La récursivité comme moyen d'explorer l'arbre de toutes les possibilités

F. Didier

26 février 2014

Résumé

La notion de récursivté en informatique est introduite en général dans des problèmes où l'on applique le principe "diviser pour règner" afin de pouvoir les résoudre (Tour de Hanoï, tri dichotomique, quick-sort, dénombrements,...). Ce n'est pas la seule utilité de la récursivité.
Il y a une seconde classes de programmes récursifs. Ce sont ceux qui doivent manipuler des structures de données dont la définition est récursive, structures de données fondamentales en informatique (listes, arbres ordonnés, arbres binaires, ...). En effet, dans ces cas, la définition récursive de la structure des données induit tout naturellement une écriture récursive des programmes qui les manipulent.
Enfin, il existe une troisième classe de programmes récursifs. C'est celle que nous nous proposons de présenter ici à travers quelques exemples.
On est souvent confronté à des problèmes où apparement la seule façon de pouvoir les résoudre se résume à l'énumération de toutes les possibilités pour éventuellement trouver une solution. Soulignons que cette façon de procéder ne permet de trouver la solution recherchée que si la combinatoire du problème n'est pas trop importante. En effet, comme on l'illustrera, si envisager tous les cas possibles fournit théoriquement la solution, celle ci ne sera trouver qu'après un calcul dont le temps est généralement prohibitif : plusieurs jours, plusieurs années, voire plusieurs siècles !

1 Un premier exemple, énumération des n-uplets

On se propose d'écrire un programme qui énumère tous les n-uplets possibles que l'on peut écrire avec les entiers $1, 2, \ldots, n$. On sait qu'il y en a n^n, on ne fera donc exécuter ce programme que pour des valeurs raisonnables de n, donc

très petites ! Mais le problème n'est pas ici le fait que pour une valeur très raisonnable de n il y a beaucoup de solutions. On n'y peut rien ! Le problème est de savoir comment s'y prendre pour écrire un programme qui répond à la question quel que soit n. Pour le choix de la structure des données, il est assez naturel de choisir un tableau T de n éléments où seront rangés les nombres qui constituent le n-uplet. Mais comment remplir de toutes les façons possibles les n composantes de ce tableau ? Une première idée est d'écrire des boucles imbriquées dont les indices varient pour chacune de 1 à n. Cela veut dire que l'on écrirait autant de programmes différents, un pour chaque valeur de n. Pour n égal à trois on écrit trois boucles imbriquées, pour n égal à quatre on écrit quatre boucles imbriquées, etc ... En procédant ainsi on n'a pas écrit un programme général. C'est le mécanisme de la récursivité qui va ici nous permettre de simuler n boucles imbiquées. En procédant ainsi on va parcourir l'arbre de tous les choix possibles pour obtenir le long de chaque branche un n-uplet.
On suppose que le tableau T est une variable globale et qu'il est indéxé à partir de 0.

Algorithme 1 (Enumération de tous les n-uplets)**.**
Entrée : Un nombre entier naturel n.
Sortie : L'affichage de tous les n-uplets possibles sur 1, 2,..., n.

```
Fonction parcours (j)
| si j = n alors
|    | affiche (T)
| sinon
|    | pour i variant de 1 jusqu'à n faire
|    |    | T[j] ← i
|    |    | parcours(j+1)
```

```
*** Programme principal ***
| n ← un nombre entier naturel
| T ← un tableau de taille n
| parcours(0)
```

La variable i et le paramètre j sont locals à la fonction parcours. A chaque niveau d'imbrication la fonction parcours place dans la composante j du tableau T la valeur de i. Après s'être imbiqué n fois, les n composantes du tableau T sont remplies, on les affiche. A chaque retour, le programme récupère les valeurs des variables locales i et j. La variable i de contrôle de la boucle est incrémentée, puis soit le corps de la boucle est exécuté en rangeant i dans la composante j du tableau T, suivi de l'appel $parcours(j+1)$, soit on sort la boucle et on quitte la

fonction en se retrouvant après l'appel de parcours(j+1) au niveau de l'imbrication précédente.

2 Un deuxième exemple, énumération des permutations

Bien souvent il est possible d'éviter de parcourir l'arbre de toute les possibilités et ainsi trouver plus rapidement la solution. Pour cela on fait en sorte que le début de la solution en construction soit compatible avec la solution recherchée. Pour illustrer cela, on se propose d'écrire un programme qui énumère l'ensemble des permutations sur les entiers $1, 2, \ldots, n$. On peut envisager d'utiliser le programme précédent en vérifiant que le n-uplet construit contient des composantes différentes deux à deux avant de l'afficher. L'inconvénient est double, on énumère beaucoup trop de n-uplets qui seront pour la plupart rejetés et pour chaque n-uplet engendré la vérification pour savoir si l'on a une permutation ou pas prend du temps. On peut procéder différement en faisant en sorte que le n-uplet en construction soit tel que les composantes remplies soient différentes deux à deux. Pour cela on peut procéder de différentes manière. Une première façon de procéder consisterait, avant de placer i dans la composante j du tableau T, de vérifier que i est différent des j premières composantes déjà remplies. Cette façon est encore trop coûteuse en temps. Un seconde façon de procéder est d'utiliser un tableau $libre$ de n booléens tel que $libre[i]$ $vrai$ si et seulement si l'entier i n'a pas encore été placé dans le tableau T. On obtient l'algorithme suivant.

Algorithme 2 (Enumération de toutes les permutations)**.**
Entrée : Un nombre entier naturel n.
Sortie : L'affichage de toutes les permutations possibles sur 1, 2, ..., n.

```
Fonction parcours (j)
| si j = n alors
|     | affiche (T)
| sinon
|     | pour i variant de 1 jusqu'à n faire
|     |     | si libre[i] alors
|     |     |     | T[j] ← i
|     |     |     | libre[i] ← False
|     |     |     | parcours(j+1)
|     |     |     | libre[i] ← True
```

*** Programme principal ***

$n \leftarrow$ un nombre entier naturel
$T \leftarrow$un tableau de taille n
$libre \leftarrow$un tableau de taille n initialisé avec **True**
parcours(0)

Le tableau $libre$, variable globale comme le tableau T, permet à chaque imbrication, en ne faisant qu'un seul test de savoir si i peut être rangé dans la composante j du tableau T ou pas. Si oui, i est rangé et $libre[i]$ reçoit la valeur **False** avant de s'imbriquer à nouveau. Au retour on repermet l'utilisation de i en affectant à $libre[i]$ la valeur **True** avant de l'incrémenter. A peu de frais on réussi à élaguer l'arbre des choix pour obtenir beaucoup plus rapidement ce que l'on recherche.

3 Exemple des huit reines sur un échiquier

Le problème est de placer huit reines sur un échiquier sans qu'elles puissent se manger entre elles. On rappelle qu'une reine peut prendre un pièce si elle se trouve soit sur la même ligne, soit sur la même colonne, soit sur une des diagonales passant par la case où elle se trouve. Ce problème ne peut être résolu à priori qu'en examinant tous les placements possibles. Le nombre de façon de poser huit reines sur un échiquier est égal à $64\times63\times62\times61\times60\times59\times58\times57 = 178462987637760$, soit de l'ordre de 10^{15} possibilités. Une année est équivalant à 31536000 secondes, soit de l'ordre de 3×10^7 secondes. Si l'on suppose que le programme effectue un placement de huit reines et les tests nécessaires pour accepter ou refuser la configuration en dix nanosecondes, le programme mettrait 10^7 secondes pour examiner toutes les configurations, soit une année environ ! On peut facilement réduire le nombre de cas à envisager en remarquant qu'il suffit d'essayer de placer une reine dans chaque colonne. Ainsi pour chaque colonne on aura huit choix possibles, soit au total $8^8 = 2^{24}$, soit environ seize millions de possibilités. Ce nombre de possibilités peut encore être réduit en faisant en sorte qu'à chaque étape les reines placées vérifient la contrainte. Ici encore, la récursivité va nous permettre de simuler huit boucles imbriquées. Chaque boucle s'efforce de placer une reine dans une colonne en tenant compte des reines déjà en place dans les colonnes précédentes, avant de passer à la boucle qui concerne la colonne suivante. Pour placer une reine dans la colonne j en ligne i, il faut s'assurer qu'il n'y a pas de reine déjà placée sur la ligne i, ni sur les deux diagonales qui passent par la case (i, j). Pour eviter de parcourir la ligne et les deux diagonales on va utiliser trois tableaux de booléens qui nous permettront de décider si le placement est possible ou pas en ne faisant que trois tests. Un tableau L de huit booléens nous indiquera si une ligne comporte déjà une reine ou pas. Toutes les cases d'une diagonale ascendante sont telles que la somme des indice ligne plus indice colonne est constante. Le tableau D1 de booléens indexé de 0 à 14 nous permettra de savoir à l'aide d'un seul test

si la diagonale ascendante passant par une case comporte déjà une reine ou pas. Pour les diagonales descendantes c'est la différence des indices ligne moins colonne qui est constante. Cette différence varie entre -7 et +7. Le tableau D2 nous permettra à l'aide d'un seul test de savoir si la diagonale descendante passant par une case est libre ou pas.

3.1 Le programme Python des huit reines sur un échiquier

```
# Chaque liste affichée indique une solution
# Le rang dans la liste donne la colonne et la valeur indique la ligne
# Ici encore la structure du programme est analogue à celles des programmes
# qui génèrent les n_uplets ou les permutations.
# Le principe est le suivant: on va essayer de placer une reine dans chacune
# des 8 colonnes, pour chaque colonne on a 8 choix possibles. A chaque placement
# d'une nouvelle reine on doit vérifier que la solution partielle est admissible.
# Cela se fait en 3 tests seulement (L[i]and D1[i+j] and D2[i-j+7]).
# La ligne i est libre, la diagonale ascendante i+j est libre ainsi que
# la diagonale descendante de numéro i-j+7

def parcours(j):
    if j==8:
        print sol
    else:
        for i in range(8):
            if L[i]and D1[i+j] and D2[i-j+7]:
                sol[j]=i        # on place la reine en (i,j)
                L[i]=False      # On interdit la ligne i
                D1[i+j]=False   # On interdit diagonale D1 qui passe par (i,j)
                D2[i-j+7]=False # On interdit diagonale D2 qui passe par (i,j)
                parcours(j+1)   # On passe à la colonne suivante
                D2[i-j+7]=True  # Au retour, on enlève la reine que l'on avait
                D1[i+j]=True    # placée en repermettant d'utiliser à nouveau la
                L[i]=True       # ligne et les diagonales qui passent par (i,j).


# Programme principal

L=[True]*8   # pour indiquer les lignes libres,celles où il n'y a pas de reine
D1=[True]*15 # pour indiquer les diagonales ascendantes libres (i+j=cst)
D2=[True]*15 # pour indiquer les diagonales descendantes libres (i-j=cst)
sol=[0]*8    # pour ranger la solution en construction
parcours(0)  # déclanche le parcours de l'arbre de toutes les possibilités

# Voici le début de l'affichage des solutions

[0, 4, 7, 5, 2, 6, 1, 3]
[0, 5, 7, 2, 6, 3, 1, 4]
[0, 6, 3, 5, 7, 1, 4, 2]
[0, 6, 4, 7, 1, 3, 5, 2]
[1, 3, 5, 7, 2, 0, 6, 4]
[1, 4, 6, 0, 2, 7, 5, 3]
[1, 4, 6, 3, 0, 7, 5, 2]
[1, 5, 0, 6, 3, 7, 2, 4]
[1, 5, 7, 2, 0, 3, 6, 4]
[1, 6, 2, 5, 7, 4, 0, 3]
```

4 Un autre exemple, le cavalier d'Euler

Le problème est de savoir si un cavalier du jeu d'échec peut parcourir toutes les cases de l'échiquier en passant qu'une seule fois par chaque case. C'est en fait un problème de théorie des graphes en associant à chaque case un sommet du graphe. On a donc un graphe ayant 64 sommets reliés par des arêtes, deux sommets étant reliés par une arête si et seulement si il est possible de passer d'une case à l'autre case par un déplacement de cavalier. Le problème pour le cavalier d'Euler revient à chercher un chemin hamiltonien et non pas eulerien comme il est parfois écrit. Un chemin hamiltonien d'un graphe est un chemin tel que l'on passe par tous les sommets sans jamais passer plus d'une seule fois par chacun d'eux. Alors qu'un chemin eulerien d'un graphe est tel que l'on parcourt toutes les arêtes une et une seule fois. Ici encore le programme va explorer l'arbre des possibilités. Pour chaque case de l'échiquier le programme choisira successivement les huit directions possibles pour le déplacement du cavalier. Pour chaque choix il se rendra à la case choisie en y inscrivant le rang du déplacement dans le parcours et examinera à nouveau à partir de cette case les huit choix possibles et ainsi de suite jusqu'à être bloqué. Lorsque il se trouve bloqué le programme retourne en arrière et fait un autre choix pour le cavalier avant de repartir de l'avant et ceci jusqu'à ce que une solution soit trouvée ou que toutes les possibilités aient été examinées. Le tableau dir de deux lignes et huit colonnes nous permet de décrire les huit directions. A chaque colonne correspond une direction, on y trouve les incréments à faire sur l'indice ligne et sur l'indice colonne pour faire ce déplacement. Il est possible dans certains cas que l'on sorte de l'échiquier. Pour simplifier et éviter des tests coûteux on borde l'échiquier de deux rangées de cases remplies avec -1, la grille qui représente l'échiquier étant initialisée avec des 0. Ainsi, lors du parcours des huit directions possibles, soit la case choisie contient 0 alors on s'y rend, soit elle contient -1 alors on est en dehors de la grille et on choisi une autre direction, soit elle contient un nombre positif qui indique que l'on est déjà passé par là, on choisi donc une autre direction.
Là encore on obtient un programme qui a une structure tout à fait analogue aux programmes précédants. Mais ici la combinatoire ne nous permet pas d'obtenir une solution pour une grille 8x8. En effet pour une dimension de 8, le programme tourne sans afficher de solution alors qu'on est sûr qu'il en existe. Comme on va essayer de s'en convaincre ce programme risque de tourner pendant plusieurs années avant d'afficher une solution ! Pour un échiquier 5x5 il y a une solution quelle que soit la case de départ, mais le temps varie suivant la case de départ. En partant de (1,1) la solution est trouvée en $\frac{1}{10}$seconde, mais en partant de (1,2) il

faut attendre 7.7 secondes pour voir s'afficher la solution. Pour un échiquier 6x6, qui admet aussi une solution quelle que soit la case de départ, l'écart de temps est encore plus important. Pour la case (1,2) on attends $\frac{4}{10}$ de seconde, mais pour la case (2,2) il faut attendre plus de 35 minutes pour voir la solution ! Pour un échiquier 7x7 on ne voit apparaître la solution dans un temps raisonnable que pour quelques cases de départ. On peut avoir une idée de la combinatoire du problème en cherchant à avoir une idée du nombre de noeuds de l'arbre de toutes les possibilités. Par exemple pour n valant 5, on trouve que le nombre moyen de choix par case est de 3.8 coups possibles. Il y a 25 cases, le nombre de feuilles dans l'arbre des choix est donc de 3.8^{25}, soit de l'ordre de 10^{14} feuilles. Ceci est une borne, en réalité le programme en parcourt beaucoup moins, ce qui permet de visualiser toutes les solutions. On appelle feuille un noeud terminal, le nombre de feuilles est identique au nombre de branches. Une branche correspond en fait à un parcours du cavalier. Pour n valant 6 le nombre de feuilles est majoré par 4.4^{36} soit de l'ordre de 10^{23} feuilles, soit 10^8 fois plus que dans le cas n égal à 5. Ceci explique qu'il faille attendre parfois plusieurs minutes avant de voir apparaître la solution. Pour n valant 7 la combinatoire est encore plus importante car le nombre de branches est cette fois majoré par $4.89^{49} \approx 10^{33}$ branches, soit de l'ordre de 10^{10} de plus que pour n égal à 6. Ceci explique que le programme ne trouve que quelques solutions dans un temps supportable. Pour les autres cases, on ne sait pas s'il y a une solution. Comme on s'y attend la combinatoire pour n valant 8 nous indique qu'il est illusoire d'obtenir une solution par ce programme. En effet le nombre de branches dans ce cas est majoré par $5.25^{64} \approx 10^{47}$ soit de l'ordre de 10^{12} de plus que pour n égal à 7. Ceci nous indique qu'il est probable que le temps d'attente soit multiplié peut-être pas par 10^{12}, mais par un nombre conséquent qui nous laisse penser que le programme risque de tourner pendant plusieurs années, voire plusieurs siècles avant d'afficher une solution. On est pourtant sûr qu'il y a des solutions pour un échiquier 8x8 (voir le programme Quick cavalier).

4.1 Le programme Python du cavalier d'Euler

```
def affiche(m):    # Affiche l'échiquier en ignorant les 2 rangées qui l'entourent.
    global taille
    for i in xrange(2,taille+2):
        for j in xrange(2,taille+2):
            print '%3d'% m[i][j],
        print

def parcours(k):
    global x,y,grille,taille,dir
    if k>taille*taille:  # Une solution a été trouvée
        affiche (grille) # On l'affiche
        exit()           # On arrête tout, car d'autres solutions
                         # ne nous intéressent pas, d'autant que
                         # si l'on demande au programme de toutes les afficher
```

```
                        # cela risquerait de prendre un temps d'exécution
                        # absolument insupportable!
    else:
        for d in xrange(8): # on parcourt les 8 déplacements possibles
            if grille[x+dir[0][d]][y+dir[1][d]]==0 :# si la case est libre
                x=x+dir[0][d]                       # on y va
                y=y+dir[1][d]
                grille[x][y]=k  # On place le cavalier

                parcours(k+1)   # on passe à l'étape suivante

                grille[x][y]=0  # Au retour, après avoir essayé tous les
                x=x-dir[0][d]   # choix possibles pour les étapes suivantes
                y=y-dir[1][d]   # On enlève le cavalier, on défait le dernier
                                # déplacement effectué à l'étape k pour en
                                # choisir un autre en incrémentant d.

# Programme principal

# Le tableau dir indique les incréments à faire sur l'indice ligne et sur
# l'indice colonne pour déplacer le cavalier dans une des huit directions.
# La première indique l'incrément sur l'indice ligne et la seconde
# l'incrément sur l'indice colonne.
dir=[[-1,-2,-2,-1,1,2,2,1],[-2,-1,1,2,2,1,-1,-2]]

taille=input("Donner la taille de l'échiquier: ")
grille=[]

# Pour éviter, lors d'un déplacement, le test qui permet
# de savoir si l'on est sorti ou pas de l'échiquier, on a choisi
# de border tout le pourtour de l'échiquier par 2 rangées de cases
# remplies avec la valeur -1. Ainsi il n'y aura pas d'erreur à l'éxecution.
# A l'intérieur de l'échiquier, 0 indiquera une case non encore parcourue,
# un nombre positif une case par laquelle le cavalier est déjà passé.
# La case de départ contiendra 1, la case correspondant au premier
# déplacement contiendra 2, la case suivante 3, ..., la dernière case
# atteinte n*n, lorsqu'une solution est trouvée.

# On commence par placer des -1 dans toute la grille de dimension taille+4
for i in range(taille+4):
    grille.append([-1]*(taille+4))

# On place des 0 dans la partie centrale de la grille, les 2 rangées
# qui entourent l'échiquier contiennent les -1
for i in xrange(2,taille+2):
    for j in xrange(2,taille+2):
        grille[i][j]=0;

# Lecture des coordonnées de la case de départ
print "Donner des nombres compris entre 1 et ", taille
xd=input("donner x  ")
yd=input("donner y  ")
x=xd+1
y=yd+1
k=1
grille[x][y]=1
parcours(2)
```

Lorsque la complexité combinatoire indique qu'il est vain d'espérer une solution à un problème dans un temps raisonnable, on essaye de le résoudre ou d'approcher la solution optimale par d'autres moyens. L'algorithme suivant en est l'illustration. Il se contente d'appliquer l'heuristique suivante pour chaque déplacement : Parmi toutes les cases où le cavalier peut se rendre en un seul déplacement, il choisira la case qui offre le moins de possibilité de fuite.

4.2 Le programme Python du Quick cavalier d'Euler

```
# Ce programme n'est pas récursif. On utilise l'heuristique suivante :
# Le cavalier se trouvant en (x,y), parmi toutes les cases où
# il peut se rendre en un seul déplacement, il choisira la case
# qui offre le moins de possibilité de fuite.
# Cette heuristique marche sans que l'on sache pourquoi! Elle permet de
# résoudre le problème pour un échiquier 8x8, là où le programme récursif
# parcourant l'arbre de toutes les possibilités échoue.

def affiche(m):
    global taille
    for i in xrange(2,taille+2):
        for j in xrange(2,taille+2):
            print  '%3d' % m[i][j],
        print

# Programme principal

dir=[[-1,-2,-2,-1,1,2,2,1],[-2,-1,1,2,2,1,-1,-2]]
taille=input("Donner la taille de l'échiquier: ")
grille=[]

# On place des -1 dans toute la grille
for i in range(taille+4):
    grille.append([-1]*(taille+4))

# On place des 0 dans la partie centrale de la grille, les 2 rangées
# qui entourent l'échiquier contiennent les -1
for i in xrange(2,taille+2):
    for j in xrange(2,taille+2):
        grille[i][j]=0;

# Lecture des coordonnées de la case de départ
print "Donner des nombres compris entre 1 et ", taille
xd=input("donner x  ")
yd=input("donner y  ")
x=xd+1
y=yd+1
k=1
grille[x][y]=1
while k<taille*taille:
    min=8
    dmin=9
    for d in xrange(8):
        a=x+dir[0][d]
        b=y+dir[1][d]
        if grille[a][b]==0:
```

```
            c=0
            for z in xrange(8):
                if grille [a+dir[0][z]][b+dir[1][z]]==0:
                    c=c+1
            if c<=min:
                min=c
                dmin=d
    if dmin==9: # à partir de la case(x,y), le cavalier est bloqué.
        print "Pas de solution"
        exit()
    x=x+dir[0][dmin]
    y=y+dir[1][dmin]
    k=k+1
    grille[x][y]=k

affiche(grille)
```

5 Un autre exemple, le jeu de Sudoku

Le jeu de Sudoku est composé d'une grille carrée de neuf cases de coté, subdivisée en neuf carrés identiques, qu'on appellera pavés. La règle du jeu est simple : chaque ligne, colonne et pavé doit contenir tous les chiffres de un à neuf. Le jeu consiste à compléter une grille préremplie en respectant les règles. L'intérêt du jeu réside dans la simplicité des règles, et dans la complexité de la solution. Les grilles publiées dans les magazines possèdent un niveau de difficulté indicatif. En général les grilles comportant un grand nombres de cases préremplies sont les plus simples, mais l'inverse n'est pas systématiquement vrai. Le nombre de grilles respectant les critères énoncés est égal à :
$9! \times 72^2 \times 2^7 \times 27704267971 \approx 6,67 \times 10^{21}$
Ce résultat a été prouvé en 2005 grâce à une recherche exhaustive. Si l'on tient compte des symétries il a été établi qu'il n'y a que 5 472 730 538 grilles possibles !
On suppose que les cases de la grille sont numérotées de 0 à 80 de haut en bas et de gauche à droite. La case numéro 0 est la case de coordonnées (0,0), la case numéro 1 est la case de coordonnées (1,0),..., la case numéro 80 est la case de coordonnées (8,8). De même les neuf pavés sont numérotés de 0 à 8 : Le pavé numéro 0 est le pavé en haut à gauche, le pavé numéro 1 est celui juste en dessous, le pavé numéro 8 est celui en bas à droite.
Quelques relations utiles pour la suite :
Si $prof$ est le numéro d'une case de la grille, on peut exprimer les coordonnées (i, j) de la case numéro $prof$ en fonction de $prof$. On a les relations $i = prof\ \%\ 9$ et $j = prof/9$, où les opérateurs % et / désignent respectivement le reste et le quotient de la division euclidienne de deux nombres entiers.
On peut également exprimer le numéro p du pavé contenant la case de coordonnées (i, j), on a la relation suivante $p = 3 * (j/3) + i/3$.

L'algorithme consiste à explorer toutes les possibilités en programmant un algorithme à essais successifs comme celui qui permet de placer 8 reines sur un échiquier. En résumé pour chaque case, on envisagera successivement les 9 choix possibles (les nombres de 1 à 9) et pour chaque choix, s'il est admissible, on fera ce choix en laissant les autres possibilités en suspens, et ainsi de suite jusqu'à parcourir les 81 cases de la grille (on affiche alors la solution trouvée) où à être bloqué. Lorsque l'algorithme atteint une situation de bloquage sur une case, il doit revenir en arrière ("backtracking") à la case précédente pour faire un autre choix laissé en suspens et repartir de l'avant.

Pour trancher rapidement la question de savoir si le nombre k peut-être placé en (i, j), on utilise trois tableaux $pave[9][10]$, $ligne[9][10]$, $col[9][10]$, qui ont la signification suivante :
- $pave[i][k]$ est égal à True si et seulement si le nombre k est déjà placé dans le pavé n°i.
- $ligne[i][k]$ est égal à True si et seulement si le nombre k est déjà placé dans la ligne n°i.
- $col[i][k]$ est égal à True si et seulement si le nombre k est déjà placé dans la colonne n°i.
Ces trois tableaux sont initialisés avec la valeur False. Par ailleurs les tableaux $grille[9][9]$ et $enonce[9][9]$ représentent respectivement la grille du sudoku et l'énoncé. On convient que la valeur 0 indique qu'une case n'est pas encore remplie.

Le tableau enonce pourra être initialisé lors de sa déclaration, en rentrant les 81 valeurs ligne par ligne. Il faudra vérifier qu'il n'y a pas d'erreur de remplissage et mettre à jour les 3 tableaux de booléens $pave$, $ligne$ et col.

5.1 Le programme Python du Sudoku

```
# les pavés sont numérotés de 0 à 8 ainsi que les lignes et les colonnes.
# Pave[i][j] vrai ssi j est présent dans le pavé n°i.
# ligne[i][j] vrai ssi j est présent dans la ligne n° i.
# col[i][j] vrai ssi j est présent dans la colonne n° i.

# possible renvoie vrai ssi on peut placer k dans la case de coordonnées (i,j)
def possible (i,j,k):
    global pave,ligne,col
    p=3*(j/3)+i/3;
    return not pave[p][k] and not ligne[i][k] and not col[j][k];

enonce=[
        [7,9,0,0,4,0,0,0,0],
        [0,0,4,0,1,0,8,7,0],
        [0,0,0,0,0,2,0,6,0],
        [0,5,6,0,0,1,0,0,3],
        [0,0,0,0,5,0,0,0,0],
        [3,0,0,8,0,0,7,1,0],
```

```
        [0,8,0,2,0,0,0,0,0],
        [0,3,5,0,8,0,1,0,0],
        [0,0,0,0,6,0,0,5,8]
       ]
# L'énoncé est rentré en donnant la suite des valeurs lors de la
# déclaration. La procédure ci-dessous doit vérifier qu'il n'y a pas de conflit et
# mettre à jour les trois tableaux de booléens pour traduire les contraintes.

def verifie_enonce():
    global pave,ligne,col,enonce
    for i in xrange(9):
        for j in xrange(9):
            if enonce[i][j]!=0:
                k=enonce[i][j]
                p=3*(j/3)+i/3
                if possible(i,j,k):
                    pave[p][k]=True
                    ligne[i][k]=True
                    col[j][k]=True
                else:
                    print"Erreur de remplissage "
                    break


def initialisations():
    global pave,ligne,col,grille,enonce
    for i in xrange(9):
        pave.append([False]*10)
        ligne.append([False]*10)
        col.append([False]*10)
        grille.append([0]*9)
    for i in xrange(9):
        for j in xrange(9):
            grille[i][j]=enonce[i][j]
    verifie_enonce()




def affiche(m):
    for i in xrange(9):
        for j in xrange(9):
            print m[i][j],
        print

def parcours(prof):
    global pave,ligne,col,grille
    if prof==81:
        print "voici la solution"
        affiche(grille)
        exit()
    else:
        i=prof % 9
        j=prof / 9
        p=3*(j/3) + i/3
        if grille[i][j]!=0:
            parcours(prof+1)
        else:
            for k in range(1,10):
```

```
                if possible(i,j,k):
                    grille[i][j]=k
                    pave[p][k]=True
                    ligne[i][k]=True
                    col[j][k]=True
                    parcours(prof+1)
                    col[j][k]=False
                    ligne[i][k]=False
                    pave[p][k]=False
                    grille[i][j]=0

# Programme principal

pave=[]
ligne=[]
col=[]
grille=[]
initialisations()
affiche(grille)
print
parcours(0)
```

6 La programmation des jeux à deux joueurs

La machine est le joueur noté PLUS. Les noeuds de l'arbre des coups joués sont alternativement des noeuds OU et des noeuds ET. La racine, correspondant aux premiers coups possibles pour l'ordinateur est un noeud OU.

Algorithme 3 (Recherche du meilleur coup possible)**.**
Entrée :
Sortie : Un nombre entier, évaluation du meilleur coup à jouer
Fonction parcours()
global $trait, valprof, pmax, mcp$

```
si prof = pmax alors
    resultat évaluation de la configuration atteinte
sinon
    si trait=PLUS alors
        val[prof] ← intMin
    sinon
        val[prof] ← intMax
    pour tous les coups jouables i faire
        jouer(i)
        echanger trait et ntrait
        prof ← prof + 1
        v ← parcours()
        prof ← prof − 1
        echanger trait et ntrait
        déjouer(i)
        si trait = PLUS alors On prend le max
            si v > val[prof] alors
                si prof = 0 alors
                mcp ← i On retient le Meilleur Coup Possible
                val[prof] ← v
        sinon  On prend le min
            si v < val[prof] alors
                val[prof] ← v
    resultat val[prof]
```