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

print(hofVersionRecursive(5))
```

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.

Exemple : le sac à dos peut contenir 10 kg

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 $V(i, m)$, 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. On note v_i la valeur de l'objet i.

1. Que représente $V(0, m)$?

1. Donner sa valeur.

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

1. Que représente $V(5, 10)$?

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

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

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

Expliciter la valeur de $V(1, 2)$ (dans la cellule notée (?)), et celle de $V(1, 3)$ (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

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

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

def affichageDoneesInitiale(tab):
    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 affichageTableau(tab):
    for i in range(0,len(tab)):
        print(tab[i])

def ElaborationTableau(n,M):
    """
    Données : deux entiers n nb total d'objets et M al masse maxima
le autorisée
    Résultat : le tableau généré par la programmation dynamiqe,
    l'entier à l'intersection de la dernière colonne et ligne étant
la valeur optimale
    """

    return

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

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

affichageDoneesInitiale(tableauValeurs)
print("le tableau rempli dynamiquement !")
affichageTableau(ElaborationTableau(6,10))
```

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 />

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