TD IPT^2 n° 1: Révisions 1/2

Rappels de programmation élémentaire: structures conditionnelles, boucles, manipulations des types, arithmétique

Procédures de base - structures conditionnelles

Exercice n°1: Nombres parfaits

Un entier naturel n est dit parfait s'il est égal à la somme de ses diviseurs naturels stricts. Par exemple, 6 est parfait puisque $6 = 1 + 2 + 3$.

❶ Ecrire une fonction `parfait` testant si le nombre passé en argument est parfait ou non.

❷ Trouver les nombres parfaits inférieurs à 10^5.

Exercice n°2: Suite ordonnée des nombres à petits diviseurs

En exploitant là-encore les opérateurs de division euclidienne et/ou de reste de la division euclidienne, construire une fonction prenant en argument un entier naturel non nul, et qui retourne la suite ordonnée des n plus petits entiers de la forme $2^p3^q5^r$, p, q, et r étant des entiers naturels ≥ 0.

On pourra par exemple présenter la liste sous la forme d'une liste de liste avec (p, q, r) indiqués dans un tuple:

```
[[n1,(p1,q1,r1)],[n2,(p2,q2,r2)],...[ni,(pi,qi,ri)],...]
```

Exercice n°3: Résolution d'une énigme par force brute

On propose dans cet exercice de résoudre l'énigme suivante:

Dans le tableau ci-contre, on veut remplacer chaque lettre par l'un des chiffres de 1 à 9, en tenant compte des contraintes suivantes:

$$\begin{cases} A+B+C = 2\times I+F \\ D+E+F = A+C+I \\ G+H+I = 2\times E+B \\ A+I = 8 \end{cases}$$

A	B	C
D	E	F
G	H	I

❶ Rédiger une fonction `listecorrecte(L)` retournant le booléen `True` si la liste L contient tous les nombres de 1 à 9 en un seul exemplaire, ou `False` sinon.

❷ *Option:* Résoudre **sur machine** le système d'équations fourni en prenant comme inconnues A, B, C, D.
On exploitera la commande `solve([equ1,equ2,...],inconnue1,inconnue2,...)` à importer du module `sympy.solvers`.

Attention: on veillera à déclarer les variables $A, B, C, D, E, F, G, H, I$ comme des symboles représentés par ces mêmes lettres à l'aide de la commande `symbols` du module `sympy`.

❸ Compléter le code suivant pour qu'il réalise la détermination de toutes les solutions à l'énigme par «force brute»:

Listing 1:

```
import time as t
debut=t.time()
##### Préparation de l'affichage des solutions #####
s="A={},␣B={},␣C={},␣D={},␣E={},␣F={},␣G={},
H={},␣I={}"
nbessais=nbsol = 0
for ----------:
    for ----------:
        for ----------:
            for ----------:
                for ----------:
                    nbessais+=1
                    ----------
                    ----------
```

```
                    ----------
                    ----------
                    if correct(----------):
                        nbsol+=1
                        print(s.format(A,B,C,D,E,F,G,H,I))
                        print('{} solutions pour {} essais'.format(nbsol,
    nbessais))
                        print(u"durée du calcul :", t.time()-début)
```

❹ *Option:* saisir et lancer le code précédent.

Manipulations de base sur les chaines

Exercice n°4: Vérification d'un palindrome

Un palindrome est un mot qui se lit de la même manière de gauche à droite ou de droite à gauche (par exemple « bob » et « laval » sont des palindromes).

Écrire une fonction `palindrome(ch)` qui teste si la chaîne de caractères `ch` est un palindrome en renvoyant `True` ou `False` suivant le cas.

Exercice n°5: Recherche d'acides aminés dans une chaîne d'ADN

On veut bâtir un script Python qui permet d'une part la saisie au clavier d'une chaine d'ADN valide, représentée par une chaîne composée exclusivement des lettres `"a"` (pour adénine), `"t"` (pour thymine), `"c"` (pour cytosine), `"g"` (pour guanine), soit les 4 bases azotées constituant l'alphabet génétique, et d'autre part d'analyser la présence d'une petite séquence (code pour une "micro"protéine) dans cette chaine.

❶ Ecrire une fonction `valide` qui renvoie `vrai` si la saisie de la chaîne d'ADN est valide ou `faux` si ce n'est pas le cas.

❷ Ecrire une fonction `saisie` qui effectue une saisie valide et renvoie la valeur saisie sous forme d'une chaîne de caractères.

❸ Ecrire une fonction proportion qui reçoit deux arguments, la chaîne et la séquence et qui retourne le nombre d'occurrences de cette séquence dans la chaîne, ainsi que sa proportion.

Exercice n°6: Recherche dans un texte

Dans les années 60, le code ASCII (American Standard Code for Information Interchange) est adopté comme standard de codage informatique des caractères. Il consiste en une table donnant la correspondance entre un caractère et son code qui est un entier. On y trouve le codage des caractères de contrôle (retour chariot, codage d'un bip sonore), alphabétiques (accentués ou pas), numériques, et de ponctuation, le tout sur 8 bits (1 octet), soit 256 caractères possibles, **numérotés de 0 à 255**. En outre, les caractères "A" à "Z" sont codés par des entiers successifs.

Dans l'exercice qui suit, on pourra exploiter les commandes `chr(i)` qui renvoie le caractère corrspondant à l'entier *i* dans la table ASCII et la commande `ord(c)` qui renvoie l'entier codant le caractère *c*.

❶ Écrire une fonction qui prend en argument un caractère (une chaîne de longueur 1) et renvoie ce caractère si c'est une lettre majuscule. Sinon la fonction renvoie le caractère correspondant à l'entier 0.

❷ Écrire une fonction `compte(s,c)` qui compte le nombre d'occurrences d'un caractère `c` dans une chaîne `s`.

❸ En utilisant la fonction `compte`, écrire une fonction `nb_lettres(s)` qui compte le nombre d'occurrences de chaque lettre majuscule dans la chaîne `s` et renvoie le résultat sous la forme d'un tableau de 26 cases.

❹ Combien de fois la chaîne `s` est-elle parcourue lorsque l'on exécute la fonction `nb_lettres` ?

❺ Ré-écrire la fonction `nb_lettres` pour qu'elle ne parcoure la chaîne `s` qu'une seule fois.

Exercice n°7: Découpage et recensement des mots dans un texte

On propose dans cet exercice de rédiger une fonction permettant de recenser les mots dans un texte, à l'instar de celles intégrées la plupart du temps dans les logiciels de traitement de texte.

❶ Ecrire une fonction `mot_suivant(expression,i)` qui prend en argument la chaine de caractère *expression*, un entier *i*, et qui:

- si $i < len(expression)$, retourne un tuple (mot, k) formé:
  - de la chaîne contigüe sans délimiteur qui débute en $j \geq i$ et de la position du mot suivant s'ils existent tous les deux;
  - de la chaîne contigüe sans délimiteur qui débute en $j \geq i$ et de la longueur totale de l'expression si seul le premier existe;
  - du mot vide et de la longueur de la chaîne sinon;
- renvoie un message d'erreur si $i \geq len(expression)$

NB: dans cette question, la fonction sera rédigée pour une chaine `expression` avec des espaces, mais sans ponctuation, **en veillant à ce que la présence de multiples espaces entre mots ou d'espace(s) en début ou fin de chaine soit également gérée**.

❷ Ecrire une fonction `liste_mots(expression)` qui prend en argument la chaîne `expression` et retourne la liste des mots composant cette chaîne.

❸ Le but étant d'obtenir la liste des mots délimités par des symboles de ponctuation, reprendre la fonction `mot_suivant` sous la forme `mot_suivant(ch,i,s)` où s est une liste de séparateurs, par exemple s = [””, ”, ”, ”;”, ”,”, ”.”, ”?”, ”!”]. Ajouter enfin un compteur de mots. Tester le code si vous êtes équipé d'une machine.

Manipulations de base sur les listes

EXERCICE N°8: Le crible d'Erathostène

Le crible d'Erathostène est un algorithme permettant de trouver tous les nombres premiers inférieurs à un entier N donné. Il existe deux versions de cet algorithme dont l'une, faisant appel à la récursivité, sera abordée dans un prochain chapitre.

Le principe est de bâtir une liste des entiers compris entre 2 et N, d'inscrire le premier de cette liste dans une liste de sortie, de le retirer de la liste, et de supprimer de celle-ci tous ses multiples. Le tri s'arrête lorsque la liste est vide. Ainsi la liste de sortie ne comportera que que les nombres premiers entre 2 et N.

Le site Wikipedia donne l'algorithme en pseudo-code suivant pour le crible d'Erathostène (en version non récursive):

```
Fonction Eratosthène(Limite)
    L = liste des entiers de 2 à Limite
    Tant que L est non vide
        Afficher le premier entier de L
        L = liste des entiers de L non multiples du premier
    Fin tant que
Fin fonction
```

❶ Bâtir le script Python de cet algorithme. On pourra exploiter la méthode `LIST.remove(arg)` qui permet de retirer à la liste `LIST` l'élément `arg`.

❷ Introduire un compteur qui s'incrémente à chaque boucle permettant d'évaluer le nombre d'itérations nécessaires en fonction de N.

❸ Conclure sur la "performance" de cet algorithme.

EXERCICE N°9: Recherche de répétitions dans une liste

Soit un entier naturel `n` non nul et une liste `t` de longueur `n` dont les termes valent `0` ou `1`. Le but de cet exercice est de trouver le nombre maximal de `0` contigus dans `t` (c'est à dire figurant dans des cases consécutives). Par exemple, le nombre maximal de zéros contigus de la liste `t1` suivante vaut `4`:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t[i]	0	1	1	1	0	0	0	1	0	1	1	0	0	0	0

❶ Ecrire une fonction `nombreZeros(t,i)`, prenant en paramètres une liste `t`, de longueur `n`, et un indice `i` compris entre `0` et `n-1`, et renvoyant :

$$\begin{cases} 0 \text{ si } t[i] = 1 \\ \text{le nombre de zéros consécutifs dans } \texttt{t} \text{ à partir de } t[i] \text{ inclus, si } t[i] = 0 \end{cases}$$

Par exemple, les appels `nombreZeros(t1,4)`, `nombreZeros(t1,1)` et `nombreZeros(t1,8)` renvoient respectivement les valeurs 3, 0 et 1.

❷ Comment obtenir le nombre maximal de zéros contigus d'une liste `t` connaissant la liste des `nombreZeros(t,i)` pour $0 \leq i \leq n-1$?
En déduire une fonction `nombreZerosMax(t)`, de paramètre `t`, renvoyant le nombre maximal de 0 contigus d'une liste `t` non vide. On utilisera la fonction `nombreZeros`.

Procédés aléatoires

EXERCICE N°10: Marche auto-évitante (d'après CCMP 2021)

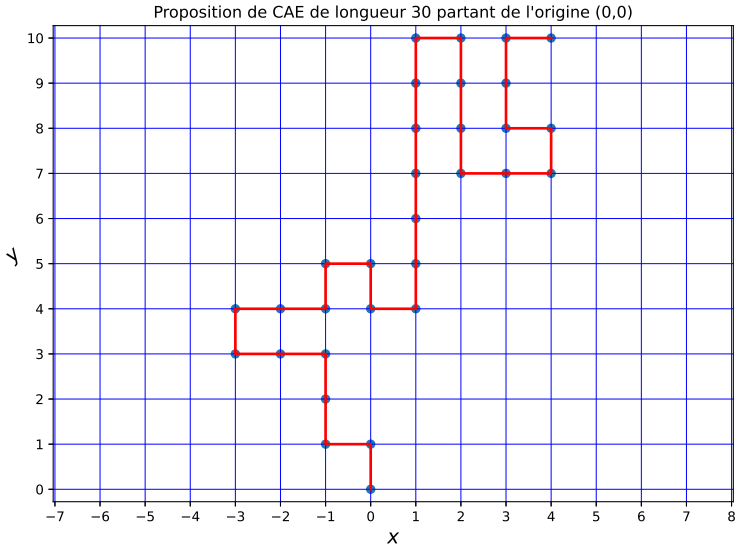
Une marche auto-évitante est un processus au cours duquel un point décrit un chemin auto-évitant, c'est-à- dire qui ne se recoupe jamais lui-même. Ce modèle peut servir entre autre dans la modélisation du comportement d'un polymère qui est une macromolécule constituée par l'assemblage en chaîne de petites molécules identiques, les monomères. En effet, deux monomères distincts de la chaîne ne peuvent pas se trouver à deux positions identiques pour des raisons d'encombrement stérique.

Dans cet exercice, on appellera chemin auto-évitant (CAE) de longueur n tout ensemble de $n+1$ points $P_i \in \mathbb{Z}^2$ pour $0 \leq i \leq n$ tels que:

- $\forall i \quad \|P_{i+1} - P_i\| = 1$
- $\forall (i,j) \quad i \neq j \Rightarrow P_i \neq P_j$

Chaque point du chemin sera représenté par une liste à deux éléments entiers précisant les deux coordonnées (par exemple $P_i = (5, -4)$ est représenté par la liste $[5, -4]$). Le chemin lui-même est constitué de la liste des points, dans l'ordre dans lequel ils sont atteints. Les codes Python produits dans cette partie devront manipuler exclusivement des coordonnées entières.

Ci-dessous un exemple de CAE:



On s'intéresse ici à une méthode naïve pour générer un chemin auto-évitant de longueur n sur une grille carrée. La méthode adoptée est une approche gloutonne:

1. Le premier point est choisi à l'origine: $P_0 = (0, 0)$.
2. En chaque position atteinte par le chemin, on recense les positions voisines accessibles pour le pas suivant et on en sélectionne une au hasard. En l'absence de positions accessibles l'algorithme échoue.
3. On itère l'étape 2 jusqu'à ce que le chemin possède la longueur désirée ou échoue.

Données:

- `randrange(a,b)` du module `random` renvoie un entier compris entre a et $b-1$
- `choice(L)` du module `random` également renvoie l'un des éléments de la liste L choisi aléatoirement selon une distribution de probabilité uniforme
- on supposera que l'expression `x in L` qui renvoie le booléen `True` ou `False` suivant la présence ou non de l'élément x dans la liste L, parcourt cette dernière de manière séquentielle.

En utilisant le canevas fourni ci-dessous et en ajoutant les imports nécessaires:

❶ Implémenter la fonction `positions_possibles(p, atteints)` qui construit la liste des positions suivantes possibles à partir du point p. La liste `atteints` contient les points déjà atteints par le chemin.

❷ Mettre en évidence graphiquement un exemple de `CAE` le plus court possible pour lequel, à une étape donnée, la fonction `positions_possibles` va renvoyer une liste vide. En prenant en compte les symétries, combien de tels chemins distincts existent pour cette longueur minimale ?

❸ Implémenter la fonction `genere_chemin_naif(n)` qui construit la liste des points représentant le chemin auto-évitant de longueur n et renvoie le résultat, ou bien renvoie la valeur spéciale *None* si à une étape `positions_possibles` renvoie une liste vide.

Listing 2: Canevas pour les fonctions positions_possibles et genere_chemin_naif

```
# import Python à compléter...
# positions auto-évitantes suivantes possibles
def positions_possibles(p, atteints):
        possibles = []
        À compléter...
        return possibles

# génération gloutonne d'un chemin de longueur n
# renvoie None en cas d'échec
def genere_chemin_naif(n):
        chemin = [ [ 0, 0 ] ] # on part de l'origine
        # À compléter...
        return chemin
N=10
print("chemin", genere_chemin_naif(N))
```

Exercice n°11: Le paradoxe des anniversaires

Lors d'un mariage réunissant une centaine d'invités, deux convives évoquent ensemble leurs dates d'anniversaire et constatent avec stupéfaction qu'ils sont nés le même jour (mais pas forcément de la même année). Bien que surprenante, cette coincidence est loin d'être exceptionnelle. On se propose d'en étudier la probabilité.

❶ Si l'on considère un groupe de N personnes, quelle est la probabilité qu'au moins deux d'entre-elles soient nées un même jour. On pourra dans un premier temps calculer la probabilité que toutes les personnes soient nées un jour différent. (On ne tient pas compte des années bissextiles, en d'autres termes: pas de naissance le 29 février)

❷ Ecrire une fonction python `para_anniversaires(N)` qui renvoie la probabilité qu'au moins deux personnes parmi N aient leur anniversaire le même jour. Faire l'application numérique dans le cas de notre mariage. Commenter.

Exercice n°12: Méthodes de Monte-Carlo

Les méthodes dites "de Monte Carlo" visent à évaluer numériquement des intégrales, donc indirectement des surfaces et des volumes (suivant la dimension de l'intégrales), par des procédés de tirages aléatoires.
Le principe est de tirer au hasard à chaque itération deux (respectivement trois) coordonnées x et y (respectivement z) d'un point M, et de vérifier si ce point M tombe **à l'intérieur** de la surface (respectivement le volume) à évaluer ou **à l'extérieur.**
Pour tous les tirages, le point doit de toute façon tomber dans une surface gabarit contenant la surface à évaluer. Pour un nombre de tirages important, le rapport du nombre de points tombés dans la surface sur le nombre total de tirages tend vers le rapport de la surface à évaluer sur la surface gabarit.

❶ Analyser "à la main" le script suivant exploitant une "méthode Monte Carlo 2D", et découvrir ce qu'il retourne finalement:

Listing 3: Sources_Python/MonteCarlo_1.py

```
# -*- coding: utf-8 -*-
""" MonteCarlo version 1"""
from random import *
import numpy as np
pi=np.pi
erreur=0
while not(erreur>=1e-9):
    erreur=input(u"Entrer la prÃ©cision desirÃ©e pour ce calcul (e>1E-9): ")
Nint=0
N=0
resexp=0
while abs(pi-resexp)>erreur:
    x=random()
    y=random()
    if np.sqrt((x-0.5)**2+(y-0.5)**2)<=0.5:
        Nint=Nint+1
        N=N+1
        resexp=Nint/(0.25*N)
    else:
        N=N+1
        resexp=Nint/(0.25*N)
    print(resexp)

print(u"La valeur approchÃ©e recherchÃ©e est: "),resexp
```

❷ Proposer une version 3D de cet algorithme.

NB: la commande `random.random()` du module `random` génère un nombre aléatoire en virgule flottante compris dans l'intervalle [0.0, 1.0].

Arithmétique

Exercice n°13: Structure du code INSEE

Le numéro INSEE est formé d'un nombre A (de 13 chiffres) porteurs de certaines informations sur l'état civil (sexe, date et lieu de naissance,. . .) suivi d'un nombre à deux chiffres K (comme "Key"), qui est une clef de détection d'erreur dans l'un des chiffres.

Le clé K (les 14$^{\text{ième}}$ et 15$^{\text{ième}}$ chiffres du code) est telle que l'on doit vérifier la condition:

$$A + K = 0[97]$$

sexe	an	mois	Dep	Loc	Rang	K (Key)
1	46	02	45	207	352	XX

Le premier chiffre ne peut prendre que la valeur 1 (pour homme) ou 2 (femme).

Ecrire un script Python qui vérifie le bon formatage d'un numéro INSEE saisi (contenant 13 chiffres et commençant par 1 ou 2) et calcule sa clef.

Exercice n°14: Vérification des codes barres

Le code *UPC* (universal product Code) utilise des nombres de 13 chiffres pour désigner un produit de consommation. Les 12 premiers chiffres désignent le produit le 13$^{\text{ième}}$ est une clef de contrôle.

Ainsi, si a_i désigne le chiffre de rang i du nombre A, soit:

$$A = a_{13}a_{12}\dots a_2a_1$$

alors la clé a_{13} est telle que:

$$3\left(\sum_{k=1}^{6} a_{2k}\right) + \sum_{k=0}^{6} a_{2k+1} = 0[10]$$

❶ Ecrire un script Python qui calcule la clé étant donné les 12 chiffres du produit.

❷ Modifier le script précédent afin de vérifier si le code barre est correct ou pas.

EXERCICE N°15: Technique du hachage des chaines

On rappelle pour cet exercice la structure du type dictionnaire[1] (type `dict`). Il s'agit d'un ensemble de couples de forme:

`<clé>:<valeur>`

les délimiteurs du dictionnaire étant les accolades ({.....})

$$\{\underbrace{\text{"}Tintin\text{"}}_{\texttt{clé}} : \underbrace{25}_{\texttt{valeur}}, \text{"}Haddock\text{"} : 55, (127, 255) : (7, 8), \text{"}pouet\text{"} : \text{"}trompette\text{"},$$
$$\text{"}ding\text{"} : \text{"}triangle\text{"}, \text{"}bong\text{"} : \text{"}gong\text{"}, \text{"}chariot\text{"} : \texttt{"supermarché"}, \text{"}haricot\text{"} : \texttt{"boite de conserve"}\} \quad (1)$$

NB: les clés peuvent être de tout type, **hormis une liste**. Les valeurs peuvent en revanche être rigoureusement de tout type.

[1] Nous reviendrons sur les `types` en Python lors du chapitre 2 de révisions

La recherche d'une donnée dans un dictionnaire (ensemble "clé:valeur") par utilisation de la clé nécessite le balayage de l'ensemble du dictionnaire par une boucle. Cette opération est quasi-instantanée lorsqu'il s'agit d'un ensemble de quelques éléments, typiquement de l'ordre de 10, mais peut évidemment réclamer beaucoup plus de temps machine si le dictionnaire comporte par exemple 10000 éléments, et que l'entrée recherchée se trouve en fin de dictionnaire.

Les listes (ou tableaux) ne présentent pas cet inconvénient puisqu'il suffit de connaitre l'indice d'un élément dans le tableau pour extraire instantanément celui-ci.

Le hachage consiste à associer un entier à chacune des clés du dictionnaire, et ce de manière unique afin de permettre une indiciation des valeurs du dictionnaire exactement comme pour les éléments d'une liste (ou d'un tableau); cette nouvelle liste constituée est appelée **table de hachage** et permet donc l'extraction immédiate d'une valeur du dictionnaire en utilisant l'indice entier correpondant à la clé.

L'exercice qui suit propose d'illustrer la technique de hachage des clés d'un dictionnaire.

Considérons des clés constituées des caractères de l'alphabet ASCII, soit 256 caractères (codage sur 1 octet) et associons à chaque clé un entier correspondant à la décomposition de la clé en base 256.

Par exemple, on fait correspondre à la clé `"pouet"` constituée des caractères `p, o, u, e, t` de valeurs ASCII respectives `112, 111, 117, 101, 116` l'entier suivant:

$$n = 112 \times 256^4 + 111 \times 256^3 + 117 \times 256^2 + 101 \times 256 + 116 = 482906301812$$

❶ Ecrire une fonction Python "naïve" `chaine_entier(ch)` qui prend en argument la chaine `ch` et renvoie l'entier correspondant, en exploitant directement l'opérateur d'exponentiation (`**`).

❷ On constate rapidement que cette méthode est très coûteuse en temps machine, aussi on se propose de réaliser cette fonction en utilisant le schéma de Horner permettant le calcul rapide d'un polynôme en x_0; en effet:

le polynôme:

$$P(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \dots a_{n-m} + \dots + a_1 x + a_0$$

peut s'écrire également:

$$P(x) = ((((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + a_{n-3}) \ldots)x + a_0$$

Ecrire la nouvelle fonction Python `chaine_entier_Horner(ch)` qui réalise le même travail que `chaine_entier(ch)` en exploitant le schéma de Horner.

❸ Ecrire une fonction Python `entier_chaine(int)` qui réalise l'opération inverse.

❹ Le problème de ce programme de conversion est qu'il engendre des entiers de grande valeur, gourmands en espace mémoire (entiers dits "longs", qui apparaissent avec un suffixe "L" lorsqu'ils sont renvoyés par Python; faites l'essai sur machine, vous constaterez ceci).

Pour éviter cela, on va procéder à une opération dite de "hachage" qui consiste à retourner **le reste** de la division Euclidienne de l'entier généré par la profondeur de la table de caractères (256 caractères numérotés de 0 à 255); la fonction de hachage est donc:

$$\begin{array}{rcccc} h & : & \mathbb{N} & \to & \mathbb{N} \\ & & n & \mapsto & (n \; mod \; 255) = n \; \% \; 255 \end{array}$$

a· Ecrire la fonction `h(ch)` en Python d'argument la chaîne `ch` qui réalise la conversion précédente en entier puis le hachage.

b· Question pratique: si vous êtes équipé d'une machine, utilisez la fonction `h(ch)` sur la chaine (clé) `"pouet"`. Faire de même avec les clés `"chariot"` et `"haricot"`; que constatez-vous sur ces deux derniers cas?

❺ En remarquant que:
$256 \equiv 1$ [255]
et que par conséquent $n \times 256 \equiv n$ [255] et finalement $256^k \equiv 1$ [255], expliquer le surprenant résultat de conversion des clés `"chariot"` et `"haricot"`.

❻ On constate donc que la présence de clés "anagrammes" perturbe le fonctionnement de l'indexation. Proposer une fonction python `dico_valide(dict)` qui prend en argument le dictionnaire et vérifie la présence éventuelle de clés anagrammes.

NB: dans tout l'exercice, on pourra exploiter les commandes `ord(char)` et `chr(n)` qui renvoient respectivement le code ASCII du caractère (chaîne!) et le caractère correspondant à l'entier `n`.