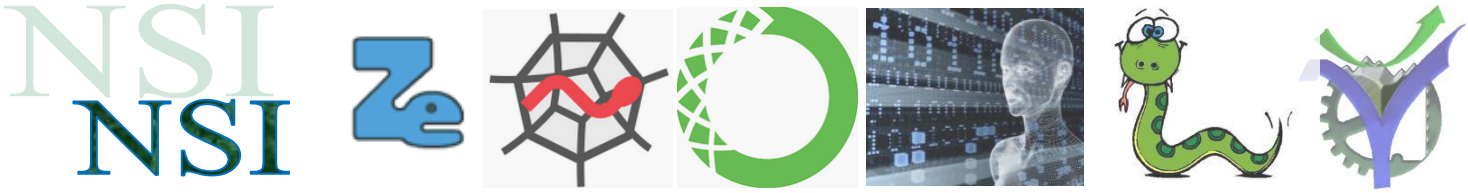
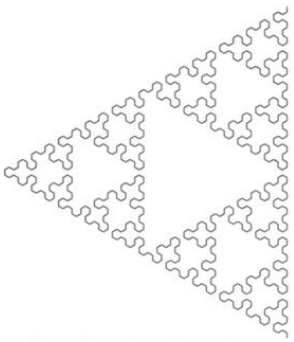


Programmation récursive



Résumé

Nous étudions dans ce cours le principe des algorithmes récursifs. Divers exemples seront étudiés et mis en œuvre pour en saisir le principe de fonctionnement.

La limitation posée par de très nombreux appels récursifs recalculant de nombreuses fois les mêmes valeurs sera contournée par le principe de la mémoïsation. Cette technique d'optimisation du temps de calcul d'un programme informatique se montrera particulièrement efficace dans le calcul des valeurs successives de la suite de Fibonacci.

Bonne lecture, bon travail.

Sommaire

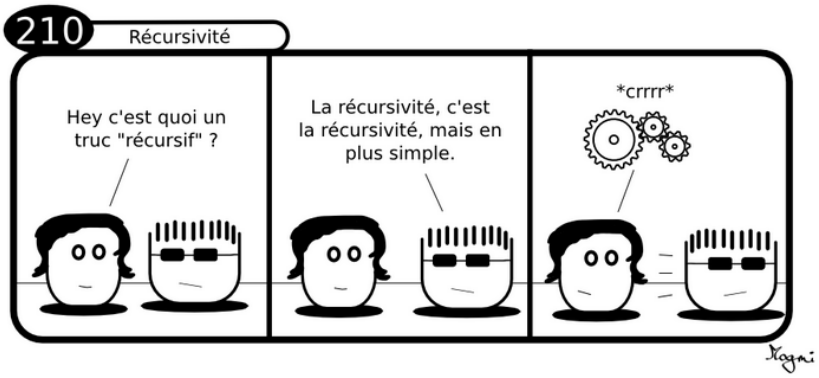


Illustration en provenance de http://igm.univ-mlv.fr/~dr/XPOSE2012/fonctionnelAvecScala/recursion.html

1 La récursivité introduction[1]

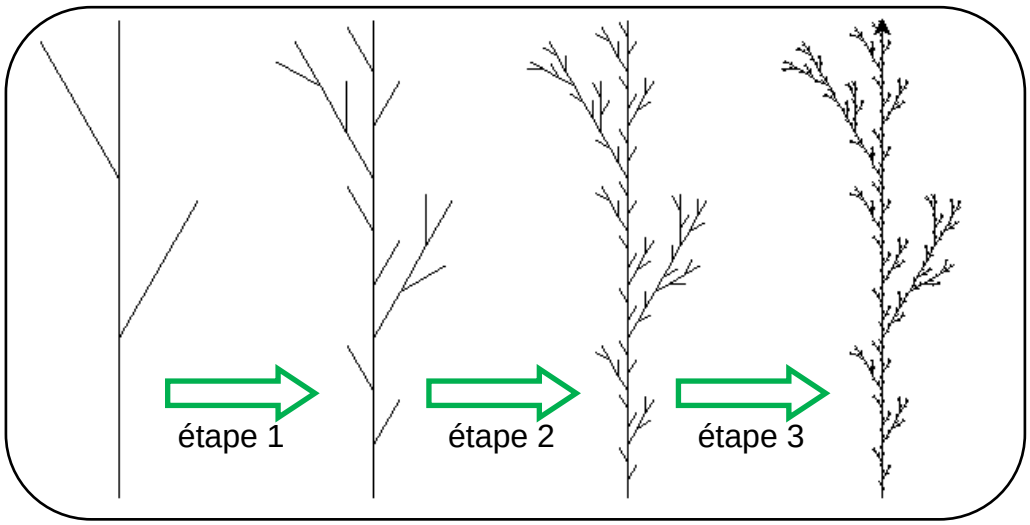
Un algorithme récursif est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème. L'approche récursive est un des concepts de base en informatique.

Les premiers langages de programmation qui ont autorisé l'emploi de la récursivité sont LISP (apparition en 1958, conçu par John Mc McCarthy) et Algol 60 (1960). Depuis, tous les langages de programmation généraux réalisent une implémentation de la récursivité.

On oppose généralement les **algorithmes récursifs** aux **algorithmes dits itératifs** qui s'exécutent sans appeler explicitement l'algorithme lui-même.

1.1 La récursivité quèsaco ?

Une illustration dans le domaine des fractals une démonstration[2] recursivite_avec_turtle.py



Quelques réflexions :

Q1. Décrivez ce qui est réalisé à chacune des étapes du processus.

Q2. Ce processus se termine t-il ?

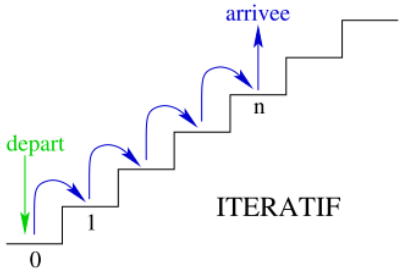
1.2 Comprendre les principes : une représentation en escaliers

Nous pouvons comparer les principes de fonctionnement des algorithmes itératifs et récursifs en utilisant une représentation sous forme d'escalier[3] .

Pour cela imaginons d'avoir besoin de calculer $a^n = a \cdot a^{n-1}$ $a^0 = 1$

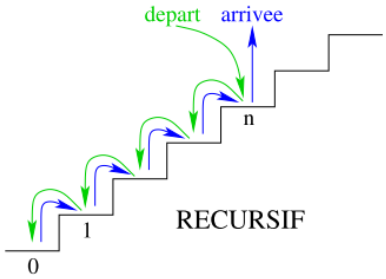
[1] https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif
[2] http://www.lacim.uqam.ca/~blondin/posts/fr/fractales.html
[3] Source S. Verel, M.-E. Voge, Algorithmes récursifs, www.i3s.unice.fr/~verel

Processus itératif



- départ : info connue
- monter vers résultat
- arêt *en haut* : *n*

Processus récursif



- départ : info cherchée
- descendre vers connue
- arêt *en bas* : 0
- monter vers résultat

Descend l'escalier

$a^n = a \cdot a^{n-1}$

Jusqu'à la première marche

$a^0 = 1$

Q3. Décrire les deux processus.

2 Formalisation mathématique[4]

Une formulation récursive d'une fonction est toujours constituée de plusieurs cas, parmi lesquels on distingue des cas de base et des cas récursifs du calcul.

- Les cas récursifs utilisent la fonction pour suivre un cheminement vers le résultat.
- Les cas de bases permettent de calculer directement une solution.

2.1 Un exemple la fonction factorielle

Cette fonction bien connue a pour expression : $n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$

Elle peut s'écrire : $n! = (n-1)! \times n$

La forme itérative du calcul s'écrit :

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

La forme récursive s'écrit :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Cas de base

Cas récursif

⇦

[4] Rédigé avec l'aide du chapitre 1 récursivité de Terminale NSI, 24 leçons avec exercices corrigés ellipses et Wikipédia https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif

2.2 Deuxième exemple la fonction x^n

Définition itérative

La fonction est définie par :

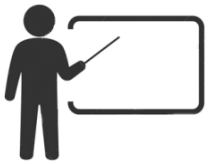
$$x^n = \underbrace{x \cdot x \cdot x \cdot x \quad \cdot x}_{\text{n fois}}$$

Définitions récursives

$$\text{puissance}(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ x \cdot \text{puissance}(x, n-1) & \text{si } n > 0. \end{cases}$$

$$\text{puissance}(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ x & \text{si } n = 1, \\ x \cdot \text{puissance}(x, n-1) & \text{si } n > 1. \end{cases}$$

Script_récursif_1. Coder la fonction définie ci-dessus à partir du script Puissance_de_x_recursif.py

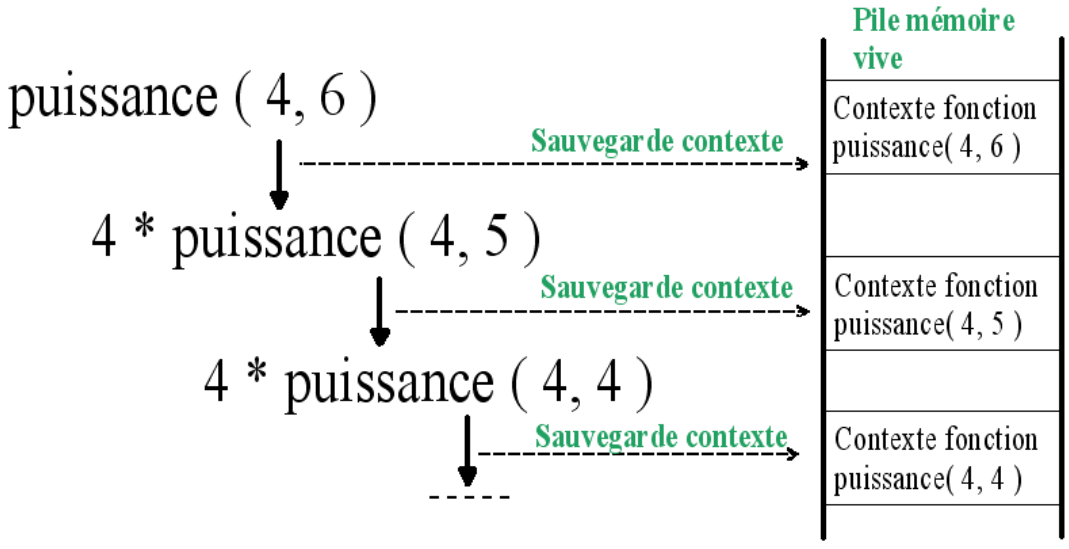


2.3 Fonctionnement de la récursion : l'arbre des appels

Le fonctionnement récursif impose d'enregistrer les données et l'état du calcul à l'étape n avant de passer à l'étape n-1. Ces enregistrements se font, quelque soit le système de calcul utilisé, dans une zone de mémoire vive particulière appelée : la pile.

Cette pile enregistre donc les contextes successifs de tous les appels ce qui permet au processus récursif de s'accomplir correctement et re revenir en arrière, en dépilant les appels, pour aboutir au résultat final.

Exemple de représentation de la pile d'appels



Chacun des appels à la fonction puissance nécessite d'enregistrer dans une zone de mémoire vive, spécialement dédié à cette usage, dénommée la pile, tout le contexte permettant d'effectuer ultérieurement le retour d'appel et de revenir au premier appel avec le résultat des calculs.

Chaque appel récursif consomme donc de la mémoire ce qui constitue une limite à une utilisation intensive.

En python le message d'erreur obtenu en cas de dépassement est le suivant :

```
RuntimeError: maximum recursion depth exceeded while calling a Python object
```

Cette limite peut être repoussée par :

```
import sys
sys.setrecursionlimit(2000)
```

```
Le résultat x**n avec n = 1500
x =  24 =>   x**n = 20742570144649314370373242272126130383051784096889760048646231737140734505526691535469066893168087542549542206847008649577385991552775241379349653054811060735184686826191295896506905833877744878879802531737697554894676051702057844724953184801463324302277541711789806606118046997969022700183859158629755025093918234163028228988007589964077131452236801269685363700699134888852310358190046853599954416866197939313792550328063209036023416295683743297864386328183272182014294377532771885115362268672187810109224018631321195733199710579442810837587335040181439040047647044155031183403707565189771775377699588628737983979860192775971525275848493044822403969397523861704079402583902111597034750764995268263790064537325384329321505796905237883662986142822251466461894437019370322291692427093860228415232649061354477121498265697893119103669350955553041026808069652303790498390989641211136906378481349134910697267793226986417759035113923215788475732954434864744822248945458216438109064157253387836196149161264046408994004894968260744268610583817000723437238740393437839130198585034542512530679923053548359296662241186783313154148207600426439520497013987746597526278952278716285595480096812925326061625712307468910538482483266152139731682503977027799621943285920462836632306622651555668213524587013443106702213651119904018972999057836535760200971121260247452172113997746135353002665651563126371082460565748835429659353422981924261651984294048175047228323090367242087574105398714261455012526551844977156097343160712651789794481137518518266459412849257891728770955205844549263761409810180815598871786779225896370897960762964256018793437029951757675937054374531162501620663133265601938822375507030743497491323365047666755059295676740329122718089110870776175474904852792662752812942396812936341568641923799437075580908594807318840995574013708684268860143592279143793358309505831814676022743038142349109593476560463368952908199381487557565989172650090737436924788538166870701964814668510871997076223384785533692753481438175667789398249034059095209047289350158633191886676612456655093 76
```

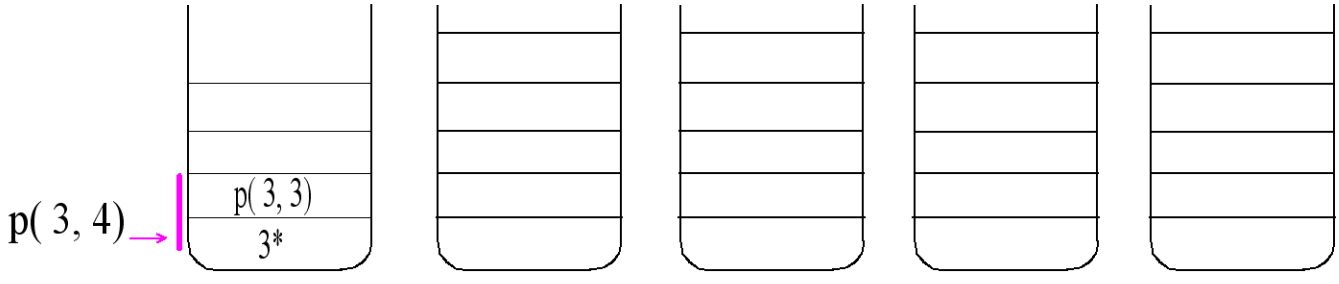
Exemple calcul de 24^{1500}

Le résultat entier est exact en python.

2.4 Exercice

Q4. Décrire sur la représentation graphique les appels nécessaires pour le calcul de puissance(3,4). Pour simplifier on notera puissance(x,n) par p(x,n). La définition utilisée est rappelée ci-contre :

$$\text{puissance}(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ x \cdot \text{puissance}(x, n-1) & \text{si } n > 0. \end{cases}$$



3 Diviser pour régner : l'exponentiation rapide

3.1 Définition récursive améliorée de la fonction puissance

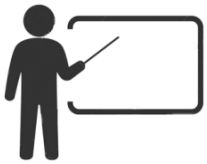
En utilisant les propriétés :

$x^n = (x^{n/2})^2$ pour n pair et $x^n = x \cdot (x^{(n-1)/2})^2$ pour n impair

On peut définir récursivement x^n avec des cas récursifs multiples :

$$\text{puissance}(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ \text{carre}(\text{puissance}(x, n/2)) & \text{si } n \geq 1 \text{ et n pair} \\ x \cdot \text{carre}(\text{puissance}(x, (n-1)/2)) & \text{si } n \geq 1 \text{ et n impair} \end{cases}$$

Script_récursif_2. Coder la fonction améliorée définie ci-dessus à partir du script Puissance_de_x_recursif_ameliore.py



3.2 Analyse de résultats

Q5. Commentez les résultats ci-dessous :

```
>>> (executing lines 1 to 73 of "Puissance_de_x_recursif.py")
Calcul récursif de x**n : calcul de base
Durée du calcul des 10000 termes   0.490849 secondes
Résultat élévation à la puissance  100

>>> (executing lines 1 to 80 of "Puissance_de_x_recursif_ameliore.py")
Calcul récursif de x**n : calcul amélioré
Durée du calcul des 10000 termes   0.066647 secondes
Résultat élévation à la puissance  100


>>> (executing lines 1 to 73 of "Puissance_de_x_recursif.py")
Calcul récursif de x**n : calcul de base
Durée du calcul des 1000000 termes   4.867256 secondes
Résultat élévation à la puissance  10

>>> (executing lines 1 to 80 of "Puissance_de_x_recursif_ameliore.py")
Calcul récursif de x**n : calcul amélioré
Durée du calcul des 1000000 termes   3.127337 secondes
Résultat élévation à la puissance  10
```

3.3 Comparaison des deux algorithmes en terme d'appels

Si nous comparons le fonctionnement des deux fonctions récursives décrites script_récursif_1 et script_récursif_2, en y ajoutant un traceur pour compter le nombre d'itérations nous observons les résultats ci-dessous :

```
Calcul récursif 'de base'
Itération N° 20
Itération N° 19
Itération N° 18
Itération N° 17
Itération N° 16
Itération N° 15
Itération N° 14
Itération N° 13
Itération N° 12
Itération N° 11
Itération N° 10
Itération N° 9
Itération N° 8
Itération N° 7
Itération N° 6
Itération N° 5
Itération N° 4
Itération N° 3
Itération N° 2
Itération N° 1
Itération N° 0
```

```
Calcul récursif 'amélioré'
Itération N° 20
Itération N° 10
Itération N° 5
Itération N° 2
Itération N° 1
```

Bien entendu les résultats sont identiques dans les deux cas :

```
Le résultat x**n avec n = 20
x =  24 =>   x**n = 4019988717840603673710821376
```

Q6. Vérifier que dans le cas amélioré le nombre d'appels au total de la fonction puissance est égal à $1 + \lfloor \log_2(n) \rfloor$

4 Pour soulager la pile : la mémoïsation[5]

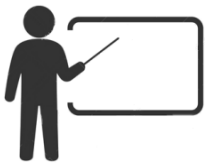
Nous allons illustrer un principe utilisé en programmation qui s'appelle la mémoïsation. Ce principe consiste à mémoriser des résultats dans les appels récursifs successifs ce qui permet d'économiser du temps de calcul. Nous l'illustrons sur un exemple : le calcul de la suite de Fibonacci.

4.1 La suite de Fibonacci

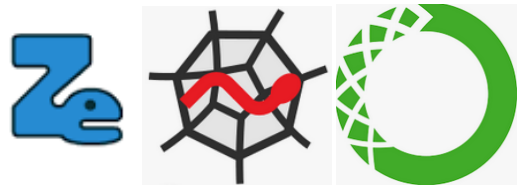
Cette suite est définie de la manière suivante : $\mathcal{F}_0=0$, $\mathcal{F}_1=1$, $\mathcal{F}_n=\mathcal{F}_{n-1}+\mathcal{F}_{n-2}$ pour n>1
Nous avons ici une double récursion.

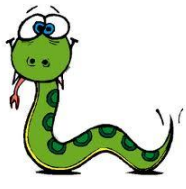
$$\text{Fibonacci}(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ \text{Fibonacci}(n-2) + \text{Fibonacci}(n-1) & \text{si } n > 1. \end{cases}$$

Script_récursif_3. Coder la fonction de Fibonacci en complétant le script Fibonacci-Simple-depart-eleve.py



[5] Paragraphe rédigé avec https://www.youtube.com/watch?v=Qk0zUZW-U_M en complément des ressources précédentes.






Résultats obtenus :

```
Calcul de la suite de Fibonacci

Durée du calcul des   34 termes     10.802 secondes
```

4.2 La mémoïsation première méthode

Nous allons mémoriser les résultats dans un dictionnaire au fur et à mesure de leurs calculs. Comme la suite de Fibonacci fait un appel intensif à des valeurs déjà calculées le gain en temps de calcul devrait être important.

Script_récursif_4. Modifier votre script n°3 pour y ajouter la mémoïsation.

Vous noterez la déclaration du dictionnaire en dehors de la fonction.

```
fibonacci_cache = {}

## Votre fonction à réaliser
def fibonacci(n):

    if n in fibonacci_cache:
        return fibonacci_cache[n]

    if n == 0:
        value = 0
    elif n == 1:
        value = 1
    elif n == 2:
        value = 1
    else:
        value = fibonacci(n-2) + fibonacci(n-1)

    fibonacci_cache[n] = value

    return value
```

Résultats obtenus :

```
Calcul de la suite de Fibonacci : mémoïsation exemple 1

Durée du calcul des   34 termes     0.071 millisecondes
```

4.3 La mémoïsation deuxième possibilité

Python possède une bibliothèque qui effectue automatiquement la mémorisation des résultats intermédiaires des fonctions et les dépose dans un cache.

```
from functools import lru_cache

## Votre fonction à réaliser
@lru_cache(maxsize = 3000)
def fibonacci(n):
    '''
    Mise en oeuvre du cache lru
    Statistiques à posteriori avec la commande
    >>>fibonacci.cache_info()
    CacheInfo(hits=1994, misses=1000, maxsize=1000, currsize=1000)
    '''
    if n == 0:
        return 0
    elif n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        return fibonacci(n-2) + fibonacci(n-1)
```

Les résultats obtenus :

```
Calcul de la suite de Fibonacci : mémoïsation exemple 2 Lru cache

Durée du calcul des   34 termes      0.510 millisecondes
```

Q7. Résumer le principe de fonctionnement et les effets de la mise en œuvre de la mémoïsation dans un programme.

5 Diviser pour régner : le tri fusion

5.1 Introduction

En informatique, le tri fusion est un algorithme de tri par comparaison stable. Sa complexité temporelle pour une entrée de taille n est de l'ordre de n log n, ce qui est asymptotiquement optimal. Ce tri est basé sur la technique algorithmique « diviser pour régner ».

L'opération principale de l'algorithme est la fusion, qui consiste à réunir deux listes triées en une seule. L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire.

Q8. Donner la signification d'un tri stable, puis d'un tri en place.

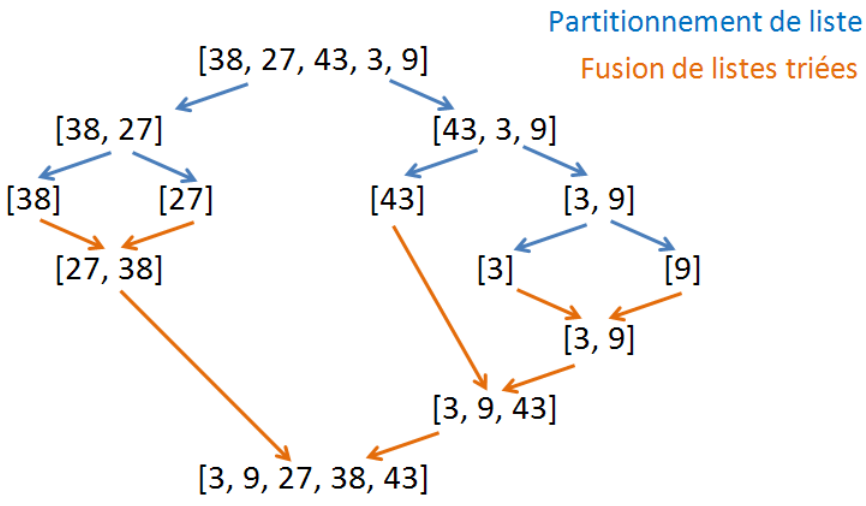
Principe de fonctionnement[6]

L'algorithme est naturellement décrit de façon récursive.

1. Si le tableau n'a qu'un élément, il est déjà trié.
2. Sinon, séparer le tableau en deux parties à peu près égales.
3. Trier récursivement les deux parties avec l'algorithme du tri fusion.
4. Fusionner les deux tableaux triés en un seul tableau trié.

Description du fonctionnement

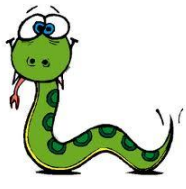
Le graphique ci-contre représente le tri fusion de la liste [38, 27, 43, 3, 9].



Q9. Résumez le fonctionnement de l'algorithme.

[6] https://fr.wikipedia.org/wiki/Tri_fusion






Q10. Décrire par un graphique le tri fusion de la liste :

[100, -2, 4, 83, 12, 28,-3]

5.2 Fusion de listes triées

Description de l'algorithme

On peut facilement construire une liste triée comportant les éléments issus de ces deux listes (leur *fusion*). Le plus petit élément de la liste à construire est soit le plus petit élément de la première liste, soit le plus petit élément de la deuxième liste. Ainsi, on peut construire la liste élément par élément en retirant tantôt le premier élément de la première liste, tantôt le premier élément de la deuxième liste.

Pseudo code de l'algorithme[7]

```
fonction fusion(A[1, …, a], B[1, …, b])
        si A est le tableau vide
                renvoyer B
        si B est le tableau vide
                renvoyer A
        si A[1] ≤ B[1]
                renvoyer A[1] ⊕ fusion(A[2, …, a], B)
        sinon
                renvoyer B[1] ⊕ fusion(A, B[2, …, b])
```

Le symbole ⊕ désigne ici la concaténation de tableaux. (L'opération + avec des listes en python)

5.3 Le tri fusion

En utilisant l'algorithme Fusion de listes triées nous pouvons décrire le pseudo code du tri fusion :

Pseudo code de l'algorithme[8]

```
fonction triFusion(T[1, …, n])
si n ≤ 1
        renvoyer  T
sinon
        renvoyer
                fusion (  triFusion(T[1, …, n/2]) , triFusion(T[n/2 + 1, …, n]) )
```

[7] Voir https://fr.wikipedia.org/wiki/Tri_fusion
[8] Voir https://fr.wikipedia.org/wiki/Tri_fusion