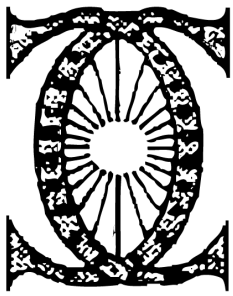
Classe de BCPST, 2ème année

Version du 30 septembre 2018, 13 :31

TPs
du cours d'Informatique

Jonathan HARTER
Lycée CHAPTAL
45 Boulevard des Batignolles
75008 Paris

😄 Merci de me signaler, s'il vous plaît, toute coquille trouvée par mail. 😄



I have deeply regretted that I did not proceed far enough at least to understand something of the great leading principles of mathematics, for men thus endowed seem to have an extra sense.

– Charles Darwin

Table des matières

Remerciements

*U*n grand merci à tous les collègues du lycée Chaptal, Grégory et Anne-Sophie notamment, pour les supports mis à ma disposition.

TP ❶ — Révisions

♣ Plan du TP

➥ Résumé du TP

Conformément au programme, nous aborderons l'apprentissage de l'informatique avec plusieurs objectifs :

❶ la compréhension et la maîtrise de différents algorithmes (recherche dans une liste, tri d'un tableau de nombres ...),

❷ l'utilisation autonome de méthodes numériques (calcul approché d'une intégrale, simulation d'une variable aléatoire, résolution d'un système linéaire ...) en lien avec le cours de mathématiques,

❸ la connaissance d'un langage de programmation et de son environnement (`Python` et `Pyzo` ici),

❹ la réalisation d'un projet présenté à l'oral aux principaux concours offerts à la classe BCPST.

Dans ce TP, nous passerons en revue les points suivants : les structures de contrôle, nous reverrons la notion de fonction et celles issues des principales bibliothèques que nous allons utiliser, et enfin nous reverrons certaines opérations afférentes aux listes.

Il est fortement conseillé, à l'issue de chaque T.P. de conserver votre fichier `.py` sur une clé USB afin de le retravailler chez vous.

1. Echauffement

Cette section constitue un rappel de connaissances de première année sur les structures de contrôles (boucles, tests logiques, *etc.*).

1— 1. L'environnement de travail

L'environnement de développement `Pyzo` est choisi pour sa simplicité de mise en oeuvre. Il est constitué

✓ d'un éditeur qui permet de saisir le programme (les mots-clé du langage sont colorés, ce qui permet une relecture facile, et l'indentation est automatique),

✓ d'une console, qui permet d'exécuter des instructions en ligne de commande (en tapant à la suite de `+` et de suivre le déroulement de son programme,

✓ d'un explorateur de variables, qui permet de connaître les valeurs contenues dans les variables en cours d'utilisation.

⚠ **Attention** — *On fera attention à bien enregistrer son travail sous la forme de fichiers `.py` de façon régulière, dans un répertoire créé à cet effet (par exemple `TP1`) et on exécutera le programme en cours à l'aide de la commande « exécution » ou de la touche F5.*

Remarque 1.1. En Python, toute ligne commençant par un # est un commentaire : son contenu sera ignoré lors de l'exécution du programme. Ainsi, il est essentiel d'écrire dans un script des commentaires précisant en toutes lettres ce que le programme fait (ou est censé faire !) afin d'en faciliter la relecture.
De plus, l'indentation (décalage du début de ligne pour aligner verticalement les instructions) est non seulement essentiel pour relire son programme (quelles sont les instructions exécutées après un `if` dans une boucle `for` ?) mais aussi obligatoire pour le bon fonctionnement du programme.
On rappelle également que l'indentation ne se fait pas n'importe comment : on utilise la touche de tabulation Tab.

1— 2. Aide-mémoire des types de données usuels

En Python, une expression est une suite de caractères définissant une valeur, qui possède un type. Saisir par exemple :

```
type(42)
type(2.5)
type('Bonjour')
```

Les types usuels sont les suivants :

Type	Exemple	Opérations
Entier `int`	42	`+ , - , *, ** , // , %`
Flottant `float`	0.3	`+ , - , *, ** , /`
Booléen `bool`	1==2	`and, or, not`
Chaîne de caractères `str`	"blablabla"	`+ , len , in`
n-uplets `tuple`	(0,1,2)	`+ , len , in`

On verra ultérieurement des structures de données plus complexes (liste, tableau multidimensionnel).

Remarque 1.2. La principale différence entre un tuple `L1=(0,1,2)` est une liste `L2=[0,1,2]` est que les coordonnés de `L1` ne peuvent être changées.

Remarque 1.3 – Conversion. Il est possible de convertir une expression de type entier en flottant (de la même façon qu'un mathématiques, un nombre entier est aussi un nombre réel).

```
a=1
type(a)
float(a)
a=float(a)
type(a)
```

Ainsi, on constate que la variable `a` possède le type de son contenu. Noter que l'affectation d'une variable se fait avec `=`.

***Exercice* 1**
Écrire un programme, qui n'utilisera pas de fonction, qui demande à l'utilisateur deux nombres entiers a et b (stockés dans les variables `a` et `b`, affiche le contenu de `a` et `b`, puis échange leur contenu et l'affiche à nouveau. On pourra utiliser les instructions `input` et `print`.

SOLUTION de l'Exercice 1.

```
a=input('entrer a')
b=input('entrer b')
print(a)
print(b)
c=a
a=b
b=c
print(a)
print(b)
```

1— 3. Les tests logiques

On considère le programme suivant :

```
x=int(input("entrer x: "))
if x%2==1:
    x=x+1
else:
    x=x//2
print(x)
```

Que fait ce programme ? Que font les instructions `+`, `//` et `==` ? Quel est le type de l'expression `x % 2 ==1` ?

> ***SOLUTION*** Demande un entier `x`, regarde si le reste de la division euclidienne de `x` par deux est un (*i.e.* `x` est un entier impair), et ajoute un si c'est le cas donc renvoie le nombre pair suivant. Sinon (il est donc pair) il renvoie le quotient.

Remarque 1.4. S'il y a plus que deux cas nécessitant un traitement différencié, on peut utiliser l'instruction `elif` (contraction de else et if). L'instruction `else` finale sera traitée seulement si les cas précédents n'ont pas été rencontrés.

1— 4. Les boucles inconditionnelles

Ce sont les boucles dont on connait à l'avance le nombre d'étapes nécessaires pour arriver au résultat. Voici l'exemple du calcul de $n!$:

```
n=int(input("entrer n: "))
fact=1
for k in range(n):
    fact=(k+1)*fact
print(fact)
```

En particulier, l'instruction `list(range(n))` produit une liste d'entiers consécutifs entre 0 et $n-1$:

```
list(range(5))
```

⚠ **Attention** — *Il faut bien faire attention au fait que `range(a,b)` **n'est pas** une liste mais un itérateur, i.e. un objet de référence pour décrire une boucle `for`. Il faut donc d'abord le convertir en liste pour afficher ses valeurs.*

1— 5. Les boucles conditionnelles

L'arrêt de la boucle est ici surbordonné à une condition. On se sait pas *a priori* lorsqu'elle va se terminer. Il convient donc de faire attention au fait que cette boucle se termine bien avant de lancer le script.

Que fait le script ci-dessous ?

```
a=int(input("entrer a: "))
b=int(input("entrer b: "))
q=0
r=a
while r>=b:
    q=q+1
    r=r-b
print("q=",q)
print("r=",r)
```

SOLUTION de l'Exercice 1. Il s'agit de l'algorithme de la division euclidienne.

2. Fonction

2— 1. Généralités

Dans le langage Python, une fonction est une suite d'instructions dépendant de paramètres. Par exemple

Script 1.1 – Fonction factorielle

```
def factorielle(n):
    ' calcule n! '
    #ces apostrophes mentionnent le rôle de la fonction
    fact=1
    for k in range(n):
        fact=(k+1)*fact
    return fact
```

On peut préciser ce que fait la fonction entre les guillemets qui suivent sa définition (c'est une chaîne de documentation), et

```
help(factorielle)
```

affichera ces renseignements. Le paramètre `n` est passé par valeur, c'est-à-dire que seule la valeur prise par `n` au moment de l'appel de la fonction est connue dans la suite des instructions.

⚠ **Attention** — *`n` n'est pas une variable, on ne peut donc pas modifier la valeur de `n` à l'intérieur de la fonction. Au même titre que l'indice d'une somme ou la variable d'intégration d'une intégrale : vous remplacez ces objets par ce que vous voulez mais en dehors de ces opérateurs, ils n'existent pas !*

De plus, une fonction peut renvoyer une valeur (le résultat) grâce à l'instruction `return`. Écrire la fonction suivante :

Script 1.2 – Passage par valeur

```
def incr(x):
    x=x+1
    return(x)
```

Prédire la valeur de `x` à chaque étape puis vérifier avec Python :

```
x=3
incr(x)
x
x=incr(x)
```

⚠ **Attention** — *Dès qu'une instruction `return` apparaît, la fonction s'arrête.*

Remarque 2.1. A l'intérieur d'une fonction, il est possible d'utiliser des variables. Par défaut, ces variables sont dites locales : leur contenu n'est accessible qu'à l'intérieur de la fonction et pendant son exécution.

Script 1.3 – Portée d'une variable

```
def g(x):
    a=x**2
    return(a)
```

Ainsi, la variable a n'existe pas hors de la fonction g. Si l'on souhaite pouvoir modifier le contenu d'une variable à l'intérieur d'une fonction, il faut lui donner une portée globale à l'aide du mot-clé `global`.

2— 2. Les fonctions de bibliothèques

Python dispose de plusieurs bibliothèques de fonctions destinées à un usage spécifique. Il convient de les charger grâce à la commande `import` avant de pouvoir les utiliser. Détaillons celles qui nous seront le plus utiles pendant l'année.

La bibliothèque `math`

Prenons l'exemple de la bibliothèque `math` : elle contient les fonctions suivantes

```
import math
math.fabs(x),math.factorial(x),math.floor(x), math.fsum(iterable), math.exp(x)
math.sqrt(x), math.acos(x), math.asin(x), math.atan(x), math.cos(x), math.hypot(x, y)
math.tan(x), math.degrees(x), math.radians(x), math.cosh(x), math.sinh(x), math.tanh(x)
math.erf(x), math.gamma(x), math.log(x[, base]), math.sin(x)

#Affichage des fonctions d'une bibliothèque
import math
dir(math)
```

On retiendra que la fonction `math.factorial` peut désormais être utilisée. Les fonctions usuelles peuvent aussi être importées depuis la bibliothèque `numpy` : ces dernières sont plus adaptées aux calculs numériques. Notamment, le cosinus de `numpy` peut s'appliquer à une liste coefficient par coefficient (avec `math` ce n'est pas le cas).

L'instruction `lambda`

Pour définir des fonctions d'une variable réelle, une syntaxe relativement commode est la suivante

```
g = lambda x: x**2
```

qui définit dans le cas présent la fonction $x \longmapsto x^2$. De manière équivalente, on aurait pu utiliser

```
def f(x):
        return x**2
```

Les bibliothèques `numpy`, `scipy`, `random` *et* `matplotlib`

Les bibliothèques `numpy` et `scipy` fournissent des outils pour le calcul scientifique. Il est possible de gagner du temps en utilisant des fonctions prédéfinies pour résoudre un problème (et se concentrer sur les aspects mathématiques par exemple). Considérons le programme suivant :

Script 1.4 – Etude graphique d'une suite récurrente

```
import numpy as np
import matplotlib.pyplot as plt
import math

def suiterec(f,u,n):
    x=np.linspace(0,5,200)
    y=f(x)
    U=np.zeros(n)
    U[0]=u
    for k in range(1,n):
        U[k]=f(U[k-1])
    Abscisse=np.zeros(2*n)
    Ordonnee=np.zeros(2*n)
```

```
    Abscisse[0]=u;
    Ordonnee[0]=u;
    for j in range(1,n):
        Abscisse[2*j-1]=U[j-1]
        Ordonnee[2*j-1]=U[j]
        Abscisse[2*j]=U[j]
        Ordonnee[2*j]=U[j]
    Abscisse[2*n-1]=U[n-1]
    Ordonnee[2*n-1]=f(U[n-1])
    return (x,y,Abscisse,Ordonnee,U)


def sin(x):
    return np.sin(x)
u=2.0
n=10
(x,y,Abs,Ord,U)=suiterec(sin,u,n)

plt.plot(x,y,'b-',x,x,'k-',Abs[1:],Ord[1:],'red',label='$u_{n+1}=f(u_n)$')
plt.ylim(ymin=0.)
plt.xlabel('$u_n$')
plt.ylabel('$f(u_n)$')
plt.title("suite recurrente")

plt.savefig("suite_rec.png")
plt.show()
```

Remarque 2.2. Les dollards dans le code précédent correspondent à du code TEXafin de donner une allure graphique aux Mathématiques ; vous pouvez bien entendu les omettre.

***Exercice* 2**
Commentez ligne par ligne le programme précédent.

***Exercice* 3**
À l'aide du programme précédent conjecturer le comportement de la suite $(\sin^{\circ n}(x))_{n\in\mathbf{N}}$ pour tout $x \in \mathbf{R}$. Le démontrer ensuite mathématiquement. On notera $u_n(x) = \sin^{\circ n}(x)$ pour tout $x \in \mathbf{R}$ on a donc $u_0(x) = \sin x$.

3. Listes

Une liste est une suite finie d'éléments pouvant être de types différents. Ces éléments sont modifiables (ou mutables), contrairement aux tuples. On peut de plus ajouter ou enlever des éléments à une liste, ce qui permet de représenter des structures de données évoluant au cours du temps.

3— 1. Définition d'une liste

Une liste peut être définie par énumération :

```
L=[3,7,42]
```

ou en utilisant l'instruction `range`. On peut également la définir en compréhension (comme un ensemble) :

```
M=[i**2 for i in range(10)]
```

Que donne le résultat de `L+M` et `[0,1]*10` ?

***Exercice* 4**
Exécuter et commenter le script suivant :

Script 1.5 – Construction itérative d'une liste

```
L=[]
for k in range(10):
    L.append(k)
```

3— 2. Opérations sur les listes

Etant donnée une liste L, on dispose des opérations suivantes :

Commande	Effet
`len(L)`	longueur
`L[0]`	premier élément
`L[-1]`	dernier élément
`L[i:j]`	liste extraite des éléments d'indices entre i (inclus) et j (exclus)
`L[i:]`	liste extraite à partir de l'indice i (inclus)
`L.append(v)`	ajoute l'élément v à la fin de la liste
`L.extend(s)`	ajoute la liste s à la fin de la liste
`L.instert(i,v)`	insert l'objet v à l'indice i
`L.pop()`	supprime le dernier élément et retourne l'élément supprimé
`L.reverse()`	retourne la liste
`del L[i]`	supprime l'élément d'indice i

Remarque 3.1 – Copie d'une liste d'éléments dans une autre. Pour recopier une liste `A` **contenant des entiers** dans une autre liste `B`, on écrira

```
B=A[:] # ou B=list(A)
```

L'instruction `B=A` est en effet impropre, car en modifiant B, la liste A sera également modifiée.
Le phénomène précédent ne se produit en revanche **pas** pour des entiers. En effet :

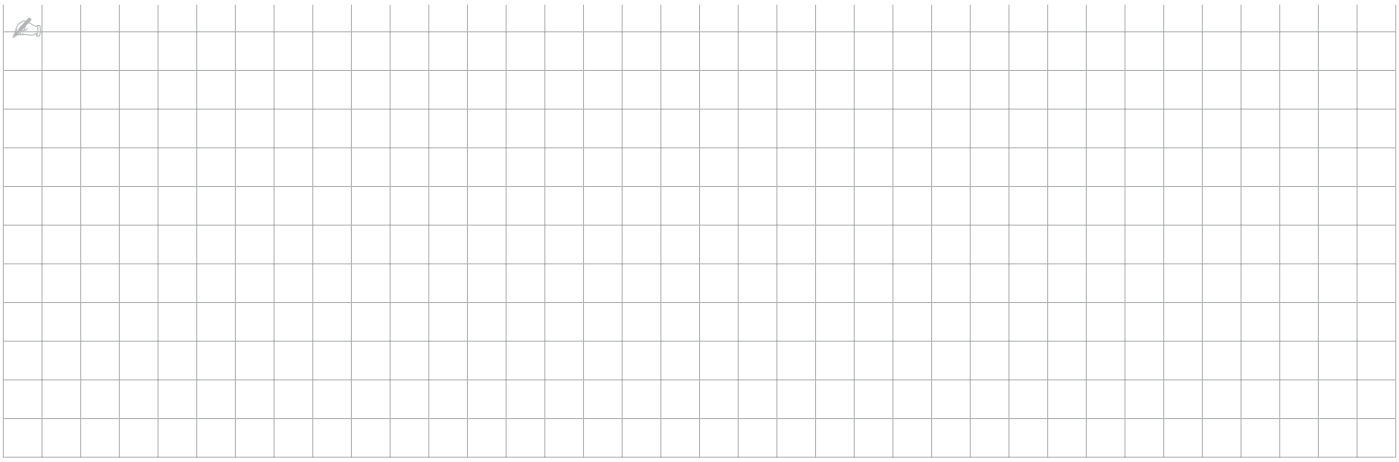
```
a=1
b=a
a=2
b
```

à l'issue de ce code l'entier b vaut toujours 1...

***Exercice* 5 – Maximum d'une liste à la main**
Écrire une fonction `maxi(L)` qui recherche le maximum de la liste L et renvoie tous les indices où ce maximum est atteint.

SOLUTION de l'Exercice 5.

```
def maxi(L):
    x=L[0]
    M=[]
    for i in range(len(L)):
        if L[i]>x:
            x=L[i]
    for i in range(len(L)):
        if L[i]==x:
            M+=[i]
    return x,M
```



***Exercice* 6**

□ **1**— Écrire une fonction `appartient(x,L)` qui détermine si la liste L contient l'élément `x`. Notez que cette fonction existe déjà : `x in L` renvoie le même résultat.
□ **2**— Écrire une fonction `indice(x,L)` qui renvoie le premier indice où l'élément x apparait dans la liste L, et `None` si L ne contient pas `x`. Notez que cette fonction existe déjà : `L.index(x)` renvoie le même résultat.

SOLUTION de l'Exercice 6.
□ **1**—

```
def appartient(x,L):
    for y in L:
        if y==x:
            return True
    return False
```

□ **2**—

```
def indice(x,L):
    if appartient(x,L)==False:
        return None
    else:
        k=0
        for i in range(len(L)):
            if L[i]==x:
                return i
```

4. Chaînes de caractères, tableaux & matrices

4— 1. Chaînes de caractères

Les chaînes de caractères (type `string` sont des listes de caractères (lettres de l'alphabet, chiffres, symboles). On les note entre guillements ou apostrophes :

```
ch1='blablabla'
ch2="toto"
```

On accède à chacun des caractères comme pour une liste : `ch1[0]` renvoie `'b'`, et on dipose des fonctions de concaténation (`ch1+ch2`), de longueur (`len`) et d'extraction de sous-chaîne (`ch1[3:6]`).

Exercice 7

□ **1**— Écrire une fonction `recherchemot(m,t)` qui recherche si le mot `m` est présent dans le texte `t`, et qui renvoie la position de la première occurence du mot s'il est présent, et `None` sinon.
□ **2**— On dit qu'une chaîne de caractère code une séquence `t` d'ADN si elle est composée uniquement des lettres `'A','T','G','C'` et qu'un *codon stop* est un triplet du type `TAA,TAG` ou `TGC`.
Écrire une fonction `codonstop(t)` qui renvoie `True` si la séquence `t` contient un codon stop et `False` sinon.

SOLUTION de l'Exercice 7.

```
def recherchemot(m,t):
    l=len(m)
    if len(m)>len(t):
        return None
    else:
        for i in range(len(t)-l+1):
            if m==t[i:i+l]:
                return True
    return None
```

□ **1**—

```
def codonstop(t):
    return recherchemot("TAA",t) or recherchemot("TAG",t) or recherchemot("TGC",t)
```

4— 2. Généralités sur les tableaux et matrices

On utilisera la bibliothèque `numpy`. Les tableaux bidimensionnels sont de type `array` (en réalité, ce sont des tableaux de tableaux unidimensionnels représentant chacune des lignes). On accède à l'élément de M situé à la ligne i et dans la colonne

j par `M[i,j]`.

Remarque 4.1. On pourrait utiliser aussi des listes de listes. L'avantage du type `array` est qu'on accède à toute une batterie de fonctions matricielles déjà définies (rang, recherche d'inverse, résolution de système linéaire, *etc.*).

Si M possède n lignes et p colonnes, elles sont numérotées de 0 à $n-1$ (*resp.* $p-1$), comme pour les listes en somme.

```
import numpy as np
M=np.array([[1,2,3],[4,5,6]])
D=np.diag([1,2,3])
np.size(M)
np.size(M,0)
np.size(M,1)
np.transpose(M)
np.dot(M,D)
2*M
np.linalg.inv(D)
np.linalg.matrix_rank(M)
```

Exercice 8

□ **1**— Résoudre le système suivant avec Python :

$$\begin{cases} 2x & + & 2y & - & 3z & = & 2 \\ -2x & - & y & - & 3z & = & -5 \\ 6x & + & 4y & + & 4z & = & 16 \end{cases}$$

On pourra utiliser la fonction `linalg.solve` de la bibliothèque `numpy` et contrôler le résultat avec son propre calcul.
□ **2**— Si l'on souhaitait concevoir un programme qui résout un système linéaire, quelles sont les fonctions que l'on devrait programmer ?
□ **3**— Résoudre ce système à la main.

SOLUTION de l'Exercice 8.
□ **1**—

```
import numpy as np
A=np.array([[2,2,-3],[-2,-1,-3],[6,4,4]])
b=np.array([[2],[-5],[16]])
X=np.linalg.solve(A,b)
```

La console renvoie $\begin{pmatrix} -14 \\ 21 \\ 4 \end{pmatrix}$.
□ **2**— Il faudrait concevoir : une fonction qui recherche le premier pivot dans une matrice (en commençant par la première colonne par exemple), puis une autre qui élimine une inconnue dans les lignes restantes (cela suffit puisque l'on recommence ensuite avec les sous-matrices extraites).
□ **3**—

Exercice 9 – Étude d'une matrice

Pour tout $n \in \mathbf{N}^*$, on appelle matrice de Hilbert d'ordre n la matrice H_n de terme général $a_{i,j} = \dfrac{1}{i+j-1}$.

□ **1**— Écrire une fonction `hilbert(n)` qui renvoie la matrice H_n.
□ **2**— Étudier le rang de H_n avec Python.
□ **3**— Conclusion ?

SOLUTION de l'Exercice 9.
□ **1**—

```
import numpy as np

def hilbert(n):
    A=np.zeros([n,n])
    for i in range(n):
        for j in range(n):
            A[i,j]=1/((i+1)+(j+1)-1)
    return A
```

□ **2**— On tape simplement :

```
np.linalg.matrix_rank(hilbert(10))
```

□ **3**— En testant pour les valeurs de $n \leqslant 10$, on remarque que la matrice de Hilbert semble de rang n. En particulier elle est donc inversible.
Cependant dès que $n > 10$, on voit que les rangs ne sont plus égaux à n. On pourrait ainsi en déduire que `hilbert(n)` n'est plus inversible pour ces n, sauf que mathématiquement nous sommes capable de démontrer le contraire. En fait, la responsable dans l'histoire est la méthode de calcul du rang utilisée par Python (gardez donc un regard critique des résultats). Pour certaines matrices dites *mal conditionnées* la méthode numérique échoue.

Remarque 4.2. Jusqu'à présent, les résolutions d'équations ou de systèmes sont des résolutions numériques, c'est-à-dire fondées sur des calculs approchés (avec des flottants). Il est possible de résoudre *formellement* une équation ou un système avec `Python` (et de réaliser la plupart des manipulations algébriques usuelles).
Beaucoup de manipulations algébriques usuelles peuvent être codées avec `sympy`.

Script 1.6 – Calcul formel avec `sympy`

```
import sympy as sm
#Déclaration des inconnues
x=sm.Symbol('x')
y=sm.Symbol('y')
solve(x**4 - 1, x)
((x+y)**2).expand()
solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])
apart(1/( (x+2)*(x+1) ), x)
B=Matrix([[1,2,1],[0,1,2],[1,0,1]])
B.inv()
```

4— 3. Applications à l'imagerie

Le langage `Python` dispose de bibliothèques de fonctions adaptées au traitement des images. On se propose ici d'en faire un rapide aperçu, le but étant de réutiliser dans la mesure du possible des fonctions prédéfinies.

Une image est représentée informatiquement par un tableau T de pixels : le contenu du pixel en haut à gauche de l'image est contenu en $T[0,0]$ et celui en bas à droite est contenu en $T[n-1,p-1]$. Une image en niveaux de gris est un tableau d'entiers entre 0 et 255, et se manipule comme tel.

Script 1.7 – Image en noir et blanc

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import scipy.misc

# image en niveaux de gris
l=scipy.misc.ascent() # Image démo
plt.gray() # Affiche l'image en niveaux de gris
plt.imshow(l)
plt.show()
```

Observer le résultat de la fonction suivante sur l'image ascent.

Script 1.8 – Image en noir et blanc

```
def floutage(image):
    (h,l)=image.shape
    res=image.copy()
    for i in range(1,h-2):
        for j in range(1,l-2):
            voisinage=image[i-1:i+2,j-1:j+2]
            res[i][j]=int(np.sum(voisinage)/9)
    return res
```

On peut appliquer de nombreux filtres à une image : flou gaussien, laplacien ...Pour utiliser une image présente dans un fichier, on doit pouvoir se placer dans le répertoire adéquat et pouvoir utiliser les commandes classiques du terminal.

Script 1.9 – Manipulation de fichiers et de répertoires

```
import os # Permet d'accéder fonctions relatives au système d'exploitation
os.getcwd() # Récupère le répertoire courant
os.chdir("\user\home\")#Change le répertoire courant
os.listdir()#Liste le contenu du répertoire courant
```

On suppose que le fichier `chaton.png` est présent dans ce répertoire (en TP, remplacer le chaton par votre photo préférée...).

Une image en couleurs est un tableau dont la case $T[i,j]$ est un tableau contenant 3 nombres nécessaires pour coder la couleur du pixel. La première valeur correspond au rouge, la seconde au vert et la troisième au bleu. La couleur obtenue est donc la superposition de ces trois couleurs fondamentales.

Script 1.10 – Chargement et utilisation d'un fichier png

```
img=mpimg.imread("chaton.png")
plt.colors()
plt.imshow(img)
plt.show()

img.shape
```

Les images se manipulent comme des tableaux usuels. On prendra garde à *copier* les tableaux avant de les modifier.

Script 1.11 – Manipulation d'une image stockée dans un tableau

```
chatonbizarre=img.copy()
sauv=chatonbizarre[0:150,0:200,:].copy()
chatonbizarre[0:150,0:200,:]=chatonbizarre[150:,200:,:]
chatonbizarre[150:,200:,:]=sauv
plt.imshow(ichatonbizarre)
plt.savefig("chatonbizarre.png")
plt.show()
```

Remarque 4.3. Si l'on veut effectuer des manipulations plus complexes sur les images, on peut conseiller l'utilisation de la bibliothèque `Image` contenue dans Python Image Library (PIL) disponible ici : http://www.pythonware.com/products/pil/ . Elle est installée par défaut par exemple avec la distribution Winpython de Windows http://winpython.sourceforge.net/ .

TP ❷ — Récursivité & Tris

♣ Plan du TP

➥ Résumé du TP

L'objectif est de voir la notion de fonction récursive (*i.e.* une fonction dont le résultat dépend d'elle-même) dans un premier temps, ce principe sera à la base du tri rapide que nous verrons ensuite. Dans un second, les principaux principes de tris de listes au programme de BCPST. Nous analyserons enfin leur complexité pour mesurer leur efficacité.

Il est fortement conseillé, à l'issue de chaque T.P. de conserver votre fichier `.py` *sur une clé USB afin de le retravailler chez vous.*

1. Récursivité

Une fonction est dite récursive lorsqu'elle s'appelle elle-même. En mathématiques, on rencontre cette notion lors de la définition d'une suite récurrente. La suite $(n!)_{n\in\mathbf{N}}$ suit une relation de récurrence simple, nous pouvons donc la coder de manière récursive.

Script 2.1 – Fonction factorielle récursive

```
def factrec(n):
    assert(n>=0 and type(n)==int)
    if n==0:
        return 1
    else:
        return n*factrec(n-1)
```

⚠ **Attention** — *On ne peut faire appel à la fonction qu'avec des valeurs du paramètre strictement inférieures à la valeur du paramètre en entrée de la fonction ! Sinon, la fonction risque de ne pas terminer .... De même, on examinera soigneusement les cas de base (comme pour une récurrence mathématique).*

***Exercice* 1**
Écrire une fonction récursive `u(n)` qui calcule le $n^{\text{ème}}$ terme de la suite définie par $u_0 = 1, u_1 = 0$ et, pour tout $n \in \mathbf{N}$: $u_{n+2} = u_{n+1} + 2u_n$.

SOLUTION de l'Exercice 1.

```
def u(n):
    if n==0:
        return 1
    if n==1:
        return 0
    return u(n-1)+2*u(n-2)
```

***Exercice* 2 – Fibonacci**
Écrire une fonction récursive `F(n)` qui calcule le $n^{\text{ème}}$ terme de la suite définie par $u_0 = a, u_1 = b$ avec $a, b \in \mathbf{R}$ et, pour tout $n \in \mathbf{N}$: $F_{n+2} = F_{n+1} + F_n$.

SOLUTION de l'Exercice 2.

```
def fib(n,a,b):
        if n==0:
            return a
        elif n==1:
            return b
        else:
      return fib(n-1,a,b)+fib(n-2,a,b)
```

2. Complexité et terminaison

On cherche ici à faire l'étude théorique d'un algorithme, c'est-à-dire de démontrer (à l'aide d'un raisonnement par récurrence généralement s'il est récursif) qu'il donne le bon résultat, et d'évaluer le nombre d'opérations nécessaires pour y arriver.

2— 1. Terminaison et correction d'une fonction

Définition 2.1 – Algorithme se terminant. On dit qu'un algorithme *termine* lorsqu'il conduit à un résultat en un nombre fini d'opérations.

La terminaison n'est pas forcément garantie lorsqu'on utilise une boucle `while` ou une fonction récursive par exemple.

Définition 2.2 – Algorithme correct. On dit qu'un algorithme est *correct* lorsqu'il renvoie la valeur attendue.

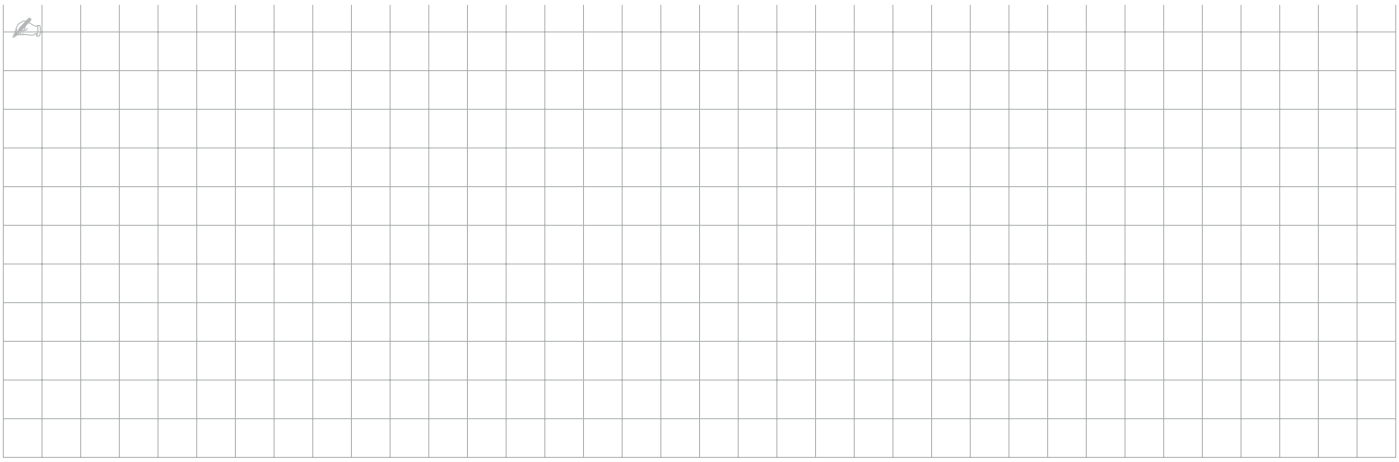
***Exercice* 3**
Montrer par récurrence que `factrec(n)` termine et renvoie la valeur $n!$.

Méthode – Montrer le caractère correct de l'algorithme. On justifie qu'il :

1. renvoie la bonne valeur pour $n = 0$,
2. renvoie la bonne valeur avant un appel récursif, et la renvoie toujours après.

La preuve est complètement analogue au raisonnement par récurrence en Mathématiques lorsqu'il est défini de manière récursive.

Méthode – Montrer que l'algorithme se termine. On exhibe une quantité/condition qui va venir terminer les appels récursifs.



***Exercice* 4**
Montrer que l'algorithme de la division euclidienne termine et est correct.

Script 2.2 – Division euclidienne pour les entiers naturels

```
a=int(input("entrer a: "))
b=int(input("entrer b: "))
q=0
r=a
while r>=b:
    q=q+1
    r=r-b
print("q=",q)
print("r=",r)
```

Méthode – pour montrer le caractère correct de l'algorithme. On justifie qu'il :

1. renvoie la bonne valeur pour $n = 0$,
2. renvoie la bonne valeur avant un appel récursif, et la renvoie toujours après.

Ici, nous justifions que l'identité de division euclidienne « $a_n = bq_n + r_n$ » est préservée à chaque étape.

Méthode – pour montrer que l'algorithme se termine. On exhibe une quantité/condition qui va venir terminer les appels récursifs.

SOLUTION de l'Exercice 4. Notons r_n, q_n la valeur des paramètres `p` et `q` après la n-ième opération. On aura : $r_0 = a$, $q_0 = 0$ et $\begin{cases} r_{n+1} &= r_n - b \\ q_{n+1} &= q_n + 1 \end{cases}$ avec $n \in \mathbf{N}$.
Notons $(\mathcal{P}_n)$: $a = bq_n + r_n$.
Initialisation. Comme $r_0 = a$ et $q_0 = 0$, on a $bq_0 + r_0 = a$ et $(\mathcal{P}_0)$ est vraie.

Hérédité. Supposons $(\mathcal{P}_n)$ vraie pour un entier n quelconque. Alors $bq_{n+1} + r_{n+1} = b(q_n + 1) + r_n - b = bq_n + r_n = a$ donc $(\mathcal{P}_{n+1})$ est vraie.

On a donc montré que pour tout entier $n \in \mathbf{N}$, l'algorithme fournit une identité du type $a = bq_n + r_n$. Puisque pour n assez grand on aura $r_n < b$ (sortie de la boucle `while`), on obtient bien la division euclidienne de a par b pour n assez grand.

2— 2. Complexité

Un problème peut avoir plusieurs solutions : en informatique, plusieurs algorithmes peuvent conduire au même résultat. On les différentie souvent en considérant leur complexité.

Définition 2.3 – Complexité.
1. On appelle *complexité temporelle* (liée au temps d'exécution de l'algorithme) d'un algorithme le nombre d'opérations élémentaires (additions, multiplications, comparaisons, échanges dans un tableau) nécessaire à son exécution.
2. On appelle *complexité spatiale* d'un algorithme l'occupation en mémoire nécessaire à son exécution.

Remarque 2.1. Lorsqu'il s'avère impossible de prédire la complexité en fonction des entrées (en fonction de la liste à trier par exemple), nous sommes obligés de recourir à un calcul de complexité *dans le pire des cas* d'une part, et/ou *dans le meilleur des cas* d'autre part. On peut aussi calculer une *complexité en moyenne* mais nous n'en parlerons pas dans la suite.

Cas des algorithmes de tris.

Nous exprimerons le plus souvent les complexités comme des suites (C_n) indexées par n où n est la taille de la liste donnée en entrée.

Définition 2.4. On dit que la *complexité d'un tri* est en $O(u_n)$ où (u_n) est une suite numérique, si $C_n \underset{n\to\infty}{=} O(u_n)$.

Dans la suite, nous omettrons la notation $n \to \infty$ en-dessous du signe égal. Par exemple, un tri en $O(\ln n)$ est un tri pour lequel il existe $K \in \mathbf{R}^+$ vérifiant :

$$\forall n \in \mathbf{N}, \quad |C_n| \leqslant K \ln(n).$$

Au niveau de la complexité spatiale, il s'agit de répondre à la question suivante : pour trier un tableau de n nombres, a-t-on besoin de recopier ce tableau ou peut-on le trier *en place*, *i.e.* directement à partir de la liste de départ ?

***Exercice* 5 – Exponentiation rapide**

Script 2.3 – Exponentiation rapide pour x^n

```
1 def expo_rapide(x,n):
2     if n==0:
3         return 1
4     else:
5         if n%2==0:
6             return expo_rapide(x,n//2)**2
7         else:
8             return expo_rapide(x,n//2)**2 * x
```

Expliquer l'algorithme d'exponentiation rapide et calculer sa complexité temporelle. *On pourra montrer que si* $2^k \leqslant n < 2^{k+1}$ *avec* $k \in \mathbf{N}$ $(\mathcal{P}_k)$, *alors il reste* $k+1$ *itérations à effectuer.*

SOLUTION de l'Exercice 5. Pour $k = 0$ alors $n = 1$, une seule itération suffit.
Supposons que $(\mathcal{P}_k)$ est vraie pour un certain entier k, et soit n tel que : $2^{k+1} \leqslant n < 2^{k+2}$. Alors on établit facilement que :
$2^k \leqslant \dfrac{n}{2} < 2^{k+1}$ en écrivant l'identité $n = 2*(n//2) + (n\%2)$ puis en encadrant et en divisant par deux.
Donc il reste $k+1$ itérations pour $n//2$ d'après l'hypothèse de récurrence et donc $k+2$ itérations au total.
Cette propriété nous permet d'établir que le nombre d'itérations k est borné par $\lfloor \log_2(n) \rfloor$, comme pour chacune d'entre elles il y a au plus deux multiplications, on en déduit que : $\boxed{C_n = O(\lfloor \log_2(n) \rfloor) = O(\ln(n))}$.

Remarque 2.2 – Comparaison avec l'élévation à la puissance n, version naïve. Pour calculer 2^n de façon naïve, on a besoin de $n-1$ multiplications.

Exercice 6 – Algorithme de Horner

Soit $P \in \mathbf{R}_n[X]$ un polynôme à coefficients réels de degré au plus n. On remarque que la fonction polynomiale associée évaluée en x est $P(x) = \sum_{k=0}^{n} a_k x^k$ s'écrit aussi

$$P(x) = (\ldots (a_n x + a_{n-1})\, x + a_{n-2})\, x + \ldots)\, x + a_0.$$

□ **1**— Écrire une fonction `horner(P,x)` prenant en paramètre la liste P des coefficients du polynôme et un réel x et qui calcule $P(x)$.
□ **2**— Calculer la complexité temporelle en terme de nombre de multiplications.

SOLUTION de l'Exercice 6.
□ **1**—

```
def horner(P,x):
    #de manière non récursive, on commence par les parenthèses intérieures : d'où reversed()
    assert(P !=[])
    S=0
    for a in reversed(P):
        S=a+x*S
        print(a," ",S)
    return S
```

On peut aussi préférer une version récursive :

```
def horner(L,a):
    if len(L)==0:
        return(0)
    else:
        b=L[len(L)-1]
        L1=L[0:len(L)-1]
        return horner(L1,a)*a+b
```

□ **2**— Notons n la longueur du polynôme donné en entrée et (C_n) la complexité temporelle en terme de multiplications. On montre facilement que cette suite vérifie : $C_n = C_{n-1} + 1$ pour tout entier $n \in \mathbf{N}$ puisque l'on réalise une multiplication à chaque appel récursif. On a ici une brave suite arithmétique de raison un

qui fournit une complexité en $O(n)$ qui est donc meilleure que la méthode naïve détaillée ci-dessous.

Remarque 2.3 – Comparaison avec l'évaluation naïve. La solution alternative bête serait de calculer séparément chacun des termes $a_k x^k$ puis de les sommer. On obtiendrait donc une complexité en $n = \deg P$ donnée par :

$$0 + 1 + \ldots + \underbrace{(1 + (n-1))}_{\substack{k-1 \text{ multiplications dans } x^k \\ +1 (\times a_k)}} = \frac{n(n+1)}{2} = O(n^2)\,.$$

On omet ici les additions, qui seraient au nombre de n, puisque $O(n^2) + n \underset{n\to\infty}{=} O(n^2)$. L'algorithme d'Horner possède donc une complexité linéaire, bien meilleure que la complexité quadratique que l'on vient de constater.

3. Tris

L'étude des tris est une partie incontournable de l'algorithmique, car elle permet de mettre en oeuvre différentes stratégies (récursivité, diviser pour régner) afin de trier des ensembles d'éléments. Nous serons amenés à étudier la complexité temporelle et spatiale de ces algorithmes. Pour commencer, mettons en place un petit test afin de savoir si une liste est triée ou non.

Une liste est-elle triée ?

***Exercice* 7**
Écrire une fonction `est_trie(L)`, qui prend en paramètre une liste `L` et qui renvoie `True` si le tableau est trié et `False` sinon.

SOLUTION de l'Exercice 7.

```
def est_triee(L):
    x=L[0]
    for i in range(1,len(L)):
        if L[i]<x:
            return False
        x=L[i]
    return True

#Ou plus simplement
def est_triee_simple(L):
    return L.reverse()==L
```

3— 1. Tri par insertion

Le tri le plus bête qui soit : le tri par insertion. Son avantage réside dans la simplicité (même si le tri rapide est également très simple à implémenter), en revanche il est extrêmement lent.

Principe et script.

L'algorithme principal du tri par insertion est un algorithme qui insère un élément dans une liste d'éléments déjà triés (par exemple, par ordre croissant).
Imaginez un joueur de cartes qui dispose de cartes numérotées. Il a des cartes triées de la plus petite à la plus grande dans sa main gauche, et une carte dans la main droite. Où placer cette carte dans la main gauche de façon à ce qu'elle reste triée ? Il faut la placer après les cartes plus petites, et avant les cartes plus grandes.
Pour trier entièrement un ensemble de cartes dans le désordre, il suffit alors de placer toutes ses cartes dans la main droite (la main gauche est donc vide), et d'insérer les cartes une à une dans la main gauche, en suivant la procédure ci-après.

Au départ, la main gauche est vide, donc elle bien triée. À chaque fois que l'on insère une carte depuis la main droite vers la main gauche, la main gauche reste triée, et la main droite (l'ensemble des cartes non triées) perd une carte. Ainsi, si la main droite comprenait au départ N cartes, en N insertions, on se retrouve avec 0 carte dans la main droite, et N cartes, triées, dans la main gauche : on a bien trié notre ensemble de cartes.

Script 2.4 – Tri par insertion

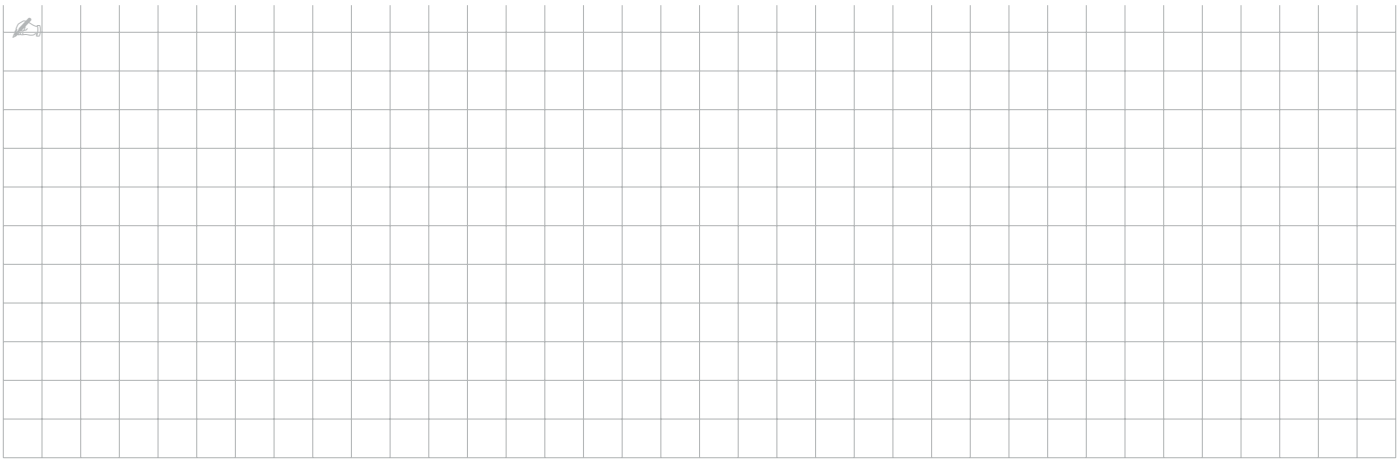
```
1  def Tri_insertion(L):
2          for k in range(1,len(L)):
3                  temp=L[k]
4                  j=k
5                  while j>0 and temp<L[j-1]:
6                          L[j]=L[j-1]
7                          j=j-1
8                          L[j]=temp
9          return L
```

Exercice 8

- **1**— Trier une liste de cinq nombres à la main à l'aide de cet algorithme.
- **2**— Le tri se fait-il en place ?

Quelques mots sur la complexité.

Si n désigne la taille de la liste à trier, la complexité en terme de nombre de comparaisons du tri par insertion est $O(n^2)$ dans le pire cas (et linéaire dans le meilleur cas). Justifiez l'affirmation dans le pire des cas.



SOLUTION de l'Exercice 8. Dans le pire cas, atteint lorsque le tableau est trié à l'envers, l'algorithme effectue de l'ordre de $n^2/2$ affectations et comparaisons. Si le tableau est déjà trié, il y a $n-1$ comparaisons et au plus n affectations.

3— 2. Tri à bulles

Principe et script.

Le tri *à bulles* ou tri *par propagation* est un algorithme de tri. Il consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air, disons de crémant, qui remonteraient rapidement à la surface d'un liquide.
Le tri à bulles est souvent enseigné en tant qu'exemple algorithmique, car son principe est simple. Mais c'est le plus lent des algorithmes de tri communément enseignés, et il n'est donc guère utilisé en pratique.

Script 2.5 – Tri à bulles

```
def Tri_bulles(L):
    n=len(L)
    echange_ok=False
    while echange_ok==False:
        echange_ok=True
        for j in range(0, n-1):
            if L[j]>L[j+1]:
                #non ordonné entre j et j+1
                echange_ok=False
                L[j],L[j+1]=L[j+1],L[j]
                #à ce stade on a encore eu quelque chose à permuter, donc on parcourt à nouveau L
                    avec la boucle for
         n=n-1
    return None
```

***Exemple* 3.1.** Par un exemple, voici ce que donne le tri à bulles sur la liste `[5,1,3,7,2]` :

1. Première étape :
$$[5,1,3,7,2] \to [1,5,3,7,2] \to [1,3,5,7,2] \to [1,3,5,7,2] \to [1,3,5,2,7],$$
2. Deuxième étape :
$$[1,3,5,2,7] \to [1,3,5,2,7] \to [1,3,5,2,7] \to [1,3,2,5,7],$$
3. Troisième étape :
$$[1,3,2,5,7] \to [1,3,2,5,7] \to [1,2,3,5,7],$$
4. Quatrième étape :
$$[1,2,3,5,7] \to [1,2,3,5,7].$$

On constate donc bien l'extrême lenteur de cet algorithme sur l'exemple précédent.

***Exercice* 9**
Que fait la fonction ci-dessus ?

Quelques mots sur la complexité.

Si n désigne la taille de la liste à trier, alors la complexité en terme de nombre de comparaisons dans le pire des cas est en $O(n^2)$. Pourquoi ? À quoi correspond le pire cas ?

3— 3. Tri rapide, ou *quicksort*

Principe.

Le tri rapide est un algorithme récursif basé sur le principe « diviser pour régner ». Etant donnée une liste L, on commence par choisir le premier élément comme *pivot*, et on sépare la liste entre deux sous-listes : la première contient des éléments inférieurs ou égaux au pivot, et la seconde contient des éléments supérieurs (strictement) au pivot. Puis on applique le tri rapide aux deux sous-listes obtenues.

***Exercice* 10**
Écrire alors une fonction récursive `TriRapide(L,gauche,droite)`.

SOLUTION de l'Exercice 10.

```
def quicksort(L):
    if L == []:
        return []
    else:
        pivot = L[0]
        L1 = []
        L2 = []
        for x in L[1:]:
            if x<pivot:
                L1.append(x)
            else:
                L2.append(x)
        return quicksort(L1)+[pivot]+quicksort(L2)
```

Quelques mots sur la complexité.

Si n désigne la taille de la liste à trier, alors la complexité en terme de nombre de comparaisons du tri rapide est en moyenne (et à une constante près) en $\boxed{O(n\log(n))}$ comparaisons/ affectations, et en $\boxed{O(n^2)}$ comparaisons/affectations dans le pire cas.
On notera qu'il est possible de démontrer que l'ordre de grandeur optimal d'un tri est de $n\log(n)$ comparaisons, on est donc en présence d'un tri optimal. Justifier la complexité du tri rapide dans le pire des cas.

SOLUTION de l'Exercice 10. Ainsi, dans le pire des cas (par exemple lorsque le tableau est initialement trié), alors `L1` est toujours vide et `L2` est de taille $n-1$, le coût C_n vérifie donc la relation de récurrence

$$C_0 = 1 = C_1, \quad C_n = C_{n-1} + n - 1.$$

D'où : $\boxed{C_n \sim \frac{n^2}{2}}$.

3— 4. Recherche dichotomique

***Exercice* 11**
Étant donnée une liste `L` *triée en ordre croissant*, écrire une fonction `recherchedicho(x,L)` qui renvoie, si elle existe, la position d'une occurence de x dans la liste `L`, et `None` sinon. *On pourra s'appuyer sur une méthode diviser pour régner en coupant le tableau en deux par un élément aléatoire, et en déterminant si* x *doit être cherché dans la partie gauche ou droite.*

SOLUTION de l'Exercice 11.

```
def recherche_dichotomique(element, liste_triee):
    if len(liste_triee)==1 :
        return 0
    m = len(liste_triee)//2
    if liste_triee[m] == element :
        return m
    elif liste_triee[m] > element :
        return recherche_dichotomique_recursive2(element, liste_triee[:m])
    else :
```

```
10          return m + recherche_dichotomique_recursive2(element, liste_triee[m:])
```

3— 5. Tri fusion

Le tri fusion est un tri également basé sur la méthode « diviser pour régner ». La différence avec le tri à bulles est qu'il n'est pas *en place* (*i.e.* il y a création d'une seconde liste avec copie, on ne modifie pas directement la liste de départ sans devoir en créer une autre), mais les deux sous-listes à trier à chaque appel récursif sont sensiblement de même taille. C'est un tri dit optimal en terme de comparaisons, car il nécessite de l'ordre de $n\log(n)$ comparaisons/affectations (à une constante près) pour trier un tableau de n nombres.

Principe et script.

À partir de deux listes triées, on peut facilement construire une liste triée comportant les éléments issus de ces deux listes (leur « fusion »). Le principe de l'algorithme de tri fusion repose sur cette observation : 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 (en fait, le plus petit des deux, à supposer qu'aucune des deux listes ne soit vide, sinon la réponse est immédiate). Ce procédé est appelé *fusion*.

***Exemple* 3.2.** Voici un exemple de tri d'une liste par fusion. Nous voyons que l'algorithme repose essentiellement sur la création d'une fonction `fusion` (flèches de la deuxième moitié du diagramme ci-dessous).

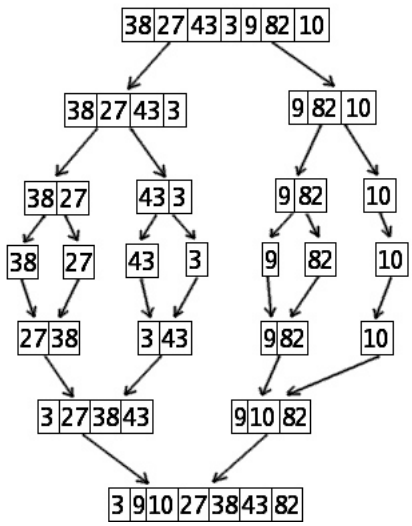


Figure 2.1 – Source Wikipedia

***Exercice* 12**
Analyser l'algorithme du tri fusion présenté ci-dessous et le faire fonctionner à la main sur la liste `[7,6,3,5,4,2,1,8]`

Nous proposons une version récursive du tri qui est beaucoup plus simple à mettre en place.

Script 2.6 – Tri fusion

```
1  def fusion(T1,T2):
2      '''renvoie la fusion des deux listes T1 T2
3      triées'''
4      if T1==[]:
5          return T2
6      if T2==[]:
7          return T1
8      if T1[0]<T2[0]:
9          return [T1[0]]+fusion(T1[1 :],T2)
10     else:
11         return [T2[0]]+fusion(T1,T2[1 :])
12
13 def trifusion(T):
14         if len(T)<=1:
15                 return T
16         T1=[T[x] for x in range(len(T)//2)]
17         T2=[T[x] for x in range(len(T)//2,len(T))]
18         return fusion(trifusion(T1),trifusion(T2))
```

Quelques mots sur la complexité.

Si n désigne la taille de la liste à trier, alors la complexité en terme de nombre de comparaisons du tri fusion est en $O(n\log(n))$ comparaisons/ affectations. Nous admettons ce fait.