuite stockés dans le fichier « declaration.txt ».

Partie 3 :

Introduction :

Cette partie concerne la création des listes chainées de « Key » et « Protected » qu'on a déjà définie précédent. Pour ce faire, nous avons besoin de deux nouvelles structures de données ***：CellKey*** *et* ***CellProtected*** *et d'un nouvel ensemble de fonctions pour effectuer des opérations entre les listes chaînées.*

On lire les données sur les fichiers qu'on crée dans la ***Partie 2****, et après, on constitue les listes sur les données qu'on obtenir.*

Introduction des fonctions importants :

Les fonctions cellKeys et cellProtected :

Ces fonctions contiennent tous les opérations possibles pour les listes chainées (allouer, desallouer, ajouter, supprimer un élément, supprimer une liste, vérifier égalité, affichage).

💡 *Les fonctions pour CellKey se terminent par _cell_key ou _list_key, et pour CellProtected, ils se termine par _cell_proetected ou _list_protected*

💡 Les opération pour les listes chainées :

*allouer : Commence par create_*

*desallouer : Commence par delete_cell_*

*ajouter : Commence par ajouter_*

*supprimer : Commence par delete_list_*

*vérifier égalité : Commence par is_equal_*

*affichage : Commence par print_*

Partie 4 :

Introduction :

Dans cette section, nous nous concentrerons sur les problèmes liés aux tables de hachage. Nous avons lu toutes les données requises dans les fichiers(keys.txt, candidates.txt, et declarations.txt) et nous allons ensuite calculer qui a reçu le plus de votes.

Pour réduire la complexité du programme, nous abandonnerons l'utilisation de liste

chaînée, car il faudrait beaucoup de temps pour traverser la liste entière. Au lieu de cela, une table de hachage prendra beaucoup moins de temps car nous pouvons aller directement à la position que nous recherchons sur l'indice.

Dans cette partie de l'exercice, nous allons utiliser la fonction : modpow() qui est définie dans la partie 1 comme fonction de hachage.

Introduction des fonctions importants :

hash_function() :

Une fonction facile mais important, en utillisent de la fonction modpow(), qui calcule la vleur de $f(x) = a^n mod m$ *avec pkey = (a, n), size = m.*

create_hashtable() :

Crée et initialise une table de hachage de taille size contenant une cellule pour chaque clé de la liste chaînée keys.

Pour inserer les données à la bonne place, on la fonction de hachage est $pos_{new} = (f(x) + 1) mod (size)$

Tous les vals sont initialisé à 0 (on va disscuter leurs fonctions plus tard).

compute_winner() :

On fait l'hypothèse que les signatures déclarations sont toutes valides.(déjà vérifier dans Partie 2)

On definie un protocole telle que

- *si un voteur vote pour un candidate, la valeur de cette cell dans le hash table doit changer à 1*
- *si un candidate a reçu un vote, la valeur de leur cells doit se augementer a 1*

Après avoir confirmé que toutes les données sont fiables, nous commencerons à les stocker. En utilisant de fonctions : find_position(), nous pouvons le faire directement. Une fois la boucle terminée, la fonction htKiArgmax() sera utilisée pour savoir qui a le plus de votes.

Interprétation des tests de performances :

Soit le nombre de candidate et de électeur sont nv et nc.

create_hashtable() :

Au tout début, nous devons initialiser la table de hachage avec toutes les valeurs NULL.

Ensuite, nous passerons la fonction de hachage(modpow()) en plaçant tour à tour les valeurs du liste chainée(CellKey) selon la clé publique dans les positions correspondantes. Dans cette boucle, si la valeur stockée dans cette position n'est pas vide, alors la valeur de l'indice sera ajoutée par un jusqu'à ce qu'une position soit trouvée où sa valeur est vide. Puis nous appellerons la fonction create_hashcell() pour le stocker.*

Par conséquent, la complexité de cette fonction est en $o(n)$.

compute_winner() :

La fonction commence par créer deux tables de hachage :

- *votersHashTable : qui stock tous les données de voteur, avec la taille égale à nv*
- *candidatesHashTable : qui stock tous les données de candidate, avec la taille égale à nc*

donc le fois qu'on entre à la mémoire est $nc + nv$

Après la, on commence de calculer le nombre de vote. À cause de la caractère de tablaux de hachage, la complexité de cette boucle est en $o(n)$. Si on utilise une liste de chaînée, la complexité est en $o(n^2)$.

Partie 5 :

Introduction :

Dans cette section, nous allons utiliser la blockchain pour compléter la dernière partie du système de vote. Cela implique trois étapes principales : le vote des électeurs, la création de nouveaux blocs et la mise à jour des informations pour tous les électeurs.

- ***Vote :** Nous avons déjà réalisé cette partie, c'est-à-dire qu'une électeur envoie une déclaration (à tous) qui doit être chiffrée par sa clé privée et réversible par sa clé publique.*
- ***Création des blocs :** Dans un certain intervalle de temps (le temps qu'il faut pour calculer le prochain hachage de bloc), le système recueillera tous les votes exprimés. Après avoir confirmé qu'il n'y a pas de votes multiples ou de votes invalides, toutes les informations seront créées dans un nouveau bloc.*
- ***Mise à jour :** Après la publication d'un nouveau bloc, chacun doit ajouter le nouveau bloc à sa base de données. Cependant, si quelqu'un essaie de tricher (falsifier un vote), il y a plusieurs possibilités différentes et une nouvelle structure de données est créée - CellTree et un nouveau protocole est introduit pour s'assurer que tout le monde dispose de la même base de données. Je l'expliquerai plus tard.*

Une fois le vote terminé (aucun nouveau bloc n'est créé), nous pouvons compter le nombre de votes pour chaque candidat sur la blockchain et annoncer la clé publique du gagnant.

Par conséquent, il n'y a pas de centre pour collecter des votes comparé au vote traditionnel. L'information est donc plus ouverte et transparente, et la fiabilité des votes est garantie, ce qui élimine pratiquement toute possibilité de tricherie (sauf si l'algorithme SHA256 peut être déduit en sens inverse).

Introduction des fonctions importants :

compute_proof_of_work() :

Pour un block, on a :

```
typedef struct block{
    Key* author;
    CellProtected* votes;
    unsigned char* hash;
    unsigned char* previous_hash;
    int nonce;
}Block;
```

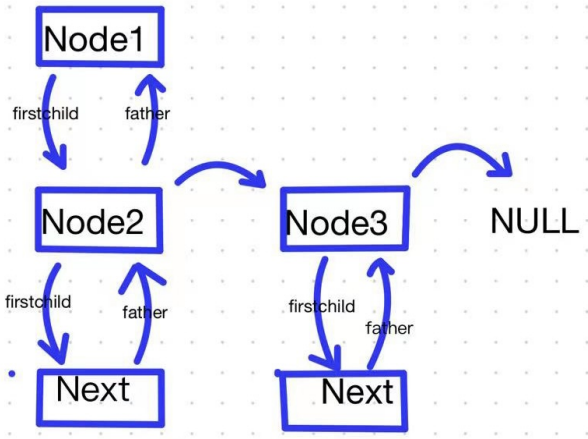
*En utilisent la fonction block_to_str(), nous prenons l'auteur d'un bloc, l*a valeur hachée du bloc précédent*, les informations de vote stockées dans le bloc et val et formons une nouvelle chaîne str, qui sera ensuite chiffrée par SHA256 en str_new.*

Le but de cette fonction est de trouver exactement un nonce qui peut chiffrer la valeur hachée du bloc (hash) commence par d zéros successifs avec nonce se augmente à 1.

En raison de la fiabilité du SHA256, il est très difficile de le décrypter en inversant le cryptage, de sorte que toute personne souhaitant publier un nouveau bloc devra effectuer de nombreux calculs.

last_node(CellTree* t) :

```
typedef struct block_tree_cell{
    Block* block;
    struct block_tree_cell* father;
    struct block_tree_cell* firstChild;
    struct block_tree_cell* nextBro;
    int height;
}CellTree;
```



En raison de la fraude, plusieurs nouveaux blocs (dont un seul est vrai) peuvent être créés pour un bloc identifié donné, c'est-à-dire qu'il existe plusieurs blocs qui ont le même "father".

Pour savoir quel bloc peut être fiable, nous observerons le principe selon lequel seule la chaîne la plus longue peut être fiable dans cette structure "CellTree".

Dans la section précédente, nous savions déjà qu'il fallait beaucoup de temps pour calculer la valeur de hachage nécessaire pour le bloc suivant. Si l'qqn voulait frauder plusieurs fois, il faudrait annoncer continuellement de nouveaux blocs avant tout le monde, ce qui nécessiterait la puissance de calcul de ordinateur très gros.

Nous utiliserons cette fonction pour stocker tous les nœuds nouvellement créés dans la plus longue chaîne de cette arborescence (le parcours qui a le plus proof_of_work).

create_block() :

La création de nouveaux nœuds repose sur le principe de la chaîne la plus longue, où chaque nouveau nœud est toujours créé dans la chaîne la plus longue.

C'est-à-dire que nous toujours prenons le enfant qui a la plus longue chaîne (se termine quand node → nextBro == NULL)d'un nœud comme "firstchild" de ce nœud.

Réponse des questions :

- Question 8.8

Pour réaliser la methode en $o(1)$, on doit ajouter un pointeur qui vers la plus loin bro de un CellTree.

- Question 8.8

Le principe technique le plus couramment utilisé pour les preuves de charge de travail est la fonction de hachage. Puisque toute valeur n de la fonction de hachage d'entrée h correspondra à un résultat h(n), et qu'un changement d'un seul bit dans n provoquera un effet d'avalanche, il est presque impossible de revenir de h(n) à n. Par conséquent, en spécifiant les caractéristiques de la recherche de h(n) et en permettant à l'utilisateur d'effectuer un grand nombre d'opérations exhaustives, une preuve de charge de travail peut être atteinte.

Comme SHA256 génère une chaîne en binaire, lorsque nous utilisons l'hexadécimal, un zéro dans cette chaîne équivaut à quatre en binaire. Par conséquent, il y a 2^4 possibilités, donc si nous voulons trouver la valeur d'un nonce qui peut faire en sorte que le hash commence avec d zéros, alors en moyenne nous avons besoin d'un total de 2^{4n}.

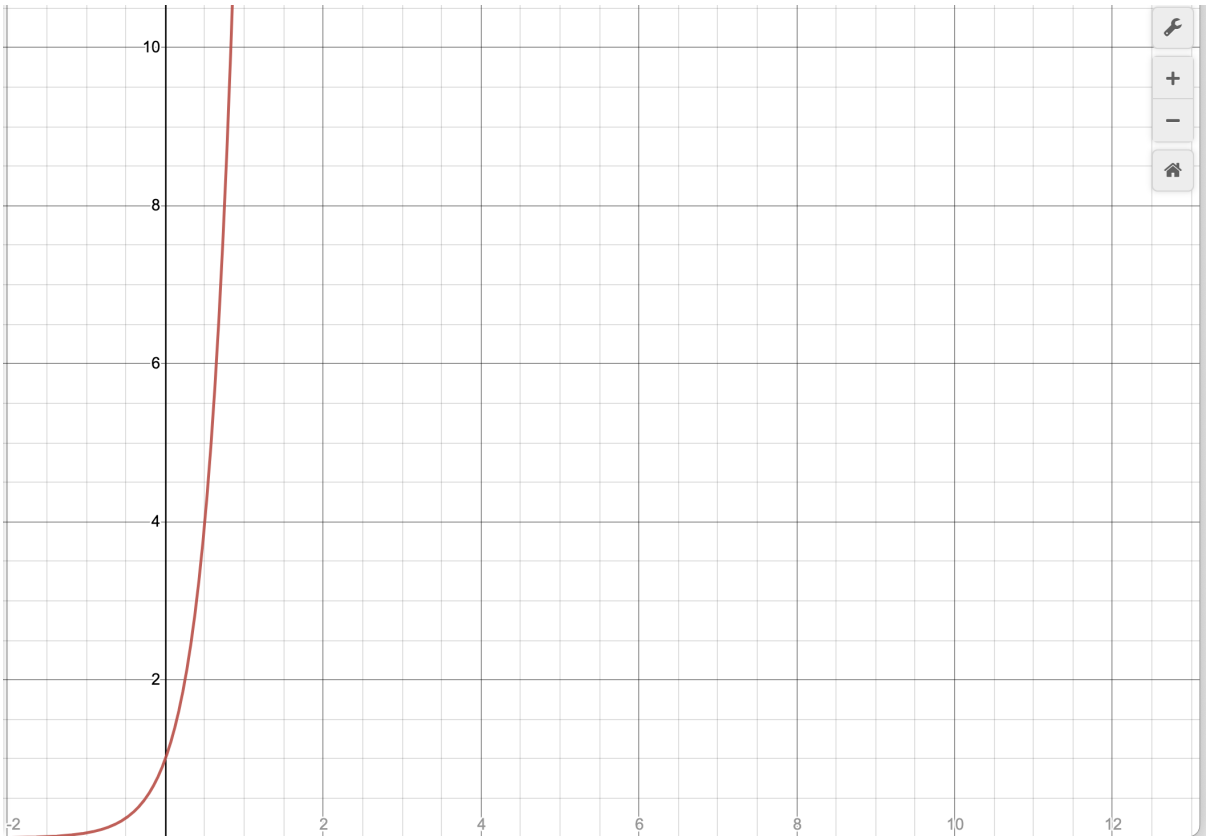
- donc $f(n) = 2^{4n}$

Si nous spécifions les quatre premières valeurs de la valeur hexadécimale de h(n) pour n, il faudra statistiquement environ 2^{16} exécutions du hachage h(n) en moyenne avant d'obtenir la réponse, mais la vérification ne devra être effectuée ainsi seulement une fois.

Si vous voulez rendre la tâche plus difficile, il suffit d'augmenter le nombre de bits spécifiés.

On fait les exprériences avec le ficher testProofwork.c. Ce ficher créer un document de txt qui nommé par sortieTime.txt, stocke les données de certains séries de n différent. La format de ce ficher est "%d %f", avec %d est la valeur de "d" et %f est le moyenne de temps pour la exécution de compute_proof_of_work(n).

En fin, on obenu le courbe suivant :



Conclusion du Projet :

Il est logique d'introduire la blockchain dans le système de vote. Par rapport aux méthodes de vote traditionnelles, la blockchain empêche la destruction des centres d'information ou l'exposition au piratage (car elle est décentralisée). Tant qu'il y a encore une personne qui conserve la base de données complète, la blockchain entière n'est pas corrompue. Dans ce système, chacun se méfie mutuellement et, en même temps, un certain degré de confiance et de sécurité est garanti.

Cette approche présente toutefois des inconvénients. Des informations provenant de

l'internet montrent (pour le bitcoin) qu'une grande quantité d'électricité est consommée chaque année en raison de “mining” pour la preuve de travail. Avec seulement 5 millions d'utilisateurs communs du bitcoin, sa consommation d'énergie est déjà bien plus élevée que le modèle de transaction bancaire traditionnel. Si tout le monde s'y mettait, cela pourrait être un test d'énergie.

De plus, le fait de faire confiance au consensus de la chaîne la plus longue ne fait pas disparaître complètement la fraude. Si lorsque trop peu de personnes sont impliquées dans le projet, c'est-à-dire que la puissance de calcul des ordinateurs des autres ne peut être comparée à celle des ordinateurs frauduleux. Dans ce cas, la base données de chacun est en danger.