Algorithmes gloutons

1 - Rendu de monnaie

On suppose donné un système monétaire constitué de pièces d'une certaine valeur. Ainsi en euros nous avons (en centimes) les pièces de 1, 2, 5, 10, 20, 50, 100 et 200.
Comment rendre de la monnaie de manière optimale ? En d'autres termes, comment rendre une somme r de monnaie en minimisant le nombre de pièces utilisées ?
Soit $S = (p_0,p_1,\cdots,p_{n-1})$ le système de pièces rangées dans l'ordre croissant. Il s'agit de trouver $(x_0,\cdots,x_{n-1})$ tels que :

$$\sum_{i=0}^{n-1} x_i p_i = r \text{ et } \sum_{i=0}^{n-1} x_i \text{ soit miminal}$$

L'algorithme glouton se déroule ainsi :

- on cherche dans l'ordre décroissant des valeurs la première pièce qui est inférieure à r ;
- on retranche à r la valeur de cette pièce ;
- on recommence en partant de la pièce déjà utilisée et on réitère le procédé jusqu'à ce que la somme à rendre soit égale à 0.

Ainsi, dans le système euro, pour rendre 8 centimes, le rendu de monnaie sera : (5,2,1).

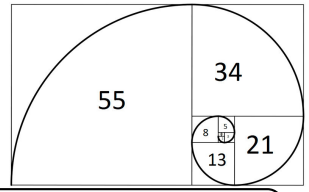
Ex. 1 :

1. Calculer à la main le rendu de monnaie proposer par l'algorithme glouton pour une somme $r = 4{,}48$ euros= 448c.
2. Ecrire une fonction `rendu_monnaie(pieces,r)` qui prend comme paramètres une liste d'entiers `pieces` correspondant au système de pièces rangées dans l'ordre croissant et à un entier `r` correspondant à la somme à rendre. Elle devra renvoyer une liste `rendu` de même longueur que `pièces` correspondant aux entiers $(x_0,\cdots,x_{n-1})$.
3. On peut démontrer que pour le système euro comme dans la plupart des systèmes monétaires mondiaux le rendu obtenu par la méthode gloutone est optimal. Mais ce n'est pas toujours le cas.
Ainsi, si le système de pièces est $(1,4,5,10,\cdots)$, quel est le rendu glouton pour 8c ? Est-ce la réponse optimale ?

2 - Décomposition de Zeckendorf

On considère la suite de Fibonacci définie par : $F_0 = 0$, $F_1 = 1$ et :

$$\forall n \in \mathbb{N},\ F_{n+2} = F_{n+1} + F_n$$



Ex. 2 :

1. Écrire une fonction `fibo(n)` prenant en argument un entier naturel n et renvoyant la valeur de F_n.
2. Vérifier que l'algorithme suivant renvoie la liste `[0,1,1,2,3,5,8,13,21,34,55]`.

```
L = []
for k in range(11):
    L.append(fibo(k))
print(L)
```

AM, JB, NG, SR

On admet le **théorème de Zeckendorf** qui énonce que tout entier naturel n peut se décomposer comme somme d'éléments de la suite $(F_n)_{n\in\mathbb{N}}$. Plus précisément, pour tout entier $n \geq 1$, il existe un entier k et un k-uplet d'entier $(c_1, \cdots, c_k)$ vérifiant :

- pour tout $i \in \{1, \cdots, k-1\}$, $c_i < c_{i+1}$;
- $n = \sum_{i=1}^{k} F_{c_i}$.

Figure 1: Édouard Zeckendorf [1901 – 1983]

Ce théorème fut démontré en 1952 par Édouard Zeckendorf [1901 – 1983], médecin belge, officier de l'armée et mathématicien.

Ex. 3 :

En utilisant une méthode gloutonne, déterminer à la main la décomposition de Zeckendorf des entiers 17 et 53.

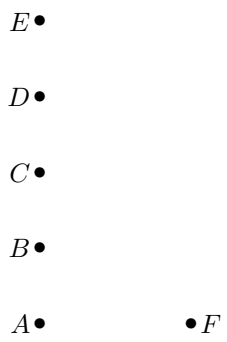
Ex. 4 :

1. Écrire une fonction `recherche(x,L)` prenant en entrée un entier naturel `x` et une liste `L` triée dans l'ordre croissant dont le premier élément est inférieur ou égal à x et le dernier strictement supérieur à x et qui renvoie le plus grand élément de la liste `L` qui soit inférieur ou égal à x.
2. Écrire une fonction `Zeckendorf(n)` qui associe à un entier naturel n une liste `T` d'entiers correspondant à une décomposition de Zeckendorf de n. On pourra procéder de la manière suivante :
   - définir une liste `L` contenant dans l'ordre, les termes de la suite $(F_n)_{n\in\mathbb{N}}$ qui sont inférieurs ou égaux à n ;
   - tant que n est différent de 0, obtenir le plus grand entier de `L` inférieur ou égal à n en utilisant la fonction `recherche` et le retrancher à n.

3 - Problème du voyageur de commerce

On suppose qu'un voyageur de commerce doit se déplacer dans N villes d'un pays. Comment doit-il organiser son voyage pour minimiser la distance parcourue ?

Si l'on cherche à tester toutes les possibilités, les calculs deviennent vite extravagants. En effet, le nombre de chemins possibles pour le voyageur est clairement $N!$. Si $N = 15$ cela fait 43589145600 possibilités à tester !
Pour déterminer le chemin du voyageur on adopte donc une stratégie gloutonne. On part d'une des villes on fait le choix localement optimal qui consiste après chaque ville visitée a aller ensuite dans la ville qui lui est la plus proche.



Ex. 5 :

Dans le graphique ci dessus, en partant de la ville A, quel est le chemin parcouru par le voyageur ? Est-ce le chemin optimal ?

On définit la fonction `dist(P,Q)` qui à deux points P et Q associe la distance euclidienne entre P et Q :

```
from math import sqrt
def dist(P,Q):
    x1,y1 = P
    x2,y2 = Q
    return sqrt((x1-x2)**2+(y1-y2)**2)
```

et la fonction `plus_proche(points,P)` déjà utilisée lors d'un TP précédent qui à une liste `points` de points distincts et un point `P` associe l'indice du point le plus proche de P dans la liste :

```python
def plus_proche(points,P):
    """renvoie l'indice du point le plus proche du point P"""
    n = len(points)
    d = dist(P,points[0])
    imin = 0
    for j in range(1,n):
        if d > dist(P,points[j]):
            imin = j
            d = dist(P,points[j])
    return imin
```

Ex. 6 :

Ecrire une fonction `chemin(points)` qui associe à une liste de points le chemin du voyageur de commerce en partant du premier point de la liste. On pourra procéder ainsi :

- Créer une liste `ch` contenant le premier point de la liste `points` ;
- Retirer ce point de `points` en utlisant la méthode `remove` ;
- Pour les points suivants, chercher le point le plus proche du dernier point dans la liste `points` à l'aide de la fonction `plus_proche`, l'ajouter à la liste `ch` et le retirer de la liste `points`.

Ex. 7 :

1. Proposer des instructions utilisant la fonction `randint` de la bibliothèque `random` pour créer une liste `points` de points distincts dont les abscisses sont compris entre -20 et 20.
2. Compléter les lignes suivantes pour tracer les points ainsi que le chemin du voyageur.

```python
ch = chemin(points)
import matplotlib.pyplot as plt
x = ...
y = ...
plt.plot(x,y,marker="o",linestyle='-')
plt.show()
```

4 - Le problème du sac à dos

On possède un sac à dos qui a une contenance maximale de 20kg. On veut y ranger un certain nombre d'objets. Chaque objet est repéré par son nom, sa valeur (en euros) et son poids (en kg). Voici la liste des objets :

Nom	Valeur	Poids
Objet1	115	14
Objet2	38	2
Objet3	21	6
Objet4	7	1
Objet5	20	7
Objet6	78	8

Pour sélectionner les objets à ranger, on procède de manière analogue au rendu de monnaie en ajoutant à chaque étape l'objet de plus grande valeur pouvant encore rentrer dans le sac.

Ex. 8 :

1. Créer une liste `objets` de tuples dont les éléments sont les triplets `(nom,valeur,poids)` rangés par ordre décroissant de valeur.
2. Ecrire une fonction `glouton(objets,poids_max)` qui prend en paramètres la liste des objets et le poids maximal du sac à dos et qui renvoie la liste des objets à ranger dans le sac à dos et la valeur totale des objets rangés.
   La liste étant rangée dans l'ordre décroissant des valeurs, on parcourt la liste des objets dans cet ordre et on ajoute un à un les objets dont le poids n'excède pas le poids maximal restant.
3. Quels sont les objets choisis pour un poids maximal de 20kg ? Quelle est la valeur totale des objets rangés ?

On pourrait aussi choisir de ranger les objets selon leur rapport valeur/poids. Pour cela, on réordonne la liste des objets en utilisant la méthode `sorted` avec une clé qui est la fonction `rapport`.

```python
def rapport(obj):
    return obj[1]/obj[2]
objets = sorted(objets,key=rapport,reverse=True)
```

Ex. 9 :

1. Quels sont les objets choisis dans ce cas là ? Quelle méthode est la meilleure pour obtenir une valeur maximale ?

2. On peut aussi chercher à remplir le sac en maximisant le nombre d'objets à ranger, sans se soucier de leurs valeurs.
   Proposer des instructions pour ranger les objets dans l'ordre croissant de leur poids et pour afficher la liste des objets choisis par cette méthode.