

PROJET D'INGÉNIERIE PEIP RÉCURSIVITÉ INFORMATIQUE

Sollier Baptiste, Montial Liam, Gagliano Adrien, Meylan Kénan

PEIP 1 - Année 2020 - 2021

Qu'est ce que la généricité informatique ?

Quelles sont les applications de la récursivité ?

Différents exemple de construction récursive

Qu'est ce que la récursivité ?

Avant même de s'intéresser aux construction récursive, intéressons nous tout d'abord a qu'est ce que la récursivité. D'après le CNRTL (Centre National de Ressources Textuelles et Lexicales), un processus récursif est un processus qui s'appelle ou se met en jeu répétitivement et automatiquement.

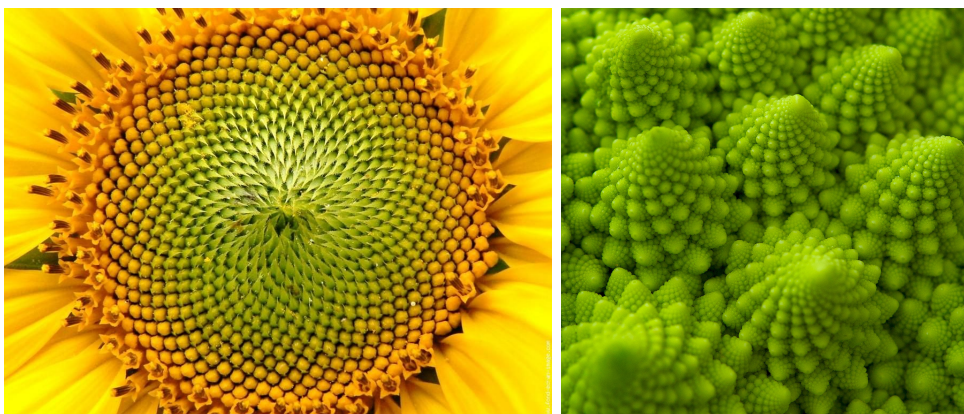
Des exemples de construction récursive

La récursivité est présente dans tous les domaines, que ce soit dans l'art par exemple, avec la mise en abime que nous pouvons voir dans la publicité par exemple, comme dans le logo de la Vache Qui Rit.



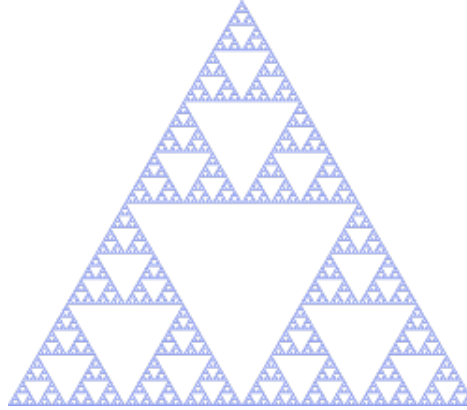
On peut bien voir que la vache est représenté aussi dans ses boucles d'oreilles qui, sur celle ci, a aussi des boucles d'oreilles sur laquelle est représenté la vaches, avec d'autre boucles d'oreilles et ainsi de suite.

La récursivité est aussi très présente dans le domaines de la biologie comme dans les fleurs de tournesol ou les choux romanesco :



On voit bien sur le chou romanesco par exemple, que sur chaque spirale, il y a d'autres spirales, elle même constitué de spirale et ainsi de suite.

Dans le domaine des mathématiques, l'exemple le plus connu de tous est évidemment la suite de Fibonacci qui est défini par : $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$. Ainsi, chaque nombre supérieur à 1 est défini par récursivité grâce au nombre précédent et au nombre précédent celui qui précède. Un autre exemple extrêmement connu de la récursivité, est les fractales. Voici le triangle de Sierpinski :



Ce triangle est généré en prenant un triangle plein initial, diviser sa taille par deux, l'accoler à deux autre triangle identique sommet a sommet pour former un autre triangle. Ainsi, chaque triangle, hormis le premier, est défini en fonction du précédent.

Intéressons nous maintenant au domaine qui nous intéresse, c'est à dire celui de l'informatique.

La récursivité dans l'informatique

Le cas de base en récursivité

En récursivité, le cas de base sert à la sortie du programme, sans celle ci, le programme continuera à l'infini, comme nous allons le voir dans l'exemple suivant, on retourne la valeur finale si $n = 2$. Pour illustrer nos propos, il suffit de regarder l'exemple de la factorielle. Pour des soucis de compréhension, une partie en python et en pseudo-code sera rédigé.

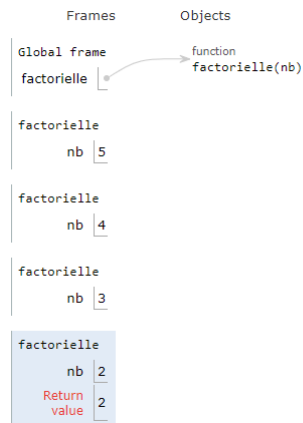
La factorielle par récursivité

```
1 def factorielle(nb):
2     if nb > 2:
3         return nb * factorielle(nb - 1)
4     return nb
5
6 Fonction factorielle(nb)
7     Si n > 2:
8         Retourner nb * factorielle(nb - 1)
9     Fin Si
10    Retourner nb
11 Fin Fonction
12 """
```

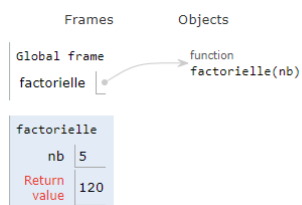
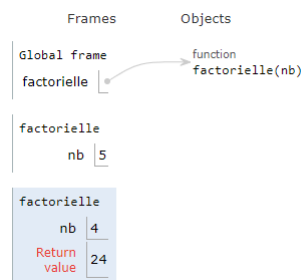
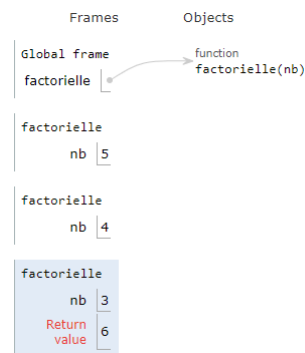
Calcul de factorielle de façon récursive

Nous savons que $0! = 1$, que $1! = 1$ et que $2! = 2$ donc, comme cas de base, nous avons : si le chiffre est inférieur à 2, nous retournons le nombre. Sinon, si le nombre est supérieur à 2, nous calculons ce nombre multiplié par la factorielle du nombre -1. En effet, la factorielle de n peut être défini de la façon suivant : $n! = n * (n - 1)!$. Ainsi, pour calculer $(n - 1)!$ cela revient à calculer $(n - 1)! = (n - 1) * (n - 2)!$ et ainsi de suite.

Nous pouvons visualiser l'exécution, pour une valeur de 5 par exemple :



On appelle la fonction jusqu'à ce que `nb` soit inférieur ou égale à 2. A 2, on retourne donc 2. On peut donc désormais calculer chaque retour de fonction lors desquelles on demandais au programme de retourner $nb * \text{factorielle}(nb - 1)$.



La Tour de Hanoï

```
1 def hanoi(n,a=1,b=2,c=3):
2     if (n > 0):
3         hanoi(n-1,a,c,b)
4         Afficher Déplace valeur de a sur valeur de c
5         hanoi(n-1,b,a,c)
6     Fin Si
7 Fin Fonction
8 """
9 Fonction hanoi(n, a = 1, b = 2, c = 3)
10     Si n > 0:
11         hanoi(n-1,a,c,b)
12         print "Déplace ",a,"sur",c
13         hanoi(n-1,b,a,c)
14     Fin si
15 Fin fonction
16 """
```

Calcul de factorielle de façon récursive

Comme dit dans l'énoncé, en informatique et en mathématiques, une fonction récursive est une fonction qui se définit en fonction d'elle même. Son analogue plus connu et plus utilisé est l'itératif, celui ci ne s'appelle pas lui même mais est constitué de boucles. Voyons donc ce parallèle :

Le parallèle itératif/récuratif

Qu'est ce qu'une itération ?

En programmation, un programme itératif est un programme qui utilise une boucle (boucle *for* le plus souvent). Une itération est une répétition, un tour de cette boucle. A chaque tour de boucle, on dit qu'il y a une nouvelle itération.

La factorielle de façon itérative

```
1 def factorial1(n):
2     r = 1
3     for i in range(1,n+1):
4         r = r * i
5     return r
6 print(factorial(5))
7
8 """
9 Fonction factorial1(n)
10     r = 1
11     Pour i de 1 jusqu'à n avec un pas de 1
12         r = r * i
13     Fin Pour
14     Retourner r
15 Fin Fonction
16 Afficher factorial de 5
17 """
```

Calcul de factorielle de façon itérative

Les avantages et inconvénient

On peut donc voir que la récursivité est donc plus courte en longueur de code. Elle est aussi un gain de temps dans la plupart des cas comme dans les algorithmes de tri.

Un algorithme de tri est aussi beaucoup plus rapide de façon récursive que de façon itérative, évidemment sur ce type d'algorithme (sur d'autres algorithmes l'itérativité serait plus rapide, mais pour généraliser, cela récursivité est très souvent un gain de temps). Avec deux algorithmes de tri, par exemples le tri par sélection et le tri rapide (en toute logique le tri rapide est le tri effectué par récursivité).

```
PS D:\Ce PC\Bureau\Projet_PEIP> python .\selection_sort.py
```

Nombre d'éléments	Temps de tri
10	1.10999999999986121e-05
100	0.00035709999999999991
1000	0.051243900000000001
10000	5.2001928
100000	602.39755020000001

```
PS D:\Ce PC\Bureau\Projet_PEIP> python .\rapid_sort.py
```

Nombre d'éléments	Temps de tri
10	1.57000000000007375e-05
100	0.000135899999999999436
1000	0.0014880000000000017
10000	0.0126193
100000	0.2224831

On peut donc voir que le code est extrêmement plus rapide par récursivité que par itérativité (soit environ un facteur de 2700 fois plus rapide par récursivité que par itérativité) Nous allons voir dans la prochaine partie, d'autres algorithmes utilisant la récursivité.

Un exemple d'application de la récursivité

Qu'est ce que le backtracking ?

Le backtracking, ou retour sur trace, ou encore retour arrière, est un type d'algorithme en testant récursivement l'ensemble des affectations possibles à une variable. Ce type d'algorithme consiste à sélectionner une variable du problème et pour chaque valeur valable de cette variable, tester récursivement si la solution est valide ou pas. Si aucune solution valide n'est trouvée, alors le programme abandonne la dernière valeur et revient sur les affectations de la variable faite précédemment.

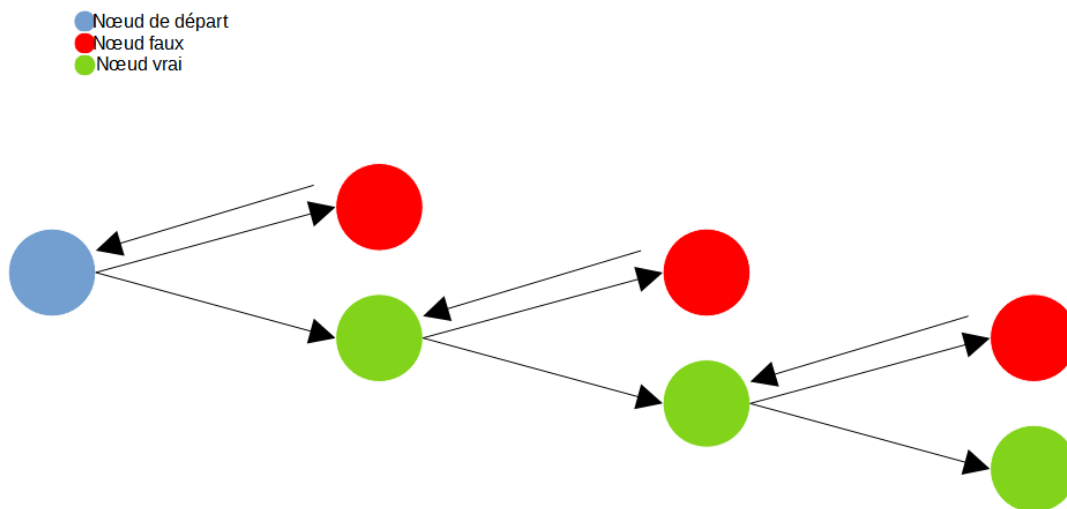


Schéma de l'arbre de décision typique d'un algorithme de backtracking

Il existe de nombreux modèles d'applications de la méthode de backtracking, comme par exemple, l'exemple le plus connu, Deep Blue, un super ordinateur de jeux d'échec. Il est surtout connu pour avoir battu le champion du monde d'échecs de l'époque, soit Garry Kasparov, en 1997. Le principe de résolution d'une partie d'échec est typiquement le même que le parcours de l'arbre des variables possible des partie d'échecs, à une différence près, c'est que le nombre de partie d'échec est bien supérieur à 10^{6000} en tenant compte de la règle des cinquante coups (il y aurait certes des parties déraisonnables (certaines parties pouvant durer près de 6000 coups)), et si l'on prend en compte seulement les parties raisonnables, on atteindrait 10^{123} combinaisons de parties d'échec différent. Alors certes l'ordinateur ne les calculera pas tous, mais ce chiffres est à titre informatif pour la suite.

Le Sudoku, un exemple d'application du backtracking

```
1 grid = [[5,3,0,0,7,0,0,0,0],
2         [6,0,0,1,9,5,0,0,0],
3         [0,9,8,0,0,0,0,6,0],
4         [8,0,0,0,6,0,0,0,3],
5         [4,0,0,8,0,3,0,0,1],
6         [7,0,0,0,2,0,0,0,6],
7         [0,6,0,0,0,0,2,8,0],
8         [0,0,0,0,1,9,0,0,5],
9         [0,0,0,0,0,0,0,0,0]]
10
11 print("----SUDOKU NON RESOLU----")
12 for k in range(len(grid)):
13     line = "? "
14     for l in range(len(grid[k])):
15         if l % 3 == 0 and l != 0 :
16             line += '? ' + f'{ grid[k][l]} '
17         elif l == 8 :
18             line += f'{ grid[k][l]} ' + '?'
19         else :
20             line += f'{ grid[k][l]} '
21     if k % 3 == 0:
22         print("|" + "-----"*2 + "-----" + "|")
23     print(line)
24 print("+" + "-----"*2 + "-----" + "+")
25
26 def possible(row, column, number, grid):
27     #Le nombre apparait-il dans la ligne ?
28     for i in range(0,9):
29         if grid[row][i] == number:
30             return False
31
32     #Le nombre apparait-il dans la colonne ?
33     for i in range(0,9):
34         if grid[i][column] == number:
35             return False
36
37     #Le nombre apparait-il dans le carré de 3*3 ?
38     x0 = (column // 3) * 3
39     y0 = (row // 3) * 3
40     for i in range(0,3):
41         for j in range(0,3):
42             if grid[y0+i][x0+j] == number:
43                 return False
44
45     return True
46
47 def solve(grid):
48     for row in range(0,9):
49         for column in range(0,9):
50             if grid[row][column] == 0:
51                 for number in range(1,10):
52                     if possible(row, column, number, grid):
53                         grid[row][column] = number
54                         solve(grid)
55                         grid[row][column] = 0
56                 return grid
57     print("??????SUDOKU RESOLU??????")
58     for k in range(len(grid)):
```



```

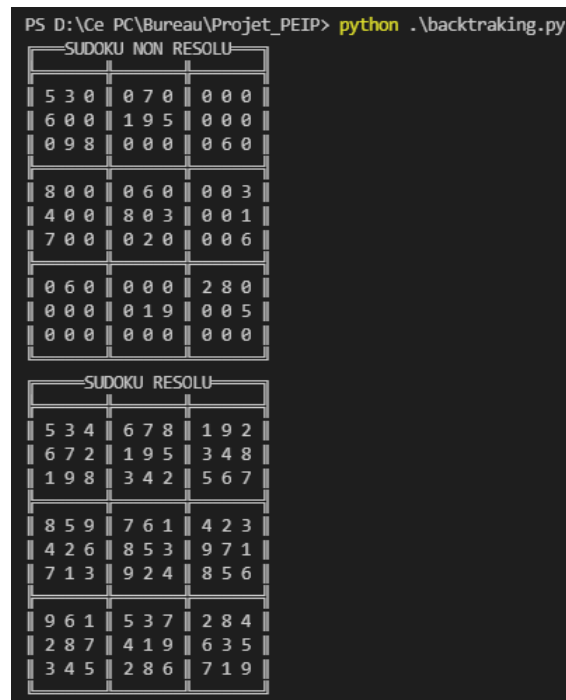
59     line = '?'
60     for l in range(len(grid[k])):
61         if l % 3 == 0 and l != 0 :
62             line += '?' + f'{ grid[k][l]} '
63         elif l == 8 :
64             line += f'{ grid[k][l]} ' + '?'
65         else :
66             line += f'{ grid[k][l]} '
67     if k % 3 == 0:
68         print("?"+"????????"*2+"????????"+"?")
69     print(line)
70     print("?"+"????????"*2+"????????"+"?")
71     exit()
72
73 solve(grid)

```

Algorithme utilisant le backtracking pour la résolution de Sudoku

A cause d'un problème d'encodage en \LaTeX , certains caractères ne ressortent pas correctement, nous vous prions de nous excuser pour ce désagrément.

Voici l'output de ce code qui resoud un sudoku.



```

PS D:\Ce PC\Bureau\Projet_PEIP> python .\backtracking.py

```

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	2	8	0
0	0	0	0	1	9	0	0	5
0	0	0	0	0	0	0	0	0

5	3	4	6	7	8	1	9	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	9	7	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	7	1	9

Si nous devons nous intéresser au pseudo code de cette algorithme, nous nous intéresserons seulement à la partie résolution, et non aux parties facultatives au fonctionnement du programme, tel que l'affichage, la grille de Sudoku qui reste la même, ...

```
1 Fonction possible(row, column, number, grid) :
2   Pour i allant de 0 à 9 avec un pas de 1 :
3     Si grid(ligne = row et colonne = i) égale a number :
4       Retourner False
5     Fin Si
6   Fin Pour
7
8   Pour i allant de 0 à 9 avec un pas de 1 :
9     Si grid(ligne = i et colonne = column) égale a number :
10      Retourner False
11    Fin Si
12  Fin Pour
13  x0 = (Reste de la division euclidienne de colonne par 3 ) * 3
14  y0 = (Reste de la division euclidienne de ligne par 3 ) * 3
15
16  Pour i allant de 0 à 3 avec un pas de 1 :
17    Pour j allant de 0 à 3 avec un pas de 1 :
18      Si grid(ligne = y0 + i et colonne = x0 + i) égale a number :
19        Retourner False
20      Fin Si
21    Fin Pour
22  Fin Pour
23
24  Retourner True
25 Fin Fonction
26
27 Fonction solve(grid):
28   Pour row allant de 0 à 9 avec un pas de 1 :
29     Pour column allant de 0 à 9 avec un pas de 1 :
30       Si grid(ligne = row et colonne = column) == 0:
31         Pour number allant de 1 à 10 avec un pas de 1:
32           Si possible(row, column, number, grid) retourne True :
33             grid(ligne = row et colonne = number) = number
34             solve(grid)
35           Fin Si
36         Fin Pour
37         grid(ligne = row et colonne = number) = 0
38         Retourner grid
39       Fin Si
40     Fin Pour
41   Fin Pour
42   Afficher grid
43 Fin Fonction
44
45 Appeler solve(grid)
```

Pseudo code de résolution de Sudoku

Le backtracking en général

Ainsi, si l'on veut écrire un algorithme en pseudo-code plus généraliste sur le backtracking, nous pourrions vous proposer celui ci :

```
1 n = variable quelconque
2
3 Fonction validité_de_la_variable(n):
4     Si n = Valide :
5         Retourner Vrai
6     Sinon :
7         Retourner Faux
8
9 Fonction backtracking(n):
10    Pour i allant de x à y avec un pas de w:
11        Si validité_de_la_variable(i) est Vraie :
12            n = i
13            backtracking(n)
14        Fin Si
15    Fin Pour
16    n = 0
17    Retourner n
18    Afficher n
19 Fin Fonction
20
21 Appeler backtracking(n)
```

Pseudo code généraliste du backtracking

Conclusion

Nous avons donc pu voir différents exemples dans la vraie vie ou dans le domaine des mathématiques de la récursivité. A partir de la nous avons défini la récursivité dans l'informatique, qui a été explicité grâce à plusieurs exemples et nous l'avons comparé à une méthode plus simple dites par itérativité. Enfin nous avons étudié une application de la récursivité dans l'informatique, via l'exemple du backtracking.

Projets Département Informatique (Luminy)

Encadrement : Alain Samuel

alain.samuel@univ-amu.fr

NB : la recherche sur *Internet* est largement recommandée mais ce que vous écrirez dans votre rapport final devra être parfaitement compris

Récurtivité informatique

On s'intéresse aux constructions récursives dont la caractéristique est de se définir par rapport à elles-mêmes.

- donner des exemples généraux de telles constructions

En programmation informatique, la solution d'un problème est récursive si elle s'exprime à partir du même problème appliqué à des cas « plus simples ».

- à quoi sert le cas de base dans ce contexte ?
- donner un exemple simple de problème récursif, expliquer.
- étudier le problème récursif des tours de Hanoï, expliquer.

La récursivité est une généralisation de la récurrence mathématique, expliquer en quoi ?

Tout problème récursif peut se résoudre de manière itérative.

- qu'est-ce qu'une itération en programmation ?
- réécrire l'exemple simple du problème récursif que vous avez précédemment choisi avec une itération.
- quels sont les avantages et les inconvénients des deux approches ?

Pour terminer ce travail, vous étudierez une application de la récursivité : par exemple le *backtracking* (ou autre) pour lequel vous donnerez une définition et quelques exemples de problèmes de cette nature, vous décrierez l'utilisation de la récursivité dans ce contexte et vous écrierez en pseudo-code un algorithme récursif général de résolution de ce type de problème.

$$8 \int_a^b f(x) dx$$