ENSAE 2010 Examen écrit (1 heure)

Initiation à la programmation et à l'algorithmique

Xavier Dupré

Tous documents autorisés.

1

1) Qu'affiche le code suivant ? (2 points)

```
l = []
for i in range (0, 4) :
    c = []
    for j in range (0, i) : c += [ j ]  # ou c.append (j)
    l += [ c ]                          # ou l.append (c)

for c in l : print c
```

2) Même question, qu'affiche le code suivant ? (1 point)

```
l = [1,4,1,5,9]
d = { }
for u in l :
    if u in d : d [u] += 1
    else : d [u] = 1

print d
```

Correction

1)

```
[]
[0]
[0, 1]
[0, 1, 2]
```

2)

```
{1: 2, 4: 1, 5: 1, 9: 1}
```

2

Les exemples suivant contiennent des erreurs, il faut en expliquer la raison et proposer une correction qui respecte l'intention de son concepteur.

1) (1 point)

```
n = 10
l = [i for i in (0,n)]
l[9]
```

```
  File "ecrit.py", line 3, in <module>
    l[9]
IndexError: list index out of range
```

2) (1 point)

```
d = {'un': 1, 'deux': 4}
print deux
```

```
  File "ecrit.py", line 2, in <module>
    print deux
NameError: name 'deux' is not defined
```

3) (1 point)

```
l = [ "mot", "second" ]
print l(0)
```

```
  File "ecrit.py", line 2, in <module>
    print l(0)
TypeError: 'list' object is not callable
```

4) (1 point)

```
l = [4, 2, 1, 3]
ll = l.sort()
print ll[0]
```

```
  File "ecrit.py", line 2, in <module>
          print ll[0]
TypeError: 'NoneType' object is unsubscriptable
```

Correction

1)

```
n = 10
l = [i for i in (0,n)]
print l                    # affiche [0, 10]
```

L'instruction `l`[9] implique que le $10^{\text{ième}}$ élément existe et ce n'est pas le cas. Il faudrait écrire :

```
l = [i for i in range (0,n)]
```

2) L'instruction `print deux` ne contient ni guillemet ni apostrophe : `deux` est considéré comme une variable et celle-ci n'existe pas. Il faudrait écrire `print` "`deux`" ou `print d`["`deux`"].

3) L'instruction `l`(`0`) suppose que `l` est une fonction et c'est en fait une liste. Il faut remplacer les parenthèses par des crochets : `l`[`0`].

4) L'instruction `l.sort`() trie la liste `l`. Elle la modifie et ne retourne aucun résultat, c'est à dire `None`. Pour corriger le programme, il faut faire une copie de la liste `l` en `ll` puis la trier.

```
l = [4, 2, 1, 3]
ll = list (l)
ll.sort()
print ll[0]
```

Dans le cas où on ne cherche que le plus petit élément, on peut aussi écrire :

```
l = [4, 2, 1, 3]
print min(l)
```

3

Une liste `l` contient des nombres et on souhaite les permuter dans un ordre aléatoire. On dispose pour cela de deux fonctions :

```
i = random.randint (0,n) # tire aléatoirement un nombre entier entre 0 et n inclus

l.sort ()                # trie la liste l quel que soit son contenu
```

Exemple :

```
l = [4,5,3,7,4]
l.sort ()
print l             # affiche [3, 4, 4, 5, 7]
```

Dans le cas d'une liste de couples (ou 2-uple), chaque élément de la liste est trié d'abord selon la première coordonnée puis, en cas d'égalité, selon la seconde coordonnée. Cela donne :

```
l = [ (1,2), (0,10), (4,3), (5,0), (0,9) ]
l.sort ()
print l             # affiche [(0, 9), (0, 10), (1, 2), (4, 3), (5, 0)]
```

Les quelques lignes qui suivent illustrent le résultat souhaité. Il reste à imaginer la fonction qui permute de façon aléatoire la liste `l`. (3 points)

```
def permutation_aleatoire (l) :
    ....
    return ...

print permutation_aleatoire ([1,2,3,4])   # affiche [3,1,4,2]
```

Correction

Pour répondre au problème, on s'inspire d'une méthode qui trie une liste tout en récupérant la position de chaque élément. Pour ce faire, on construit une liste de couples (élément, position).

```
tab = ["zéro", "un", "deux"] # tableau à trier
pos = [ (tab [i],i) for i in range (0, len (tab)) ] # tableau de couples
pos.sort () # tri
print pos # affiche [('deux', 2), ('un', 1), ('zéro', 0)]
```

Pour effectuer une permutation, on construit une liste de couples (position aléatoire, élement). En triant sur une position aléatoire, on dérange la liste des éléments de façon aléatoire. Il suffit alors de récupérer la seconde colonne de cette liste de couples.

```
from random import randint

def permutation (liste) :
    alea        = [ randint (0,len (liste)) for i in liste ]
    couple      = [ (r,l) for r,l in zip (alea,liste) ]
    couple.sort ()
    permutation = [ l[1] for l in couple ]
    return permutation

liste = [ i*2 for i in range (0,10) ]
permu = permutation (liste)
print liste  # affiche [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
print permu  # affiche [2, 6, 16, 10, 8, 14, 0, 12, 18, 4]
```

Ce n'était pas la seule réponse possible mais les autres solutions ne nécessitaient pas de tri comme celle-ci :

```
from random import randint
import copy

def permutation (liste) :
    li = copy.copy (liste)
    for i in xrange (0, len (liste)) :
        n = randint (0, len (liste)-1)
        m = randint (0, len (liste)-1)
        s = li [n]
        li [n] = li [m]
        li [m] = s
    return li
```

Une permutation peut se décomposer en une somme d'au plus n transpositions. Une transposition est une permutation qui échange uniquement deux éléments. La fonction `permutation` choisit donc n transpositions aléatoires, si on considère que l'identité est aussi une transposition. Une autre possibilité était :

```
from random import randint
import copy

def permutation (liste) :
    li = copy.copy (liste)
    for i in xrange (0, len (liste)) :
        n = i
        m = randint (0, len (liste)-1)
        s = li [n]
        li [n] = li [m]
        li [m] = s
    return li
```

Une autre option consistait à reproduire un tirage avec remise :

```
from random import randint
import copy

def permutation (liste) :
    li = [ ]
    n  = len (liste)
    for i in xrange (0, n) :
```

```
        m = randint (0, len (liste)-1)
        li.append (liste [m])
        del liste [m]          # on supprime l'élément tiré au hasard
    return li
```

La plupart des élèves ont oublié qu'il était possible de supprimer un élément d'une liste et ont reconstruit une nouvelle liste d'où l'élément choisi aléatoirement était absent. Une erreur fréquente était de confondre éléments de la liste et positions, ce qui amenait parfois à écrire :

```
from random import randint
import copy

def permutation (liste) :
    li = [ ]
    n  = len (liste)
    for i in xrange (0, n) :
        m = randint (0, len (liste)-1)
        if liste [m] not in li :
            li.append (liste [m])
    return li
```

Cette dernière version a deux problèmes. Le premier est que si on tire un élément déjà tiré, on n'ajoute aucun élément et la liste qu'on retourne est plus courte que la liste à permuter. Le second problème découle du premier : ce cas est toujours vrai si la liste à permuter contient deux éléments identiques. Les positions sont uniques mais pas forcément les éléments à trier.

Au vu des réponses des élèves, celles qui échangeaient les positions des éléments semblaient plus intuitives que celles s'appuyant sur un tri. Son coût est d'ailleurs plus élevé puisqu'il est égal à celui du tri : $O(n \ln n)$. Son avantage est de pouvoir construire une permutation d'une liste démesurément grande voire de l'effectuer sur plusieurs ordinateurs simultanément. Supposons que la liste à permuter ne tienne pas dans la mémoire de l'ordinateur. La première étape qui consiste à donner un nombre aléatoire à un élément peut très bien se faire en ne conservant en mémoire que l'élément courant. Le tri lui aussi, s'il est effectué par fusion, pourra se faire par morceaux. En revanche, les méthodes qui utlisent les transpositions requiert la liste en mémoire dans sa globalité puisqu'il faut accéder à n'importe quel élément à tout moment. Dans le premier cas (tri), on dit que l'accès est séquentiel. Dans le second cas (transposition), il est aléatoire. Un dernier avantage : la méthode avec tri utilise des fonctions standard qui réduisent son écriture et son temps de conception par voie de conséquence.

4

$N = 1000$ personnes sont classées en quatre catégories. On souhaite convertir un numéro de classe (1, 2, 3, ou 4) en un nom de catégorie plus explicite. On considère la fonction suivante :

```
def nom_csp (csp) :                    # csp est un entier compris entre 1 et 4
    if   csp == 1  : return "cat. A"
    elif csp == 2  : return "cat. B"
    elif csp == 3  : return "cat. C"
    else           : return "cat. D"
```

1) Connaissant le nombre de gens n_i dans chaque catégorie pour ces $N = 1000$ personnes ($n_1 + n_2 + n_3 + n_4 = N$), quel est le nombre de comparaisons [1] effectuées lors des N appels à la fonction

1. Une comparaison correspond ici à un test, soit d'égalité ==, mais aussi d'infériorité <= par la suite.

`nom_csp` en fonction des nombres (n_i) ? (1 point)

2) On suppose que les $(n_i)_{i\in\{1,2,3,4\}}$ sont tous différents, le nombre de comparaisons effectuées lors des N appels à la fonction `nom_csp` dépend-il de l'ordre des lignes ? Si oui, y a-t-il un ordre optimal qui minimise le nombre de comparaisons pour effectuer ces N conversions ? (2 points)

3) On propose une autre fonction :

```
def nom_csp (csp) :                     # csp est un entier compris entre 1 et 4
    if csp <= 2  :
        if csp == 1 : return "cat. A"
        else        : return "cat. B"
    else :
        if csp == 3 : return "cat. C"
        else        : return "cat. D"
```

Combien de comparaisons la fonction `nom_csp` effectue si le paramètre `csp` vaut respectivement 1, 2, 3 ou 4 ? (1 point)

4) On suppose maintenant que $n_2 = n_3 = n_4$ et que $n_1 + n_2 + n_3 + n_4 = N$. Est-il préférable de choisir :

1. toujours la fonction de la question 1
2. toujours la fonction de la question 3
3. ça dépend de n_1, dans ce dernier cas, préciser quelle condition doit vérifier n_1 pour choisir la fonction de la question 1.

(2 points)

Correction

1) On peut résumer le coût de la fonction `nom_csp` en fonction de `csp` par le tableau suivant :

`csp`	1	2	3	4
nombre de tests	1	2	3	3

On en déduit le nombre de tests C pour classer 1000 observations :

$$C = n_1 + 2n_2 + 3n_3 + 3n_4$$

2) Si les $(n_i)_i$ sont tous différents, le nombre de tests dépend évidemment de l'ordre des lignes de la fonction `nom_csp`. Pour minimiser le nombre de tests, il suffit de les ordonner en plaçant la catégorie la plus importante en premier, la moins importante en dernier : il faut trier les $(n_i)_i$ par ordre décroissant. On suppose que :

$$n_{i_1} > n_{i_2} > n_{i_3} > n_{i_4}$$

Supposons que cet ordre correspondant à : cat C. > cat A. > cat B. > cat D. Dans ce cas, on devrait écrire :

```
def nom_csp (csp) :
    if   csp == 3  : return "cat. C"
    elif csp == 1  : return "cat. A"
    elif csp == 2  : return "cat. B"
    else           : return "cat. D"
```

Cette assertion se démontre mathématiquement par l'absurde. On suppose que la configuration optimale n'est pas celle qui respecte l'ordre décroissant des $(n_i)_i$. Dans cette configuration, il existe k, l tels que : $n_{i_k} < n_{i_l}$. On vérifie aisément qu'en les permutant, on obtient une configuration meilleure que la précédente. C'est contradictoire avec l'hypothèse de départ.

3) On construit le même tableau que celui de la première question pour cette nouvelle fonction :

`csp`	1	2	3	4
nombre de tests	2	2	2	2

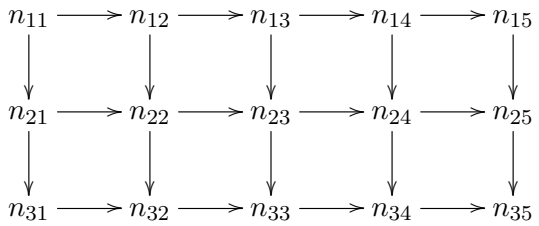
4) La réponse est : cela dépend. Dans le premier cas, le coût de la fonction est : $n_1 + (2+3+3)n_2 = n_1 + 8\frac{N-n_1}{3}$. La seconde fonction à pour coût : $2N$. On choisit la première version si :

$$\begin{aligned} n_1 + 8\frac{N-n_1}{3} &< 2N \\ n_1\left(1-\frac{8}{3}\right) &< N\left(2-\frac{8}{3}\right) \\ -\frac{5}{3}n_1 &< -\frac{2}{3}N \\ n_1 &> \frac{2}{5}N \end{aligned}$$

La seconde fonction est le plus souvent la bonne manière de faire. Elle a également l'avantage d'être aussi performante quel que soit le poids de chaque catégorie. C'est celle qu'utilise les dictionnaires. Mais dans les cas presque dégénérés, la première méthode est plus rapide à condition que ce soit toujours les mêmes catégories qui soient les plus fréquentes. Il est même possible d'envisager un mélange des deux : on traite d'abord les quelques catégories très fréquentes avec la première fonction, on traite les autres avec la seconde fonction.

5

Cet exercice ne nécessite aucune connaissance particulière sur les graphes bien que la figure suivante décrive un graphe orienté : chaque flèche représente un chemin possible entre deux nœuds ou points de passage, on ne peut les parcourir que dans le sens de la flèche. On veut répondre à la question : combien y a-t-il de chemins possibles pour aller du nœud n_{11} au nœud n_{35} sachant qu'on ne peut que descendre ou aller à droite ?



1) Pour arriver au nœud n_{23}, on peut soit venir du nœud n_{22} soit du nœud n_{13}. On note $N(n_{ij})$ le nombre de chemins possibles pour arriver au nœud n_{ij} en partant du nœud n_{11}. Il n'y a que deux

façons d'arriver au nœud n_{23}, on en déduit que :

$$N(n_{23}) = N(n_{22}) + N(n_{13})$$

On utilise cette formule pour construire le programme suivant :

```
def calculeN (I,J) :  # I et J sont des entiers comme par exemple I=3 et J=5

    nombre = [ [0 for i in range (0, J) ] for i in range (0, I) ]
      # nombre est alors égal à [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

    for i in range (0, I) :
        nombre [i][0] = 1

    for j in range (0, I) :
        nombre [0][j] = 1

    for i in range (1, I) :
        for j in range (1, J) :
            nombre [i][j] = nombre [i-1][j] + nombre [i][j-1]

    return nombre

nombre = calculeN (3,5)
for l in nombre : print l
```

Le programme retourne le résultat suivant :

```
[1, 1, 1, 0, 0]
[1, 2, 3, 3, 3]
[1, 3, 6, 9, 12]
```

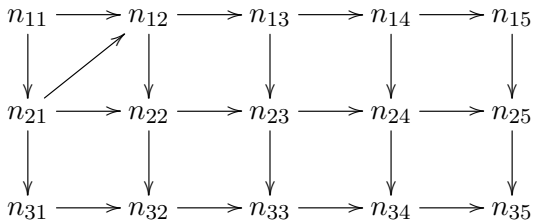
Le programme est incorrect. Indiquez où se trouve l'erreur ou les erreurs. Quelle est finalement la valeur du résultat pour le nœud n_{35} ? (1 point)

2) On observe que le résultat de la fonction `calculeN` une fois corrigée vérifie :

$$N(n_{ij}) = \begin{pmatrix} i+j-2 \\ i-1 \end{pmatrix} = \frac{(i+j-2)!}{(i-1)!\,(j-1)!}$$

Expliquer pourquoi c'est vrai. (1 point)

3) On ajoute un chemin entre les nœuds n_{21} et n_{12}. Quelle instruction faut-il ajouter à la fonction `calculeN` pour tenir compte de cet arc et obtenir le nombre de chemins possibles entre les nœuds n_{11} et n_{35} ? (2 points)



Correction

1) Les deux zéros sur la première ligne de résultat sont surprenant. Ils sont dûs au fait que $J > I$ lorsqu'on écrit les deux lignes suivantes :

```
    for j in range (0, I) :   # il faut écrire range (0, J)
        nombre [0][j] = 1
```

On corrige ces deux zéros et les deux dernières colonnes pour obtenir :

```
[1, 1, 1, 1, 1]
[1, 2, 3, 4, 5]
[1, 3, 6, 10, 15]
```

2) A chaque nœud, il est possible soit de descendre, soit d'aller à droite. Pour aller au nœud (i, j), il faut descendre i fois et aller j fois à droite et peu importe l'ordre. Le nombre de chemins correspond au nombre de combinaisons possibles entre i "descendre" et j "à droite". Ceci explique la formule citée dans l'énoncé.

3) Il n'existe pas de solution unique. Il suffit de remarquer qu'il existe maintenant 2 façons d'aller au nœud n_{12} ainsi qu'aux nœuds n_{13}, n_{14}, n_{15}. Il n'est toujours pas possible de revenir en arrière. On peut donc modifier le programme comme suit :

```
def calculeN (I,J) :

    nombre = [ [0 for i in range (0, J) ] for i in range (0, I) ]

    for i in range (0, I) :
        nombre [i][0] = 1

    for j in range (1, J) :
        nombre [0][j] = 1
    nombre [0][1:5] = [2,] * (J-1)  ##### ligne ajoutée

    for i in range (1, I) :
        for j in range (1, J) :
            nombre [i][j] = nombre [i-1][j] + nombre [i][j-1]

    return nombre

nombre = calculeN (3,5)
for l in nombre : print l
```

On obtient :

```
[1, 2, 2, 2, 2]
[1, 3, 5, 7, 9]
[1, 4, 9, 16, 25]
```