Prologin 2021 - Epreuve écrite (21/02/2021)

1) L'énoncé nous précise que les transactions sont inscrites dans l'ordre chronologique, donc il faudrait une structure de données dans laquelle l'ordre importe. Ainsi, nous pourrions choisir un tableau dynamique contenant des triplets (émetteur, receveur, montant)

2) En Python :

```
def addTransaction(registre, émetteur, receveur, montant):
    registre.append((émetteur, receveur, montant))
```

3)

```
def nbCrepes(registre, user):
    n = 0
    for emet, recev, amount in registre:
        if emet == user:
            n -= amount
        elif recev == user:
            n += amount
    return n
```

Remarque : Il est toutefois possible d'implémenter un petit détecteur de fraude en rajoutant ligne 4 :
`assert emet != recev, "Fraude detectée"`

4)

```
def crepesFournies(registre):
    n = 0
    for emet, _, amount in registre:
        if emet == "bureau":
            n += amount
    return n
```

5) def nouvelle_transaction(emet, recev, montant, clefS, reg):
    transaction = Signe(emet + recev + str(montant), clefS)
    reg.append(emet, recev, montant, transaction)

Remarque : On suppose ici que le registre est un tableau contenant les signatures en plus dans chaque triplet (émetteur, receveur, montant).

```
def verifie_registre(clefsP, registre):
    i = 0
    tailleRegistre = len(registre)
    while i < tailleRegistre:
        emet, recev, amt, signature = registre[i]
        if verifie((emet, recev, amt), signature):
            i += 1
        else:
            del registre[i]
            tailleRegistre -= 1
```

6) On veut des identifiants uniques car sinon la fonction vérifie ne pourrait fonctionner (en effet, s'il existe plusieurs transactions qui donnent un même identifiant, alors vérifie ne pourrait déterminer si une signature provient d'une certaine transaction).
Je ne suis pas sûr, mais cela veut dire que la fonction signe doit être bijective.

⇒ Il faut signer la transaction puis ajouter l'identifiant car la signature prend seulement la transaction en argument et ne dépend donc pas de l'identifiant.

8) Soit E l'événement : " il y a au moins une collision"

$P(E) = 1 - P(\bar{E})$

avec $\bar{E}$: "il n'y a aucune collision"

S'il n'y a aucune collision :

$A_N^m = \frac{N!}{(N-m)!}$.

Or il y a N^m tirages possibles

d'où $p(\bar{E}) = \frac{A_N^m}{N^m}$

$P(E) = 1 - \frac{A_N^m}{N^m}$

3) On cherche m tel que :

$P(E) \geq \frac{1}{2}$ et $N = 10^6$

$\Leftrightarrow 1 - \frac{A_{10^6}^m}{(10^6)^m} \geq \frac{1}{2}$

$\Leftrightarrow \frac{10^6!}{(10^6-m)!} \leq \frac{1}{2} \times 10^{6m}$

~~$\Leftrightarrow 2 \times 10^6! \leq (10^6-m)!$~~

Je ne saurais approximer une valeur plus généralement :

$P(E) \geq \frac{1}{2}$

$\Leftrightarrow 1 - \frac{\frac{N!}{(N-m)!}}{N^m} \geq \frac{1}{2}$

$\Leftrightarrow \frac{N!}{(N-m)!} \times \frac{1}{N^m} \leq \frac{1}{2}$

$\Leftrightarrow \frac{N!}{(N-m)!} \leq \frac{1}{2} N^m$

10) def existeDoublons(registre):
    visited = set()
    for _*, identifiant in registre:
        if identifiant in visited:
            return True
        else:
            visited.add(identifiant)
    return False

11) Généralement : O(n) car "in" s'effectue en O(1) pour les sets (table de hachage)
Pire cas : $O(n^2)$ à cause des collisions.

12) Nous pourrions utiliser un tableau et pour chaque noeud à l'indice i, ses deux fils sont à l'indice 2i (celui qui est inférieur) et 2i + 1 (celui qui est supérieur)
NB : le tableau commence à l'indice 1.

13) def isIn(ABR, x):
    i = 1
    if x > ABR[i]:
        i = 2*i + 1

def isIn(ABR, x, i = 1):
    if ABR[i] == x:
        return True
    elif ABR[i] is None:
        return False
    elif x > ABR[i]:
        return isIn(ABR, x, 2*i+1)
    else:
        return isIn(ABR, x, 2*i)

14) def addElem (ABR, x):

```
    if isIn (ABR, x):
        return False  # erreur
    else:
        i = 1
        while ABR[i] is not None:
            if x > ABR[i]:
                i = 2*i + 1
            else:
                i = 2*i
        ABR[i] = x
        ABR[2*i] = None
        ABR[2*i + 1] = None
```

15) Manque de temps

16) On modifie isIn telle que la fonction renvoie l'indice

```
def remove (ABR, x):
    _, i = isIn (ABR, x)
    ABR[i] = None
```

Soit n la taille du tableau contenant les noeuds de l'arbre:

17) 13 : $O(\log(n))$ ⇒ recherche dichotomique

14 : $O(\log(n))$ (similaire à isIn)

16 : $O(\log(n))$ car elle utilise isIn

18) Oui, un tableau trié (recherche dichotomique en $\log n$)

19)

```
def isNegative (registre):
    for emet, recev, qte, _ * in registre :
        qteCrepes[emet] -= qte
        qteCrepes[recev] += qte
```

```
        if qteCrepes[emet] < 0 and emet != "bureau":
            return True # on retourne "Erreur" ici pour la question 20
    return False
```

Remarque : On suppose ici qu'il existe un dictionnaire associant à chaque personne le nombre de crêpes qu'il/elle détient.

2) On modifie isNegative en rajoutant l'initialisation de qteCrepes : qteCrepes = {personne: 0 for personne in (orga)} avec orga la liste des organisateurs
et on retourne le dictionnaire qteCrepes à la fin de la fonction.

11) ~~def repartir (qteCrepes, orga):~~
~~transferts = [("bureau", orga[0]~~
~~aRepartir = abs (qteCrepes["bureau"])~~

```
def repartir (qteCrepes, orga):
    aRepartir = abs (qteCrepes["bureau"])
    transferts : [("bureau", orga[0], aRepartir)]
    # aRepartir -= qteCrepes[orga[0]]
    for org in orga[1:]:
        transferts.append((orga[0], org, qteCrepes[org]))
    return transferts
```

Remarque : L'idée est qu'un orga reçoive tout et répartisse tout (désolé pour lui)
J'ai mis aRepartir en commentaire car on ne se préoccupe pas de combien de crêpes il reste à distribuer tant que le bureau les fournit.

22) Première idée (bon c'est un peu jouer sur les mots):

Puisque l'énoncé nous dit que les crêpes seront données à "certains organisateurs en particulier", le plus optimal serait que le bureau distribue à tous les orgas le nombre de crêpes dont ils ont besoin (donc le nombre de transferts entre orgas est de 0).

Autre idée: L'orga donne toutes les crêpes à une personne, qui prend ce dont elle a besoin et passe le reste au prochain orga (1 transfert par orga).

- on implémente l'autre idée:

```
def repartir(qteCrepes, orgas):
    aRep = abs(qteCrepes["bureau"])
    transferts = [("bureau", orgas[0], aRep)]
    aRep -= qteCrepes[orgas[0]]
    for i in range(1, len(orgas)):
        transferts.append((orgas[i-1], orgas[i], aRep))
        aRep -= qteCrepes[orgas[i]]
    return transferts
```

- complexité : $O(n)$

23) Puisque la fonction associe à une donnée un entier codé sur k bits et puisque peu importe la taille de la donnée, alors chacun pourra rentrer chez lui avec l'addition de tous les transferts du registre codée sur un seul entier. S'il est modifié entre temps, l'entier issu de la fonction de hachage sera différent.

24) 1. Oui, car la somme est difficile à décomposer de manière à retrouver les bons caractères

2. Non, il est simple de reconstruire une somme semblable (?)

(sauf si l'on ne connait pas k).

3. Non, à cause du modulo

Par exemple ab $\to$ 21

$21 \equiv 1 [2]$ en supposant $k = 1$.

mais 01 $\to$ 1

$1 \equiv 1 [2]$

25) Tableau commençant à l'indice i tel que les 2 fils de i soient 2i et 2i+1

26)
```
def Merkle (registre):
    if len (registre) == 1:
        return H (registre [0])
    else:
        n = len (registre) // 2
        return H ( Merkle (registre [:n]) + Merkle (registre [n:]))
```

Calculer le ~~résultat~~ de top Hash

27)
```
def Merkle (registre):
    arbre = [None] * (len (registre) + 1)
```

J'ai trop pris mon temps au début, je n'ai pu terminer

Desolé ! :)

PS : Le sujet était trop bien.