LIN Pierre 28705907
ROALDES Enzo 28633274

RAPPORT PROJET LU2IN006

I. Sujet

Le but de ce projet est la mise en place d'une simulation d'un processus électoral d'abord centralisé, puis décentralisé avec un système de blockchain.
La blockchain est un système qui permet de garder la trace d'un ensemble d'opérations (ici les votes), de manière décentralisée et surtout sécurisée, sous forme d'une chaîne de blocs.
De par sa sécurité, la blockchain permet à chaque citoyen de pouvoir vérifier le bon déroulement de l'élection.

II. Description des fichiers & jeux d'essais :

La description de nos fonctions ainsi que des remarques sur leur fonctionnement se trouvent en commentaire dans le code (au-dessus de la fonction en question).

Explication générale de nos fichiers .c et .h avec les structures utilisées :

- prime.h, prime.c : contient les fonctions de l'exercice 1 pour gérer la génération de nombres premiers.

- crypto.h, crypto.c : fonctions de l'exercice 2 sur le chiffrement/déchiffrement de clés publiques et privées.

- vote.h, vote.c : fonctions de l'exercice 3 pour mettre en place le vote pour les citoyens.

    - structures utilisées :

- Key pour modéliser une clé publique (s,n) ou privée (p,n)
- Signature pour la signature, une signature est un tableau d'entiers générés par l'émetteur grâce à sa clé secrète lorsqu'il vote.
- Protected correspond à une déclaration (un vote), la structure est composée de la clé publique du voteur, un message (qui est la clé publique en hexadécimal d'un candidat), et la signature du voteur. On peut vérifier la validité d'une déclaration grâce à ces champs.

- syst_central.h, syst_central.c : fonctions des exercices 4, 5 et 6 pour mettre en place une simulation de vote.

    - structures utilisées :

- CellKey est une liste chaînée de Key, cela permettra de stocker la liste des clés des citoyens et candidats.
- CellProtected est une liste chaînée de Protected, cela permettra de stocker la liste des déclarations de vote des citoyens.
- HashCell est une case de la table de hachage HashTable (définie ci-dessous). Une case contiendra une clé et un nombre val, cela permet deux choses :
  1 - Pour chaque candidat, on pourra stocker le nombre de votes en sa faveur dans val et l'identifier avec la clé.
  2 - Pour les citoyens, on pourra vérifier si le citoyen a déjà voté selon la valeur de val (0 ou 1 ici), et vérifier que le citoyen est bien sur la liste électorale.
- HashTable est une structure de table de hachage, où size correspond au nombre d'éléments dans la table de hachage tab. Cette table sert donc à stocker les citoyens et candidats.

- block.h, block.c : fonctions des exercices 7, 8 et 9 pour simuler une élection via la blockchain

    - structures utilisées :

- Block contient les votes de N citoyens (ici nous avons pris N = 10), un bloc contient un auteur (c'est la clé de la personne qui vérifie les votes contenus dans le bloc).
  Chaque bloc correspond à un nœud d'un arbre contenant tous les votes, d'où les champs hash et previous_hash qui servent à identifier le bloc et celui qui le précède.
  De plus, le champ nonce sert à vérifier la validité des blocs grâce au mécanisme de proof of work.
- CellTree est un arbre où chaque nœud peut avoir autant de fils que nécessaire, l'arbre contient tous les votes des citoyens stockés dans les blocs. Depuis chaque nœud de l'arbre nous avons accès à son père, à son premier fils et à son prochain frère. L'arbre servira à déterminer le gagnant d'une élection en faisant confiance à la chaîne de l'arbre la plus longue.

- main.c : main pour comparer les temps d'exécution entre modpow_naive() et modpow() pour différentes puissances. Et pour comparer is_prime_naive() et is_prime_miller().

- test.c : 2 main, un pour tester les fonctions de crypto.c (exercice 2) et l'autre pour tester les fonctions de vote.c (exercice 3)

- simulation.c : 2 main, un pour tester les premières fonctions de syst_central.c (exercice 5) en créant une simulation avec des citoyens et candidats. Et l'autre pour tester les fonctions restantes de syst_central.c (exercice 6) en essayant de déterminer le vainqueur d'une élection simulée.

- utilities.h, utilities.c : Nous avons seulement une fonction utilitaire verif() qui prend en paramètre une condition cond (int) et un indice i pour afficher le i-ème message d'erreur d'un tableau prédéfini en cas de condition non vérifiée (cond = 0).

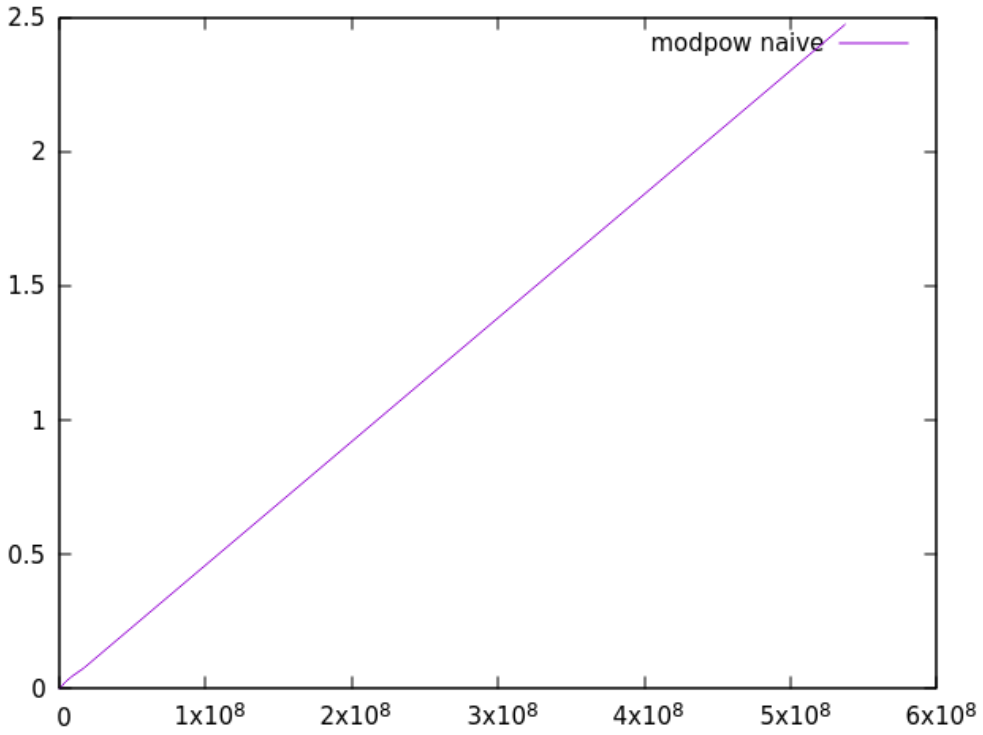
III. Réponses aux questions :

1.1) La complexité de notre fonction est en O(p) car nous faisons une boucle qui va de 3 à p dans le pire cas (où p est un nombre premier).

1.2) Le plus grand nombre premier que nous testons en moins de 2 millièmes de seconde est : 301000619

1.3) La complexité de notre fonction est en O(m) car nous faisons une boucle avec m tour.

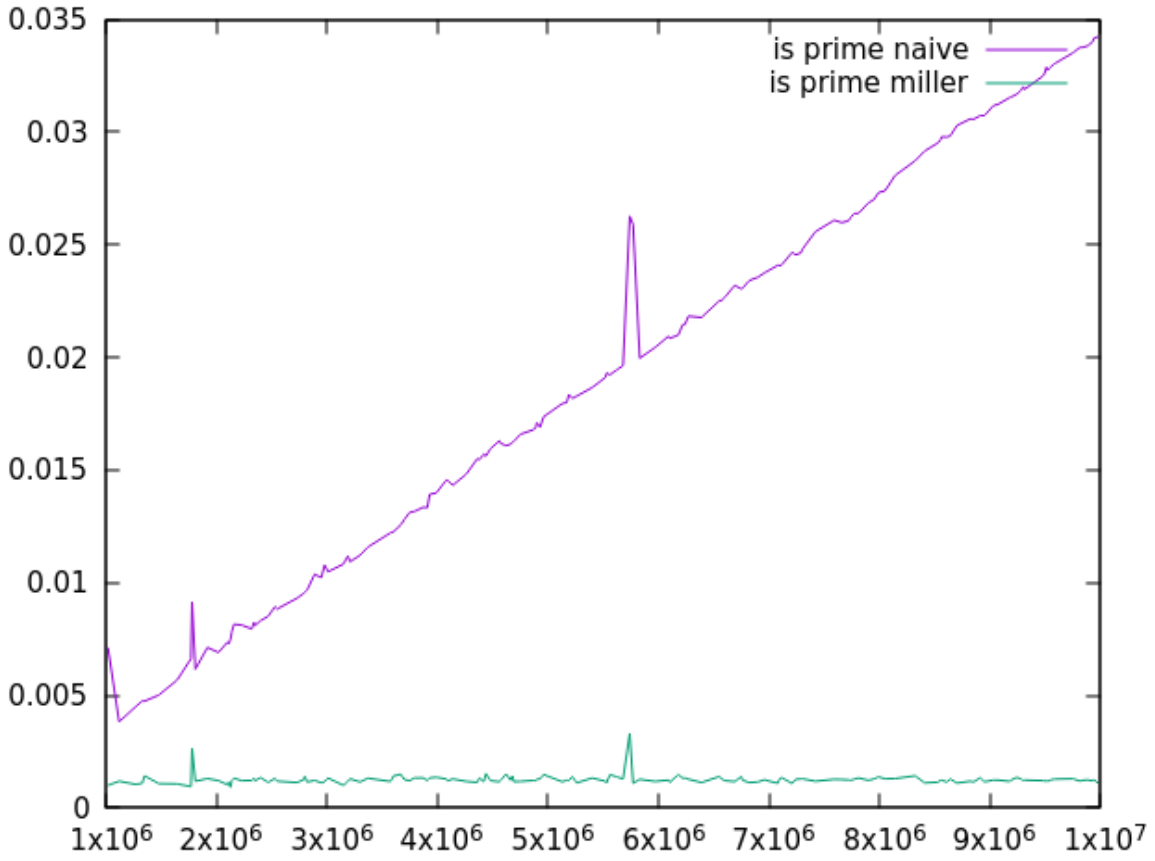
1.5) Les deux fonctions retournent le même résultat à chaque fois comme nous le voyons dans main.c.
Nous avons un temps plus faible pour modpow() car modpow() est en O(log2(m)) alors que modpow_naive() est en O(m). Nous pouvons voir les temps de la fonction modpow_naive() en ordonnées pour une puissance m en abscisse sur ce graphique :



On observe que la complexité de modpow_naive() est bien linéaire.
Pour modpow() nous avons décidé de ne pas faire de graphique car pour une puissance : p1 = 1152921504606846976 ≈ 10^18 le temps d'exécution était toujours de 0.000001 seconde donc on ne verrait rien. Ce temps est normal car log(1152921504606846976) ≈ 44. Mais cela montre bien que modpow() est bien plus rapide que modpow_naive().

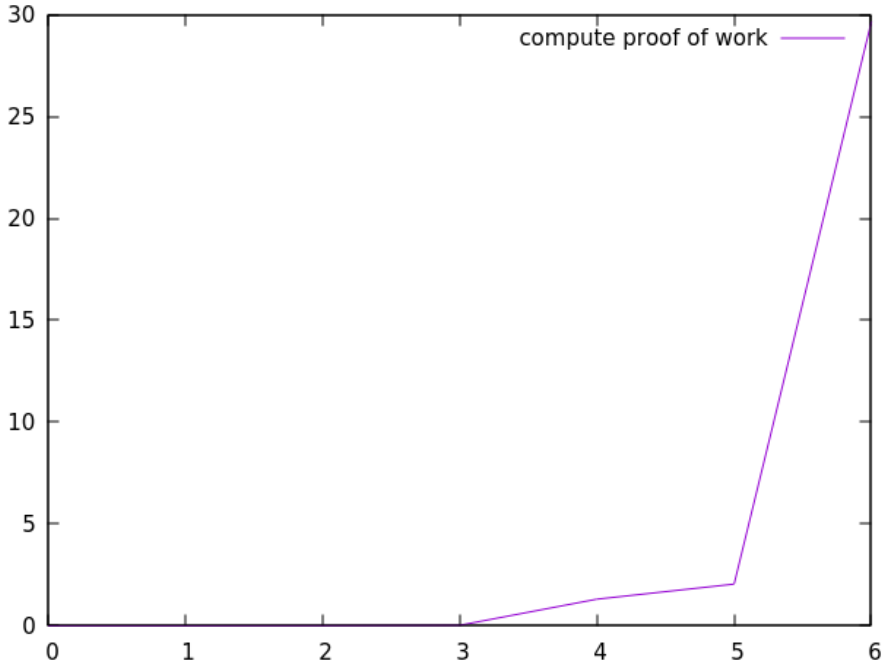
1.7) La borne supérieure est (1/4)^k, car pour chaque valeur, on a 1/4 chance qu'elle ne soit pas un témoin de Miller, et donc à la fin on a (1/4)^k chance de n'avoir aucun témoin de Miller pour un entier p non premier. La fonction a donc (1/4)^k chance de dire qu'un entier p non premier est premier.
Cette probabilité devient très faible à mesure que k augmente. On préférera donc utiliser cette fonction plutôt que is_prime_naive() qui est toujours vraie mais très lente.

Comparaison entre is_prime_naive() et is_prime_miller() :



En ordonnée nous avons le temps que prennent les fonctions pour tester si un nombre p en abscisse est premier ou non. Ici nous avons décidé de ne garder que les temps pour les nombres qui sont premiers car sinon, en général, les deux fonctions trouvent instantanément qu'un nombre n'est pas premier. Nous voyons bien que le temps d'exécution de is_prime_miller() est constant (la fonction est en O(k), nous avons pris k=5000 et avec ce k la probabilité que la fonction se trompe est très faible) et que is_prime_naive() est en O(p) car sa courbe est linéaire.

7.8) En abscisse nous avons la valeur de d, et en ordonnée le temps moyen (pour 20 essais avec chaque d) que prend la fonction compute_proof_of_work() en secondes. On voit que le temps moyen commence à exploser à partir de d = 6, nous avons aussi testé pour d = 7 et ça prenait plus de 29 minutes (nous ne l'avons pas inclus sur le graphique pour des raisons d'échelle).



8.8) Notre fonction est en O(len(P2)) avec P2 la liste de déclarations n°2.
Pour avoir du O(1) on pourrait utiliser des listes doublement chaînées.
Cela permettrait de nous rendre à la fin de la première liste et au début de la seconde en O(1) afin de les fusionner en O(1) en ajoutant le début de la seconde liste à la fin de la première.

9.7) L'utilisation de la blockchain est une très bonne idée puisqu'elle permet de s'assurer qu'il n'y a pas de fraude lors de l'élection.
En effet, frauder est inutile dans ce système puisqu'il est beaucoup plus compliqué de frauder que de perpétuer la chaîne.
Pour expliquer cela, nous pouvons considérer que quelqu'un réussit à créer une fraude avec la même vitesse qu'un bloc valide.
Toutefois, comme nous devons faire confiance à la chaîne la plus longue, le fraudeur doit continuer sa création de blocs frauduleux à la même vitesse que la branche "valide".
Hors la branche valide est créée en association par la majeure partie de la communauté (ceux qui ne fraudent pas) alors que la branche frauduleuse ne peut être créée que par un certain groupe d'individus (les fraudeurs).
De ce fait, la vitesse de création de la branche valide est bien plus rapide et il est donc très difficile pour la branche frauduleuse d'être plus longue que la valide.
Par exemple, la création d'un bloc valide pour d=7 demande 29min et comme la période de vote est limitée (une journée en France par exemple), en choisissant bien le d cela empêche les fraudeurs de créer des blocs très rapidement.
Aussi nous sommes assurés que la majorité de la communauté ne fraudera pas car il est dans leur intérêt de suivre les règles (créer des blocs valides et faire confiance à la chaîne la plus longue).