Examen Programmation ENSAE

première année 2009

Examen écrit (1 heure)

Tous documents autorisés.

1 Portée des variables

1) Répondre aux deux questions qui suivent ? (2 points)

```
def f():
    x = 0
    print x


f()

def g():
    print x

g()
```

Le premier programme produit l'erreur qui suit. Pourquoi ?

```
Traceback (most recent call last):
  File "p1.py", line 11, in <module>
    g()
  File "p1.py", line 9, in g
    print x
NameError: global name 'x' is not defined
```

```
def f():
    x = 0
    print x

x = 1
f()

def g():
    print x

g()
```

Qu'affiche le second programme ?

2) Le premier des deux programmes suivant s'exécute sans erreur, qu'affiche-t-il ? (1 point)

```
a       = [ None, 0 ]
b       = [ None, 1 ]
a [0]   = b
b [0]   = a
print a [0][0] [0][0] [0][0] [1]
```

```
import copy
a       = [ None, 0 ]
b       = [ None, 1 ]
a [0]   = copy.copy (b)
b [0]   = a
print a [0][0] [0][0] [0][0] [1]
```

3) On ne sait rien du second programme qui utilise le module `copy`. S'exécute-t-il sans erreur ? Si la réponse est positive, qu'affiche-t-il ? Si la réponse est négative, pourquoi ? (1 point)

Correction

1) L'erreur se produit dans la fonction `g` car il n'existe aucune variable globale ou locale portant le nom `x`, l'instruction `print x` ne peut donc être exécutée. La variable `x` définie dans la fonction `f` est une variable locale et son existence est limité à la fonction `f`.

A contrario, le second programme définit une variable globale `x = 1` avant l'appel à la fonction `g`. Le programme affiche donc :

```
0
1
```

2) L'instruction `print a[0][0][0][0][0][0][1]` peut surprendre mais elle est tout-à-fait correcte dans le premier programme. Lorsqu'il s'agit d'une liste, une affectation ne copie pas la liste mais donne à cette liste un second identifiant :

```
a       = [ None, 0 ]
b       = [ None, 1 ]
a [0]   = b   # b et a [0] désignent la même liste
b [0]   = a   # b [0] et a désignent la même liste
```

On peut résumer les listes `a` et `b` par l'écriture suivante :

$$\begin{array}{l} a \Longleftrightarrow [b, 0] \\ b \Longleftrightarrow [a, 0] \end{array}$$

On peut écrire que :

$$\begin{array}{rr} & a[0][0][0][0][0][0][1] \\ = & b[0][0][0][0][0][1] \\ = & a[0][0][0][0][1] \\ = & a[0][0][1] \\ = & a[1] \\ = & 0 \end{array}$$

Le programme affiche donc :

```
0
```

3) Dans le second programme, l'instruction `copy.copy(b)` créée une copie `c` de la liste `b`. On peut résumer les listes `a` et `b` par l'écriture suivante :

$$\begin{array}{l} a \Longleftrightarrow [c, 0] \Longleftrightarrow [[None, 1], 0] \\ b \Longleftrightarrow [a, 0] \end{array}$$

Par conséquent :

$$\begin{array}{rr} & a[0][0][0][0][0][0][1] \\ = & c[0][0][0][0][0][1] \\ = & None[0][0][0][0][1] \\ \Longleftarrow & erreur \end{array}$$

2 Somme d'éléments aléatoires

1) On considère le programme suivant :

```
import random
def somme_alea (tableau, n) :
    s = 0
    for i in range (0, n) :
        h = random.randint (0, len (tableau))
        s += tableau [h]
    print s / n
x = somme_alea ( range (0,100), 10)
print "x = ", x
```

A sa première exécution, il affiche :

```
44
x =  None
```

Ce n'est pas tout-à-fait le résultat attendu, le terme `None` ne devrait pas apparaître, que faut-il faire pour le corriger ? (1 point)

2) On s'aperçoit également que le programme provoque l'erreur suivante environ une fois sur dix :

```
Traceback (most recent call last):
  File "examen2009.py", line 10, in <module>
    print somme_alea ( range (0,100), 10)
  File "examen2009.py", line 8, in somme_alea
    s += tableau [h]
IndexError: list index out of range
```

Pourquoi ? Expliquer également pourquoi le programme ne provoque cette erreur qu'une fois sur dix environ ? Une réponse statistique est attendue. (2 points)

3) Enfin, le résultat attendu est réel alors que le résultat du programme est toujours entier, pourquoi ? (1 point)

Correction

1) Il suffit de remplacer l'instruction `print s/n` par rapport `return s/n`. Le programme affichera :

```
44
```

2) L'erreur nous apprend que l'instruction `s+ = tableau[n]` provoque une erreur parce que la variable `h` contient un entier soit négatif, soit supérieur ou égal à `len(tableau)`. C'est ce second cas qui se produit parfois. La fonction `random.randint` retourne un entier aléatoire compris entre `0` et `len(tableau)` inclus. Il faudrait changer l'instruction précédente en :

```
h = random.randint (0, len (tableau)-1)
```

L'erreur ne se produit pas toujours car le programme tire un nombre aléatoire qui peut être égal à `len(tableau)`. En fait, il faut calculer la probabilité que le programme a de tirer le nombre `len(tableau)` que l'on notera n dans la suite. Le programme fait la moyenne de 10 nombres aléatoires tirés au hasard mais uniformément dans l'ensemble $\{0, \ldots, n\}$. La probabilité que le programme fonctionne est la probabilité p de ne jamais tomber sur le nombre n en 10 tirages :

$$p = \left(\frac{n-1}{n}\right)^{10} = \left(1 - \frac{1}{n}\right)^{10} \tag{1}$$

Pour le programme en exemple :

$$p = (1 - \frac{1}{101})^{10} \sim 1 - \frac{10}{101} \sim 0,9 \tag{2}$$

Si $0,9$ est la probabilité que le programme fonctionne, 0,1 est la probabilité que le programme produise une erreur. On retrouve l'estimation citée dans l'énoncé. Le calcul non approché donne une valeur de 0,0947.

Beaucoup d'élèves ont répondu que le programme produisait une erreur une fois sur dix environ car appeler 10 fois une fonction qui produit une erreur une fois sur 100 revient à faire le calcul : $10 * \frac{1}{101} \sim \frac{1}{10}$. Même si ce raisonnement aboutit à résultat proche de la bonne réponse, il est tout-à-fait faux. Il suffit pour s'en convaincre d'écrire à un programme qui appelle 200 fois la fonction `somme_alea`. D'après ce raisonnement, la probabilité que le programme produit une erreur serait de $200 * \frac{1}{101} \sim 2$ ce qui est impossible.

3) Le programme ne fait que manipuler des entiers, l'instruction `print s/n` effectue donc une division entière. Le programme ne peut produire qu'un résultat entier et non réel. Il faudrait écrire pour cela `print float(s)/float(n)`.

3 Coût d'une moyenne mobile centrée

1) Quel est le coût de la fonction `moyenne_mobile` en fonction de la longueur `n` de la suite `suite` et de la largeur de la moyenne mobile `m` à un facteur multiplicatif près ? (1 point)

```
def moyenne_mobile (suite, m) :
    res = []
    for i in range (m, len (suite)-m) :
        s = 0
        for j in range (-m,m+1) : s += suite [i+j]
        res.append (  float (s) / (m*2+1) )
    return res
```

2) On peut faire plus rapide en remplaçant les lignes en pointillés dans le programme qui suit. (1 point)

```
def moyenne_mobile (suite, m) :
    res = []
    for i in range (m, len (suite)-m) :
        if i == m :
            # ... ligne à remplacer
        else :
            # ... ligne à remplacer
        res.append (  float (s) / (m*2+1) )
    return res
```

3) Quel est le coût du nouveau programme toujours en fonction de la longueur `n` de la suite `suite` et de la largeur de la moyenne mobile `m` ? (1 point)

Correction

1) La fonction `moyenne_mobile` inclut deux boucles imbriquée. La première est de longueur $n - 2m$, la seconde est de longueur $2m$. Le coût de la fonction est donc en $O((n-2m)2m) = O(2mn - 4m^2)$.

On peut supposer que m est plus petit devant n et négliger le second terme. Le coût recherché est : $O(mn)$.

2) Lorsqu'on calcule la valeur de la somme `s` pour les itérations i et $i+1$, on ajoute presque les mêmes termes (on cote $\texttt{s} = \texttt{suite}$ pour simplifier) :

$$\begin{array}{l|llllll} \texttt{i} & \texttt{s}[\texttt{i}-\texttt{m}] + & \texttt{s}[\texttt{i}-\texttt{m}+1] + & \ldots & +\texttt{s}[\texttt{i}+\texttt{m}-1] & +\texttt{s}[\texttt{i}-\texttt{m}] & \\ \texttt{i}+1 & & \texttt{s}[\texttt{i}-\texttt{m}+1] + & \ldots & +\texttt{s}[\texttt{i}+\texttt{m}-1] & +\texttt{s}[\texttt{i}-\texttt{m}] & +\texttt{s}[\texttt{i}-\texttt{m}+1] \end{array}$$

Pour passer d'une somme à la suivante, il suffit d'ajouter un terme et d'en soustraire un. Il faut également calculer la première somme lorsque $\texttt{i} = \texttt{m}$. Ceci donne :

```
def moyenne_mobile (suite, m) :
    res = []
    for i in range (m, len (suite)-m) :
        if i == m :
            s = sum ( suite [i-m : i+m+1] )
        else :
            s = s - suite [i-m-1] + suite [i+m]
        res.append (  float (s) / (m*2+1) )
    return res
```

3) La fonction contient toujours deux boucles imbriquées (la fonction `sum` cache une boucle) mais la seconde boucle n'est exécutée qu'une seule fois. Le coût de la fonction n'est plus que $O(n+m)$. Comme n est souvent beaucoup plus grand que m, on peut négliger le second terme : $O(n)$.

4 Deviner l'objectif d'une fonction

1) Que fait la fonction suivante ? (2 points)

```
def fonction_mystere (tableau) :
    for i in range (0, len (tableau)) :
        if tableau [i] % 2 != 0 : continue
        pos = i
        for j in range (i+1, len (tableau)) :
            if tableau [j] % 2 != 0 : continue
            if tableau [pos] < tableau [j] : pos = j
        ech           = tableau [i]
        tableau [i]   = tableau [pos]
        tableau [pos] = ech
```

Correction

1) Pour comprendre l'objectif de la fonction, il suffit d'enlever les deux lignes suivantes :

```
        if tableau [i] % 2 != 0 : continue
            if tableau [pos] < tableau [j] : pos = j
```

La fonction devient :

```
def fonction_mystere (tableau) :
    for i in range (0, len (tableau)) :
        pos = i
```

```
        for j in range (i+1, len (tableau)) :
            if tableau [pos] < tableau [j] : pos = j
        ech           = tableau [i]
        tableau [i]   = tableau [pos]
        tableau [pos] = ech
```

Cette fonction correspond à un tri par ordre décroissant. Les deux lignes supprimées ne font qu'éviter l'exécution des itérations pour les éléments impairs. La fonction trie par ordre décroissant les éléments pairs d'un tableau tout en laissant inchangés les éléments impairs. Par exemple :

```
a = [ 0, 1, 4, 3, 5, 2 ]
print a                     # affiche [ 0, 1, 4, 3, 5, 2 ]
b = fonction_mystere (a)
print b                     # affiche [ 4, 1, 2, 3, 5, 0 ]
```

5 Calcul du plus court chemin dans un graphe

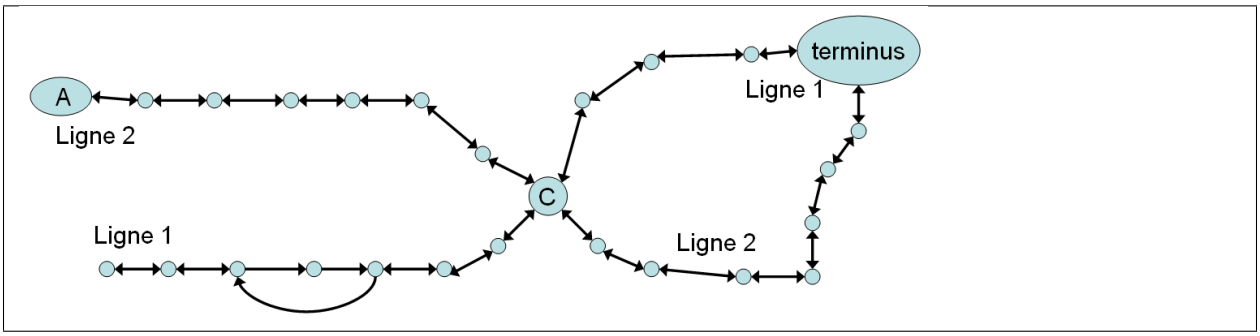

Figure 1 : *Figure représentant deux lignes de métro qui se croisent en C et qui se rejoignent en terminus. Chaque station est représentée par un point, chaque arc indique que deux stations sont reliées par la même ligne de métro. Tous les arcs sont à double sens sauf trois car dans ce cas, le métro ne suit pas le même chemin dans les deux sens.*

La figure 1 représente deux lignes de métro. On représente ce graphe par une matrice carrée $M = (m_{ij})$ dans laquelle il y a autant de lignes que de stations. Chaque élément m_{ij} représente la durée de transport entre les stations i et j dans le sens $i \longrightarrow j$. m_{ij} est positif si la liaison existe, infini (ou très grand) si elle n'existe pas.

1) La matrice M est-elle symétrique ? Pourquoi ? (1 point)

2) On détermine le chemin le plus court entre les stations A et *terminus* grâce à un algorithme (Bellman, Dikjstra, ...). La solution indique qu'il faut changer de ligne à la station C. Toutefois, cette solution ne prend pas en compte la durée du changement de ligne. C'est pourquoi on souhaite modifier la matrice M en y insérant la durée du changement de ligne à la station C de façon à pouvoir utiliser le même algorithme pour déterminer le plus court chemin. Comment faire ? La réponse peut être soit rédigée, soit graphique en s'inspirant de la figure 1 sans nécessairement la reproduire en entier. (2 points)

Correction

1) La matrice n'est pas symétrique sur la ligne 1, il existe trois arcs orientés : le passage du métro ne s'effectue que dans un seul sens. Si on nomme i, j, k les trois stations concernées, les coefficients m_{ij} et m_{ji} sont différents. L'un des deux sera infini puisque la connexion n'existe pas.

2) Il faut nécessairement ajouter des lignes ou des colonnes dans la matrice M puisqu'il faut ajouter une nouvelle durée au précédent problème. Pour introduire les durées de changements, on considère que les nœuds du graphe représente un couple $(station, ligne)$. De cette façon, un coefficient m_{ij} de la matrice représente :

– une durée de transport si les nœuds i et j sont deux stations différents de la même ligne
– une durée de changement si les nœuds i et j sont en fait la même station mais sur deux lignes différentes.

La réponse graphique est représentée par la figure 2. Il faut noter également que cette solution permet de prendre également en compte un couple de stations reliées par deux lignes différentes ce qui n'était pas possible avec la matrice décrite dans l'énoncé. C'est le cas par exemple des stations Concorde et Madeleine qui sont reliées par les lignes 8 et 12 du métro parisien.

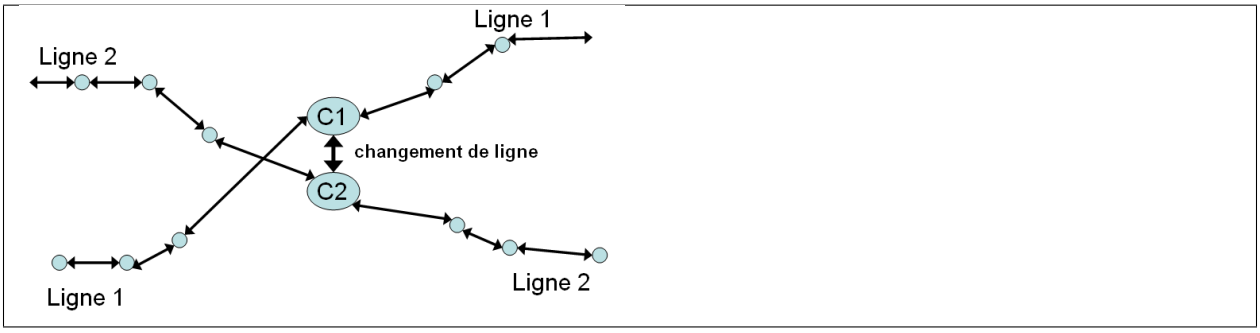

Figure 2 : *Chaque nœud du graphe représente une station pour une ligne donnée. Cela signifie que si une station est présente sur deux lignes, elle est représentée par deux nœuds. Un arc peut donc relier deux stations différents de la même ligne ou deux stations identiques de deux lignes différentes. Dans ce second cas, c'est la durée d'un changement de ligne.*

Quelques élèves ont ajouté trois stations au lieu d'une seule, la station C étant représentée par quatre stations, ces quatre stations étant reliées par des arcs de longueurs nulles ou égales au temps de changement de ligne. C'est une solution même si elle est inutilement compliquée.

Un élève a proposé une solution qui n'implique pas l'ajout d'une station. Cette solution est envisageable dans le cas de deux lignes de métro mais elle peut devenir compliquée à mettre en place dans le cas de plusieurs lignes avec plusieurs stations consécutives sur la même ligne (voir figure 3).

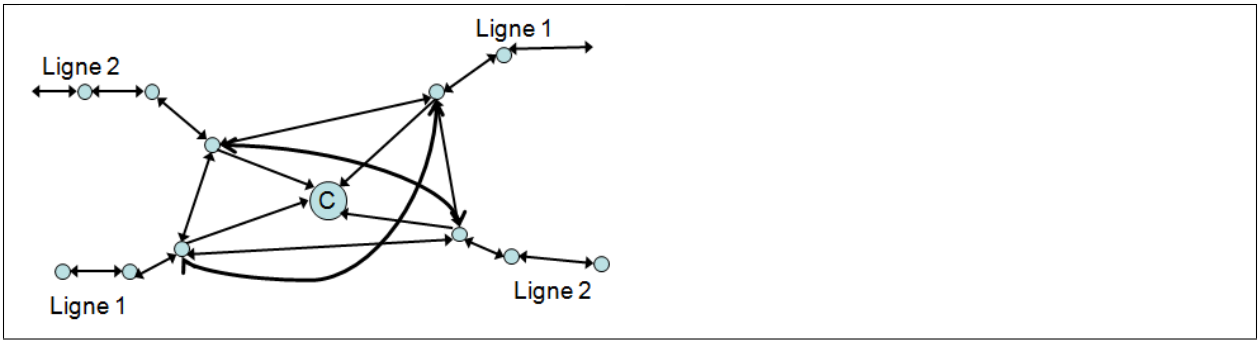

Figure 3 : *Toutes les connexions menant à la station C sont maintenant à sens unique : elles ne permettent que d'arriver en C mais pas d'en repartir. Pour aller à une autre station voisine de C, il faut passer par l'un des 6 arcs ajoutés qui incluent ou non un temps de transfert selon que ces stations appartiennent à la même ligne. Cette solution n'est pas aussi simple que la première et peut rapidement devenir compliquée si une même station appartient à trois, quatre, lignes.*

La solution impliquait nécessairement la création d'arcs (ou de connexion) entre différentes stations. Il ne suffisait pas d'ajouter le temps de changement à des coefficients de la matrice de transition, cette opération a pour conséquence de modifier les distances d'au moins un voyage sur une même ligne.

6 Héritage

1) On construit une classe dont le but est de faire une boucle.

```
class Boucle :
    def __init__ (self, a, b) :
        self.a = a
        self.b = b
    def iteration (self, i) :              # contenu d'une itération
        pass
    def boucle (self) :                    # opère la boucle
        for i in range (self.a, self.b) :
            self.iteration (i)
```

On l'utilise pour faire une somme : que fait le programme suivant ? Quel est le résultat numérique affiché ? (1 point)

```
class Gauss (Boucle) :
    def __init__ (self, b) :
        Boucle.__init__ (self, 1, b+1)
        self.x = 0

    def iteration (self,i) :
        self.x += i

g = Gauss (10)
g.boucle ()
print g.x
```

2) De façon la plus concise possible, écrire la classe `Carre` qui hérite de la classe `Gauss` et qui fait la somme des `n` premiers entiers au carré. (1 point)

3) On souhaite écrire la classe `Boucle2` qui effectue une double boucle : une boucle imbriquée dans une boucle. Le barême de cette question est le suivant :
– 1 point si le code de la classe `Boucle2` contient deux boucles (`for` ou `while`).
– 2 points si le code de la classe `Boucle2` ne contient qu'une boucle (`for` ou `while`).

Correction

1) Le programme effectue la somme des entiers compris entre 1 et 10 inclus : $10 * 11/2 = 55$. Le programme affiche 55.

2) Le plus court est de repartir de la classe `Gauss` qui redéfinit le constructeur dont on a besoin.

```
class Carre (Gauss) :
    def iteration (self,i) :
        self.x += i*i           # ligne changée

g = Carre (10)
g.boucle ()
print g.x
```

3) La version incluant deux boucles est la plus simple à présenter. On modifie la classe `Boucle` de façon à avoir deux boucles. La classe `Gauss2` est un exemple d'utilisation de la classe `Boucle2`.

```
class Boucle2 (Boucle) :
    def __init__ (self, a, b, c, d) :
        Boucle.__init__ (self, a, b)
        self.c = c
        self.d = d
    def iteration (self, i, j) :
        pass
    def boucle (self) :
        for j in range (self.c, self.d) :
            for i in range (self.a, self.b) :
                self.iteration (i,j)

class Gauss2 (Boucle2) :
    def __init__ (self, a, b) :
        Boucle2.__init__ (self, 1, a+1, 1, b+1)
        self.x = 0
    def iteration (self, i,j) :
        self.x += i+j
```

La méthode `iteration` inclut maintenant deux paramètres `i` et `j` et il paraît difficile de réutiliser la méthode `iteration` de la classe `boucle`. Un moyen consiste à ajouter l'attribut `i` à la classe `Boucle2`. Il faut également modifier la classe `Gauss2`.

```
class Boucle2 (Boucle) :
    def __init__ (self, a, b, c, d) :
        Boucle.__init__ (self, a, b)
        self.c = c
        self.d = d
    def iteration (self, i) :
        pass
    def boucle (self) :
        for j in range (self.c, self.d) :
            self.j = j
```

```
            for i in range (self.a, self.b) :
                self.iteration (i)

class Gauss2 (Boucle2) :
    def __init__ (self, a, b) :
        Boucle2.__init__ (self, 1, a+1, 1, b+1)
        self.x = 0
    def iteration (self, i) :
        self.x += self.j+i
```

On peut maintenant supprimer une boucle dans la méthode `boucle` de la classe `Boucle2`. La classe `Gauss2` reste quant à elle inchangée.

```
class Boucle2 (Boucle) :
    def __init__ (self, a, b, c, d) :
        Boucle.__init__ (self, a, b)
        self.c = c
        self.d = d
    def iteration (self, i) :
        pass
    def boucle (self) :
        for j in range (self.c, self.d) :
            self.j = j
            Boucle.boucle (self)  # suppression d'une boucle
                                  # caché dans la classe Boucle
```

La solution suivante est sans doute la plus simple puisqu'on n'utilise pas de subterfuge tel que `self.j = j`. On surcharge cette fois la méthode `iteration` et non plus `boucle`. Mais il faut créer une seconde méthode `iteration2` acceptant `i, j` comme paramètres [1].

```
class Boucle2 (Boucle) :
    def __init__ (self, a, b, c, d) :
        Boucle.__init__ (self, a, b)
        self.c = c
        self.d = d

    def iteration (self, i) :
        for j in range (self.c, self.d) :
            self.iteration2 (i,j)

    def iteration2 (self, i,j) :
        pass


class Gauss2 (Boucle2) :
    def __init__ (self, a, b) :
        Boucle2.__init__ (self, 1, a+1, 1, b+1)
        self.x = 0
    def iteration2 (self, i, j) :
        self.x += j+i
```

Il existait une autre méthode pour faire disparaître une boucle sans doute plus simple que ces manipulations qui utilise une façon simple de manipuler les indices. C'est le même mécanisme que celui utilisé pour représenter une matrice avec une seule liste et non une liste de listes. Ce n'était pas la réponse souhaitée mais elle n'introduit qu'une seule boucle également.

1. Le langage *C++* permettrait d'appeler la méthode `iteration(self, i, j)` car ce langage différencie les méthodes acceptant des nombres différents de paramètres.

```
class Boucle2 (Boucle) :
    def __init__ (self, a, b, c, d) :
        Boucle.__init__ (self, a, b)
        self.c = c
        self.d = d
    def iteration (self, i, j) :
        pass
    def boucle (self) :
        ab = self.b - self.a
        cd = self.d - self.c
        n  = ab * cd
        for k in range (0, n) :
            i = k % ab + self.a
            j = k / ab + self.c
            self.iteration (i,j)
```