Rapport intermédiaire du Projet de Structure de Base de données

Ce projet se divise en plusieurs partie. La première nous a fait mettre en place un système de cryptographie, la deuxième permet de créer et de manipuler des un système de déclaration à l'aide des clés que nous avons introduits dans la partie , la partie 3 qui permet de manipuler les déclarations de la partie 2, et enfin la partie 4 qui permet de sécuriser les déclaration en vérifiant le consentement de chaque vote.

Pour organiser notre dossier, nous avons :
-Le répertoire src, contient le main et un répertoire lib contenant tout les fichiers .c et leurs headers dans le repertoire headers

-Le makefile nous permet de compiler l'ensemble des fichiers du répertoire lib ainsi que le main avec la commande « make all » puis de l'executer en tapant « ./main ».

-Le repertoire bin contient tout les fichiers .o créés par notre makefile.

-Un fichier keys.txt avec l'ensemble des couples clés privé/publics générés par la fonction generate_random_data

-Un fichier candidates.txt avec les couples des candidats choisis par la fonction generate_random_data

-Un fichier déclaration.txt avec l'ensembles des couples et leur déclaration de vote.

Les premières fonctions seront utilisé pour les structures dans la suite:

-rsa.c:
-modpow_naive/modpow -> (a^m)mod n
-is_prime_naive->1 si l'argument est premier et 0 sinon
-random_prime_number -> renvoie un nombre premier aléatoire entre 2 bornes données en paramètres
-generate_key_values -> génère une paire de clé secrete et clé publique

-D’autres fonction sont implémenté mais elles sont données dans le sujet.

-encryption.c:

-decrypt-> décrypte la chaîne donnée avec les deux longs n et s donnés qui ont servi à la crypter.

-encrypt -> crypte la chaîne donnée avec les deux longs n et s donnés.

Nous avons donc crée plusieurs structure et des fonctions qui permettent de les manipuler:

-Key : valeur de la clé avec n=p*q deux nombres premiers permettant de trouver la valeur de la clé.

-init_key -> initialise la clé ;

-init_pair_keys -> initialise une paire de clé publique et clé privé ;

-key_to_str -> retourne la clé sous forme de chaîne de caractère ;

-str_to_key -> initialise une clé grâce à une clé sous forme de chaîne de caractère.

-cellKey : cellule de Liste chaînée de Key.

-create_cell_key -> crée une cellule de liste chaînée de clé avec une clé donné en argument ;

-ajout_en_tete->ajoute en tête d’une liste chaînée de clé donné en argument une Key donnée en argument ;

-read_public_keys -> renvoie une liste chaînée de cellKey contenant les clés (Key) contenus dans un fichier donné en argument;

-print_list_keys -> affiche toute les cellule d’une cellKey ;

-delete_cell_key -> supprime une cellule de cellKey et libère la mémoire allouée ;

-delete_liste_key -> supprime une liste chaînée de clé donné en paramètre et libère la mémoire;

-Signature : message encrypté avec sa taille.

-init_signature->initialise une signature avec un message encrypté(le pointeur) et sa taille donné en paramètre ;

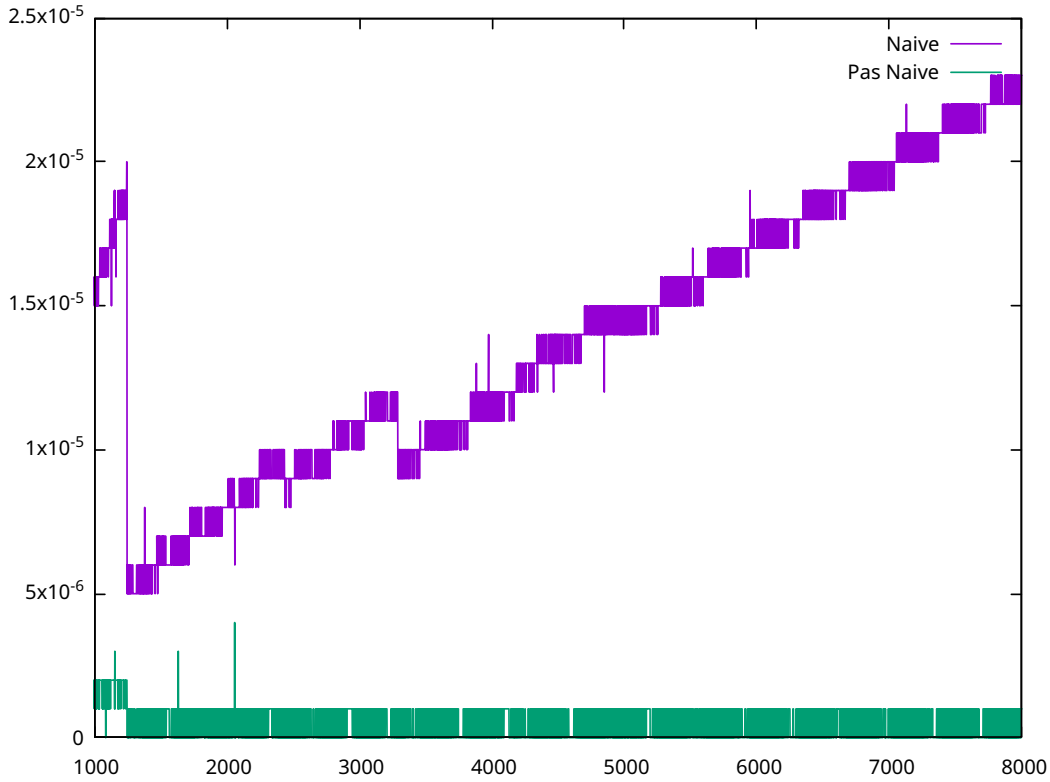
-sign->initialisé une signature avec un message donnée en paramètre qui sera encrypté avec une clé donné en paramètre également (en utilisant la fonction encrypt);
-signature_to_str(fourni) ;
-str_to_signature (fourni) ;
-delete_signature -> supprime une signature et libère la mémoire.

-protected : contient une Key avec un message et une signature.
-init_protected-> initialise une déclaration de type Protected avec un message, une Key et une signature donné en paramètre;
-verify-> vérification de la signature avec la clé contenu dans la déclaration et le message originale;
-protected_to_str-> renvoie une chaîne de caractère contenant les information du Protected en argument;
-str_to_protected->initialise un Protected avec les information de la chaîne de caractère passée en argument;
-delete_protected-> supprime un Protected et libère la mémoire.

-cellProtected : cellule de liste chaînée de Protected.
-create_cell_protected->creation d'une cellProtected contenant le Protected passé en argument;
-ajout_en_tete_protected->ajoute en tête d'une liste chaînée de cellProtected donné en argument une cellProtected donné en argument ;
-afficher_cell_protected-> affiche toutes les cellules d'une liste chaînée cellProtected;
-read_protected -> renvoie une liste chaînée cellProtected contenant les Protected contenus dans un fichier donné en argument;
-delete_cell_protected -> supprime une cellule d'une cellProtected en libérant la mémoire allouée ;
-delete_liste_protected -> supprime une liste chaîne de cellProtected en libérant la mémoire
-verification_fraude -> vérifie les signature de chaque cellule d'une liste chaînée cellProtected.

Réponses Aux Questions:

1.1) La complexité est en 0(n).
1.2) Le nombre premier le plus grand calculable en 2seconde avec cette fonction est de l'orde de 10^9.



1.3) La complexité est en O(m).

1.5) On a bien que modpow_naive qui est bien plus lente que modpow: modpow est quasi constante tandis que modpow_naive est croissante. modpow est en complexité O(log(n)) tant dis que modpow_naive est en O(n)

1.7) (p-3)*(1/3) est la borne supérieure d'erreur.

Main.c

Tout d'abord, nous avons implémenté une fonction « generate_random_data(int nv, int nc) » qui génère nv couple de clé de votants (dans keys.txt) et qui choisit parmi les nv votants nc couple de clé candidats (dans candidates.txt) . Puis elle génère une déclarations par citoyen qui est le vote du citoyen pour candidat aléatoire (dans declarations.txt).

Dans notre main, nous avons dans un premier temps testé nos fonctions grâce aux plusieurs main fournis, puis , nous avons simulé une population de

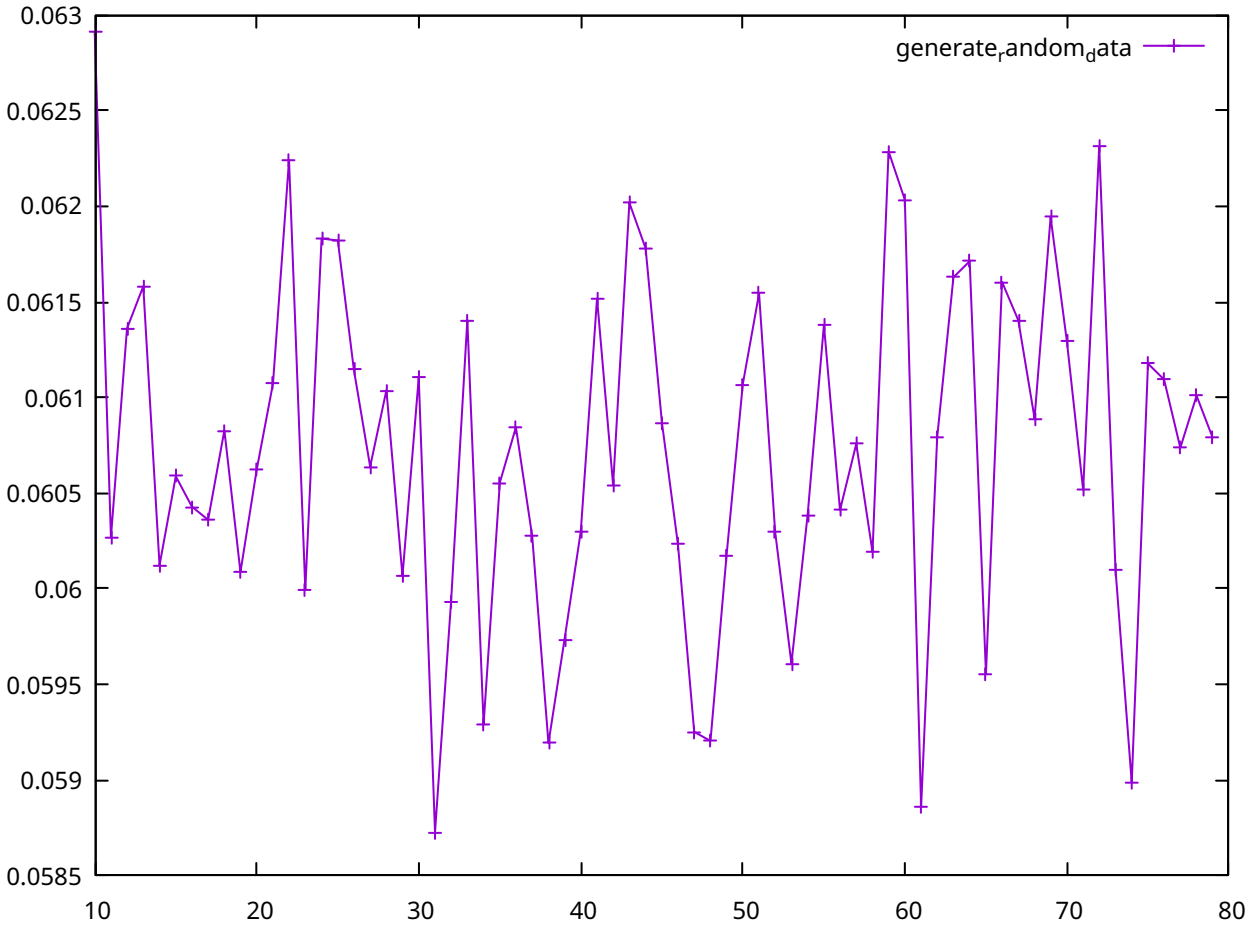
100 citoyens avec 10 candidats et nous avons vérifié les fraudes, ce qui nous donnent donc 3 fichiers txt à la fin de l'executions du main :

-keys.txt avec les 100 couples de clés des citoyens créés.

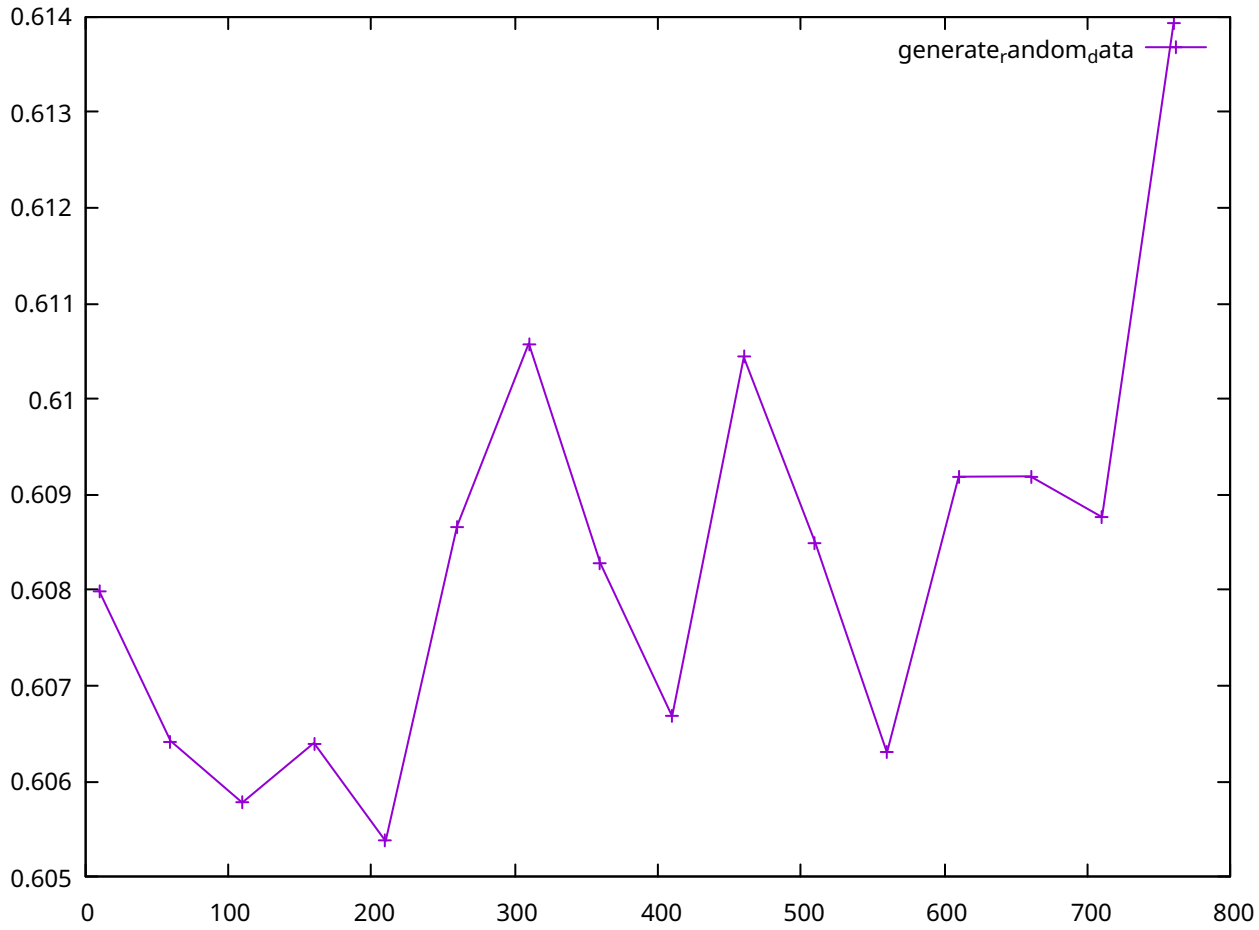
-candidates.txt qui contient les 10 couples de clés des candidats choisit aléatoirement.

-declaration.txt qui contient tout les couples de clés des votants et leurs déclaration (à condition qu'ils n'aient pas fraudé).

À la fin de notre test, nous introduisons une déclarations frauduleuse pour vérifier que notre fonction de vérification supprime bel et bien la fraude de la liste chaînée. Nous affichons la liste avant le test de vérification des fraudes puis une fois après pour montrer que la fonction supprime bien la déclaration frauduleuse



Voici le tableau des performances de notre fonction generate_random_data pour une 100 votants et un nombre de candidats allant de 1 à 80.
On voit que les performances sont assez aléatoire de part la nature même de la fonction.



Voici le tableau des performances de notre fonction generate_random_data pour une 1000 votants et un nombre de candidats allant de 1 à 800 en augmentant de 50 pour éviter une temps de calcul conséquent.
On voit que les performances sont assez aléatoire de part la nature même de la fonction.

Nous tenons à vous informer que nous avons été aidés par Philippe et Sevag pour certaine fonctions, cependant toutes les fonctions sont comprises et nous les avons nous même travaillés.

Nous avons également travaillé en utilisant un répertoire sur GitHub. Voici le lien https://github.com/paolo944/TheProjet