Mathis ROMERO - 3808418
Flavio SENS - 28616341
Groupe 4

Projet Blockchain

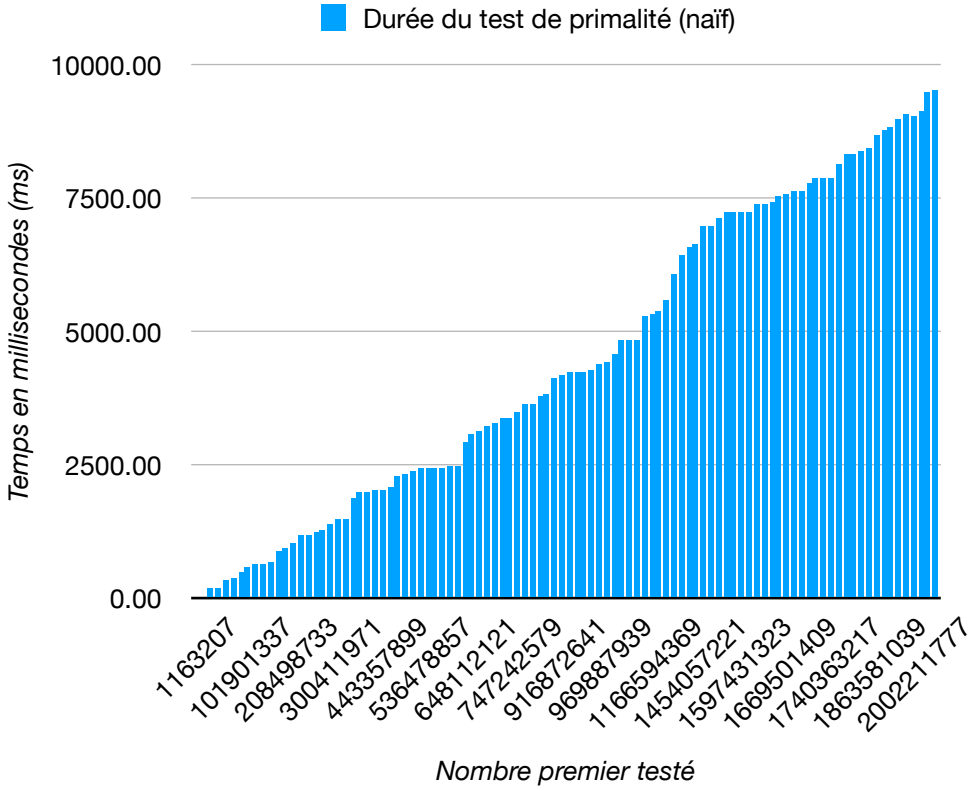
Au cours de ce projet, nous chercherons à modéliser l'organisation d'un processus électoral. Pour ce faire, nous mettrons en place des outils cryptographiques afin de créer des déclarations sécurisées par chiffrement asymétrique et ainsi créer une base centralisée ou décentralisée de déclarations au travers d'un mécanisme de consensus.

Les différentes parties sont réparties dans des fichiers différents avec leurs headers respectifs, et le main est constitué d'un grand switch permettant de tester indépendamment les différentes fonctionnalités du programme.

Les scripts shell fournis permettent de créer une batterie d'appels afin de tester facilement les différentes fonctionnalités, ainsi que de tester la complexité de certains algorithmes. Vous pourrez trouver les jeux d'essai utilisés pour tester les fonctions via ces scripts.
Pour utiliser le fichier d'instructions au lieu d'entrer les essais à la main, il suffit de passer la constante INPUT de 0 à 1.

1.1) La complexité de ***is_prime_naive*** est en O(p), une complexité linéaire. Cela s'explique de par la boucle for qui parcourt tous les nombres impairs entre 3 et p.

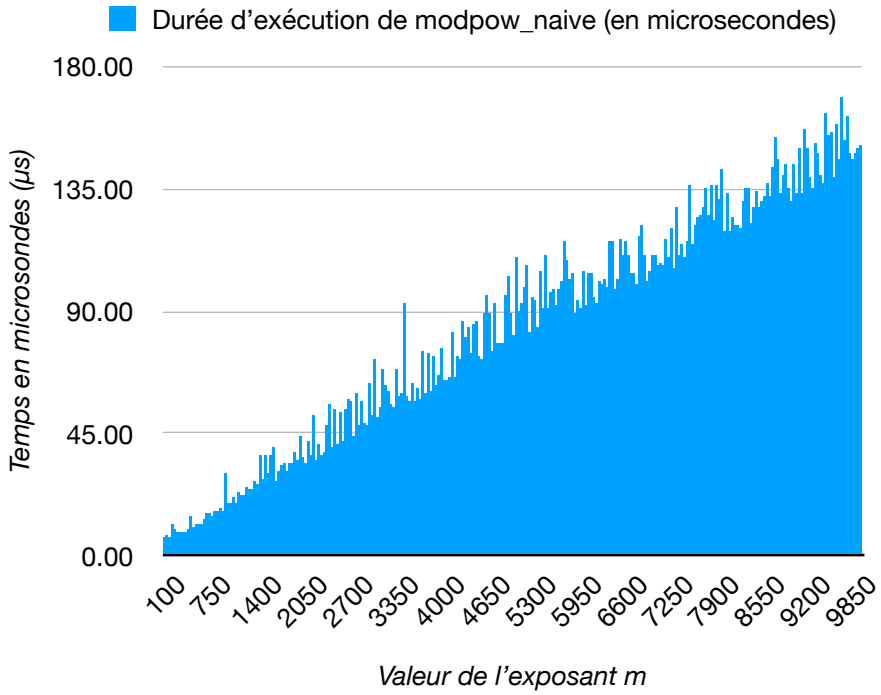
On le constate notamment au travers du graphique suivant, ainsi que le fait que l'algorithme devient très rapidement très lent à mesure que le nombre premier testé est grand : cela justifie le recours au test de Miller-Rabin.
1.2) En moins de 2 millièmes de secondes, nous arrivons à tester des nombres premiers



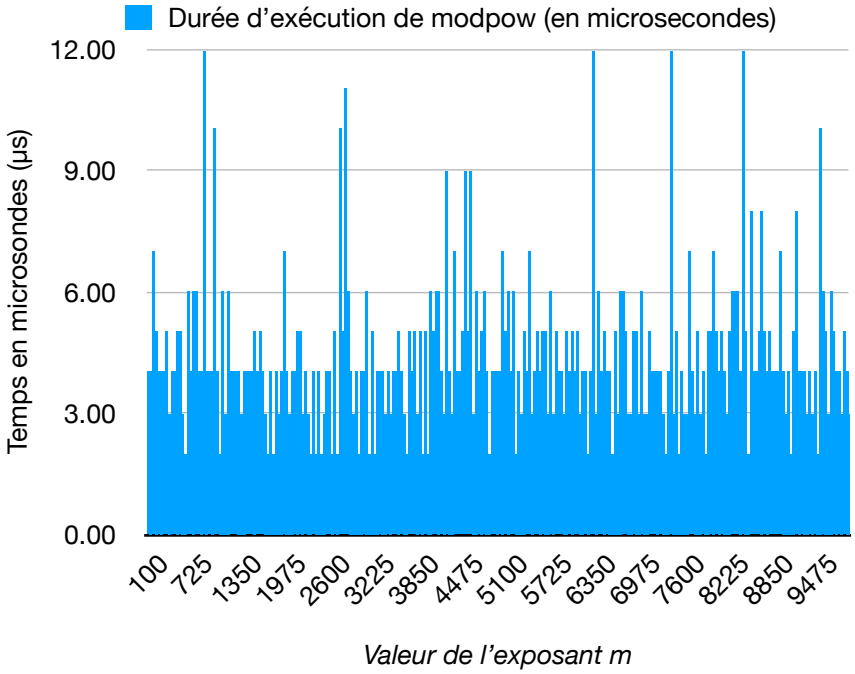
jusqu'aux alentours de 345041 (voir action numero 2 du main pour effectuer le test).

1.3) La complexité de ***modpow_naive*** est en O(m), donc une complexité linéaire, induite par la boucle for qui effectue 2 opérations m fois.

1.5) A l'aide du script ***3-test_modpow_naive.sh***, nous avons pu confirmer la complexité linéaire au travers de ce graphique :



En utilisant le même jeu de données avec la fonction ***modpow***, on obtient ceci :

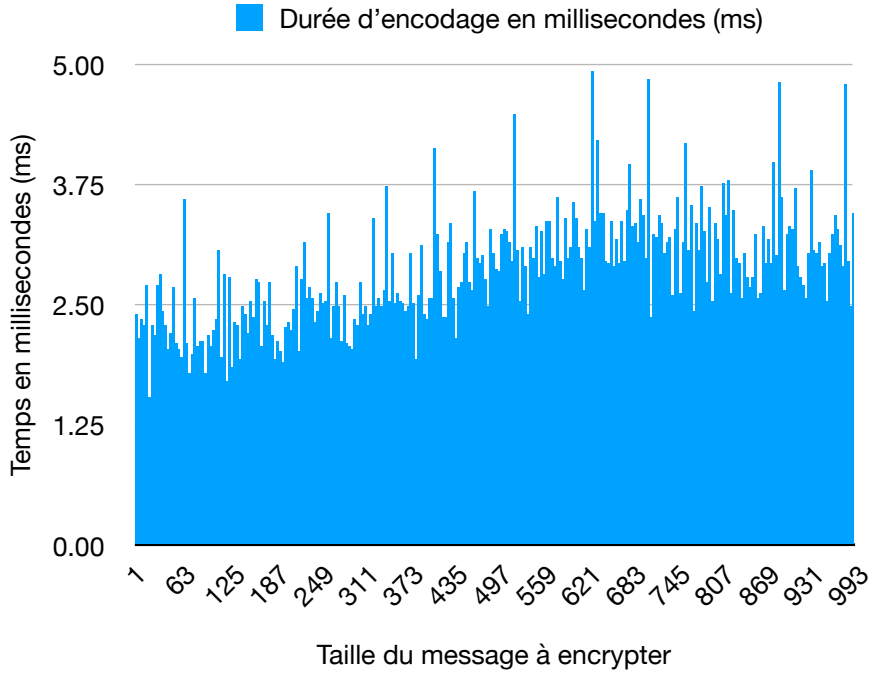


L'exposant m étant divisé par 2 à chaque appel récursif, on s'attend à avoir une complexité logarithmique, donc en O(log(m)).
C'est effectivement ce que l'on retrouve grâce à notre graphique, même s'il nous manque des valeurs pour que cela apparaisse clairement, mais nous sommes rapidement limités par les valeurs atteignables par une variable *long*.

1.7) Il y a une probabilité de 3/4 pour qu'une valeur soit un témoin de Miller d'un nombre p non premier, ainsi il y a donc 1 chance sur 4 pour qu'elle n'en soit pas témoin.
Si on test avec k valeurs, il y a donc une chance de $1/4^k$ pour qu'une valeur testée ne soit un témoin.

Au final, une borne supérieur sur la probabilité d'erreur de l'algorithme basé sur Miller-Rabin est $4^{-}k$.

Exercice 2 :

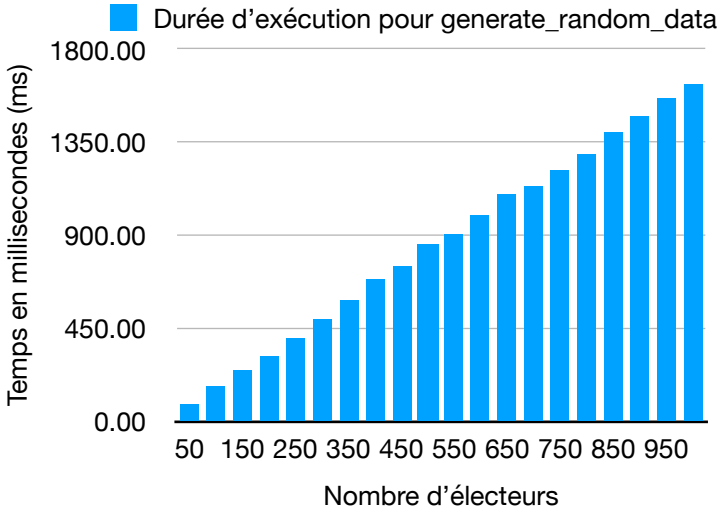


Après l'implémentation complète du processus d'encodage et de décodage, on remarque au travers du graphique que ces 2 opérations profitent d'une complexité logarithmique.
En effet, la complexité logarithmique de modpow bénéficie aux algorithmes d'encodage et de décodage.

Exercice 3 :

Toutes les fonctions définies au cours de l'exercice trois fonctionnent parfaitement et peuvent être testées par le biais de l'**option numéro 7** dans le main.

Exercice 4 :



La complexité de ***generate_random_data*** est linéaire. Ce résultat n'est en rien surprenant étant donné les différentes boucles for non imbriquées, notamment celle qui lit le fichier d'électeurs pour trouver les candidats.
On se rend compte que cette version est rapidement limité, car déjà aux alentours d'un millier d'électeurs, on dépasse la seconde en durée d'exécution. Il faut donc l'optimiser afin qu'elle soit applicable en situation réelle.

Exercice 5 :

5.2) On crée la fonction ***append_head_list*** qui fait appel à la fonction ***create_cell_key***.

5.7) Pour ajouter une déclaration signée en tête de liste, on utilisera la fonction
void append_head_listp(CellProtected **l,Protected *pr)

5.9) La fonction d'affichage pour vérifier la fonction de lecture est :
void print_list_protected(CellProtected *LCK)

Exercice 6 :

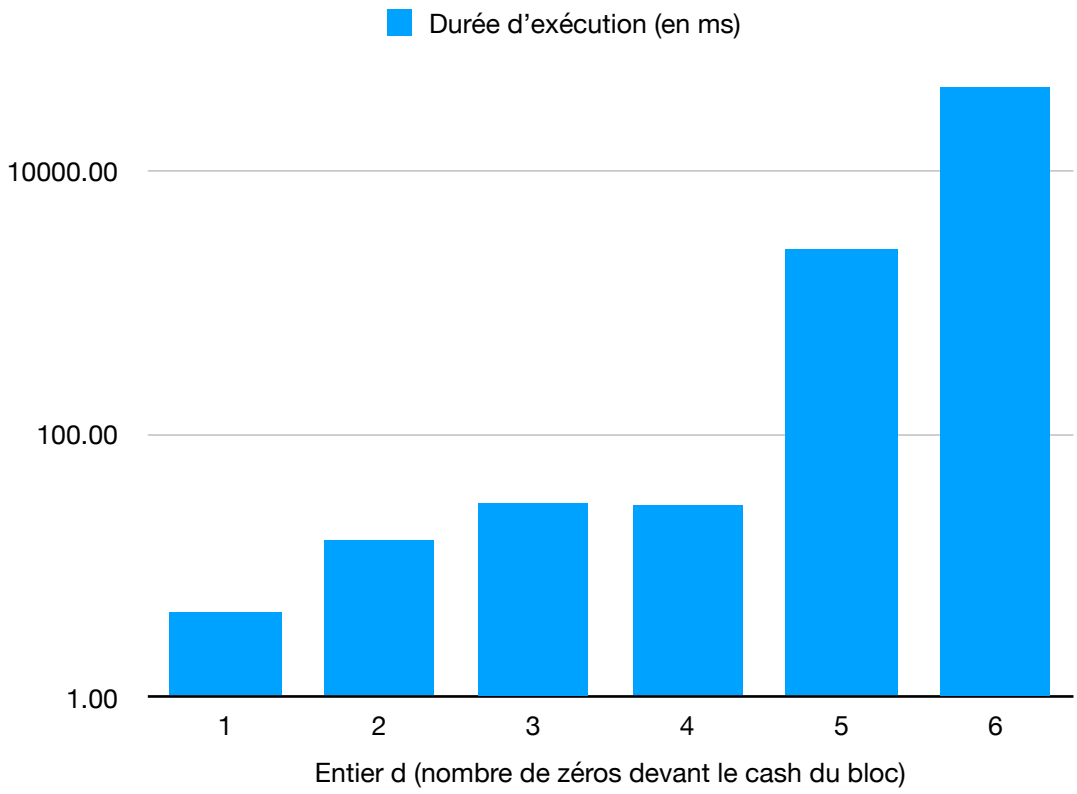
6.1) La fonction **void verify_list(CellProtected **liste)** permet de supprimer toutes les déclarations dont la signature n'est pas valide.

Exercice 7 :

7.1 et 7.2) Pour l'écriture et la lecture de blocs, on définit les fonctions **void write_block(Block *b, char* file)** et **Block * read_block(char *file).**

7.5) Pour utiliser l'algorithme SHA256, on utilisera la fonction **unsigned char * SHA_256(const char *c).**

7.8)



Attention : L'axe des ordonnées est en échelle logarithmique pour mettre en évidence la complexité exponentielle de la fonction compute_proof_of_work().

Exercice 8 :

8.8) La fonction permettant de fusionner deux listes chaînées de déclarations signées est **CellProtected * fusion(CellProtected *liste1, CellProtected *liste2).**

La complexité de cette fonction est en o(n), pour avoir une complexite o(1) il faudrait utiliser des listes doublement chaînées.

Exercice 9 :

9.6) Pour faire appel à cette fonction dans le main, il suffit de demander l'option 11 au main et de rentrer le nombre de citoyens et de candidats désirés.

9.7) Il est effectivement intéressant d'avoir un système décentralisé, comme la blockchain, pour un processus de vote, cela permet de d'assurer de l'indépendance et de la fiabilité des résultats.

Cependant, utiliser le consensus de faire confiance à la plus longue chaîne peut s'avérer très dangereux en pratique à partir du moment où les machines en charge de l'authentification de l'envoi des blocs ne sont pas assez nombreuses ou bien qu'une organisation malveillante possède un très grand nombre de machines permettant l'envoi massif de blocs frauduleux, créant ainsi une chaîne rapidement plus longue que la chaîne authentique.

Dans le cas d'une élection, on imagine facilement une puissance étrangère voulant influencer une élection et ainsi mettant en place des moyens énormes pour envoyer ces blocks frauduleux. C'est d'autant plus facile si comme suggéré dans l'énoncé, les « assesseurs » sont bénévoles : cela limite la sécurité car les citoyens sont moins incités à participer au processus d'authentification.

Un programme malveillant ayant réussit à se propager massivement pourrait aussi mettre à mal ce consensus de chaîne la plus longue. Cela permettrait au groupe à l'origine de ce malware de facilement disposer d'une puissance de calcul phénoménale et ainsi faire passer leurs blocs frauduleux comme authentiques.