
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. *Complexité d'un algorithme de programmation dynamique*
On considère la fonction g 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 et m deux entiers fixés.

Algorithme 1 Une implémentation d'une fonction inconnue g

Entrée: Deux entiers m et n tels que $m \geq 0$ et $n \geq 0$
  si $m = 0$ ou $n = 0$ **alors**
    retourner 1
  sinon
    retourner $g(m-1,n) + g(m,n-1) + g(m-1,n-1)$
  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 :

```
# Créer une table pour stocker les résultats intermédiaires
memo = [[-1] * (n + 1) for _ in range(m + 1)]

# Fonction récursive avec mémoïsation
def g_dp(m, n):
    # Cas de base pour m = 0 ou n = 0
    if m == 0 or n == 0:
        return 1

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

    # Sinon calculer le résultat en utilisant la récursion
    result = g_dp(m - 1, n) + g_dp(m, n - 1) + g_dp(m - 1, n - 1)

    # Stocker le résultat dans la table
    memo[m][n] = result

    return result
```

2. Quelle est la complexité de votre algorithme utilisant la programmation dynamique en fonction de m et n ?

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

Exercice 2. *Écriture d'un algorithme de programmation dynamique*
Soit n un entier positif. On considère le nombre de partitions de n, c'est-à-dire le nombre de façons d'écrire n comme une somme d'entiers positifs. Dans cet exercice, nous décidons de **tenir compte de l'ordre** ("1+2" n'est pas la même somme que "2+1"). Par exemple, il y a 8 partitions pour $n = 4$:

"4", "3 + 1", "2 + 2", "2 + 1 + 1", "1 + 3", "1 + 2 + 1", "1 + 1 + 2", "1 + 1 + 1 + 1"

On souhaite écrire un algorithme pour calculer le nombre de partitions de n.

> **Problème #Partition**
> **Entrée :** n un entier positif
> **Sortie :** Le nombre de partitions de n (en tenant compte de l'ordre dans la somme)

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

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

```
def nombre_partitions_recursive(n):
    # Cas de base pour n = 0 (on fixe cela par convention : pratique pour la suite)
    if n == 0:
        return 0

    # Initialiser le nombre de partitions pour n
    # On compte d'abord la partition avec un seul élément : [n])
    partitions = 1

    # Explorer les différentes façons de partitionner n
    for i in range(1, n):
        partitions += nombre_partitions_recursive(n - i)

    return partitions
```

Cet algorithme explore toutes les façons de partitionner n en fixant dans la boucle le premier nombre i de la somme est en appelant récursivement $n - i$.

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

```
# Initialiser un tableau pour stocker les résultats intermédiaires
memo = [None] * (n + 1)

# Fonction récursive avec mémoïsation
def partitions_dp(n):
    # Cas de base : une seule partition pour n = 0
    if n == 0:
        return 1

    # Vérifier si le résultat est déjà calculé
    if memo[n] is not None:
        return memo[n]

    partitions = 1
    for i in range(1, n):
        partitions += nombre_partitions_recursive(n - i)

    # Stocker le résultat dans le tableau memo
    memo[n] = partitions
    return partitions
```

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

La complexité de l'algorithme est de $O(n^2)$ avec n cases de `memo` à calculer en un temps $O(n)$. Ce résultat est semblable à l'algorithme de "découpe" vu en TD (d'ailleurs, l'exercice entier l'est).

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

```
def nombre_partitions_iterative(n):
    # Initialiser un tableau pour stocker les résultats intermédiaires
    memo = [1] * (n + 1)

    # Il y a une seule partition pour n=0
    memo[0] = 1

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

    # Le résultat final est stocké dans la dernière case du tableau
    return memo[n]
```

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

Par exemple, pour $n = 4$*, les solutions seraient* [″4″, ″3 + 1″, ″2 + 2″, ″2 + 1 + 1″, ″1 + 3″, ″1 + 2 + 1″, ″1 + 1 + 2″, ″1 + 1 + 1 + 1″]*.*

```
def partitions_avec_solutions(n):
    # Initialiser un tableau pour stocker les résultats intermédiaires
    memo = [[str(i)] for i in range(n + 1)]

    # Par convention, il y a une seule solution pour n = 0 (la chaîne vide)
    memo[0] = [""]

    # Remplir le tableau en utilisant la programmation dynamique
    for i in range(1, n + 1):
        for j in range(1, i):
            for solution in memo[i - j]:
                memo[i].append(solution + "+" + str(j))

    # Le résultat final est stocké dans la dernière case du tableau
    return memo[n]
```

Un appel à `"(partitions_avec_solutions(4))"` affiche :
`['4', '3+1', '2+1+1', '1+1+1+1', '1+2+1', '2+2', '1+1+2', '1+3']`

Exercice 3. *Retour sur le TP*

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

Nous rappelons 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 $f \subseteq V(g) \backslash B$ et tel que $c(B)$ soit minimum avec :
>
> $$c(B) = \sum_{(u,v) \in \{(u,v) \in E(g) | u \in B \text{ et } v \notin B\}} cap(u, v)$$

Nous rappelons également le problème de MinCut vu dans le TP :

> **Problème MinCut**
> **Entrée :** Un triplet (g,t,s) avec g un graphe orienté avec des capacités sur les arcs ($cap(u, v)$ est la capacité de l'arc $u \to v$) et s et t deux sommets de g, avec $s \neq t$.
> **Sortie :** Un sous-ensemble de sommets B tel que $s \in B$ et $t \notin B$ et tel que $c(B)$ soit minimum avec :
>
> $$c(B) = \sum_{(u,v) \in \{(u,v) \in E(g) | u \in B \text{ et } v \notin B\}} cap(u, v)$$

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

Il existe une **réduction** polynomiale du problème SegmentationImageGraphe au problème MinCut. *[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 MinCut. Proposez une méthode permettant de résoudre une instance du problème SegmentationImageGraphe.

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

→ Voir le TP.