Anagrammes et réactions

INFORMATIQUE COMMUNE - Devoir n° 1 - Olivier Reynet

A Réactions dynamiques

a Optimisation d'un processus de synthèse chimique

Vous êtes un chimiste travaillant sur l'optimisation d'un processus de synthèse chimique pour produire un nouveau composé. Vous disposez d'une quantité limitée de réactif principal, soit 100 g, et vous devez choisir les réactions secondaires à inclure dans le processus.

Chaque réaction secondaire nécessite une certaine quantité du réactif principal et contribue à améliorer la pureté du produit final. Votre objectif est de maximiser la pureté du composé en choisissant les réactions à réaliser, tout en respectant la quantité limitée de réactif principal disponible. Une réaction secondaire ne peut être appliquée qu'une seule fois, pour des raisons logistiques. On fait l'hypothèse que les gains en pourcentage sont additifs.

b Quelles réactions inclure ?

Voici les réactions secondaires disponibles :

Réaction secondaire	**Consommation de réactif (g)**	**Gain en pureté (%)**
Catalyse par ajout d'un métal	20	15
Réaction de précipitation	30	25
Filtration par adsorption	10	10
Réaction enzymatique	40	35
Purification par distillation	25	20
Réaction redox	15	12
Chromatographie	35	30
Réaction acide-base	20	18

Quantité totale de réactif disponible : 100 g

c Problème :

Choisir les réactions secondaires à réaliser de sorte que la **consommation totale de réactif principal ne dépasse pas 100 g** et que le **gain en pureté** du produit final soit maximal.

R En Python, on choisit de représenter les réactions secondaires par une liste de tuples (gain, poids), où gain représente le gain en pureté dû à la réaction secondaire et poids le poids nécessaire de réactif principal pour réaliser cette réaction.

```
REACTIONS = [(15, 20), (25, 30), (10, 10), (35, 40), (20, 25), (12, 15), (30, 35), (18, 20)]
```

A1. Proposer une stratégie gloutonne pour résoudre ce problème et l'expliquer en français.

Solution : On pourrait par exemple choisir d'utiliser les réactions secondaires dont le gain en pureté est le plus fort en premier, jusqu'à atteindre le poids limite de réactif principal. Cette solution ne sera pas systématiquement optimale. On trouverait dans ce cas ([(35, 40), (30, 35), (20, 25)], 100, 85), c'est-à-dire un gain de 85% en pureté. Tout le réactif principal serait utilisé.

A2. Écrire une fonction de signature `glouton(reactions, pmax)` qui renvoie le gain en pureté obtenu en appliquant la stratégie précédente. On pourra réutiliser la fonction `insert_sort` de la partie B, en expliquant ce qu'il faudrait modifier afin de pouvoir l'utiliser correctement.

Solution :

```
def glouton_kp(reactions, pmax):
    poids = 0  # poids total de réactif principal
    gain = 0  # gain en pureté total
    selection = []  # selection des réactions secondaires
    reactions = sorted(reactions)  # tri ascendant en fonction du gain
    while len(reactions) > 0 and poids <= pmax:
        g, p = reactions.pop()  # choix glouton : le gain le plus grand
        if poids + p <= pmax:  # a-t-on suffisamment de réactif principal ?
            selection.append((g, p))
            poids += p
            gain += g
    return selection, poids, gain
```

A3. On cherche maintenant à trouver la solution optimale à ce problème. On désigne par $S(n,p)$ la pureté totale maximale que l'on peut obtenir en choisissant les n premières réactions secondaires et une limite de p grammes de réactif principal. On notera :

- g_n le gain de pureté liée à la réaction secondaire n,
- p_n la quantité de réactif principal nécessaire pour la réaction secondaire n.

On dispose de la récurrence suivante :

$$S(n,p) = \begin{cases} 0 & \text{si } n = 0 \text{ ou si } p = 0 \\ \max\big(g_n + S(n-1, p-p_n), S(n-1,p)\big) & \text{si } p_n \leqslant p \\ S(n-1,p) & \text{si } p_n > p \end{cases} \quad (1)$$

Interpréter en français cette relation de récurrence.

Solution : Formellement, on exprime cette récursivité ainsi :

$$S(n,p) = \begin{cases} 0 & \text{si } n = 0 \text{ ou si } p = 0 \\ \max\big(g_n + S(n-1, p-p_n), S(n-1,p)\big) & \text{si } p_n \leqslant p \\ S(n-1,p) & \text{si } p_n > p \end{cases} \quad (2)$$

Ce qu'il faut interpréter comme :

- Le gain en pureté est nul si on ne dispose par de réactif principal ou si on ne dispose pas de réaction secondaires.

- Si la quantité de réactif principal nécessité par la réaction secondaire n est inférieure à la quantité de réactif principal disponible, on peut choisir cette réaction secondaire. Néanmoins, elle peut ne pas faire partie de la solution optimale. On considère donc deux cas : la solution optimale est le maximum de :
  - la sélection de la réaction n et donc le gain en pureté associé g_n plus la solution optimale au sous-problème avec les $n-1$ réactions restantes et $p-p_n$ grammes restants de réactif principal.
  - la solution du sous-problème sans utiliser cette réaction : $S(n-1,p)$.
- Si la quantité de réactif principal nécessité par la réaction secondaire n est supérieure à la quantité de réactif principal disponible, on ne peut pas choisir cette réaction secondaire. Donc la solution optimale est la solution du sous-problème sans utiliser cette réaction : $S(n-1,p)$.

A4. Écrire une fonction de signature `pdyn_asc(reactions, pmax)` qui renvoie la solution optimale au problème avec `n` réactions secondaires possibles et un poids de réactif principal maximum de `pmax`. Cette fonction procède de manière ascendante, impérativement en complétant un tableau.

Solution :

```
def pdyn_asc(reactions, pmax):  # Programmation dynamique
    n = len(reactions)
    s = [[0 for _ in range(pmax + 1)] for _ in range(n + 1)]
    for i in range(n + 1):
        for p in range(pmax + 1):
            if i == 0 or p == 0:
                s[i][p] = 0  # pas de gain possible
            else:
                gi, pi = reactions[i - 1]  # on considère la ième réaction
                if pi <= p:
                    s[i][p] = max(gi + s[i - 1][p - pi], s[i - 1][p])
                else:
                    s[i][p] = s[i - 1][p]
    return s[n][pmax]
```

A5. Écrire une fonction de signature `mem(n, pmax, S)` qui renvoie la solution optimale au problème avec `n` réactions secondaires possibles et un poids de réactif principal maximum de `pmax`. Cette fonction procède de manière descendante en utilisant la mémoïsation dans un dictionnaire `S`.

Solution :

```
def mem(n, pmax, S):  # récursif avec memoïsation
    if (n, pmax) in S:
        return S[(n, pmax)]  # déjà mémorisé, on s'en sert
    elif n == 0 or pmax == 0:
        return 0  # condition d'arrêt
    else:
        g, p = REACTIONS[n - 1]  # on considère la nième réaction
        if p > pmax:
```

```
            S[(n, pmax)] = mem(n - 1, pmax, S)  # mémoïsation
            return S[(n, pmax)]
        else:
            S[(n, pmax)] = max(g + mem(n - 1, pmax - p, S), mem(n - 1, pmax, S))
            return S[(n, pmax)]
```

A6. Sachant qu'il est possible d'atteindre les 88% de gain en pureté avec 100g de réactif, l'approche gloutonne est-elle optimale? Quelles réactions devez-vous inclure dans le processus de synthèse pour obtenir le produit le plus pur possible tout en respectant la quantité limitée de réactif?

Solution : L'approche gloutonne n'est pas optimale, car on trouve 85%.
Il faudra choisir :

Réaction secondaire	**Consommation de réactif (g)**	**Gain en pureté (%)**
Réaction de précipitation	30	25
Filtration par adsorption	10	10
Réaction enzymatique	40	35
Réaction acide-base	20	18

A7. Écrire une fonction de signature `select_reactions(S: list[list[int]])-> (int, list):` qui renvoie les réactions secondaires sélectionnées d'après le tableau `S` créé par la programmation dynamique ascendante. Pour cela, il suffit de partir de la case contenant la solution et de parcourir le tableau ligne par ligne en descendant pour retrouver les réactions sélectionnées.

Solution :

```
def select_reactions(S: list[list[int]]) -> list:
    i = len(S) - 1
    pmax = len(S[0]) - 1
    selection = []
    while i > 0:
        gi, pi = REACTIONS[i - 1]
        if pi <= pmax:
            if gi + S[i - 1][pmax - pi] > S[i - 1][pmax]:
                selection.append((gi, pi))
                pmax -= pi
        i -= 1
    if pmax < 0:
        selection.pop()
    return selection
```

B Anagrammes

Soient deux chaînes de caractères a et b. On cherche à savoir si ces deux chaînes sont anagrammes l'une de l'autre.

■ **Exemple 1 — Exemples d'anagrammes.** En français on trouve notamment :

- raper et parer
- aspirine et parisien
- dispute et stupide
- engrais et graines

On rappelle que la fonction `ord(c : str)—> int` renvoie un nombre entier sur 8 bits qui code un caractère en machine. On note que `ord('a')` vaut 97, `ord('b')` vaut 98 et ainsi de suite dans l'ordre alphabétique.

R On ne considère par la suite que des mots composés de caractères **standards**, c'est-à-dire des mots sans signes de ponctuation, chiffres ou lettres diacritiques comme les accents ou la cédille. Par ailleurs, on suppose que tous les mots sont écrits en **minuscules**.

B1. Écrire une fonction de signature `indice(c : str)—> int` qui renvoie l'indice de la lettre c dans l'alphabet. On numérote les lettres à partir de 0 comme tout bon informaticien.

Solution :

```
def indice(c : str) —> int:
    return ord(c)—ord('a')
```

a Résolution par tri

Une première méthode pour résoudre le problème est de trier selon l'ordre lexicographique les deux mots : s'ils sont anagrammes l'un de l'autre, alors les trier résulte en une même séquence de lettres. Il suffit alors de comparer cette séquence.

B2. Écrire une fonction de signature `to_list(ch: str)—> list[int]` qui renvoie la liste des indices des lettres d'une chaîne de caractères. Par exemple, `to_list('aspirine')` revoie `[0, 18, 15, 8, 17, 8, 13, 4]`.

Solution :

```
def to_list(ch: str) —> list[int]:
    L = []
    for c in ch:
        L.append(indice(c))
    return L
```

Grâce à cette fonction, un mot peut être représenté par une liste d'entier.

B3. Écrire une fonction de signature `insert_sort(w: list[int])` qui tri la liste d'entiers w en utilisant le tri par insertion en programmation impérative, c'est-à-dire non récursivement. Cette fonction trie la liste en place, c'est-à-dire qu'elle ne crée pas une nouvelle liste pour stocker le résultat du tri.

Solution :

```
def insert_sort(w: list[int]):
    for i in range(1, len(w)):
        to_insert = w[i]
        j = i
        while j > 0 and w[j - 1] > to_insert:
            w[j] = w[j - 1]
            j -= 1
        w[j] = to_insert
```

B4. Écrire une fonction de signature `compare(w1: list[int], w2: list[int])-> bool` qui teste si deux listes sont identiques. Au début de la fonction, on s'assurera par une assertion que les deux listes possèdent la même longueur. L'usage de l'opérateur == pour comparer deux listes n'est pas autorisé.

Solution :

```
def compare(w1: list[int], w2: list[int]) -> bool:
    assert len(w1) == len(w2)
    for i in range(len(w1)):
        if w1[i] != w2[i]:
            return False
    return True
```

B5. Écrire une fonction de signature `tc_anagrammes(m1:str , m2: str)-> bool` qui teste si deux mots sont anagrammes l'un de l'autre. On utilisera les fonctions précédentes.

Solution :

```
def tc_anagrammes(m1:str , m2: str) -> bool:
    w1 = to_list(m1)
    w2 = to_list(m2)
    insert_sort(w1)
    insert_sort(w2)
    return compare(w1, w2)
```

B6. Quelle est la complexité de la fonction `tc_anagrammes` dans le pire des cas? On notera n la longueur des deux mots qu'on supposera identique. Justifier votre réponse.

Solution : La fonction `tc_anagrammes` appelle :

- deux fois `to_list` qui est de complexité linéaire par rapport à n quelque soit le cas,
- deux fois `insert_sort` qui est de complexité $O(n^2)$ dans le pire des cas,
- une fois la méthode `compare` qui est de complexité $O(n)$.

On en conclut donc que $C(n) = 2n + 2n^2 + n = O(n^2)$. La complexité est quadratique.

B7. En utilisant un autre tri générique, pourrait-on améliorer la complextié de `tc_anagrammes` dans le pire des cas ? Justifier votre réponse.

> **Solution :** Le meilleur tri générique (fusion) est de complexité $O(n\log n)$ dans le pire des cas. C'est pourquoi on aurait : $C(n) = 2n + 2n\log n + n = O(n\log n)$. La complexité serait linéarithmique.

B8. Pourrait-on utiliser le tri par comptage et si oui quelle serait la complexité de `tc_anagrammes` ? Justifier votre réponse.

> **Solution :** Le tri par comptage ne s'applique qu'à des entiers, donc il est possible de l'utiliser ici. Sa complexité est en $O(n + v_{max})$ dans le pire des cas. En ce qui nous concerne, v_{max} vaut 25. C'est pourquoi on aurait : $C(n) = 2n + 2(n+25) + n = O(n+25)$. La complexité serait linéaire dans le pire des cas.

b Résolution par comptage et dictionnaire

On déduit de la question précédente une autre méthode qui consiste à compter le nombre d'occurences de chaque lettre des mots : si ce décompte est identique, alors les mots sont anagrammes l'un de l'autre. On se propose d'utiliser un dictionnaire et de réaliser ce comptage de la manière suivante :

1. Créer un dictionnaire vide.
2. Compter les lettres du premier mot (remplir le dictionnaire)
3. Parcourir les lettres du deuxième mot :
   (a) Si la lettre n'est pas présente dans le dictionnaire, renvoyer faux.
   (b) Sinon, décrémenter le nombre d'occurrences de cette lettre. S'il n'y a plus aucune occurence de la lettre, effacer l'entrée associée à la lettre dans le dictionnaire.
4. Renvoyer vrai si le dictionnaire est vide, faux sinon.

R La syntaxe pour effacer une entrée associée à la clef `key` d'un dictionnaire d est `del d[key]`.

B9. Écrire une fonction de signature `dc_anagrammes(m1 : str, m2 : str)—> bool` qui teste si deux mots sont anagrammes l'un de l'autre en utilisant la méthode décrite ci-dessus. En début de fonction, on vérifiera par une assertion que les deux mots ont la même longueur.

> **Solution :**

```
def dc_anagrammes(m1 : str, m2 : str) -> bool:
    assert len(m1) == len(m2)
    compteur = {}
    for c in m1:
        if c in compteur:
            compteur[c] += 1
        else:
            compteur[c] = 1
```

```
        for c in m2:
            if c not in compteur:
                return False
            else:
                compteur[c] -= 1
                if compteur[c] == 0:
                    del compteur[c]

        return compteur == {}
```

B10. Quelle est la complexité de la fonction `dc_anagrammes` dans le pire de cas ? Justifier votre réponse, en tenant compte du fait que tester si un dictionnaire est vide s'effectue en temps constant.

Solution : Tester si un dictionnaire est vide s'effectue en un temps constant. Toutes les opérations de test de la présence ou non d'une clef dans un dictionnaire sont opérées en $O(1)$, par construction des dictionnaires en table de hachage Python. Donc toutes les opérations des corps de boucles sont effectuées en temps constant. On en déduit que cette fonction est de complextié linéaire par rapport à la taille de la chaîne de caractère, c'est-à-dire $C(n) = n + n = O(n)$