Le Sudoku, un casse-tête japonais

Grille **initiale**

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

Compléter chaque case vide
avec un chiffre de 1 à 9

Grille **complétée**

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

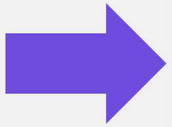
Pas de répétition de chiffres sur
chaque ligne, colonne et bloc

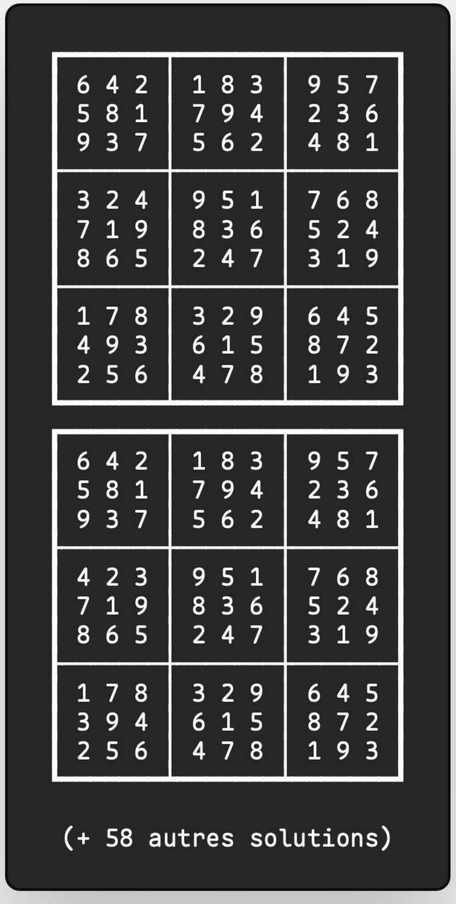
La fonction qu'on souhaite écrire…

`remplir(`

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

`)`



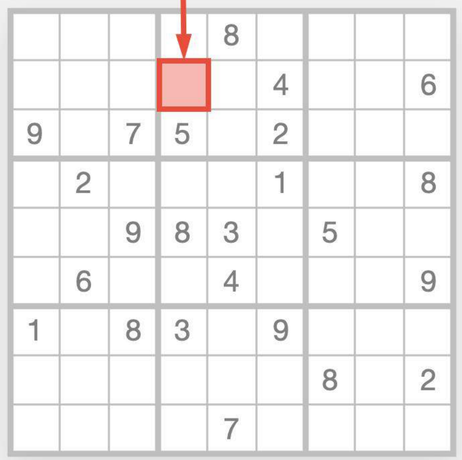


Cette fonction tente de remplir la grille donnée et **affiche les solutions possibles** (aucune, une ou plusieurs)

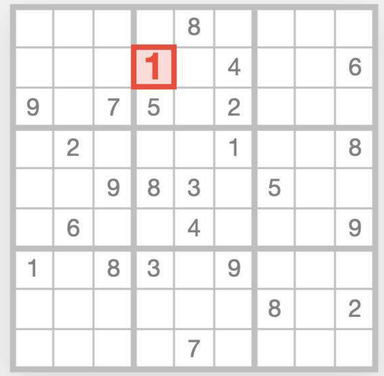
La fonction qu'on souhaite écrire… récursivement.

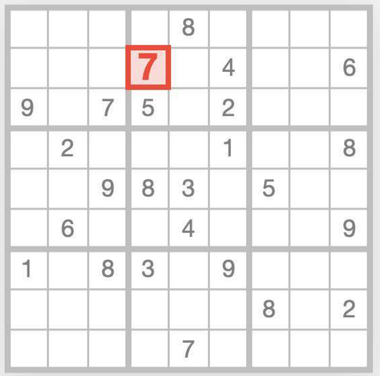
On cherche une **case vide** de la grille.

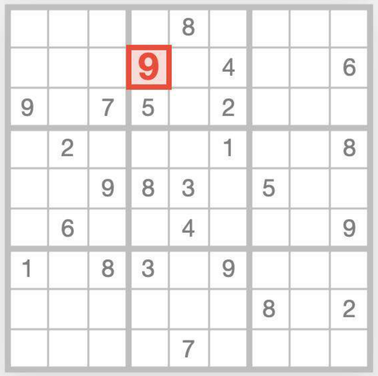
Ici, 1, 7 et 9 sont possibles.

`remplir(`  `)`

Cela revient au même d'appeler

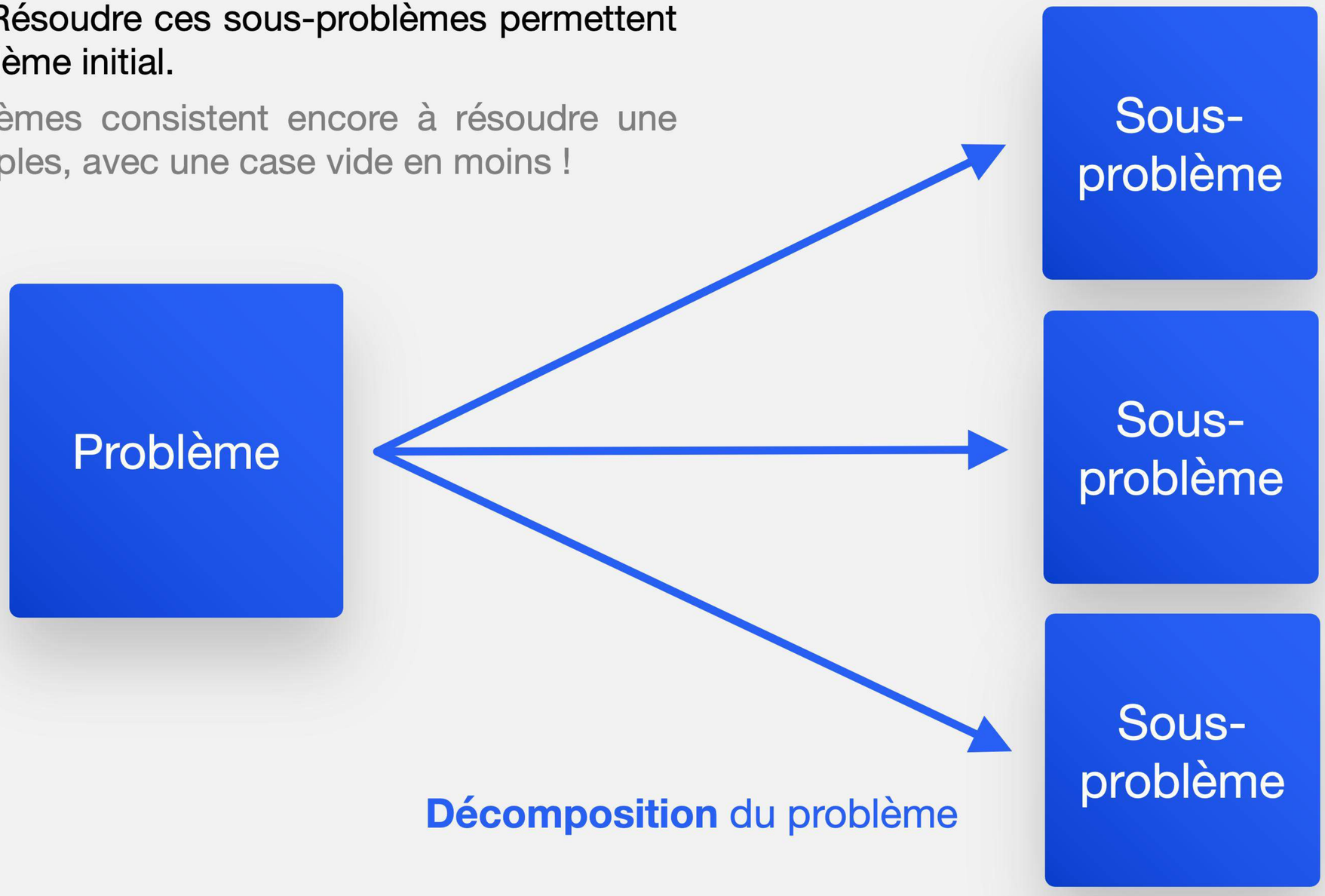
`remplir(`  `)`

`remplir(`  `)`

`remplir(`  `)`

Un sous-problème est **de même nature** que le problème, mais **plus simple**. Résoudre ces sous-problèmes permettent de résoudre le problème initial.

Ici, nos sous-problèmes consistent encore à résoudre une grille, mais plus simples, avec une case vide en moins !

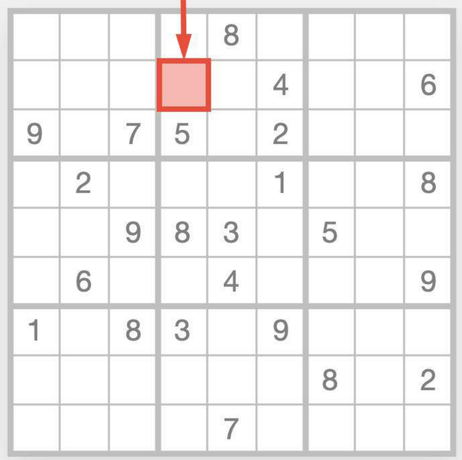


Décomposition du problème

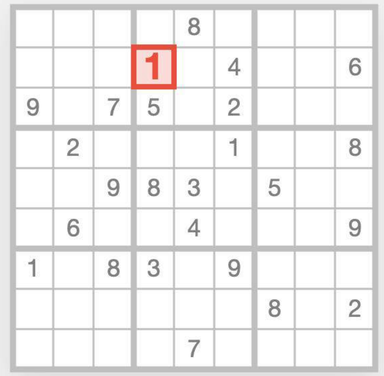
La fonction qu'on souhaite écrire… récursivement.

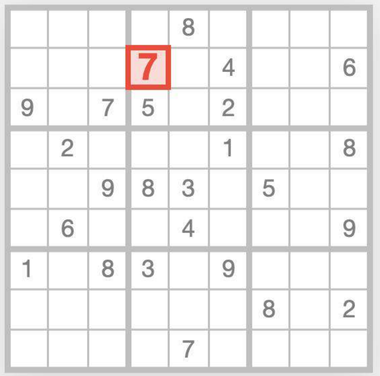
On cherche une **case vide** de la grille.

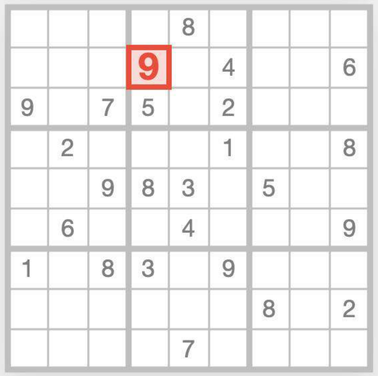
Ici, 1, 7 et 9 sont possibles.

`remplir(`  `)`

Cela revient au même d'appeler

`remplir(`  `)`

`remplir(`  `)`

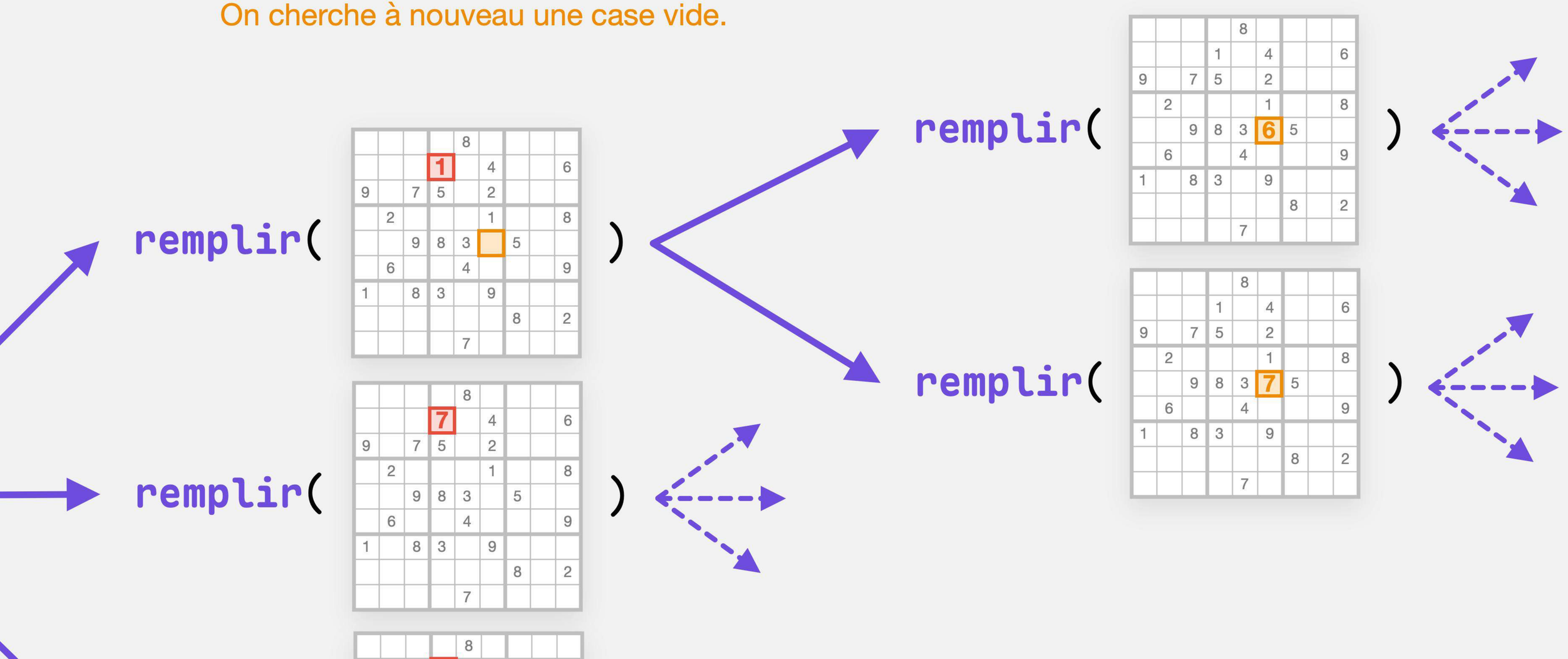
`remplir(`  `)`

La fonction qu'on souhaite écrire… récursivement.

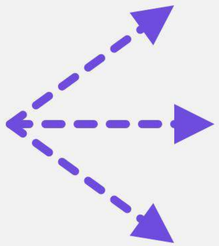
Notre définition de `remplir` est récursive.

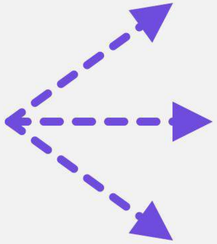
On poursuit cette décomposition.

On cherche à nouveau une case vide.



Décomposer jusqu'où ? Le(s) cas de base.





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

La grille est **complète**.
On l'affiche !

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

Une case vide est impossible à compléter.
On est tombé dans une impasse, tant pis, on ne fait rien.

Le pseudocode

```
fonction remplir(grille)
    chercher une case vide
    si aucune case vide alors
        afficher la grille et stop
    sinon
        déterminer les chiffres possibles pour cette case vide
        si pas de chiffres possibles alors
            stop
        sinon
            pour chaque chiffre possible
                créer grille suivante avec ce chiffre dans la case vide
                appeler remplir(grille suivante)
```

Le pseudocode

```
fonction remplir(grille)
    chercher une case vide                                    Cas de base
    si aucune case vide alors
        afficher la grille et stop
    sinon
        déterminer les chiffres possibles pour cette case vide
        si pas de chiffres possibles alors
            stop
        sinon                                                 Cas récursif
            pour chaque chiffre possible
                créer grille suivante avec ce chiffre dans la case vide
                appeler remplir(grille suivante)
```

Le pseudocode

```
fonction remplir(grille)
    chercher une case vide
    si aucune case vide alors
        afficher la grille et stop
    sinon
        déterminer les chiffres possibles pour cette case vide
        si pas de chiffres possibles alors
            stop
        sinon
            pour chaque chiffre possible
                créer grille suivante avec ce chiffre dans la case vide
                appeler remplir(grille suivante)
```

Code Python

```
def remplir(grille):
    x, y = chercher_case_vide(grille)
    if (x, y) == (None, None):
        afficher_grille(grille)
    else:
        chiffres = chiffres_possibles(grille, x, y)
        if not chiffres:
            return
        else:
            for chiffre in chiffres:
                grille_suivante = deepcopy(grille)
                grille_suivante[x][y] = chiffre
                remplir(grille_suivante)
```

Code Python

```
def remplir(grille):
    x, y = chercher_case_vide(grille)
    if (x, y) == (None, None):
        afficher_grille(grille)
    else:
        chiffres = chiffres_possibles(grille, x, y)
        if not chiffres:
            return
        else:
            for chiffre in chiffres:
                grille_suivante = deepcopy(grille)
                grille_suivante[x][y] = chiffre
                remplir(grille_suivante)
```

Renvoie les coordonnées d'une case vide de la `grille` donnée si elle existe, `(None, None)` sinon.

Affiche joliment la `grille` donnée.

Renvoie une liste des chiffres possibles pour la case aux coordonnées `(x, y)` de la `grille` donnée.

Code Python

```
def remplir(grille):
    x, y = chercher_case_vide(grille)
    if (x, y) == (None, None):
        afficher_grille(grille)
    else:
        chiffres = chiffres_possibles(grille, x, y)
        if not chiffres:
            return
        else:
            for chiffre in chiffres:
                grille_suivante = deepcopy(grille)
                grille_suivante[x][y] = chiffre
                remplir(grille_suivante)
```

Effectue une copie réelle de la `grille` (liste de listes).

```
from copy import deepcopy
```

Pourquoi ne pas simplement écrire ?

```
grille_suivante = grille
```

Rappel sur les références

```python
a = 7
b = a
b += 12
print(a)
```

```
7
```

Nom	Valeur
a	7
b	19

```python
a = [1, 2, 3, 4, 5]
b = a
b.append(42)
print(a)
```

```
[1, 2, 3, 4, 5, 42]
```

Nom	Valeur
a	
b	

a ne stocke pas la liste, mais une **référence** vers la liste.

Une référence indique où cette liste se situe en mémoire, c'est une adresse.

Il y a bien eu une copie, mais une **copie de référence** ! Donc a et b font référence à la même liste.

Une parfaite analogie

Vous copiez la **vidéo.**

Environ 240 Mo.

Vous partagez une **référence** (lien hypertexte) vers la vidéo.

Quelques octets.

Quel intérêt ?

```
def fonction(param1, param2):
    ...

a = 10
b = 7
fonction(a, b)
```

```
param1 = a
param2 = b
```

Lors d'un appel de fonction, les valeurs sont **copiées** dans les paramètres.

Nom	Valeur
a	10
b	7
param1	10
param2	7

Quel intérêt ?

```python
def fonction(param1, param2):
    ...

a = [42, 72, ..., 252]
b = 7
fonction(a, b)
```

```
param1 = a
param2 = b
```

Lors d'un appel de fonction, les valeurs sont **copiées** dans les paramètres.

Nom	Valeur
a	●
b	7
param1	●
param2	7

Une liste est **passée par référence**, car ça serait **trop coûteux** d'en faire une copie réelle !

La plupart du temps, une fonction lit les éléments de la liste, sans la modifier.

À part les types de base (`int` `float` `str` `bool`),

tout est copié par référence !

Et si je veux faire une **vraie** copie d'une liste ?

Méthode `copy`

```
a = [42, 37, 11, 84]
b = a.copy()
```

Copie élément par élément

```
a = [42, 37, 11, 84]
b = []
for val in a:
    b.append(val)
```

`val` est copiée, mais si `val` était une référence ?

Slice (tranche)

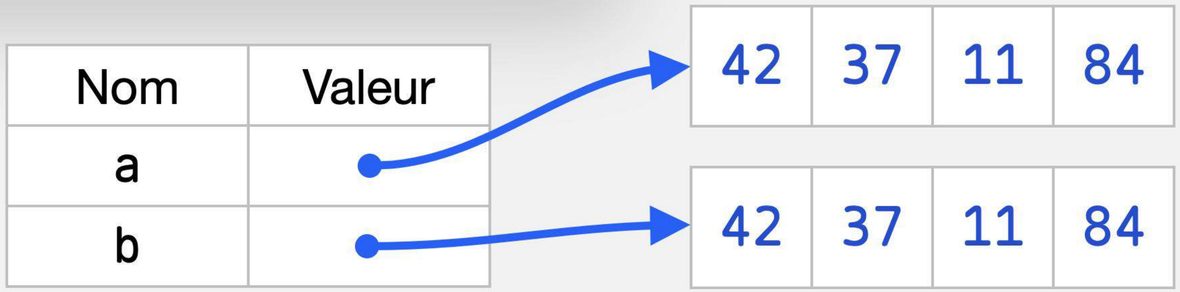
```
a = [42, 37, 11, 84]
b = a[:]
```

Compréhension de liste

```
a = [42, 37, 11, 84]
b = [val for val in a]
```

Euh ? Et **`deepcopy`** ?

Toutes ces copies sont en fait des copies **superficielles** !



Et si je veux faire une **vraie** copie d'une liste ?

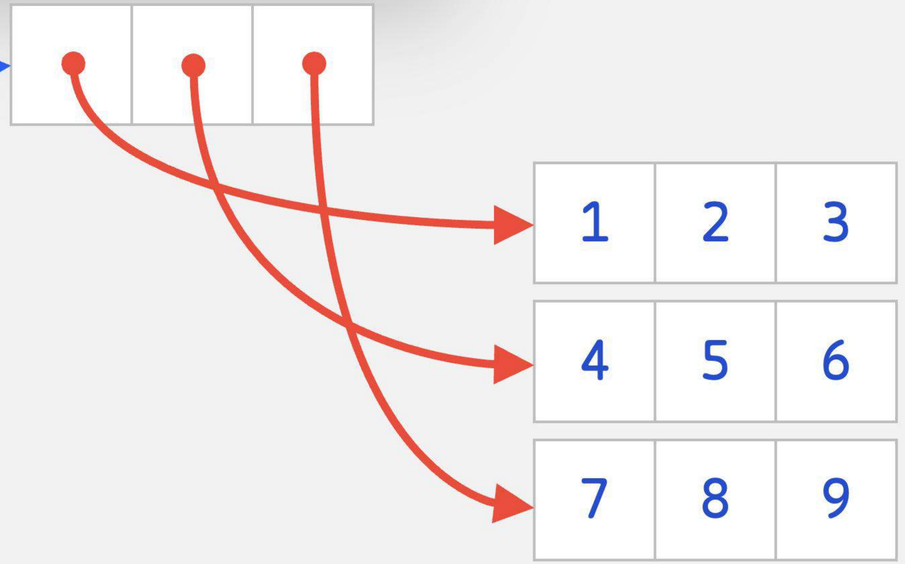
Représentation d'une grille par **une liste de listes**

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

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

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

Nom	Valeur
a	



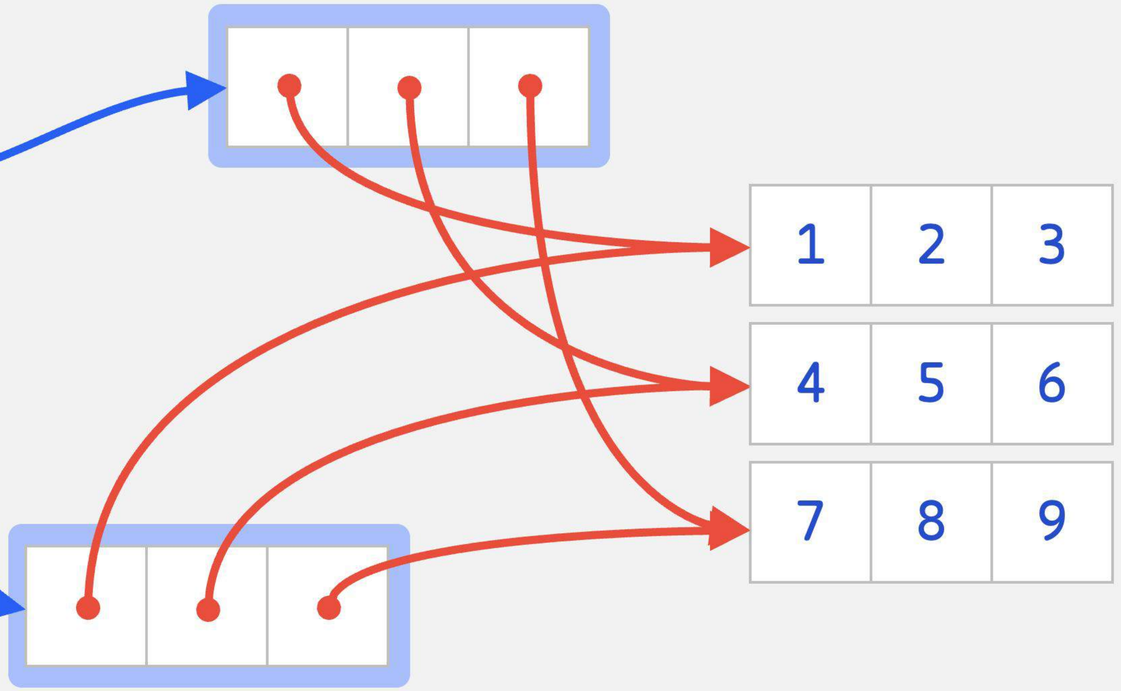
a est une référence d'une liste de références 🤯

Et si je veux faire une **vraie** copie d'une liste ?

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

b.copy()
```

Nom	Valeur
a	
b	

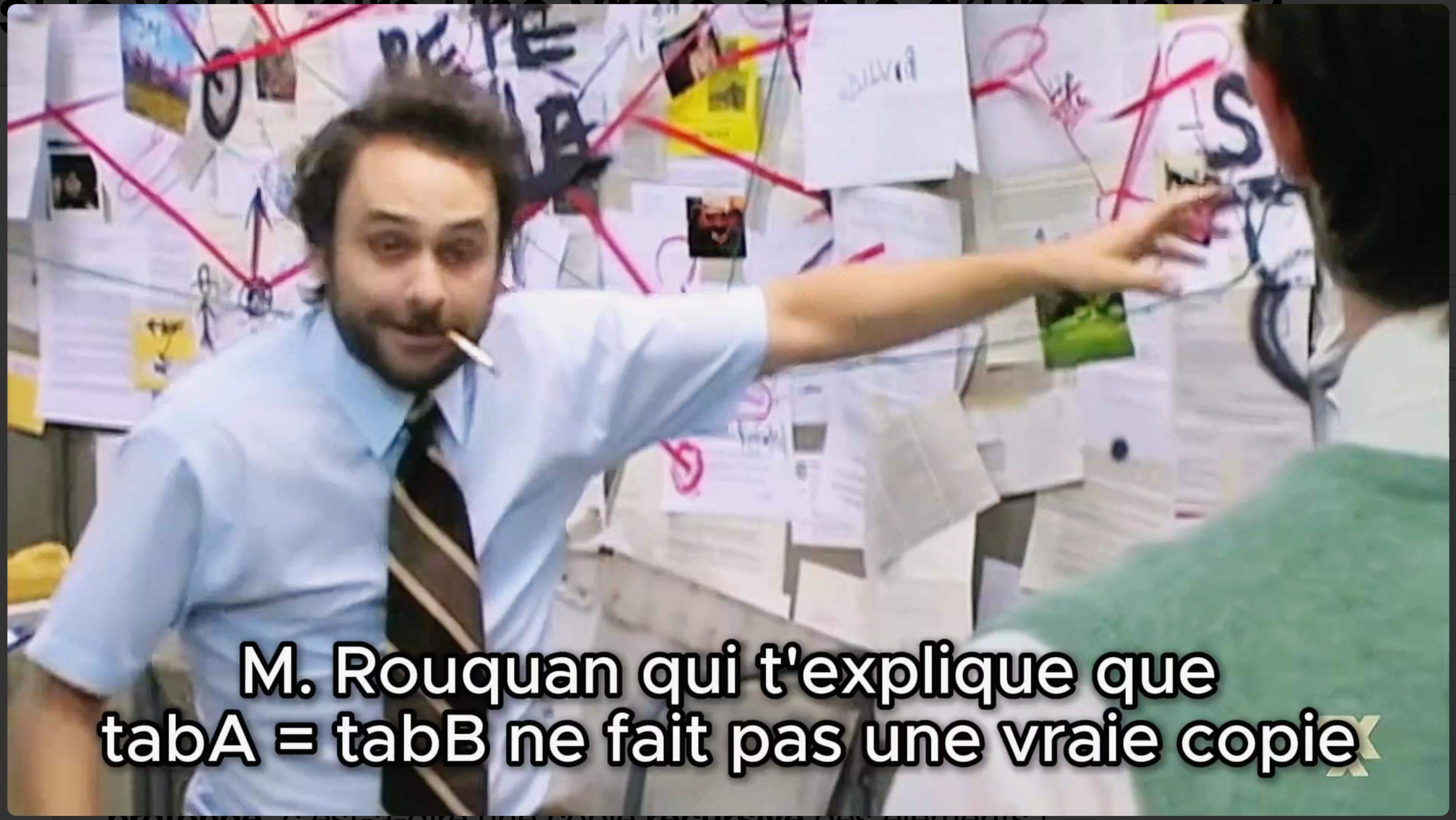


a est une référence d'une liste de références 🤯

La liste de références est bien copiée, mais pas les listes plus « profondes » !
`copy` effectue une copie **superficielle**.

`deepcopy` résout ce problème, il effectue une copie **profonde**, c'est-à-dire une copie **récursive** des éléments !

Et si je veux faire une vraie copie d'une liste ?



3

6

9

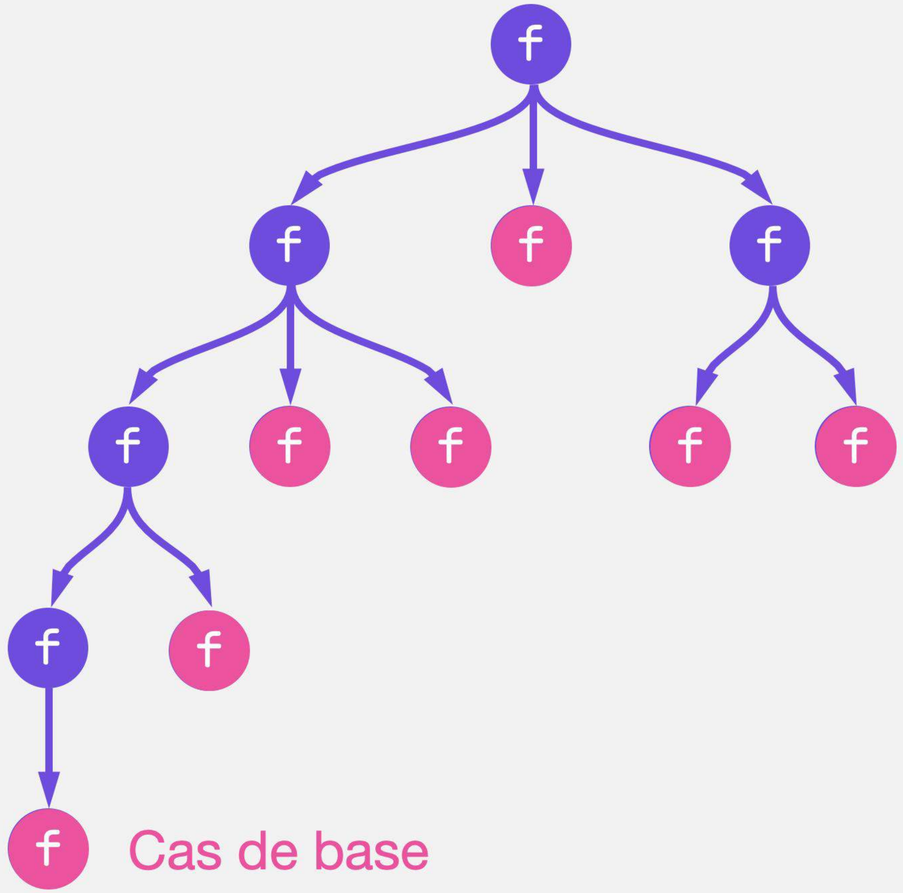
profonde, c'est-à-dire une copie **récursive** des éléments !

Code Python

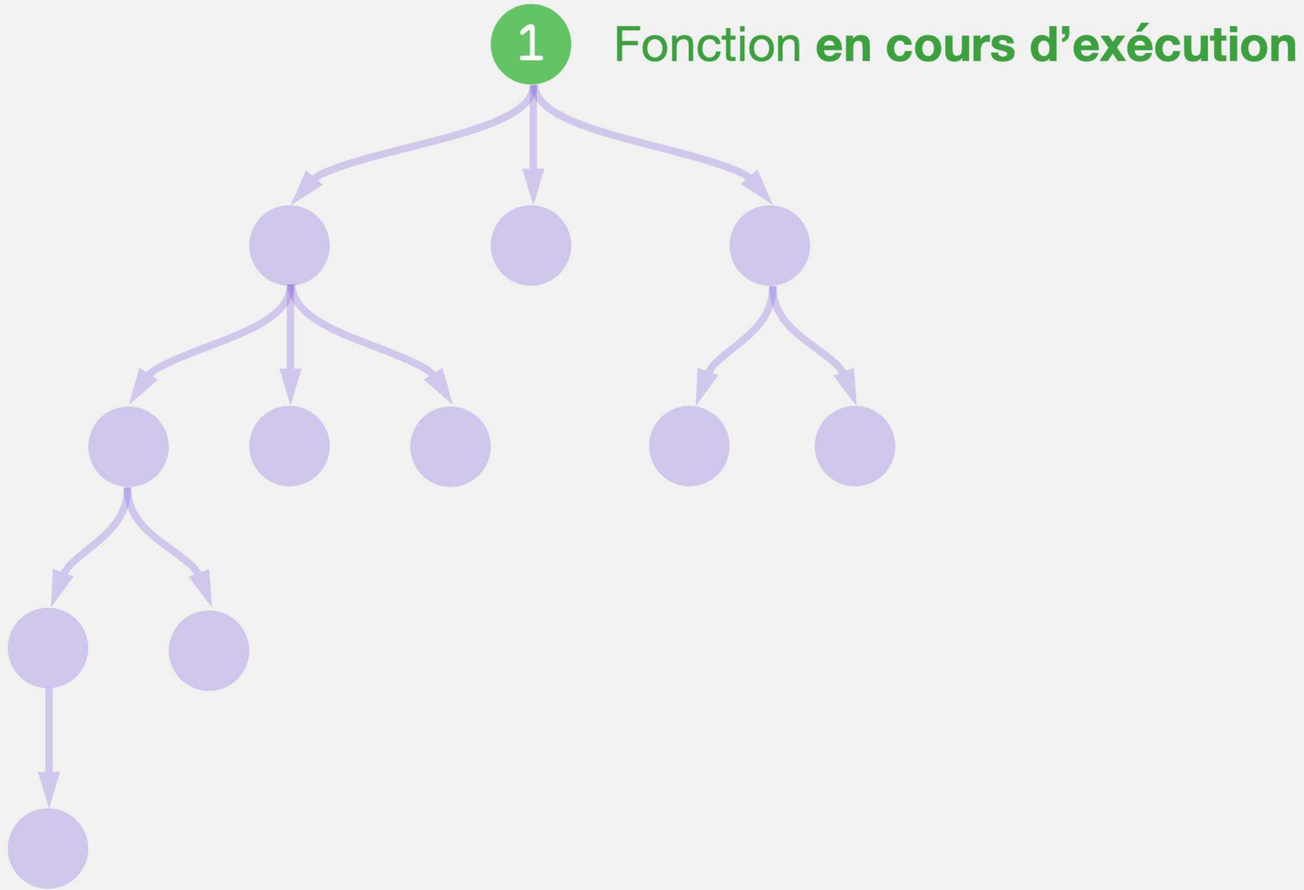
```
def remplir(grille):
    x, y = chercher_case_vide(grille)
    if (x, y) == (None, None):
        afficher_grille(grille)
    else:
        chiffres = chiffres_possibles(grille, x, y)
        if not chiffres:
            return
        else:
            for chiffre in chiffres:
                grille_suivante = deepcopy(grille)
                grille_suivante[x][y] = chiffre
                remplir(grille_suivante)
```

À chaque appel de **remplir** une copie de grille est effectuée. Euh, ça va pas exploser ? 💥

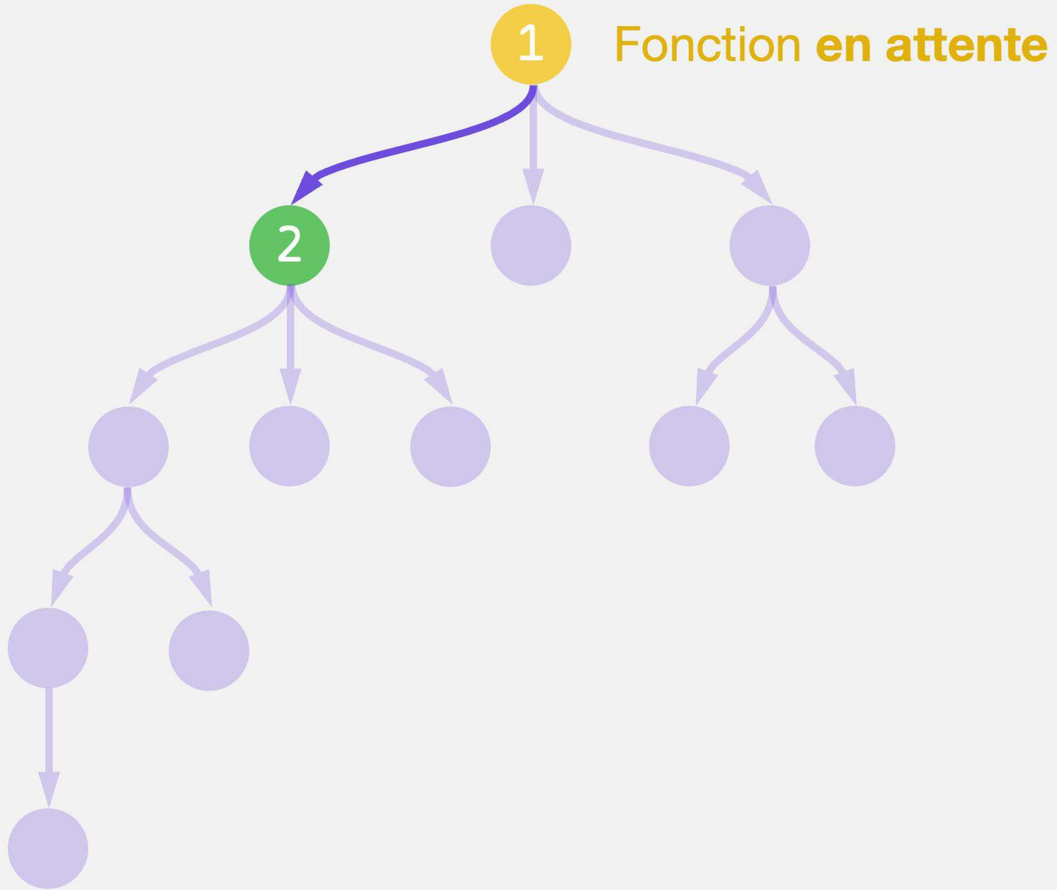
L’ordre des appels



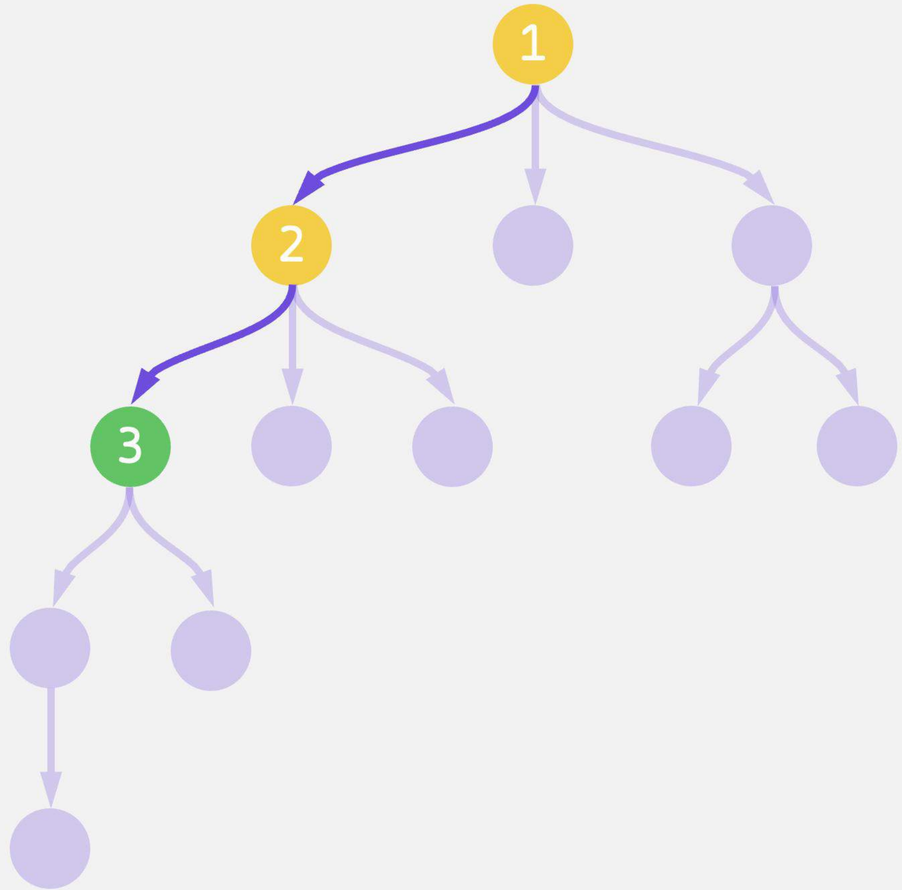
L’ordre des appels



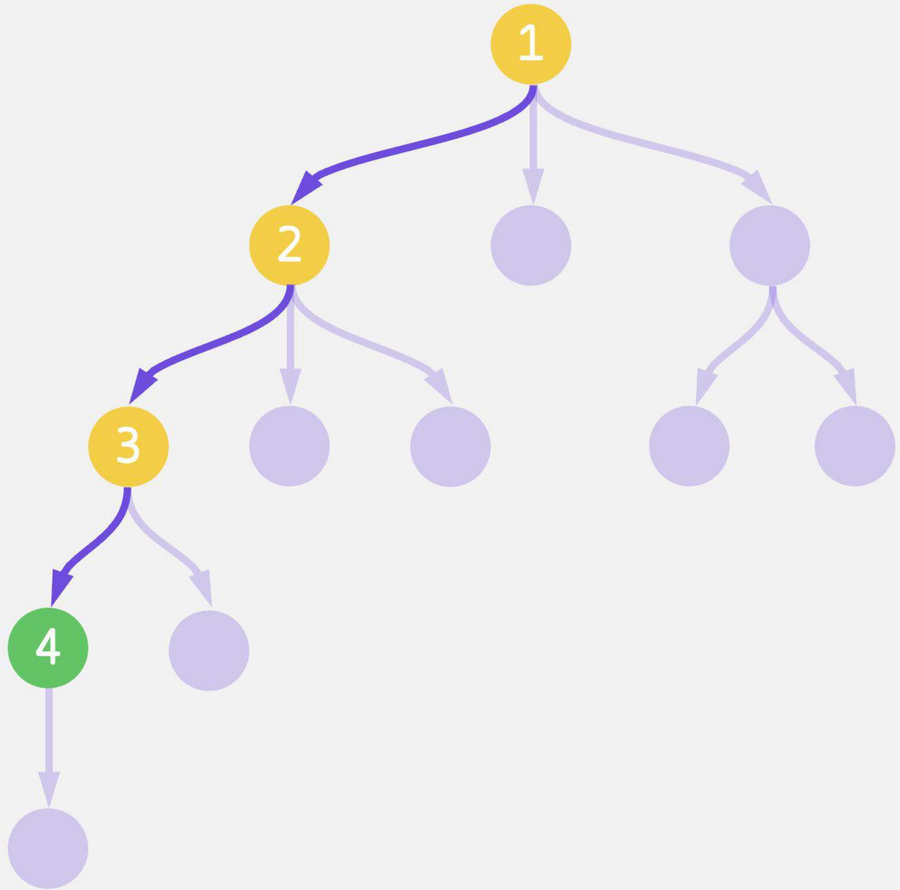
L’ordre des appels



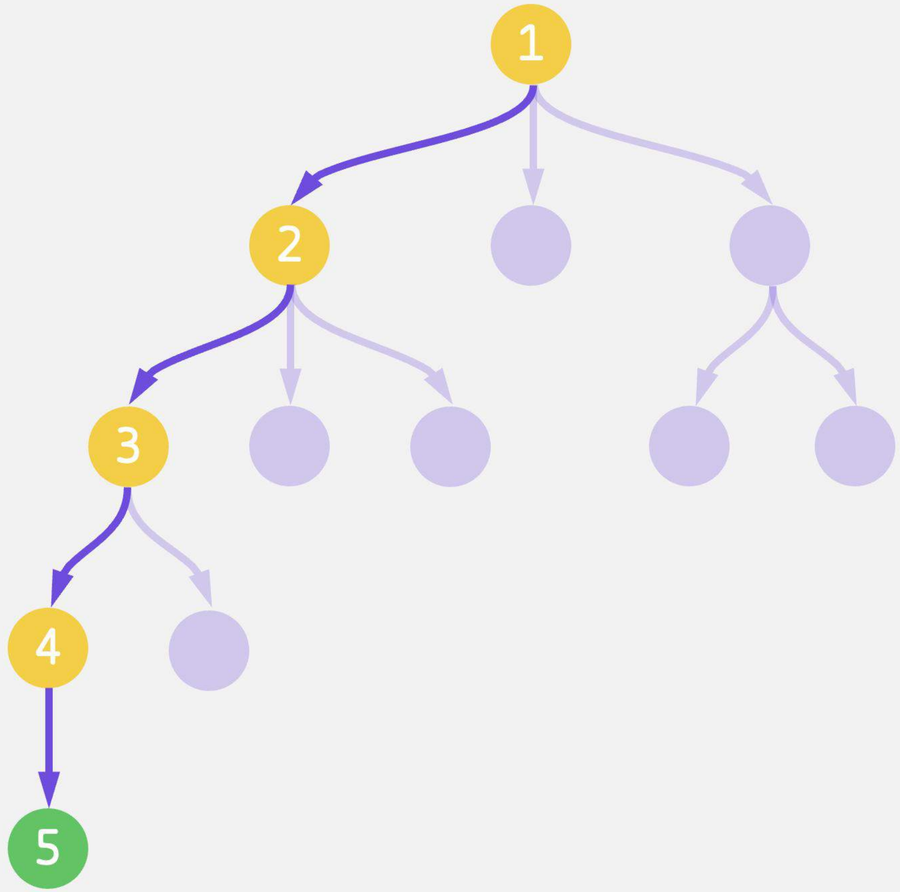
L’ordre des appels



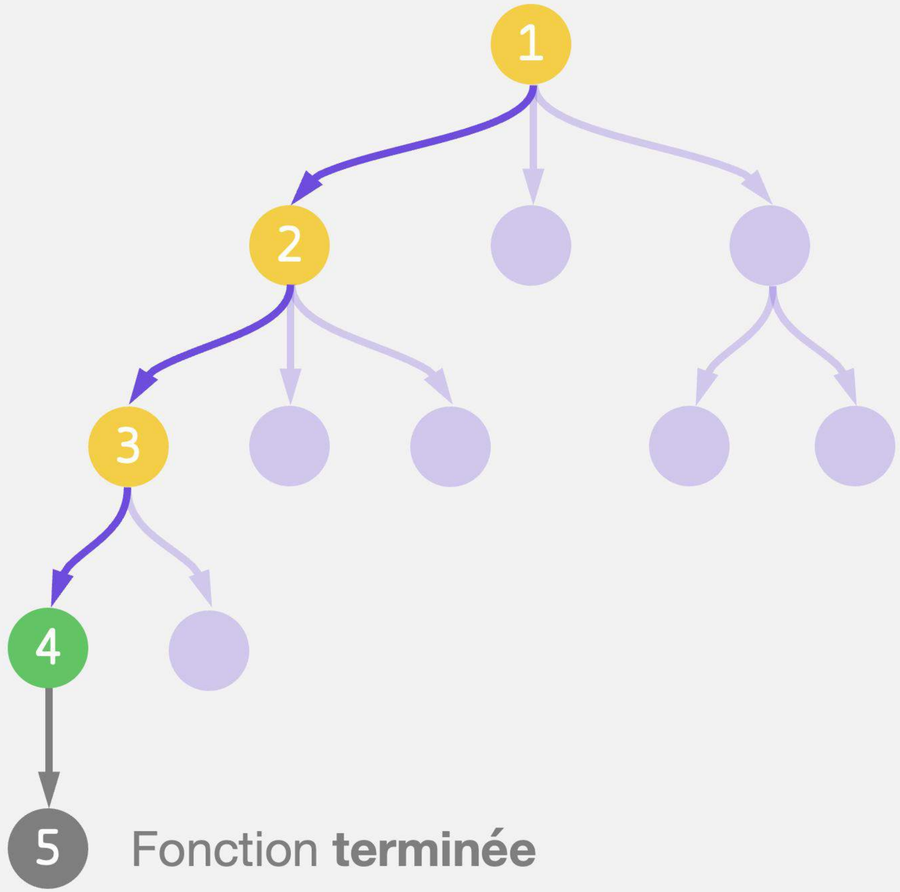
L’ordre des appels



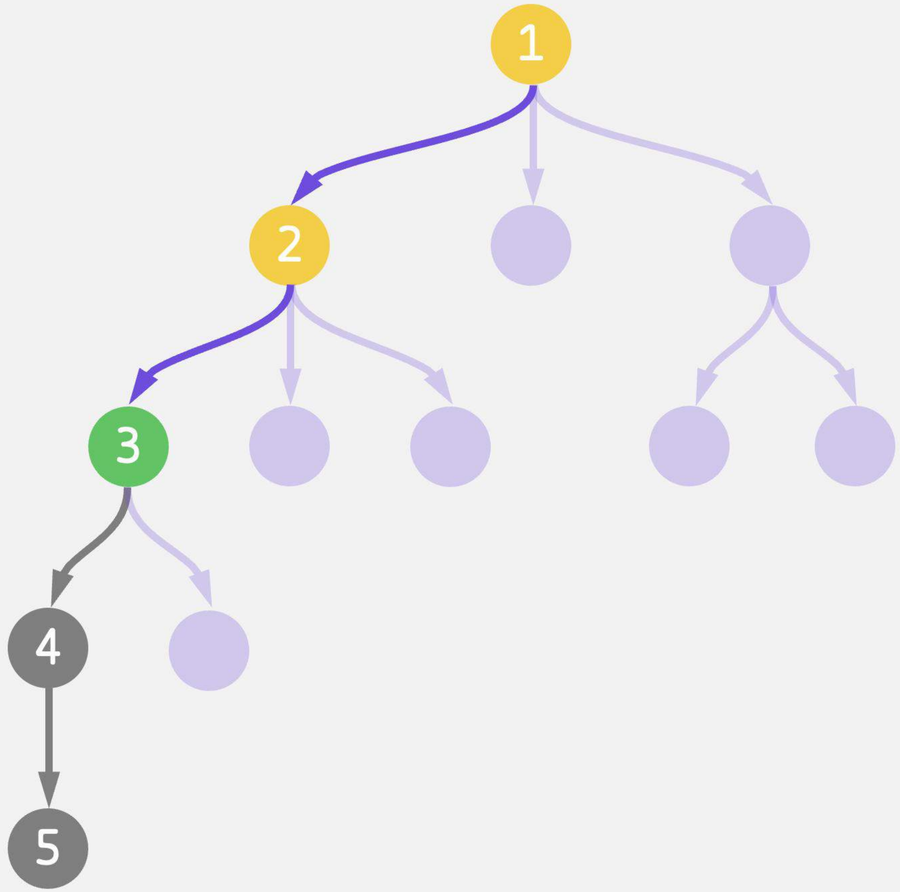
L’ordre des appels



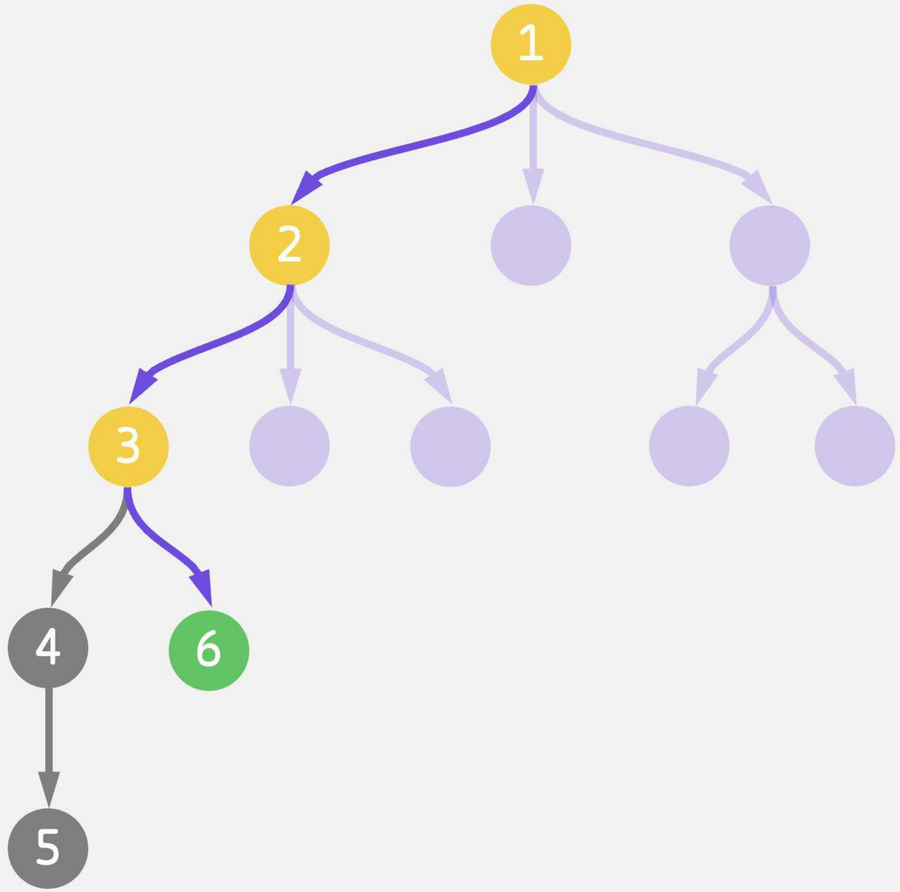
L’ordre des appels



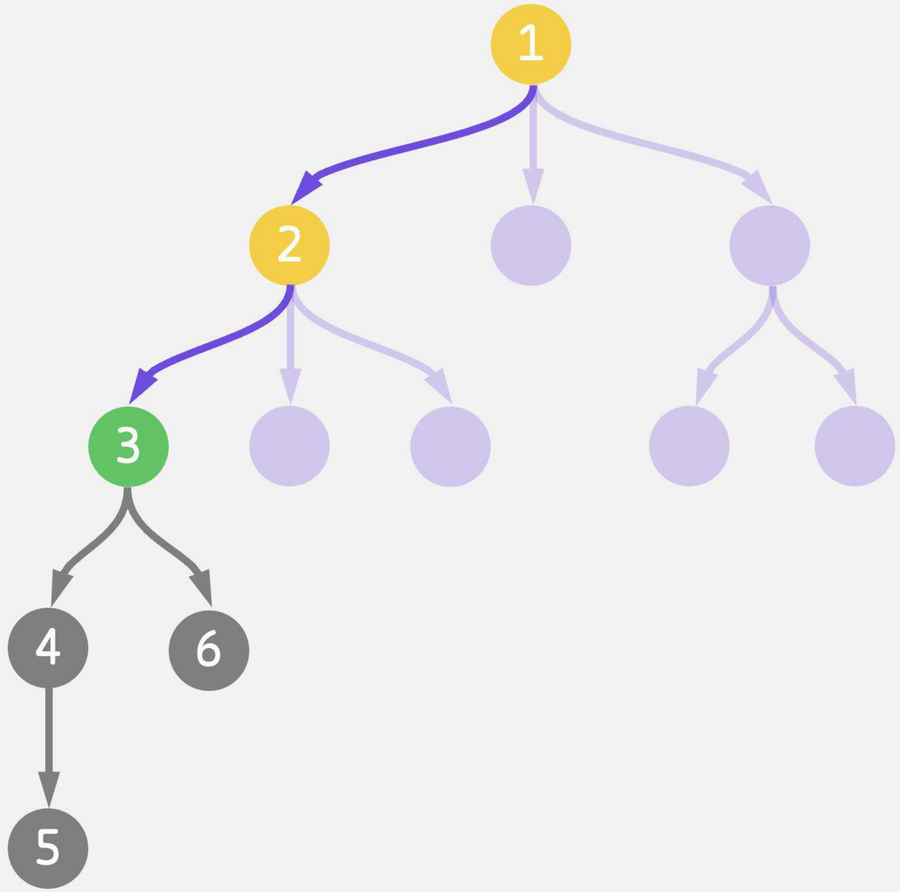
L’ordre des appels



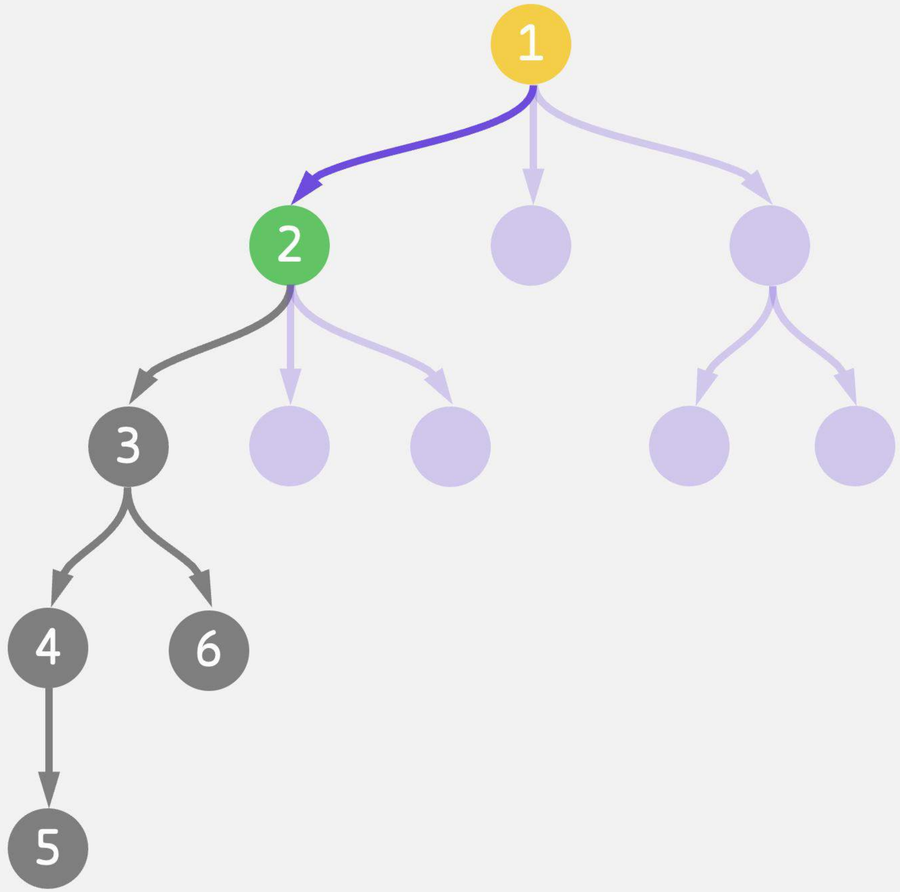
L’ordre des appels



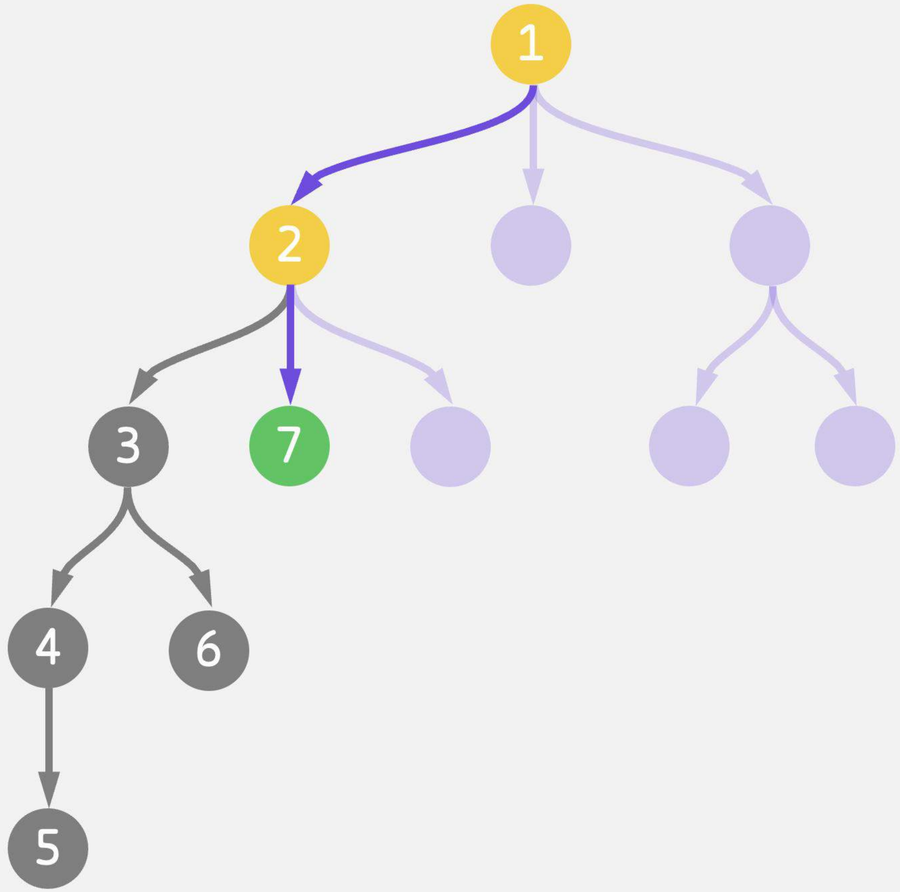
L’ordre des appels



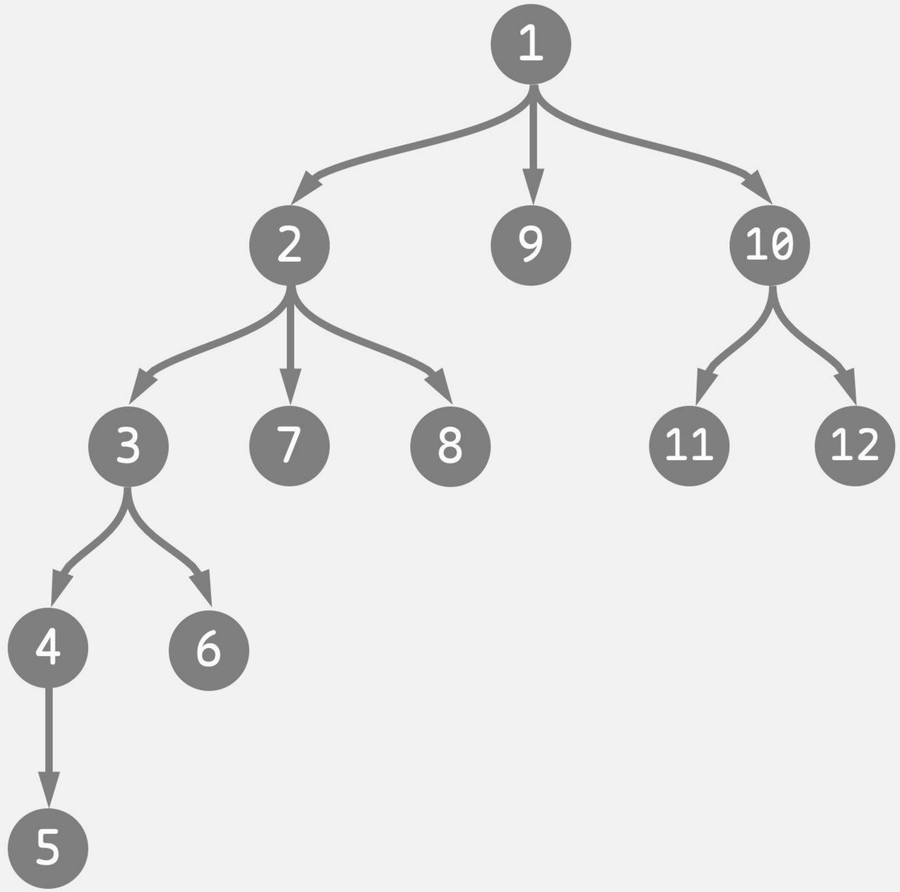
L’ordre des appels



L’ordre des appels

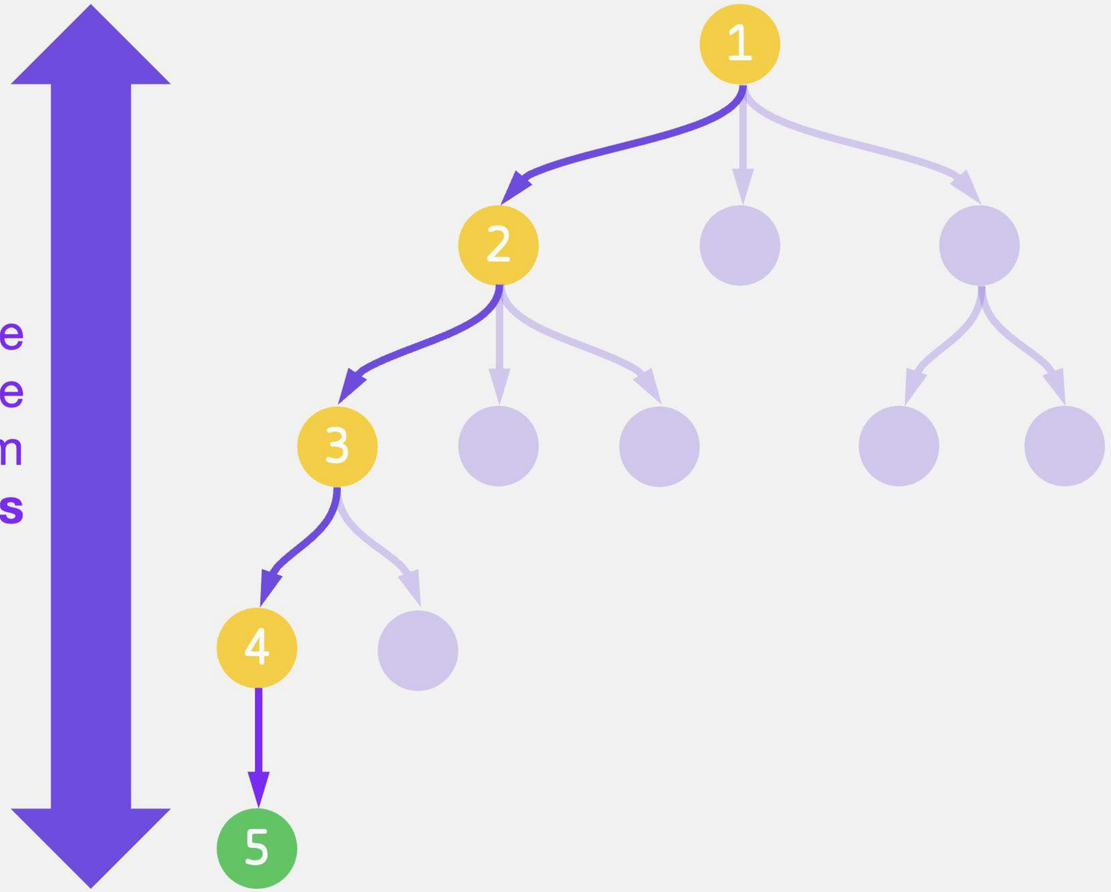


L’ordre des appels



L’ordre des appels

La profondeur de l’arbre d’appels donne le nombre maximum de **fonctions actives** en même temps.



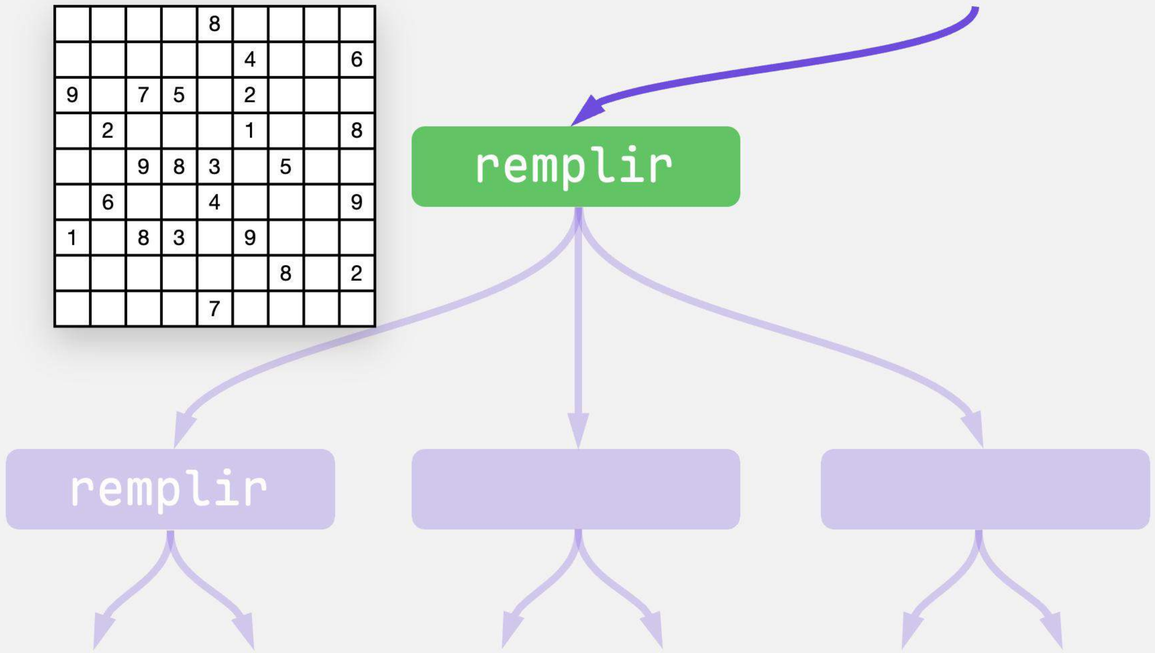
Pour la fonction `remplir`, quel est le nombre de fonctions actives au maximum ?

On remplit 1 case à chaque appel, donc la profondeur de l’arbre est égale au **nombre de cases vides** !

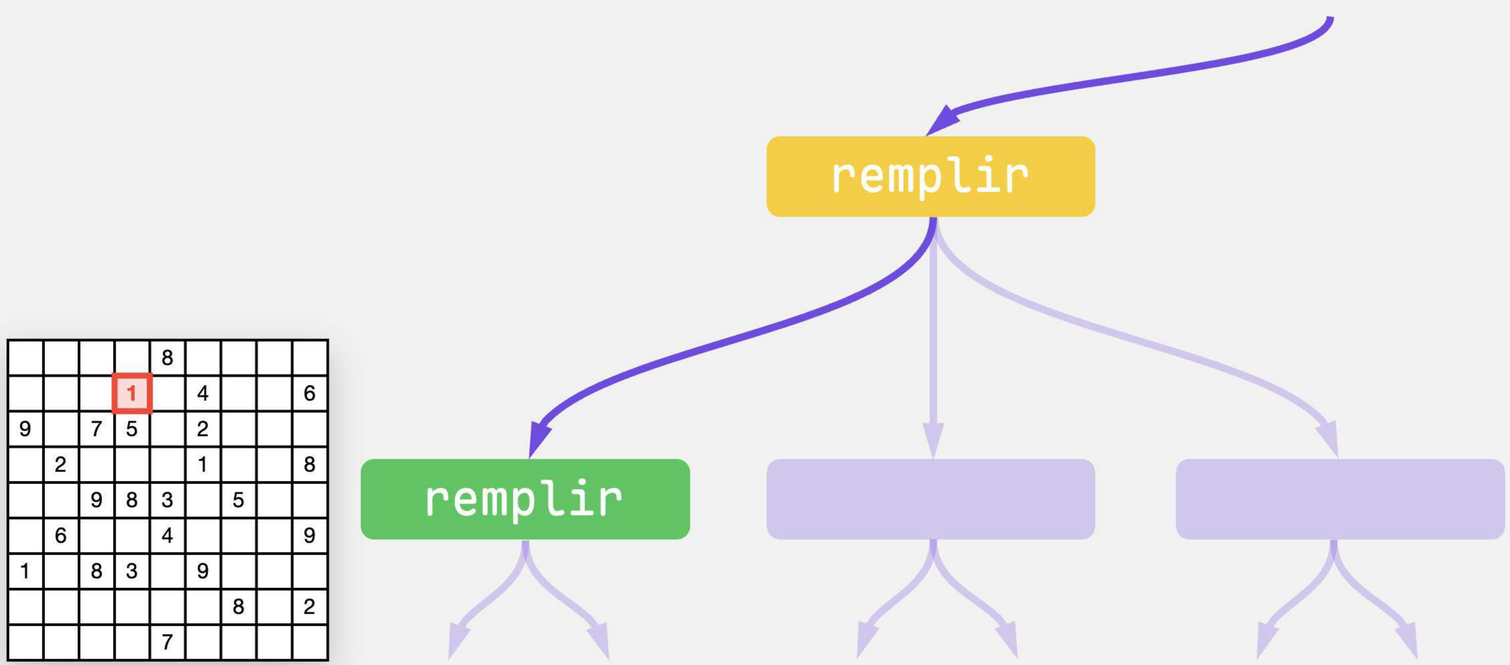
Il y a donc au maximum 9 × 9 = **81** fonctions `remplir` actives en même temps.

Soit au maximum 81 grilles actives en mémoire en même temps. Ouf.

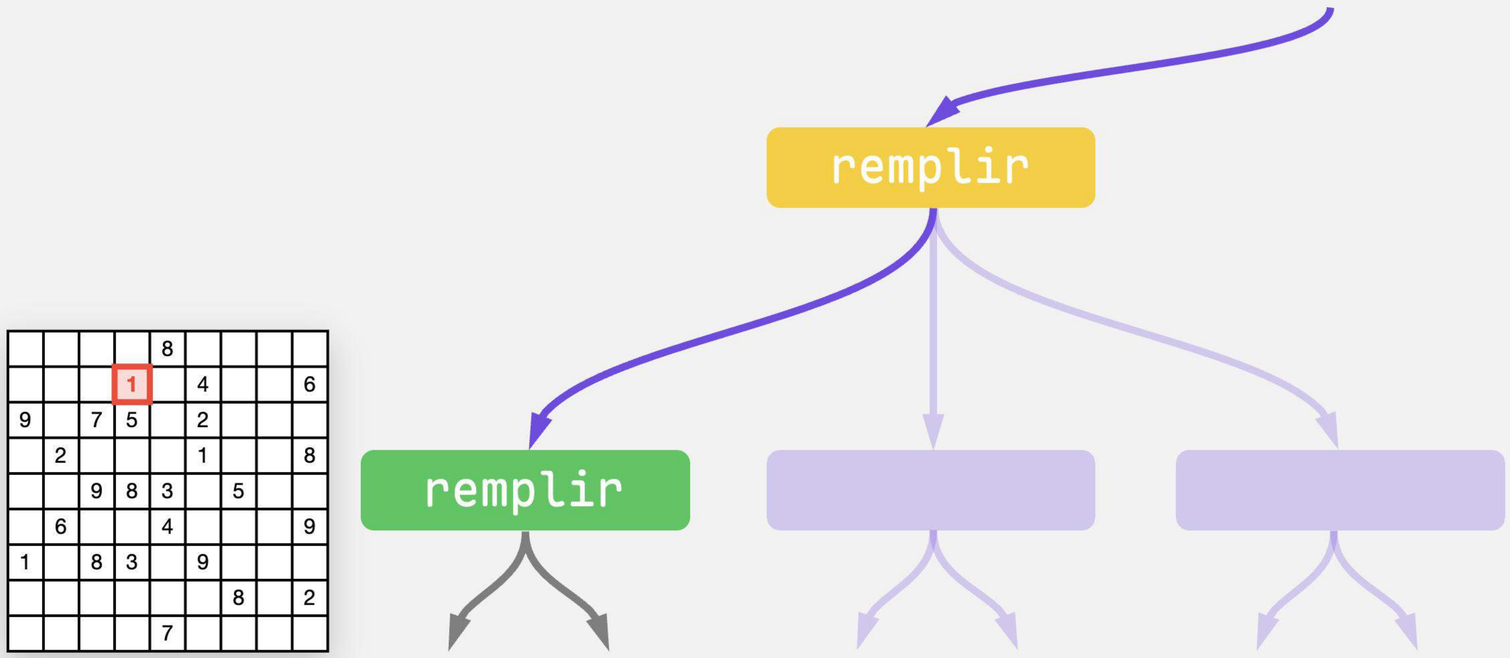
Mais une seule grille suffit



Mais une seule grille suffit

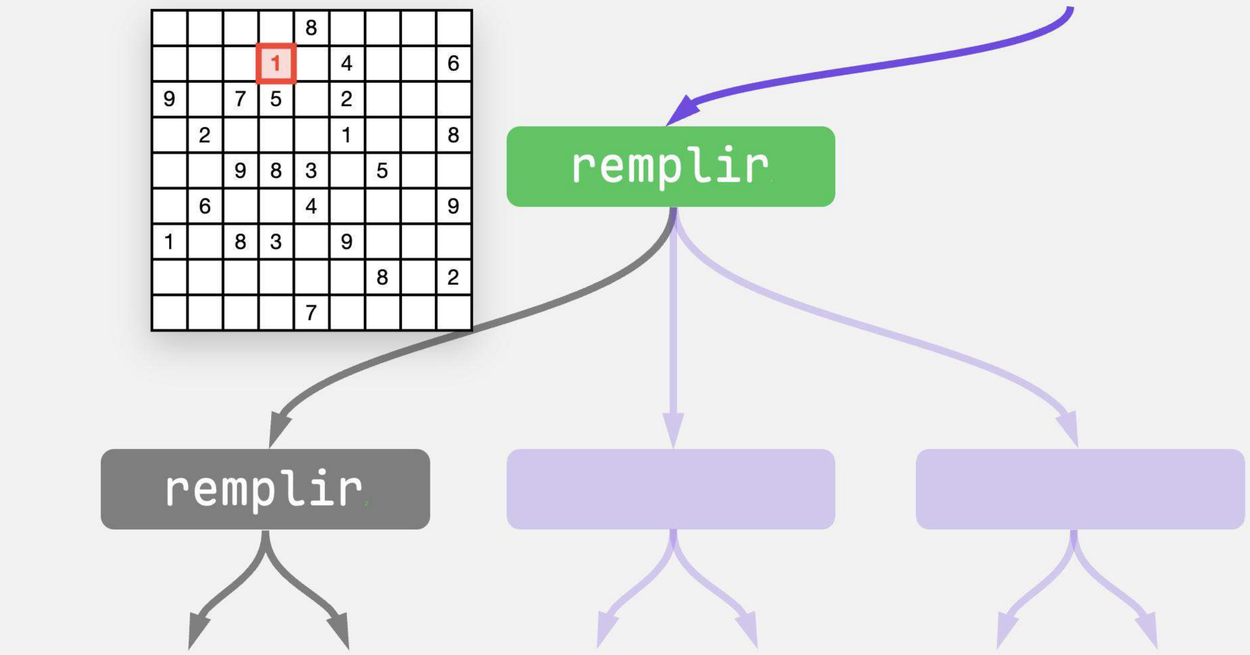


Mais une seule grille suffit



On suppose qu'on a
exploré la descendance

Mais une seule grille suffit



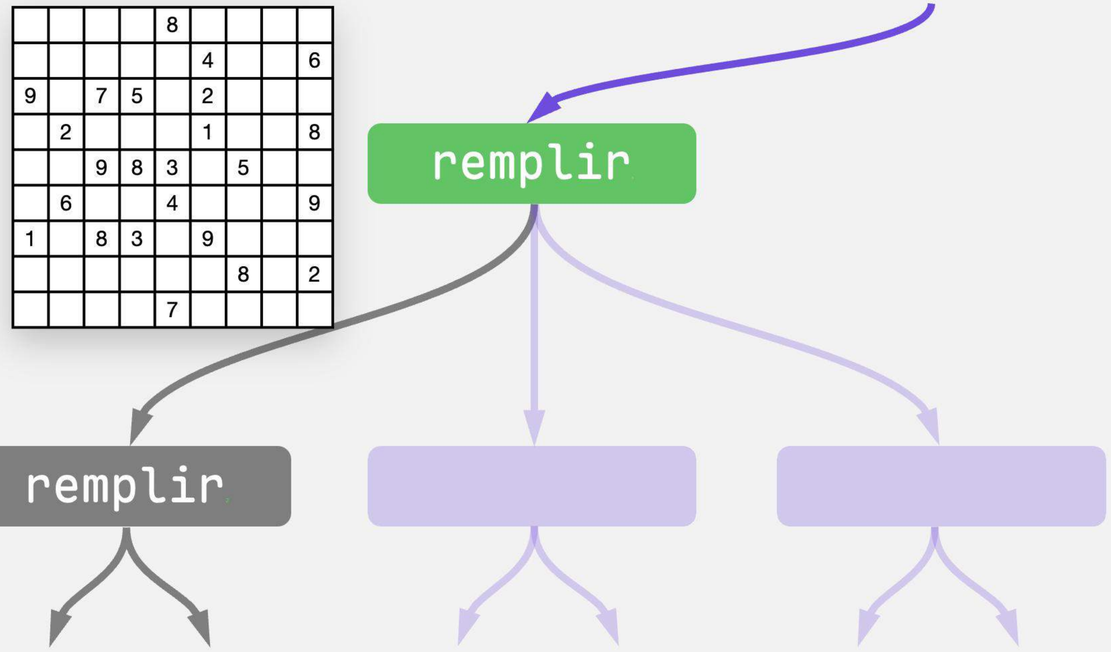
On suppose qu'on a exploré la descendance

Mais une seule grille suffit

Après l'appel, on **annule** l'action qu'on a faite !

Ici, **on vide la case.**

Et on poursuit…



On suppose qu'on a exploré la descendance

Cette technique est appelée

💕✨ **BACKTRACKING** ✨💕

ou **retour sur trace** en français

Code Python

Le **backtracking** est une stratégie de recherche **exhaustive** et **récursive** essai/erreur avec retour en arrière.

```python
def remplir(grille):
    x, y = chercher_case_vide(grille)
    if (x, y) == (None, None):
        afficher_grille(grille)
    else:
        chiffres = chiffres_possibles(grille, x, y)
        if not chiffres:
            return
        else:
            for chiffre in chiffres:
                grille[x][y] = chiffre
                remplir(grille)
                grille[x][y] = 0
```

On **fait** un choix

On poursuit la résolution avec ce choix

On **annule** le choix

Benchmark

Sur la grille d’exemple

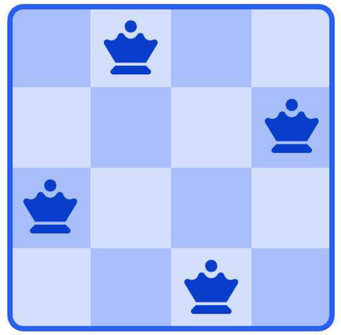
Technique	Durée d’exécution	Mémoire utilisée
`deepcopy`	2.91s	111.3 Ko
Backtracking	**0.3s**	**7.5 Ko**

Sur une grille très difficile

Technique	Durée d’exécution	Mémoire utilisée
`deepcopy`	72.97s	117.11 Ko
Backtracking	**8.36s**	**8.0 Ko**

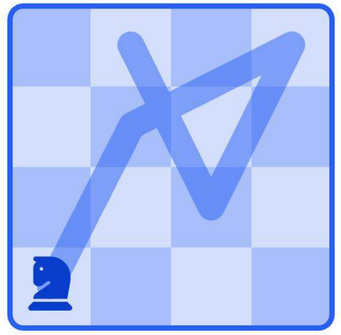
Backtracking, un jutsu sympa

Problème des **8 dames**



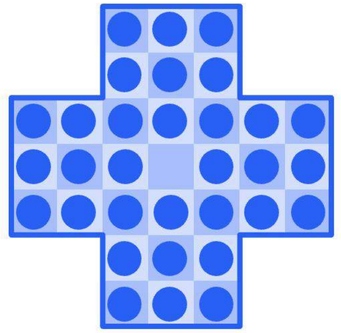
Comment placer 8 dames sur un échiquier 8 × 8 sans qu'elles puissent se menacer mutuellement ?

Problème du **cavalier**



Un cavalier est placé sur un échiquier 8 × 8 et doit visiter chaque case sans y repasser.

Résolution du **Solitaire (casse-tête)**



Si la case d'arrivée est vide, une bille peut sauter une bille adjacente et la retirer du plateau. Le but est d'en avoir plus qu'une seule.

Sortir d'un labyrinthe

Et si la technique de backtracking n'était rien d'autre que celle que vous utilisez pour sortir d'un labyrinthe instinctivement ?