Chapitre 14 : Programmation dynamique

Table des matières

Énoncé : Exemple de découpe de barres

Cet exemple est largement inspiré d'un passage du livre : Algorithmique de Cormen, Leiserson, Rivest, Stein.

Une entreprise achète des barres d'acier et les recoupe pour ensuite les revendre. Les découpes effectuées sont toujours un nombre entier de mètres. Le prix de revente des ces barres est le suivant :

longueur i (en m)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
prix pi (en €)	1	5	8	9	10	17	17	20	24	30

La direction de l'entreprise veut connaître la meilleur façon de couper les barres afin d'obtenir un profit maximum.

partie A : Algorithme glouton

Exercice A1 : Tableau des densités

Une barre de 3m étant vendu 8€, on gagne 2,67 €/m.

Dans l'exemple donné, on peut calculer pour chaque longueur de barre, le nombre d'euros par mètre.

Compléter le tableau ci-dessous :

longueur i (en m)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
prix pi (en €)	1	5	8	9	10	17	17	20	24	30
rentabilité €/m	1	2,5	2,67	2,25						

Lever la main pour valider cet exercice

Exercice A2 : Expérimentation

Exemple : Si l'on dispose d'une barre de 4m.

On note que parmi toutes les découpes possibles 1m, 2m, 3m, 4m, celle qui rapporte le plus au mètre est celle de 3m.

On décide donc de la couper en une sous barre de 3m et il reste une barre de 1m que l'on ne peut pas redécouper, avec cette méthode on gagnera donc :

p3 + p1 = 8 + 1 = 9€

On applique là un algorithme glouton. Un algorithme glouton est un algorithme qui cherche a augmenté le gain immédiat. On prends ce qui rapporte le plus au moment actuel.

Question : Appliquer la même démarche pour une barre de 8m et donner le gain correspondant.

Lever la main pour valider cet exercice

Exercice A3 : Pseudo-code

Compléter pseudo code ci-dessous

```
prix = [0, 1, 5, 8 ...]

fonction rentabilite(l)

""" fonction qui renvoie le nombre d'euros par mètre pour une barre de longueur l """

...



fonction longueur_gain_max(longueur)

""" fonction qui renvoie la sous-longueur pour la laquelle la rentabilité est maximale """

...



fonction coupe_glouton(longueur)

 """ fonction qui applique l'algorithme glouton et renvoie le gain calculé """

gain = 0

tant que longueur > 0

    longueur_coupe = ....

    gain = gain + ...

    longueur = longueur - longueur coupe

renvoyer gain
```

Lever la main pour valider cet exercice

Exercice A4: Implémentation en python

Créer un fichier nommé coupe_barre.py.

Programmer en python les 3 fonctions vues dans le pseudo-code.

Faire un test avec une barre de 4m et de 8m et vérifier que vous obtenez les même résultats que ceux vus dans l'exercice 2.

Lever la main pour valider cet exercice

Partie B Programmation dynamique - structure optimale - récursivité

Exercice B1 : Structure optimale

1. On considère une barre de 4m.

   a) Déterminer toutes les différentes façons de la couper et pour chaque façon donner le total des prix de vente de chaque morceau.

   b) En déduire le gain optimal que l'on peut espérer avec une barre de 4m.

   c) Que remarquez-vous par rapport à l'algorithme glouton

2. Recommencer mais cette fois ci en considérant une barre de 8m.

Lever la main pour valider cet exercice

Explication sur la recherche récursive de la solution

On cherche à établir un algorithme récursif.

Pour une barre de 4m, on veut faire un découpage qui rapporte le maximum d'argent.

On note v_opt(4) , la valeur optimale que l'on peut tirer d'une barre de 4m.

On peut considérer 4 cas :

a) Soit la découpe contient comme première barre, une barre de 1m, et dans ce cas là, il restera une barre de 3m dont on cherchera la valeur optimale que l'on peut en tirer.

Dans ce cas v_opt(4) = p1 + v_opt(3)

b) Soit la découpe contient comme première barre, une barre de 2m, et dans ce cas là, il restera une barre de 2m dont on cherchera la valeur optimale que l'on peut en tirer.

Dans ce cas v_opt4) = p2 + v_opt(2)

c) Soit la découpe contient comme première barre, une barre de 3m, et dans ce cas là, il restera une barre de 1m dont on cherchera la valeur optimale que l'on peut en tirer.

Dans ce cas v_opt(4) = p3 + v_opt(1)

d) Soit la découpe contient comme première barre, une barre de 4m, et dans ce cas là, il n'y a pas de découpe. Pour simplifier la suite on considérera qu'il reste une barre de 0m rapportant 0€.

Dans ce cas v_opt(4) = p4 + v_opt(0)

La valeur optimale que l'on peux tirer d'une barre de 4m est la plus grande des 4 valeurs obtenues.

v4 = maximum (p1 + v_opt(3) , p2 + v_opt(2), p3 + v_opt(1), p4 + v_opt(0)).

On peut noter cela d'une autre façon :

v_opt(4) = Max (pi + v_opt(4-i)) pour i variant de 1 à 4

On remarque que pour trouver une solution optimale, il faut résoudre des sous problèmes de même nature mais de taille plus petite.

Exercice B2 : Pseudo-code

Dans le pseudo-code :

- on ne se limite pas à une barre de 4m.
- on note coupe_r(longueur) la fonction qui retourne valeur optimal que l'on peut tirer d'une barre d'une longueur donnée.

Compléter pseudo code ci-dessous

```
prix = [0, 1, 5, 8 ...]

fonction coupe_r(longueur)

si longueur = 0

alors

    .....

sinon

    vmax = -1

    pour i variant de 1 à longueur

        vmax = max( vmax, prix(i) + coupe_r(longueur - i))

renvoyer .…
```

☝ **Lever la main pour valider cet exercice**

Exercice B3 Implémentation en python

Compléter le fichier nommé coupe_barre.py.

Programmer en python la fonction coupe_r vue dans le pseudo-code.

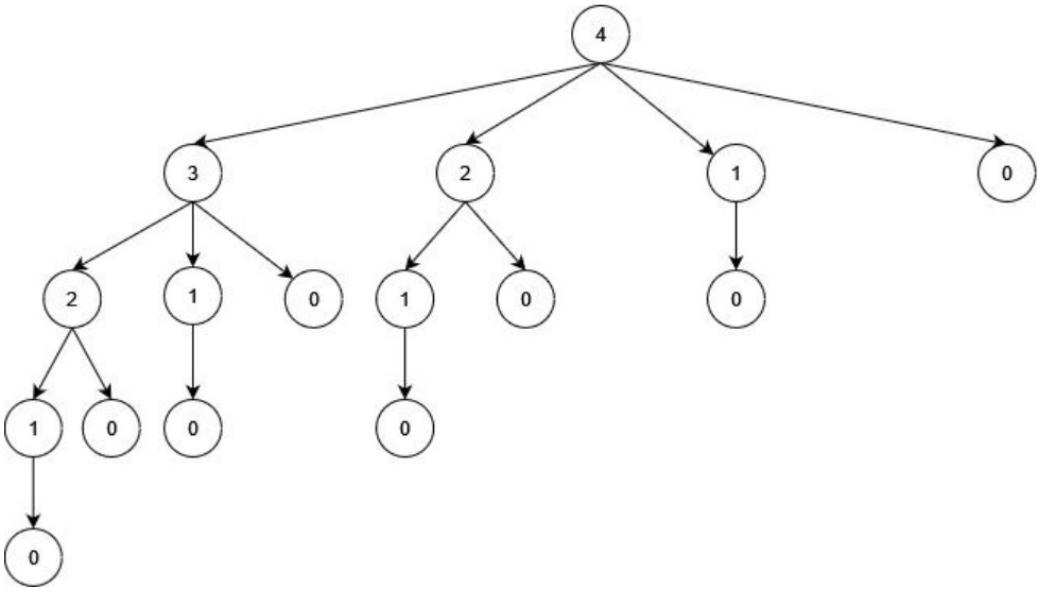
Faire un test avec une barre de 4m et de 8m et vérifier que vous obtenez les même résultats que ceux vus dans l'exercice 1.

☝ **Lever la main pour valider cet exercice**

Partie C Programmation dynamique – efficacité - mémoïsation

Exercice C1 : Arbre des appels

Voici un arbre des différents appels à la fonction coupe_r lorsqu'on exécute la fonction coupe_r(4)



1. Combien de fois est appelée la fonction coupe_r ?
2. Combien y a t-il d'appels redondants (qui se répètent) ?

Lever la main pour valider cet exercice

Exercice C2 : Efficacité

Beaucoup de sous problèmes se chevauchent.

On constate dans la partie ci-dessus, qu'on recalcule un grand nombre de fois la même chose.

On choisit maintenant des barres mesurées jusqu'à 40m.

prix = [0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30, 36, 39, 40, 42, 42, 44, 45, 46, 47, 49, 50, 50, 51, 52, 53, 53, 54, 55, 55, 55, 56, 56, 57, 57, 58, 58, 58, 58, 59, 60]

1. Lancer la fonction coupe_r(23). Que constatez-vous ?
2. Ajouter un compteur comme variable globale et initialiser le à 0.
3. Incrémenter ce compteur à chaque appel de la fonction et afficher le.
4. Vérifier que ce compteur renvoie pour coupe_r(4) la valeur trouvée dans Ex1.1
5. Quelle est la valeur de ce compteur après l'appel de coupe_r(10) ?
6. D'après-vous, comment s'expriment la complexité de cet algorithme en fonction de la longueur de la barre ?

Lever la main pour valider cet exercice

Principe de mémoïsation

Dans l’arbre précédent on constate que coupe_r(1) est calculé 4 fois alors qu’il suffirait de le calculer une fois pour toute et de stocker l’information.

C’est le principe de **mémoïsation**.

Nous allons donc tout stocker dans un tableau.

- Si une valeur de coupe_r a déjà été stockée dans le tableau, on ne la recalculera pas.
- Si une valeur de coupe_r n’a pas été stockée dans le tableau, on la calculera puis on la stockera dans le tableau pour la suite.

Pseudo-code

On utilise un tableau comme variable globale de taille longueur + 1.

On mets -1 dans chaque case pour dire que la valeur n’a pas été calculée.

```
tableau = [-1, -1, -1, -1.......]

fonction coupe_memo(longueur)

si coupe_memo(longueur) ≠ -1

alors

    renvoyer tableau(longueur)

sinon

    vmax = -1

    pour i variant de 1 à longueur

        vmax = max( vmax, prix(i) + coupe_r(longueur - i))

renvoyer vmax
```

Exercice C3 Implémentation en python

Compléter le fichier nommé coupe_barre.py.

1. Programmer en python la fonction coupe_memo vue dans le pseudo-code.
2. Faire un test avec une barre de 23m et de 40m.
3. D'après-vous, comment s'expriment la complexité de cet algorithme en fonction de la longueur de la barre ?
4. Récupérer en ressource le fichier generateur.py et copier son contenu à l'intérieur de coupe_barre.py.
5. Ajouter l'instruction prix = genere_prix(100) pour avoir une liste de prix pour des barres allant jusqu'à 100m. Lancer coupe_memo(100).
6. Pour étendre la taille de la pile d'appel, ajouter les deux instructions suivantes au début de votre programme :

```
from sys import setrecursionlimit
setrecursionlimit(10000)
```

   Expérimenter pour voir jusqu'à quelle taille de barre l'algorithme coupe_memo fonctionne dans un temps raisonnable (moins de 30s).

Lever la main pour valider cet exercice

Partie D Programmation dynamique - ascendante(bottom up)

Principe de l’approche ascendante

Si on observe l’arbre vu dans la partie C Ex1, on remarque que pour calculer la valeur optimale pour une barre de 4m, il est nécessaire de déterminer la valeur optimale pour 1m, 2m et 3m. Le principe de mémoïsation permet d’éviter de calculer ces dernières plusieurs fois.

L’approche ascendante consiste à partir du bas de l’arbre pour aller vers le haut (bottom-up).

On résout d’abord le problème pour une barre de 1m, puis pour 2m, puis 3m puis enfin 4m et on stocke à chaque étape les résultats obtenus.

On résout les problèmes par ordre croissant de taille et à chaque fois on s’appuie sur les résultats précédents qui on été stockés.

Comme pour la méthode avec mémoïsation, on doit calculer tous les valeurs intermédiaires. En revanche, on procède de façon itérative, il n’y a pas de récursivité et on n’est pas limité par la taille de la pile d’appel des fonctions.

Pseudo-code

On utilise un tableau comme variable globale de taille longueur + 1.

On mets -1 dans chaque case pour dire que la valeur n’a pas été calculé.

La première case contient un “0” c’est l’initialisation car une barre de 0m rapporte 0€.

```
tableau = [0, -1, -1, -1.......]

fonction coupe_ascendante(longueur)

pour j variant de 1 à longueur

    vmax = -1

        pour i variant de 1 à j

            vmax = max( vmax, prix(i) + coupe_r(longueur - i))

        tableau(j) = vmax

renvoyer(tableau(longueur))
```

Exercice D1 Implémentation en python

Compléter le fichier nommé coupe_barre.py.

1. Programmer en python la fonction coupe_ascendante vue dans le pseudo-code.
2. Faire un test avec une barre de 23m et de 40m.
3. D'après-vous, comment s'expriment la complexité de cet algorithme en fonction de la longueur de la barre ?
4. Réutiliser la fonction genere_prix vue dans la partie C et ajouter l'instruction :

   prix = genere_prix(100) pour avoir une liste de prix pour des barres allant jusqu'à 100m.

   Lancer coupe_ascendante(100).
5. Expérimenter pour voir jusqu'à quelle taille de barre l'algorithme coupe_ascendante fonctionne dans un temps raisonnable (moins de 30s).

On admettra que la complexité de la fonction coupe_ascendante est la même que celle de la fonction coupe_memo.

Lever la main pour valider cet exercice

Partie E programmation dynamique - construction de la solution

Solution complète

Les différents fonctions coupe_r, coupe_memo et coupe_ascendante retournent la valeur optimale que l'on peut tirer d'une barre mais elle ne donne pas le découpage correspondant.

exemple :

coupe_ascendante(4) renvoie 10 cela signifie que l'on peut tirer 10€ d'une barre de 4m mais cela ne précise pas qu'il faut faire la première découpe à 2m.

On peut étendre l'approche de façon à enregistrer non seulement la valeur optimale pour chaque sous-problème mais aussi une solution de découpage correspondant (Cette solution peut ne pas être unique).

Pseudo-code

On utilise un second tableau nommé coupure comme variable globale de taille longueur + 1

qui pour chaque longueur associera la longueur de la première découpe à faire pour obtenir un une valeur optimale.

Ex : coupure(4) = 2.

Et on initialise coupure(0) à [0] : une barre de 0m ne peut être coupé qu'à 0m.

```
tableau = [0, -1, -1, -1.......]

coupure = [[0]]

fonction coupe_ascendante_solution(longueur)

pour j variant de 1 à longueur

    vmax = -1

        pour i variant de 1 à j

            vmax = max( vmax, prix(i) + coupe_r(longueur - i))

            coupure(j) = i

        tableau(j) = vmax

renvoyer(tableau,coupure)

fonction affiche_solution(longueur)

(t,c) = coupure_solution(longueur)

tant que longueur > 0

    afficher(c(longueur))

    longueur = longueur - c(longueur)
```

Avec l'exemple de prix initial, les informations stockés en mémoire seraient :

longueur i (en m)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
tableau (valeur optimale)	0	1	5	8	9	10	17	17	20	24	30
coupure	0	1	2	3	2	2	6	1	2	3	10

Exercice E1 Implémentation en python

Compléter le fichier nommé coupe_barre.py.

1. Programmer en python la fonction coupe_ascendante_solution et affiche_solution vue dans le pseudo-code.
2. Faire un test avec une barre de de 4m, de 23m et de 40m.

   Lever la main pour valider cet exercice

Partie F Synthèse sur la programmation dynamique

La programmation dynamique, comme la méthode diviser pour régner, résout des problèmes en combinant des solutions de sous problèmes.

Les algorithmes diviser pour régner partitionnent le problème en sous problème indépendants qu'ils résolvent récursivement, puis combinent leurs solutions pour résoudre le problème initial.

La programmation dynamique peut s'appliquer même lorsque les sous problèmes se recoupent, c'est à dire lorsque les sous problèmes ont des sous problèmes communs.

Un algorithme de programmation dynamique résout chaque sous-problème une seule fois et mémorise sa réponse, évitant de calculer plusieurs fois la même chose.

La programmation dynamique s'applique généralement aux problèmes d'optimisation.

Le développement d'un algorithme de programmation dynamique se divise en quatre étapes :

1. Caractériser la structure d'une solution optimale.
2. Définir récursivement la structure d'une solution optimale.
3. Calculer la valeur d'une solution optimale (généralement de façon ascendante).
4. Construire une solution optimale à partir des informations calculées.

Remarque : l'étape 4 nécessite parfois que l'on ajoute des informations supplémentaires dans l''étape 3.

