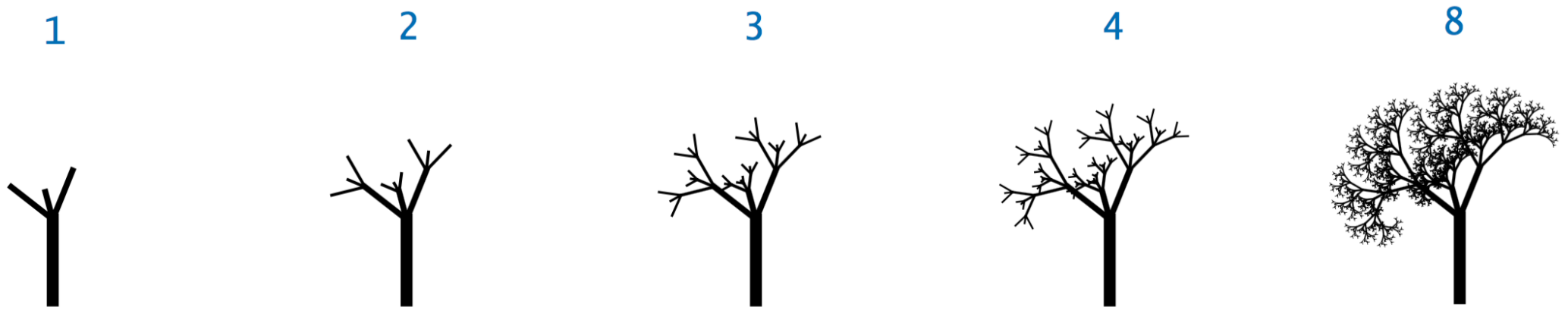
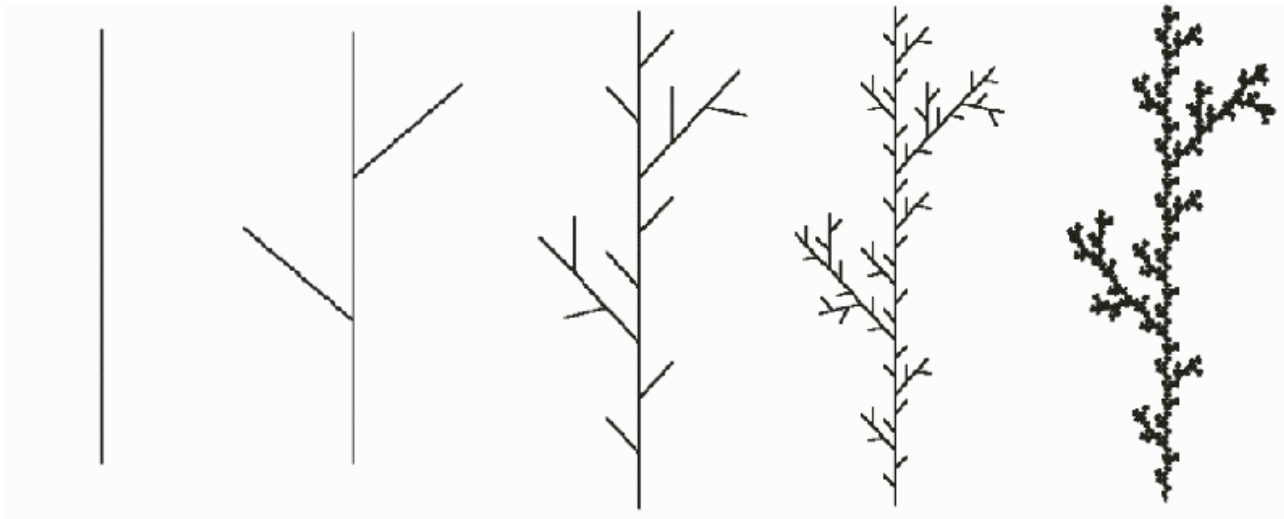
La récursivité

La récursivité dans la nature

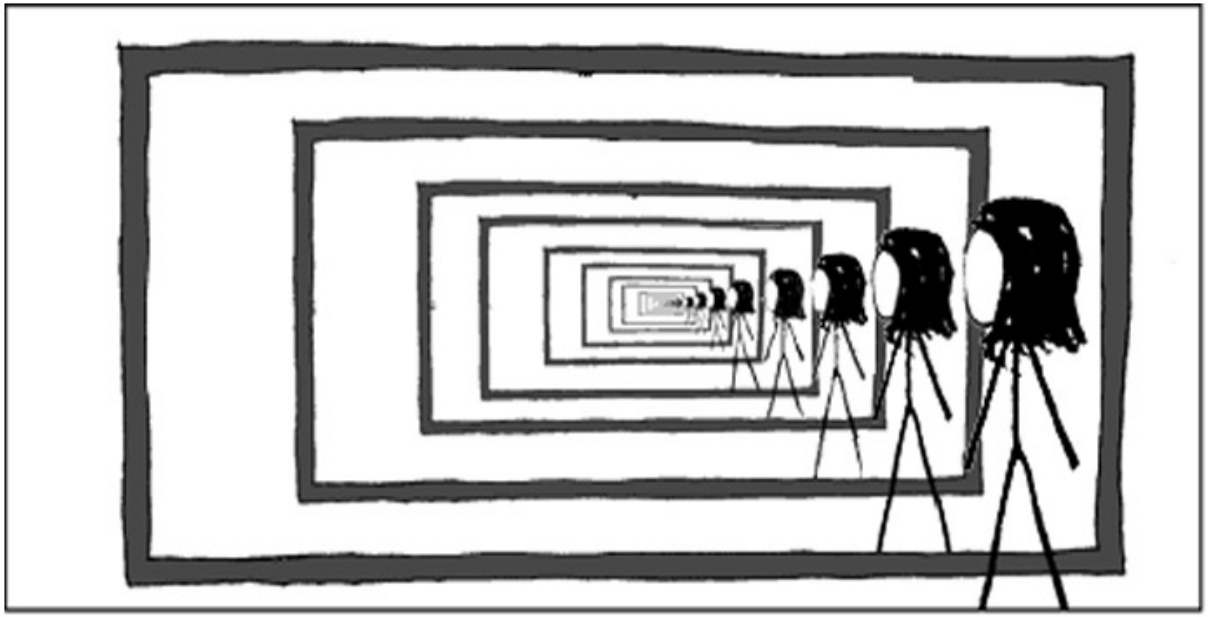
La récursivité dans la nature

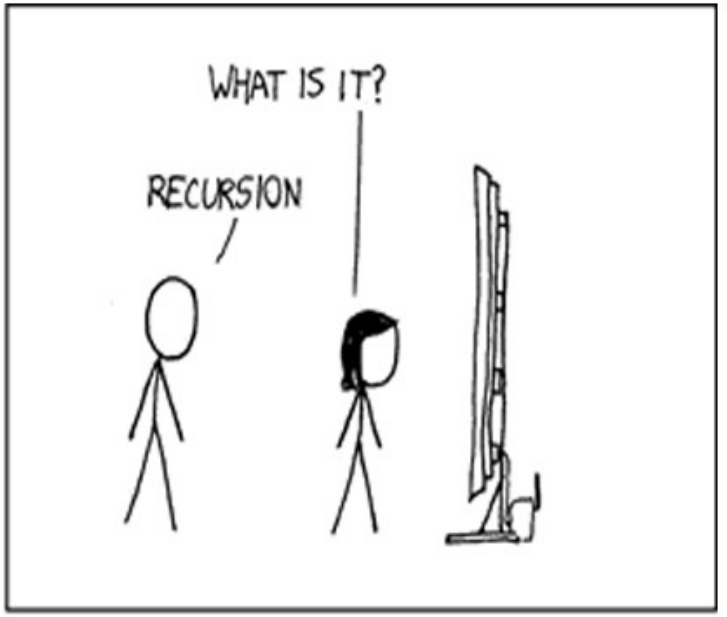




Le saviez-vous, il existe 2 types de personnes :

- **Celles qui comprennent la récursivité**
- **Et celles qui ne comprennent pas qu'il y a 2 types de personnes :**
  - Celles qui comprennent la récursivité
  - Et celles qui ne comprennent pas qu'il y a 2 types de personnes :
    - Celles qui comprennent la récursivité
    - Et celles qui ne comprennent pas qu'il y a 2 types de personnes :
      - …




WHAT IS IT?
RECURSION

Les 3 règles de la récursivité

Lors de l'écriture d'une fonction récursive, 3 règles doivent toujours être vérifiées :

- La fonction s'appelle elle-même.
- La fonction comporte un "cas de base" ou "cas terminal" qui correspond à une condition d'arrêt.
- L'algorithme conduit vers le cas de base : il n'y a pas une infinité d'appels récursifs.

Itératif vs Récursif

	Itératif	Récursif
Principe :	**Avec une boucle (un ensemble d'instructions exécutées de manière répétitive)**	**Avec un auto-appel (la fonction s'appelle elle-même)**
Utilisation :	Lorsque le nombre d'itérations est connu.	Lorsque le nombre d'itérations n'est pas défini. Pour manipuler des structures de données récursives, comme des listes chaînées ou des arbres.
Terminaison (fin) :	Lorsque la condition de l'itérateur cesse d'être vraie.	Dans le cas de base : où il n'y a pas d'auto-appel
Avantages :	La complexité temporelle est limitée.	Très petite longueur de code. L'espace mémoire utilisé est limité.
Inconvénients :	Plus grande longueur de code. Peut utiliser beaucoup d'espace mémoire.	La complexité temporelle peut devenir exponentielle.
En cas de répétition infinie :	S'arrête lorsque la mémoire est épuisée.	Crash (blocage du système)

Itératif vs Récursif : Calculer une somme

```python
def calcule_somme_iteratif(nombres:list)->int:
    somme = 0
    for i in range(0, len(nombres)):
        somme += nombres[i]
    return somme
```

```python
def calcule_somme_recursif(nombres:list)->int:
    if len(nombres) == 0:
        return 0
    else:
        valeur = nombres.pop(0)
        return valeur + calcule_somme_recursif(nombres)
```

Exécution récursive

```
calcule_somme_recursif( [1, 2, 3, 4, 5] )
      --> return 1 + calcule_somme_recursif( [2, 3, 4, 5] )
calcule_somme_recursif( [2, 3, 4, 5] )
      --> return 2 + calcule_somme_recursif( [3, 4, 5] )
calcule_somme_recursif( [3, 4, 5] )
      --> return 3 + calcule_somme_recursif( [4, 5] )
calcule_somme_recursif( [4, 5] )
      --> return 4 + calcule_somme_recursif( [5] )
calcule_somme_recursif( [5] )
      --> return 5 + calcule_somme_recursif( [] )
calcule_somme_recursif( [] )
      --> cas terminal: return 0
```

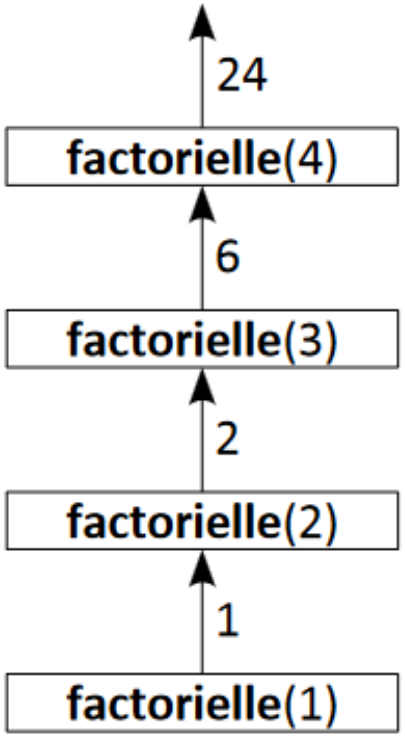
Exemple : calculer factorielle

- la première factorielle est 1

- pour passer de factorielle(*n*-1) à factorielle(*n*), je **multiplie par *n***

```
Algorithme factorielle
Entrées : entier n
Type de sortie : entier
Variable : entier resultat
Début
        Si n=1 alors :
                resultat ← 1
        sinon :
                resultat ← n x factorielle(n-1)
        Fin si
        renvoyer resultat
Fin
```

Trace de **factorielle**(4) :



Squelette de toute fonction récursive

```
def function f(param) :
    # Cas de base (terminal)
    if condition d'arrêt :
        return valeur
    # Cas récursif (récurrence)
    else :
        return valeur + f(param modifié)
```

En cas de crash système

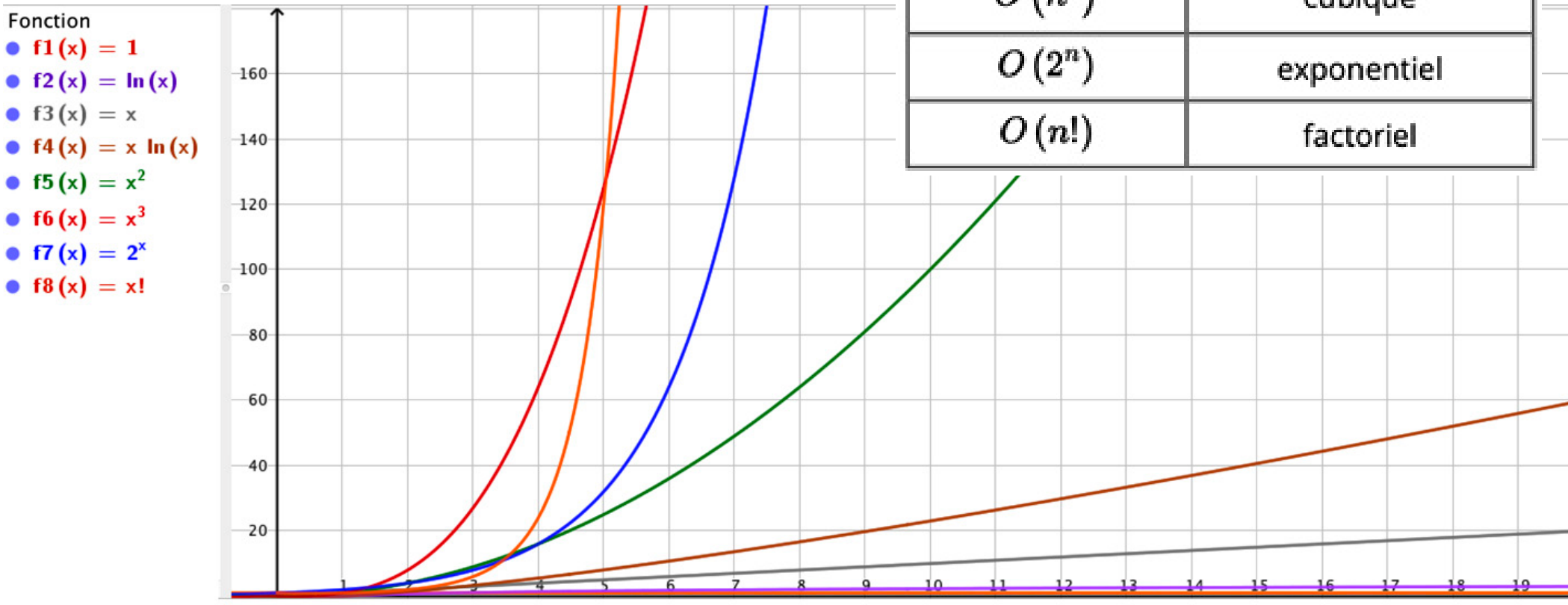
- Ouvrir un terminal
- Exécuter la commande suivante pour tuer le processus :

```
killall python
```

Complexité

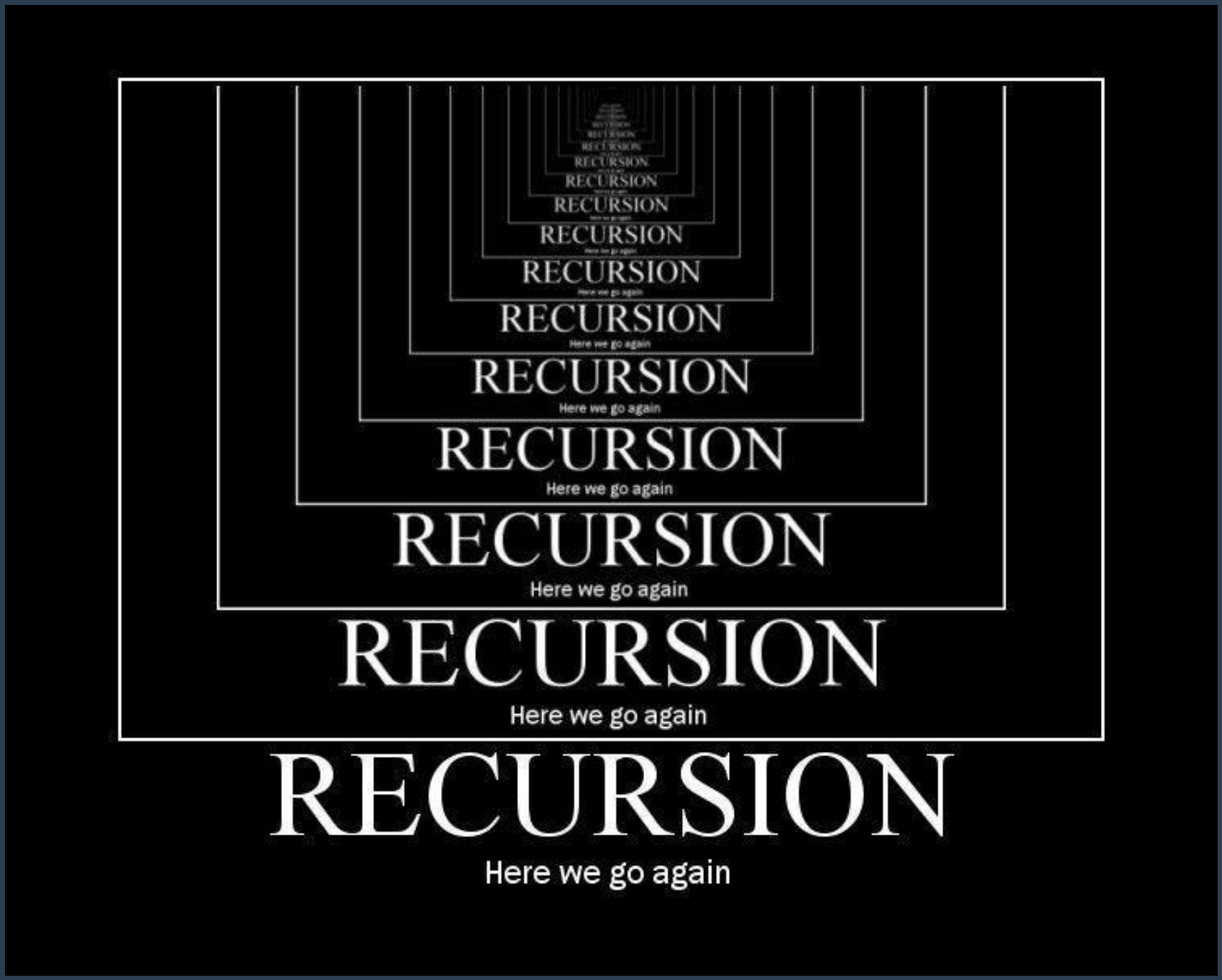
= quantité de ressources (en temps et en mémoire) dont a besoin un algorithme pour résoudre un problème

O	Type de complexité
$O(1)$	constant
$O(\log(n))$	logarithmique
$O(n)$	linéaire
$O(n \times \log(n))$	quasi-linéaire
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel
$O(n!)$	factoriel



Complexité

Temps	Type de complexité	Temps pour n = 5	Temps pour n = 10	Temps pour n = 20	Temps pour n = 50	Temps pour n = 250	Temps pour n = 1 000	Temps pour n = 10 000	Temps pour n = 1 000 000	Problème exemple
$O(1)$	complexité constante	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	accès à une cellule de tableau
$O(\log(n))$	complexité logarithmique	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns	recherche dichotomique
$O(n)$	complexité linéaire	50 ns	100 ns	200 ns	500 ns	2.5 µs	10 µs	100 µs	10 ms	parcours de liste
$O(n \log(n))$	complexité linéarithmique	40 ns	100 ns	260 ns	850 ns	6 µs	30 µs	400 µs	60 ms	tris par comparaisons optimaux (comme le tri fusion ou le tri par tas)
$O(n^2)$	complexité quadratique (polynomiale)	250 ns	1 µs	4 µs	25 µs	625 µs	10 ms	1 s	2.8 heures	parcours de tableaux 2D
$O(n^3)$	complexité cubique (polynomiale)	1.25 µs	10 µs	80 µs	1.25 ms	156 ms	10 s	2.7 heures	316 ans	multiplication matricielle naïve
$2^{\mathrm{poly}(n)}$	complexité exponentielle	320 ns	10 µs	10 ms	130 jours	10^{59} ans	...	...	...	problème du sac à dos par force brute
$O(n!)$	complexité factorielle	1.2 µs	36 ms	770 ans	10^{48} ans	...	...	...	...	problème du voyageur de commerce avec une approche naïve


RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again