

# Projet : Blockchain appliquée à un processus électoral

## Partie 1 : Implémentation d'outils de cryptographie

### Exercice 1 – Résolution de problème de primalité

1) Tout d'abord on implémente une fonction **int is\_prime\_naïve(long p)** qui prend un entier impair  $p$  et renvoie 1 s'il est premier, 0 sinon. Cette fonction est de complexité  **$O(p)$**  puisque **is\_prime\_naïve(long p)** vérifie sa primalité avec chaque entier entre 3 et  $p-1$ .

2) Le plus grand nombre premier qu'on puisse tester en 2 secondes est 230063.

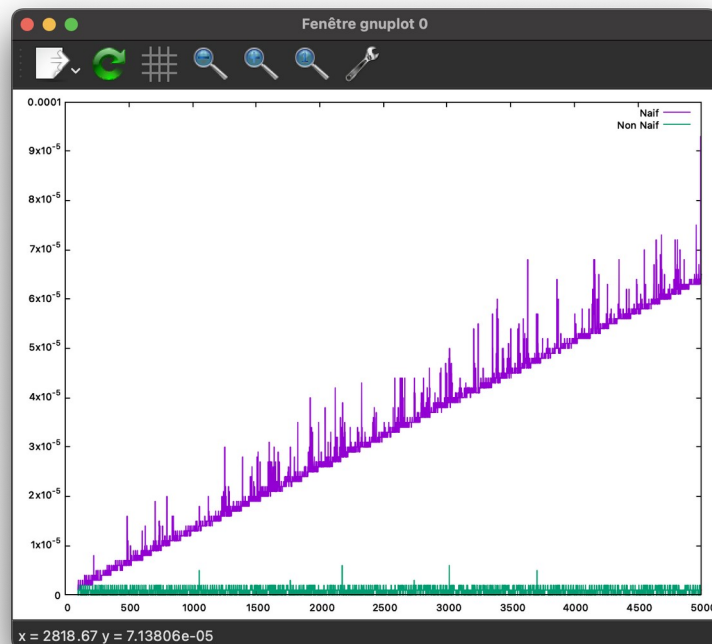
Remarque: Quand le nombre est trop grand il sera codé en négatif donc renvoie un 0 ;

3) On a ensuite implémenté la fonction **long modpow\_naïve(long a, long m, long n)** qui prend un long  $a$ , le multiplie  $m$  fois et applique sur le résultat le modulo  $n$  à chaque itération ( $m$  fois), cette fonction renvoie au final  $a^m \% n$  et possède une complexité  **$O(m)$** , puisqu'on est contraint d'appliquer cette méthode  $m$  fois.

4) Pour améliorer la complexité de la fonction **long modpow(long a, long m, long n)**, on a réalisé des élévations au carré pour passer de  **$O(m)$**  à  **$O(\log_2(m))$** . Ce qui nous a permis d'implémenter la fonction **modpow(long a, long m, long n)** qui réalise le même résultat que sa version naïve.

5) Nous avons ensuite calculé les performances des 2 fonctions modpow à l'aide de **gnuplot** ce qui nous a permis de tracer un graphique.

Pour le tracer, nous avons utilisé un échantillon de 5000 entier, calculé grâce aux fonctions modpow, ensuite nous avons calculé le temps CPU de chaque itérations et on a écrit les résultats dans un fichier **compmod.txt** ([main.c 1.5](#)), qui va servir de base de donnée pour le graphique. Avec la commande **gnuplot -p < commande.txt**, on affiche le graphique, voici le résultat :



On peut ainsi constater que la version naïve augmente de façon linéaire ce qui est trop lent, et j'ai seulement testé avec  $m$  appartenant à  $[100, 5000]$ , on peut déjà constater le gouffre entre la version naïve et non-naïve, puisqu'on va manipuler des entiers très grand la méthode non-naïve est celle qu'on va utiliser.

7) Puisque  $3/4$  des valeurs entre 2 et  $p-1$  sont des témoins de Miller  $p$ , la probabilité d'erreur est de  $3/4^k$ . Plus  $k$  (nombre de tests) est grand, moins il y aura de chance d'erreur.

8) On va implémenter une fonction **random\_prime\_number(int low\_size, int up\_size, int k)**, qui génère un entier entre  $2^{\text{low\_size}}$  et  $2^{\text{up\_size}+1}-1$  et tester sa primalité en fonction de **k**, renvoie l'entier généré s'il est conforme au teste de Miller. (ligne 87-117)

Je commence par créer les bornes inférieurs et supérieurs en fonction du paramètre **low\_size** et **up\_size** respectivement, à l'aide de 2 boucles, ensuite je génère un entier p entre borneSup et borneInf-1. Je créer de nouveau une boucle qui va tester la primalité de p avec le test de Miller.

- Si p ne vérifie pas le test, on va rester dans la boucle et générer de nouveau un entier p
- si l'entier p vérifie le test on peut quitter la boucle et le retourner

```
730     printf("modpow_naive : %ld\n", modpow_naive(2, 9, 29));
731     printf("modpow : %ld\n", modpow(2, 9, 29));
732
733     printf("is_prime_miller : %d\n", is_prime_miller(21, 1));
734     printf("rand_long : %ld\n", rand_long(4, 31));
735
736     printf("random_prime_number : %ld\n", random_prime_number(4, 7, 3));
737
```

PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

```
MacBook-Pro-de-Zhuang:Projet jm$ ./exo1
modpow_naive : 19
modpow : 19
is_prime_miller : 0
rand_long : 26
random_prime_number : 104
```

## Exercice 2 - Implémentation du protocole RSA

1) On a implémenté la fonction **long generate\_keys(long p, long q, long\* n, long\* s)** qui permet de générer la clé publique **pKey = (s,n)** et la clé secrète **sKey = (u,n)** à partir des nombres premiers **p** et **q**.

Je génère **s**, j'applique le gcd sur **s** et je vérifie toutes ses conditions en même temps, dans la boucle pour pouvoir la quitter :

- **s** de la clé publique doit être premier
- le gcd de **s** doit être = 1
- **u** de la clé secrète doit être positif
- **s** et **u** doit être différent

2) On a implémenté la fonction **long\* encrypt(char\* chaine, long s, long n)**, qui chiffre la chaîne de caractère **chaine**, avec la clé publique **pKey = (s,n)** et renvoie la chaîne chiffrée en un tableau de long.

Tant que **chaine[i]** est différent de '\0', on applique modpow sur **chaine[i]** et la clé publique et on le stock dans **msg[i]** et on va retourner le message chiffré **msg**.

3) Pour finir on implémente la fonction **long\* decrypt(long\* crypted, int size, long s, long n)** qui déchiffre un message **crypted**, à l'aide de la clé secrète **sKey = (u,n)** et la taille du tableau (**size**) et renvoie le message non codé.

On fait l'inverse de la question précédente pour ensuite retrouver le message initial.

```
MacBook-Pro-de-Zhuang:Projet jm$ ./exo1
cle publique = (fb3, 10d5)
cle privee = (1df, 10d5)
Initial message : Hello
Encoded representation :
Vector : [ec3  786  aa6  aa6  ce9  ]
Decoded : Hello
MacBook-Pro-de-Zhuang:Projet jm$
```

J'utilise un mac et donc je ne peux pas utiliser valgrind mais j'ai une commande qui permet de me montrer les éventuelles leaks, et il y en a pas à ce stade du projet, sachant que j'ai mis les exercices précédents en commentaire.

```
Analysis Tool:  /usr/bin/leaks

Physical footprint:      2084K
Physical footprint (peak): 2084K
-----

leaks Report Version: 4.0
Process 10278: 170 nodes malloced for 13 KB
Process 10278: 0 leaks for 0 total leaked bytes.

MacBook-Pro-de-Zhuang:Projet jm$ █
```

## Partie 2 : Création d'un système de déclarations sécurisés par chiffrement asymétrique

### Exercice 3 – Manipulations de structures sécurisées

1) Chaque citoyen possède une carte électorale définie par le couple de clé publique et clé secrète, nous allons implémenter les clés grâce à la structure **Key** qui sera composé de ses attributs **v** et **n** telle que **key = (v,n)**.

2) Nous allons implémenter la fonction **void init\_key(Key\* key, long val, long n)** qui va initialisée une clé déjà existante (**key**) avec les paramètres **val** et **n**.

3) La fonction void **init\_pair\_keys(Key\* pKey, Key\* sKey, long low\_size, long up\_size)** va créer la carte électorale du citoyen en déterminant **s** et **u** de la clé publique et privée respectivement en fonction des paramètres **low\_size** et **up\_size**, en d'autre terme le protocole RSA . Sachant que **pKey** et **sKey** doivent être allouées au préalable.

4) **char\* key\_to\_str(Key\* key)** et **Key\* str\_to\_key(char\* str)** passe d'une clé à sa forme sous chaîne de caractère et inversement, la chaîne de caractère doit être de la forme (v,n) telle que **v** et **n** les attributs de la structure **Key**.

```
pKey : 346d , 42d9
sKey : 1ef5 , 42d9
key to str : (346d,42d9)
str to key : 346d , 42d9
(346d,42d9) vote pour (7d63,a405)
```

5) La structure **Signature** est composé d'un tableau qui représente le message chiffré et sa taille ([exo1.h](#))

6) La fonction **Signature\* init\_signature(long\* content, int size)** alloue une structure **Signature**, la « remplit » avec le tableau **content** et la taille **size** pour ensuite renvoyer la **Signature**.

7) La fonction **Signature\* sign(char\* mess, Key\* sKey)** signe le message **mess** grâce à la fonction **encrypt** et le paramètre **sKey** et initialise la signature engendrée.

9) La structure **Protected** est composé de la clé publique **pKey** de l'électeur, son message **mess**, dans notre cas la clé du candidat pour qui l'électeur a voté, et la signature **sgn** qui est censé être le message **mess** chiffré grâce à la clé privée de l'électeur. ([exo1.h](#))

10) La fonction **Protected\* init\_protected(Key\* pKey, char\* mess, Signature\* sgn)**, alloue une signature et l'affecte avec les paramètres **pKey, mess, sgn**. Sans pour autant vérifier si la signature est conforme.

11) On a ensuite implémenté la fonction **int verify(Protected\* pr)** qui décrypte la signature contenu dans **pr** la compare à mess, renvoie 1 si elle est conforme sinon 0.

12) Pour finir on implémente les fonctions **char\* protected\_to\_str(Protected\* pr)** et **Protected\* str\_to\_protected(char\* str)**, qui passe d'un Protected à sa version sous chaîne de caractère et inversement.

```
(346d,42d9) vote pour (7d63,a405)
signature : Vector : [299      27f5      3c73      3470      19b1      2ff5      2554      2308      15cd      3512      2275      ]
signature to str : #299#27f5#3c73#3470#19b1#2ff5#2554#2308#15cd#3512#2275#
str to signature : Vector : [299      27f5      3c73      3470      19b1      2ff5      2554      2308      15cd      3512      2275      ]
Signature valide
protected to str : (346d,42d9) (7d63,a405) #299#27f5#3c73#3470#19b1#2ff5#2554#2308#15cd#3512#2275#
str to protected : (346d,42d9) (7d63,a405) #299#27f5#3c73#3470#19b1#2ff5#2554#2308#15cd#3512#2275#
```

Les problèmes de leaks ont aussi été résolus avec beaucoup beaucoup de free et pour le moment il n'y a aucun problème à signaler.

```
806     free(pKey); free(sKey); free(pKeyC); free(sKeyC);
807     free(k); free(chaine); free(mess);
808     free(sgn->content); free(sgn); free(signStr); free(strSign->content); free(strSign); free(pr); free(c3);
809     free(pr1->mess); free(pr1->pKey); free(pr1->sgn->content); free(pr1->sgn); free(pr1);
810     free(key); free(s);
9.1.1

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

leaks Report Version: 4.0
Process 11077: 170 nodes malloced for 13 KB
Process 11077: 0 leaks for 0 total leaked bytes.
```

## Exercice 4 – Création de données pour simuler le processus de vote

**void generate\_random\_data(int nv, int nc).**

Si le nombre de candidats > nombre de voteurs, on quitte la fonction.

On va ouvrir un flux vers keys.txt, j'alloue un tableau de chaîne de caractère **keys** qui aura la taille du nombre de voteurs.

Je fais une boucle sur le nombre de voteurs, puis j'initialise le couple clé publique clé privée et je les transforme en chaîne de caractère à l'aide de key\_to\_str.

On va parcourir le tableau **keys** avec une seconde boucle qui sera imbriquée à celle des voteurs et vérifier si le couple de clé généré a déjà été vu, autrement dit, le couple appartient ou non au tableau **keys**

- S'il appartient à **keys** je fais i-- pour éviter de passer à l'itération suivante car je n'ai pas créé le couple et pour finir je fais un break pour éviter de vérifier les éléments suivants de **keys**.
- S'il n'appartient pas à **keys** c'est à dire qu'il a parcouru tout le tableau **keys** et qu'il ne l'a pas trouvé et donc que j == i.  
Donc je peux l'écrire dans le fichier keys.txt et le mettre dans le tableau **keys** puisqu'il a été vu

Idem pour le fichier candidates.txt, sauf que je rajoute un nombre aléatoire entre 0 et nv-1 qui va déterminer les candidats parmi les voteurs.

Pour le fichier declarations.txt, je fais une boucle sur le nombre de voteurs, je génère un nombre aléatoire entre 0 et nc-1 (**index**) pour déterminer pour qui l'électeur va voter.

Je stock le couple de clé publique et clé privée dans les variables **public** et **private** respectivement à l'aide du sscanf.

Je transforme **private** en Key\* avec str\_to\_key pour signer le message (**clé publique du candidat**). La signature engendrée sera elle aussi transformée, grâce à signature\_to\_str (**vote**) et je fini par écrire dans le fichier declarations.txt en utilisant les variables **public**, **candidates[index]**, et **vote**.

```
833  /* ----- Exercice 4 ----- */
834
835  generate_random_data(10, 3);
836
837  /* ----- Exercice 5 ----- */
838
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

leaks Report Version: 4.0  
Process 11842: 169 nodes malloced for 9 KB  
Process 11842: 0 leaks for 0 total leaked bytes.



## keys.txt

```
≡ keys.txt
1 (95cf,ce9b) (1aff,ce9b)
2 (e3,4db) (20b,4db)
3 (11c5,4271) (e4d,4271)
4 (11,2119) (b51,2119)
5 (1ead,447d) (1c45,447d)
6 (95,197) (1d,197)
7 (18f5,7691) (209,7691)
8 (1df,733) (29f,733)
9 (3b,1bbb) (6cb,1bbb)
10 (8963,9dab) (447b,9dab)
```

## candidates.txt

```
≡ candidates.txt
1 (95cf,ce9b)
2 (e3,4db)
3 (3b,1bbb)
```

## declarations.txt

```
≡ declarations.txt
1 (95cf,ce9b) (95cf,ce9b) #51c#57a4#32f2#378#5979#3427#378#5ce4#57a4#be8a#820#
2 (e3,4db) (3b,1bbb) #4b1#23e#f1#294#466#f1#f1#f1#216#
3 (11c5,4271) (95cf,ce9b) #4bd#2e30#2a5b#407d#3211#2133#407d#cd0#2e30#1f0f#14f5#
4 (11,2119) (e3,4db) #62b#8dc#18d4#136e#c8e#c78#1225#b18#
5 (1ead,447d) (95cf,ce9b) #1333#3282#15bd#36e5#cbf#126b#36e5#1208#3282#cd3#33d0#
6 (95,197) (95cf,ce9b) #fa#192#47#176#190#134#176#10e#192#112#5f#
7 (18f5,7691) (3b,1bbb) #14bd#6936#46fe#537b#743#46fe#46fe#46fe#11ac#
8 (1df,733) (3b,1bbb) #682#294#2a8#2fd#48e#2a8#2a8#2a8#411#
9 (3b,1bbb) (3b,1bbb) #1003#1447#f7a#ef1#1b14#f7a#f7a#f7a#649#
10 (8963,9dab) (e3,4db) #984a#1237#2246#982#89bd#6e20#957d#1e99#
```

## Partie 3 : Manipulation d'une base centralisée de déclarations

### Exercice 5 - Lecture et stockage des données dans des listes chaînées

- 1) En utilisant la structure **CellKey**, on crée une fonction **CellKey\* create\_cell\_key(Key\* key)** qui alloue une liste chaînée de type **CellKey\*** avec l'élément **key**.
- 2) On a implémenté une fonction **void tete\_keys(CellKey\*\* list, Key\* key)**, qui ajoute l'élément **key** dans la liste chaînée **list** en tête de liste.
- 3) La fonction **CellKey\* read\_public\_keys(char\* fichier)** lit soit le fichier « **keys.txt** » soit « **candidates.txt** » et pas un autre sinon la fonction se termine. Elle crée une liste chaînée de **CellKey\*** avec les clés publiques présentes dans l'un des 2 fichiers. A la fin de la fonction, on aura la version miroir du fichier car on ajoute en tête à chaque itérations.  
Lit la ligne avec **fgets** et injecte dans les variables **public** et **prive** ce qui se trouve dans ligne avec **sscanf**, transforme **public** en **Key\*** et l'ajoute en tête pour créer la liste des clés.
- 4) La fonction **void print\_list\_keys(CellKey\* LCK)** affiche les clés publiques présentent dans **LCK**.  
On va parcourir chaque élément de **LCK**, les transformer en chaîne avec **key\_to\_str** et on affiche les affiches.
- 5) La fonction **void delete\_cell\_key(CellKey\* c)** permet de supprimer une cellule de la liste chaînée **c**, quant à **void delete\_list\_keys(CellKey\* c)**, elle supprime TOUTES les cellules contenues dans **c** en se servant de **delete\_cell\_key**.
- 6) **CellProtected\* create\_cell\_protected(Protected\* pr)** alloue et initialise une liste de **protected** en commençant par **pr**.
- 7) **void tete\_protected(CellProtected\*\* list, Protected\* pr)** ajoute une déclaration **pr** en tête de la liste **list**
- 8) **CellProtected\* read\_protected(char\* fichier)** lit un fichier contenant des déclarations à l'intérieur et crée une liste à partir de ses déclarations.  
Pour implémenter cela, je lis avec **fgets** la ligne et j'insère ce qu'il a dans la ligne dans les variables **pKey**, **candidat** et **sgn** (**sscanf**). Avec les 3 variables, je fais un **sprintf** dans une variable de type **char\*** qui se nommera **declarations**, je transforme **declarations** en

protected avec str\_to\_protected et je l'ajoute en tête et ainsi de suite jusqu'à que fgets lit une ligne vide.

9) **void print\_list\_protected(CellProtected\* LCP)** idem que print\_list\_keys mais on utilise la fonction protected\_to\_str pour les éléments de LCP et on les affiche.

10) **void delete\_cell\_protected(CellProtected\* c)** désalloue la cellule **c** entière et **void delete\_list\_protected(CellProtected\* c)** applique la fonction dite précédemment n fois sachant que n est le nombre de déclarations.

```
leaks Report Version: 4.0
Process 12311: 170 nodes malloced for 13 KB
Process 12311: 0 leaks for 0 total leaked bytes.
```

Toujours pas de fuite mémoire ici aussi

```
844  /* ----- Exercice 4 ----- */
845
846  generate_random_data(10, 3);
847
848  /* ----- Exercice 5 ----- */
849
850
851  CellKey* cKey = read_public_keys("candidates.txt");
852  print_list_keys(cKey);
853  delete_list_keys(cKey);
854
855
856  CellProtected* cp = read_protected("declarations.txt");
857  print_list_protected(cp);
858  delete_list_protected(cp);
859
860
861  /* ----- Exercice 6 ----- */
```

PROBLEMS	OUTPUT	TERMINAL	DEBUG CONSOLE
MacBook-Pro-de-Zhuang:Projet jm\$ ./exo1			
---	1 (1525,404f) ---		
---	2 (139d,6335) ---		
---	3 (ad,995) ---		
-----	1 (8f9,22cd) (ad,995) #e57#f33#1076#1456#1b#1b3e#1d23#		
-----	2 (2c5,1123) (1525,404f) #10a7#176#916#bba#916#e67#1b7#de2#1b7#121#cf9#		
-----	3 (1105,68cb) (1525,404f) #680#4dd8#2866#139d#2866#5aec#25e1#320#25e1#5cef#1de2#		
-----	4 (4325,4805) (ad,995) #3f0f#3b45#35e0#35c9#3a86#3a86#3f82#45f6#		
-----	5 (b,30b) (139d,6335) #1c2#c5#248#39#2a2#2fd#28a#248#1dd#cd#		
-----	6 (44f,1399) (1525,404f) #b8c#587#e99#270#e99#3e6a#687#e6a#11e9#a7d#		
-----	7 (1525,404f) (139d,6335) #2fe7#291f#23b7#2cb5#360c#c80#2896#23b7#23b7#12fc#168#		
-----	8 (ad,995) (1525,404f) #355#5ba#886#3e6#886#32e#84d#35f#84d#34d#627#		
-----	9 (139d,6335) (139d,6335) #4d60#5360#3421#3d24#79#52ce#36ef#3421#3421#119a#3310#		
-----	10 (5f3,6cb) (ad,995) #446#3ba#563#10c#116#116#67d#572#		

## Partie 4 : Implémentation d'un mécanisme de consensus

### Exercice 6 - Détermination du gagnant de l'élection

1) Nous allons créer une fonction nommée **void fraude(CellProtected\*\* LCP)** qui supprime toutes les tentatives de fraudes c'est-à-dire les signatures non conformes. Après application, il ne reste que les déclarations valides dans **LCP**.

- Si la déclaration est vérifiée, on passe au suivant.
- Sinon on vérifie si l'élément courant se trouve en tête de liste
  - Si oui l'élément suivant devient la tête de **LCP**
  - Sinon on relie l'élément précédent de l'élément courant à son suivantEt ensuite on supprime l'élément courant.

2) La fonction **HashCell\* create\_hashcell(Key\* key)** alloue une cellule de la table de hachage et l'initialise avec **key** et son champ valeur à 0.

3) La fonction **int hash\_function(Key\* key, int size)** retourne la position en fonction de **v** et **n** telle que **key = (v,n)** entre **0** et **size - 1**.

4) La fonction **int find\_position(HashTable\* t, Key\* key)** cherche dans tableau de hachage **t** la position de **key**, si elle existe. Si l'électeur a été trouvé on renvoie son indice sinon on applique le probing pour lui attribuer une case vide. Pour implémenter cela, j'initialise une boucle qui va commencer par la position trouver par **hash\_function i** et s'il fait un tour complet, ça veut dire qu'il n'y a soit plus de place libre soit que l'élément n'est pas présent.

5) La fonction **HashTable\* create\_hashtable(CellKey\* keys, int size)** crée un tableau de hachage de taille **size** et trouve une place et l'ajoute à l'intérieur de la table pour chacune des clés présentent dans **keys**.

6) La fonction **void delete\_hashtable(HashTable\* t)** désalloue la table de hachage **t** et toutes les cellules à l'intérieur mais pas le contenu des cellules car ils seront désalloués avec **delete\_list\_keys** et **delete\_list\_protected**.

7) Pour finir, la fonction **Key\* compute\_winner(CellProtected\* decl, CellKey\* candidates, CellKey\* voters, int sizeC, int sizeV)** détermine le vainqueur de l'élection dépendant d'une liste de déclarations valide **decl**, des candidats **candidates**, des voteurs **voters**. Pour cela, nous allons créer 2 tables de hachage, 1 pour les voteurs de taille **sizeV** et déterminer s'ils ont déjà voté, 1 pour les candidates de taille **sizeC**, et comptabiliser les votes pour eux.

On va ensuite parcourir la liste de déclarations, et recherché le voteur et le candidat pour qui il a voté avec `find_position`. Grâce aux indices retourner par `find_position`, on pourra déterminer facilement si le message **mess** correspond bien à un candidat et si le voteur n'a pas déjà voté,

si ces conditions sont vérifiées, on incrémente le nombre de vote pour le candidat et l'attribut **val** du voteur sera égal à 1 pour qu'il ne vote pas une 2<sup>ème</sup> fois.

Après avoir comptabilisé toutes les déclarations, on va rechercher le gagnant de l'élection en parcourant la totalité du tableau de hachage des candidats **Hc**, on va comparer chaque itérations avec le **max** (correspond à celui qui a le plus de vote). Pour finir on renvoie le vainqueur, c'est celui qui a **max** votes.

```

186
187     generate_random_data(100, 2);
188     CellKey* candidats = read_public_keys("candidates.txt");
189     print_list_keys(candidates);
190     printf("\n");
191     CellKey* voters = read_public_keys("keys.txt");
192     print_list_keys(voters);
193     printf("\n");
194     CellProtected* cp = read_protected("declarations.txt");
195     print_list_protected(cp);
196     printf("\n");
197
198     compute_winner(cp, candidats, voters, 3, 100);
199
200     // Pour régler les fuites mémoires
201     delete_list_keys(voters);
202     delete_list_keys(candidates);
203     delete_list_protected(cp);
204
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
-----
87 (1b6b,2015) (b57,12ef) #16e9#3a5#5f#f92#7e6#144c#1c07#736#bf8#558# -----
88 (2e3,4f9) (b57,12ef) #37f#289#295#4ab#383#1d3#13c#16f#4ed#3cc# -----
89 (4bd,3e3b) (b57,12ef) #1cb1#ecb#2fa6#aa5#6cd#2e0e#1d6#236a#222f#139# -----
90 (517,12d3) (b57,12ef) #edb#4f9#836#dcb#5c#da8#11ea#8f#240e#2a# -----
91 (1de1,778f) (229d,2aa9) #3973#768c#768c#f9b#660a#1c6a#768c#4dd1#4dd1#f9b#4cb5# -----
92 (229d,2aa9) (b57,12ef) #640#29c9#2142#86a#1927#28c9#26ea#743#1b4b#f28# -----
93 (755,1b1f) (229d,2aa9) #13f1#18af#18af#b12#349#1a25#18af#166b#166b#b12#6c3# -----
94 (20ff,23d1) (229d,2aa9) #1061#13d8#13d8#2193#1db3#1f03#13d8#ccb#ccb#2193#171d# -----
95 (923,459d) (229d,2aa9) #395c#dc2#dc2#37c4#43f7#2094#dc2#4d1#4d1#37c4#2744# -----
96 (3c5b,6307) (b57,12ef) #5575#308a#ba1#f8#3c31#1a3b#208d#5667#66#52b1# -----
97 (8f8d,a07f) (229d,2aa9) #881f#89fb#89fb#55fd#3fd6#4f52#89fb#e8c#e8c#55fd#660f# -----
98 (5e69,61bb) (b57,12ef) #15f5#16fb#4e3e#42b4#5cef#2152#12bf#ec5#20f1#1ad3# -----
99 (18fd,3f1d) (229d,2aa9) #28ad#2d7f#2d7f#2f4b#252e#1ad3#2d7f#1594#1594#2f4b#2272# -----
100 (12bf,1601) (b57,12ef) #411#4fb#e2a#ce0#e9e#10f0#fc4#310#60a#b70# -----

----- Vainqueur : (229d,2aa9) avec 53 votes ! -----
MacBook-Pro-de-Zhuang:Projet jm$ c

```

## Exercice 7 -Structure d'un block et persistance

1)La fonction `void ecriture_fichier(Block* b, char* fichier)` permet d'écrire dans un fichier les données d'un block. Tout d'abord on ouvre le fichier en paramètre en mode `w`, on vérifie si le flux est bien ouvert puis on écrit dans le fichier avec des `fprintf` les contenu du block `b`, et pour la liste des déclaration on fait une boucle `while` pour parcourir la liste et on fait des `fprintf` de `protected` par `protected`.

2)La fonction `Block* lecture_fichier(char* fichier)` permet de lire un block du fichier en paramètre et de le retourner. Tout d'abord on ouvre le fichier en paramètre en mode `r`, on vérifie si le flux est bien ouvert, on alloue avec un `malloc` la place pour un block et on alloue de la place pour les attributs du block, puis on lit dans le fichier avec le `fscanf` selon le format qu'on a écrit puis on a utilisé une boucle `while` et des `fgets` pour les déclaration.

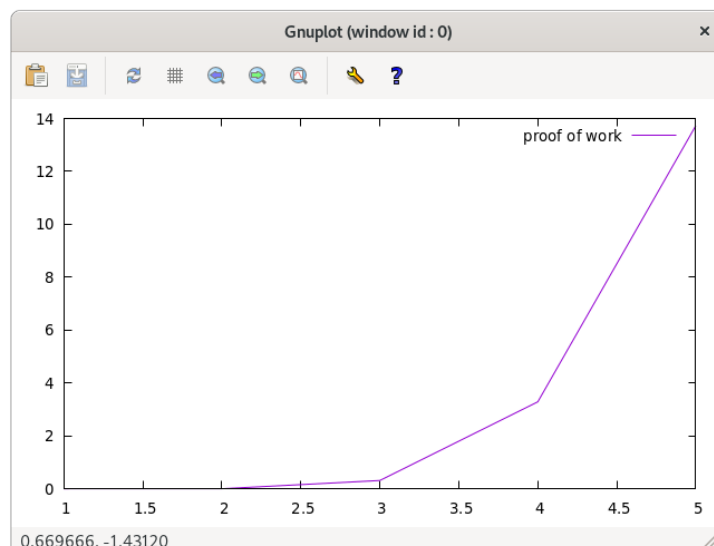
3)La fonction `char* block_to_str(Block* block)` permet de génère une chaîne de caractères représentant le block en paramètre. Tout d'abord on a alloué de la place pour le str qu'on veut retourner on a utilisé un `malloc` avec un très grand chiffre parce que avec les `realloc` on a toujours des problème. Puis on écrit dans `str` les contenu du block avec `sprintf` mais pour la liste des déclaration on a fait un `while` sur les `protected` et à chaque fois on fait un `strcat` du `str` avec le `protected_to_str`(du `protected` courant).

4-5)La fonction `unsigned char* hache(char *chaine)` retourne la valeur hachée de la chaîne de caractère en paramètre avec l'algorithme `SHA256`.

6)La fonction `void compute_proof_work(Block *B,int d)` permet de changer la valeur de hachée du Block `b` en qui commence par `d` zéro successifs. Pour réaliser ces opération on initialise le nonce de `b` à 0, et on implémente deux boucles la première permet de changer valeur hachée et la seconde boucle permet de vérifier si la valeur hachée commence par `d` 0 successifs.

7)La fonction `int verify_block(Block* b,int d)` permet de vérifier le block `b` est valide c'est-à-dire si `b→hash` commence par `d` 0 successifs.

8)



On voit que pour trois 0 successif la fonction prend moins d'une seconde, et ensuite la fonction prend une allure exponentielle.

9) La fonction void delete\_block(Block\* b) permet de supprimer le block en paramètre. Sans supprimer l'auteur et les éléments de la liste chaînée votes.

```
209 // Pour tester block_to_str
210 Block *b = (Block *)malloc(sizeof(Block));
211 generate_random_data(20, 2);
212 b->votes = read_protected("declarations.txt");
213 fraude(&(b->votes));
214 b->author = (Key *)malloc(sizeof(Key));
215 init_key(b->author, 551, 288);
216 compute_proof_work(b, 2);
217 char* block = block_to_str(b);
218 printf("\nBlock to str = %s\n", block);
219 printf("Verify block = %d\n", verify_block(b, 2));
220 // Il reste que les protected à désallouer
221 free(block);
222 free(b->author);
223 delete_block(b);
224
225
226 /*
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/string.h:77:43
int strcmp(const char *__s1, const char *__s2);

4 warnings generated.
gcc -o main main.o exo1.o exo2.o exo3.o exo4.o exo5.o exo6.o exo7.o exo8.o exo9.o
MacBook-Pro-de-Zhuang:Projet jm$ ./main

Block to str = (227,120) (null) 76 (5f3,19b7) (bdd,23d1) #412#8d4#1534#1534#7b
7f,1535) (bdd,23d1) #4d9#35#580#580#1090#e57#59e#580#acc#b11#(1aab,338f) (bdd,
a03#1a03#342d#3dcf#444e#1a03#1b63#4a08#(5d1,111b) (5d1,111b) #f6b#109e#e22#e5f
b2a#1302#(396b,4e23) (bdd,23d1) #40ef#4c9c#4153#4153#3163#24ec#31b7#4153#38b7#
#4452#7f6#2a91#2a91#156#1e7e#443#2a91#4a6c#1929#(fa1,1e97) (bdd,23d1) #1924#1
3#165#45c#(e3b,39d0) (5d1,111b) #3007#16a5#cbd#1dfa#264f#1dfa#1dfa#1dfa#2e82#c
3d1) #17b7#1def#ba1#ba1#d1d#15a4#190c#ba1#1024#942#(e3,114d) (bdd,23d1) #977#2
2#41d#(3871,7d6f) (5d1,111b) #3b5d#640#42ea#65f8#192#65f8#65f8#65f8#7021#3a8#

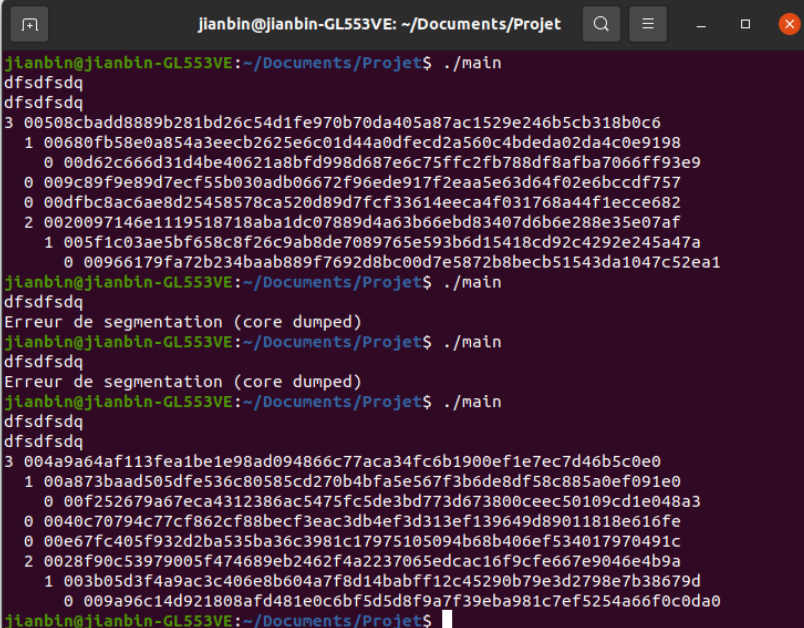
Verify block = 1
MacBook-Pro-de-Zhuang:Projet jm$
```

**Pour les fuites mémoires, on a perdu les pointeurs vers les protected quand on a fait delete\_block donc on ne peut plus les désallouer, personnellement je vois pas pourquoi on effacerait les CellProtected sans désallouer leur contenu, et j'en déduis que nos fuites mémoires proviennent de cela jusqu'à la fin de l'exo 9.**

On ne comprend pas pourquoi avec le même exécutable, des fois il y a des erreurs de segmentation et des fois tout marche sans problème.

Pour « éviter » ce problème on exécute plusieurs fois jusqu'à que ça marche, si ça ne marche pas de votre côté, vous pouvez faire de même (pour les exo8 et 9).

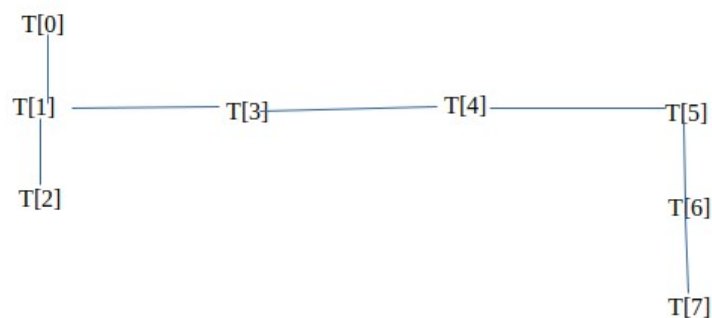
Je ne peux pas afficher le main pour l'exo 8 et 9 car il est beaucoup trop long, vous pouvez le faire de votre côté en décommentant.

A terminal window titled 'jlanbin@jlanbin-GL553VE: ~/Documents/Projet' with standard window controls. The terminal shows a sequence of commands and outputs. The first run of './main' produces a large block of hexadecimal data. The second run results in a segmentation fault error: 'Erreur de segmentation (core dumped)'. The third and fourth runs of './main' also result in the same error. The fifth run produces another large block of hexadecimal data. The prompt is always 'jlanbin@jlanbin-GL553VE:~/Documents/Projet\$'.



## Exercice-8 Structure arborescente

L'arbre qu'on a utilisé pour tester dans le main :



1) La fonction `CellTree* create_node(Block* b)` permet de créer et initialiser un noeud avec le block en paramètre et une hauteur égale à 0.

2) La fonction `int update_height(CellTree* father, CellTree* child)` permet de mettre à jour la hauteur de father quand il y a une modification d'un de ses fils. On modifie la hauteur du père lorsque la hauteur du fils portant la modification +1 est supérieure à la hauteur du père

3) La fonction `void add_child(CellTree* father, CellTree* child)` permet de ajouter un fils à un noeud et de mettre à jour de tous les ascendants. Tout d'abord on test si le père est Null ou pas si oui le remplace le père par le fils, sinon on distingue deux cas, le premier cas si le père n'as de firstchild alors on ajoute le child au firstchild du père , dans l'autre cas si le père possède au moins un firstchild on fait un ajoute en tête de child a firstchild du père.

Dans les deux cas on modifie `child→father` et le `child_>previous hash` pour qu'ils pointent sur le nouveau père. Et enfin on fait un boucle `while` pour changer la hauteur de tout les ascendants.

4) La fonction `void print_tree(CellTree* tree)` permet de afficher un arbre. On a crée une fonction `void print_node (CellTree* tree,int etage)` le paramètre pour gérer les espaces selon le niveau de l'arbre. Tout d'abord la fonction commence avec une boucle `while` qui permet gérer les espaces, puis on fait la récursion sur le `tree→firstchild` descendre le bas possible et puis on effectue la récursion avec `tree→nextBro` pour afficher tous ses `nextBro` jusqu'à `NULL`.

5) La fonction `void delete_node(CellTree* node)` permet de supprimer un noeud de l'arbre et tout ses descendants.

Tout d'abord on commence avec un test si le nœud en paramètre a un père, si oui on distingue deux cas le premier cas si le nœud qu'on veut supprimé est le `firstchild` de son père alors on effectue le changement du `father→firstchild=node→nextBro` dans le deuxième cas c'est pareil on parcourt la liste des fils sur père du nœud et le moment qu'on trouve le nœud, on modifie le pointeur pour faire sortir le nœud qu'on veut supprimer.

Après avoir régler les problème des pointeur portant sur le pointeur du `nextBro` du nœud, on va distinguer deux cas pour la suppression du nœud. D'abord si le nœud n'a pas de `firstchild` alors on le supprime avec la fonction `delete_block` et `free(noeud)`. S'il possède un fils alors on va d'abord faire une boucle qui permet de la récursion sur son `firstchild` et de parcourir tous ses `NextBro` puis on supprimer le nœud après le `while`.

La fonction `void delete_tree(CellTree* tree)` permet de supprimer l'arbre en entier. On fait une boucle `while` pour trouver la racine de l'arbre et on utilise la fonction `delete_node` sur la racine trouver.

6) La fonction `CellTree* highest_child(CellTree* cell)` permet de retourner le noeud fils avec la plus grande hauteur. On suppose qu'il existe une unique noeud fils avec la plus grande hauteur. Tout d'abord on va créer deux variable de type `celltree` `max` pour stocker le le nœud `max` et `cur` pour parcourir la liste des fils de nœud en paramètre. Si on trouve la hauteur d'un fils est plus grand que `max` on remplace `max` par le nœud. A la fin du boucle `while` on aura le nœud fils avec la plus grande hauteur.

7) La fonction `CellTree* last_node(CellTree* tree)` permet de retourner le nœud de la plus longue chaîne.

Dans cette fonction on utilise la fonction `highest_child` pour trouver le grand fils du nœud et on effectue une boucle `while` sur `highest_child(max)` qui permet de descendre dans l'arbre et de prendre a chaque fois le plus des fils. A la fin du boucle `while` on aurait bien le nœud de la plus longue chaîne.

8) La fonction `CellProtected* fusion_liste(CellProtected* l1, CellProtected* l2)` permet de fusionner deux listes.

L'idée était de parcourir la liste l1 jusqu'à la fin puis on met le pointeur du dernier élément sur la liste l2. La complexité est de  $O(\text{length}(l1))$ .

On peut modifier la structure en ajoutant un pointeur `previous` qui permet de pointer sur le dernier nœud de la liste l1. Et du coup on peut directement atteindre la fin de la liste l1 et on pointe le `next` sur l2 mais il faut changer le `previous` du premier élément de la liste l2 pour qu'il pointe sur le dernier de la liste l1. Et le dernier de la liste l2 pointe sur le premier de l1. Qui crée une liste chaînée circulaire.

Mais on a rencontré des problèmes de free qu'on utilise l'implémentation précédente, donc on a décidé de créer une nouvelle `CellProtected` res qui copie la liste chaînée de l1 et l2. La complexité de notre fonction est de  $\text{length}(l1) + \text{length}(l2)$ .

9) On a implémenté une fonction `CellProtected* longueur_max(CellTree* tree)` qui permet de renvoyer la liste de déclaration des blocks se situant dans la plus longue chaîne. On fait une boucle `while` avec la fonction `highest_child` pour se positionner sur la longue chaîne et aussi on utilise la fonction `fusion_liste` qui permet de fusionner la liste du nœud max courant avec une variable chaîne qui permet de stocker la fusion de toutes les listes de votes précédant.

## Exercice-9 Simulation du processus de vote

1) La fonction `void submit_vote(Protected* p)` permet d'ajouter le vote d'un citoyen à la fin du fichier «Pending\_votes.txt». On ouvre le fichier Pending\_votes.txt avec le mode `a` qui permet de écrire à la fin du fichier. Une écriture dans le fichier en utilisant `fprintf` avec la chaîne de caractère `protected_to_str(p)`

2) La fonction `void create_block(CellTree* tree, Key* author, int d)` permet de créer un block valide contenant les votes dans le fichier Pending\_votes.txt et de supprimer le fichier Pending\_votes.txt après avoir créé le bloc. Et écrit le bloc obtenu dans un fichier appelé Pending\_block.

Tout d'abord on fait un `malloc` pour le block qu'on veut supprimer. Puis on initialise la liste de votes du block en faisant une lecture du fichier Pending\_votes avec la fonction `read_protected` et . Puis on initialise le `previous_hash` du block avec la valeur hachée du dernier nœud (en utilisant `last_node(tree)`) et on initialise l'auteur avec le paramètre `author`. Et on applique la fonction `compute_proof_work` avec le block et `d` en paramètre pour initialiser la valeur hachée. Et enfin on supprime le fichier Pending\_votes.txt et on écrit le block dans le fichier Pending\_block avec la fonction (`écriture_fichier exo7`) et on ajoute le block au fils du `last_node`.

3) La fonction `void add_block(int d, char* name)` qui vérifie que le bloc représenté par le fichier Pending\_block est valide. Si oui, la fonction crée un fichier appelé `name` représentant le bloc puis l'ajoute dans le répertoire Blockchain. Et supprime le fichier Pending\_block à la fin.

4) La fonction `CellTree* read_tree()`

Tout d'abord on parcourt une fois le répertoire Blockchain pour compter le nombre de fichiers. On ajoute dans le `if` avec `strcom(dir->name, ".DS_Store")!=0` car on est sur mac. Puis on alloue un tableau `T` avec la taille le nombre de fichiers dans le répertoire et on reparcourt le répertoire et pour chaque fichier on lit le block à l'intérieur et on l'insère dans le tableau `T`.

Dans la deuxième partie de la fonction on parcourt le tableau `T` pour chaque `T[i]`

on chercher tous ses fils  $T[j]$  en utilisant la comparaison de `previous_hash` de  $T[j]$  avec le hash de  $T[i]$ . Si oui on effectue un ajout en tête avec la fonction `add_child`. Enfin on parcourt encore une fois le tableau  $T$  pour trouver la racine de l'arbre et le retourner.

5) La fonction `Key* compute_winner_BT(CellTree* tree, CellKey* candidates, Cellkey* voters, int sizeC, int sizeV)` permet d'extraire la liste des déclarations de vote en utilisant `fusion_liste` et de supprimer les déclarations de vote non valides avec la fonction `fraude` et enfin calcule le vainqueur de l'élection grâce à la fonction `compute_winner`.

6) La fonction `main` dans le fichier `le_main`.

7) L'utilisation d'une blockchain dans le cadre d'un processus de vote est une très bonne idée car cela réduit considérablement le risque d'une fraude comme ce système repose sur un très grand nombre de contributeurs indépendants. Cependant, le consensus consistant à faire confiance à la plus longue chaîne ne permet pas d'éviter toutes les fraudes car si une personne réussit à contrôler plus de 50% de la puissance de calculs de la blockchain, alors cette dernière pourra avoir un effet sur le système.