EXERCICES DE COURS : TABLES ET FONCTIONS DE HACHAGE

EXERCICES DU CHAPITRE I : INTRODUCTION

Exercice I.1 – Est-ce qu'une table de hachage est adaptée ?

Un magasin veut stocker pour chaque code d'article (par ex 10542, 75601, etc.) la quantité en stock (3, 5, 12,…)

Proposer une manière de stocker ces informations avec :

a) une liste Python,
b) un dictionnaire Python.

Pour chaque solution, indiquer :

- comment on récupère la quantité du code 75601,
- combien d'opérations cela prend « en gros » si la liste contient 10 000 articles.

Laquelle des deux structures semble la plus adaptée ? Pourquoi ?

Exercice I.2 – Clé ou valeur ?

On veut concevoir une structure de données pour chaque situation. Pour chaque cas, proposer un choix de clé.

1. Gestion des employés d'une entreprise : on dispose du nom, prénom, numéro de sécu, date d'embauche.
2. Catalogue d'un site de vente en ligne : on a un identifiant numérique, le nom du produit, et le prix.
3. Agenda d'événements : chaque événement a une date/heure, un titre, un lieu.

Pour chaque scénario, préciser pour le choix de clé :

- si la clé est « naturellement unique » ;
- si elle est facilement hashable (nombre, chaîne raisonnable) ;
- si elle risque de changer (mauvais choix de clé).

Quizz 1 – Notion de clé & dictionnaire

1. Dans une table de hachage, une clé sert principalement à :
    a. Stocker les données elles-mêmes
    b. Indiquer la taille de la table
    c. Identifier de manière (presque) unique une valeur
    d. Trier les valeurs par ordre croissant
2. Dans un dictionnaire Python :

```
d = {}
d["Alice"] = 1
d["Alice"] = 2
```

    a. On obtient une erreur
    b. Le dictionnaire contient deux entrées pour "Alice"
    c. La valeur associée à "Alice" est 2
    d. La valeur associée à "Alice" est 1

3. Laquelle de ces propositions décrit le mieux l'intérêt d'une table de hachage ?
    a. Trier des données très rapidement
    b. Rechercher, insérer, supprimer par clé en temps moyen constant
    c. Compresser des données
    d. Représenter des données en forme d'arbre
4. Pour stocker des numéros de sécurité sociale associés à des fiches, la meilleure structure est :
    a. Une liste triée de fiches
    b. Une pile
    c. Une table de hachage indexée par le numéro de sécu
    d. Une file FIFO
5. (Vrai/Faux) Une même valeur peut apparaître plusieurs fois dans un dictionnaire Python.
6. (Vrai/Faux) Une même clé peut apparaître plusieurs fois dans un dictionnaire Python.

Exercice Python 1 – Déduplication avec set / dict

On reçoit une liste d'adresses IP (sous forme de chaînes). Compléter la fonction pour renvoyer la liste des IP distinctes, dans l'ordre d'apparition.

```
def ip_uniques(ips):
    vues = set()
    resultat = []
    for ip in ips:
        # TODO : si ip n'a pas encore été vue,
        #        l'ajouter à "vues" et à "resultat"
        if ____________________________:
            ____________________________
            ____________________________
    return resultat
```

Test possible :

```
ips = ["A", "B", "A", "C", "B", "D"]
print(ip_uniques(ips))  # attendu : ["A", "B", "C", "D"]
```

Exercice Python 2 – Compter les occurrences de mots

Compléter cette fonction qui compte le nombre d'occurrences de chaque mot dans une liste.

```
def compter_mots(mots):
    d = {}
    for mot in mots:
        # TODO : utiliser d[mot] pour compter les occurrences
        if mot in d:
            ________________________
        else:
            ________________________
    return d
```

Test possible :

```
mots = ["un", "deux", "un", "trois", "deux", "un"]
print(compter_mots(mots))  # {"un": 3, "deux": 2, "trois": 1}
```

EXERCICES DU CHAPITRE II : OPÉRATIONS PRISES EN CHARGE PAR LES TABLES DE HACHAGE

Exercice II.1 – Identifier les opérations

On considère une table de hachage qui mémorise les clients d'un site avec leur identifiant client.
Pour chacune des actions suivantes, indiquer si c'est une recherche, une insertion ou une suppression.

1. Vérifier si le client 1234 existe déjà dans la base.
2. Ajouter un nouveau client 5678.
3. Supprimer le compte du client 1234.
4. Mettre à jour l'adresse e-mail du client 5678.
   Question bonus : en termes d'opérations de base sur la table de hachage, que se cache-t-il derrière cette mise à jour ?

Exercice II.2 – Tableau impossible

1. On suppose que l'on indexe un tableau par toutes les chaînes de longueur exactement 5 sur l'alphabet A–Z.
   a. Combien de cases comporte ce tableau ?
   b. Pourquoi est-ce irréaliste en pratique (mémoire) ?
2. On suppose que l'on n'utilise finalement que 1000 prénoms distincts parmi toutes ces chaînes possibles.
   a. Combien de cases du tableau sont effectivement utilisées ?
   b. Quelle proportion de cases est vide ?
3. Expliquer en quelques lignes pourquoi une table de hachage qui utilise en gros une case par élément stocké est plus intéressante que ce tableau « gigantesque mais presque vide ».

EXERCICES DU CHAPITRE III : EXEMPLES D'APPLICATIONS

Exercice III.1 – Simuler une déduplication

On reçoit un flux d'adresses IP (simplifiées) : [A, B, C, A, D, B, E, C, F]
On veut ne garder qu'une occurrence de chaque adresse, en utilisant, pour chaque nouvelle IP x :

- si x est déjà présente dans la table de hachage, on l'ignore ;
- sinon, on l'insère.

1. À la fin, quelles sont les IP distinctes observées ?
2. Si on voulait compter le nombre de visites par IP, comment adapter l'algorithme (en quelques phrases) ?

Exercice III.2 – Pourquoi mémoriser les sommets visités ?

On considère un graphe simple (avec au moins un cycle).

1. Expliquer ce qui se passe si, dans un parcours en largeur (BFS), on ne mémorise pas les sommets déjà visités.
2. On décide d'utiliser une table de hachage `visited`.
   a. Que stocke-t-on comme clé dans cette table ?
   b. Quand ajoute-t-on un sommet dans `visited` ?

Exercice Python 3 – BFS avec ensemble visited

On représente un graphe non orienté par un dictionnaire adj : chaque sommet a la liste de ses voisins. Compléter un BFS qui renvoie les sommets atteignables depuis une source s.

```python
from collections import deque

adj = {
    0: [1, 2],
    1: [0, 3],
    2: [0],
    3: [1],
}

def bfs(adj, s):
    visites = []
    file = deque()
    resultat = []

    # TODO : initialisation
    ____________________________
    ____________________________

    while file:
        u = file.popleft()
        resultat.append(u)
        for v in adj[u]:
            # TODO : si v n'a pas été visité,
            # le marquer et l'ajouter à la file
            if ____________________________:
                ____________________________
                ____________________________

    return resultat
```

EXERCICES DU CHAPITRE IV : IMPLÉMENTATION - IDÉES GÉNÉRALES

Exercice IV.1 – Pourquoi la liste n'est pas suffisante

On stocke 10 000 adresses IP ayant visité un site dans une liste Python.

1. Pour vérifier si une IP donnée ip0 est dans la liste, expliquer ce que fait Python dans le pire cas (combien d'éléments examinés ?).
2. On remplace la liste par une table de hachage.
    a. Quelle information supplémentaire utilise-t-on pour éviter de balayer toute la structure ?
    b. En quelques mots, pourquoi cela permet (en général) d'avoir un temps proche de constant ?

Exercice IV.2 – Calculer des indices de hachage

On suppose que les clés sont des nombres entiers et qu'on utilise une table de taille n = 10 avec :

$$h(k) = k \bmod 10$$

1. Calculer h(k) pour k = 7, 12, 25, 30, 44.
2. On insère ces clés dans la table (on ignore pour l'instant la gestion des collisions). Indiquer dans quelles cases elles vont.
3. Est-ce que cette fonction de hachage semble répartir raisonnablement ces valeurs sur {0,…,9} ?

Exercice IV.3 – Collisions inévitables

1. Expliquer ce que dit le principe des tiroirs et pourquoi il garantit qu'avec plus de clés que de cases, il y aura au moins une collision.
2. On rappelle que 23 personnes suffisent pour avoir ≈ 50 % de chances qu'au moins deux partagent le même jour d'anniversaire. En quoi cet exemple montre que les collisions arrivent bien plus tôt qu'on ne le pense intuitivement ?
3. Pourquoi ces idées justifient qu'on cesse d'espérer une table « sans collisions » dès qu'elle commence à se remplir ?

Exercice IV.4 – Simuler une table avec chaînage

Table avec 5 seaux (indices 0 à 4).
Fonction de hachage : $h(k)=k \bmod 5$.
Collisions gérées par chaînage (liste dans chaque seau).

On insère successivement les clés : 7, 12, 17, 22, 3, 8.

1. Pour chaque clé, calculer h(k) et indiquer dans quel seau elle est ajoutée.
2. Donner l'état final de la table, sous la forme :
    a. 0 : […]
    b. 1 : […]
    c. …
3. Combien d'éléments contient, en moyenne, un seau ?
4. Si l'on recherche la clé 22, quels éléments sont inspectés, et dans quel ordre ?

Exercice IV.5 – Sondage linéaire à la main

Table de taille 7 (indices 0…6). Fonction de hachage : $h(k)=k \bmod 7$.
On utilise le sondage linéaire : indices testés h(k), h(k)+1, h(k)+2, … (mod 7).
On insère : 10, 24, 31, 45, 18.

1. Pour chaque clé, indiquer :
    a. h(k) initial,
    b. la suite des indices sondés,
    c. l'indice final d'insertion.
2. Donner l'état final de la table.
3. Pour rechercher 18, indiquer quels indices sont sondés (et pourquoi on s'arrête).
4. Pour rechercher 11, quels indices seraient sondés, et à quel moment conclure qu'il n'est pas dans la table ?

Exercice IV.6 – Bonne ou mauvaise fonction de hachage ?

On suppose une table de taille n = 1000. Pour chaque fonction, dire si, intuitivement, elle est « correcte » ou dangereuse pour les données décrites.

1. Cas 1 :
    - $h(k)=0$
    - Données : n'importe quelles clés entières.
2. Cas 2 :
    - $h(k)=k \bmod 1000$
    - Données : les salaires de l'entreprise, multiples de 1000 (23 000, 45 000, …).
3. Cas 3 :
    - $h(k)=(a \cdot k+b) \bmod 1000$ avec a,b fixés.
    - Données : salaires multiples de 1000 + d'autres nombres variés.
4. Cas 4 :
    - h(chaine) = hachage polynomial + modulo n, avec n choisi premier, loin d'une puissance de 2 ou de 10.
    - Données : mots de passe ou chaînes très variées.
5. Pour chaque cas :
    - Où risque-t-on des collisions massives ?
    - Quels cas semblent plus « robustes » ?

Quizz 2 – Fonction de hachage & collisions

1. Une fonction de hachage $h : U \rightarrow \{0, \ldots, n-1\}$ doit être :
    a. Lente et très compliquée
    b. Déterministe et rapide
    c. Aléatoire à chaque appel (pour une même clé)
    d. Obligatoirement injective
2. Une collision survient quand :
    a. Deux clés différentes ont des valeurs différentes
    b. Deux clés différentes ont la même valeur
    c. Deux clés différentes sont envoyées dans le même compartiment
    d. La table est pleine
3. Le principe des tiroirs dit que s'il y a :
    a. Moins de clés que de compartiments, il y a forcément une collision
    b. Plus de clés que de compartiments, il y a forcément une collision
    c. Autant de clés que de compartiments, il n'y a jamais de collision
    d. Au moins deux compartiments vides
4. Parmi ces fonctions, laquelle est manifestement une très mauvaise fonction de hachage ?
    a. $h(k) = k \bmod n$
    b. $h(k) = 0$
    c. $h(k) = (3k + 5) \bmod n$
    d. $h(k) = (k^2 + 1) \bmod n$
5. (Vrai/Faux) On peut concevoir une fonction de hachage qui n'a jamais de collision, quel que soit le jeu de données, si on choisit bien n.

Quizz 3 – Chaînage & adressage ouvert

1. Avec le chaînage, chaque compartiment de la table contient :
   a. Une seule clé
   b. Une liste de toutes les clés qui y tombent
   c. Un pointeur vers une autre table de hachage
   d. Une valeur booléenne
2. Avec l'adressage ouvert (sondage linéaire), lorsqu'une case est occupée :
   a. On remplace son contenu
   b. On redimensionne immédiatement la table
   c. On cherche une autre case selon une séquence définie
   d. On insère la clé dans une liste en dehors de la table
3. Un inconvénient du sondage linéaire est :
   a. Il ne gère pas les collisions
   b. Il est impossible à implémenter
   c. Il provoque du clustering (grappes de cases occupées)
   d. Il nécessite que n soit premier
4. (Vrai/Faux) Avec l'adressage ouvert, certaines cases peuvent être marquées comme « supprimées » (DUMMY) sans redevenir complètement vides.
5. (Vrai/Faux) Avec le chaînage, la longueur moyenne des listes augmente avec le facteur de charge α.

Exercice Python 4 – Mini table avec chaînage

On veut implémenter une mini table de hachage avec chaînage, où les clés sont des entiers. On utilise une liste de listes table et une fonction h(k) = k % len(table).

Compléter les fonctions `inserer` et `rechercher` :

```
def creer_table(n):
    # crée une table avec n seaux vides
    return [[] for _ in range(n)]

def h(k, n):
    return k % n

def inserer(table, cle):
    n = len(table)
    i = h(cle, n)
    seau = table[i]
    # insérer "cle" dans le seau si elle ne s'y trouve pas déjà
    if ______________________________:
        ______________________________

def rechercher(table, cle):
    n = len(table)
    i = h(cle, n)
    seau = table[i]
    # renvoyer True si "cle" est dans le seau, False sinon
    for x in seau:
        if ________________________:
            return True
    return ________________________
```

Exercice Python 6 – Simuler l'adressage ouvert

On veut écrire une mini table avec adressage ouvert et sondage linéaire pour des clés entières. Les cases vides contiennent None. Compléter la fonction `inserer`.

```python
def creer_table(n):
    return [None] * n

def h(k, n):
    return k % n

def inserer(table, cle):
    n = len(table)
    i = h(cle, n)
    # sonder jusqu'à trouver une case vide ou la clé déjà présente
    for _ in range(n):   # on limite à n sondes
        if table[i] is None:
            ___________________________  # insérer la clé ici
            return
        elif table[i] == cle:
            return   # déjà présent
        i = (i + 1) % n   # case suivante

tab = creer_table(5)
for k in [1, 6, 11]:
    inserer(tab, k)
print(tab)  # on doit voir que toutes les clés sont présentes
```

EXERCICES DU CHAPITRE V : FACTEUR DE CHARGE ET PERFORMANCES

Exercice V.1 – Calculer et interpréter α

Une table de hachage possède un tableau de taille n = 100. On considère différents nombres d'éléments stockés :

1. 20 éléments
2. 50 éléments
3. 80 éléments

Pour chacun :

- calculer le facteur de charge α,
- commenter en une phrase l'effet intuitif sur les performances (chaînage ou adressage ouvert).

Exercice V.2 – Admissible ou pas ?

Pour chaque situation suivante, dire si la recherche d'un élément est en pratique plutôt O(1) ou tend vers O(n), et pourquoi :

1. Table avec chaînage, n=1000, α≈0,5, bonne fonction de hachage.
2. Table avec chaînage, n=1000, mais toutes les clés tombent dans le même seau.
3. Table avec adressage ouvert (double hachage), n=10 000, α≈0,7.
4. Table avec adressage ouvert (sondage linéaire), n=10 000, α≈0,95.

Exercice V.3 – Faut-il redimensionner ?

On a une table de hachage avec n = 1000 seaux.
On insère peu à peu des éléments, et on suit α :

1. À 400 éléments : que vaut α ? Redimensionner ou non ?
2. À 800 éléments : que vaut α ? Redimensionner ou non ?
3. On décide de redimensionner à n = 2000 quand α dépasse 0,7.
    a. Que devient α juste après le redimensionnement si on avait 800 éléments ?
    b. Pourquoi accepter ce « gros coût ponctuel » (rehachage de tous les éléments) malgré tout ?

Quizz 4 – Facteur de charge & performances

1. Le facteur de charge α d'une table de hachage est :
    a. n / (nombre d'éléments)
    b. (nombre d'éléments) / n
    c. (nombre de collisions) / n
    d. Toujours égal à 1
2. Si α augmente fortement (proche de 1) dans une table avec adressage ouvert, alors :
    a. Les temps de recherche deviennent proches de O(1)
    b. Les temps de recherche tendent vers O(n)
    c. Cela ne change rien
    d. Les clés sont triées automatiquement
3. Redimensionner une table de hachage (augmenter n et tout ré-insérer) :
    a. Est toujours une mauvaise idée
    b. Ne change pas le facteur de charge
    c. Peut coûter cher ponctuellement mais améliore le temps moyen
    d. Supprime toutes les collisions futures
4. (Vrai/Faux) On redimensionne souvent la table quand le facteur de charge dépasse un certain seuil (par ex. 0,7).
5. (Vrai/Faux) Le coût amorti d'une insertion reste O(1) même si, de temps en temps, une insertion déclenche un redimensionnement coûteux.

EXERCICES DU CHAPITRE VI : FONCTIONS DE HACHAGE UNIVERSELLES

Exercice VI.1 – Pourquoi une seule fonction ne suffit pas ?

1. Expliquer pourquoi aucune fonction de hachage fixe h ne peut garantir de bonnes performances pour tous les jeux de données possibles.
   (Indication : imaginer un adversaire qui choisit les clés après avoir vu h)
2. Comparer les deux garanties :
    - (A) « Pour chaque jeu de données, il existe une fonction de hachage qui se comporte bien. »
    - (B) « Pour chaque jeu de données, si on choisit une fonction au hasard dans $\mathcal{H}$, elle se comporte bien en moyenne. »
3. Pourquoi (B) est-elle plus intéressante pour l'analyse de la table de hachage en pratique ?

Exercice VI.2 – Cette famille est-elle universelle ?

On considère deux familles de fonctions de hachage $\mathcal{H}_1$ et $\mathcal{H}_2$, de U vers {0, …, n−1}.

1. Famille $\mathcal{H}_1$: l'ensemble de toutes les fonctions de U vers {0, …, n−1}.
    a. Montrer que $\mathcal{H}_1$ est une famille universelle, en utilisant $Pr[h(x)=h(y)] \leq 1/n$ pour $x \neq y$.
2. Famille $\mathcal{H}_2$: n fonctions constantes : pour chaque $i \in \{0, \ldots, n-1\}$, $h_i(k)=i$ pour toutes les clés k.
    a. Vérifier que pour toute clé k fixée et pour tout i, la probabilité que $h(k)=i$ vaut $1/n$.
    b. Expliquer pourquoi, malgré cela, $\mathcal{H}_2$ n'est pas universelle (en étudiant $Pr[h(x)=h(y)]$ pour $x \neq y$).

Exercice VI.3 – Hachage d'adresses IP : calcul concret

On se place dans un exemple :

- n = 11 (nombre de compartiments, premier),
- a = (a1, a2, a3, a4) = (2, 5, 3, 1)
- Deux « IP » : x = (10, 0, 5, 3) et y = (10, 0, 5, 4).

On définit : $h_a(z)=a \cdot z \bmod 11$

1. Calculer $h_a(x)$ et $h_a(y)$.
2. Y a-t-il collision pour ce choix de a ?
3. Combien d'opérations élémentaires (multiplications, additions, modulo) as-t-on effectué pour un hachage ? Pourquoi peut-on dire que c'est en temps constant ?
4. Expliquer pourquoi il est utile que 11 soit premier.

Exercice VI.4 – Pourquoi choisir n premier, > 255 ?

1. Expliquer pourquoi il est raisonnable d'imposer n > 255 quand chaque composante d'une IP vaut entre 0 et 255.
2. Donner deux raisons pour lesquelles il est préférable que n soit premier plutôt que 512 (2^9) :
    a. du point de vue théorique (structure de $\mathbb{Z}/n\mathbb{Z}$),
    b. du point de vue de la répartition des valeurs modulo n.
3. Relier ce choix aux recommandations déjà vues pour le nombre de compartiments d'une table de hachage.

Exercice VI.5 – Longueur moyenne d'un seau avec hachage universel

Table avec chaînage, n compartiments, m clés, facteur de charge $\alpha=m/n$.

1. On fixe une clé x (non présente dans la table) et on regarde le compartiment h(x). Soit S l'ensemble des m clés insérées.
    - Pour chaque $y \in S$, définir $X_y = 1$ si y tombe dans le même compartiment que x, 0 sinon par une relation faisant intervenir h(x) et h(y).
    - Exprimer la longueur du seau h(x) en fonction des X_y.
2. En utilisant l'universalité, donner la borne supérieure de $E[X_y]$ pour $y \neq x$ en fonction de n.
3. En déduire que l'espérance de la longueur du seau h(x) est au plus α.
4. Interpréter : que signifie « le temps d'une recherche infructueuse est en $O(1 + \alpha)$ » si α est borné (par ex. $\alpha \leq 2$) ?

Quizz 5 – Hachage universel

1. Une famille de hachage universelle garantit (grossièrement) que pour toutes paires x≠y :
   A. h(x) = h(y) toujours
   B. h(x) ≠ h(y) toujours
   C. La probabilité que h(x) = h(y) est petite (≤ 1/n)
   D. Les collisions sont impossibles
2. Le but de choisir une fonction de hachage au hasard dans une famille est :
   A. D'empêcher totalement les collisions
   B. De rendre la fonction plus lente
   C. De n'avoir aucune garantie
   D. De garantir de bonnes performances en moyenne, même pour des données adverses

EXERCICES DU CHAPITRE VII : LES DICTIONNAIRES PYTHON

Exercice VII.1 – Table d'indices vs table compacte

Répondre en quelques phrases :

1. Rôle de la table d'indices : que contient une case ? pourquoi est-elle clairsemée ?
2. Rôle de la table compacte d'entrées : que contient chaque entrée en mode combiné ? pourquoi est-elle dense ?
3. Comment cette organisation permet-elle de préserver l'ordre d'insertion lors de l'itération sur le dictionnaire ?
4. Pourquoi cette table compacte améliore-t-elle la localité cache par rapport à des pointeurs dispersés ?

Exercice VII.2 – Comprendre i = h & (m – 1)

On suppose : m = 8, donc m = 2^3 et m−1 = 7 ; trois valeurs de hash : $h_1 = 13$, $h_2 = 42$ et $h_3 = 57$

1. Calculer pour chaque h_i :
   a. h_i mod 8
   b. h_i & 7
2. Vérifier que les deux résultats coïncident. Conclure que i = h & (m - 1) équivaut à i = h % m quand m est une puissance de 2.
3. Donner deux raisons pour lesquelles cette opération de masquage peut être plus rapide qu'un modulo général.

Exercice VII.4 – Insertion avec EMPTY et DUMMY

Table d'indices de taille 8 (indices 0 à 7) et table compacte avec deux entrées.

Table compacte :

Adresse mémoire	**0**	**1**
	(hash("a"), ptr_"a", ptr_1)	(hash("b"), ptr_"b", ptr_2)

Table d'indices :

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Valeur	-	0	DUMMY	1	EMPTY	EMPTY	EMPTY	EMPTY

On suppose :

hash("c") compressé → indice initial 1,
hash("b") compressé → indice initial 3,
sondage linéaire : i, i + 1, i + 2, … (mod 8).

On effectue :

1. d["c"] = 3
2. d["b"] = 5 (mise à jour)
3. del d["a"] (on met DUMMY à son indice)
4. d["d"] = 4 avec hash compressé donnant l'indice initial 1.

Pour chaque opération :

- donner la suite d'indices sondés ;
- dire si on rencontre EMPTY, DUMMY ou une entrée existante, et ce qui se passe ;
- donner l'état final de la table d'indices et de la table compacte.

Exercice VII.5 – Coût moyen vs pire cas dans dict

Pour chaque situation, dire si le coût moyen d'une opération (d[clé], d[clé] = val, del d[clé]) est pratiquement O(1) ou peut approcher O(n), et pourquoi.

1. Dictionnaire classique, 10 000 entrées, $\alpha \approx 0{,}6$, clés variées, pas d'attaque.
2. Dictionnaire attaqué : des milliers de chaînes choisies pour provoquer des collisions (sans hachage randomisé).
3. Même situation qu'en 2, mais avec hachage randomisé pour les str et redimensionnement (comme en Python moderne).
4. Dictionnaire très gros où une unique insertion déclenche un redimensionnement et le re-hachage de toutes les entrées.

Quizz 6 – Dictionnaires Python

1. Dans un dictionnaire Python moderne, la table interne d'indices :
   A. Est toujours pleine à 100 %
   B. Est souvent clairsemée (sparse)
   C. Contient toutes les données (clés et valeurs)
   D. Est triée par ordre alphabétique
2. Dans un dict Python, les clés doivent être :
   A. Mutables (objet qui peut être modifié après sa création)
   B. Non hashables
   C. Hashables et (en pratique) immuables (une fois créé, le contenu ne change plus)
   D. Obligatoirement des chaînes
3. (Vrai/Faux) Le hachage des chaînes (str) en Python est randomisé pour limiter les attaques par collisions.
4. (Vrai/Faux) Parcourir un dictionnaire avec for k in d: se fait dans l'ordre de hachage des clés, sans lien avec l'ordre d'insertion.

Exercice Python 5 – Clés hashables ou non

Pour chaque insertion ci-dessous, dire si elle fonctionne ou provoque une erreur, et pourquoi. Proposer une correction pour les lignes problématiques.

```
d = {}

d[42] = "entier"
d["hello"] = "chaine"
d[(1, 2, 3)] = "tuple"

liste = [1, 2, 3]
d[liste] = "liste mutable"

ens = {1, 2, 3}
d[ens] = "ensemble mutable"
```

Python 6 – Compter les occurrences de mots

```
def compter_mots(mots):
    d = {}
    for mot in mots:
        if mot in d:
            d[mot] += 1
        else:
            d[mot] = 1
    return d

mots = ["un", "deux", "un", "trois", "deux", "un"]
print(compter_mots(mots))  # {"un": 3, "deux": 2, "trois": 1}
```

EXERCICES DU COURS CLASSÉS PAR CHAPITRE

Chapitre du cours	Exercices écrits	Quiz associés	Exercices Python associés
I – INTRODUCTION I.1 Présentation des tables de hachage I.2 Qu'est-ce qu'une clé ?	**Exercice I.1 – Est-ce qu'une table de hachage est adaptée ?** **Exercice I.2 – Clé ou valeur ?**	**Quizz 1 – Clés & dictionnaires**	**Python 1 – Déduplication avec set/dict** **Python 2 – Compter les occurrences de mots**
II – OPÉRATIONS PRISES EN CHARGE PAR LES TABLES DE HACHAGE	**Exercice II.1 – Identifier les opérations** **Exercice II.2 – Tableau impossible**		
III – EXEMPLES D'APPLICATIONS III.1 Déduplication III.2 Recherches dans un vaste espace d'états	**Exercice III.1 – Simuler une déduplication** **Exercice III.2 – Pourquoi mémoriser les sommets visités ?**		**Python 3 – BFS avec ensemble visited**
IV – IMPLÉMENTATION : IDÉES GÉNÉRALES IV.1 Liste en Python IV.2 Fonction de hachage & table IV.3 Collisions IV.4 Chaînage IV.5 Adressage ouvert IV.6 Choisir une bonne fonction de hachage	**Exercice IV.1 – Pourquoi la liste n'est pas suffisante** **Exercice IV.2 – Calculer des indices de hachage** **Exercice IV.3 – Collisions inévitables** **Exercice IV.4 – Simuler une table avec chaînage** **Exercice IV.5 – Sondage linéaire à la main** **Exercice IV.6 – Bonne ou mauvaise fonction de hachage ?**	**Quizz 2 – Fonction de hachage & collisions** **Quizz 3 – Chaînage & adressage ouvert**	**Python 4 – Mini table de hachage avec chaînage** **Python 5 – Simuler l'adressage ouvert**
V – FACTEUR DE CHARGE ET PERFORMANCES V.1 Charge & performances V.2 Gestion du facteur de charge V.3 Choisir une bonne fonction de hachage V.4 Choisir la stratégie de collision	**Exercice V.1 – Calculer et interpréter α** **Exercice V.2 – Admissible ou pas ?** **Exercice V.3 – Faut-il redimensionner ?**	**Quizz 4 – Facteur de charge & performances**	
VI – FONCTIONS DE HACHAGE UNIVERSELLES VI.1 Définition mathématique VI.2 Exemple : hachage d'IP VI.3 Complexité avec chaînage VI.4 Complexité avec adressage ouvert	**Exercice VI.1 – Pourquoi une seule fonction ne suffit pas ?** **Exercice VI.2 – Famille universelle ou pas ?** **Exercice VI.3 – Hachage d'adresses IP : calcul concret** **Exercice VI.4 – Pourquoi choisir n premier, > 255 ?** **Exercice VI.5 – Longueur moyenne d'un seau**	**Quizz 5 – Hachage universel**	
VII – LES DICTIONNAIRES PYTHON VII.1 Structure générale VII.2 Insertion d'une valeur VII.3 Suppression d'une valeur VII.4 Recherche d'une valeur	**Exercice VII.1 – Table d'indices vs table compacte** **Exercice VII.2 – Comprendre i = h & (m−1)** **Exercice VII.3 – Insertion avec EMPTY et DUMMY** **Exercice VII.4 – Coût moyen vs pire cas dans dict**	**Quizz 6 – Dictionnaires Python**	**Python 6 – Clés hashables ou non** **Python 7 – Compter les occurrences**