Recherches et Cribles

D'après un TP de Stéphane Gonnord

Buts du TP

- Programmer des algorithmes élémentaires de recherche dans un tableau ou une chaîne de caractères.
- Réaliser des tests simples mais convaincants pour valider ou invalider des petites fonctions qu'on vient d'écrire.
- Mettre en place le crible d'Ératosthène.

1 Recherche d'un élément dans un tableau

EXERCICE 1 *Voici le code d'une fonction mystere :*

```
def mystere(tab, seuil):
    k = 0
    while k < len(tab) and tab[k] < seuil:
        k += 1
    return k == len(tab)
```

Et voici les valeurs retournées par deux appels de cette fonction sur des tableaux de 50 entiers tirés aléatoirement dans $[\![1;1000]\!]$.

```
>>> from random import randint
>>> tabalea = [randint(1, 1000) for _ in range(50)]
>>> mystere(tabalea, 998)
False
>>> tabalea = [randint(1, 1000) for _ in range(50)]
>>> mystere(tabalea, 998)
True
```

1. *Soit* `t` *un tableau de 50 entiers tirés aléatoirement dans* $[\![1;100]\!]$, *quelle est la probabilité que l'appel* `mystere(t, 99)` *retourne* `True` *?*
2. *Permuter les opérandes du* `and` *et tester au moins 4 appels* `mystere(tabalea, 998)` *comme ci-dessus. Que se passe-t-il ?*
3. *Ecrire une fonction* `test(tab,seuil)` *qui détermine si un tableau d'entiers* `tab` *contient au moins un élément supérieur à l'entier* `seuil`.

EXERCICE 2 *Écrire une fonction testant l'appartenance d'un objet à un tableau. Proposer un code avec une boucle* `for` *et un autre avec une boucle* `while`.

```
def appartient(x, t):
    ....

>>> appartient(42, [5, 12, 17, 42,])
True
>>> appartient(24, [5, 12, 17, 42])
False
```

EXERCICE 3 *Quel est, dans le pire des cas (à préciser !), le nombre maximum d'accès au tableau effectués pour tester l'appartenance de* x *à* T *?* (Par accès au tableau, on entend : consultation/modification via `... = t[i]` ou `t[i] = ...` ou itération d'une boucle de la forme `for y in t: ...`)

EXERCICE 4 *Écrire une fonction calculant la liste (éventuellement vide) des positions d'un objet dans un tableau.*

```
def positions(x, t):
    ....

>>> positions(42, [12, 17, 42, 5])
[2]
>>> positions(24, [12, 17, 42, 5])
[]
>>> positions(42, [42, 12, 17, 42, 5])
[0, 3]
```

EXERCICE 5 *En cas de doute sur l'orthographe de « ornithorynque », cherchez-vous ce mot sur le dictionnaire*[1] *de façon croissante de « abba » à « zz top » ?*

Le principe de la *recherche dichotomique* est le suivant : on veut tester l'appartenance d'un entier b à un tableau d'entiers trié dans l'ordre croissant.

- Si l'élément du milieu vaut b, on peut conclure. S'il est strictement supérieur à b, alors b ne peut pas appartenir à la moitié supérieure : on continue la recherche dans la première moitié (et la seconde dans le cas symétrique).
- Si l'élément du milieu de la moitié de tableau qui reste vaut b, on peut conclure ; sinon, on continue la recherche dans la seule moitié de moitié possible.
- ...
- Jusqu'au moment où on se retrouve avec une zone avec au plus un élément, et on peut alors conclure !

Bref, on casse en deux[2] à chaque étape. En regardant l'élément au milieu de la zone, on peut conclure quant à l'appartenance, ou bien diviser par deux la taille de la zone. Pour écrire l'algorithme, on commence par préciser comment représenter l'intervalle de recherche : ce sera $[d, f]$ (d pour début et f pour fin, avec les éléments d'indices d et f inclus, contrairement à la notation `t[d:f]` qui exclut `t[f]`).

On fait bien attention :

- à l'initialisation de ces valeurs ;
- au nombre d'éléments de cet intervalle (c'est $f - d + 1$ et non $f - d$) ;
- au cas de terminaison de la boucle : c'est lorsqu'il y a *au plus* un élément (on aurait aussi pu choisir « au plus deux éléments » ou encore « zéro élément », et ça aurait très bien marché. Il faut juste faire un choix clair et s'y tenir) ;
- à l'élément « du milieu » : si d et f ont même parité (il y a alors un nombre impair d'éléments dans la zone) c'est $\frac{d+f}{2}$; mais sinon, l'un des deux morceaux mis de côté aura un élément de plus que l'autre. En prenant $m = \dfrac{d+f-1}{2}$, les zones $[d, m-1]$ et $[m+1, f]$ auront respectivement k et $k+1$ habitants, avec $k = \dfrac{f-d-1}{2}$.

  Dans les deux cas, on prend donc $m = \left\lfloor \frac{d+f}{2} \right\rfloor$ et comme les indices d et f sont positifs, c'est aussi la valeur renvoyée en Python par `(d+f)//2`.

Entrées : b, T
$d, f \leftarrow 0, |T| - 1$
tant que $f > d$ **faire**
  $m \leftarrow \left\lfloor \frac{d+f}{2} \right\rfloor$
  si $T[m] = b$ **alors**
    Résultat : `True`
  si $T[m] > b$ **alors**
    $f \leftarrow m - 1$ # continuer à gauche
  sinon
    $d \leftarrow m + 1$ # continuer à droite
arrivé ici, $f = d$ (il reste un élément) ou $f = d - 1$ (il n'y a plus rien)
si $d = f$ *et* $T[d] = b$ **alors**
  Résultat : `True`
Résultat : `False`

1. Ces vieux machins en papier qu'utilisaient vos ancêtres avant google.
2. En grec : dikhotomia = « division en deux parties ».

EXERCICE 6 *Programmer effectivement cet algorithme. Effectuer quelques tests (attention le tableau doit être déjà trié).*

```
>>> recherche_dichotomique(42, [5, 12, 17, 42])
True
>>> recherche_dichotomique(24, [5, 12, 17, 42])
False
```

EXERCICE 7 *Montrer que le corps de la boucle « tant que » est exécuté au plus $\lceil \ln_2 n \rceil$ fois ($\lceil x \rceil$ désigne le plus petit entier supérieur à x : $\lceil \pi \rceil = 4$ et $\lceil 3 \rceil = 3$; on pourra montrer qu'à chaque itération, le nombre d'habitants de la zone de travail est « au moins divisé par deux »).* Ainsi, la recherche d'un élément dans une liste *triée* passe d'un coût linéaire « à un coût logarithmique ».

2 Recherche d'un sous-mot

La liste $[1,3,5]$ peut être vue comme une sous-liste de la liste $[12,-15,1,3,5,19,23]$ (ici, par sous-liste, on entend « en un seul morceau » : $[1,3,5]$ n'est pas vue comme une sous-liste de $[1,2,3,4,5]$). De même le mot `tag` est un sous-mot de `pouettagada`.

On cherche dans ce paragraphe à déterminer si une liste t_1 (respectivement un mot m_1) est une sous-liste (respectivement un sous-mot) d'une liste t_2 (respectivement un mot m_2). Les manipulations de listes et de chaînes de caractères étant très proches [3], les programmes doivent normalement fonctionner indifféremment sur les listes et les chaînes de caractères.

Si on reprend l'exemple `m1 = "tag"` et `m2 = "pouettagada"`, alors `m2[5:8] == m1`. Une façon assez simple de répondre au problème serait alors :

Entrées : m_1, m_2
$lg_1, lg_2 \leftarrow |m_1|, |m_2|$ # les longueurs
pour *d allant de* 0 *à* $lg_2 - lg_1$ **faire**
    si $m_2[d : d + lg_1] = m_1$ **alors**
        Résultat : `True`
Résultat : `False`

Mais on va s'interdire le *slicing* : on veut uniquement faire des comparaisons « lettre-à-lettre ». On a alors deux possibilités : on écrit à l'extérieur une fonction chargée de renvoyer le résultat de la comparaison `m2[d:d+lg1] == m1`, ou bien on écrit cette comparaison à l'intérieur du programme de recherche. On va privilégier le premier point de vue. En effet, la recherche va donner lieu à des sorties prématurées, naturelles via un `return` dans un programme. C'est faisable aussi dans une boucle via `break/continue` mais un peu plus délicat à manipuler (et hors programme).

Entrées : m_1, m_2, p
Retourne `True` si on trouve m_1 dans m_2 en position p et `False` sinon.
si $p + |m_1| > |m_2|$ **alors**
    Résultat : `False`
pour *i allant de* 0 *à* $|m_1| - 1$ **faire**
    si $m_2[p + i] != m_1[i]$ **alors**
        Résultat : `False`
Résultat : `True`

EXERCICE 8 *Écrire une fonction implémentant ce dernier algorithme (recherche d'un motif en une position donnée) ; la tester pour différents mots m_1 et m_2, puis vérifier si elle marche quand on lui donne des* listes *en entrée.*

EXERCICE 9 *Écrire une fonction prenant en entrée deux chaînes de caractères et retournant* `True` *ou* `False` *selon que la première est ou n'est pas un sous-mot de la seconde.*

Tester cette fonction judicieusement (ce qui doit inclure au moins 4 cas : absence, présence au bord gauche, au bord droit, et au milieu). Tester à nouveau sur des listes

EXERCICE 10 *Écrire une fonction prenant en entrée deux chaînes de caractères et retournant la liste (éventuellement vide) des positions dans m_2 où on trouve le mot m_1. Tester sur différents exemples.*

```
>>> positions_sous_mot("tag", "pouettagada")
[5]
>>> positions_sous_mot("plouf", "pouettagada")
[]
>>> positions_sous_mot("ta", "taratata")
[0, 4, 6]
```

EXERCICE 11 *Importer la fonction* `chaine_aleatoire` *du fichier* `cadeau.py` *créant une chaîne aléatoire sur l'alphabet $\{A, C, G, T\}$; comprendre son fonctionnement (en lisant le fichier source) et la tester.*

3. Indexation, longueur via `len`, slicing...

EXERCICE 12 *Créer une première chaîne aléatoire m_1 de longueur 5 (avec uniquement les lettres $\{A, C, G, T\}$) et une deuxième de même type mais de longueur 10^4. Déterminer le nombre de positions de m_2 auxquelles on trouve m_1. Ce nombre vous semble-t-il raisonnable ?*

Faire la moyenne sur une centaine d'exemples.

3 Crible d'Ératosthène

Une façon de tester le caractère premier ou non de $n \geqslant 4$ consiste à tester sa divisibilité par les entiers $k \in [\![2, n-1]\!]$.

- si l'un de ces k divise n, alors n est composé ;
- sinon, n est premier.

On notera le caractère *asymétrique* de la conclusion : la divisibilité par UN entier permet de conclure dans un sens, mais c'est la non-divisibilité par TOUS les k qui permet de conclure dans l'autre sens. Enfin, si n est composé, alors son plus petit diviseur est majoré par $\sqrt{n}$, ce qui limite en fait le nombre de tests de divisibilité à réaliser :

Entrées : n
si $n \leqslant 3$ **alors**
  Résultat : fastoche
pour k *allant de* 2 *à* $\lfloor\sqrt{n}\rfloor$ **faire**
  si k *divise* n **alors**
    Résultat : `False`
Résultat : `True`

Le test de divisibilité pourra se faire par exemple via `(n % k) == 0`

EXERCICE 13 *Programmer ce test de primalité.*

```
def est_premier(n):
    ....

>>> for i in range(15):
        if est_premier(i):
            print(i)
2
3
5
7
11
13
```

Vérifier qu'il y a exactement 168 *entiers majorés par* 1000 *qui sont premiers.*

Pour chronométrer le temps pris par l'ordinateur pour effectuer un calcul, on peut utiliser la fonction `time` de la bibliothèque `time` (oui je sais, ce n'est pas bien malin comme choix...). Un appel à `time.time()` fournit un « temps absolu » en secondes. Par différence, on obtient donc une durée entre deux « TOP » : dans l'exemple suivant, $t_1 - t_0$ est la durée du calcul :

```
t0 = time.time()
[mon calcul]
t1 = time.time()
```

EXERCICE 14 *Chronométrer le temps T_k (exprimé en secondes) effectué pour compter le nombre P_k d'entiers inférieurs à $n_k = 2^k$ qui sont premiers, et ceci pour $k \in [\![10, 16]\!]$. On résumera ceci dans un tableau de cette forme :*

k	10	11	12	13	14	15	16
$n_k = 2^k$	1024	2048	4096	8192	16384	32768	65536
P_k	172						6542
T_k							

Optionnellement, on pourra évaluer les rapports $\frac{T_{k+1}}{T_k}$: dans le cas d'une complexité linéaire (respectivement quadratique), ceux-ci devraient être de l'ordre de 2 *(respectivement* 4*).*

EXERCICE 15 *Donner (en fonction de N) un majorant raisonnable du temps de calcul nécessaire pour déterminer le caractère premier de chacun des entiers majorés par N.* On supposera que chaque division euclidienne se fait en temps constant.

Comparer aux résultats expérimentaux.

L'algorithme du *crible d'Ératosthène* permet de déterminer les entiers majorés par N qui sont premiers... en un temps meilleur qu'avec l'algorithme vu précédemment. Le principe en est le suivant :

- On initialise un tableau (disons T) de $N+1$ booléens à `True` (sauf les deux premiers, qui correspondent aux entiers non premiers 0 et 1). En k-ème case, le `True` s'interprète « ben jusqu'ici, et jusqu'à preuve du contraire, k semble être premier ».
- Tous les multiples (stricts) de 2 sont composés : on réaffecte donc à `False` tous les $T[2i]$ pour $2 \leqslant i \leqslant N/2$.
- Tous les multiples de 3 sont composés : on passe donc à `False` tous les $T[3i]$ pour $3 \leqslant i \leqslant N/3$.
- On constate que 4 est composé ($T[4]$ a été mis à `False` lors du premier passage) : inutile de « rayer » les multiples de 4, qui l'ont déjà été.
- Tous les multiples de 5 sont composés : on passe donc à `False` tous les $T[5i]$ pour $5 \leqslant i \leqslant N/5$.
- ...

On notera que pour $k=3$ par exemple, il est inutile de basculer $T[k*2]$ à `False` : ça a déjà été fait lors du traitement de $k=2$.

On balaye ainsi tout le tableau de nombreuses fois : environ $\sqrt{N}$ fois. Plus formellement, voici l'algorithme permettant de calculer le caractère premier ou non des entiers majorés par N. On utilise ici des tableaux à la Python, donc indexés depuis 0, donc de taille $N+1$:

```
Entrées : N
T ← [False,False] + [True] * (N − 1) # ben oui...
pour k allant de 2 à ⌊√N⌋ faire
    si T[k] alors
        # k est premier : on va « rayer » ses multiples
        pour i allant de k à ⌊N/k⌋ faire
            T[k * i] ←False
Résultat : T
```

EXERCICE 16 *Programmer effectivement l'algorithme du crible.*

```
def crible(n):
    ...
>>> crible(10)
[False, False, True, True, False, True, False, True, False, False, False]
```

EXERCICE 17 *Reprendre l'exercice 14, en utilisant cette fois le crible d'Ératosthène.*

On peut montrer relativement élémentairement que le temps de calcul est majoré par quelque chose de la forme $n \ln n$. En finassant, on obtient même $n \ln(\ln n)$: c'est presque linéaire.

EXERCICE 18 *Comparer l'espace mémoire nécessaire dans le cas de l'algorithme naïf et dans celui du crible.*

4 Pour ceux qui s'ennuient

EXERCICE 19 Project Euler ; problem 10 (summation of primes)
The sum of the primes below 10 *is* $2+3+5+7=17$.

Find the sum of all the primes below two million.

EXERCICE 20 Project Euler ; problem 21 (amical numbers)

Let $d(n)$ be defined as the sum of proper divisors of n (numbers less than n which divide evenly into n). If $d(a)=b$ and $d(b)=a$, where $a \neq b$, then a and b are an amicable pair and each of a and b are called amicable numbers.

For example, the proper divisors of 220 *are* $1,2,4,5,10,11,20,22,44,55$ *and* 110 *; therefore* $d(220)=284$. *The proper divisors of* 284 *are* $1,2,4,71$ *and* 142 *; so* $d(284)=220$.

Evaluate the sum of all the amicable numbers under 10000.

EXERCICE 21 Nombre moyen de diviseurs
Évaluer, pour $k \in [\![10,16]\!]$, le nombre moyen de diviseurs pour les entiers majorés par $n=2^k$.

Ce nombre de diviseurs varie beaucoup : pour $N=2^k$ il vaut $1+k=1+\ln_2 N$ mais pour N premier il vaut 2. On peut par contre montrer que le nombre moyen, pour les entiers majorés par n, est équivalent à $\ln n$; la (une) preuve étant elle-même basée sur un principe de crible !

EXERCICE 22 Project Euler ; problem 124 (ordered radicals)
The radical of n*,* $rad(n)$*, is the product of distinct prime factors of* n*. For example,* $504 = 2^3 \times 3^2 \times 7$*, so* $rad(504) = 2 \times 3 \times 7 = 42$.

If we calculate $rad(n)$ *for* $1 \leqslant n \leqslant 10$*, then sort them on* $rad(n)$*, and sorting on* n *if the radical values are equal, we get :*

Unsorted	
n	$rad(n)$
1	1
2	2
3	3
4	2
5	5
6	6
7	7
8	2
9	3
10	10

Sorted		
n	$rad(n)$	k
1	1	1
2	2	2
4	2	3
8	2	4
3	3	5
9	3	6
5	5	7
6	6	8
7	7	9
10	10	10

Let $E(k)$ *be the* k*th element in the sorted* n *column ; for example,* $E(4) = 8$ *and* $E(6) = 9$.
If $rad(n)$ *is sorted for* $1 \leqslant n \leqslant 100000$*, find* $E(10000)$.

EXERCICE 23 Project Euler ; problem 35 (circular primes)

The number 197*, is called a circular prime because all rotations of the digits :* 197, 971*, and* 719*, are themselves prime.*
There are thirteen such primes below 100 *:* 2, 3, 5, 7, 11, 13, 17, 31, 37, 71, 73, 79*, and* 97.
How many circular primes are there below one million ?

5 Besoin d'indications ?

- Exercice 2. Si on trouve l'élément, on peut renvoyer *immédiatement* le résultat `True`. La valeur `False` ne peut être renvoyée que *après* avoir fait *tous* les tests, et en aucun cas après le premier :

  Entrées : x, t
  pour i *allant de* 0 *à* $|t|-1$ **faire**
      si $t[i] = x$ **alors**
          Résultat : `True`
  Résultat : `False`

  On peut aussi boucler sur les éléments du tableau plutôt que les positions (`for y in t:...`) ce qui serait plus dans l'esprit Python.

- Exercice 4. On tient à jour une liste (initialement vide) de positions qu'on peut « augmenter » via la méthode `append`

- Exercice 20. On parcourt les entiers de 2 à 1000 en testant s'ils sont amicaux et on crible en mettant à jour un tableau de booléens.

- Exercice 21. On peut cribler, en mettant à jour un tableau non pas de booléens, mais de nombre de diviseurs. On crible selon tous les entiers $k \leqslant n$. Pour chacun de ces k, on informe chaque multiple de k qu'il possède k comme facteur... en incrémentant de 1 tous les $T[i \times k]$ pour $i \leqslant \frac{n}{k}$.

- Exercice 22. C'est presque le même principe qu'à l'exercice 21 question précédente : cette fois, le tableau que l'on tient à jour contient le produit des diviseurs *premiers* rencontrés jusqu'ici...

- Exercice 23 On peut mémoizer les entiers premiers dans un tableau ou un dictionnaire.