KHELLA André et DANQUIGNY Julie

Rapport Projet

Blockchain appliquée à un processus électoral

Sujet :

Dans ce projet nous allons considérer l'organisation d'un processus électoral par scrutin uninominal majoritaire à deux tours, comme en France. L'objectif est de proposer une piste de réflexion sur les protocoles et sur les structures de données à mettre en place pour permettre l'implémentation efficace du processus de désignation du vainqueur de l'élection, tout en garantissant l'intégrité, la sécurité et la transparence de l'élection.

Partie 1 : Développement d'outils cryptographiques

Nous allons développer des fonctions permettant de chiffrer un message de façon asymétrique, en faisant intervenir deux clés :

- Une clé publique : transmise à l'envoyeur pour chiffrer son message
- Une clé secrète (privée) : permet de déchiffrer les messages à la réception

Exercice 1 : Résolution du problème de primalité

Dans le fichier primalite.c, nous avons les fonctions suivantes :

Fonctions :	Explications :
int is_prime_naive(long p)	Renvoie 1 si p est un nombre premier, sinon 0, sachant que p est un nombre entier impair
long modpow_naive(long a, long m, long n)	Retourne la valeur de $a^b \ mod \ n$ de manière naïve
long modpow(long a, long m, long n)	Retourne la valeur de $a^b \ mod \ n$ en réalisant des élévations au carré
int witness(long a, long b, long d, long p)	Teste si a est un témoin de Miller pour p un entier donné
long rand_long(long low, long up)	Retourne un entier long généré aléatoirement compris entre low et up inclus
int is_prime_miller(long p, int k)	o Réalise le test de Miller-Rabin en générant k valeurs de a au hasard o Teste si chaque valeur de a est un témoin de Miller pour p o Retourne 0 dès qu'un témoin de Miller est trouvé (p n'est pas premier) o Retourne 1 si aucun témoin n'est trouvé (p est très probablement premier)
long random_prime_numer(int low_size, int up_size, int k)	Étant donnés : o Deux entiers low_size et up_size représentant respectivement la taille minimale et maximale du nombre premier à générer

	○ Un entier k représentant le nombre de tests de Miller à réaliser Retourne un nombre premier de taille comprise entre low_size et up_size

Q°1

La complexité de la fonction is_prime_naive est en O(p).

Q°2

Le plus grand nombre premier que nous arrivons à tester en moins de 2 millième de seconde (0,002 s) est 377 231. Pour trouver cela, nous avons comparé le temps d'exécution de la fonction is_prime_naive, pour 377 231 on est autour de 0.001115 secondes et pour 377257 on est autour de 0.0022 secondes.

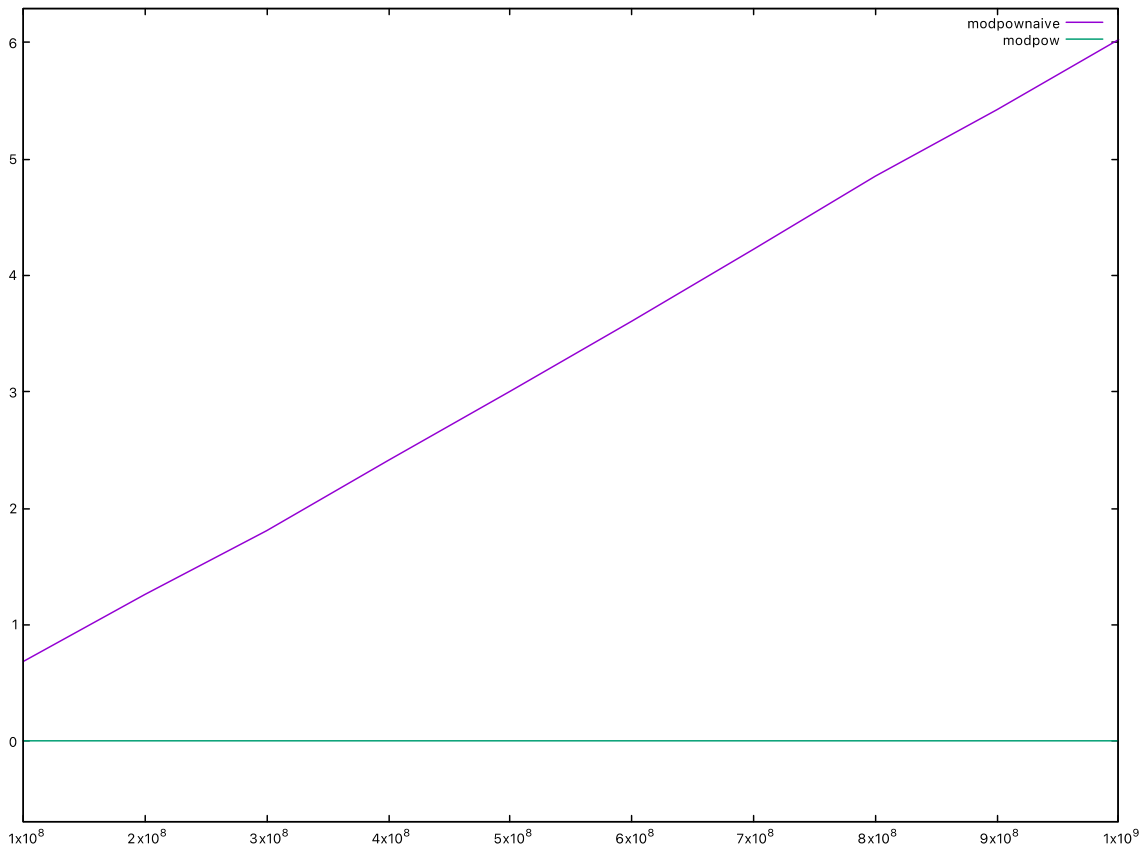
Q°3

La complexité de la fonction modpow_naive est en O(m), car on itère un nombre fini d'opérations m fois.

Q°4

La complexité de la fonction modpow est en $O(\log_2(m))$.

Q°5



On observe très clairement que plus m augmente, plus la fonction modpow_naive met du temps à s'exécuter, contrairement à modpow qui est tellement rapide que son temps d'exécution est de quasi 0 seconde. Voici les données numériques obtenues :

```
100000000 0.681667 0.000003
200000000 1.257597 0.000002
300000000 1.804900 0.000001
400000000 2.411834 0.000002
500000000 3.003333 0.000002
600000000 3.607022 0.000001
700000000 4.225000 0.000001
800000000 4.853035 0.000001
900000000 5.424466 0.000001
1000000000 6.026717 0.000001
```

La première colonne correspond à la valeur de m, la deuxième correspond au temps mis par la fonction modpow_naive, et la troisième au temps mis par la fonction modpow.

On voit alors que la différence de temps est très nette, et que la fonction modpow_naive met de plus en plus de temps à s'exécuter lorsque la valeur de m augmente, ce qui correspond bien avec leur complexité.

Q°7

On sait que pour tout entier p non premier quelconque, au moins $\frac{3}{4}$ des valeurs entre 2 et p-1 sont des témoins de Miller pour p. On sait également que sa complexité pire cas est en $O(k(\log_2(p))^3)$.
Cela signifie alors que la probabilité de tomber sur un nombre qui n'est pas un témoin de Miller pour p est de $\frac{1}{4}$, et comme on répète k fois l'opération, on obtient une probabilité d'erreur de $(\frac{1}{4})^k$.
On observe alors que plus k est grand, plus cette probabilité se rapproche de 0, et donc plus le test sera fiable.

$((\frac{1}{4})^k) \longrightarrow 0$ lorsque $k \longrightarrow \infty$

Exercice 2 : Implémentation du protocole RSA

Dans le fichier proto_rsa.c, nous avons les fonctions suivantes :

long extended_gcd(long s, long t, long* u, long* v)	Algorithme d'Euclide étendu
void generate_key_values(long p, long q, long* n, long* s, long* u)	Génère la clé publique pkey = (s,n) et la clé secrète skey = (u,n) à partir des nombres premiers p et q en suivant le protocole RSA
long* encrypt(char* chaine, long s, long n)	Chiffre la chaîne de caractère (le message) avec la clé publique pkey = (s,n)
char* decrypt(long* crypted, int size, long u, long n)	Déchiffre le message avec la clé secrète skey = (u,n) en connaissant la taille du tableau d'entiers
void print_long_vector(long* result, size_t size)	Affiche le résultat sous forme de vecteur

Partie 2 : Déclarations sécurisées

Dans cette partie, nous allons nous intéresser au problème de vote. Un citoyen interagit pendant les élections en effectuant des déclarations, l'ensemble des candidats est déjà connu, et les citoyens ont juste à soumettre des déclarations de vote.

Exercice 3 : Manipulations de structures sécurisées

Chaque citoyen possède une carte électorale définie par un couple de clé :

- Une clé secrète : il l'utilise pour signer la déclaration de vote
- Une clé publique : permet aux autres citoyens d'attester de l'authenticité de sa déclaration. Cette clé est aussi utilisée pour l'identifier dans une déclaration de vote, non seulement quand il vote, mais aussi quand quelqu'un souhaite voter en sa faveur.

Dans le fichier manip_struct.c, nous avons les fonctions suivantes :

void init_key(Key* key, long val, long n)	Initialise une clé déjà allouée
void init_pair_keys(Key* pkey, Key* sKey, long low_size, long up_size)	Utilise le protocole RSA pour initialiser une clé publique et une clé secrète (déjà allouées)
char* key_to_str(Key* key)	Permet de passer d'une variable de type Key à sa représentation sous forme de chaîne de caractères
Key* str_to_key(char* str)	Permet de passer d'une chaîne de caractères à sa représentation en une variable de type Key
Signature* init_signature(long* content, int size)	Alloue et remplit une signature avec un tableau de long déjà alloué et initialisé
Signature* sign(char* mess, Key* sKey)	Créer une signature à partir du message mess (déclaration de vote) et de la clé secrète de l'émetteur
char* signature_to_str(Signature* sgn)	Permet de passer d'une Signature à sa représentation sous forme de chaîne de caractères
Signature* str_to_signature(char* str)	Permet de passer d'une chaîne de caractères à sa représentation sous forme d'une Signature
Protected* init_protected(Key* pkey, char* mess, Signature* sgn)	Alloue et initialise la structure Protected
int verify(Protected* pr)	Vérifie que la signature contenue dans pr correspond bien au message et à la personne contenue dans pr
char* protected_to_str(Protected* p)	Permet de passer d'un Protected à sa représentation sous forme de chaîne de caractères
Protected* str_to_protected(char* str)	Permet de passer d'une chaîne de caractères à sa représentation sous forme de Protected
void free_signatures(Signature* sgn)	Supprime une signature

Q°1

La structure Key dans le fichier manip_struct.h :

```
typedef struct {
    long val;
    long n;
} Key;
```

Q°5

La structure Signature dans le fichier manip_struct.h :

```
typedef struct {
    long* tableau;
    int taille;
} Signature;
```

Q°9

La structure Protected dans le fichier manip_struct.h :

```
typedef struct{
    Key *pKey;
    Signature *signature;
    char *message;
} Protected;
```

Exercice 4 : Création de données pour simuler le processus de vote

Dans cette exercice, nous allons simuler le processus de vote à l'aide de trois fichiers :

- Un fichier contenant les clés de tous les citoyens
- Un fichier indiquant les candidats
- Un fichier contenant des déclarations signées

Dans le fichier simulation_ex4.c, nous avons les fonctions suivantes :

int recherche_element(int val, int* tab, int size)	Cherche si la valeur val est dans le tableau tab de taille size
void generate_random_data(int nv, int nc)	○ Génère nv couples de clés (publique, secrète) différents représentant les nouveaux citoyens ○ Crée un fichier keys.txt contenant tous ces couples de clés ○ Sélectionne nc clés publiques aléatoirement pour définir les nc candidats ○ Créer un fichier candidats.txt contenant la clé publique de tous les candidats ○ Génère une déclaration de vote signée pour chaque citoyen ○ Crée un fichier declarations.txt contenant toutes les déclarations signées

Partie 3 : Base de déclarations centralisée

Dans cette partie, nous considérons un système de vote décentralisé dans lequel toutes les déclarations de vote sont envoyées au système de vote, qui collecte les votes et annonce le vainqueur de l'élection aux citoyens.

Exercice 5 : Lecture et stockage des données dans des listes chaînées

On s'intéresse à la lecture et au stockage des données sous forme de listes simplement chaînées.

Dans le fichier stockage_dans_les_listes.c, nous avons les fonctions suivantes :

CellKey* create_cell_key(Key* key)	Alloue et initialise une cellule de liste chaînée
void ajout_cle_en_tete(CellKey** LCK, Key* key)	Ajoute une clé en tête de liste
CellKey* read_public_keys(char* nom_fichier)	Retourne une liste chaînée contenant toutes les clés publiques d'un fichier
void print_list_keys(CellKey* LCK)	Affiche une liste chaînée de clés
void delete_cell_key(CellKey* c)	Supprime une cellule de liste chaînée de clés
void delete_list_keys(CellKey* LCK)	Supprime une liste chaînée de clés
CellProtected* create_cell_protected(Protected* pr)	Alloue et initialise une cellule de liste chaînée
CellProtected* insert_cell_protected(CellProtected** list, Protected* pr)	Ajoute une déclaration signée en tête de liste
CellProtected* read_protected(char* filename)	Lit le fichier declarations.txt et crée une liste contenant toutes les déclarations signées du fichier
void print_list_protected_keys(CelleProtected* LPK)	Affiche la liste des déclarations signées
void delete_cell_protected(CellProtected* c)	Supprime une cellule de liste chaînée de déclarations signées
void delete_list_protected(CellProtected* LPK)	Supprime la liste chaînée de déclarations signées
void fusion_list_protected(CellProtected** list1, CellProtected** list2)	Fusionne deux listes et supprime list2

Exercice 6 : Détermination du gagnant de l'élection

Dans le fichier gagnant.c, nous avons les fonctions suivantes :

void delete_fake_protected(CellProtected** LCK)	Supprime toutes les déclarations dont les signatures ne sont pas valides dans une liste chaînée
HashCell* create_hashcell(Key* key)	Alloue une cellule de la table de hachage et initialise ses champs en mettant la valeur à 0
int hash_function(Key* key, int size)	Retourne la position d'un élément dans la table de hachage
int find_position(HashTable* t, Key* key)	Cherche dans la table s'il existe un élément dont la clé publique est key Collision → probing linéaire Si l'élément est trouvé, on retourne sa position, sinon la position où il aurait dû être

HashTable* create_hashtable(CellKey* keys, int size)	Crée et initialise une table de hachage de taille size contenant une cellule pour chaque clé de la liste chaînée keys
void delete_hash_cellule(HashCell* hashcell)	Supprime une cellule de la table de hachage
void delete_hash_table(HashTable* t)	Supprime la table de hachage
Key* computeWinner(CellProtected* decl, CellKey* candidates, CellKey* voters, size_t sizeC, size_t sizeV)	Calcule le vainqueur de l'élection étant donné une liste de déclarations avec signatures valides, une liste de candidats, et une liste de personnes autorisées à voter La fonction réalise les actions suivantes : o Crée deux tables de hachage (une pour les candidats, une pour les votants) o Parcourt la liste de déclarations et vérifie que la personne a le droit et n'a pas déjà voté, et que la personne sur qui porte le vote est bien candidat o Conditions vérifiées → vote comptabilisé dans la table de hachage des candidats et la table de hachage des votants est mise à jour pour indiquer qu'il a bien voté o Détermine le gagnant en utilisant la table de hachage des candidats
void afficher_tableH(HashTable* t)	Affiche la table de hachage t

Partie 4 : Blocs et persistances des données

Dans la partie 4, nous allons utiliser une blockchain, une base de données décentralisée et sécurisée dans laquelle les citoyens possèderont une copie de la base contenant les déclarations de vote signées.

Le protocole de vote est le suivant :

- Vote
- Création de blocs
- Mise à jour des données

Pour rendre la fraude difficile, on va utiliser un mécanisme de consensus dit par *proof of work* fondée sur une fonction de hachage cryptographique (facile à calculer mais difficile à inverser)

Exercice 7 : Structure d'un block et persistance

Dans cet exercice on va s'intéresser à la gestion des blocs n utilisant la structure Block, afin qu'un bloc contienne :

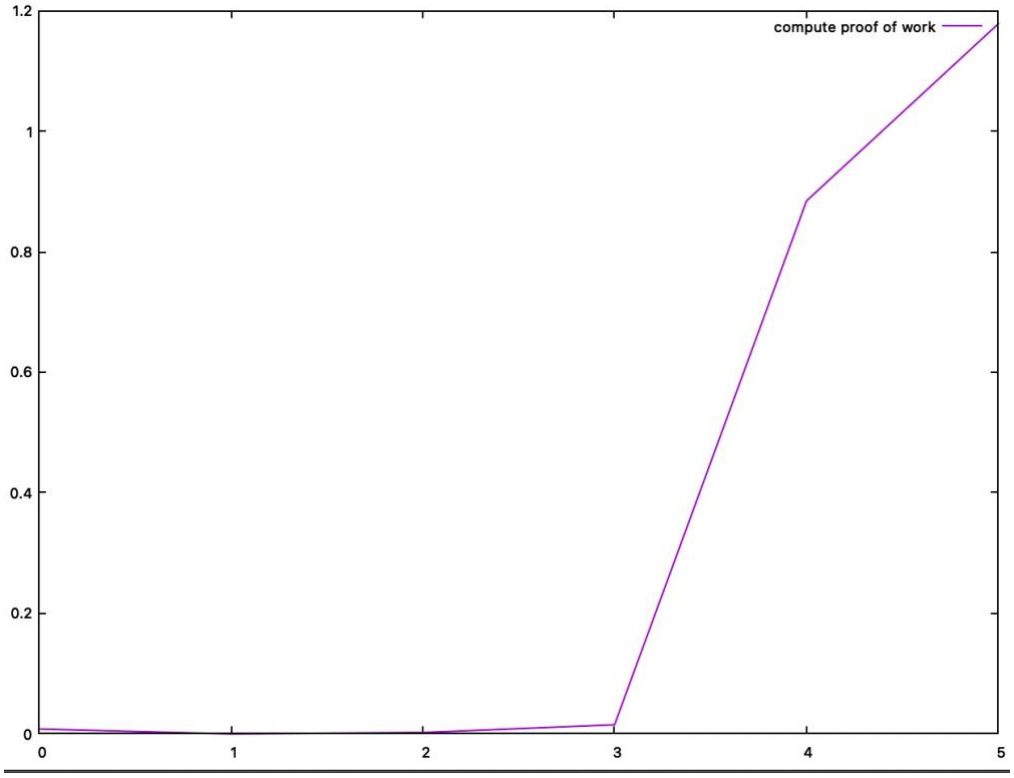
- La clé publique de son créateur
- Une liste de déclarations de vote
- La valeur hachée du bloc
- La valeur hachée du bloc précédent
- Une preuve de travail

Dans le fichier structure_du_block.c, nous avons les fonctions suivantes :

void write_fichier(char* filename, Block* block)	Permet d'écrire un bloc dans un fichier
Block* read_block(char* filename)	Permet de lire un bloc à partir d'un fichier
char* block_to_str(Block* block)	Génère une chaîne de caractères représentant un bloc, qui contient : o La clé de l'auteur o La valeur hachée du bloc précédent o Une représentation de ses votes o La preuve de travail
unsigned char* decrypt_sha(const char* chaine)	Retourne la valeur hachée obtenue par l'algorithme SHA256 de la chaîne de caractères
void compute_proof_of_work(Block* b, int d)	Rend un bloc valide en commençant avec l'attribut nonce à 0, puis en l'incrémentant jusqu'à ce que la valeur hachée du bloc commence par d 0 successifs
int verify_block(Block* b, int d)	Vérifie qu'un bloc est valide
void delete_block(Block* b)	Supprime un bloc mais ne libère pas la mémoire associée au champs author. Pour la liste chaînée de votes, on libère les éléments de la liste chaînée mais pas leur contenu

Q°8

En comparant le temps moyen de la fonction compute_proof_of_work selon la valeur de d, on obtient la courbe suivante :



Courbe représentant le temps moyen de la fonction compute_proof_of_work en fonction de d.

On en déduit alors que la valeur de d à partir de laquelle ce temps dépasse une seconde, est de 4, or on sait que le nombre de 0 d exigés sur la représentation hexadécimale revient à exiger que la représentation binaire débute par au moins 4d 0, ce qui correspond à notre graphique.

Exercice 8 : Structure arborescente

En cas de triche on peut se retrouver avec plusieurs blocs indiquant le même bloc précédent ce qui conduit à une structure arborescente. Pour cela, on va faire confiance à la chaîne la plus longue pour retomber sur une chaîne de blocs.

Dans le fichier structure_arbre_arborescente.c, nous avons les fonctions suivantes :

CellTree* create_node(Block* b)	Crée et initialise un nœud avec une hauteur égale à 0
int update_height(CellTree* father, CellTree* child)	Met à jour la hauteur du nœud father quand l'un de ses fils a été modifié. Retourne 1 si la hauteur du nœud father a changé, sinon 0
void add_child(CellTree* father, CellTree* child)	Ajoute un fils à un nœud en mettant à jour la hauteur de tous les ascendants
void print_tree(CellTree* racine)	Affiche un arbre : pour chaque nœud il faut afficher la hauteur du nœud et la valeur hachée du bloc coreespondant
void delete_node(CellTree* node)	Supprime un noeud de l'arbre en faisant appel àa la fonction delete_block (Exercice 7 Q°9)
void delete_tree(CellTree* tree)	Supprime l'arbre
CellTree* highest_child(CellTree* cell)	Renvoie le noeud fils avec la plus grande hauteur
CellTree* last_node(CellTree* tree)	Retourne la valeur hachée du dernier bloc de cette plus longue chaîne
void fusion_cell_protected(CellProtected* first, CellProtected* second)	Fusionne deux listes chaînées de déclarations signées
CellProtected* fusion_highest_CP(CellTree* racine)	Retourne la liste obtenue par fusion des listes chaînées de déclarations contenues dans les blocs de la plus longue chaîne, en utilisant (la fonction du dessus) et highest_child

Q°8

La complexité de la fonction fusion_cell_protected est en O(n) car si on a n la taille de la première liste et m la taille de la deuxième liste, la fonction fusion_cell_protected va parcourir la première liste puis chainer son dernier élément avec le premier élément de la deuxième liste.
Pour avoir une fusion en O(1), il faudrait modifier la structure CellProtected et ajouter un pointeur vers le dernier élément de la liste.

Exercice 9 : Simulation du processus de vote

On va simuler le fonctionnement d'une blockchain en utilisant le répertoire et les fichiers suivants :

- Blockchain
- Pending_block
- Pending_votes.txt

Dans le fichier simulation_process-vote.c, nous avons les fonctions suivantes :

void submit_vote(Protected* p)	Permet à un citoyen de soumettre son vote (de l'ajouter au fichier « Pending_votes.txt ») Si le fichier n'existe pas, on le crée
void create_block(CellTree* tree, Key* author, int d)	○ Crée un bloc valide contenant les votes en attente dans le fichier « Pending_votes.txt » ○ Supprime le fichier « Pending_votes.txt » après avoir créé le bloc ○ Écrit le bloc obtenu dans un fichier appelé « Pending_block » Utilise last_node, read_protected, compute_proof_of_work et write_block
void add_block(int d, char* name)	Vérifie que le bloc représenté par le fichier « Pending_block » est valide Si c'est le cas, elle crée un fichier appelé name représentant le bloc, l'ajoute dans le répertoire « Blockchain » « Pending_block » est supprimé
CellTree* read_tree()	○ Crée un nœud de l'arbre pour chaque bloc contenu dans le répertoire et stocke tous les nœuds dans un tableau T de type CellTree** ○ Parcourt le tableau T de nœuds, et pour chaque nœud recherche tous ses fils, chaque fils est ajouté à la liste des fils du nœud (avec add_child) ○ Parcourt le tableau T pour trouver la racine de l'arbre et la retourne
Key* compute_winnerBT(CellTree* tree, CellKey* candidates, CellKey* voters, int sizeC, int sizeV)	Détermine le gagnant de l'élection en se basant sur la plus longue chaîne de l'arbre

Dans le fichier main.c, on a testé nos différentes fonctions tout le long du projet, et la fonction main réalise ces différentes tâches :

int main()	○ Génère un problème de vote avec 1 000 citoyens et 5 candidats ○ Lit les déclarations de vote des candidats et des citoyens ○ Soumet tous les votes avec la création d'un bloc valide tous les 10 votes soumis, suivi par l'ajout du bloc dans la blockchain ○ Lit et affiche l'arbre final ○ Calcul et affiche le gagnant

Q°7 : Conclusion du Projet

L'utilisation de blockchain dans le cadre d'un processus de vote montre ses avantages principalement dans la transparence, la sécurité, et l'anonymat lors d'une élection. D'autres avantages sont aussi à noter comme par exemple la décentralisation du système de vote, ou encore le fait de permettre de voter en ligne, car comme nous avons pu le voir avec les élections présidentielles 2022, le taux d'abstention est assez élevé (26,31% au premier tour selon https://www.resultats-elections.interieur.gouv.fr/presidentielle-2022/FE.html)

Malgré cela, il y a pas mal d'inconvénients, comme par exemple le volume des calculs, qui nécessitent tout de même des machines assez puissantes pour réaliser cela.

Le consensus consistant à faire confiance à la plus longue chaîne ne permet pas d'éviter toutes les fraudes car on ne sait pas si d'autres personnes peuvent trouver un moyen de créer une chaine plus longue, et donc de modifier les résultats de l'élection.

Cependant nous avons trouvé ce projet un peu long et répétitif, même s'il nous a permis d'utiliser tous les types de structures de données vues en cours. Il reste tout de même intéressant et d'actualité, donc cela a permis de le rendre plus agréable.