DEVOIR MAISON
Corrigé

I. Hachage

Question 1

Chaque fonction reçoit une chaîne et renvoie un entier. Les codes sont translatés par `ord('A')`, de sorte que « A » donne 0.

Dans h_4, les parenthèses sont nécessaires : en Python, `*` est prioritaire sur `^`. Sans elles, le produit aurait lieu avant le ou exclusif.

```
def h1(ch):
    return ord(ch[0]) - ord("A")


def h2(ch):
    s = 0
    for c in ch:
        s = s + ord(c) - ord("A")
    return s


def h3(ch):
    res = 0
    for k in range(len(ch)):
        res = res + h1(ch[k]) * 26**k
    return res


def h4(ch):
    x = 2166136261
    for c in ch:
        x = (ord(c) ^ x) * 16777619
        x = x % 2**32
    return x
```

Contrôle sur la chaîne `"ITC"` : $h_1 = 8$, $h_2 = 29$, $h_3 = 1854$, $h_4 = 3553794089$.

Pour h_3, les lettres I, T et C valent 8, 19 et 2, donc

$$8 \times 26^0 + 19 \times 26^1 + 2 \times 26^2 = 8 + 494 + 1352 = 1854.$$

Question 2

```
with open("mots.txt", "r", encoding="utf-8") as fichier:
    mots = [ligne.strip() for ligne in fichier]
```

Le fichier est ouvert en lecture, et `strip()` retire le saut de ligne. Le lexique contient 130 557 mots, en capitales, de longueur 2 à 15. Les quatre premiers sont `AA`, `AH`, `AI`, `AN`.

Question 3

```
from random import sample

def tirage(liste):
    return sample(liste, 2)
```

`sample` tire sans remise : les deux mots sont distincts.

Question 4

L'indice dans une table de taille $n = 10\,000$ est $h(k)$ mod n. Il y a collision si et seulement si les deux indices sont égaux.

```
def collision(h, liste):
    a, b = tirage(liste)
    return h(a) % 10000 == h(b) % 10000
```

Par exemple, `A` et `AD` collisionnent pour h_1 (même initiale) et pas pour h_2.

Question 5

```
def proba(h, liste):
    compte = 0
    N = 10**6
    for i in range(N):
        if collision(h, liste):
            compte = compte + 1
    return compte / N
```

La fréquence est le nombre de collisions divisé par le nombre de tirages, pour chacune des quatre fonctions. Un tirage d'un million de couples donne des valeurs de l'ordre de :

	h_1	h_2	h_3	h_4
Fréquence	7×10^{-2}	10^{-2}	10^{-4}	10^{-4}

Les deux dernières fluctuent d'un tirage à l'autre autour de 10^{-4} : pour une probabilité 10^{-4} et $N = 10^6$, l'écart-type de la fréquence vaut environ 10^{-5}.

Question 6

Sous l'hypothèse de hachage uniforme simple, deux clés indépendantes tombent uniformément dans $\{0, \ldots, n-1\}$.

Le premier indice étant fixé, le second le retrouve avec probabilité $1/n$. Donc

$$\mathrm{P}(\text{collision}) = \frac{1}{n} = \frac{1}{10\,000} = 10^{-4}.$$

On peut aussi décomposer selon la case du premier mot :

$$\sum_{i=0}^{n-1} \mathrm{P}(h(k_1) = i)\,\mathrm{P}(h(k_2) = i) = \sum_{i=0}^{n-1} \left(\frac{1}{n}\right)^2 = \frac{1}{n}.$$

Question 7

La qualité se juge par la proximité de la fréquence mesurée avec $1/n = 10^{-4}$. Une probabilité plus petite que $1/n$ n'est pas le signe d'un meilleur hachage : si p_i est la probabilité de tomber dans la case i, deux clés collisionnent avec probabilité $\sum p_i^2$, qui est minimale pour la loi uniforme, et ce minimum vaut $1/n$.

- ⋆ h_1 est insuffisante : environ 7×10^{-2}, soit plusieurs centaines de fois la référence.
- ⋆ h_2 est insuffisante : environ 10^{-2}. La somme des codes d'un mot de longueur au plus 15 est au plus $25 \times 15 = 375$, donc moins de 400 cases sur 10 000 sont utilisées.
- ⋆ h_3 et h_4 sont satisfaisantes : leurs fréquences sont compatibles avec 10^{-4}.

Question 8

h_1 ne dépend que de la première lettre. Son image est contenue dans $\{0, \dots, 25\}$: au plus 26 cases d'une table de taille 10 000 sont atteignables, d'où des collisions massives.

Même dans une table de taille 26, ce hachage ne serait pas uniforme : les initiales du lexique sont très inégalement réparties.

II. Programmation dynamique

On note $f(i, j)$ le booléen vrai lorsqu'un sous-ensemble des i premiers éléments de E a pour somme j. On cherche $f(\text{len}(E), S)$.

Dans tout ce qui suit, $E = [3, 34, 4, 12, 5, 2]$. On doit obtenir `True` pour $S = 9$ et `False` pour $S = 30$.

Question 1

Si $j = 0$, le sous-ensemble vide convient, y compris lorsque $i = 0$: ce test vient en premier. Si $i = 0$ et $j > 0$, aucun élément n'est disponible. Si $E[i-1]$ dépasse le reste à faire, la seule branche est l'exclusion. Sinon, on renvoie le ou logique de l'exclusion et de l'inclusion, cette dernière avec le reste $j - E[i-1]$.

```
def somme_naive(E, i, j):
    if j == 0:
        return True
    if i == 0:
        return False
    if E[i - 1] > j:
        return somme_naive(E, i - 1, j)
    return somme_naive(E, i - 1, j) or somme_naive(E, i - 1, j - E[i - 1])
```

Question 2

Mémoïsation

Le dictionnaire est créé une seule fois, dans la fonction englobante, et partagé par tous les appels. Un couple (i, j) déjà connu est renvoyé sans recalcul. Le résultat est enregistré avant d'être renvoyé. La récurrence est celle de la version naïve.

```python
def somme_memo(E, i, j):
    D = {}

    def inter(a, b):
        if (a, b) in D:
            return D[(a, b)]
        if b == 0:
            res = True
        elif a == 0:
            res = False
        elif E[a - 1] > b:
            res = inter(a - 1, b)
        else:
            res = inter(a - 1, b) or inter(a - 1, b - E[a - 1])
        D[(a, b)] = res
        return res

    return inter(i, j)
```

Complexité

Dans la version naïve, chaque appel lance jusqu'à deux appels, et l'indice décroît de 1 à chaque appel : l'arbre a un branchement 2 et une profondeur au plus $n = \text{len}(E)$. Le nombre d'appels est donc en $O(2^n)$. La profondeur, elle, est $O(n)$: ce n'est pas le nombre d'appels.

Dans la version mémoïsée, les états sont les couples (i, j) avec $0 \leqslant i \leqslant n$ et $0 \leqslant j \leqslant S$, soit $(n+1)(S+1)$ états. Chacun est calculé au plus une fois, avec un travail constant hors appels récursifs. La complexité est $O(nS)$.

La mémoïsation réduit le nombre d'appels parce que le même couple (i, j) revient de nombreuses fois dans l'arbre naïf, par des chemins différents, et n'est recalculé qu'une fois.

Question 3

Version itérative

La matrice a $(n+1)$ lignes et $(S+1)$ colonnes. La colonne $j = 0$ est vraie : la somme nulle est atteinte par le sous-ensemble vide. La ligne $i = 0$ reste fausse pour $j > 0$, grâce à l'initialisation. On remplit par i croissant, afin que la ligne $i - 1$ soit déjà connue, avec la même récurrence que $f(i, j)$. La valeur renvoyée est $M[n][S]$.

```python
def somme_iterative(E, S):
    n = len(E)
    M = [[False for j in range(S + 1)] for i in range(n + 1)]
    for i in range(n + 1):
        M[i][0] = True
    for i in range(1, n + 1):
        for j in range(1, S + 1):
            if E[i - 1] > j:
                M[i][j] = M[i - 1][j]
            else:
                M[i][j] = M[i - 1][j] or M[i - 1][j - E[i - 1]]
    return M[n][S]
```

Complexité

Deux boucles imbriquées parcourent les cases, et chaque case se remplit en temps constant. La complexité est $O(nS)$, comme pour la mémoïsation.

Question 4

On recalcule la matrice. Si $M[n][S]$ est faux, aucun sous-ensemble ne convient et l'on renvoie `None`. Sinon on remonte depuis (n, S) : si $M[i-1][j]$ est vrai, l'élément $E[i-1]$ est écarté ; sinon il est pris, et le reste à atteindre devient $j - E[i-1]$. On ne quitte une case vraie que vers une case vraie.

```
def solution(E, S):
    n = len(E)
    M = [[False for j in range(S + 1)] for i in range(n + 1)]
    for i in range(n + 1):
        M[i][0] = True
    for i in range(1, n + 1):
        for j in range(1, S + 1):
            if E[i - 1] > j:
                M[i][j] = M[i - 1][j]
            else:
                M[i][j] = M[i - 1][j] or M[i - 1][j - E[i - 1]]
    if not M[n][S]:
        return None
    sol = []
    i, j = n, S
    while i > 0 and j > 0:
        if M[i - 1][j]:
            i = i - 1
        else:
            sol.append(E[i - 1])
            j = j - E[i - 1]
            i = i - 1
    sol.reverse()
    return sol
```

Sur l'exemple du sujet, `solution(E, 9)` renvoie `[4, 5]`, de somme 9, et `solution(E, 30)` renvoie `None`. Pour $S = 0$, la liste vide est la bonne réponse : le sous-ensemble vide convient. Lorsqu'il existe plusieurs sous-ensembles, la fonction en choisit un, celui qui écarte un élément dès que c'est possible.