Exercice 1 – Ordre lexicographique

http ://mathematice.fr/ – Serge Bays

L'objectif est d'écrire un programme qui trie une liste de mots et les range dans l'ordre lexicographique (ordre des dictionnaires).

1. Écrire la définition de la variable "alphabet" : `alphabet="AaàBbCcDdEeéèFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuùVvWwX`
2. Écrire une fonction `ordre_alphabetique(c1,c2)` qui prend en argument deux caractères alphabétiques et renvoie -1 si `c1` est avant `c2`, 1 si `c2` est avant `c1` et 0 si `c1 = c2`. On pourra utiliser la méthode index qui renvoie l'indice d'un élément dans une chaîne de caractère.
3. Écrire une fonction `ordre_lexicographique(m1,m2)` qui prend en argument deux mots et renvoie -1 si "`m1 < m2`" pour l'ordre lexicographique, 0 si "`m1 = m2`" et 1 si "`m1 > m2`". On utilisera la fonction `ordre_alphabetique`.
4. Écrire une fonction `tri_lexicographique(liste)` qui prend an argument une liste de mots et renvoie la liste triée. On utilisera la fonction `ordre_alphabetique` et l'algorithme du tri par insertion.

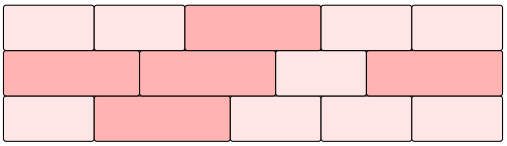
Correction

```
alphabet='AaàBbCcDdEeéèFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuùVvWwXxYyZz'
def ordre_alphabetique(c1,c2):
    if alphabet.index(c1)<alphabet.index(c2):
        return -1
    elif alphabet.index(c2)<alphabet.index(c1):
        return 1
    else:
        return 0

def ordre_lexicographique(m1,m2):
    n=min(len(m1),len(m2))
    for i in range(n):
        if ordre_alphabetique(m1[i],m2[i])==-1:
            return -1
        elif ordre_alphabetique(m1[i],m2[i])==1:
            return 1
    if len(m1)<len(m2):
        return -1
    elif len(m2)<len(m1):
        return 1
    else:
        return 0

def tri_lexicographique(liste):
    for i in range(len(liste)-1):
        k = i+1 # indice de la cle
        cle=liste[k]
        while ordre_lexicographique(cle,liste[k-1])==-1 and k>0:
            liste[k] = liste[k-1]
            k = k-1
        liste[k]=cle
        print("etape",i," :\t",liste)
    return liste

t=['moi', 'toi', 'bonjour','salut','bon']
print("liste :\t\t",t)
print("resultat : \t",tri_lexicographique(t))
```

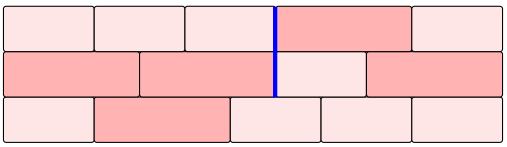
Exercice 2 : Problème de maçon

D'après ressources d'Anthony Meurdefroid.

On cherche à construire un mur à partir de briques horizontales de taille 2×1 et 3×1. Un mur est correctement construit si la jointure verticale entre deux briques ne se trouve jamais immédiatement au dessus d'une autre jointure verticale. Ainsi le mur suivant (de hauteur $h = 3$ et de longueur $l = 11$)est correctement construit. En revanche, le mur d'après ne l'est pas.



Exemple de mur valide

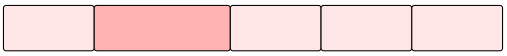


Exemple de mur non valide

L'objectif de ce problème est de dénombrer le nombre de façons différentes de construire un mur de longueur l et de hauteur h.

Codage du problème et fonctions utiles

Chaque rangée de briques (une ligne) de longueur l va être codée comme une chaîne de caractères, composée uniquement de `'0'` et `'1'`, de longueur $l-1$. Par exemple, la rangée de taille 11 est codée dans une chaîne de caractères de longueur 10.

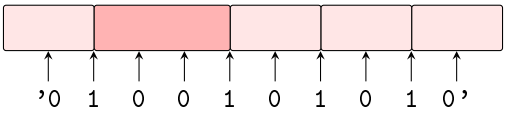


Exemple de rangée de briques

`'0100101010'`

Codage de la rangée de briques

Dans le détail, le $i^{\text{ième}}$ élément (chiffre) de la chaîne de caractère indique si au niveau de la $i^{\text{ième}}$ unité de longueur il y a oui ou non une jointure entre deux briques (`'1'` si il y a une jointure, `'0'` sinon). On n'inclut pas les deux extrémités - ce qui donne :



Question 1 *Dessiner la rangée de briques correspondant à la chaîne de caractère suivante :* `'0100100'`. *De quelle taille est cette rangée ?*

Correction

La rangée de briques est : 1 brique de taille 2, puis 2 de taille 3, soit une rangée de taille 8.

Question 2 *Écrire une fonction* `imprimeRangee(rangee)` *qui prend en argument une liste des deux chiffres 2 et 3 correspondant à une rangée de briques et qui renvoie la rangée de briques sous la forme codée (type* `string`*). Exemple :* `imprimeRangee([2,3,2,2,2])` *renvoie* `'0100101010'`.

Correction

```
def imprimeRangee(rangee):
    '''rangee est une liste de 2 et de 3'''
    c=''
    for element in rangee:
        if element==2:
            c+='01'
        else:
```

```
            c+='001'
    return c[:-1]
```

Deux rangées sont dites compatibles si elles peuvent être placées l'une sur l'autre, c'est-à-dire que les jointures ne sont pas au même endroit.

Question 3 *Écrire une fonction* `sontCompatibles(r1,r2)` *prenant en argument* `r1` *et* `r2`*, 2 rangées de briques sous la forme codée (type* `string` *- ex :* `'011100'`*), et qui renvoie* `True` *si les deux rangées données sont compatibles,* `False` *sinon.*

Correction

```
def sontCompatibles(r1,r2):
    for i in range(len(r1)):
        if r1[i]=='1' and r2[i]=='1':
            return(False)
    return(True)
```

Résolution du problème

Pour dénombrer le nombre de façons différentes de construire un mur de longueur l et de hauteur h, on adopte le principe de résolution suivant :

- **phase 1 :** déterminer toutes les combinaisons possibles des briques permettant de construire une rangée de longueur l. Par exemple, il n'y a que deux façons possibles pour les rangées de longueur 5 : une brique de longueur 2, puis une brique de longueur 3 ; et l'inverse. Les codages correspondants sont : `'0100'` et `'0010'`. Soit R l'ensemble de ces codages de rangées de briques ;
- **phase 2 :** pour chaque rangée r de R, déterminer le nombre de murs de hauteurs h dont la première rangée de briques (celle tout en bas) est r. Cette valeur est notée `compteRangee(R, r, h)` ;
- **phase 3 :** calculer $\sum_{r\in R}$ `compteRangee(R,r,h)` pour toutes les rangées possibles de longueur l. C'est le résultat final attendu.

Étude de la phase 1

Dans un premier temps, on aimerait connaître le nombre de combinaisons possibles distinctes afin de remplir la rangée de briques avec des briques élémentaires de dimensions 2 et 3.

Question 4 *Proposer une fonction récursive* `combinaisons(l)` *prenant en argument la longueur du mur* l *et renvoyant le nombre de solutions possibles distinctes afin de remplir une rangée. Vous pourrez construire votre algorithme en vous appuyant sur un schéma (arbre) ayant pour sommet* $l = 11$.

Correction

```
def combinaisons(n):
    if n==1:
        return(0)
    elif n==2 or n==3:
        return(1)
    else:
        return(combinaisons(n-2)+combinaisons(n-3))
```

Question 5 *Préciser la complexité de votre fonction.*

Correction On a : $C(n) = C(n-2)+C(n-3)$, alors on obtient une complexité en $C(n) = O(\lambda^n \times 2^n)$ soit une complexité exponentiell

Pour la phase 1, on suppose écrite une fonction `remplissage(l)` qui renvoie la liste de toutes les rangées possibles de longueur l. Par exemple, `remplissage(5)` renvoie les deux rangées : `'0100'` et `'0010'`.

Question 6 *Sans écrire aucun code, quelle serait la structure du code de la fonction* `remplissage` *?*

Correction
Il est nécessaire de créer une fonction récursive qui ajoute dans la branche de gauche une brique de 2 (concaténation de la chaîne de caractère en cours avec `'10'`), puis dans la branche de droite une brique de trois (concaténation de la chaîne de caractère en cours avec `'100'`). Il faut gérer la longueur du mur en cours (`w` dans le code ci-dessous) - ceci constituera notre cas de base dès lors que le mur en cours fera la longueur souhaitée. Enfin, il faut remplir une liste de toutes les combinaisons. J'ai utilisé une variable globale (très utile ici), en s'appuyant sur la génération pour une brique initialement de 2 puis une de 3. Le code à titre informatif :

```
def remplissage(w,r,tailleMax):
    if w==tailleMax:
        toutesLesCombis.append(r)
    elif w<tailleMax:
        remplissage(w+2,r+'10',tailleMax)
        remplissage(w+3,r+'100',tailleMax)

global toutesLesCombis
toutesLesCombis=[]

tailleMax=9

remplissage(2,'0',tailleMax)
remplissage(3,'00',tailleMax)
```

Étude de la phase 2

Pour dénombrer les murs (phase 2 de la résolution), on utilise une fonction récursive `compteRangee(R, r, h)` qui détermine le nombre de murs dont la première rangée de briques est r et dont la hauteur est h. L'objet de cette sous-section est de comprendre cet algorithme et son implémentation, qui sont donnés dans le fichier macon.py

Question 7 *Pourquoi la quantité* `compteRangee(R, r, 1)` *vaut-elle 1 pour tout r ?*

Correction Parce qu'il n'y a qu'une seule façon de faire un mur de hauteur 1 avec une rangée donnée!

Question 8 *On suppose $l = 5$. Développer sur votre copie les différentes étapes de calcul (avec les appels récursifs) de l'appel* `compteRangee(combinaisonsCinq,combinaisonsCinq[0],2)`, `combinaisonsCinq` *étant la liste comportant les deux chaînes de caractères,* `'0100'` *et* `'0010'`, *codant les rangées.*

Correction On va donc dérouler les appels récursifs (arbre) :
- appel à `compteRangee(combinaisonsCinq,'0100',2)`
- initialement, `somme` vaut 0 ; nbCombi vaut 2 ; i vaut 0
- i<2 ici 0<2 vrai
- SontCompatibles('0100','0100') False ; i=i+1=1
- i<2 ici 1<2 vrai
- SontCompatibles('0010','0100') True
  - appel à `compteRangee(combinaisonsCinq,'0010',1)`
  - h=1 ; la fonction renvoie 1 ; somme=somme+1=1
  - i=2 ; boucle while interrompue
- à la fin `somme` vaut 1, et on renvoie 1.

Étude de la phase 3

Afin de réaliser l'objectif, la fonction `compteTout(toutesLesCombinaisonsLigne,hauteur)` sera écrite.

Question 9 *Écrire la fonction* `compteTout(toutesLesCombinaisonsLigne,hauteur)` *prenant en argument une liste de chaînes de caractères et la hauteur du mur souhaitée, et renvoyant un entier nombre de combinaisons possibles afin de construire le mur d'une manière correcte.*

Correction def compteTout(toutesLesCombinaisonsLigne,hauteur) : nbCombi=len(toutesLesCombinaisonsLigne) somme=0 i=0 while i<nbCombi : temp=compteRangee(toutesLesCombinaisonsLigne,toutesLesCombinaisonsLigne[i],hauteur) somme+=temp i+=1 return(somme)

Dans cette sous-section, nous chercherons les limites de notre algorithme naïf proposé dans la sous-section précédente, et nous chercherons donc à l'améliorer.

Question 10 *En s'appuyant sur le calcul des murs de longueur 9 et de hauteur 3, donner le principal défaut de l'algorithme naïf proposé précédemment.*

Correction Pour le calcul des murs de longueur 9 et de hauteur 3, une bête réflexion au début de chaque appel récursif nous montre que l'on appelle deux fois `compteRangee(toutesLesCombinaisons,'00100100',2)`. Ce phénomène se répétant très souvent, il rend très lent notre algorithme.

Question 11 *Proposer alors une amélioration de l'algorithme.*

Correction Si on reprend l'exemple précédent, on peut donc effectuer ce calcul une fois, puis stocker ce nombre dans une table. Cela devient très fortement intéressant à partir du moment où l'on demande par exemple le calcul des murs de longueur 30 et de hauteur 10.