TP: programmation dynamique

M2 MEEF SD NSI

Table 1: Extrait du BO (programme de NSI pour la terminale).

Contenus	Capacités attendues	Commentaires
Programmation dynamique	Utiliser la programmation dynamique pour écrire un algorithme.	Les exemples de l'alignement de séquences ou du rendu de monnaie peuvent être présentés. La discussion sur le coût en mémoire peut être développée.

Suite de Fibonacci

On commence, pour s'échauffer, par l'exemple bien connu de la suite de Fibonacci, définie par $F_0 = F_1 = 1$ et, pour tout $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$. On peut la calculer par le programme récursif suivant:

```
def fibonacci(n):
    if n <= 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Question 1. Combien d'appels récursifs sont nécessaires au total pour calculer la valeur de F_n, en fonction de n ?

Question 2. Écrire une fonction non récursive réalisant le calcul de manière ascendante. A-t-on besoin de stocker tous les résultats des appels précédents ?

Une autre technique que le calcul ascendant existe. Il s'agit de la *mémoïsation* (terme introduit par l'informaticien Donald Michie en 1968, provenant du latin *memorandum*, littéralement *qui doit être rappelé*) qui consiste à conserver la fonction récursive, mais à stocker les résultats des appels récursifs au fur et à mesure de leur retour. La façon la plus simple de réaliser la mémoïsation dans un programme récursif est d'utiliser un dictionnaire global qu'on utilise au sein de la fonction elle-même:

```
fibonacci_resultat = {}
def fibonacci_memo(n):
    if n <= 1:
        return 1
    if n not in fibonacci_resultat:
        fibonacci_resultat[n] = fibonacci_memo(n - 1) + fibonacci_memo(n - 2)
    return fibonacci_resultat[n]
```

Question 3. Combien de fois la fonction `fibonacci_memo` est-elle appelée lors du calcul de F_n, en fonction de n ?

Remarque sur Python. La méthode précédente demande de modifier la fonction récursive que l'on écrit. On peut éviter cela en Python, à l'aide de *décorateurs*: on écrit une fonction `memoize` qui

prend en argument une fonction (récursive) et qui doit la modifier pour appliquer la mémoïsation. Ensuite, on utilise un décorateur `@memoize` juste avant la définition de la fonction `fibonacci` et le tour est joué !

```
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def fibonacci(n):
    if n <= 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Finalement, on se rend compte qu'on a réinventé une technique déjà disponible dans la bibliothèque `functools`, à base de *cache LRU* (pour *least recently used*):

```
from functools import lru_cache
@lru_cache(maxsize=None) # ou maxsize=n pour une taille de cache de n éléments
def fibonacci(n):
    if n <= 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Il est alors intéressant de pouvoir avoir des statistiques sur l'utilisation du cache. Essayez cela par exemple…

```
[fibonacci(n) for n in range(16)]
fibonacci.cache_info()
```

Formatage équilibré d'un paragraphe

On s'intéresse au problème du formatage équilibré d'un paragraphe. On se donne une suite de n mots $m_1, \dots, m_n$ de longueurs $\ell_1, \ell_2, \dots, \ell_n$ et on souhaite imprimer ce paragraphe de manière équilibrée sur des lignes ne pouvant contenir qu'un maximum de M caractères chacune (on suppose donc que $\ell_i \leq M$ pour tout i).

Si une ligne contient les mots de m_i à m_j inclus et qu'on laisse une espace entre chaque mot, le nombre de d'espaces supplémentaires à la fin de la ligne est

$$f(i,j) = M - (j-i) - \sum_{k=i}^{j} \ell_k.$$

L'objectif est de minimiser la somme des cubes des nombres de caractères d'espacement présents à la fin de chaque ligne, hormis la dernière ligne (qui peut donc être presque vide si nécessaire). On appelle *déséquilibre de la ligne* la quantité $f(i,j)^3$.

Par exemple, on a trois possibilités pour formater la première ligne de la Déclaration universelle des droits de l'homme, en lignes de 20 caractères (puisque le quatrième mot ferait dépasser):

ligne	espaces	déséquilibre
`Tous.....`	16	4096

ligne	espaces	déséquilibre
`Tous.les.....`	12	1728
`Tous.les.êtres.....`	6	216

Question 4. Définir une fonction pour calculer le déséquilibre d'une ligne.

```
def desequilibre(mots, largeur):
    """Renvoie le déséquilibre pour la liste de mots donnée en fonction de la
    largeur donnée.

    >>> desequilibre(['Tous'], 20)
    4096
    >>> desequilibre(['Tous', 'les'], 20)
    1728
    >>> desequilibre(['Tous', 'les', 'êtres'], 20)
    216
    """
```

Question 5. Écrire une fonction `desequilibre_total` qui calcule la somme des déséquilibres des lignes d'un paragraphe en fonction d'une largeur donnée (sans compter la dernière ligne). Le paragraphe est passé comme une liste de liste de mots.

Un premier algorithme de formatage équilibré de paragraphe est la stratégie gloutonne consistant à remplir les lignes les unes après les autres en mettant à chaque fois le plus de mots possibles sur la ligne en cours.

Question 6. Implémenter une fonction `formate_glouton` qui prend en arguments une liste de mots et une largeur et applique l'algorithme glouton pour calculer un découpage.

Question 7. Trouver un exemple où la stratégie gloutonne ne produit pas un formatage optimal et le vérifier à l'aide de la fonction `desequilibre_total`.

On veut maintenant procéder par programmation dynamique. On note $d(i)$ la valeur minimale du déséquilibre total occasionné par le formatage du paragraphe formé des mots $m_i, m_{i+1}, \dots, m_n$.

Question 8. Trouver une formule récursive pour calculer $d(i)$, en n'oubliant pas le cas où tous les mots peuvent tenir sur la dernière ligne.

Question 9 En déduire un algorithme de programmation dynamique calculant le déséquilibre minimal qu'on puisse obtenir pour une valeur de M et des longueurs $\ell_1, \dots, \ell_n$ données (qu'on suppose toutes inférieures à M).

Question 10. Implémenter cet algorithme sous forme d'une fonction `formate` qui répond à la même spécification que `formate_glouton` définie plus haut.

Question 11. Quelle est la complexité de l'algorithme précédent ?

Question 12. Écrire un programme qui prend un texte sur l'entrée standard et produit sur la sortie standard le même texte avec les paragraphes formatés à une largeur donnée. Dans le texte donné en entrée, les paragraphes sont séparés par une ou plusieurs lignes vides et les mots sont séparés par une ou plusieurs espaces, un retour à la ligne comptant comme espace. Dans le texte produit en sortie, les mêmes conventions s'appliquent (donc appliquer le programme une seconde fois ne doit rien changer au résultat).