
Contrôle Continu
Programmation Dynamique

Durée indicative de 60 minutes, durée précise indiquée au début du contrôle. Aucun document n'est autorisé.
Demandez toutes clarifications utiles à l'examinateur.
Barème indicatif : Exercice 1 sur 4 points ; Exercice 2 sur 12 points ; Exercice 3 sur 4 points.

Exercice 1. *Transformation d'un algorithme*
On considère la fonction f suivante. Ne cherchez pas à comprendre ce qu'elle calcule, remarquez seulement qu'elle est correctement définie, au sens où elle termine toujours. Soit n un entier fixé et // l'opérateur représentant la division entière.

Algorithme 1 Une implémentation d'une fonction inconnue f

Entrée: Un entier i tel que $0 \leq i \leq n$
  si $i = 0$ **alors**
    retourner 1
  sinon
    retourner $f(i//2) + f(i//3) + f(i-1)$
  fin si

1. Transformez l'implémentation pour utiliser la programmation dynamique.

Pour transformer l'implémentation pour utiliser la programmation dynamique, nous allons la mémoïsation qui consiste à stocker les résultats déjà calculés pour éviter de les recalculer à chaque fois :

```
# Initialisez un dictionnaire pour stocker les résultats déjà calculés
memo = {}

def f(i):
    # Vérifiez d'abord si le résultat pour i est déjà calculé
    if i in memo:
        return memo[i]

    # Appliquez l'algorithme d'origine si le résultat n'est pas déjà connu
    if i == 0:
        result = 1
    else:
        result = f(i // 2) + f(i // 3) + f(i - 1)

    # Stockez le résultat dans le dictionnaire
    memo[i] = result

    # Retournez le résultat
    return result
```

2. Quel est le type de complexité de votre algorithme utilisant la programmation dynamique en fonction de n ?

L'algorithme a une complexité de $O(n)$. Chaque valeur de $f(i)$ est calculée au plus (et au moins) une fois et les résultats sont stockés dans le dictionnaire *memo* pour les prochains appels.

Exercice 2. *Écriture d'un algorithme de programmation dynamique*
Soit n et m deux entiers positifs. On considère une grille de taille $n \times m$ avec la case de position $(0, 0)$ en haut à gauche et la case de position $(n-1, m-1)$ en bas à droite. On souhaite écrire un algorithme pour calculer le nombre de chemins possibles pour aller du point $(0, 0)$ au point $(n-1, m-1)$, en ne descendant que vers le bas ou vers la droite (pas de diagonale).

Problème #CheminGrille
Entrée : n un entier positif, m un entier positif
Sortie : Nombre de chemins possibles pour aller du point $(0, 0)$ au point $(n-1, m-1)$ sur la grille, en ne descendant que vers le bas ou vers la droite à chaque étape.

1. Proposez un algorithme récursif pour résoudre le problème #CheminGrille.

L'implémentation n'utilise pas pour le moment les principes de la programmation dynamique.

```
# Algorithme récursif pour le problème du nombre de chemins dans une grille
def nombre_chemins_grille(n, m):
    # Cas de base : une seule façon de se déplacer dans une grille 1x1
    if n == 1 or m == 1:
        return 1

    # Appels récursifs pour les déplacements vers le bas et vers la droite
    chemins_depuis_le_haut = nombre_chemins_grille(n - 1, m)
    chemins_depuis_la_gauche = nombre_chemins_grille(n, m - 1)

    # Le nombre total de chemins est la somme des chemins
    return chemins_depuis_le_haut + chemins_depuis_la_gauche
```

2. Modifiez l'algorithme précédent pour utiliser la programmation dynamique.

```
def nombre_chemins_grille_dp(n, m, memo=None):
    # Initialiser le tableau memo lors du premier appel
    if memo is None:
        memo = [[-1] * m for _ in range(n)]

    # Cas de base : une seule façon de se déplacer dans une grille 1x1
    if n == 1 or m == 1:
        return 1

    # Vérifier si le résultat pour ces dimensions est déjà calculé
    if memo[n - 1][m - 1] != -1:
        return memo[n - 1][m - 1]

    # Appels récursifs pour les déplacements vers le bas et vers la droite
    chemins_depuis_le_haut = nombre_chemins_grille_dp(n - 1, m, memo)
    chemins_depuis_la_gauche = nombre_chemins_grille_dp(n, m - 1, memo)

    # Le nombre total de chemins est la somme des chemins
    memo[n - 1][m - 1] = chemins_depuis_le_haut + chemins_depuis_la_gauche

    return memo[n - 1][m - 1]
```

3. Quelle est la complexité de votre algorithme utilisant la programmation dynamique ?

La complexité de l'algorithme est $O(n \times m)$.

4. Proposez une version itérative de l'algorithme, toujours en utilisant la programmation dynamique.

```python
def nombre_chemins_grille_dp_it(n, m):
    # Créer un tableau pour stocker les résultats intermédiaires
    memo = [[0] * m for _ in range(n)]

    # Remplir la première ligne et la première colonne avec des 1 (cas de base)
    for i in range(n):
        memo[i][0] = 1
    for j in range(m):
        memo[0][j] = 1

    # Remplir le tableau en utilisant la programmation dynamique
    for i in range(1, n):
        for j in range(1, m):
            memo[i][j] = memo[i - 1][j] + memo[i][j - 1]

    # Le résultat final est stocké dans le coin inférieur droit du tableau
    return memo[n - 1][m - 1]
```

5. Proposez un algorithme similaire qui retourne également les solutions.

C'est-à-dire la suite de direction à suivre. Par exemple, pour $n = m = 2$, les solutions seraient $[[\rightarrow, \downarrow], [\downarrow, \rightarrow]]$.

```python
def nombre_chemins_grille_dp_it(n, m):
    # Créer un tableau pour stocker les résultats intermédiaires
    memo = [[0] * m for _ in range(n)]

    # Créer un tableau pour stocker les chemins intermédiaires
    chemins [[] * m for _ in range(n)]

    # Remplir la première ligne et la première colonne avec des 1 (cas de base) et le chemin correspo
    for i in range(n):
        memo[i][0] = 1
        chemins[i][0] = [">" * i]
    for j in range(m):
        memo[0][j] = 1
        chemins[0][j] = ["v" * j]

    # Remplir le tableau en utilisant la programmation dynamique
    for i in range(1, n):
        for j in range(1, m):
            memo[i][j] = memo[i - 1][j] + memo[i][j - 1]

            # On ajoute la dernière direction (> ou v) à tous les chemins possible en haut et à gauch

            for chem in chemins[i - 1][j]
                chemins[i][j].add(chem + ">")
            for chem in chemins[i][j-1]
                chemins[i][j].add(chem + "v")

    # Le résultat final est stocké dans le coin inférieur droit du tableau
    return memo[n - 1][m - 1], chemins[n - 1][m - 1]
```

Exercice 3. *Retour sur le TP*

L'encadré suivant est un bref rappel du TP sur la segmentation d'image :

Soit deux entiers niv_x et niv_y dans $[0, 255]$, on définit la fonction $penalite(niv_x, niv_y)$:

$$penalite(niv_x, niv_y) = \begin{cases} 1000 & \text{si } |niv_x - niv_y| \leq 10 \\ 100 & \text{si } 10 < |niv_x - niv_y| \leq 35 \\ 10 & \text{si } 35 < |niv_x - niv_y| \leq 90 \\ 1 & \text{si } 90 < |niv_x - niv_y| \leq 210 \\ 0 & \text{sinon} \end{cases} \quad (1)$$

Nous rappelons brièvement le problème de segmentation d'image vu dans le TP :

Problème SegmentationImage
Entrée : · im une image en niveaux de gris stockée sous la forme d'une grille de pixels. Tout pixel de coordonnée (i, j) est un entier dans $[0, 255]$ avec 0 correspondant à un pixel noir, et 255 un pixel blanc.
· f et b deux listes de pixels qui sont imposés respectivement appartenant au *premier-plan* et l'*arrière-plan*, avec $f \cap b = \emptyset$.
Sortie : Un sous-ensemble de pixels B tel que $b \subseteq B$ et $f \subseteq \mathcal{P} \setminus B$ et tel que $c(B)$ soit minimum avec :

$$c(B) = \sum_{\text{Pixel de coordonnée } (i,j) \in B \text{ et pixel } (i',j') \notin B} penalite(im[(i, j)], im[(i', j')])$$

Nous rappelons également brièvement le problème de SegmentationImageGraphe vu dans le TP :

Problème SegmentationImageGraphe
Entrée : · g un graphe orienté avec des capacités sur les arcs. On note $cap(u, v)$ la capacité de l'arc $u \to v$.
· f et b deux sous ensembles de sommets de g, avec $f \cap b = \emptyset$
Sortie : Un sous ensemble de sommets B tel que $b \subseteq B$ et $f \subseteq V(g) \setminus B$ et tel que $c_2(B)$ soit minimum avec :

$$c_2(B) = \sum_{(u,v) \in \{(u,v) \text{ un arc de } g \text{ tel que } u \in B \text{ et } v \notin B\}} cap(u, v)$$

1. Donnez en quelques mots le "lien" qui existe entre le problème SegmentationImage et le problème SegmentationImageGraphe que nous avons vu lors du TP.

Il existe une **réduction** polynomiale du problème SegmentationImage au problème SegmentationImageGraphe. *[Cela signifie qu'il existe une "transformation" efficace d'une instance du premier problème vers une instance "équivalente" du second problème.]*

2. Supposons que l'on ait une implémentation efficace pour résoudre n'importe quelle instance du problème SegmentationImageGraphe. Proposez une méthode permettant de résoudre une instance du problème SegmentationImage.

Détaillez l'ensemble de la méthode : une personne qui ne connait aucun de ces problèmes doit être capable de résoudre une instance de SegmentationImage avec seulement un algorithme pour résoudre une instance du problème SegmentationImageGraphe en suivant vos étapes à la lettre.

→ Voir le TP.