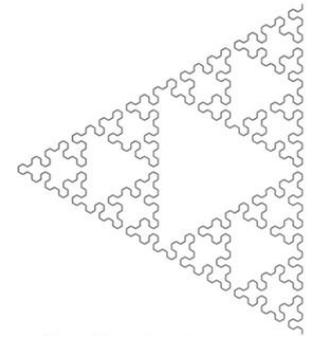




Programmation récursive



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 mémorisation. 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

1	La récursivité introduction	2
1.1	<i>La récursivité quèsaco ?.....</i>	2
1.2	<i>Comprendre les principes : une représentation en escaliers</i>	2
2	Formalisation mathématique.....	3
2.1	<i>Un exemple la fonction factorielle</i>	3
2.2	<i>Deuxième exemple la fonction x^n.....</i>	3
2.3	<i>Fonctionnement de la récursion : l'arbre des appels.....</i>	4
2.4	<i>Exercice.....</i>	5
3	Diviser pour régner : l'exponentiation rapide.....	5
3.1	<i>Définition récursive améliorée de la fonction puissance</i>	5
3.2	<i>Analyse de résultats.....</i>	5
3.3	<i>Comparaison des deux algorithmes en terme d'appels.....</i>	6
4	Pour soulager la pile : la mémorisation	6
4.1	<i>La suite de Fibonacci.....</i>	6
4.2	<i>La mémorisation première méthode</i>	7
4.3	<i>La mémorisation deuxième possibilité</i>	7
5	Diviser pour régner : le tri fusion	8
5.1	<i>Introduction</i>	8
5.2	<i>Fusion de listes triées</i>	9
5.3	<i>Le tri fusion.....</i>	9



1 La récursivité introduction¹

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.

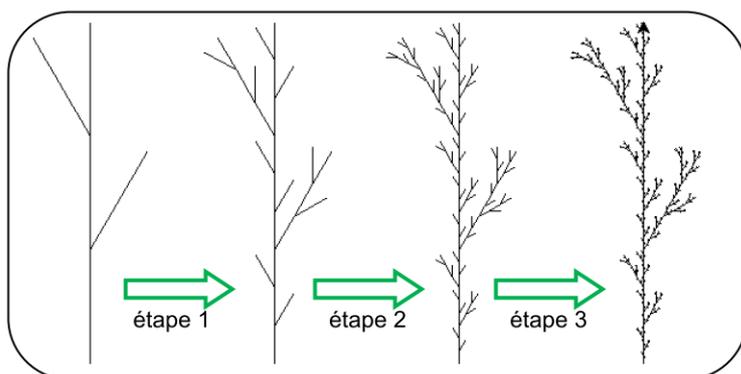
1.1 La récursivité quèsaco ?

Une illustration dans le domaine des fractals une démonstration²

 `recursivite_avec_turtle.py`

Quelques réflexions :

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



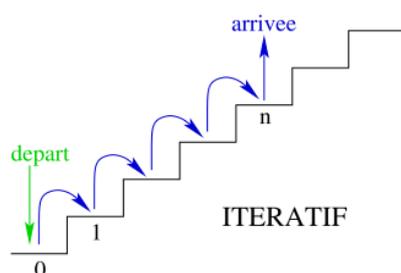
Q2. Ce processus se termine-t-il ?

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'escalier³.

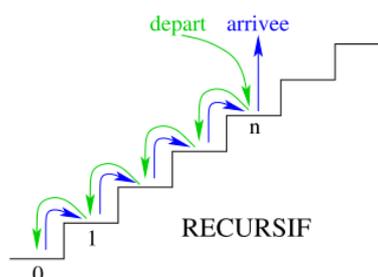
Pour cela imaginons d'avoir besoin de calculer $a^n = a \cdot a^{n-1}$ $a^0 = 1$

Processus itératif



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

Processus récursif



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

Descend l'escalier

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

Jusqu'à la première marche

$$a^0 = 1$$

 Décrire les deux processus.



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

2 Formalisation mathématique⁴

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

- Les cas récurifs 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 : $n! = 1 \times 2 \times 3 \times \dots \times (n - 1) \times n$

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

La forme itérative du calcul s'écrit :

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

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 récurif

2.2 Deuxième exemple la fonction x^n

Définition itérative

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

La fonction est définie par :

Définitions récursives

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



Script_récurif_1. Coder la fonction définie ci-dessus
à partir du script  Puissance_de_x_recurif.py



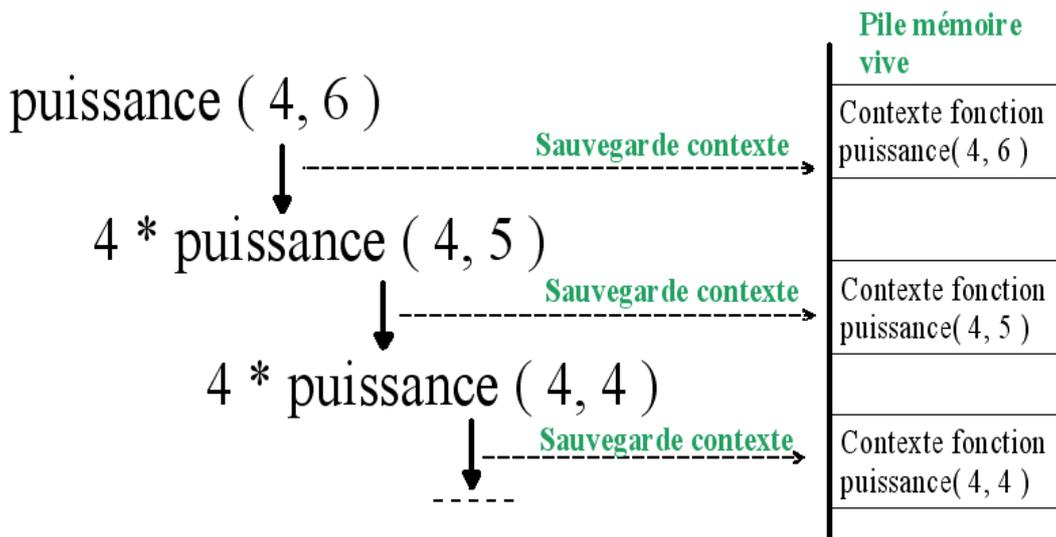
⁴ Rédigé avec l'aide du chapitre 1 récursivité de Terminale NSI, 24 leçons avec exercices corrigés ellipses et Wikipédia https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9curif

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 de 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 = 207425701446493143703732422721261303830517
31737140734505526691535469066893168087542549542206847008649577
3496530548110607351846868261912958965069058338774487887980253
17020578447249531848014633243022775417117890866061180469979690
5502509391823416302822898007589964077131452236801269685363700
19004685359995441686619793931379255032806320903602341629568374
21820142943775327718851153622686721878101092240186313211957331
87335040181439040047647044155031183403707565189771775377699588
77597152527584049304402240396939752306170487948258390211159703
0064537253843293215057969052378836629061428222514664618944370
9386022841523649061354477121498265697893119103669350955530416
49839098964121113690637848134913491069726779322698641775903511392321578847573295
4434864744822489454582164381090641572533878361961491612640464089940048949682607
44268610583817000723437238740393437839130190585034542512530679923053548359296602
24118678331315414820760042643952049701398774659752627895227871628559548009681292
53260616257123074689105384824832661521397316825039770277996219432859204628366323
06622651555668213524587013443106702213651119904018972999057836535760200971121260
24745217211399774613535300266565156312637108246056574883542965935342298192426105
19042040481750472283230903672420875741053987142614559125265518449771569973431607
12651789789448113751851826645941284025789172877095520504454926376140081018081559
58871786779225896370897960762964256018793437029951757675937054374531162501620663
1332656019388223755070307434974913233650476667550529567674032912271880911087077
61754749048527926627528129423968129363415686419237994370755808908594807318840995
57401370868426860014359227914379335830950583181467602274303814234910959347656046
336895290819938148755756509891726500907374369247885381668707019648146685108719970
76223384785533692753481438175667789398249034059095209047289350158633191886676612
45665509376
```

Exemple calcul de 24^{1500}

Le résultat entier est exact en python.

Trop fort Python !!

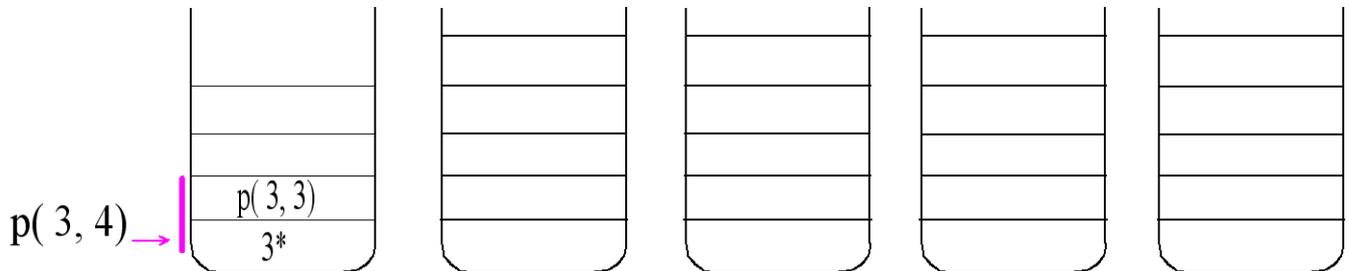


2.4 Exercice



Q3. 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 :

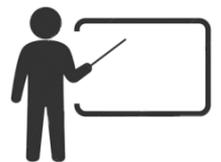
$$x^n = (x^{n/2})^2 \text{ pour } n \text{ pair et } x^n = x \cdot (x^{(n-1)/2})^2 \text{ pour } n \text{ impair}$$

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

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



Script_récurif_2. Coder la fonction améliorée définie ci-dessus à partir du script [Puissance_de_x_recurif_ameliore.py](#)



3.2 Analyse de résultats



Q4. Commentez les résultats ci-dessous :

```
>>> (executing lines 1 to 73 of "Puissance_de_x_recurif.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_recurif_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_recurusif.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_recurusif_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_récurusif_1 et script_récurusif_2, en y ajoutant un traceur pour compter le nombre d'itérations nous observons les résultats ci-dessous :

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

```
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
```



Q5. 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 mémoïsation⁵

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.

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



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



Script_récuratif_3. Coder la fonction de Fibonacci en complétant le script  Fibonacci-Simple-depart-eleve.py



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_récuratif_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, currsz=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

 **Q6.** 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.

 **Q7.** Donner la signification d'un tri stable, puis d'un tri en place.

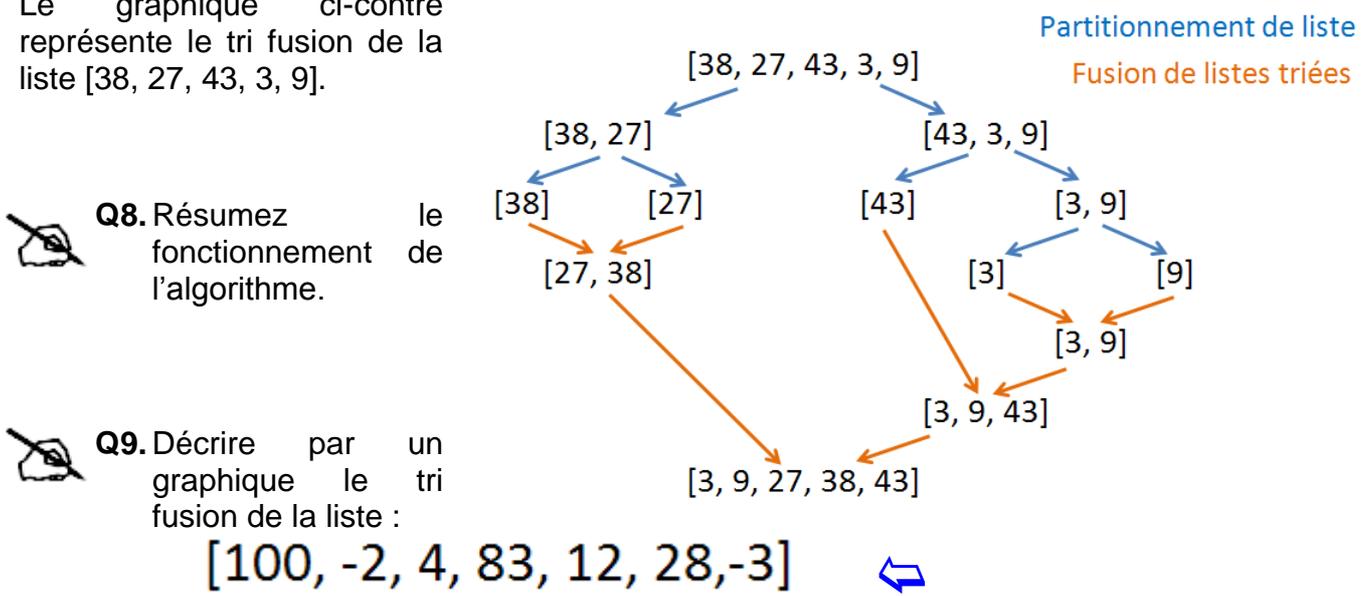
Principe de fonctionnement⁶

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].



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

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'algorithme⁷

```
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 \oplus 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'algorithme⁸

```
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