Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique commune
G. Dewaele, A. Troesch

2015/2016

TP n° 7 : Algorithmes élémentaires classiques

Nous étudions dans ce TP plusieurs algorithmes indépendants, qui se retouvent, sous cette forme ou de façon adaptée, à la base de nombreux algorithmes plus complexes.

Correction de l'exercice 1 –

1. On utilise le fait que l'instruction `return` tue la fonction (notamment, la boucle est interrompue). Au lieu d'itérer sur l'indice des éléments de la liste (itération sur un compteur numérique), par commodité, on itère directement sur les éléments de la liste (facilité accordée par la syntaxe Python).

```
def IsInList(L,a):
    for x in L:
        if x == a:
            return True
    return False
```

2. Pour obtenir le premier indice, on ajoute un compteur, ou alors, on itère cette fois sur l'indice, au lieu de l'élément de la liste. On propose deux versions, la première renvoyant -1 si l'élément n'est pas dans la liste, la seconde retournant un message d'erreur (ce qui interrompt le programme).

```
def FirstIndex(L,a):
    n = len(L)
    for i in range(n):
        if  L[i] == a:
            return i
    return -1

def FirstIndexIfExists(L,a):
    n = len(L)
    for i in range(n):
        if  L[i] == a:
            return i
    raise ValueError("Cette␣valeur␣n'est␣pas␣élément␣de␣la␣liste")
```

3. Pour le nombre d'occurrences, on introduit un compteur d'occurrences, et cette fois, on parcourt la totalité de la liste. On peut itérer directement sur les éléments de la liste.

```
def Occurrences(L,a):
    n = 0
    for x in L:
        if x == a:
            n += 1
    return n
```

4. Pour la moyenne (moment d'ordre 1), on crée une nouvelle liste à chaque itération, évidemment ! On utilise `FirstIndex`, qui revoie (à un décalage près de 1), le nombre de passages dans la boucle de la première fonction, sauf si le résultat est -1. Dans ce cas, le nombre de passages est n, la longueur de la liste. On note N la borne supérieure de l'intervalle d'entiers aléatoires, n la longueur de la liste, nb le nombre d'itérations pour le calcul de la moyenne.

```
def Moyenne(N,n,nb):
    M1 = 0
    for i in range(nb):
        L = [ random.randint(0,N-1) for k in range(n)]
        a = random.randint(0,N-1)
        m = FirstIndex(L,a)
        if m == -1:
            M1+= n
        else:
            M1+= m+1
    return M1 / nb
```

On lance la simulation pour différentes valeurs de N, avec $n = 10000$ et $nb = 1000$:

```
for N in [10,50,100,200,1000,5000,10000]:
    print(Moyenne(N,10000,1000))
```

On obtient les résultats suivants :

```
10.153
49.284
98.214
200.345
1023.801
4394.744
6417.509
```

Ces résultats corroborent le fait que l'espérance est de l'ordre de $\frac{1}{N}$ lorsque N est petit devant n, mais que ce n'est plus le cas lorsque cette hypothèse n'est plus satisfaite.

En effet, lorsque n est grand devant N, la liste contient presque systématiquement la valeur recherchée, et l'expérience n'est pas très différente du temps d'attente d'un premier succès dans une suite infinie d'expériences de Bernoulli indépendantes. Ainsi, nous sommes presque en présence d'une loi géométrique de paramètre $p = \frac{1}{N}$ (probabilité d'un succès lors d'une expérience de Bernoulli, correspondant ici à l'obtention d'un a à une position donnée), dont l'espérance vaut $\frac{1}{p}$

Ce résultat (espérance d'une loi géométrique) sera un résultat du cours. Il se démontre par la formule du binôme négatif (dérivées des sommes géométriques, qu'on a déjà rencontrées en exercice) :

$$\sum_{n=k}^{+\infty} n(n-1)\cdots(n-k+1)x^{n-k} = \frac{(k-1)!}{(1-x)^k}.$$

Si X est une variable aléatoire suivant une loi géométrique de paramètre p, on a alors :

$$E(X) = \sum_{n=1}^{+\infty} nP(X=n) = \sum_{n=1}^{+\infty} nq^{n-1}p = \frac{p}{(1-q)^2} = \frac{1}{p}.$$

Ici, p désigne la probabilité d'un succès, $q = 1-p$ la probabilité d'un échec. L'événement $[X = n]$ est réalisé si et seulement on a obtenu $n-1$ échecs suivi d'un succès, les expériences étant indépendantes, ce qui justifie l'expression de la probabilité de cet événement.

Lorsque n n'est plus assez grand face à n, on arrête plus souvent l'expérience sans avoir trouvé la valeur dans le tableau : si on avait considéré un tableau infini, on aurait dû aller plus loin pour trouver le premier succès ; ainsi les valeurs obtenues sont en moyenne inférieures aux valeurs obtenues pour une variable suivant une loi géométrique de même paramètre. cela explique que la moyenne obtenue soit inférieure à $\frac{1}{N}$, lorsque N devient trop grand.

5. On calcule la variance à l'aide de la formule de König-Huygens :

$$V(X) = E((X - E(X))^2) = E(X^2) - 2E(X)^2 + E(X)^2 = E(X^2) - E(X)^2.$$

On calcule donc la moyenne des valeurs obtenues (moment d'ordre 1) et la moyenne des carrés (moment d'ordre 2, pour une estimation de $E(X^2)$). Ces moyennes doivent être calculées sur la même série d'expériences.

```
def EcartType(N,n,nb):
    M1 = 0
    M2 = 0
    for i in range(nb):
        L = [ random.randint(0,N-1) for k in range(n)]
        a = random.randint(0,N-1)
        m = FirstIndex(L,a)
        if m == -1:
            M1+= n
            M2+= n * n
        else:
            M1+= m+1
            M2+= (m+1)*(m+1)
    M1 /= nb
    M2 /= nb
    return math.sqrt(M2 - M1 * M1)
```

On ne fait l'expérience que pour 4 valeurs de N, en poussant le nombre d'expériences à 3000, pour avoir un résultat plus précis :

```
for N in [10,50,100,200]:
    print(EcartType(N,10000,3000))
```

Les résultats obtenus :

```
9.594528388618171
48.4143414828935
101.20993691607339
203.4032349562481
```

On observe que le carré de l'écart-type est à peu près égal à $\frac{q}{p^2}$, ce qui correspond à la variance d'une loi géométrique, qu'on calculera plus tard dans le cours de mathématiques. Vvous pouvez faire ce calcul dès à présent avec la formule de König-Huygens, la formule du binôme négatif, et l'expression du moment d'ordre 2 :

$$E(X^2) = \sum_{n=1}^{+\infty} n^2 P(X = n).$$

Correction de l'exercice 2 –

1. On considère des indices $i < j$, initialisés comme les indices extrêmes du tableau, et tels que si a est dans le tableau, son indice est *strictement* entre i et j. Cela nécessite de se débarrasser au début des cas où ce n'est pas vérifié initialement. Ensuite, on coupe la tranche en son milieu. Si au milieu, on tombe sur la valeur de a, on l'a trouvée et on arrête le programme, sinon, on garde la moitié dans laquelle on a une chance de trouver a (moitié qu'on est en mesure de déterminer, le tableau étant supposé trié). Si la tranche n'est plus constituée que d'un ou deux termes, elle ne peut pas contenir a (puisque a est différent des bornes de la tranche), et on arrête le programme.

   Cela peut se programmer récursivement :

```
def dicho(T,a,i,j):
    """ Recherche de a dans un tableau T trié par ordre croissant
    entre i et j"""
    if j-i < 2:
        return -1
    k = (i+j)//2
    if T[k] == a:
        return k
    if T[k] < a:
        i = k
    else:
        j = k
    return dicho(T,a,i,j)

def RechercheDichotomiqueRec(T,a):
    if T[0] > a:
        return -1
    if T[-1] < a:
        return -1
    if T[0] == a:
        return 0
    if T[-1] == a:
        return len(T)
    return dicho(T,a,0,len(T))
```

   La version non récursive est très similaire :

```
def RechercheDichotomique(T,a):
    if T[0] > a:
        return -1, 0
    if T[-1] < a:
        return -1, 0
    if T[0] == a:
        return 0, 0
    if T[-1] == a:
        return len(T), 0
    i = 0
    j = len(T)
    c = 0
    while j-i >= 2:
```

```
        k = (i+j)//2
        c += 1
        if T[k] == a:
            return k, c
        if T[k] < a:
            i = k
        else:
            j = k
    return -1, c
```

Dans cette version, on a déjà inclus le compteur c de passages dans la boucle, en vue de répondre à la question suivante. Notre batterie de tests :

```
T= [1,3,5,7,9,11,13,15,17,19,21,23,25]

print(RechercheDichotomiqueRec(T,17))
print(RechercheDichotomiqueRec(T,18))
print(RechercheDichotomiqueRec(T,28))
print(RechercheDichotomiqueRec(T,-3))
print(RechercheDichotomique(T,17))
print(RechercheDichotomique(T,18))
print(RechercheDichotomique(T,28))
print(RechercheDichotomique(T,-3))
```

... et les résulats, qui sont concluants :

```
8
-1
-1
-1
(8, 4)
(-1, 4)
(-1, 0)
(-1, 0)
```

2. On calcule la moyenne du nombre de passages dans la boucle sur 1000 expériences, et on fait le test pour différentes valeurs de n. Pour ces valeurs on donne en parallèle la valeur de $\log_2(n)$, pour comparaison.

```
def moyenne(n):
    M = 0
    for i in range(1000):
        T = [random.randint(0,2*n) for j in range(n)]
        a = random.randint(0,2*n)
        T.sort()
        b,c = RechercheDichotomique(T,a)
        M += c
    return M / 1000

for n in [10,100,1000,5000,10000,50000]:
    print(moyenne(n), math.log2(n))
```

Les résultats obtenus :

```
2.224 3.321928094887362
6.052 6.643856189774724
9.439 9.965784284662087
11.901 12.287712379549449
12.969 13.287712379549449
15.171 15.609640474436812
```

On se rend compte que l'on suit une évolution similaire au logarithme en base 2, en restant un peu en dessous, ce qui est normal : si on n'interrompait pas le programme dès qu'on trouve la valeur recherchée, on effectuerait à peu près $\log_2(n)$ étapes. Le fait d'interrompre parfois un peu plus tôt diminue en moyenne le nombre d'étapes.

3. Pour trouver la première occurrence, on conserve l'égalité possible de la borne supérieure de l'intervalle avec a, l'inégalité étant stricte sur la borne inférieure. Lorsqu'il ne reste plus que deux éléments, le plus grand des deux, s'il est égal à a, est la première occurrence de a.

   On trouve de façon symétrique la dernière occurrence, en considérant cette fois une inégalité large pour la borne inférieure, et une inégalité stricte pour la borne supérieure. Voici les fonctions obtenues, avec la batterie de tests :

```
def PremiereOccurrence(T,a):
    if T[0] > a:
        return -1
    if T[-1] < a:
        return -1
    if T[0] == a:
        return 0
    i = 0
    j = len(T)
    while j-i >= 2:
        k = (i+j)//2
        if T[k] < a:
            i = k
        else:
            j = k
    if T[j] == a:
        return j
    else:
        return -1

def DerniereOccurrence(T,a):
    if T[0] > a:
        return -1
    if T[-1] < a:
        return -1
    if T[-1] == a:
        return len(T)
    i = 0
    j = len(T)
    while j-i >= 2:
```

```
        k = (i+j)//2
        if T[k] <= a:
            i = k
        else:
            j = k
    if T[i] == a:
        return i
    else:
        return -1

T = [0, 3, 6, 6, 6, 6, 6, 6, 6, 8, 8, 8, 9]
print(PremiereOccurrence(T,6))
print(PremiereOccurrence(T,7))
print(PremiereOccurrence(T,8))
print(DerniereOccurrence(T,6))
print(DerniereOccurrence(T,7))
print(DerniereOccurrence(T,8))
```

Les résultats, concluants :

```
2
-1
9
8
-1
11
```

Correction de l'exercice 3 –

1. On positionne mot à chaque emplacement du texte (débutant à l'indice i) et on compare les lettres les unes après les autres, en s'arrêtant à la première différence : s'il y en a une, on décale le mot et on recommence ; sinon, on a trouvé une occurrence et on s'arrête.

```
def cherche(texte,mot):
    n = len(texte)
    m = len(mot)
    i = 0
    while i+ m <= n:
        j = 0
        b = True
        while j < m and b:
            b = (mot[j] == texte[i+j])
            j+=1
        if b:
            return i  # cela arrête l'exécution de la fonction
        i += 1
    return -1 # cas où le mot n'a pas été trouvé

print(cherche("je␣cherche␣'cherche'","cherche"))
```

```
print(cherche("je␣cherche␣'cherche'","'cherche'"))
```

Les résultats obtenus pour les deux tests sont 3 et 11, ce qui est cohérent.

2. On itère, en comptant le nombre de fois où on n'obtient pas -1 dans la fonction de la question 1. On écrit une fonction plus générale, prenant en argument une longueur quelconque de texte, un mot quelconque à rechercher, et un nombre quelconque d'itérations.

```
def frequence(n,mot,N):
    c = 0
    for i in range(N):
        if cherche(motaleatoire(n),mot) != -1:
            c += 1
    return c / N

print(frequence(1000,'llg',1000))
```

Le résultat obtenu est 0.056. Ce résultat est cohérent : obtenir llg à un rang donné a une probabilité de $p = \frac{1}{26^3}$. En négligeant les problèmes de non indépendance, le nombre de llg suit à peu près une loi binomiale de paramètres 997 et p. La probabilité de ne pas obtenir de séquence est donc $P(X = 0) = (1-p)^{997} \simeq 0.945$, donc la probabilité recherchée est à peu près 0.055.

Évidemment, le raisonnement fait est très approximatif : on n'a pas indépendance de l'obtention de llg à deux rangs consécutifs. On peut calculer de façon exacte cette probabilité, mais cette argument approximatif donne déjà un bon ordre de grandeur !

La raison pour laquelle l'hypothèse d'indépendance reste valide est que la probabilité d'obtention de llg est faible : si on a obtenu llg, on ne peut pas l'obtenir aux deux rangs suivants : on remplace quelque chose de petit par quelque chose de nul ; et si on n'a pas obtenu llg, on n'a pas obtenu beaucoup d'informations, donc cela ne conditionne pas grandement l'obtention de llg au rang suivant.

Correction de l'exercice 4 – Voici un petit paradoxe classique (et facile à expliquer) : dans une succession de Pile/Face indépendants, le temps d'attente moyen des séquences PPF et FPP est le même, et pourtant il est plus probable que FPP apparaisse avant PPF. Le paradoxe peut être levé en remarquant que en moyenne, si PPF apparaît avant, FPP ne peut pas apparître au rang suivant, ce qui le retarde ; si PPF apparaît d'abord, alors FPP peut apparaître dès le rang suivant. Ce décalage explique que même si en moyenne, PPF apparaît avant FPP, lorsque ce n'est pas le cas, la différence des rangs est plus importante, et rétablit l'equilibre.

On étudie les séquences de 3 tirages consécutifs, en décalant de 1 à chaque fois. On donne la globalité du programme :

```
import random

def PPF():
    n=3
    u = random.randint(0,1)
    v = random.randint(0,1)
    w = random.randint(0,1)
    while (u,v,w) != (1,1,0):
        u,v = v,w
        w = random.randint(0,1)
        n += 1
    return n
```

```
def esperancePPF():
    S = 0
    for k in range(100000):
        S += PPF()
    return S/100000

def FPP():
    n=3
    u = random.randint(0,1)
    v = random.randint(0,1)
    w = random.randint(0,1)
    while (u,v,w) != (0,1,1):
        u,v = v,w
        w = random.randint(0,1)
        n += 1
    return n

def esperanceFPP():
    S = 0
    for k in range(100000):
        S += FPP()
    return S/100000

def PPFavantFPP():
    u = random.randint(0,1)
    v = random.randint(0,1)
    w = random.randint(0,1)
    while True:
        if (u,v,w) == (1,1,0):
            return 1
        if (u,v,w) == (0,1,1):
            return 0
        u,v = v,w
        w = random.randint(0,1)

def probaPPFavantFPP():
    S = 0
    for k in range(100000):
        S += PPFavantFPP()
    return S/100000
```

On obtient à peu près 8 pour le temps d'attente du premier PPF ou FPP (le dernier tirage effectué pour obtenir cette séquence). Le calcul de cet espérance est classique (et vaut effectivement 8). On aura l'occasion de le faire plus tard dans l'année.

On obtient à peu 0.25 pour la probabilité d'obtenir PPF avant FPP.

Ce déséquilibre, peu intuitif de prime abord, s'explique bien. En fait, tout se détermine par les 2 premiers tirages :

- si les deux premiers tirages ont au moins un F, pour obtenir PPF, il faut avoir obtenu une séquence PP, mais

la première séquence PP est nécessairement précédée de F : FPP apparaît avant PPF.

- si les deux premiers tirages sont PP, pour obtenir FPP, il faut avoir tiré au moins un F avant. Mais le premier F est précédé uniquement de P, en nombre supérieur à 2, donc PPF apparaît avant FPP.

Cela donne bien une probabilité que PPF apparaisse avant FPP égale à $\frac{1}{4}$.