TP : Programmation dynamique

I) Dictionnaires (TD)

Rappel de la syntaxe des *dictionnaire*.
Une liste `L` est un ensemble ordonné d'éléments auxquels on accède via leur indice : `L[i]`.
Un dictionnaire suit le même principe, mais au lieu d'un indice, on utilise une clef que l'on définit soi-même : c'est un semble de paires *clef : valeur*. Un exemple en forme d'annuaire :

`mesNum = {'Ariane': 0620, 'Pierre': 0720}`	Analogue de `L = [2, 4, 1]`.
`mesNum['Ariane']` `mesNum['Thesee'] = 0454`	Analogue de `L[i]` pour une liste. Rentre un nouveau couple clef : valeur.
`'Egee' in mesNum`	Teste la présence d'une clef.

Exercice 1
Le magasin *Geek and sons* gère la « liste » des articles disponibles, ainsi que leurs prix, à l'aide d'un dictionnaire nommé `Produits`, dont les clés sont le nom des articles (type chaîne de caractère), et les valeurs associées sont les prix correspondants (type réel).
Par exemple :
`Produits ={'Sabre laser':229.0, 'Mitendo DS':127.30, 'Coussin Linux':74.50,...}`

1) Ecrire une fonction qui prend en entrée le nom d'un produit (sous forme d'une chaîne de caractère) et qui renvoie `True` si le produit est disponible, `False` sinon.

2) Trouver le produit dont le prix est le plus cher.

3) Ecrire une fonction `fourchette(mini, maxi)`, qui renvoie la liste des noms des objets dont le prix est entre `mini` et `maxi`.

4) Le *panier* est un concept omniprésent dans les sites marchands, *Geeks and sons* n'échappe pas à la règle. En Python, le panier du client sera représenté par un dictionnaire dont les clés sont les noms des produits, et les valeurs la quantité que le consommateur veut acheter.

a) Donner une expression Python correspondant à l'achat de 3 sabres lasers, de 2 coussins Linux et de 1 Mitendo DS.

b) Ecrire une fonction `prix` qui prend en entrée un panier, et qui retourne le prix total correspondant.
Remarque : on supposera que tous les articles du panier sont disponibles.

II) La suite de Fibonacci

On définit la suite (u_n) par

$$u_0 = 0 \qquad u_1 = 1 \qquad \forall n \in \mathbb{N}\backslash\{0,1\} \quad u_n = u_{n-1} + u_{n-2}$$

C'est la « suite de Fibonacci », une suite d'entiers qui a été utilisée au XIIIe siècle pour décrire la croissance d'une population de lapins (on compte les couples à chaque génération, sans décès).
Vous remarquerez que c'est une suite récurrente linéaire : vous connaissez des techniques pour calculer directement le n-ième terme. Mais ces techniques, appliquées à l'informatique, font utiliser des **float** et donc potentiellement perdre en précision (en fait, avec le théorème des accroissements finis on réussit à maîtriser l'erreur... mais ce n'est pas le sujet ici [1]).
Vous aurez éventuellement besoin d'augmenter le niveau de récursion autorisé par Python :

```
1 import sys
2 sys.setrecursionlimit(10000)
```

1. Un test en Python sur une machine 64 bits donne un résultat qui n'est plus entier dès $n = 4$, et dont la valeur entière la plus proche n'est pas la valeur correcte pour $n = 71$.

A) Version récursive naïve

Exercice 2

1) Écrire une fonction récursive qui calcule le n-ième terme de la suite de Fibonacci.

2) Tester la rapidité (un n raisonnable pourra être obtenu par dichotomie – à la main). Que constate-t-on ?

3) Tracer le nombre d'appels récursifs en fonction de n, à l'aide d'un compteur. Conjecturer la complexité en fonction de n.

4) Déterminer, par une preuve rigoureuse, la complexité de l'algorithme. On comptera le nombre d'appel récursif.

B) De bas en haut : méthode ascendante

Exercice 3

Écrire un algorithme itératif (c'est-à-dire une boucle) qui calcule u_n « du bas vers le haut », donc en partant de u_0 et u_1. Donner la complexité de cet algorithme. On comptera le nombre d'additions.

C) Méthode descendante : versions avec mémoïsation

On revient à des algorithmes récursifs, en essayant d'améliorer la complexité.

Exercice 4 (Formule de récurrence usuelle)

Améliorer l'algorithme récursif naïf en stockant les u_k déjà calculés (et donc en vérifiant, à chaque étape, si on a ou non déjà calculé le u_n que l'on cherche). Donner la complexité du nouvel algorithme (on pourra commencer par tracer la courbe du compteur en fonction de n) en comptant le nombre d'additions.

Exercice 5 (Autre formule de récurrence)

On admet les formules : $\forall n \in \mathbb{N}^*$ $\begin{cases} u_{2n+1} &= u_{n+1}^2 + u_n^2 \\ u_{2n} &= 2u_n u_{n-1} + u_n^2 \end{cases}$

Écrire un algorithme récursif avec mémoïsation qui utilise ces formules. Quelle est sa complexité ? (sans preuve : on pourra se contenter de faire un calcul dans le cas où n est d'une forme sympathique).

III) Rendu de monnaie

On cherche à rendre une somme v de monnaie à l'aide de différents types de pièces et de billets.
Les types de pièces et billets sont appelés « système de monnaie », représenté par un n-uplet $S = (s_1, \dots, s_n)$ rangé par ordre croissant ($s_1 < \cdots < s_n$).
Une infinité de pièces de chaque type est disponible : elles ne s'épuisent jamais.
Le résultat sera, dans un premier temps, la liste décroissante des pièces rendues.

Exemple 1
Le système de monnaie de l'euro (tronqué des billets les plus élevés) est $S = \{1, 2, 5, 10, 20, 50\}$. Quelques exemples de rendus :

1) Pour atteindre 63 avec S, on rendra par exemple 1 billet de 50, 1 billet de 10, 1 pièce de 2 et 1 pièce de 1, ce qui peut se noter

$$(50, 10, 2, 1) \qquad 63 = 1 + 2 + 10 + 50$$

2) Pour rendre 7, il y 6 possibilités :

$$(1,1,1,1,1,1,1),\ (2,1,1,1,1,1),\ (2,2,1,1,1),\ (2,2,2,1),\ (5,1,1),\ (5,2)$$

Définition 1 (solution)
Une solution pour rendre la somme v avec le système $S = (s_1, \dots, s_n)$ est une liste ℓ d'éléments de S tels que

$$\sum_{i=1}^{k} \ell_i = v$$

Définition 2 (solution optimale)
Une solution est appelée « optimale » si elle rend le minimum de pièces : la longueur k de la liste ℓ doit être minimale parmi toutes les solutions possibles.

Exemple 2 **1)** Avec S comme ci-dessus, $\ell = (5, 2)$ est optimale pour rendre $v = 7$.

2) Avec $S = \{1, 3, 4\}$ et $v = 6$, la solution optimale sera $\ell = (3, 3)$.

A) Rappels : l'algorithme glouton

La méthode « usuelle » pour rendre la monnaie est celle de l'algorithme glouton : tant qu'il reste quelque chose à rendre, choisir la plus grosse pièce disponible (sans rendre trop). C'est un algorithme très simple et rapide.
On appelle canonique un système S de pièces pour lequel cet algorithme donne une solution optimale quelle que soit la valeur à rendre : c'est le cas de la plupart des système de monnaie en circulation.

Plus généralement, qu'est-ce qu'un algorithme glouton ? Lorsqu'il y a un choix à faire lors d'une étape d'un algorithme (ici, choisir quelle pièce rendre, mais on peut imaginer choisir une conférence à placer dans une salle (Activité 8 an dernier), ou choisir une arête à un embranchement lorsqu'on cherche un plus court chemin dans un graphe), on utilise un critère simple à calculer (ici, la plus grande pièce inférieure à la valeur restant à rendre). Pour le problème du choix de l'activité, on choisissait la conférence qui se terminait le plus tôt.

Ces algorithmes ne donnent pas forcément la solution optimale, ni même une solution, selon le problème traité. Mais ils ont souvent une bonne complexité.

Exemple 3

1) Les solutions données dans les exemples 1.1 ($63 = 1+2+10+50$) et 2.1 ($7 = 5+2$), sont des solutions gloutonnes.

2) Dans l'exemple 2.2, avec $S = \{1, 3, 4\}$ et $v = 6$, la solution gloutonne sera $(4, 1, 1)$, donc ne sera pas une solution optimale : $(3, 3)$ est optimale.

Exercice 6 (algorithme glouton)
Dans cet exercice, on se donne un ensemble de pièces `S` avec lequel on cherche à rendre une somme v.

1) Écrire une fonction `maximum(L,borne)` qui retourne le maximum des éléments de la liste `L` inférieurs ou égaux à `borne`.

2) Écrire en python un algorithme `renduGlouton(v, S)` qui prend en entrée une valeur entière `v` et un tuple `S`, et retourne une liste `L` d'éléments de S de longueur k telle que

$$v = \sum_{i=0}^{k-1} L[i]$$

On utilisera un algorithme glouton, de préférence récursif. Testez sur les exemples donnés ci-dessus.

3) Êtes-vous certain que votre programme se termine ? Trouver un exemple de S et v tel que l'algorithme glouton n'arrive pas à rendre la monnaie, alors que le problème de rendu de monnaie a une solution. Corriger votre algorithme au besoin.

4) (facultatif) Proposer une version itérative de l'algorithme ci-dessus.

5) (facultatif) Proposer une version de l'algorithme qui retourne, au lieu de la liste `L`, un dictionnaire `d` dont les clefs sont des pièces de `S`, et les valeurs `d[s]` le nombre de pièce s rendues, c'est-à-dire tel que

$$v = \sum_{s \in d.keys()} d[s] \times s$$

B) Version dynamique

Cherchons désormais une solution optimale au problème de rendu de monnaie.
Il s'agit dans le cas général d'un problème NP-complet (en particulier, on pense qu'il n'existe pas d'algorithme de complexité polynomiale en n permettant de le résoudre).

L'algorithme proposé ici pour chercher la solution optimale teste toutes les décompositions possibles de v.

Le principe consiste à parcourir les valeurs possibles s des pièces utilisables – donc inférieures à v – et à rappeler la fonction pour rendre la valeur $v - s$.
Pour ne pas tester deux fois le même rendu (pour rendre 7, 5 puis 2 ou 2 puis 5), s devra aussi être inférieur ou égal à la plus petite pièce déjà utilisée.
Pour une version animé et interactive du parcours de l'arbre des appels récursifs par l'algorithme, on peut consulter la page suivante :
`http://mpechaud.fr/scripts/recursivite/recursivite.html#rendu`

Exercice 7 (Version dynamique)

1) Écrire en python un algorithme `rendusAf(v, S, L=[])` qui prend en entrée une valeur entière `v`, un tuple `S`, et la liste `L` du rendu, et affiche tous les rendus possibles pour la somme v sous forme de listes `L` comme à l'exercice 6.

2) Écrire en python un algorithme `rendus(v, S, L=[], resultat=[])` qui prend en entrée une valeur entière `v`, le tuple `S`, la liste `L` comme ci-dessus, et une liste `resultat` qui contiendra le resultat, et retourne la liste de tous les rendus possibles pour la somme v comme ci-dessus. On veillera à éviter de calculer plusieurs fois la décomposition d'une même somme w.