NOTIONS DE COMPLEXITE
RECHERCHE UN TABLEAU

I) Complexité d'un algorithme

Plusieurs algorithmes pour un même problème

Pour traiter un problème, nous n'avons pas forcément une seule méthode pour le résoudre. En effet, même si deux algorithmes sont corrects et répondent bien au probleme posé (ce que l'on peut vérifier à l'aide ***d'invariant de boucle*** par exemple), ils ne se valent pas nécessairement : bien sûr, on peut vouloir rechercher la clarté de l'algorithme, du choix des variables ou des fonctions annexes écrites ou importées, mais on peut aussi considérer le temps de réponse ou la place mémoire prise pour arriver au résultat. L'objet de l'étude de la complexité des algorithmes est d'évaluer le temps d'exécution de ces algorithmes (ce que l'on appelle ***la complexité en temps***) voire d'évaluer la quantité de mémoire utilisée par ces algorithmes (***la complexité en mémoire***) ce qui peut d'ailleurs également influer sur le temps d'exécution.

Prenons un exemple : on veut déterminer l'ensemble des diviseurs d'un entier. Nous disposons des deux algorithmes suivants :

```
def diviseurs1(n):
    for i in range (1, n+1):
        if n % i == 0:
            print(i)
```

```
def diviseurs2(n):
    for i in range (1, int(n**0.5)+1):
        if n % i == 0:
            print(i)
            if i**2 != n:
                print(n // i)
```

La première procédure effectue n tests de divisibilité et p affichages (avec p est le nombre de diviseurs de n) : on considère les n entiers compris entre 1 et n et on cherche ceux pour lesquels les restes de la division euclidienne de n par ceux-ci sont nuls.
La seconde procédure effectue un calcul de $\sqrt{n}$, $\sqrt{n}$ tests de divisibilité, p/2 multiplications et p affichages (mais pas dans le même ordre…) : en effet on constate que si d divise n alors n//d est aussi un diviseur de n et de plus, l'un des deux est inférieur ou égal à $\sqrt{n}$.

En supposant que toutes les opérations élémentaires effectuées ont une durée d'exécution équivalente (ce qui n'est pas totalement exact…) et en constatant que le nombre de diviseurs d'un entier n est "négligeable" devant n (et aussi devant $\sqrt{n}$), le nombre d'opérations élémentaires pour trouver les diviseurs de n (et donc sa durée d'exécution) est à peu près proportionnel à n pour la première procédure, et à $\sqrt{n}$ pour la seconde.

On dit que la complexité du premier algorithme est en O(n) et le second en $O(\sqrt{n})$

Pour ce qui est de la complexité en mémoire, les deux algorithmes sont équivalents car il n'utilise qu'une seule variable i qui change à chaque itération. Si l'on avait gardé en mémoire la liste de tous les diviseurs de n (il faut remplacer "print(i)" par L += [i], la liste L ayant été initialisée avant la boucle), la quantité de mémoire utilisée est de l'ordre de p… que l'on ne connait pas précisément : si n est un nombre premier, on a p = 2… mais il y a d'autres cas..

Complexité dans le meilleur des cas, dans le pire des cas, en moyenne.

Dans l'exemple précédent, la complexité en temps se calculait aisément en fonction de n… mais pas la complexité en mémoire (en supposant que l'on crée la liste des diviseurs et que l'on se contente pas de les afficher)…

En fait, dans le cas le plus simple (lorsque n est premier), la complexité en mémoire est 2 i.e. en O(1). On dit que ***la complexité*** (en mémoire) ***dans le meilleur des cas*** est O(1)

Dans le cas où il y a plus de diviseurs, c'est-à-dire lorsque n est divisible par 2, 3, 4, 5, 6, 7, 8, …., le calcul du nombre d(n) de diviseurs de n est plus compliqué. Pour $n < 10^{25}$, on montre que l'on a toujours : $d(n) < (2 * \ln(n))^3$. Ainsi ***la complexité dans le pire des cas*** est en $O((\ln(n))^3)$ pour des entiers ayant moins de 25 chiffres.

Enfin, on peut montrer mathématiquement que, si on note S(n) la somme des d(k) pour k entre 1 et n, on a : $S(n) \sim n \ln(n)$. Ainsi, la complexité de mémoire en moyenne (soit S(n) / n), est de l'ordre de ln(n) : ***la complexité en moyenne*** est donc en O(ln(n)).

II) Recherche dans un tableau

Recherche d'un élément dans un tableau

On veut savoir si la valeur **a** appartient au tableau **tab**. Pour ce faire on a plusieurs algorithmes possibles.

Un premier algorithme consiste à décrire un à un les éléments du tableau et à s'arrêter lorsque l'on a trouvé l'élément **a** … ou lorsque l'on a visité tout le tableau…

Un programme correspondant peut être celui-ci (à gauche une version répondant True ou False, à droite une version répondant False si **a** n'est pas dans **tab** et l'indice de **a** dans **tab** sinon)

```
def appartient(a, tab) :
    for i in range (len(tab)) :
        if tab[i] == a :
            return (True)
    return (False)
```

```
def indice(a, tab) :
    for i in range (len(tab)) :
        if tab[i] == a :
            return (i)
    return (False)
```

Si on note n la taille du tableau, la complexité (en temps) de cet algorithme est :

- 1 donc O(1) dans le meilleur des cas (si a est au début de tableau)
- n donc O(n) dans le pire des cas (si a n'est pas dans le tableau ou si a est à la fin du tableau)
- environ (n+1) / 2 donc O(n) en moyenne (en supposant qu'il y a autant de chances que a soit dans le tableau qu'à une place quelconque du tableau)

Recherche dichotomique d'une valeur dans un tableau trié

Un second algorithme, beaucoup plus rapide que le précédent n'est valable que dans un tableau trié. Il consiste à découper le tableau en deux et tester si **a** est à chercher dans le premier sous-tableau ou dans le second. On effectue alors le même travail sur le sous-tableau sélectionné. On réitère le processus tant que le sous-tableau sélectionné possède au moins 2 éléments.

Un programme correspondant peut être celui-ci :

```
def recherche_dichotomique (a, tab) :
    gauche , droite = 0, len(tab)
    while gauche <= droite :
        m = (gauche + droite) // 2
        if a == tab[m] :
            return (m)
        if tab[m] < a :
            gauche = m - 1
        else :
            droite = m + 1
    return (False)
```

Si on note n la taille du tableau, la complexité (en temps) de cet algorithme est :

- 1 donc O(1) dans le meilleur des cas (si a est au milieu du tableau)
- $\log_2(n)$ donc O(ln(n)) dans le pire des cas (si a n'est pas dans le tableau ou s'il est au début)

III) Recherche d'un mot dans une chaine de caractères

Recherche d'un mot dans un texte

On cherche une séquence de valeurs dans un tableau, ou, ce qui revient au même, un mot dans une chaine de caractères..

L' algorithme basique (dit *naïf*) consiste à décrire une à une les positions possibles du mot dans la chaine, et, pour une position donnée, tester si le k-ième caractère de mot correspond au k-ième caractère de la sous-chaine commençant à la position testée.

Un programme correspondant peut être celui-ci : (à gauche une version répondant True ou False, à droite une version répondant False si **a** n'est pas dans **tab** et l'indice de **a** dans **tab** sinon)

```
def recherche_mot (mot, chaine) :
    for i in range (1 + len(chaine) – len(mot)) :
        j = 0 :
        while j < len(mot)  and  mot[j] == chaine[i + j] :
            j = j + 1
        if j = len(mot) :
            return (i)
    return (False)
```

Si on note n la taille de **chaine** et p celle de **mot**, la complexité de cet algorithme est :

- p dans le meilleur des cas (si le mot est au début de la chaine)
- p * (n – p + 1) dans le pire des cas (si on cherche le mot XXXXX….XY dans un texte uniquement constitué de XXXX….XXXX)

La plus grande complexité dans le pire des cas est obtenue pour n = 2p et est donc en $O(n^2)$

Remarque : amélioration de la lisibilité en utilisant le test d'égalité de chaine de caractères

```
def recherche_motbis (mot, chaine) :
    for i in range (1 + len(chaine) – len(mot)) :
        if mot == chaine[i : i + len(mot)] :
            return (i)
    return (False)
```

IV) Quelques améliorations spécifiques à Python

enumerate

Lorsque tab est un tableau, enumerate(tab) donne le tableau bidimensionnel des couples : indice, valeur. Par exemple enumerate([5,8,9,4]) donne [[0,5], [1,8], [2,9], [3,4]]

Ainsi la fonction vue précédemment et appelée indice, peut s'écrire

```
def indice(a, tab) :
    for i, y in enumerate(tab) :
        if y == a :
            return (i)
    return (False)
```

méthodes de l'objet list

Lorsque **tab** est une liste, on possède des méthodes Python s'y appliquant :

méthode	effet	syntaxe	effet avec x = 3 et tab = [2, 6, 3, 5, 5, 3 ,3]
sort	trie le tableau dans l'ordre croissant (si ses données sont numériques)	tab.sort()	[2, 3, 3, 3, 5, 5, 6]
index	donne l'indice de la première occurrence de x dans le tableau	tab.index(x)	2
count	compte le nombre d'occurrences de x dans le tableau	tab.count(x)	3

V) Exercices

1) Déterminer les complexités en mémoire et en temps des fonctions suivantes

```
def fonc1(n):
    L = []
    for k in range(11):
        L += [k * n]
        print (k * n)
    return (L)
```

```
def fonc2(n):
    L = []
    for k in range(n):
        L += [k * n]
        print (k * n)
    return (L)
```

```
def fonc3(n):
    L = []
    for k in range(1, n):
        for j in range(k):
            print (k * j)
        L += [k * n]
    return (L)
```

2) Expliquer ce que fait la fonction suivante et déterminer les complexités en temps dans le pire et le meilleur des cas (et en moyenne..) : (on supposera que n est un entier naturel supérieur ou égal à 2)

```
def fonct(n):
    for k in range(2, int(n)**0.5 + 1):
        if n % k != 0:
            return (False)
    return (True)
```