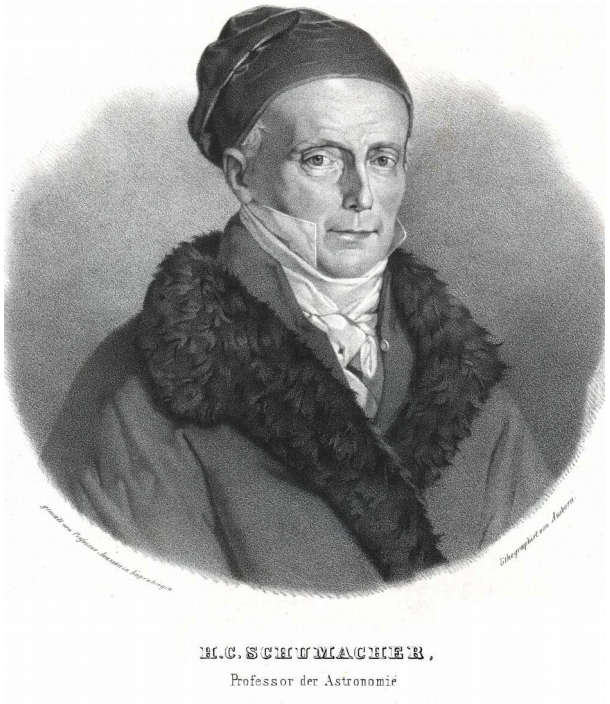
Backtracking...

Logique et approche mathématique de la programmation, cours 4
M. Rigo

Backtracking...



Problème des 8 reines attribué à Max Bezzel (1848)

François-Joseph Eustache Lionnet (1869)
propose de généraliser le problème à un échiquier *n x n*

Lettre de Gauss à Heinrich Schumacher (1850) :

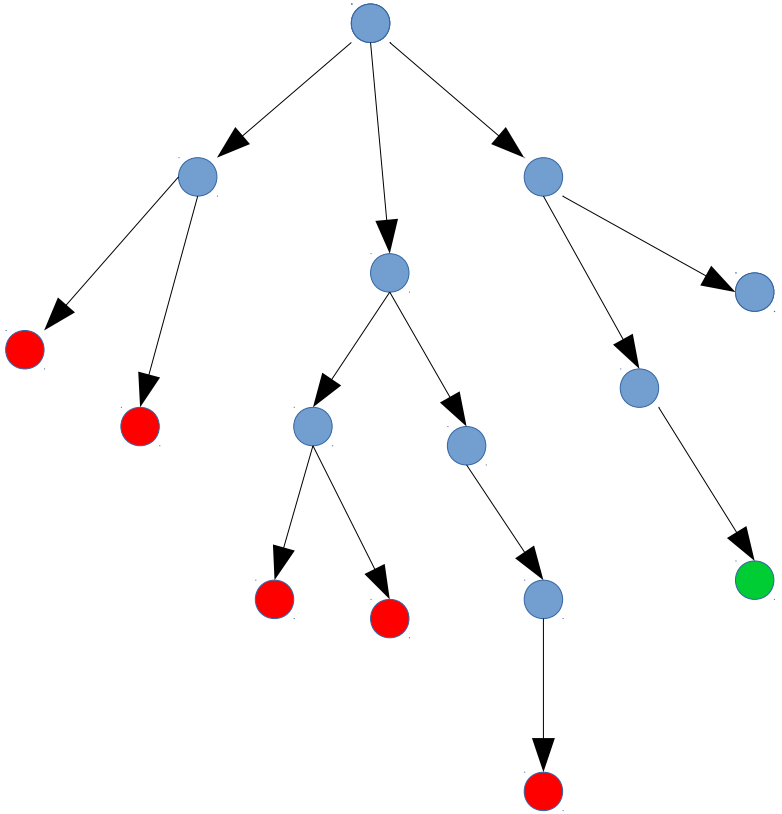
"*Schwer ist es übrigens nicht, durch ein methodisches Tatonniren sich diese Gewissheit zu verschaffen, wenn man oder ein paar Stunden daran wenden will.*"

Gauss propose alors une méthode récursive de résolution, reprise par Edouard Lucas (1882)

On place méthodiquement une reine par colonne, de la gauche vers la droite :

- Pour une colonne donnée, si une case est sous l'emprise d'une reine placée précédemment, passer à la case suivante (méthodiquement, de haut en bas) ;
- Sinon, placer une reine sur cette case et continuer récursivement avec le reste de l'échiquier
- S'il n'y a pas de case disponible dans la colonne, revenir à la dernière configuration qui peut encore être "incrémentée".

Principe du backtracking



Espace d'états à visiter

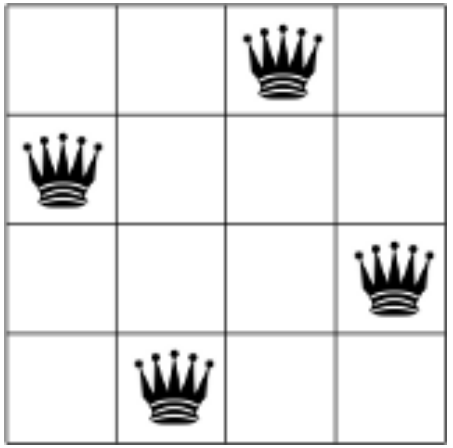
Si un état ne mène qu'à des échecs, revenir à son ancêtre direct et tester une alternative

Répéter récursivement

On peut vouloir :

- Trouver une "solution"
- Enumérer toutes les "solutions"
- Trouver une solution "optimale"

Il s'agit de variantes d'une même question...



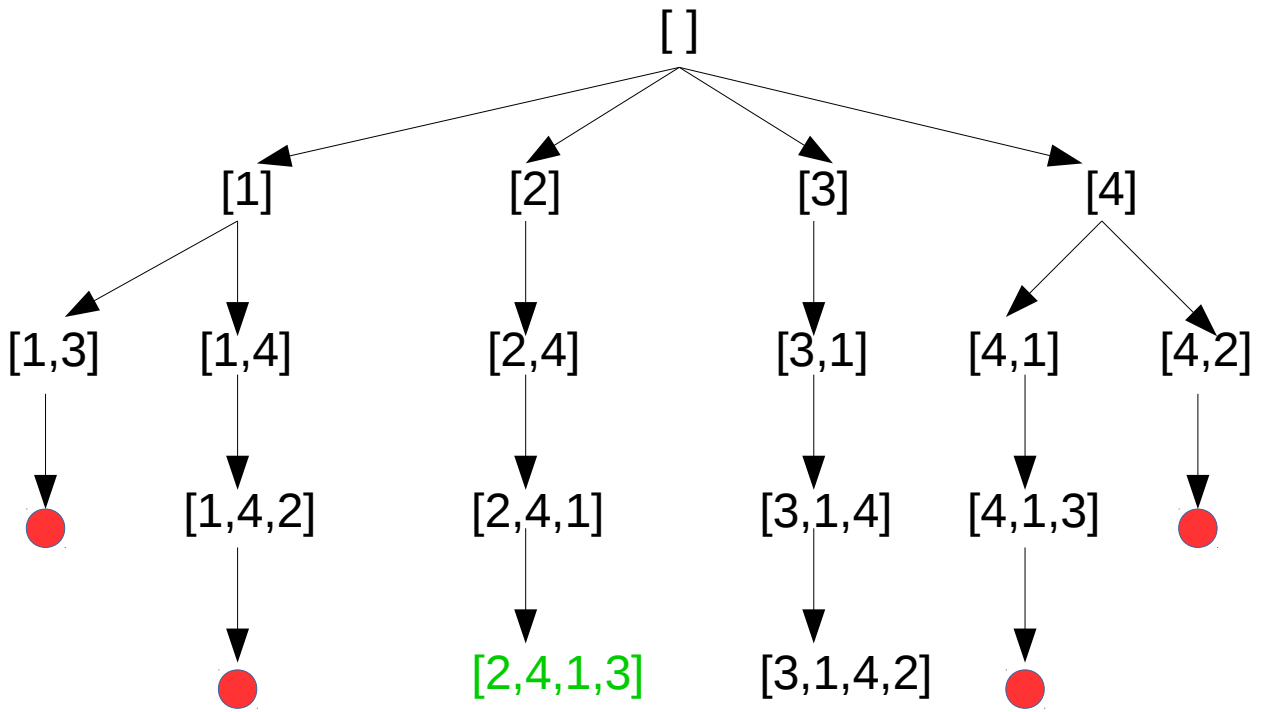
Commençons avec le problème des *4 reines*

Gardons en tête, la possible généralisation à *n* reines

Une configuration est codée par une liste ex. [2,4,1,3]

Si on dispose d'une solution partielle valide
→ liste de longueur *k*

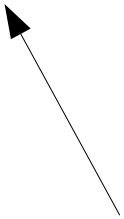
Alors on essaie de compléter la *(k+1)e* colonne



```python
# définition d'une constante, la taille de l'échiquier
n=4

# partant d'une liste de longueur k,
# renvoie True si on peut compléter celle-ci
# par une reine en position new dans la colonne k+1

def ajoutValide(liste,new):
    sortie=True
    for i in range(0,len(liste)):
        if liste[i]==new or liste[i]==new-len(liste)+i or liste[i]==new+len(liste)-i:
            sortie=False
    return sortie
```

Remarque : on aurait pu aussi écrire une fonction testant si une liste correspondait à une configuration valide de l'échiquier

```
# partant d'une liste,
# on passe en revue toutes les solutions possibles

def nouvelleReine(liste):
    # si on a une liste de longueur n, c'est une solution
    if len(liste)==n:
        print(liste)
    else:
        # sinon, on essaie d'étendre de toutes les façons possibles
        for j in range(1,n+1):
            if ajoutValide(liste,j):
                nouvelleReine(liste+[j]) # récursif

nouvelleReine([])
```



On initialise avec une liste vide

Que fait le programme ?

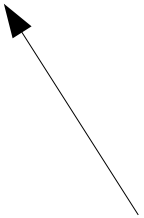
```
# partant d'une liste,
# on passe en revue toutes les solutions possibles

def nouvelleReine(liste):
    print(liste) # on verra l'évolution
    # si on a une liste de longueur n, c'est une solution
    if len(liste)==n:
        print(liste)
    else:
        # sinon, on essaie d'étendre de toutes les façons possibles
        for j in range(1,n+1):
            if ajoutValide(liste,j):
                nouvelleReine(liste+[j]) # récursif

nouvelleReine([])
```

Sortie :

```
[ ]
[1]
[1, 3]
[1, 4]
[1, 4, 2]
[2]
[2, 4]
[2, 4, 1]
[2, 4, 1, 3]
[2, 4, 1, 3]
[3]
[3, 1]
[3, 1, 4]
[3, 1, 4, 2]
[3, 1, 4, 2]
[4]
[4, 1]
[4, 1, 3]
[4, 2]
```

Double affichage
Vu les 2 instructions “print”

```
nouvelleReine([2,4])
```



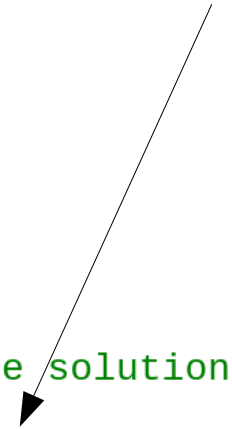
Si on initialise avec une liste non vide,
on recherche les solutions débutant par [2,4]

Sortie finale :
[2,4,1,3]

Si on ne cherche qu'une solution
(la première dans le parcours d'arbre)

```
# on définit un parcours dans l'arbre
# [X,i] -> [X,i+1] si i<n
# [X,n] -> le successeur de [X]
# par exemple, [2,n,n] -> [3]
def suivant(liste):
    if liste==[]:
        return [1]
    elif liste[-1]<n:
        return liste[:-1]+[liste[-1]+1]
    else:
        return suivant(liste[:-1])

# on explore l'arbre jusqu'à trouver une solution
config=[1]
while len(config)<n or not ajoutValide(config[:-1],config[-1]):
    # si on a une configuration valide, on descend d'un niveau
    if ajoutValide(config[:-1],config[-1]):
        config=config+[1]
    else:
        config=suivant(config)

print(config)
```

tester la validité
d'une configuration...



Évolution de *config*

[1]
[1, 1]
[1, 2]
[1, 3]
[1, 3, 1]
[1, 3, 2]
[1, 3, 3]
[1, 3, 4]
[1, 4]
[1, 4, 1]
[1, 4, 2]
[1, 4, 2, 1]
[1, 4, 2, 2]
[1, 4, 2, 3]
[1, 4, 2, 4]
[1, 4, 3]
[1, 4, 4]
[2]
[2, 1]
[2, 2]
[2, 3]
[2, 4]
[2, 4, 1]
[2, 4, 1, 1]
[2, 4, 1, 2]
[2, 4, 1, 3]

Si on démarrait par exemple avec

config[3]

On trouve : [3, 1, 7, 5, 8, 2, 4, 6]

Si on remplace *n=15*

On trouve : [1, 3, 5, 2, 10, 12, 14, 4, 13, 9, 6, 15, 7, 11, 8]

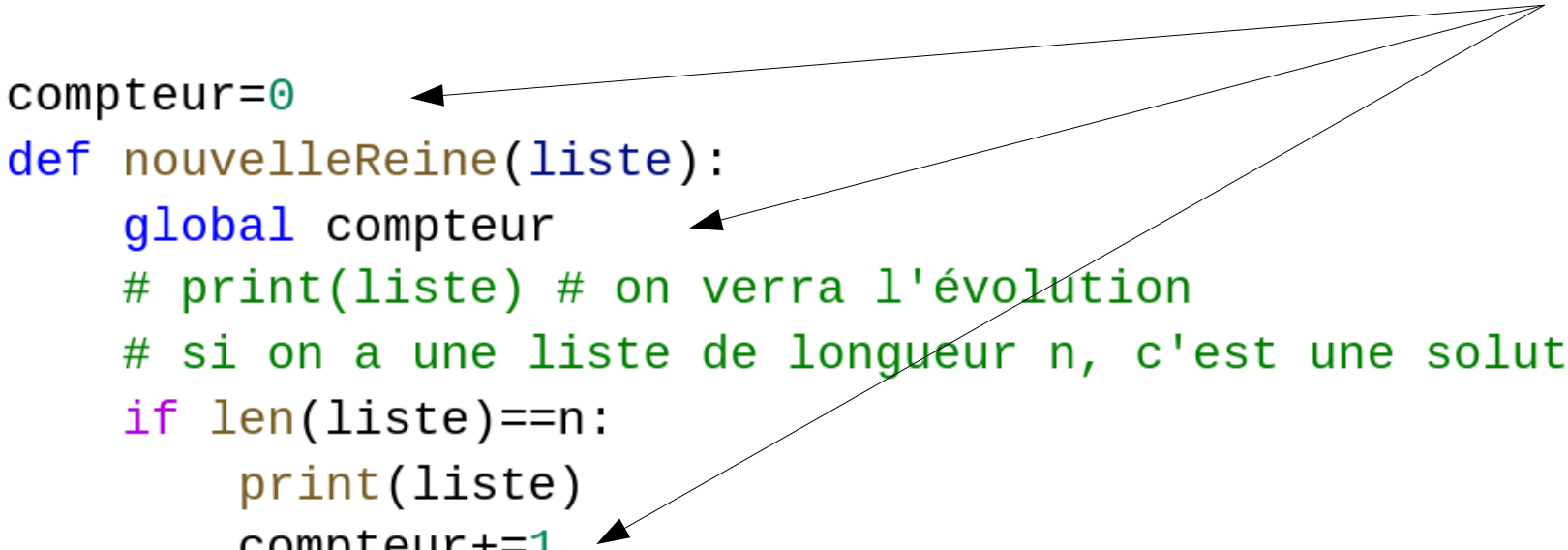
Et pour *n=20*

[1, 3, 5, 2, 4, 13, 15, 12, 18, 20, 17, 9, 16, 19, 8, 10, 7, 14, 6, 11]

On aurait pu aussi en faire une fonction (variante)

```
def premiereSolution():
# on explore l'arbre jusqu'à trouver une solution
    config=[1]
    while len(config)<n or not ajoutValide(config[:-1],config[-1]):
        # si on a une configuration valide, on descend d'un niveau
        if ajoutValide(config[:-1],config[-1]):
            config=config+[1]
        else:
            config=suivant(config)
    return(config)

print(premiereSolution())
```

Une autre variante : compter le nombre total de solutions ...



```
compteur=0
def nouvelleReine(liste):
    global compteur
    # print(liste) # on verra l'évolution
    # si on a une liste de longueur n, c'est une solution
    if len(liste)==n:
        print(liste)
        compteur+=1
    else:

# sinon, on essaie d'étendre de toutes les façons possibles
        for j in range(1,n+1):
            if ajoutValide(liste,j):
                nouvelleReine(liste+[j]) # récursif
```

Une autre variante : compter le nombre total de solutions …

[1, 5, 8, 6, 3, 7, 2, 4]
[1, 5, 8, 6, 3, 7, 2, 4]
[1, 6, 8, 3, 7, 4, 2, 5]
[1, 7, 4, 6, 8, 2, 5, 3]
[1, 7, 5, 8, 2, 4, 6, 3]
[2, 4, 6, 8, 3, 1, 7, 5]
[2, 5, 7, 1, 3, 8, 6, 4]
[2, 5, 7, 4, 1, 8, 6, 3]
…

[7, 4, 2, 5, 8, 1, 3, 6]
[7, 4, 2, 8, 6, 1, 3, 5]
[7, 5, 3, 1, 6, 8, 2, 4]
[8, 2, 4, 1, 7, 5, 3, 6]
[8, 2, 5, 3, 1, 7, 4, 6]
[8, 3, 1, 6, 2, 5, 7, 4]
[8, 4, 1, 3, 6, 2, 7, 5]

92

Le Backtracking peut s'appliquer
à des problèmes variés :

- Problème des reines
- Ballade d'un cavalier sur l'échiquier
- Sudoku
- Tectonik
- Coloriage de graphes
- ...