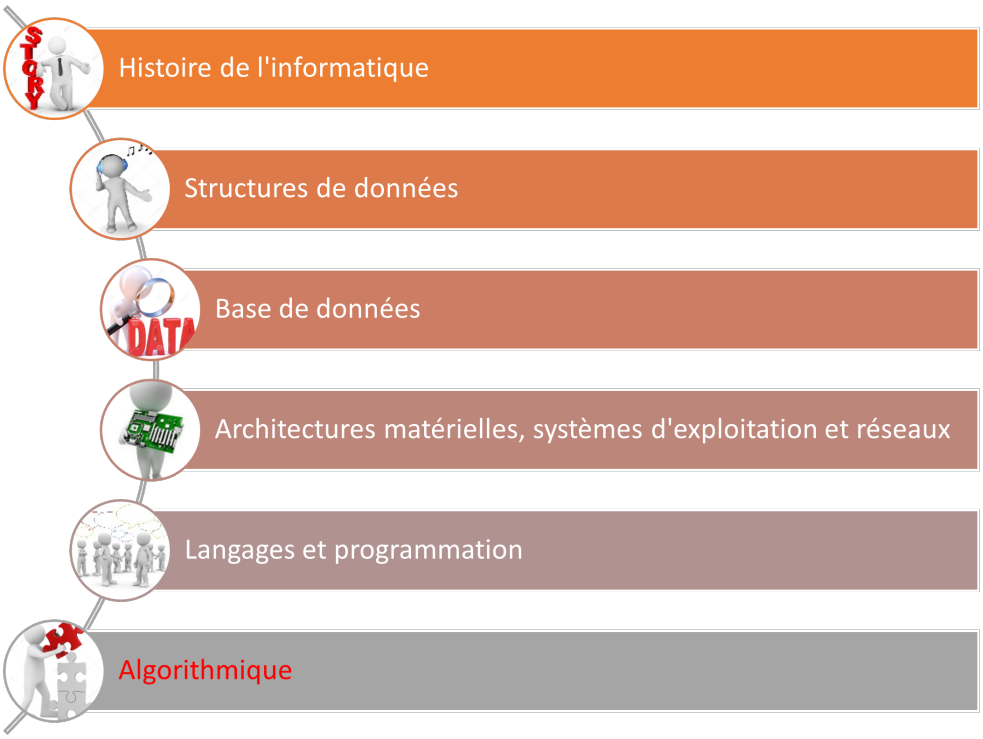

Table des matières

1 Un premier exemple : la suite de Hofstadter

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

$$\begin{cases} Q_0 = 1 \\ Q_1 = 1 \\ Q_n = Q_{n-Q_{n-1}} + Q_{n-Q_{n-2}} \text{ pour tout entier naturel n supérieur ou égal à 2} \end{cases}$$

Cette suite est le premier exemple de suite méta-Fibonacci, on admettra que tous ses termes sont entiers. C'est une suite qui présente à la fois des régularités, et du chaos. 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.

Partie A : Un premier algorithme

1. Sur le papier, comprendre comment sont calculés les termes de la suite. Pour cela, calculer à la main les dix premiers.
2. Écrire un programme récursif pour calculer un terme donné de la suite :

```
def hofVersionRecursive(rang):
  return
assert(hofVersionRecursive(10) == 6)
assert(hofVersionRecursive(13) == 8)
assert(hofVersionRecursive(25) == 14)

```

3. Représenter à l'aide d'un arbre les appels récursifs nécessaire pour calculer Q_4 (voire Q_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.

Partie B : Mémoïsation

On va écrire un deuxième programme récursif, qui stockera les valeurs déjà calculées dans un dictionnaire pour éviter de les calculer à nouveau :

```
def hofMemo(rang, memoire = {0 : 1, 1 : 1}):
  return
assert(hofMemo(10) == 6)
assert(hofMemo(13) == 8)
assert(hofMemo(25) == 14)
```

Définition :
La mémoïsation désigne le procédé algorithmique dans lequel on a mémorisation des résultats intermédiaires pour une réutilisation future.

Partie C : programmation dynamique

On peut encore simplifier le calcul d'un terme de la suite de Hofstader, du moins du point de vue algorithmique. 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.

```
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 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.

2 Problème du sac à dos

La programmation dynamique est souvent utilisée pour résoudre des problèmes complexes d'optimisation, comme ceux vus 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	6
Valeur en euros	10	32	42	18	85	114

2.1 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 ?
2. On utilise un algorithme glouton.
   (a) Rappelez le principe, puis complétez le tableau en faisant apparaître le rapport valeur/masse, et déterminer avec cet algorithme le choix obtenu (on fera tourner l'algorithme "à la main").

Numéro objet	0	1	2	3	4	5
Masse en Kg	1	2	3	4	5	6
Valeur en euros	10	32	42	18	85	114
Valeur/masse						

   (b) Quelle est la complexité de l'algorithme glouton ?
   (c) Le résultat est-il optimal ?

2.2 Un algorithme de programmation dynamique

La difficulté à laquelle on est confrontée 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 0 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 respectivement v_i et m_i la valeur et la masse de l'objet i.

1. Que représente $V(0,\ m)$?
2. Donner sa valeur.
3. Que représente $V(i,\ 0)$? Donner sa valeur.
4. Que représente $V(5,\ 10)$?
5. Comment calculer $V(i, m)$ en fonction des valeurs précédentes de $V(i',\ m')$?
   Deux possibilités
   (a) Soit l'objet numéroté i n'a pas pu être choisi (sa masse étant supérieure à la masse disponible) :
   Dans ce cas $V(i,\ m) = \ldots$
   (b) Soit l'objet numéroté i a pu être choisi (mais il n'est pas certain que cela soit intéressant) :
   Dans ce cas $V(i,\ m) = \ldots$
   Cette relation de récurrence va nous permettre de remplir le tableau suivant :

Numéro objet (i)	0	1	2	3	4	5
Masse en Kg (m_i)	1	2	3	4	5	6
Valeur en euros	10	32	42	18	85	114

i \ 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 les deux lignes suivantes du tableau. Il est conseillé de recopier ce tableau sur papier, la partie reconstruction sera plus facile à comprendre.
6. Écrire sur papier l'algorithme correspondant.
7. Puis le programmer en Python.

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

1. sauver le tableau en entier ;
2. 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.

3 Synthèse

Définition :
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 résultat 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. Écrire un algorithme itératif (et surtout pas récursif) pour obtenir l'optimum. On part des plus petits problèmes et on va vers les plus grands, en stockant dans un tableau les résultats intermédiaires du calcul afin de les réutiliser.
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.

4 Exercices

Exercice n° 1.

La suite de Fibonacci est la suite de sentiers naturels (F_n) définie par :

$$\begin{cases} F_0 = 1 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ pour tout enbtier naturel n supérieur ou égal à 2} \end{cases}$$

Écrire une fonction `fibonacci(n)` qui calcule et et renvoie la valeur de F_n en utilisant la programmation dynamique.

Exercice n° 2. Le problème du rendu de monnaie

Il s'agit de rendre une somme entière `valeur` avec un système de pièces : `s = [`p_1, p_2, ..., p_n`]` où p_1, p_2, ..., p_n représentent les valeurs des différentes pièces. Les pièces sont en quantité suffisante et la question est d'utiliser le moins de pièces possible.

On suppose que $p_1 = 1$, donc le problème a toujours au moins une solution et $p_1 < p_2 < \ldots < p_n$.
Nous avons vu en première comment résoudre ce problème avec des algorithmes gloutons, qui donnent la solution optimale dans la plupart des systèmes de pièces en vigueur.
Dans cette exercice, nous vous proposons de réaliser une solution à l'aide de la programmation dynamique.

On commencera par définir la liste `Nb_min` telle que `Nb_min[i]` est le nombre minimal de pièces à donner pour rendre la somme `i`.
Le système de pièces contient la pièce $p_1 = 1$, ainsi on sait que l'on peut rendre la somme `i` avec `i` pièces de 1. On initialise donc la liste ainsi :

```
Nb_min = [i for i in range(valeur + 1)]
```

1. Pour rendre une somme quelconque `v` (avec `v` $\leqslant$ `valeur`), s'il est possible de rendre la pièce p_i, il reste alors à rendre la somme `v -` p_i. Donc si on sait rendre toute somme inférieure à `v` de manière optimale, il suffit de tester les différentes possibilités de rendre `v -` p_i pour toute pièce p_i (si l'on peut la rendre !) et de choisir la meilleure. (Cela revient à considérer successivement les pièces de la liste `s = [`p_1, p_2, ..., p_n`]` si elles sont inférieures à `v`.)
   Exprimer alors `Nb_min[v]` en fonction de `Nb_min[v -` p_i`]`.
2. Proposez alors une fonction `monnaie_dyn(valeur, systeme)` (codée en programmation dynamique !) qui renvoie le nombre minimal de pièces de monnaies pour rendre une somme : `valeur`, avec un système de pièces qui est : `systeme`.
   On pourra tester :

```
>>> systeme = [1, 2, 5, 10]
>>> monnaie_dyn(14, systeme)
3

```

3. Votre programme renvoie le nombre de pièces minimal pour rendre la somme `valeur` mais il ne donne pas la liste des pièces à rendre. On se propose dans cette question de construire cette solution. Il suffit d'ajouter un second tableau `sol` qui contient pour chaque somme de `0` à `valeur` une solution minimale pour cette somme. On remplit ce tableau exactement au même moment où on remplissait le tableau précédent dans le code de la fonction `monnaie_dyn(valeur, systeme)`.
   Coder cette fonction :

```
def monnaie_dyn_solution(valeur, systeme):
    """renvoie une liste minimale de pièces pour faire
    la somme valeur avec le système de pièces"""
    Nb_min = [i for i in range(valeur + 1)]
    sol = [[]] * (valeur + 1)
    ...
    return sol[valeur]

```

Exercice n° 3.

On considère une grille $n \times m$ (n lignes et m colonnes), on cherche à coder `chemin(n, m)` une fonction qui calcule le nombre de chemins qui mènent du coin supérieur gauche au coin inférieur droit, en se déplaçant uniquement le long des traits horizontaux vers la droite et le long des traits verticaux vers le bas.
On commencera par bien réfléchir à une relation de récurrence.
On vérifiera que `chemin(10, 10)` renvoie 184 756