Corrigé du TP 06 Recherches et Crible

Chenevois-Jouhet-Junier

```
import math, random  #imports des bibliothèques/modules nécessaires
```

1 Recherche d'un élément dans un tableau

1.1 Exercice 1

```
def mystere(tab, seuil):
    '''Retourne True si tous les éléments de tab sont < seuil'''
    k = 0
    while k < len(tab) and tab[k] < seuil:
        k += 1
    return k == len(tab)
```

Soit un tableau t de 50 entiers tirés aléatoirement dans `[1 ; 100] ]`, `mystere(t, 98)` retourne `True` si tous les éléments du tableau sont inférieurs à 98.

La probabilité de cet événement est donc de $(98/100)^{50} \approx 0,364\ldots$

```
def mystere2(tab, seuil):
    k = 0
    while tab[k] < seuil and k < len(tab):
        k += 1
    return k == len(tab)
```

Si on permute les opérandes du `and` on peut obtenir une erreur à l'éxécution. En effet si tous les éléments de `tab` sont inférieurs à 998 (probabilité de $(998/1000)^{50} \approx 0,904$) alors la condition sera évaluée pour `k == len(tab)`. Le premier opérande du `and`, `tab[len(tab)]` sera évalué ce qui provoquera une exception du type `IndexError`. Si on évalue d'abord `k < len(tab)`, cette comparaison retourne `False` pour `k == len(tab)` et le `and` étant **paresseux**, le second opérande n'est pas évalué et aucune exception n'est levée.

```
def test(tab, seuil):
    '''Retourne True si au moins un élément de tab est > seuil'''
    k = 0
    while k < len(tab) and tab[k] <= seuil:
        k += 1
    return k < len(tab)
"""
In [21]: from random import randint
In [22]: tabalea = [randint(1, 1000) for _ in range(50)]
In [23]: mystere2(tabalea, 998)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-23-3a5599684257> in <module>()
----> 1 mystere2(tabalea, 998)

/home/correcTP6-recherches-cribles.py in mystere2(tab, seuil)
     20 def mystere2(tab, seuil):
     21     k = 0
```

```
---> 22     while tab[k] < seuil and k < len(tab):
     23         k += 1
     24     return k == len(tab)

IndexError: list index out of range
"""
"""
In [30]: test([1, 2 ,5], 5), test([1, 2 ,5], 4), test([1, 2 ,5], 6)
Out[30]: (False, True, False)
"""
```

1.2 Exo 2 Recherche séquentielle

```
def appartient_while(x,t):
    '''redéfinition de l'opérateur in'''
    k = 0
    while k < len(t) and t[k] != x:
        k += 1
    return k < len(t)

def appartient_for(x,t):
    '''redéfinition de l'opérateur in'''
    for terme in t:
        if x==terme:
            return True
    #si on arrive la x n'appartient au t
    return False

"""
In [32]: appartient_for(42,[12,17,42]), appartient_while(42,[12,17,42])
Out[32]: (True, True)

In [33]: appartient_for(24,[12,17,42]), appartient_while(24,[12,17,42])
Out[33]: (False, False)
"""
```

1.3 Exo 3 Complexité de la Recherche séquentielle

Dans le pire des cas (x apparait une seule fois en dernière position) #le nombre d'accès au tableau effectués pour tester l'appartenance de x à t est égal à la taille de t (**complexité linéaire**)

1.4 Exo 4

```
def positions(x,t):
    '''retourne la liste des positions de x dans t'''
    L = []
    for indice in range(len(t)):
        if x == t[indice]:
            L.append(indice)
    return L

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

```
[0, 3]
"""
```

1.5 Exo 6 Recherche dichotomique dans un tableau trié (ordre croissant par défaut)

```
def recherche_dichotomique(x,t):
    #indices de debut, fin
    d,f = 0,len(t)-1
    while f-d > 0:
        #indice du milieu
        m = (f+d)//2
        if t[m] == x:
            return True
        elif t[m] > x:
            f = m-1
        else:
            d = m+1
    #arrivé là soit d==f et on peut avoir x==t[d] (ou pas)
    #soit f<d et dans ce cas x n'est pas dans le tableau
    if f == d and t[d] == x:
        return True
    return False
```

Attention la recherche dichotomique ne peut s'appliquer qu'à des tableaux triés !!!

```
"""
>>> recherche_dichotomique(5,[5,14,27,42])
True
>>> recherche_dichotomique(42,[5,14,27,42])
True
>>> recherche_dichotomique(14,[5,14,27,42,69])
True
>>> recherche_dichotomique(18,[5,14,27,42,69])
"""
```

Partie A Terminaison et complexité de l'algorithme de recherche dichotomique :

- Après p itérations effectués, notons d(p) et f(p) les indices du début et de fin de la zone de recherche et n(p) = f(p) -d(p) + 1 le nombre d'éléments de cette zone.
- Si l'itération p+1 est effectuée alors :
  - **1er cas :** d(p) et f(p) sont de meme parité

    Si on note m(p) = (d(p) + f(p))/2 alors on aura soit d(p+1) = m(p) + 1 = (d(p)+f(p)+2)/2 et f(p+1) = f(p) soit d(p+1) = d(p) et f(p+1) = m(p) - 1 = (d(p)+f(p)-2)/2

    Pour ces deux sous-cas, la nouvelle zone de recherche comptera
    n(p+1) = f(p+1) - d(p+1) + 1 n(p+1) = (f(p) - d(p))/2 et donc n(p+1) < n(p)/2
  - **2eme cas :** d(p) et f(p) n'ont pas meme parité

    Si on note m(p) = (d(p) + f(p))//2 = (d(p) + f(p) - 1)/2 alors on aura :
    - soit d(p+1) = m(p) + 1 = (d(p)+f(p)+1)/2 et f(p+1) = f(p) et la nouvelle zone de recherche comptera n(p+1) = f(p+1) - d(p+1) + 1 = (f(p) - d(p) + 1)/2 = n(p)/2 éléments
    - soit d(p+1) = d(p) et f(p+1) = m(p) - 1 = (d(p)+f(p) - 3)/2 et la nouvelle zone de recherche comptera n(p+1) = f(p+1) - d(p+1) + 1 = (f(p) - d(p) -1)/2 = n(p)/2 - 1 éléments

    Dans ce 2eme cas, la nouvelle zone de recherche comptera toujours : n(p+1) <= n(p)/2 éléments après la p+1 eme itération.

Dans tous les cas, après p+1 itérations on aura n(p+1) <= n(p)/2, le nombre d'éléments de la zone de recherche est au moins divisé par 2.

Par récurrence on montre alors que n(p) <= $n(0)/2^p$ après p itérations. Or n(0) = f - d +1 est le nombre d'éléments dans la zone initiale de recherche.

La p-ième itération est effectuée ssi après la (p-1)-ème la zone de recherche comprend au moins 2 éléments c'est-à-dire :

n(p-1) >= 2 donc $n(0)/2^{p-1}$ >= 2 donc ln(n(0)/2)/ln(2) >= p - 1

donc p <= 1 + ln(n(0)/2)/ln(2) donc p <= ln((f - d + 1)/2)/ln(2) + 1

Donc la boucle est exécutée au plus un nombre de fois égal au + petit entier supérieur à ln((f - d + 1)/2)/ln(2)

Ainsi l'algorithme de recherche dichotomique se termine et pour une zone de recherche initiale [f ; d] sa complexité est de l'ordre de ln(f-d+1).

Par rapport à la recherche séquentielle, on passe d'une **complexité linéaire** à une **complexité logaritmique** mais **il faut que le tableau soit déjà trié**.

Partie B Preuve de correction de l'algorithme de recherche dichotomique

Prouvons que l'algorithme retourne False si x pas dans t et True sinon.

- **Premier cas :** x pas dans t

  La boucle se termine (déjà prouvé), la condition f == d and t[d] == x est évaluée à False et donc la fonction retourne bien False

- **Deuxième cas :** x dans t

  t étant trié, on a avant la boucle t[d] <= x <= t[f]. Notons d(p) et f(p) les indices de début et de fin de la zone de recherche avant l'itération p.

  Démontrons que la propriété t[d(p)] <= x <= t[f(p)] est un invariant de boucle

  - **Initialisation :** on a t[d(1)] <= x <= t[f(1)] comme vu ci-dessus
  - **Induction :**

    On suppose que t[d(p)] <= x <= t[f(p)] et que l'itération p se réalise On calcule d'abord m(p) = (d(p) + f(p))//2

    Si t[m(p)] == x alors d(p+1) = f(p+1) = x et donc t[d(p+1)] <= x <= t[f(p+1)] est vérifiée

    Sinon si t[m(p)] < x alors d(p+1) = m(p) + 1 et f(p+1) = f(p) et puisque t[d(p)] <= x <= t[f(p)] et que t est trié dans l'ordre croissant, on a t[d(p+1)] <= x <= t[f(p+1)]

    Sinon si t[m(p)] > x alors d(p+1) = d(p) et f(p+1) = m(p) - 1 et puisque t[d(p)] <= x <= t[f(p)] et que t est trié dans l'ordre croissant, on a t[d(p+1)] <= x <= t[f(p+1)]

  - **Conclusion:**

    Lorsque la boucle se termine, avant la dernière itération p non réalisée on a : t[d(p)] <= x <= t[f(p)] De plus comme l'itération p ne se réalise pas on a f(p) <= d(p)

    * Premier sous-cas :f(p) < d(p)

    Celà signifie qu'à la fin de la dernière itération on a eu f(p) = (d(p-1)+f(p-1))//2 - 1 et d(p) = d(p-1) et ce n'est possible que si (d(p-1)+f(p-1))//2 = d(p-1) et t[d(p-1)] > x c'est-à-dire x pas dans t. On a déjà prouvé que dans ce cas, la fonction retourne False

    * Deuxième sous cas: f(p) == d(p)

    Comme t[d(p)] <= x <= t[f(p)] celà signifie que t[d(p)] == x == t[f(p)] la condition f == d and t[d] == x est évaluée à True et donc la fonction retourne bien True

2 Recherche d'un sous-mot

2.1 Exo 8

```
def recherche_motif_posfixe(m1,m2,pos):
    '''Retourne True si m1 est sous-motif de m2 à  partir de pos'''
    L1, L2 = len(m1), len(m2)
    #s'il n'y a pas la place pour m1 dans m2 à partir de pos
```

```
    if pos+L1 > L2:
        return False
    #on compare terme à terme les éléments de m1 et de m2 à partir de pos
    for i in range(L1):
        if m2[pos+i] != m1[i]:
            return False
    return True
"""
>>> recherche_motif_posfixe('abc','eabcd',1)
True
>>> recherche_motif_posfixe('abc','eabcd',0)
False
>>> recherche_motif_posfixe([1,2,3],[0,1,2,3,4],0)
False
>>> recherche_motif_posfixe([1,2,3],[0,1,2,3,4],1)
True
"""
```

2.2 Exo 9

```
def est_sous_mot(m1, m2):
    '''Retourne True si m1 est un sous-mot de m2'''
    L1, L2 = len(m1), len(m2)
    assert L1 <= L2
    for i in range(L2-L1+1):
        if recherche_motif_posfixe(m1, m2, i):
            return True
    return False

"""
In [2]: [est_sous_mot('abc', mot) for mot in ['abcd', 'dabc', 'dabce', 'cdab']]
Out[2]: [True, True, True, False]
In[5]:[est_sous_mot([1,2], mot) for mot in [[1,2,3], [5,1,2,3], [5,1,2],[5,1]]]
Out[5]: [True, True, True, False]
"""
```

2.3 Exo 10

```
def positions_sous_mot(m1,m2):
    '''Retourne la liste des positions dans m2 où m1 sous-mot de m1'''
    L1,L2 = len(m1),len(m2)
    Lpos = []
    assert L1 <= L2
    for i in range(L2-L1+1):
        if recherche_motif_posfixe(m1,m2,i):
            Lpos.append(i)
    return Lpos

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

2.4 Exo 11

```
def chaine_aleatoire(n):
    '''Return une chaine aléatoire de taille n sur l'alphabet {A,C,G,T}
    pour l'importer : from cadeau.py import chaine_aleatoire'''
    from random import randint
    alphabet = ['A','C','G','T']
    chaine = ''
    for i in range(n):
        chaine += alphabet[randint(0,3)]
    return chaine

"""
>>> chaine_aleatoire(12)
'TCAGGTATCATC'
"""
```

2.5 Exo 12

```
def stats_positions(n):
    '''Moyenne sur n exemples du nombre de positions où une chaine aleatoire
    de longueur 5 de l'alphabet {A,C,G,T} apparait dans une chaine aléatoire
    de longueur 10**4'''
    somme = 0
    for exemple in range(n):
        chaine1 = chaine_aleatoire(5)
        chaine2 = chaine_aleatoire(10**4)
        somme += len(positions_sous_mot(chaine1,chaine2))
    return 'Moyenne sur %s exemples : %.3f'%(n,float(somme)/n)
"""
>>> stats_positions(100)
'Moyenne sur 100 exemples : 9.650'
"""
```

Résultat prévisible : la probabilité d'une chaine aléatoire de longueur 5 prise dans un alphabet de 4 lettres est $\frac{1}{4^5}$.

Une chaine de longueurs 10000 correspond à la répétition indépendante de 9996 chaines de longueur 5.

La variable aléatoire donnant le nombre de positions d'une chaine fixée sur une chaine de longueur 10000 a donc pour espérance :

```
"""
>>> 1./4**5*9996
9.76171875
"""
```

3 Crible d'Ératosthène

3.1 Exo 13

```
def est_premier(n):
    '''test de primalité'''
    if n <= 1:
       return False
    for d in range(2,int(math.sqrt(n))+1):
        if n%d==0:
            return False
    #si on arrive là, n est premier
    return True

def decompte_premiers(n):
```

```
    '''retourne le nombre d'entiers premiers inférieurs ou égaux à n'''
    compteur = 0
    for i in range(2,n+1):
        if est_premier(i):
            compteur += 1
    return compteur
"""
>>> decompte_premiers(1000)
168
"""
```

3.2 Exo 14

```
def test_exo14():
    from time import time
    for k in range(10, 17):
        t0 = time()
        nk = 2**k
        pk = decompte_premiers(nk)
        t1 = time()
        t = t1-t0
        if k>10:
            print('k={:2} | nk={:5} |  pk={:4} | t(k)={:5.3f}| t(k)/t(k-1)={:5.3f}'.format(k,nk,pk,t,t/tpreced)
        else:
            print('k={:2} | nk={:5} |  pk={:4} |'
            ' t(k)={:5.3f}'.format(k,nk,pk,t))
        tpreced = t

"""
In [10]: test_exo14()
k=10 | nk= 1024 |  pk= 172 | t(k)=0.005
k=11 | nk= 2048 |  pk= 309 | t(k)=0.005| t(k)/t(k-1)=1.031
k=12 | nk= 4096 |  pk= 564 | t(k)=0.011| t(k)/t(k-1)=2.124
k=13 | nk= 8192 |  pk=1028 | t(k)=0.024| t(k)/t(k-1)=2.164
k=14 | nk=16384 |  pk=1900 | t(k)=0.041| t(k)/t(k-1)=1.703
k=15 | nk=32768 |  pk=3512 | t(k)=0.094| t(k)/t(k-1)=2.285
k=16 | nk=65536 |  pk=6542 | t(k)=0.230| t(k)/t(k-1)=2.443
"""
```

On peut conjecturer que le décompte des entiers premiers inférieurs à 2^k est un algorithme de complexité comprise entre **linéaire** (temps multiplié par 2 lorsque la taille de l'entrée multipliée par 2) et **quadratique** (temps multiplié par 2^2 lorsque la taille est multipliée par 2).

3.3 Exo 15

Pour chaque entier k majoré par N la complexité du test de primalité est majorée par $\sqrt{k}$.

Si on teste successivement tous les entiers entre 2 et N la complexité est majorée par $\sqrt{2}+\sqrt{3}+\cdots+\sqrt{N}$.

La fonction racine est croissante et continue sur $[0;+\infty[$ donc on peut majorer la somme précédente par $\int_2^{N+1}\sqrt{x}\,\mathrm{d}x$ c'est à dire par $\frac{2}{3}(N+1)\sqrt{N+1}-2\sqrt{2}$.

Un majorant de la complexité est donc de l'ordre $N\sqrt{N}$.

Un minorant est naturellement N.

3.4 Exo 16

```
def crible(n):
    '''retourne un tableau de taille n+1 rempli de True ou de False
    selon que l'indice i est premier ou non'''
    t = [False,False] + [True]*(n-1)
    #d est le plus petit diviseur
    for d in range(2, int(math.sqrt(n))+1):
        #si d n'est pas composé
        if t[d]:
            #on met à False tous les multiples de d pas encore marqués
            for i in range(d, n//d+1):
                t[d*i] = False
    return t

"""
>>> crible(10)
[False, False, True, True, False, True, False, True, False, False, False]
>>> crible(1000).count(True)
168
"""
```

3.5 Exo 17

```
def test_exo17():
    from time import time
    for k in range(10, 17):
        t0 = time()
        nk = 2**k
        pk = crible(nk).count(True)
        t1 = time()
        t = t1-t0
        if k>10:
            print('k={:2} | nk={:5} |  pk={:4} | t(k)={:5.3f}| t(k)/t(k-1)={:5.3f}'.format(k,nk,pk,t,t/tpreced)
        else:
            print('k={:2} | nk={:5} |  pk={:4} |'
            ' t(k)={:5.3f}'.format(k,nk,pk,t))
        tpreced = t
```

On vérifie expérimentalemnt que la complexité en temps de l'algorithme du crible est quasiment linéaire (rapport très proche de 2)

```
"""
In [29]: test_exo17()
k=10 | nk= 1024 |  pk= 172 | t(k)=0.001
k=11 | nk= 2048 |  pk= 309 | t(k)=0.001| t(k)/t(k-1)=2.085
k=12 | nk= 4096 |  pk= 564 | t(k)=0.003| t(k)/t(k-1)=2.179
k=13 | nk= 8192 |  pk=1028 | t(k)=0.003| t(k)/t(k-1)=0.985
k=14 | nk=16384 |  pk=1900 | t(k)=0.003| t(k)/t(k-1)=1.324
k=15 | nk=32768 |  pk=3512 | t(k)=0.007| t(k)/t(k-1)=2.074
k=16 | nk=65536 |  pk=6542 | t(k)=0.015| t(k)/t(k-1)=2.108
"""
```

3.6 Exo 18

Pour l'algorithme du crible l'espace mémoire est proportionnel à n puisqu'on utilise une liste de taille $n+1$.

Dans l'algorithme naif l'espace mémoire est constant puisqu'on utilise toujours le même nombre de variables (et pas de listes).

La complexité en temps est donc meilleure pour l'algorithme du crible mais sa complexité en espace est moins bonne.

3.7 Exo 19 Project Euler problem 10

```
def sum_of_prime(n):
    '''Project Euler problème 10 Summation of primes'''
    t = crible(n)
    s = 0
    for k in range(len(t)):
        if t[k]:
            s += k
    return s

"""
In [5]: sum_of_prime(2*10**6)
Out[5]: 142913828922
"""
```

3.8 Exo 20 Project Euler problem 21

```
def diviseurs(n):
    '''Retourne le tableau des diviseurs propres de n'''
    tab = [1]
    #boucle sur le plus petit diviseur
    for d in range(2, int(sqrt(n)) + 1):
        if n%d == 0:
            tab.append(d)
            quotient = n//d
            if d != quotient:
                tab.append(quotient)
    return tab

def somme(tab):
    '''Retourne la somme des éléments d'un tableau d'entiers'''
    s = 0
    for element in tab:
        s += element
    return s

def sum_of_amical(n):
    '''Retourne la somme des entiers amicaux entre 1 et n'''
    #tableaux pour cribler les entiers amicaux
    amicaux = [False]*n
    #somme
    s = 0
    for n in range(2, n + 1):
        #si n est déjà memoizé on passe son tour
        if amicaux[n - 1]:
            continue
        sommediv = somme(diviseurs(n))
        if sommediv != n and somme(diviseurs(sommediv)) == n:
            amicaux[n - 1] = amicaux[sommediv - 1] = True
            s += n + sommediv
    return 'Somme des entiers amicaux <= 10000 : %d'%s

"""
In [8]: sum_of_amical(10000)
Out[8]: 'Somme des entiers amicaux <= 10000 : 31626'
"""
```

3.9 Exo 21

```
def nbmoyen_diviseurs(n):
    '''Retourne le nombre moyen de diviseurs pour les entiers <=n'''
    crible = [0,1] + [2] * (n-1)
    #k plus petit diviseur
    for k in range(2, int(math.sqrt(n)) + 1):
        #on crible tous les entiers divisible par k avec k + petit diviseur
        crible[k*k] += 1
        for j in range(k+1, n//k + 1):
            crible[j*k] += 2
    return crible, sum(crible)/n

def test_exo21():
    from time import time
    for k in range(10, 17):
        t0 = time()
        nk = 2**k
        #nombre moyen de diviseurs
        mk = nbmoyen_diviseurs(nk)[1]
        t1 = time()
        t = t1-t0
        if k>10:
            print('k={:2} | nk={:5} |  mk={:6.3f} | log(nk) = {:6.3f} |'
            ' t(k)={:5.3f}| t(k)/t(k-1)={:5.3f}'.format(k, nk, mk,
             math.log(nk),t,t/tpreced))
        else:
            print('k={:2} | nk={:5} |  mk={:6.3f} | log(nk) = {:6.3f} |'
            ' t(k)={:5.3f}| '.format(k,nk,mk, math.log(nk),t))
        tpreced = t

"""
In [86]: test_exo21()
k=10 | nk= 1024 |  mk= 7.092 | log(nk) =  6.931 | t(k)=0.000|
k=11 | nk= 2048 |  mk= 7.782 | log(nk) =  7.625 | t(k)=0.001| t(k)/t(k-1)=2.257
k=12 | nk= 4096 |  mk= 8.477 | log(nk) =  8.318 | t(k)=0.002| t(k)/t(k-1)=2.445
k=13 | nk= 8192 |  mk= 9.168 | log(nk) =  9.011 | t(k)=0.004| t(k)/t(k-1)=2.020
k=14 | nk=16384 |  mk= 9.860 | log(nk) =  9.704 | t(k)=0.012| t(k)/t(k-1)=3.219
k=15 | nk=32768 |  mk=10.553 | log(nk) = 10.397 | t(k)=0.018| t(k)/t(k-1)=1.585
k=16 | nk=65536 |  mk=11.245 | log(nk) = 11.090 | t(k)=0.040| t(k)/t(k-1)=2.186
"""
```

Effectivement le nombre moyen de diviseurs pour les entiers majorés par n semble équivalent à $\log(n)$.

3.10 Exo 22 Project Euler problem 124 Ordered radicals

```
def rad(n):
    radicaux = [[i, 1] for i in range(1, n+1)]
    #boucle sur tous les diviseurs premiers possibles <= n
    for k in range(2, n + 1):
        #si k est premier (un seul diviseur premier lui-meme)
        if radicaux[k-1][1] == 1:
            #boucle sur tous les multiples possibles de k
            #à l'inverse du crible d'Eratosthene on commence à 1
            #car le radical d'un entier premier est lui-meme
            for j in range(1, n//k + 1):
                radicaux[j*k - 1][1] *= k
    radicaux.sort(key=lambda t : t[1])
    return radicaux
```

```
"""
In [10]: t = rad(100000)
In [11]: t[9999]
Out[11]: [21417, 1947]
"""
```

3.11 Exo 23

```
def rotation(n, exposantmax):
    '''rotation d'un entier n,  10**(exposantmax) <= n < 10**(exposantmax+1)'''
    return (n%10)*10**exposantmax + n//10

def est_premiercirculaire(n, uncrible):
    ecriture_decimale = str(n)
    exposantmax = len(ecriture_decimale) - 1
    for k in range(exposantmax):
        n = rotation(n, exposantmax)
        if not uncrible[n]:
            return False
    return True

def circular_primes(n):
    '''Retourne le nombre de premiers circulaires <= n'''
    #crible d'Eratosthene des entiers premiers <= n
    moncrible = crible(n)
    compteur = 0
    for k in range(n + 1):
        if  moncrible[k]:
            if est_premiercirculaire(k, moncrible):
                compteur += 1
    return compteur

"""
In [14]: circular_primes(10**6)
Out[14]: 55
"""
```