Programmation dynamique

Programmation dynamique

Exercice 1 *QCM d'activation*

Pour chaque question, une seule réponse est correcte.

1. La programmation dynamique est particulièrement adaptée aux problèmes qui présentent :
   A. des sous-problèmes indépendants (comme dans « diviser pour régner »)
   B. des sous-problèmes chevauchants (un même sous-problème apparaît plusieurs fois)
   C. une seule solution possible
   D. une complexité toujours linéaire
2. La version récursive naïve de Fibonacci a une complexité temporelle :
   A. $O(n)$
   B. $O(n^2)$
   C. $O(2^n)$
   D. $O(n \log n)$
3. La mémoïsation (approche *top-down*) consiste à :
   A. remplir un tableau du plus petit sous-problème au plus grand
   B. stocker les résultats des appels récursifs déjà calculés pour éviter les recalculs
   C. utiliser un algorithme glouton avec un cache
   D. résoudre le problème sans récursivité
4. Dans le problème du sac à dos par programmation dynamique, le tableau contient en ligne i et colonne p :
   A. le poids maximal qu'on peut porter avec i objets
   B. le nombre d'objets de poids p
   C. la valeur maximale atteignable avec les i premiers objets et un poids maximal de p
   D. le nombre de façons de remplir le sac avec i objets
5. Pour le système de monnaie $(1, 3, 4)$ et la somme $s = 6$, l'algorithme glouton donne :
   A. deux pièces de 3 (solution optimale)
   B. une pièce de 4, puis une de 1, puis une de 1 (trois pièces)
   C. une pièce de 3, puis une de 3 (solution optimale)
   D. six pièces de 1

Exercice 2 *Fibonacci : comparer les approches (guidé)*

Cet exercice compare les trois approches vues en cours pour calculer les termes de la suite de Fibonacci.

Partie A. Approche récursive naïve

1. Écrire la fonction récursive `fibo(n)` (cas de base : $F_0 = 0$, $F_1 = 1$).
2. Dessiner l'arbre des appels pour `fibo(5)`. Combien de fois `fibo(2)` est-il calculé ?
3. Tester `fibo(35)`. Combien de temps cela prend-il approximativement ?

Partie B. Mémoïsation (*top-down*)

4. Compléter la fonction suivante qui utilise un dictionnaire pour stocker les résultats :

```
def fibo_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n in ________:
        return ________
    if n == 0 or n == 1:
        return n
    memo[n] = fibo_memo(________, memo) + fibo_memo(________, memo)
    return ________
```

5. Tester avec `fibo_memo(100)`. Comparer le temps d'exécution avec la version naïve.

Partie C. Approche ascendante (*bottom-up*)

6. Compléter la fonction suivante qui remplit un tableau de gauche à droite :

```
def fibo_dyn(n):
    if n == 0:
        return 0
    t = [0] * (n + 1)
    t[1] = ________
    for i in range(________, ________):
        t[i] = ________ + ________
    return t[n]
```

7. Quelle est la complexité en temps et en espace de cette fonction ? Peut-on réduire l'espace mémoire utilisé ?

Exercice 3 *Rendu de monnaie (guidé)*

On dispose du système de monnaie `monnaie = [1, 3, 4]` et on souhaite rendre la somme $s = 6$ avec le minimum de pièces.

Partie A. Algorithme glouton

1. Programmer la fonction `rendu_glouton(monnaie, s)` qui applique l'algorithme glouton : on rend toujours la plus grande pièce possible.
2. Tester avec `rendu_glouton([1, 3, 4], 6)`. La solution est-elle optimale ?

Partie B. Programmation dynamique

3. Remplir à la main le tableau suivant, où chaque case contient le nombre minimal de pièces pour atteindre la somme en colonne avec les pièces disponibles en ligne :

monnaie \ somme	**0**	**1**	**2**	**3**	**4**	**5**	**6**
{} (aucune)	0	∞	∞	∞	∞	∞	∞
{1}							
{1, 3}							
{1, 3, 4}							

4. Quelle est la relation de récurrence permettant de remplir la case (i, j) du tableau ?
5. Programmer la fonction `rendu_dynamique(monnaie, s)` qui retourne le nombre minimal de pièces pour rendre la somme `s`.
6. Modifier la fonction pour qu'elle retourne également la décomposition en pièces (par exemple `[3, 3]` pour rendre 6 avec le système `[1, 3, 4]`).

Exercice 4 *Sac à dos*

On dispose des objets suivants et d'un sac pouvant porter au maximum $w = 7$ kg :

Objet	**Valeur**	**Poids**
A	5	3
B	8	4
C	4	2
D	6	1

1. Appliquer l'algorithme glouton « meilleur rapport valeur/poids » et donner la solution obtenue. Est-elle optimale ?
2. Construire le tableau de programmation dynamique de taille 5×8 (quatre objets + ligne 0, poids de 0 à 7).
3. En déduire la valeur maximale et les objets sélectionnés en remontant dans le tableau.
4. Programmer la fonction `sac_a_dos(objets, w)` où `objets` est une liste de tuples `(valeur, poids)`.

Exercice 5 *Distance d'édition*

La distance d'édition (ou distance de Levenshtein) entre deux chaînes est le nombre minimal d'opérations (insertion, suppression, substitution) pour transformer l'une en l'autre. Ce problème est fondamental en bio-informatique pour comparer des séquences ADN.

1. Construire à la main le tableau de programmation dynamique pour transformer `"BAIN"` en `"BRIN"`.

   Rappel : la case (i, j) contient la distance d'édition entre les i premiers caractères de la première chaîne et les j premiers de la seconde.
2. En déduire la distance d'édition et les opérations effectuées.
3. Programmer la fonction `distance_edition(a, b)` qui retourne la distance d'édition entre deux chaînes `a` et `b`.
4. Tester avec les paires suivantes : `("CHAT", "CHATS")`, `("NICHE", "CHIEN")`, `("ALGORITHME", "ALTRUISME")`.

Exercice 6 *Montée d'escalier (synthèse)*

Un escalier comporte n marches numérotées de 1 à n. Un marcheur part du sol (marche 0) et peut monter d'une ou deux marches à chaque pas. On note $f(n)$ le nombre de façons d'atteindre la marche n.

1. Calculer à la main $f(1)$, $f(2)$, $f(3)$, $f(4)$ et $f(5)$.
2. Justifier que $f(n) = f(n-1) + f(n-2)$ pour $n \geqslant 3$, avec $f(1) = 1$ et $f(2) = 2$. Quelle suite connue reconnaît-on ?
3. Écrire une version récursive naïve `montee_recursive(n)`, puis la tester pour $n = 40$. Que constate-t-on ?
4. Écrire une version avec mémoïsation `montee_memo(n)`.
5. Écrire une version *bottom-up* `montee_dyn(n)`.
6. **Extension :** chaque marche a maintenant un coût c_i (l'énergie dépensée pour y poser le pied). On cherche à atteindre la marche n en minimisant le coût total. On note $C(i)$ le coût minimal pour atteindre la marche i. Donner la relation de récurrence, puis programmer la fonction `cout_minimal(couts)` où `couts` est la liste des coûts des marches de 1 à n.