# Algorithmique et programmation

## Cours d'informatique de PSI

P.E LEROY

11 juin 2019

Cette œuvre est mise à disposition sous licence
*Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 3.0 non transposé*

Pour voir une copie de cette licence, visitez
`http://creativecommons.org/licenses/by-nc-sa/3.0/`
ou écrivez à
`Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.`

# Table des matières

**Avant-propos** **3**

**Exemples d'algorithmes simples** **4**

**Les piles** **6**

**La complexité d'un algorithme** **9**

**La récursivité** **12**
La récursivité . . . . . 12
Exemple de calcul récursif . . . . . 13
Concevoir un algorithme récursif . . . . . 15
Complexité d'un algorithme récursif . . . . . 16

**Le tri d'un tableau** **17**
Position du problème . . . . . 17
Le tri par insertion . . . . . 18
Le tri rapide . . . . . 20
Le tri fusion . . . . . 23
Coûts comparés des algorithmes de tri . . . . . 26

**Bibliographie** **29**

# Avant-propos

Ce fascicule comporte une partie des cartes mentales présentées en cours d'informatique durant le premier semestre de seconde année PSI.

Ce cours est un approfondissement de la partie algorithmique et programmation de la classe de première année de CPGE. Le choix de son orientation est avant tout pragmatique, issu de la volonté de présenter les concepts de manière simple. Le langage choisi pour illustrer ces notions est le Python, dans sa version 3.

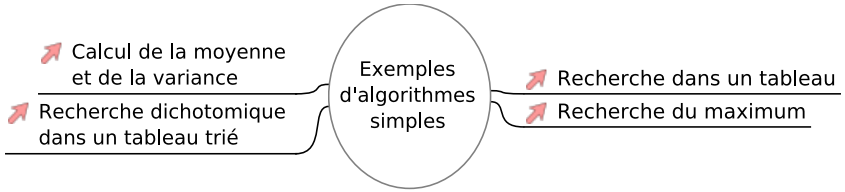
Les cartes mentales, utilisées comme support de pésentation, ont pour originalité de présenter les notions de manière éclatée, et non linéaire comme dans un cours classique. Elles permettent de relier efficacement les notions entre elles, et d'avoir en permanence une vue globale de l'ensemble des notions abordées.

Elles s'utilisent donc comme des aides à la prise de note, permettant ainsi aisément de les compléter et de les agrémenter.

Ces cartes sont régulièrement mises à jour, complétées et corrigées, n'hésitez pas à me signaler toute erreur. Vous pouvez télécharger la dernière version sur mon blog `http://psi-pel.blogspot.fr`.

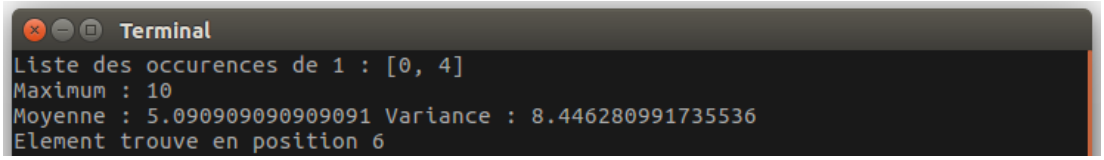
Je vous souhaite bon apprentissage !

Pierre-Emmanuel LEROY
Professeur de Physique-Chimie et Informatique en PSI
`http://psi-pel.blogspot.fr`


Exemples d'algorithmes simples
Calcul de la moyenne et de la variance
Recherche dichotomique dans un tableau trié
Recherche dans un tableau
Recherche du maximum

```python
# Exemples d'algorithmes
liste=[1,6,2,5,1,4,5,10,5,8,9]

# Recherche de toutes les occurences d'un element dans un tableau
def recherche_element(element,liste):
  liste_pos=[]
  for i in range(len(liste)):
    if liste[i]==element:
      liste_pos.append(i)
  print('Liste des occurences de',element,':',liste_pos)
  return(liste_pos)
recherche_element(1,liste)


# Recherche du maximum dans un tableau
def recherche_max(liste):
  maximum=liste[0]
  for element in liste:
    if element>maximum:
      maximum=element
  print('Maximum :',maximum)
  return(maximum)
recherche_max(liste)

# Calcul de la moyenne et de la variance des valeurs d'un tableau
def calcul_moyenne_variance(liste):
  somme=0
  somme_quad_ecarts=0
  for element in liste:
    somme=somme+element
  moyenne=somme/len(liste)
  for element in liste:
    somme_quad_ecarts=somme_quad_ecarts+(element-moyenne)**2
  variance=somme_quad_ecarts/len(liste)
  print('Moyenne :',moyenne,'Variance :',variance)
  return(moyenne,variance)
calcul_moyenne_variance(liste)

# Recherche dichotomique dans un tableau trie
liste=[1,2,4,5,7,9,11,12,14,18,19]
def recherche_dichotomie(element,liste):
  g=0
  d=len(liste)-1
  while g<=d:
    m=(g+d)//2
    if liste[m]==element:
      print('Element trouve en position',m)
      return(m)
    if liste[m]<element:
      g=m+1
    else:
      d=m-1
  print('Element non trouve')
  return(None)
recherche_dichotomie(11,liste)
```

../cartes_mentales/algorithmique_programmation/exemples_algorithmes.py



- Les piles
  - Pourquoi une nouvelle structure ?
    - On a déjà vu deux stuctures de données, qui ne répondent pas aux mêmes besoins.
      - Listes
      - Tuples
    - Les << piles >> sont une structure différente répondant à des besoins particuliers.
  - Structure de << piles >>
    - Elle correspond à l'image d'une pile, d'assiettes, de cartes, etc...

    -

      - Un élément situé << en bas >> ne sera accessible que si l'on << dépile >> les éléments du dessus.
      - Un nouvel élément << s'empile >> sur le haut de la pile.
      - Le premier élément sorti est le dernier arrivé (<< Last In First Out >> ou LIFO.
    - Quand l'utiliser ?
      - **Lorsque l'on peut ou que l'on doit accéder uniquement au dernier élément stocké**
      - **Lorsque l'on veut stocker des éléments sans en connaître le nombre à l'avance**
      - Dans les autres cas, un tableau sera le plus souvent plus approprié.
    - Exemples
      - Historique des actions effectuées
      - Historique des parenthèses utilisées dans une opération
      - Etc...
    - Remarque : il existe une structure proche, la struture de << file >> s'apparentant à une file d'attente. Le premier élément sorti est le premier arrivé (<< First In First Out >> ou FIFO).
  - Opérations associées à une structure de pile
    - `creer_pile()` Renvoie une nouvelle pile vide
    - `depiler(p)` Dépile et renvoie le sommet de la pile `p`
    - `empiler(p,v)` Empile la valeur `v` sur le sommet de la pile `p`
    - `taille(p)` Renvoie le nombre d'éléments de la pile `p`
    - `est_vide(p)` Indique si la pile `p` est vide
    - `sommet(p)` Renvoie sans dépiler le sommet de la pile `p`
  - Exemples
    - Analyse du parenthésage d'une expression arithmétique
    - Gestion d'un historique de commandes
  - Ce qui se passe derrière...

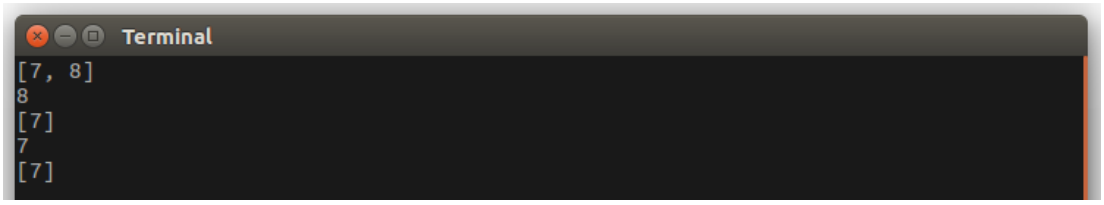
En python, les listes sont en réalité des tableaux redimensionnables. Lors de l'ajout d'un nouvel élément, soit le tableau est assez grand pour l'acceuillir, soit **sa taille est directement doublée**. Ainsi une liste sera successivement, au cours de son extension, représentée par un tableau de dimension 1, 2, 4, ..., 2^k.

L'avantage se situe dans le coût moyen de l'opération. Augmenter la taille d'un tableau demande la recopie des valeurs présentes dans le nouveau tableau. Augmenter la taille d'une unité à chaque fois est plus couteux en performances que de doubler la taille lorsque c'est nécessaire.

  - Réalisation d'une structure de pile
    - Principe
      - On utilise une liste pour simuler une pile.
      - On utilise les méthodes de listes pour réaliser les opérations associées.
    - Procédure
      - On créé une liste vide `[]`.
      - On ajoute ou enlève des éléments à volonté à l'aide des méthodes `append()` et `pop()`.
      - On détermine la taille à l'aide de la fonction `len`.
      - On vérifie que la liste est vide en déterminant sa dimension.
      - Le sommet est déterminé en utilisant l'indice `-1`.
    - Codage

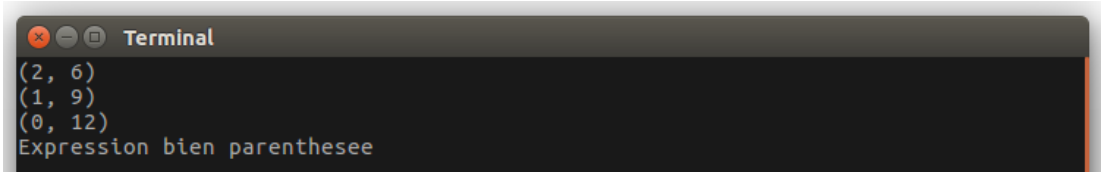
```python
# Realisation d'une structure de pile

def creer_pile():
    return([]) # Renvoie une liste vide

def depiler(p):
    return(p.pop()) # Supprime et renvoie la derniere valeur de la liste p

def empiler(p,v):
    p.append(v) # Ajoute la valeur v en derniere position de la liste p

def taille(p):
    return(len(p)) # Renvoie la taille de la liste p

def est_vide(p):
    return(taille(p)==0)

def sommet(p):
    return(p[-1])

pile=creer_pile()

empiler(pile,7)
empiler(pile,8)
print(pile)

element_recupere=depiler(pile)
print(element_recupere)
print(pile)

valeur_sommet=sommet(pile)
print(valeur_sommet)
print(pile)
```

../cartes_mentales/algorithmique_programmation/piles_structure.py



```python
# Analyse du parenthesage d'une expression arithmetique

# Fonctions de la structure de pile
def creer_pile():
    return([]) # Renvoie une liste vide

def depiler(p):
    return(p.pop()) # Supprime et renvoie la derniere valeur de la liste p

def empiler(p,v):
    p.append(v) # Ajoute la valeur v en derniere position de la liste p

def taille(p):
    return(len(p)) # Renvoie la taille de la liste p

def est_vide(p):
    return(taille(p)==0)

def sommet(p):
    return(p[-1])

# Fonction analyse de l'expression arithmetique, retournant les positions des
# couples de parentheses puis Vrai si elle est bien parenthesee, Faux sinon
def analyse_expression(expression):
    p=creer_pile()
    for i in range(len(expression)):
        if expression[i]=='(':
            empiler(p,i)
        elif expression[i]==')':
            if est_vide(p):
                return(False)
            print((depiler(p),i)) # Retourne les positions des parentheses
    return(est_vide(p))

# Corps du programme
expression='(((3+6)-4)*6)-5'
test=analyse_expression(expression)

if test==True:
    print('Expression bien parenthesee')
else:
    print('Expression mal parenthesee')
```

../cartes_mentales/algorithmique_programmation/piles_parenthesage.py



# La complexité d'un algorithme

## Position du problème

Pour traiter un même problème, plusieurs algorithmes sont possibles.

La durée d'exécution d'un algorithme peut devenir cruciale lorsque cet algorithme est fortement sollicité.

On va donc chercher à estimer le temps d'exécution ou << coût >> d'un algorithme.

## Exemple de l'affichage des diviseurs d'un nombre `n`

### Algorithme naïf

On teste tous les nombres jusqu'à `n`.

```
def diviseurs_naif(n):
    for i in range(1,n+1):
        if n%i==0:
            print(i)
```

Cet algorithme réalise `n` calculs de restes de divisions euclidiennes, `n` comparaisons et au plus `n` affichages.

Le coût doit donc être proportionnel à : $n$

### Autre algorithme

On teste tous les nombres jusqu'à la racine carrée de `n`. Si un diviseur `p` est découvert, alors `q=n/p` est aussi un diviseur.

On prend soin de ne pas afficher deux fois sa racine carrée.

```
def diviseurs(n):
    for i in range(1,int(math.sqrt(n))+1):
        if n%i==0:
            print(i)
            if n//i!=i:
                print(n//i)
```

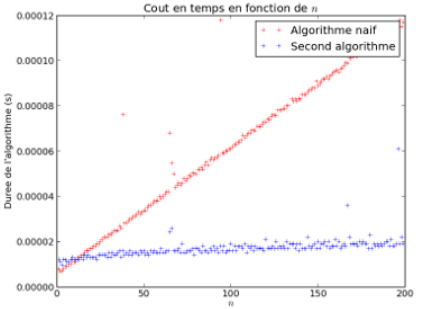
Cet autre algorithme ne réalise que la racine carré de `n` boucles, chacune réalisant `1` calcul de reste, `1` voire `2` comparaisons, et entre `0` et `2` affichages.

Le coût doit donc être proportionnel à : $\sqrt{n}$

### Le calcul de la durée d'exécution d'un algorithme peut s'estimer à l'aide de la fonction `time` de la bibiothèque `time`.

```
import time
t1=time.time()
...
t2=time.time()
print(t2-t1)
```

### On retrouve les résultats de l'analyse précédente.




## Déterminer le coût d'un algorithme

On se base comme sur les exemples précédents sur un modèle de complexité simple : **chaque opération élémentaire (affectation, comparaison ou évaluation d'une opération arithmétique) est comptée comme `1` unité de mesure du coût d'un algorithme**.

On détermine alors le nombre d'opérations élémentaires effectuées par l'algorithme, appelé << complexité >>, en fonction de la quantité des données en entrée, appelée << taille >> du système.

La complexité, **pour une taille importante du système**, tend à être proportionnel à une fonction *f(n)*. On dira que l'algorithme << a une complexité en >> : $O(f(n))$

Remarque : ce modèle ne permet de donner qu'un ordre d'idée, par exemple les coûts d'opérations comme l'addition de deux entiers long ou l'affichage d'une valeur à l'écran sont très supérieurs aux autres.

## Inventaire des complexités courantes, dans l'ordre croissant

- Temps constant $O(1)$ — Le temps d'exécution ne dépend pas des données traitées.
- Logarithmique $O(\ln n)$
- Linéaire $O(n)$
- Quadratique $O(n^2)$
- Polynomiale $O(n^k)$
- Exponentielle $O(a^n)$

## Compléments

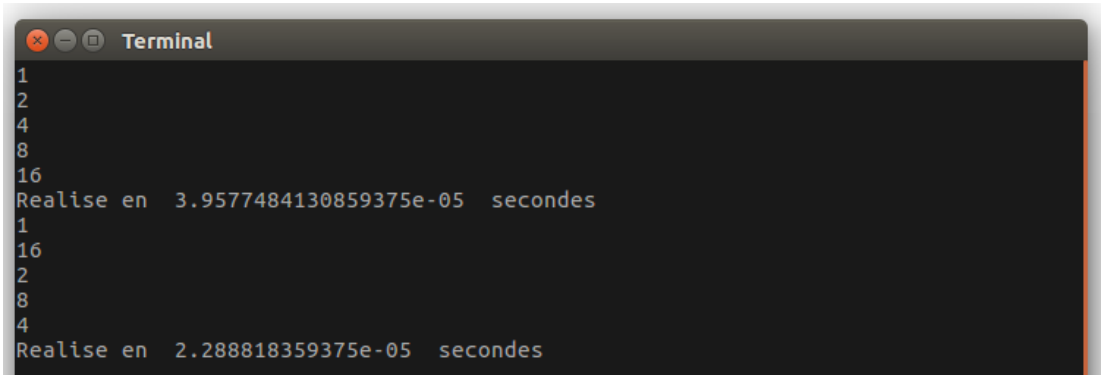
La valeur de la complexité peut fortement varier selon les données d'entrée.

- Complexité au pire
- Complexité dans le meilleur des cas
- Complexité en moyenne et complexité amortie

Complexité en espace

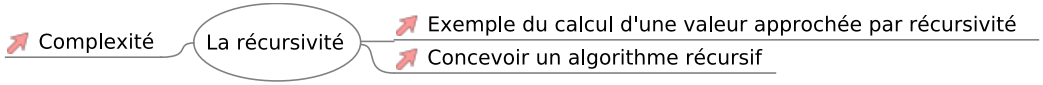
```python
# Exemple de l'affichage des diviseurs d'un nombre entier n
import time
import math

def diviseurs_naif(n):
    for i in range(1,n+1):
        if n%i==0:
            print(i)

def diviseurs(n):
    for i in range(1,int(math.sqrt(n))+1):
        if n%i==0:
            print(i)
            if n//i!=i:
                print(n//i)

t1=time.time()
diviseurs_naif(16)
t2=time.time()
print('Realise en ',t2-t1,' secondes')

t1=time.time()
diviseurs(16)
t2=time.time()
print('Realise en ',t2-t1,' secondes')
```

../cartes_mentales/algorithmique_programmation/complexite_exemple.py



```python
# Comparaison du cout en temps de deux algorithmes de
# determination des diviseurs d'un nombre entier n
import time
import math
import matplotlib.pyplot as plt

def diviseurs_naif(n):
    for i in range(1,n+1):
        if n%i==0:
#            print(i)
            pass # Remplace la fonction 'print', d'un cout tres superieur

def diviseurs(n):
    for i in range(1,int(math.sqrt(n))+1):
        if n%i==0:
#            print(i)
            pass # Remplace la fonction 'print', d'un cout tres superieur
            if n//i!=i:
#                print(n//i)
                pass # Remplace la fonction 'print', d'un cout tres superieur

def cout_diviseurs_naif(n):
    t1=time.time()
    diviseurs_naif(n)
    t2=time.time()
    return(t2-t1)

def cout_diviseurs(n):
    t1=time.time()
    diviseurs(n)
    t2=time.time()
    return(t2-t1)

liste_x=[]
liste_yn=[]
liste_y=[]
for i in range(800):
    liste_yn.append(cout_diviseurs_naif(i))
    liste_y.append(cout_diviseurs(i))
    liste_x.append(i)

plt.plot(liste_x,liste_yn,'r+',liste_x,liste_y,'b+')
#plt.plot(liste_x,liste_y,'b+')
plt.xlabel('$n$')
plt.ylabel('Duree de l\'algorithme (s)')
plt.title('Cout en temps en fonction de $n$')
plt.legend(('Algorithme naif','Second algorithme'))
plt.show()
```

../cartes_mentales/algorithmique_programmation/complexite_exemple_durees.py


Complexité
La récursivité
Exemple du calcul d'une valeur approchée par récursivité
Concevoir un algorithme récursif

Exemple du calcul d'une valeur approchée par récursivité

Calcul de la valeur approchée de : $\sqrt{3}$

Principe

On utilise la suite ci-dessous qui converge vers la valeur cherchée.

$$u_0 = 2$$
$$u_n = \frac{1}{2}\left(u_{n-1} + \frac{3}{u_{n-1}}\right)$$

Codage classique

```
def u(n):
    r=2.
    for i in range(n):
        r=0.5*(r+3./r)
    return(r)

print(u(2))
```

Codage récursif

```
def u(n):
    if n==0:
        return(2.)
    else:
        x=u(n-1)
        return(0.5*(x+3./x))

print(u(2))
```

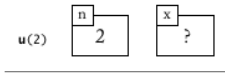
x est utilisée comme variable intermédiaire pour limiter les appels à `u(n-1)`.

Visualisation du déroulement à l'aide du site *Online Python Tutor*

Déroulement

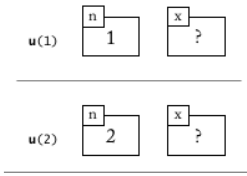
Appel de `u` et création d'une variable locale `n` qui prend la valeur `2`

`print(u(2))`



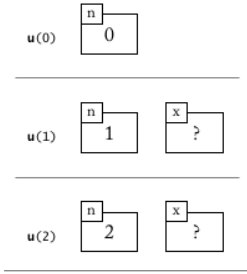

Nouvel appel de `u` et création d'une nouvelle variable locale `n` qui prend la valeur `1`

`x=u(n-1)`




Nouvel appel de `u` et création d'une nouvelle variable locale `n` qui prend la valeur `0`

`x=u(n-1)`

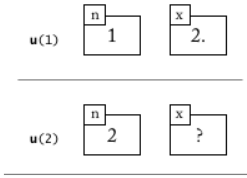



La valeur `2.` est retournée.

`return(2)`

La variable `x` de l'appel précédent prend alors la valeur `2.`

`x=u(n-1)`

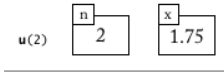



La valeur `1.75` est retournée.

`return(0.5*(x+3./x))`

La variable `x` de l'appel précédent prend alors la valeur `1.75`

`x=u(n-1)`



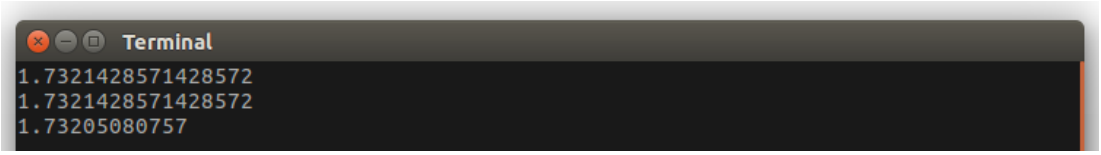

La valeur `1.7321...` est retournée.

`return(0.5*(x+3./x))`

Ce code est basé sur un << algorithme récursif >>. Les calculs sont les mêmes que dans l'algorithme précédent.

```python
# Exemple du calcul d'une approximation de racine(3) par recursivite
import scipy as sp

# Codage classique
def u(n):
    r=2.
    for i in range(n):
        r=0.5*(r+3./r)
    return(r)

print(u(2))


# Codage recursif
def u(n):
    if n==0:
        return(2.)
    else:
        x=u(n-1)
        return(0.5*(x+3./x))

print(u(2))
print(sp.sqrt(3.))
```

../cartes_mentales/algorithmique_programmation/recursivite_exemple.py



## Concevoir un algorithme récursif

### Schéma de récursivité simple

Le principe est celui de l'exemple précédent.

```
def u(n):
    if n==0:
        return(...)
    else:
        return(...u(n-1)...)
```

Exemples

Fonction factorielle $n!$

```
def factorielle(n):
    if n==0:
        return(1)
    else:
        return(n*factorielle(n-1))
```

Fonction puissance $x^n$

```
def puissance(x,n):
    if n==0:
        return(1)
    else:
        return(x*puissance(x,n-1))
```

### Schéma de récursivité forte

Les appels récursifs peuvent être effectués sur d'autres valeurs (inférieures) à `n-1`.

Exemple de la fonction puissance $x^n$

Un algorithme plus rapide est basé sur le schéma de récursivité ci-dessous, suivant la parité de `n` :

$$x^{2k} = (x^k)^2$$
$$x^{2k+1} = x(x^k)^2$$

```
def puissance_rapide(x,n):
    if n==0:
        return(1)
    else:
        r=puissance_rapide(x,n//2)
        if n%2==0:
            return(r*r)
        else:
            return(x*r*r)
```

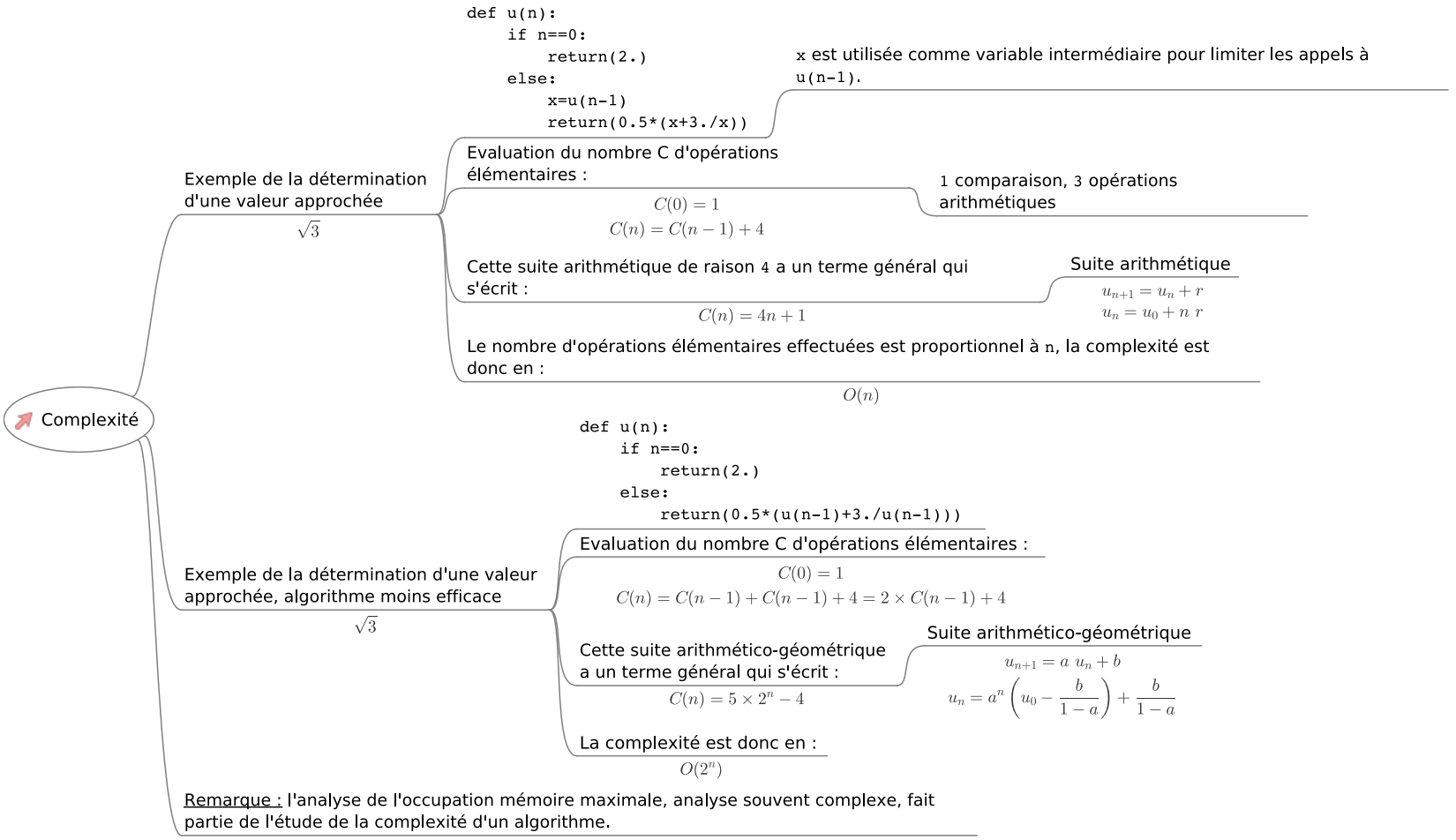
Exemple de la détermination du nombre de façons de construire une rangée de longueur `n` avec des briques de longueurs `2` et `3` (on ajoute récursivement le nombre de façons où la rangée se termine par une brique de `2` à celui où elle se termine par une brique de `3`).

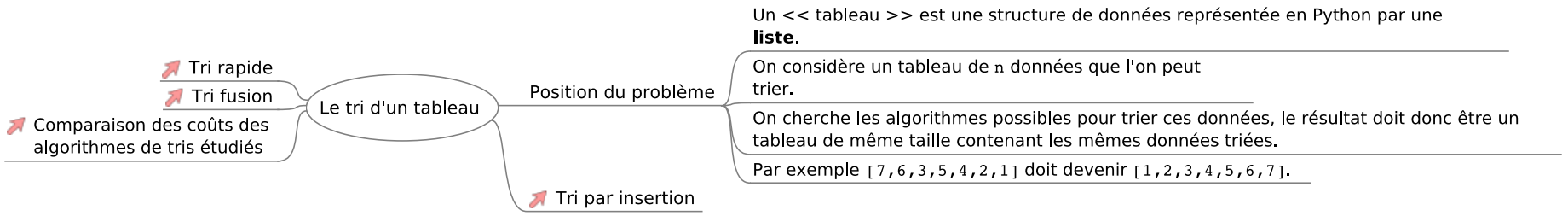
```
def nb_facons(n):
    if n==1:
        return(0)
    elif n==2 or n==3:
        return(1)
    else:
        return(nb_facons(n-2)+nb_facons(n-3))
```

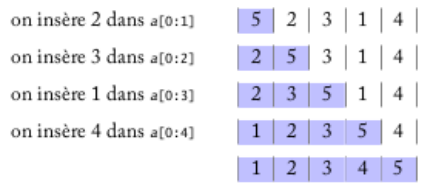
Une fonction peut effectuer des appels récursifs sur une autre fonction qui elle-même effectue un appel récursif sur la première fonction : on parle de << récursivité mutuelle >>.

Exemple de la détermination de la parité d'un nombre `n`.

```
def pair(n):
    return(n==0 or impair(n-1))

def impair(n):
    return(n!=0 or pair(n-1))
```

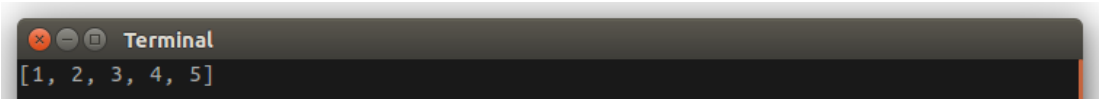

Complexité
Exemple de la détermination d'une valeur approchée
$\sqrt{3}$
def u(n):
if n==0:
return(2.)
else:
x=u(n-1)
return(0.5*(x+3./x))
x est utilisée comme variable intermédiaire pour limiter les appels à u(n-1).
Evaluation du nombre C d'opérations élémentaires :
$C(0) = 1$
$C(n) = C(n-1) + 4$
1 comparaison, 3 opérations arithmétiques
Cette suite arithmétique de raison 4 a un terme général qui s'écrit :
$C(n) = 4n + 1$
Suite arithmétique
$u_{n+1} = u_n + r$
$u_n = u_0 + n\ r$
Le nombre d'opérations élémentaires effectuées est proportionnel à n, la complexité est donc en :
$O(n)$
Exemple de la détermination d'une valeur approchée, algorithme moins efficace
$\sqrt{3}$
def u(n):
if n==0:
return(2.)
else:
return(0.5*(u(n-1)+3./u(n-1)))
Evaluation du nombre C d'opérations élémentaires :
$C(0) = 1$
$C(n) = C(n-1) + C(n-1) + 4 = 2 \times C(n-1) + 4$
Cette suite arithmético-géométrique a un terme général qui s'écrit :
$C(n) = 5 \times 2^n - 4$
Suite arithmético-géométrique
$u_{n+1} = a\ u_n + b$
$u_n = a^n \left( u_0 - \frac{b}{1-a} \right) + \frac{b}{1-a}$
La complexité est donc en :
$O(2^n)$
Remarque : l'analyse de l'occupation mémoire maximale, analyse souvent complexe, fait partie de l'étude de la complexité d'un algorithme.


Le tri d'un tableau
Position du problème
Un << tableau >> est une structure de données représentée en Python par une **liste**.
On considère un tableau de `n` données que l'on peut trier.
On cherche les algorithmes possibles pour trier ces données, le résultat doit donc être un tableau de même taille contenant les mêmes données triées.
Par exemple `[7,6,3,5,4,2,1]` doit devenir `[1,2,3,4,5,6,7]`.
Tri par insertion
Tri rapide
Tri fusion
Comparaison des coûts des algorithmes de tris étudiés

- Tri par insertion
  - C'est la méthode de tri la plus naturelle.
  - Exemple sur un tableau `[5,2,3,1,4]`
  - Principe
    - Le premier élément est pris comme référence. Le second est comparé au premier puis éventuellement inséré avant. Le troisième est comparé aux deux premiers puis inséré au bon endroit, et ainsi de suite.
    - 
      
  - Procédure
    - On parcourt la liste et pour chaque élément, on déplace les éléments précédents qui lui sont postérieurs puis l'on l'insère.
  - Codage
    - Visualisation du déroulement à l'aide de *Online Python Tutor*
  - Complexité
    - La complexité va fortement dépendre du désordre initial des données présentes dans le tableau.
    - Le nombre de comparaisons et d'affectations est toujours le même dans la boucle `while` (responsable du coût variable de l'algorithme), il suffit donc de décompter l'un ou l'autre pour déterminer la complexité.
    - Complexité au mieux
      - C'est le cas où les données sont déjà triées.
      - On se contente de parcourir les `n` éléments du tableau et d'effectuer `1` comparaison à chaque fois.
      - La complexité au mieux est donc en :
        $$O(n)$$
    - Complexité au pire
      - C'est le cas où les données sont ordonnées en sens inverse.
      - On parcourt les `n` éléments du tableau et on effectue `1`, puis `2` puis `3` puis... `n-1` comparaisons à chaque fois. Le nombre d'opération élémentaires est donc proportionnel à la somme de ces termes :
        $$\frac{n(n-1)}{2}$$
      - La complexité au pire est donc en :
        $$O(n^2)$$
    - Remarque : le coût en mémoire est constant puisqu'aucun autre tableau n'est requis.

```python
# Tri d'un tableau (liste) par insertion

def tri_insertion(t):
    for i,v in enumerate(t):
        j=i
        while j>0 and v<t[j-1]:
            t[j]=t[j-1]
            j=j-1
        t[j]=v

liste=[5,2,3,1,4]
tri_insertion(liste)
print(liste)
```

../cartes_mentales/algorithmique_programmation/tri_insertion.py



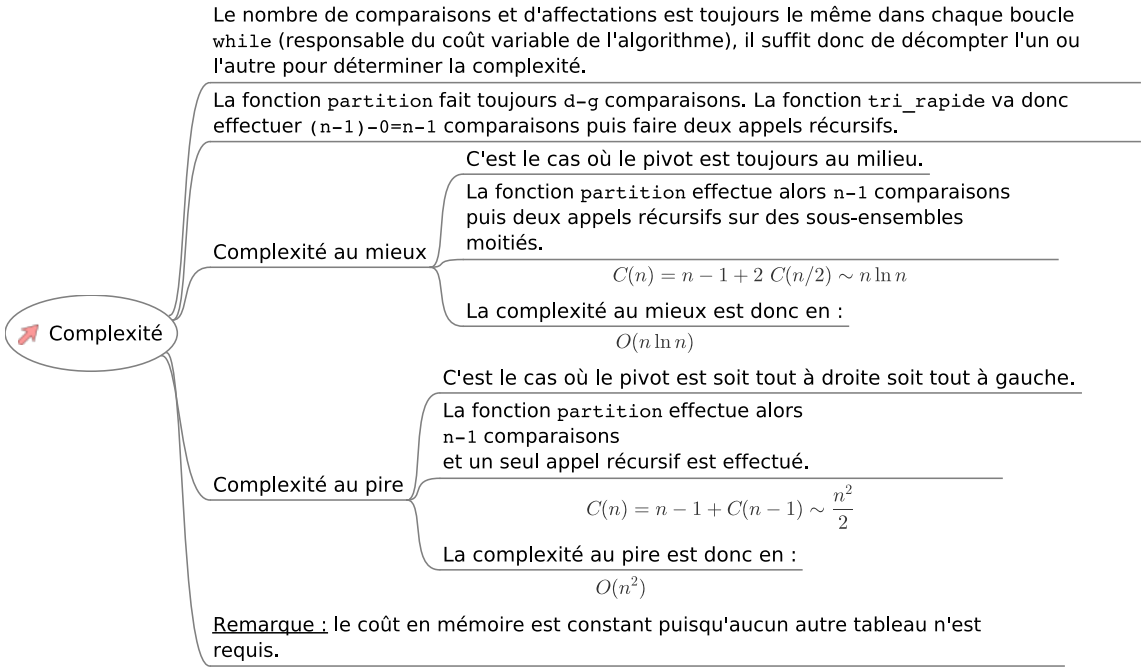
- Tri rapide
  - Principe
    - On utilise la méthode << diviser pour mieux régner >>.
    - Exemple sur un tableau
      `[7,6,3,5,4,2,1]`
    - Un élément est pris au hasard comme << pivot >>. On place alors à sa gauche les élements plus petits, à droite les éléments plus grands. On recommence pour chaque sous-ensemble.

  - Procédure
    - **On créé une fonction `partition` qui effectue la répartition autour d'un pivot choisi arbitrairement.**
      - On met le pivot en première position (par un échange avec le premier élément).
      - On repère la position future du pivot avec l'indice `m`.
      - On parcourt le tableau `t` du second élément au dernier, avec l'indice `i`. Si `t[i]` est inférieur à la valeur du pivot, on incrémente m et on échange `t[i]` avec `t[m]`. On se retrouve ainsi avec les valeurs inférieurs à gauche, la position de la dernière valeur inférieure étant repérée par l'indice `m`.
      - On place le pivot à sa place par un échange entre la première valeur et la dernière valeur inférieure repérée par la position `m`.
    - **On créé une fonction `tri_rapide` qui appelle la fonction `partition` puis qui s'appelle elle-même pour effectuer le tri des deux autres sous-ensembles (tri des sous-ensembles par récursivité).**
      - On appelle la fonction `partition` et on récupère la position `m` du pivot.
      - On appelle la fonction `tri_rapide` pour trier les sous-ensembles autour de la position `m` du pivot.
      - Si le sous-ensemble a une seule valeur, on interrompt les appels récursifs.
  - Complexité
  - Codage
    - Visualisation du déroulement à l'aide de *Online Python Tutor*
    - Python limite le nombre d'appels récursifs à 1000.
  - Amélioration du codage
    - On peut éliminer un appel récursif à l'aide d'une boucle.

```
def tri_rapide(t,g,d):
    while g<d:
        m=partition(t,g,d)
        tri_rapide(t,g,m-1)
        g=m+1
```

      - Mais si le pivot se trouve plus souvent sur la partie droite que gauche, l'appel récursif sera fortement sollicité (grand sous-ensemble à traiter).
    - On peut limiter l'appel récursif au plus petit sous-ensemble.

```
def tri_rapide(t,g,d):
    while g<d:
        m=partition(t,g,d)
        if m-g<d-m:
            tri_rapide(t,g,m-1)
            g=m+1
        else:
            tri_rapide(t,m+1,d)
            d=m
```


Complexité
Le nombre de comparaisons et d'affectations est toujours le même dans chaque boucle while (responsable du coût variable de l'algorithme), il suffit donc de décompter l'un ou l'autre pour déterminer la complexité.
La fonction partition fait toujours d-g comparaisons. La fonction tri_rapide va donc effectuer (n-1)-0=n-1 comparaisons puis faire deux appels récursifs.
Complexité au mieux
C'est le cas où le pivot est toujours au milieu.
La fonction partition effectue alors n-1 comparaisons puis deux appels récursifs sur des sous-ensembles moitiés.
$C(n) = n - 1 + 2\ C(n/2) \sim n \ln n$
La complexité au mieux est donc en :
$O(n \ln n)$
Complexité au pire
C'est le cas où le pivot est soit tout à droite soit tout à gauche.
La fonction partition effectue alors n-1 comparaisons et un seul appel récursif est effectué.
$C(n) = n - 1 + C(n-1) \sim \frac{n^2}{2}$
La complexité au pire est donc en :
$O(n^2)$
Remarque : le coût en mémoire est constant puisqu'aucun autre tableau n'est requis.

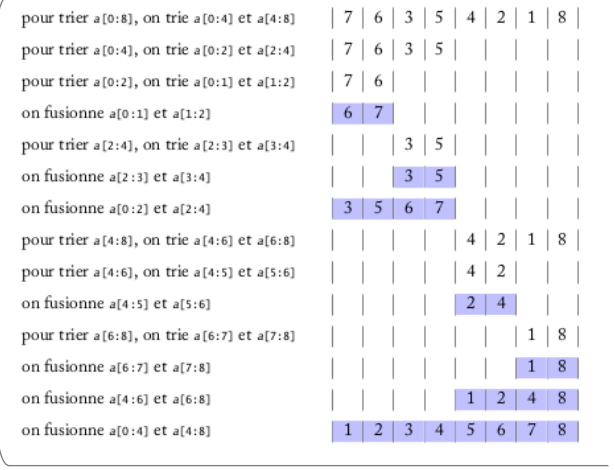
```python
# Tri rapide d'un tableau (liste)
import random

def echange(liste,i,j):
    liste[i],liste[j]=liste[j],liste[i]

def partition(t,g,d):
#    echange(t,random.randint(g,d),g)
    m=g # m est l'indice du pivot
    for i in range(g+1,d+1):
        if t[i]<t[g]:
            m=m+1
            if i!=m:
                echange(t,i,m)
    if m!=g:
        echange(t,g,m)
    return(m)

def tri_rapide(t,g,d):
    if g>=d:
        return()
    m=partition(t,g,d)
    tri_rapide(t,g,m-1)
    tri_rapide(t,m+1,d)


tableau=[7,6,3,5,4,2,1]
tri_rapide(tableau,0,len(tableau)-1)
print(tableau)
```

../cartes_mentales/algorithmique_programmation/tri_rapide.py

```python
# Tri rapide d'un tableau (liste)
import random

def echange(liste,i,j):
    liste[i],liste[j]=liste[j],liste[i]

def partition(t,g,d):
#    echange(t,random.randint(g,d),g)
    m=g # m est l'indice du pivot
    for i in range(g+1,d+1):
        if t[i]<t[g]:
            m=m+1
            if i!=m:
                echange(t,i,m)
    if m!=g:
        echange(t,g,m)
    return(m)

def tri_rapide_ameliore(t,g,d):
    while g<d:
        m=partition(t,g,d)
        if m-g<d-m:
            tri_rapide_ameliore(t,g,m-1)
            g=m+1
        else:
            tri_rapide_ameliore(t,m+1,d)
            d=m-1


tableau=[7,6,3,5,4,2,1]
tri_rapide_ameliore(tableau,0,len(tableau)-1)
print(tableau)
```

../cartes_mentales/algorithmique_programmation/tri_rapide_amelioration.py

- Tri fusion
  - Principe
    - On utilise aussi la méthode << diviser pour mieux régner >>.
    - Exemple sur un tableau `[7,6,3,5,4,2,1]`
    - On partage le tableau en deux sous-ensembles égaux, sans chercher à en trier les éléments. On recommence pour chaque sous-ensemble jusqu'à avoir des sous-ensembles unités. On finit par fusionner chaque sous-ensemble en les triant.

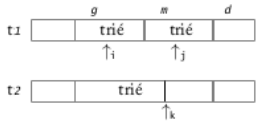



  - Procédure

**On créé une fonction `fusion` qui, partant d'un tableau comportant deux sous-ensembles triés, va recopier les valeurs dans un second tableau en les triant, formant un seul ensemble trié.**

On repère les valeurs extrêmes du tableau `t1` à fusionner par les indices `g` (gauche) et `d` (droite). On repère les éléments du premier sous-ensemble par l'indice `i`, et ceux du second sous-ensemble par l'indice `j`.

On repère les éléments du second tableau `t2` (issu de la fusion des deux sous-ensembles du tableau `t1`) par l'indice `k`.

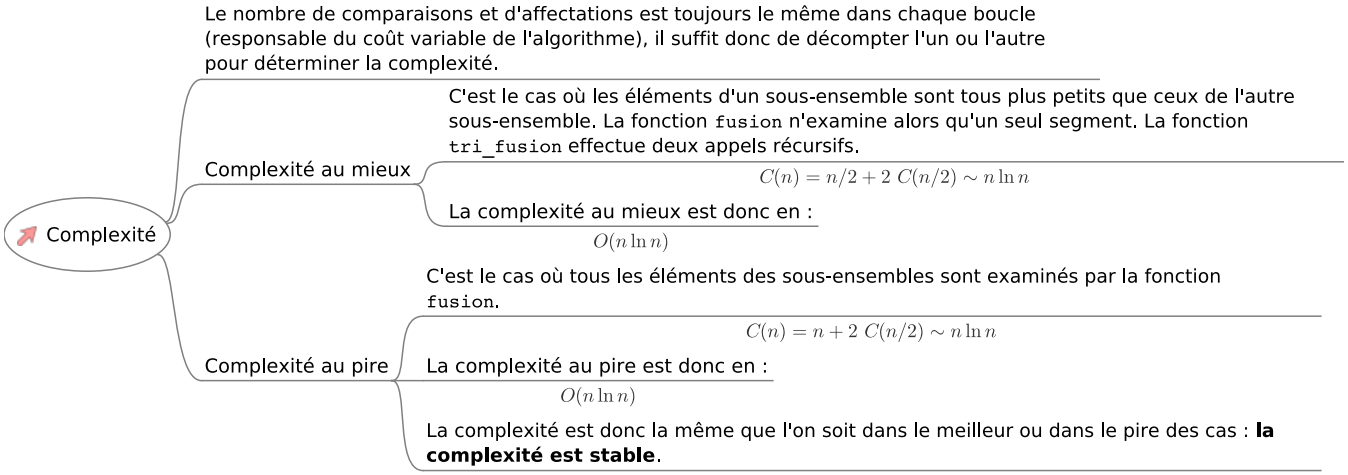



On parcourt alors le tableau `t1` en comparant `t1[i]` et `t1[j]`, puis on recopie la plus petite valeur dans `t2[k]` en incrémentant les indices correspondants. Le cas où l'un des deux sous-ensemble est plus court que l'autre doit être pris en compte.

**On créé une fonction `tri_fusion`, qui va diviser le tableau en deux sous-ensembles en repérant la position de l'élément du milieu par l'indice `m`, puis s'appeler elle-même pour faire de même sur les sous-ensembles (par récursivité). On fusionne ensuite chaque sous-ensemble obtenu en utilisant un tableau temporaire `tmp`.**

Visualisation du déroulement à l'aide de *Online Python Tutor*

  - Codage
  - Complexité


Complexité
Le nombre de comparaisons et d'affectations est toujours le même dans chaque boucle (responsable du coût variable de l'algorithme), il suffit donc de décompter l'un ou l'autre pour déterminer la complexité.
Complexité au mieux
C'est le cas où les éléments d'un sous-ensemble sont tous plus petits que ceux de l'autre sous-ensemble. La fonction fusion n'examine alors qu'un seul segment. La fonction tri_fusion effectue deux appels récursifs.
$C(n) = n/2 + 2\ C(n/2) \sim n \ln n$
La complexité au mieux est donc en :
$O(n \ln n)$
Complexité au pire
C'est le cas où tous les éléments des sous-ensembles sont examinés par la fonction fusion.
$C(n) = n + 2\ C(n/2) \sim n \ln n$
La complexité au pire est donc en :
$O(n \ln n)$
La complexité est donc la même que l'on soit dans le meilleur ou dans le pire des cas : la complexité est stable.

```python
# Tri fusion d'un tableau

def fusion(t1,t2,g,d,m):
    i,j=g,m
    for k in range(g,d+1):
        if i<m and (j>d or t1[i]<=t1[j]): # Bien mettre 'j>d' en premiere position
            t2[k]=t1[i]
            i=i+1
        else:
            t2[k]=t1[j]
            j=j+1

def tri_fusion(t,g,d):
    if g>=d:
        return()
    m=(g+d+1)//2
    tri_fusion(t,g,m-1)
    tri_fusion(t,m,d)
    tmp[g:d+1]=t[g:d+1] # t[a:b] comporte les elements de a jusqu'a b-1
    fusion(tmp,t,g,d,m)

tableau=[7,6,3,5,4,2,1]
tmp=[0]*len(tableau) # Creation d'un tableau temporaire vide de meme taille que
    tableau
tri_fusion(tableau,0,len(tableau)-1)
print(tableau)
```

../cartes_mentales/algorithmique_programmation/tri_fusion.py

Comparaison des coûts des algorithmes de tris étudiés

Tableau trié

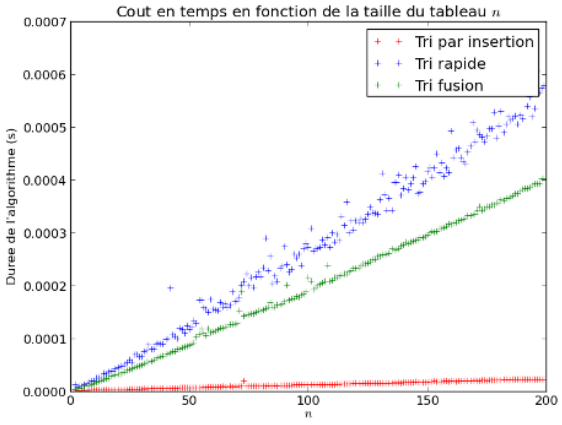
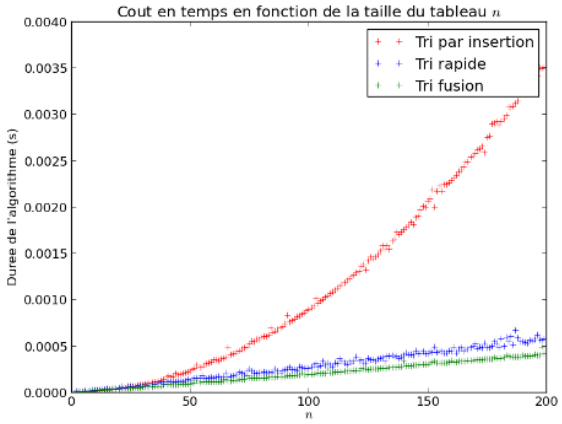


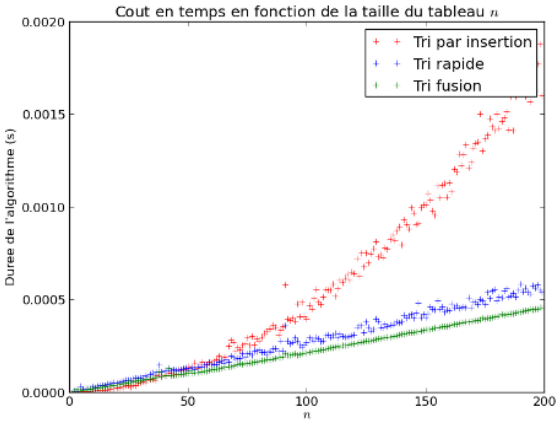

Tableau inversé




On se place dans le cas d'un tableau dont les éléments sont mélangés au hasard, triés dans l'ordre croissant puis triés dans l'ordre décroissant.

On mesure le coût de chaque algorithme en fonction du nombre `n` d'éléments du tableau, choisi comme taille du système.

Tableau à répartition aléatoire



```python
# Comparaison du cout en temps des algorithmes de
# tri d'un tableau de dimension n
import time
import math
import matplotlib.pyplot as plt
import random

# Algorithme de tri par insertion
def tri_insertion(t):
    for i,v in enumerate(t):
        j=i
        while j>0 and v<t[j-1]:
            t[j]=t[j-1]
            j=j-1
        t[j]=v
    return(t)

def cout_tri_insertion(t):
    t1=time.time()
    tri_insertion(t)
    t2=time.time()
    return(t2-t1)

#Agorithme de tri rapide
def echange(liste,i,j):
    liste[i],liste[j]=liste[j],liste[i]

def partition(t,g,d):
    echange(t,random.randint(g,d),g)
    m=g # m est l'indice du pivot
    for i in range(g+1,d+1):
        if t[i]<t[g]:
            m=m+1
            if i!=m:
                echange(t,i,m)
    if m!=g:
        echange(t,g,m)
    return(m)

def tri_rapide(t,g,d):
    if g>=d:
        return()
    m=partition(t,g,d)
    tri_rapide(t,g,m-1)
    tri_rapide(t,m+1,d)

def cout_tri_rapide(t):
    t1=time.time()
    tri_rapide(t,0,len(t)-1)
    t2=time.time()
    return(t2-t1)

# Algorithme de tri fusion
def fusion(t1,t2,g,d,m):
    i,j=g,m
    for k in range(g,d+1):
        if i<m and (j>d or t1[i]<=t1[j]): # Bien mettre 'j>d' en premiere position
            t2[k]=t1[i]
            i=i+1
        else:
            t2[k]=t1[j]
            j=j+1

def tri_fusion(t,g,d):
#    tmp=[0]*len(t) # Creation d'un tableau temporaire vide de meme taille que t
```

```
    if g>=d:
        return()
    m=(g+d+1)//2
    tri_fusion(t,g,m-1)
    tri_fusion(t,m,d)
    tmp[g:d+1]=t[g:d+1] # t[a:b] comporte les elements de a jusqu'a b-1
    fusion(tmp,t,g,d,m)

def cout_tri_fusion(t):
    t1=time.time()
    tri_fusion(t,0,len(t)-1)
    t2=time.time()
    return(t2-t1)

liste_x=[]
liste_yti=[]
liste_ytr=[]
liste_ytf=[]
lim=200
for i in range(2,lim):
    tableau=[j for j in range(1,i+1)]
#    random.shuffle(tableau)
    tableau.reverse()
    ti=list(tableau)
    liste_yti.append(cout_tri_insertion(ti))
    tr=list(tableau)
    liste_ytr.append(cout_tri_rapide(tr))
    tf=list(tableau)
    tmp=[0]*len(tableau)
    liste_ytf.append(cout_tri_fusion(tf))
    liste_x.append(i)

plt.plot(liste_x,liste_yti,'r+',liste_x,liste_ytr,'b+',liste_x,liste_ytf,'g+')
plt.xlabel('$n$')
plt.ylabel('Duree de l\'algorithme (s)')
plt.title('Cout en temps en fonction de la taille du tableau $n$')
plt.legend(('Tri par insertion','Tri rapide','Tri fusion'))
plt.show()
```

../cartes_mentales/algorithmique_programmation/tri_durees.py

## Références

[1] Wack, Benjamin, Conchon, Sylvain, Courant, Judicaël, De Falco, Marc, Dowek, Gilles, Filliâtre, Jean-Christophe, Gonnord, Stéphane, *Informatique pour tous en classes préparatoires aux grandes écoles*, Eyrolles, 2013.

[2] Beauquier, D., Berstel, J., Chrétienne, Ph., *Éléments d'algorithmique*, `http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.pdf`

[3] *Online Python Tutor*, `http://www.pythontutor.com/`