Chapitre 10 000

Programmation dynamique

Dans ce Cours-TD, nous allons découvrir les rouages de la programmation dynamique, notamment en termes de complexité. Élégante et surtout performante, cette façon de programmer est particulièrement recherchée lors des concours de recrutement en Info.

#Récursivité #TopDown #BottomUp #TOBLERONE© #Complexité

Introduction : à compléter, au tableau

On se contente ici de partager une barre de Toblerone de manière optimale, puis on la mange.

Exercice 1 : Comment partager une barre en acier de manière optimale ?

Cet exercice est un classique du genre (CORMEN, Algorithmique, MIT). Une entreprise du secteur de l'acier coule de longues barres en acier de longueur L. Elle peut alors vendre à ses clients des *morceaux* de barre découpés sur mesure. Pour simplifier et surtout pour satisfaire les plus #gourmands d'entre vous, la barre en acier sera modélisée par une barre de #TOBLERONE©. Cette barre est composée de n sections découpables :

Mais voilà, le prix des barres n'est pas proportionnel à la longueur des morceaux de barre ! C'est la loi du marché ! La liste de prix qui suit donne le prix actuel du marché, susceptible d'être modifiée chaque jour :

Taille	1	2	3	4	5	6	7	8	9	10
Prix	1	5	8	9	10	17	17	20	24	26

Dans cet exercice n désigne le nombre total de morceaux sectionnables de la barre chocolatée.

Préliminaire

- On se place dans le cas où $n = 4$. Envisager toutes les manières de couper la barre ou de ne pas la couper (ce peut être le choix à faire). Des solutions identiques peuvent apparaître.
- Déterminer la solution optimale en termes de revenu pour chaque $n \in \{1, 2, 3, 4\}$.
- Vous avez devant vous une barre de longueur $n = 10$: déterminer les coupes nécessaires pour tirer un revenu optimal de cette barre. Vous pouvez également décider de la laisser entière.

Illustration 1 : Cette découpe est-elle optimale ?

PARTIE A : une solution récursive bien peu optimisée… #ForceBrute

Q1 : implémentation naïve

On envisage ici une implémentation descendante récursive naïve (de type *top down*).

> → Pour résoudre le problème de taille n, on résout des problèmes de même type mais de taille inférieure. Après une première coupe, on peut considérer les deux morceaux comme des instances indépendantes du problème de coupage de barre. La solution optimale globale incorpore les solutions optimales des deux sous problèmes associés, lesquelles maximisent le revenu de chacun de ces deux morceaux. On dit que le problème de coupage de barre exhibe une *sous-structure optimale*.

La solution récursive naïve proposée ici consiste à découper un premier morceau de taille i, situé à gauche de la barre de taille n, le « reste » étant alors de longueur $(n - i)$. On recoupe ensuite uniquement le « reste », situé à droite.

En notant $revenu(n)$ la valeur optimale de la découpe d'une barre de taille n et en posant $revenu(0) = 0$, on obtient l'équation suivante :

$$revenu(n) = \max_{1 \le i \le n} (coût(i) + revenu(n - i))$$

- Écrire une fonction `maxi(a,b)` qui retourne le maximum de deux entiers a et b, donnés en paramètres.
- Implémenter en Python une solution récursive `decoupeBarre(listePrix,n)` qui renvoie le revenu optimal, basé sur l'équation donnée ci-dessus.
  Ainsi `decoupeBarre(listePrix,10)` doit renvoyer 27. On remarquera qu'on ne demande pas ici de renvoyer la séquence de découpage optimale, mais uniquement le revenu optimal.

Aide n° 1 : c'est une option !
(Tu la prends ou tu ne la prends pas 😉)
L'algorithme en pseudo-code est donné. Attention, les valeurs des indices doivent être adaptés à votre langage de programmation préféré !

- À l'aide d'un chronomètre perf_Counter (voir TD sur les tri) mesurer le temps de calcul dans les cas n=5 et n=10.

Q2 :

- D'où provient l'inefficacité de la fonction `decoupeBarre(listePrix,n)` ?
- Illustrer ceci avec un $arbre\ récursif$ (voir intro du cours) pour $n = 4$ montrant tous les appels récursifs. Chaque sommet de l'arbre représentera un appel récursif et chaque feuille un appel récursif « terminal ».
- Vérifier que cet arbre comporte bien 2^n sommets et 2^{n-1} feuilles.

Q3 : Complexité

Soit $T(n)$ le nombre d'appels récursifs à `decoupeBarre(listePrix,n)`. Cette expression est égale au nombre de nœuds d'un sous arbre dont la racine est étiquetée n dans l'arbre récursif. On a ainsi $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$. Montrer que $T(n) = 2^n$.

PARTIE B : Programmation dynamique

Peut-on faire mieux ? Oui ! En effet, les sous problèmes de taille inférieure sont ici identiques : il y a *redondance*. On va s'arranger pour que chaque sous-problème ne soit traité qu'une seule fois, en mémorisant sa solution, dans une liste dédiée.

→ COURS :

La programmation dynamique s'applique à des problèmes d'optimisations ayant les deux propriétés suivantes :

- une sous-structure optimale : une solution contient des solutions à des instances plus petites du même problème ;
- un chevauchement des sous-problèmes : une solution récursive en suivant le principe du «diviser pour régner» conduit à résoudre plusieurs fois les mêmes sous-problèmes.

Il existe deux facettes **équivalentes** de la programmation dynamique :

- La programmation descendante *récursive* avec mémoïsation appelée ***top-down*** ;
- La programmation ascendante *itérative*, appelée ***bottom up***.

Ces deux versions ont très souvent le même temps d'exécution asymptotique.

→ COURS : Top-down ou bottom-up ?

Une fois que l'on a identifié une relation de récurrence se prêtant bien à une solution par programmation dynamique, on a deux choix :

- l'approche de bas en haut (bottom-up) consiste (en gros) à résoudre d'abord toutes les instances de taille 1, puis celles de taille 2 et ainsi de suite jusqu'à celle qui nous intéresse. Typiquement, on obtient une solution itérative ;
- l'approche de haut en bas (top-down) consiste à traduire directement la relation de récurrence (à l'aide d'une fonction récursive) et à mémoïser les appels. En règle générale, **l'approche** top-down est plus simple à écrire (et donc à privilégier, sauf si l'on s'interdit la récursivité).

Au niveau des performances (et surtout de la complexité en espace), tout dépend du problème. S'il est simple de déterminer exactement quels calculs vont être nécessaires, l'approche bottom-up peut être intéressante.
En particulier, on peut dans certains cas ne garder en mémoire que les résultats récents (ceux correspondant à des instances de taille n−1), et donc faire passer la complexité en espace de $O(n^2)$ à $O(n)$ (ou de $O(n)$ à $O(1)$, ou …). En revanche, s'il n'est pas évident de déterminer a priori quelles instances il va réellement falloir résoudre, l'approche top-down peut éviter des calculs inutiles.

Q4 : Un essai Top down

Écrire une fonction topDown(listePrix,n), une version mémoïsée de decoupeBarre(listePrix,n), qui renvoie le revenu optimal de la vente en morceaux d'une barre de taille n.

Indication :

- On pourra initialiser une liste appelée memo de 20 (ou plus) valeurs égales à -100, qui servira à stocker les valeurs optimales des revenus pour les différentes valeurs de n. La valeur -100 indiquera que la valeur optimale du revenu n'est pas déjà connue, et qu'il faut donc la calculer, puis la stocker dans la liste « mémo ».
- On passera la liste memo en argument afin de permettre sa mise à jour « on the fly ».
- Attention à ne pas réinitialiser la liste memo à chaque tour !!!

Aide n° 2 : c'est encore une option ! 😉
L'algorithme en pseudo-code est donné. Attention, les valeurs des indices doivent être adaptés à votre langage de programmation préféré !

Q5 : Un essai Bottom Up

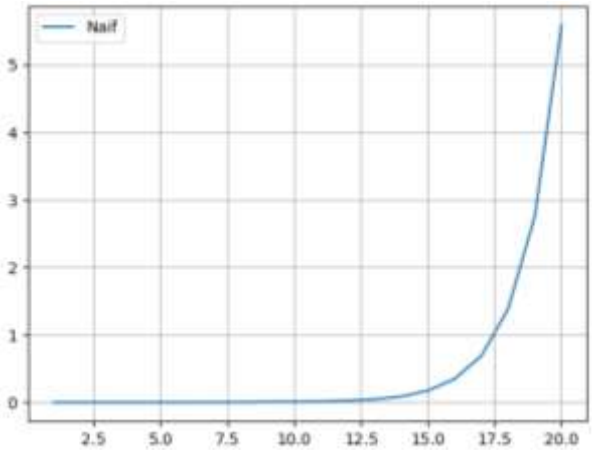
Prenons le problème à l'envers et abandonnons l'idée de récursivité.
On va ici déterminer le revenu optimal en prenant les barres une par une *par ordre croissant de taille*. On détermine ainsi le revenu pour une barre de taille j, sachant que l'on connaît déjà la solution optimale pour la barre de longueur $(j-1)$. On progresse ainsi par taille croissante jusqu'à atteindre la taille de barre souhaitée.

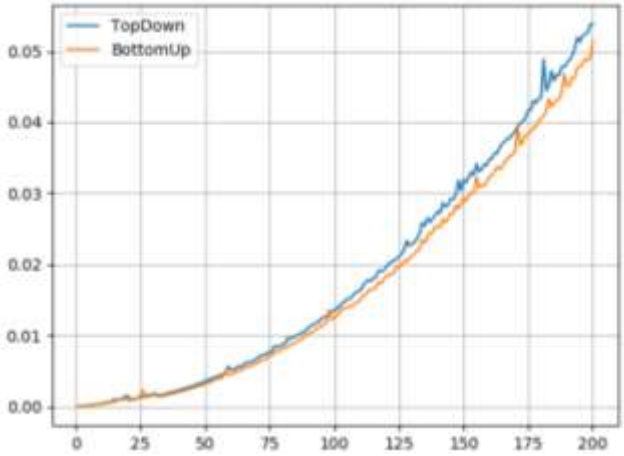
Écrire une fonction BottomUp(listePrix,n), non récursive qui renvoie le revenu optimal de la vente en morceaux d'une barre de taille n.

Indication : un **for** peut en cacher un autre **!**

Aide n° 3 : c'est encore une option ! 😉
L'algorithme en pseudo-code est donné. Attention, les valeurs des indices doivent être adaptés à votre langage de programmation préféré !

Prolongement : les listes de prix fluctuent ici selon les cours de la bourse. La liste de prix est donc modifiée à chaque calcul. On a mesuré la durée d'exécution des trois algorithmes Naïf, TopDown et BottomUp pour n variant de 0 à 200. (Graphes S. CHAMBEU).
Les résultats sont-ils cohérents ?





Exercice 2 : Fibonacci par ci, Fibonacci par là …

La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle doit son nom à Leonardo Fibonacci qui, dans un problème récréatif posé dans l'ouvrage « Liber abaci » publié en 1202, décrit la croissance d'une population de lapins.

Cette suite est fortement liée au nombre d'or. Ce nombre intervient dans l'expression du terme général de la suite. Inversement, la suite de Fibonacci intervient dans l'écriture des réduites de l'expression de φ en fraction continue : les quotients de deux termes consécutifs de la suite de Fibonacci sont les meilleures approximations du nombre d'or.

On propose dans cet exercice d'écrire quatre versions différentes d'une fonction retournant la valeur du n[ième] terme de la suite de Léonard de Pise (alias Fibonacci).
La suite de Fibonacci est définie ainsi :

$$\begin{cases} U_0 = 1 \\ U_1 = 1 \\ U_{n+2} = U_n + U_{n+1} \quad \text{pour tout } n \in \mathbb{N} \end{cases}$$

Les termes de cette suite sont appelés nombres de Fibonacci (suite A000045 de l'OEIS) :

$\mathcal{F}_0$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\mathcal{F}_4$	$\mathcal{F}_5$	$\mathcal{F}_6$	$\mathcal{F}_7$	$\mathcal{F}_8$	$\mathcal{F}_9$	$\mathcal{F}_{10}$	$\mathcal{F}_{11}$	$\mathcal{F}_{12}$	$\mathcal{F}_{13}$	$\mathcal{F}_{14}$	$\mathcal{F}_{15}$	$\mathcal{F}_{16}$	...	$\mathcal{F}_n$
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	...	$\mathcal{F}_{n-1} + \mathcal{F}_{n-2}$

Q1 : Version 1 itérative # PourquoiPas ?

Écrire une fonction naïve itérative `fibo1ter(n)` qui retourne la valeur de U_n.

- Évaluer sa complexité.

Q2 : Version 2 récursive # ForceBrute

- Écrire une fonction récursive `fiboRec(n)` qui retourne la valeur de U_n.
- Construire l'arbre des appels récursifs pour $n = 4$
- Évaluer sa complexité.

Q3 : Version 3 descendante récursive avec mémoïsation (Top Down !) # C'estJoli

- Écrire une fonction récursive `fiboRecMemoTopDown(n)` qui renvoie U_n en sauvegardant dans une liste appelée **memo** les termes de la suite déjà calculés.

Q4 : Version 4 ascendante sans récursivité (Bottom Up !) # C'estCool !

- Écrire une fonction récursive `fiboMemoBottomUp(n)` qui calcule de manière ascendante.

Q5 : Comparaison des quatre versions

Comparer les temps de calcul des quatre versions pour calculer U_{34} On utilisera la méthode time.perf_counter() du module time, vue lors du TD sur le tri rapide.

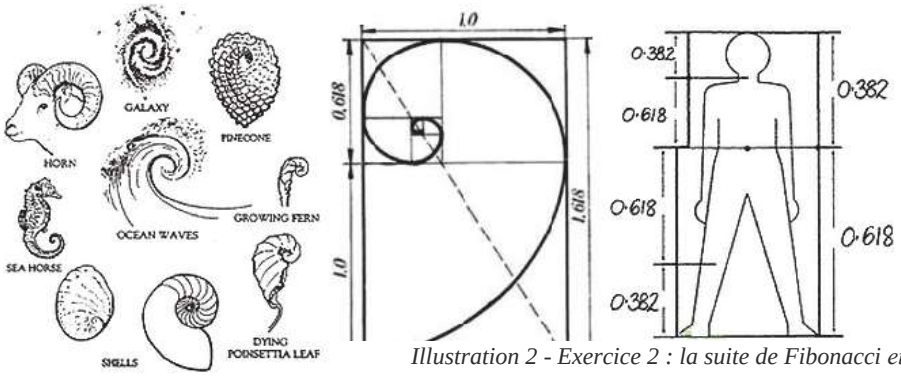


Illustration 2 - Exercice 2 : la suite de Fibonacci et le nombre d'or dans la nature

Exercice 3 : Optimisation d'un chemin dans une matrice

Les problèmes d'optimisation dynamique ont des applications importantes aussi bien dans l'industrie qu'en gestion. Il s'agit de minimiser le coût d'une trajectoire dans un espace d'états.

Un exemple simple, mais classique, est celui du calcul des plus courts chemins dans un graphe, par l'algorithme de Floyd (1962) ; on doit plutôt parler de chemins à moindre coût, ce problème n'ayant aucune signification « métrique ».

On propose ici de trouver le chemin à moindre coût pour traverser la matrice carrée suivante, composée d'entiers naturels, depuis la case (0,0) jusqu'à la case (6,6).

$$M = \begin{pmatrix} 1 & 5 & 2 & 5 & 7 & 9 \\ 7 & 3 & 4 & 1 & 2 & 4 \\ 1 & 0 & 4 & 7 & 2 & 1 \\ 2 & 6 & 2 & 1 & 0 & 5 \\ 0 & 1 & 3 & 8 & 9 & 3 \\ 5 & 0 & 7 & 1 & 2 & 5 \end{pmatrix}$$

Les seuls déplacements autorisés sont les déplacements vers la droite et vers le bas. On appelle cout d'un chemin, la somme des entiers situés sur ce chemin.

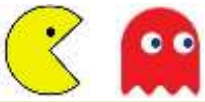
Par exemple : le chemin surligné en jaune à travers la matrice M a un coût de $1+7+1+0+6+2+1+8+1+2+5=34$. Ce chemin n'est pas minimal, il n'est donc pas optimal.

Les deux algorithmes qui suivent mettent en œuvre deux stratégies très différentes :

1. Le premier est un algorithme naïf de type *glouton*, non optimal.
2. Le deuxième propose une démarche de programmation dynamique de type Bottom Up, avec mémoïsation.

PARTIE A : une solution naïve et gloutonne

La première idée consiste à choisir à chaque embranchement d'aller vers la droite si le coût de la case à droite est plus faible (ou égal) au cout de la case en bas et d'aller vers le bas sinon. Cette façon de progresser est caractéristique de l'algorithme glouton.



Qu'est-ce qu'un algorithme glouton ?

Les algorithmes pour problèmes d'optimisation exécutent en général une série d'étapes, chaque étape proposant un ensemble de choix. Pour de nombreux problèmes d'optimisation, la programmation dynamique est une approche bien trop lourde pour déterminer les meilleurs choix ; d'autres algorithmes, plus simples et plus efficaces, peuvent faire l'affaire. **Un algorithme glouton fait toujours le choix qui lui semble le meilleur sur le moment**. Autrement dit, il fait un choix localement optimal dans l'espoir que ce choix mènera à une solution globalement optimale.

Les algorithmes gloutons n'aboutissent pas toujours à des solutions optimales, mais ils y arrivent dans de nombreux cas !

La première idée consiste à choisir à chaque embranchement d'aller vers la droite si le coût de la case à droite est plus faible (ou égal) au cout de la case en bas et d'aller vers le bas sinon. Si on arrive sur la dernière ligne de M on ne pourra se déplacer que vers la droite et si on arrive sur la dernière colonne on ne pourra se déplacer que vers le bas. Par exemple avec M_0 on aura le chemin $[D, D, B, D, D, B, B, D, B, B]$, dont le cout est égal à 30.

$$M = \begin{pmatrix} 1 & 5 & 2 & 5 & 7 & 9 \\ 7 & 3 & 4 & 1 & 2 & 4 \\ 1 & 0 & 4 & 7 & 2 & 1 \\ 2 & 6 & 2 & 1 & 0 & 5 \\ 0 & 1 & 3 & 8 & 9 & 3 \\ 5 & 0 & 7 & 1 & 2 & 5 \end{pmatrix}$$

Écrire une fonction CheminEtCout(M) renvoyant le chemin à suivre ainsi que le coût de ce chemin sur la matrice M.

PARTIE B : une solution dynamique de type Bottom Up avec mémoïsation, ascendante.

Cette solution part donc de la case d'arrivée et remonte jusqu'à la case de départ.

On utilise une matrice $MatriceCout$ et une matrice $MatriceChoix$:

- $MatriceCout[i,j]$ indique le poids minimal d'un chemin qui mène de la case (i,j) à la case d'arrivée
- $MatriceChoix[i,j]$ indique le bon choix à faire si l'on se trouve dans la case (i,j) pour continuer sur un chemin **optimal** ; on adopte la convention que 1 indique d'aller à droite et −1 d'aller vers le bas.

$$M = \begin{pmatrix} 1 & 5 & 2 & 5 & 7 & 9 \\ 7 & 3 & 4 & 1 & 2 & 4 \\ 1 & 0 & 4 & 7 & 2 & 1 \\ 2 & 6 & 2 & 1 & 0 & 5 \\ 0 & 1 & 3 & 8 & 9 & 3 \\ 5 & 0 & 7 & 1 & 2 & 5 \end{pmatrix} \qquad MatriceCout = \begin{pmatrix} 27 & 28 & 24 & 23 & 24 & 27 \\ 26 & 23 & 22 & 18 & 17 & 18 \\ 19 & 20 & 20 & 21 & 15 & 14 \\ 18 & 22 & 16 & 14 & 13 & 13 \\ 16 & 16 & 18 & 16 & 16 & 8 \\ 20 & 15 & 15 & 8 & 7 & 5 \end{pmatrix}$$

Et $$MatriceChoix = \begin{pmatrix} -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 \\ -1 & 1 & -1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 & 1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Le cout optimal est alors égal à 27.

La case en haut à gauche de la matrice $MatriceCout$ contient le poids optimal et, grâce à la matrice $MatriceChoix$, il est possible de reconstituer le chemin optimal (surligné sur l'exemple).

Pour construire ces matrices :

- On commencera par construire la dernière ligne de chaque matrice de droite à gauche (en utilisant une boucle inversée du type **for j in range(n-2,-1,-1))**
- On complétera ensuite les lignes (de la ligne n − 2 à la ligne 0) de droite à gauche en utilisant (sans le justifier) que pour $0 \leq i \leq n-2$ et pour $0 \leq j \leq n-2$:

$$\text{MatriceCout[i, j]} = \text{M[i, j]} + \min(\text{MatriceCout[i, j + 1]}, \text{MatriceCout[i + 1, j]})$$

Écrire une fonction construireMatrice(M) qui crée et remplit les matrices $MatriceCout$ et $MatriceChoix$ à partir de M.

Écrire une fonction chemin(M) qui renvoie le chemin (sous la forme d'une liste de 'D' et de 'B') ainsi que le coût du chemin optimal.