Programmation dynamique

Méta-Fibonacci

On veut calculer un terme de rang donné de la suite Q de Hofstadter définie (par récurrence) sur $\mathbb{N}$ par :

$$\begin{cases} Q_0 = 1 \\ Q_1 = 1 \\ Q_n = Q_{n-Q_{n-1}} + Q_{n-Q_{n-2}} \end{cases}$$

Pour la culture générale, cette suite est le premier exemple de suite méta-Fibonacci. C'est une suite qui présente à la fois des régularités, et du chaos. Elle apparaît dans le livre Gödel-Escher-Bach (https://fr.wikipedia.org/wiki/G%C3%B6del,_Escher,_Bach_:_Les_Brins_d%27une_Guirlande_%C3%89ternelle), de Douglas Hofstadter. Ce livre, à partir d'une exploration des relations entre la logique, la musique et l'art, montre comment la réflexion et le sens peuvent émerger dans nos processus cognitifs. Ce résumé peut montrer le livre comme ardu, il ne l'est pas (il n'est pas facile non plus), et il est plutôt ludique, avec des énigmes et un côté rigolo. Il est téléchargeable sur le web... en anglais et peut-être pas très légalement. Si vous voulez devenir célèbre, trouvez des résultats sur une des suites de Hofstadter (https://en.wikipedia.org/wiki/Hofstadter_sequence). En effet on ne connaît quasiment rien sur ces suites.

Questions :

1. Sur le papier, comprendre comment sont calculés les termes de la suite. Pour cela, calculer les dix premiers à la main permet de clarifier la formule.
2. Ecrire un programme récursif pour calculer un terme donné de la suite $(Q_n)_{n\in\mathbb{N}}$
3. Représenter à l'aide d'un arbre les appels récursifs nécessaire pour calculer Q_4 (voire Q_5)

In [ ]:
```
def hofVersionRecursive(rang):
    return

hofVersionRecursive(4)
```

Un petit tour sur Python tutor (http://pythontutor.com/visualize.html#mode=edit) permet de visualiser la complexité et l'évolution de la pile d'appels.
Comme on peut le constater, un programme purement récursif n'est pas efficace pour ce type de situation. Par ailleurs écrire un programme simple en itératif, comme ce que l'on peut faire pour une suite "normale" paraît très complexe. Il faut donc inventer autre chose. On va écrire un deuxième programme récursif, qui stockera les valeurs déjà calculées dans un dictionnaire pour éviter de les calculer à nouveau.

In [ ]:
```
def hofMemo(rang,memoire ={0:1 , 1:1}):

        return


# 2ème version
def hofVersionMemo(n):
    tableau = [None] * (n+1)
    return hofVersionRecursive2(n,tableau)


def hofVersionRecursive2 (n,tableau):

    return

assert(hof(10) == 6)
assert(hof(13) == 8)
assert(hof(25) == 14)
```

Mémoïsation

C'est l'affreux nom jargonnesque (inventé juste pour se la péter, puisque cela signifie simplement "mémorisation des résultats pour réutilisation") de la méthode que vous avez utilisé pour répondre à la question précédente.

Un nouveau paradigme algorithmique

On peut encore simplifier le calcul d'un terme de la suite de Hofstader, du moins du point de vue alogrithmique. En effet calculer toutes les valeurs successives, en les stockant dans une structure de données adaptée, jusqu'au résultat cherché est plus facile à programmer. Faites-le.

In [ ]:
```
def hofDynamique(n):

  return
```

Calculer toutes les valeurs successives jusqu'au résultat ressemble beaucoup à la mémoïsation. Le calcul se fait *a priori* plutôt qu'*a posteriori*. le code est plus simple, et en mémoire machine on gagne un peu (sur le stockage des appels récursifs).

C'est la deuxième étape de la **programmation dynamique**. La première étape est l'obtention d'une formule de récurrence, et la troisième étape consiste à trouver une solution au problème posé ; dans le cas de la suite Q la formule de récurrence est donnée et l'obtention de cette solution est immédiate. Nous allons voir avec l'exemple suivant que ce n'est pas toujours le cas.

Retour sur le sac à dos

La programmation dynamique est souvent utilisée pour résoudre des problèmes complexes d'optimisation, comme ceux vu en première avec les algorithmes gloutons.

Dans le problème du sac à dos, on a des objets d'un certain poids et d'une certaine valeur, à ranger dans un sac à dos de contenance limitée. On cherche à optimiser la valeur totale que l'on peut transporter, sachant que l'on ne peut prendre qu'une seule fois un objet.

Exemple :

Numéro objet	**0**	**1**	**2**	**3**	**4**	**5**
Masse en Kg	1	2	3	4	5	7
Valeur en euros	10	32	40	18	81	112

Les méthodes vues en première

1. On adopte une méthode brute : on évalue le poids et la valeur correspondante de toutes les solutions possibles. Quelle est alors la complexité de l'algorithme ?
   Réponse :

1. On utilise un algorithme glouton. Rappelez le principe, puis complétez le tableau en faisant apparaître le rapport valeur/poids, et déterminer avec cet algorithme le choix obtenu (on fera tourner l'algorithme "à la main")
   Réponses :

Numéro objet	**0**	**1**	**2**	**3**	**4**	**5**
Masse en Kg	1	2	3	4	5	7
Valeur en euros	10	32	40	18	81	112
Valeur/poids						

   Le choix de l'algorithme glouton est :

1. Quelle est la complexité de l'algortihme glouton ?

1. Le résultat est-il optimal ?

Un algorithme de programmation dynamique

La difficulté à laquelle on est confronté est de trouver une formule de récurrence pour résoudre le problème. On va procéder par étapes.
Notons $Val(i, m)$, pour i entier allant de 1 à 5 , et m entier allant de 0 à 10, la valeur maximale que l'on peut atteindre en prenant des objets parmi ceux numérotés de 0 à i et pour une masse totale ne dépassant pas m, c'est-à-dire la contenance maximale du sac à dos. On note v_i la valeur de l'objet i, et m_i sa masse.

1. Que représente $Val(0, m)$?

1. Donner sa valeur.

1. Que représente $Val(i, 0)$? Donner sa valeur.

1. Que représente $Val(5, 10)$?

1. Comment calculer $Val(i, m)$ en fonction des valeurs précédentes de $Val(truc, zaffair)$?
   Deux possibilités

   - Soit l'objet numéroté i n'a pas pu être choisi (sa masse étant supérieure à la masse disponible) : dans ce cas $Val(i, m) =...$
   - Soit l'objet numéroté i a pu être choisi (mais il n'est pas certain que cela soit intéressant) : Dans ce cas $Val(i, m) =...$

1. Cette relation de récurrence va nous permettre de remplir le tableau Val suivant :

Numéro objet	**0**	**1**	**2**	**3**	**4**	**5**
Masse en Kg	1	2	3	4	5	7
Valeur en euros	10	32	40	18	81	112

$i \setminus m$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
objet 0	0	10	10	10	10	10	10	10	10	10	10
objet 1	0	10	(?)	(??)							
objet 2	0										
objet 3	0										
objet 4	0										
objet 5	0										

On rappelle que dans ce tableau, la ligne objet i indique que l'on peut prendre tous les objets de 0 à i

Expliciter la valeur de $Val(1, 2)$ (dans la cellule notée (?)), et celle de $Val(1, 3)$ (dans la cellule notée (??)), puis compléter le tableau et conclure sur la valeur maximale que l'on peut porter dans le sac. *Il est conseillé de recopier ce tableau sur papier, la partie reconstruction sera plus facile à comprendre si on peut gribouiller)*.

7 . Ecrire sur papier l'algorithme correspondant.

8 . Puis le programmer en Python.

9 . Reconstruction : *La reconstruction se fait en partant du résultat final*

Puisque 154 est différent du nombre immédiatement au dessus (153), c'est que le dernier objet a été choisi. On a donc pris l'objet 5 qui pèse 7 kg et vaut 112 brézoufs. 154 - 112 = 42 et 10 - 7 = 3 : on est ramenés à la colonne d'indice [3], il reste à trouver les 42 €.
Dans cette colonne, on remonte ligne à ligne : la valeur change entre la ligne d'indice [1] et la ligne d'indice [0]. Cela signifie que l'on a sélectionné l'objet 1 (indice de ligne [1]), de valeur 32 brézoufs et de poids 2 kg.
On répète : 3 kg - 2kg = 1 kg et 42 - 32 = 10. On est en colonne d'indice [1]. La valeur 10 est présente dès la première ligne, d'indice [0]. On a donc pris également l'objet 0 (indice de ligne [0]).
La valeur restante est 0 : on a fini la reconstruction.
Les objets choisis sont 0, 1 et 5 en inversant l'ordre de récupération des objets (ceci dit on se fiche de l'ordre dans lequel les objets sont pris !).
Remarquez la similitude avec la reconstruction du chemin optimal dans Dijkstra.

Algorithme de reconstruction de la solution optimale du sac à dos, obtenue par programmation dynamique
Données :

- le tableau val construit lors de la recherche de la solution
- le tableau $tableau_objet$ initial, donnant la masse et la valeur des objets
  Renvoie :
- le tableau $objets_pris$ des objets retenus pour la solution optimale

$objets_pris \leftarrow$ liste vide
Indice permettant de travailler sur les numéros d'objets :
$indice_objet \leftarrow$ indice de la dernière ligne de $objets_pris$
Indice permettant de travailler sur la contenance restante :
$indice_contenance \leftarrow$ indice de la dernière colonne de $objets_pris$
Valeur dans le sac à dos :
$valeur \leftarrow$ dernière cas en bas à droite du tableau val

Tant que $valeur \neq 0$:

> # L'objet est pris si l'indice de ligne vaut 0 (vu qu'on est dans la boucle, on est sûr que $valeur \neq 0$)
> # ou si dans le tableau val, la cellule juste au dessus de la cellule courante est différente
> **si** $indice_ligne \neq 0$ **ou** $val(cellule_courante) \neq val(cellule_au_dessus)$:
>
> > ajouter $indice_objet$ à $objets_pris$
> > décrémenter $indice_contenance$ du poids de l'objet
> > décrémenter $valeur$ de la valeur de l'objet
>
> # Dans tous les cas on diminue l'indice de ligne/d'objet en passant à la ligne au dessus
> décrémenter $indice_objet$

Renvoyer $objets_pris$

Implémenter cet algorithme en Python

In [ ]:
```
########### Importation des modules ou fonctions externes #########
#####

################## Définition des fonctions locales ###############
#####

def affichage_donnees_initiales(tab):
    # Utilitaire utilisé pour afficher le tableau des poinds et des
valeurs
    print("\tnuméro \t masse \t valeur")
    for i in range(0,len(tab)):
        numero=i
        masse=tab[i][0]
        valeur=tab[i][1]

        print("\t %s \t %s  \t %s"%(numero,masse,valeur))

def affichage_tableau(tab):
    for i in _tange(0,len(tab)):
        print(tab[i])

def construction_tab_val(objets, M):
    """
    @param objets : tableau des couples (masse, valeur) des objets
que l'on peut mettre dans le sac à dos
    @param M : entier positif, la contenance du sac à dos
    @return val : tableau construit en programmation dynamique, rés
olvant le problème du sac à dos
                l'entier à l'intersection de la dernière colonne et
ligne étant la valeur optimale
    """
    # Construction du tableau val

    # On complète la première ligne

    # On boucle pour compléter ligne à ligne

    return

def reconstruction(tableau_objets, val):
    return

################### corps principal du programme ##################
#####

tableau_objets = [(1, 10), (2, 32), (3, 40), (4, 18), (5, 81), (7,
112)]

affichage_donnees_initiales(tableau_objets)

val = construction_tab_val(tableau_objets, 10)
print("Valeur maximale : ", val[-1][-1])
print("\nle tableau rempli dynamiquement :")
affichage_tableau(val)
retenus = reconstruction(tableau_objets, val)
```

```
print("\nObjets retenus :", retenus)
print("soit les valeurs : ", end = "")
for i in range(len(retenus)) :
    print(tableau_objets[retenus[i]][1], end = " ")
print()

print("\n\nDans un ordre différent :")
shuffle(tableau_objets)
affichage_donnees_initiales(tableau_objets)
val = construction_tab_val(tableau_objets, 10)
print("Valeur maximale : ", val[-1][-1])
print("\nle tableau rempli dynamiquement :")
affichage_tableau(val)
retenus = reconstruction(tableau_objets, val)
print("\nObjets retenus :", retenus)
print("soit les valeurs : ", end = "")
for i in range(len(retenus)) :
    print(tableau_objets[retenus[i]][1], end = " ")
print()
```

L'algorithme précédent renvoie la valeur maximale, mais pas la composition du sac. Le modifier et compléter de manière à :

- sauver le tableau en entier ;
- reconstruire en partant de la solution la composition du sac (on "remonte" objet par objet).
  Puis programmer la fonction "reconstruction" afin de trouver la composition du sac.

A retenir

La programmation dynamique est surtout utilisée pour résoudre des problèmes d'optimisation. Le problème doit pouvoir se résoudre à partir de sous-problèmes du même type, la solution dépendant du résulat de ces sous-problèmes. Résoudre un problème en programmation dynamique se fait en trois étapes :

1. Trouver une formule de récurrence pour obtenir la valeur optimale (c'est la partie la plus difficile)
2. Ecrire un algorithme itératif (et surtout pas récursif) pour obtenir l'optimum. On part des plus petits problèmes et on va vers les plus grands
3. Reconstruire la solution optimale *a posteriori*. Cette dernière étape est parfois ignorée dans le cas de problèmes avec des données très lourdes. On part du plus grand problème et on redescend vers les petits.

<hr style="color:black; height:1px />

<div style="float:left;margin:0 10px 10px 0" markdown="1" style = "font-size = "x-small">

(http://creativecommons.org/licenses/by-nc-sa/4.0/)
Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International (http://creativecommons.org/licenses/by-nc-sa/4.0/).

frederic.mandon @ ac-montpellier.fr, Lycée Jean Jaurès - Saint Clément de Rivière - France
Hélène Carles, Lycée Frédéric Bazille, Montpellier, France