Je sais comment il fait…

Il ne développe pas la combinatoire ainsi que nous avons commencé par le faire.

Pas de regret.

C’est très pédagogique.

1. On essaye - et on arrive - à écrire une fonction récursive qui énumère tous les anagrammes d’une chaine – Il y en a n ! (factorielle n) si n est le nombre de lettres. Si on a un dictionnaire et j’en ai trouvé un qui contient plus de 4 millions de mots français, incluant les formes fléchies (pluriel, féminin, conjugué), on élimine des anagrammes trouvés ceux qui ne sont pas dans le dictionnaire. Sauf que : problème. Dès qu’on prend un mot un peu long et que l’on prétend mettre tous ses anagrammes dans une liste en mémoire, ça pète !  Plus de place dans la mémoire.

D’où, réflexion, *sur le métier, remettre l’ouvrage*…

1. On se dit que l’on veut les anagrammes, mais qu’on ne veut pas les stocker tous à la fois, donc : au lieu de faire une fonction qui les liste, on fait un générateur, qui en donne un à la fois, jusqu’à tant qu’il n’y en ait plus. Pour le coup, c’est complexe, parce que l’algorithme qui énumère les permutations et donne la suivante est plutôt de forme itérative et m’a quand même pris ½ journée de réflexion…  Faire un générateur. Avec le générateur, faire un itérateur. Puis pour chaque permutation (cad « anagramme ») trouvée, regarder s’elle est dans le dictionnaire de 4,+ Millions de mots.

Et là, on voit que pour les mots courts, on a les mêmes résultats que par l’algorithme récursif et que pour les mots longs, cela ne plante plus !!!

Victoire !

Sauf que, ça prend du temps.

Sur mon i7, tour, Win7 à la maison –en faisant autre chose à côté- j’ai du laisser passer plus de 40 minutes pour énumérer tous les anagrammes de « maraudeur » et obtenir la réponse que seul « maraudeur » est dans le dictionnaire.

Trop c’est trop. On teste 400 000 combinaisons… contre un dictionnaire de 4,+ Millions de mots

Donc, on finit par se dire qu’il faut organiser le dictionnaire…

*100 fois sur le métier, remettre l’ouvrage… One more time !*

1. Le dictionnaire… Un fichier texte en utf-8. On le lit. Il va dans une chaine. On éclate la chaine sur les fins de lignes pour avoir les mots qui se retrouvent dans une liste.

Et la performance est à mourir de vieillesse et d’ennui, parce que lorsque l’on recherche un mot dans le dico, on parcourt la liste séquentiellement. En moyenne, on trouve à la moitié et quand on ne trouve pas, on est obligé  d’aller jusqu’à la fin. Car on fait une recherche séquentielle dans une liste non triée.

On pourrait s’assurer que la liste est triée et faire une recherche séquentielle dans une liste triée. Cela améliorerait la performance des recherches infructueuses : dès que l’on dépasse la place à laquelle un mot devrait se trouver, ce n’est plus la peine de continuer, on est sûr qu’il n’est pas dans le dico.

Mais c’est du gagne-petit et cela demande du code pour réécrire ce que fait le mot clé «  x in liste » de Python avec une boucle… donc, on préfère une meilleure idée !

Au lieu de laisser le dictionnaire dans la liste, on le convertit en un ensemble (« set ») avec pour effet de nous débarrasser d’éventuels doublons et d’améliorer la performance de manière très notable parce que « x in set » de Python est beaucoup plus performant que « x in liste ». C’est en effet, un accès direct aux éléments par une méthode de hash-coding qui est utilisée probablement par Python.

Mais même comme ça on doit faire l’analyse des 400 000 combinaisons pour un mot de 9 lettres. Le temps est non négligeable. Il empire pour des mots plus longs de manière exponentielle.

Comment étudier moins de combinaisons, voire, ne pas développer du tout de combinatoire ?

*Tzi ! Tzi !   Tzé potzible …*

Je reprends l’exemple de « maraudeur ».

Il ne faut pas chercher « maraudeur » mais construire, comme les chimistes ou comme les joueurs de Scrabble, la « formule brute » qui donne la composition de ce mot. J’explique : dans ce mot, il y a 2 fois « a », 1 fois « d », 1 fois « e », 1 fois « m », 2 fois « r » et 2 fois « u ». Noter que j’ai suivi l’ordre de l’alphabet. La formule brute est donc : a2d1e1m1r2u2.

Pour la petite histoire, pour les chimistes, cela marche pareil : l’eau, c’est H2O (2 Hydrogènes et un Oxygène), le méthane, CH4(1 Carbone et 4 Hydrogènes) etc. Les chimistes ne mettent pas les 1 : on n’écrit pas H2O1 mais seulement H2O.

Donc, au lieu de trier le dictionnaire dans l’ordre alphabétique des mots, il faut l’organiser par formule brute et, pour chaque entrée, donner la liste des mots qui correspondent à cette formule. Alors c’est un peu fastidieux à exemplifier, mais si je prends un mot très court comme « en » qui a comme anagramme « ne », dans le dico optimisé j’aurai !

 e1n1  =>  [ en,  ne ]

Donc, on me donne un mot, je calcule sa formule brute. Ensuite je cherche dans un dico organisé par formule brute. Si je ne trouve rien, c’est que ce n’est pas un mot du dictionnaire. Si je trouve quelque chose, j’ai tout de suite tous les anagrammes du mot. C’est instantané !

Sauf que…

Le dico n’est pas organisé comme cela. Il faut le faire et cela prend du temps. Mais on le fait une fois. Eventuellement on le sauve dans un fichier pour ne pas recommencer à chaque fois qu’on lance l’appli et voilou, on est les rois du Pétrole !

Comment organiser le dico dans  la mémoire ?

On va vite aller vers une structure de dictionnaire avec la clé qui sera la formule brute et la valeur qui sera une liste triée ou un ensemble, comme on voudra.

Comment écrire ce dico optimisé dans un fichier pour qu’il soit relu facilement ? rapidement ?

Facilement ou rapidement : il va falloir choisir.

*100 fois sur le métier,* remettre le bazar…

La facilité ce sera d’employer un fichier texte, facile à consulter avec l’éditeur de texte, facile à corriger et à enrichir.

La rapidité ce sera d’employer un fichier binaire… faire revenir le dictionnaire du disque en mémoire (sérialisation, désérialisation) ou bien construire une structure d’index spécifique et inclure les entrées sorties dans l’application. Il y a plein de possibilités toutes plus pédagogiques ( !) les unes que les autres…

Donc, oui, c’est un très bon exercice. En plein dans l’air du temps…C’est du bon usage de Python et je te suis très reconnaissant de me l’avoir fait connaître.  
J’en parlerai sûrement en cours et je t’invite à le creuser. Je n’ai pas mis de code dans ce qui précède pour ne pas te priver de l’apprentissage que cela comporte.

Je relirai volontiers ce que tu as écrit et le moment venu, je te ferai part de mes propres élucubrations.

Amitiés !

Georges

**De :** Georges Georgoulis [<mailto:ggeorgoulis@alteractifs.org>]   
**Envoyé :** jeudi 10 octobre 2019 19:28  
**À :** 'Benjamin frizon de lamotte'  
**Objet :** RE:

Ok pour en parler quand tu veux !

**De :** Benjamin frizon de lamotte [<mailto:bfdlmdr@gmail.com>]   
**Envoyé :** jeudi 10 octobre 2019 18:50  
**À :** Georges Georgoulis  
**Objet :** Re:

Merci Beaucoup !

Je reprends ça un peu plus tard.

(quand j'aurais digéré les cours de la semaine : )

Le mer. 9 oct. 2019 à 20:13, Georges Georgoulis <[ggeorgoulis@alteractifs.org](mailto:ggeorgoulis@alteractifs.org)> a écrit :

Bonsoir Benjamin,

Voici toutes les permutations d’une chaîne de caractères.

Ok pour en discuter avec toi demain ou quand tu veux !

Amitiés !

cid:image001.png@01D4909E.030308B0

Georges Georgoulis – Ingénieur Informatique et Télécoms

06 12 68 40 06 – [ggeorgoulis@alteractifs.org](mailto:ggeorgoulis@alteractifs.org) - [www.alteractifs.org](http://www.alteractifs.org/)

"""

    generate all possible anagrams of a string, ie, permutations of the chars

"""

def permut(s):

    #

    # 1 trivial case - easy case found

    #

    if len(s) <= 1:

        return list(s)

    #

    # 1 general - recursive case - method

    # for all chars in s, pick-up one char and assemble an anagram

    # with this char first and each permutation of the remaining chars

    # of the original string

    #

    l=[]

    for ic in range(len(s)):

        c=s[ic] #char at ic

        sic = s[:ic]+s[ic+1:] #string privated from char @ ic

        l+=list(map(lambda x: c+x,permut(sic)))

    return l

for m in ['', 'a', 'ab', 'abc', 'abcd'] :

    print (permut(m), len(permut(m)))

|  |  |
| --- | --- |
| [Image supprimée par l'expéditeur.](http://www.avg.com/email-signature?utm_medium=email&utm_source=link&utm_campaign=sig-email&utm_content=emailclient) | Garanti sans virus. [www.avg.com](http://www.avg.com/email-signature?utm_medium=email&utm_source=link&utm_campaign=sig-email&utm_content=emailclient) |