CPGE
scientifiques

Nicolas Nguyen
Gweltaz Chatel

PRÉPAS SCIENCES

COLLECTION DIRIGÉE PAR **BERTRAND HAUCHECORNE**

IPT

INFORMATIQUE POUR TOUS

Algorithmique, programmation Python, ingénierie numérique et simulation

(bibliothèques Numpy et Scipy de Python)

Les 2 années
en 1 clin d'œil

ellipses

PRÉPAS
SCIENCES

collection dirigée par Bertrand Hauchecorne

IPT

Informatique pour tous

Algorithmique, programmation Python, ingénierie numérique et simulation

(bibliothèques Numpy et Scipy de Python)

Nicolas NGUYEN
Professeur au lycée F. Rabelais (Saint-Brieuc)

Gweltaz CHATEL
Professeur au lycée F. Rabelais (Saint-Brieuc)

ellipses

COLLECTION
PRÉPAS SCIENCES

Retrouvez tous les titres de la collection
et des extraits sur www.editions-ellipses.fr

ISBN 9782340-026346

32, rue Bargue 75740 Paris cedex 15

www.editions-ellipses.fr

Clique ici pour plus de livres : https://t.me/formations8

Avant-propos

La réussite en prépa nécessite une acquisition solide des connaissances et une capacité à les mettre en réseau ; elle exige par ailleurs d'en avoir une vision claire et de savoir les mobiliser rapidement.

Si les ouvrages de la collection Prépas Sciences répondent au premier besoin, ce formulaire d'**Informatique Pour Tous** (IPT) vous permettra de satisfaire à la deuxième exigence. Il présente en effet de manière claire et concise tous les thèmes et compétences du programme d'IPT des classes préparatoires aux grandes écoles scientifiques et permet de visualiser en un clin d'oeil :

- La base de connaissances théoriques
- Les algorithmes et programmes à connaître
- Les compétences indispensables.

Ces thèmes sont traités en 20 chapitres (10 de première année et 10 de deuxième année), tous organisés de la manière suivante :

- Description du thème scientifique et de la méthode numérique
- Algorithme et preuve de l'algorithme
- Implémentation en PYTHON.

Enfin, une boîte à outils mentionnant les fonctions PYTHON, natives ou issues de ses bibliothèques complètent utilement chaque chapitre.

En fin d'ouvrage, deux index très précis (un index général et un index des commandes) permettent de trouver rapidement et sans hésitation la notion cherchée.

Ainsi, ce formulaire sera le compagnon idéal pour réviser avant les concours, mais également comme ressource au quotidien pour préparer les séances de Travaux Pratiques d'informatique et de TIPE.

Clique ici pour plus de livres : https://t.me/formations8

Sommaire

Partie 1. Base de connaissance

Partie 2. Algorithmes de première année

Partie 3. Algorithmes de deuxième année

Index

Partie 1

Base de connaissance

LE SAVIEZ-VOUS ?

Ordinateur

Le mot *ordinator* existait déjà en latin. Chez les Romains, il désignait celui qui met en ordre, qui règle ou qui préside à la remise en ordre. Les premiers Chrétiens l'utilisaient pour nommer celui qui dirige les cérémonies : c'est pourquoi on parle encore de l'ordination des prêtres.

Francisé en ordinateur, il apparaît en français, dans le même sens, vers 1600 mais reste très peu employé. En 1955, la société IBM s'adresse à un latiniste, Jacques **Perret**, pour trouver un nom à ce que les Anglais appellent un computer, mot construit sur le latin *computare* qui signifie *calculer*. Celui-ci propose le terme d'ordinateur qui s'est désormais imposé dans notre langue.

On peut s'en réjouir, moins par le souci d'utiliser un mot français, que pour donner un sens plus large à la terminologie. En effet, le rôle de l'ordinateur dépasse largement celui du calculateur puisqu'il met en ordre et gère de multiples activités.

1. Types d'objets en Python

Nombres, listes, t-uplets, chaînes de caractères, ensembles

Tous les objets que manipule PYTHON ont un type défini à la fois par l'espace mémoire utilisé pour le stocker mais également par les règles de représentation, d'opération ou encore de construction qui s'y appliquent.

Les différents types

On peut distinguer deux types d'objets, les simples et les composés.

Types simples

Il s'agit essentiellement de types numériques.

`bool`	un booléen peut être *True* ou *False*
`int`	un entier relatif est exact en machine
`float`	un flottant est une valeur approchée d'un réel
`complex`	un complexe est de la forme a+jb

Types composés

Comme leur nom l'indiquent, ces objets sont des collections d'objets simples.

• On peut déterminer si un objet simple `a` appartient ou non à un objet composé `A` au moyen de la relation « `a in A` » qui renvoie un booléen. Ces objets composés des **itérables**.

- Lorsque ces objets composés sont ordonnés, on peut accéder directement à leurs composants à l'aide de « A[k] » qui renvoie l'élément de rang k. Ils sont alors dits **indexables**.
- Finalement, si on peut directement modifier les objets simples qui les composent, on dit qu'ils sont **mutables**.

Le tableau ci-dessous donne les types composés les plus fréquemment utilisés.

list	une liste est un objet mutable, indexable, itérable
tuple	un t-uplet est non mutable, indexable, itérable
str	une chaîne de caractères est non mutable, indexable, itérable
set	un ensemble est non mutable, non indexable mais itérable

Les types simples

Les tableaux ci-dessous résument les différents opérateurs sur les objets simples : opérations arithmétiques, comparaisons, constructions ou conversions.

Opérations

a+b	somme des deux nombres (int,float,complex)
a - b	différence des deux nombres (int,float,complex)
a*b	produit des deux nombres (int,float,complex)
a/b	quotient de deux flottants
a//b	quotient de la division euclidienne d'entiers
a%b	reste de la division euclidienne d'entiers
divmod(a,b)	quotient et reste de la division euclidienne
a**b pow(a,b)	a élevé à la puissance b
abs(a)	valeur absolue ou module de a

Comparaisons et opérations logiques

a==b	test d'égalité, renvoie un booléen
a !=b	test de différence, renvoie un booléen
a>b, a<b, a>=b, a<=b	tous ces tests de comparaison, respectivement $a > b$, $a < b$, $a \geqslant b$, $a \leqslant b$ renvoient un booléen
P and Q	conjonction de deux booléens
P or Q	disjonction de deux booléens
not P	négation d'un booléen

Conversions

int(a)	convertit en entier un flottant ou une chaîne
float(a)	convertit en flottant un entier ou une chaîne
complex(a,b)	convertit en complexe un couple de flottants
int(car,b)	donne la valeur de l'entier donné par son écriture car (de type chaîne) en base b
bin(a)	renvoie l'écriture en binaire de l'entier a
oct(a)	renvoie l'écriture en octal de l'entier a
hex(a)	renvoie l'écriture en hexadecimal de l'entier a

Les types composés

Définitions

• Une **liste** est une suite d'éléments quelconques (pas nécessairement de même type) séparés par des virgules et encadrée par des crochets. Par exemple [a, b, c] est une liste. Les listes sont itérables, indexables et mutables.

• Un **t-uplet** est une suite d'éléments quelconques (et pas nécessairement de même type) séparés par des virgules et encadrée par des

parenthèses (ou rien). Par exemple (a, b, c) et d, e, f sont des t-uplets. Les t-uplets sont itérables, indexables mais non mutables.

- Un **ensemble** est une collection d'objets séparés par des virgules et encadrée par des accolades. Ainsi, {a, b, c} est un ensemble. Dans un ensemble, ni l'ordre, ni les répétitions éventuelles des éléments ne changent l'ensemble. Les ensembles sont itérables mais non indexables et non mutables.
- Une **chaîne de caractère** est une suite de caractères quelconques encadrée par une paire de simples ou de doubles guillemets. Par exemple, "a b c" et 'd e f' sont des chaînes de caractères. Comme les listes, les chaînes de caractères sont itérables, indexables mais non mutables.

Les objets séquentiels que sont les listes, les t-uplets et les chaînes, sont toujours indexés à partir de 0. A[0] désigne le premier élément de la séquence, A[1] le deuxième, etc.

Opérations

len(A)	retourne la longueur de A (liste, t-uplet, chaîne, ensemble)
A[k]	accède à l'élément d'indice k de la séquence A (liste, t-uplet, chaîne)
A + B	opération de concaténation, juxtapose les deux séquences (liste,t-uplet,chaîne)
n * A	opération de duplication, juxtapose n (entier) fois la séquence A (liste,t-uplet,chaîne)
max(A)	retourne l'objet maximal de la collection A (liste, t-uplet, chaîne, ensemble)
min(A)	retourne l'objet minimal de la collection A (liste, t-uplet, chaîne, ensemble)
sum(A)	retourne la somme des éléments d'une collection A de nombres (liste, t-uplet, chaîne, ensemble)

Comparaisons entre objets

a in A	teste l'appartenance d'un objet à A (liste, t-uplet, chaîne, ensemble) et retourne un booléen
a not in A	teste la non appartenance d'un objet à A (liste, t-uplet, chaîne, ensemble) et retourne un booléen
A==B	teste l'égalité de A et B (listes, t-uplets, chaînes, ensembles), retourne un booléen
A != B	teste la non égalité de A et B (listes, t-uplets, chaînes, ensembles), retourne un booléen
A < B	test d'inégalité dans l'ordre lexicographique, lorsque A et B sont des séquences (liste, t-uplet, chaîne), retourne un booléen
A < B	test d'inclusion lorsque A et B sont des ensembles, retourne un booléen

Constructions et conversions

[a,b,c]	crée la liste d'objets a, b, c
[] ou list()	crée la liste vide
(a,b,c)	crée le t-uplet d'objets a, b, c
a,b,c	crée l'ensemble d'éléments a, b, c
set()	crée l'ensemble vide
"abc" ou 'abc'	crée la chaîne de caractères a, b, c
"" ou '' ou str()	crée la chaîne de caractères vide
list(A)	convertit en liste tout type d'objet composé (liste, t-uplet, chaîne, ensemble)
set(A)	convertit en ensemble. Élimine les répétitions et range les éléments par ordre croissant, si possible
str(A)	convertit en chaîne de caractères tout type d'objet composé (liste, t-uplet, chaîne, ensemble)

Variables

Notion de variable en Python

Pour mémoriser une valeur, on la stocke en mémoire. On représente l'adresse mémoire à l'aide d'un identificateur (chaîne de caractères). Le couple (identificateur, valeur) est appelé une **variable**.

Comme nom de la variable, on peut utiliser toute chaîne de caractères alphanumériques (qui ne commence pas par un chiffre) à l'exception de quelques mots réservés.

PYTHON distingue les majuscules des minuscules. Ainsi, les chaînes 'Aa' et 'aa' sont deux identificateurs différents.

Une variable peut contenir des données de n'importe quel type déjà présenté. En PYTHON, il n'est pas nécessaire de préciser le type de valeur que l'on va placer dans une variable. En fait, le type est déterminé seulement au moment de l'évaluation, c'est ce que l'on appelle un **typage dynamique**.

Affectation d'une valeur à une variable

Déclaration d'une variable

Avant d'utiliser une variable a, il faut toujours lui donner une valeur initiale. Pour attribuer une valeur à une variable, on utlise l'**opérateur d'affectation**, noté « = » en PYTHON.

```
>>> a = 10
```

Affectation d'une variable

Pour modifier la valeur d'une variable, on utilise l'opérateur d'affectation. On ne confondra pas cet opérateur avec une égalité mathématique. En effet, l'opérateur d'affectation est non symétrique, non transitif.

```
>>> x = 1
>>> x = x + a
```

Dans une affectation, on commence d'abord par évaluer l'expression de droite, puis on affecte cette valeur à la variable de gauche.

Affectations simultanées

PYTHON permet de réaliser, des affectations simultanées de plusieurs variables, de même type ou non :

```
>>> x,y,z = 'abc',6,[]
```

En particulier, il est aisé d'échanger le contenu de plusieurs variables en PYTHON :

```
>>> x,y,z = y,z,x
```

Opérations sur les variables

Comme une variable représente un objet d'un certain type, tous les opérateurs rencontrés (opérations, comparaisons, constructions ou conversions) s'appliquent également aux variables de ce même type.

On appelle **expression** toute opération composée qui met en jeu variables et constantes. Lorsque les composants sont de même type et les opérations bien définies, l'expression finale garde le même type. Lorsque les constantes et les variables sont de types hétérogènes, l'expression prend le type minimal dans lequel toutes les opérations sont compatibles.

Opérateurs avec assignation

Il est très fréquent d'avoir à incrémenter une variable, c'est-à-dire à augmenter sa valeur de 1. PYTHON permet de raccourcir l'instruction `x = x + 1` en `x += 1`. On dit que `+=` est un opérateur avec assignation.

x += y	équivaut à l'affectation x = x + y
x -= y	équivaut à l'affectation x = x - y
x *= y	équivaut à l'affectation x = x * y
x **=y	équivaut à l'affectation x = x ** y
x /= y	équivaut à l'affectation x = x / y
x //= y	équivaut à l'affectation x = x // y
x %= y	équivaut à l'affectation `x = x % y`

Égalités structurelle et physique

Pour tester l'égalité de deux objets, a et b on a vu l'opérateur de comparaison a == b. Une variable est définie comme une adresse sur le disque et une valeur enregistrée à cette adresse.

On peut donc distinguer deux sortes d'égalités pour des variables.

- x et y sont structurellement égales lorsqu'elles ont même valeur.
- x et y sont physiquement égales lorsqu'elles pointent vers la même adresse sur le disque.

Lorsque ces variables sont de type non mutable (types numériques, t-uplet, chaînes de caractères), ces deux notions d'égalité coïncident. En effet, par définition, on ne peut modifier la valeur d'une variable de type non mutable sans changer son adresse.

En revanche, lorsqu'il s'agit de listes, la différence existe : l'égalité physique entraîne l'égalité structurelle mais la réciproque est fausse.

x == y	teste l'égalité structurelle de x et y, retourne un booléen
x is y	teste l'égalité physique de x et y, retourne un booléen
id(x)	retourne l'adresse physique de la variable x

Listes en Python

Modes de construction

Pour construire une liste, plusieurs manières sont possibles.

L=[] ou L=list() L=[a,b,c]	crée la liste vide crée la liste d'objets a, b, c
list(A)	convertit en liste tout type d'objet composé (liste, t-uplet, chaîne, ensemble)
L=[] for i in iterable : . . . L = L + [expr[i]]	crée la liste par concaténation à partir d'une expression dépendant de l'indice i
L = n*[a] ou L= [a]*n	crée la liste L en dupliquant n fois la liste à un élément [a]
L = [expr[i] for i in iterable]	crée la liste L en compréhension à partir d'une expression dépendant de l'indice i
range(n) range(n,m) range(n,m,p)	crée la liste de n entiers consécutifs de 0 à $n-1$ crée la liste des entiers consécutifs de n à $m-1$ crée la liste des entiers partant de n de p en p sans atteindre m

Remarque : Comme les listes sont des séquences d'objets de tous types, on peut également construire et manipuler des listes de listes. Par exemple, l'instruction suivante permet de créer en compréhension une liste de listes

$$(L[i][j])_{\substack{0\leqslant i\leqslant n-1\\0\leqslant j\leqslant m-1}}$$

```
L=[[expr[j] for j in range(m)] for i in range(n)]
```

On peut également utiliser des tableaux T[i,j] du type ndarray de NUMPY.

Opérations sur les listes

Opérations communes pour les objets composés

len(L)	retourne la longueur de la liste L
x in L	teste l'appartenance de l'objet x à la liste L, retourne un booléen
L + M	concaténation de deux listes
n * L	duplication de la liste L
max(L)	retourne l'objet maximal de la liste L
min(L)	retourne l'objet minimal de la liste L
sum(L)	retourne la somme des éléments d'une liste de nombres L

Accès aux éléments et extraction de sous-liste

L[i]	accède à l'élément d'indice i de la liste L
L[-1]	accède au dernier élément de la liste L
L[i : j]	retourne la sous-liste des élements de L d'indice variant de i à j-1
L[i : j : k]	retourne la sous-liste des élements de L d'indice variant de i à j par pas de k sans atteindre j
L[i :]	retourne la sous-liste des élements de L d'indice supérieur ou égal à de i
L[: j]	retourne la sous-liste des élements de L d'indice strictement inférieur à j
L[:]	retourne la liste de tous les élements de L

Modifications d'une liste

Affectation à un élément ou à un bloc d'éléments

L[i]=a	remplace la valeur du terme d'indice i de la liste L
L[i : j]=M	remplace la sous-liste des élements de L d'indice variant de i à j-1 par la liste M qui n'a pas nécessairement la même longueur
L[i, i+1]=[]	supprime le terme d'indice i de la liste L

Copie d'une liste

Avant de modifier une liste L, il est parfois utile de créer une copie de celle-ci, c'est-à-dire une nouvelle liste qui contient les mêmes éléments que L.

M=L[:]	crée une copie de la liste L

«Nous ne savons pas où nous allons, mais du moins il nous reste bien des choses à faire.»

Alan Turing, mathématicien anglais.

2. Programmation en Python

Structures séquentielles, conditionnelles, itératives

Notion de programme

Cinq types d'instructions

On appelle **instruction** tout ordre de modification de l'état courant des variables. Les instructions sont les éléments constitutifs des programmes. Leur assemblage, dans un ordre précis conduit au résultat attendu.

On distingue cinq types d'instructions :

- La déclaration de variable qui rajoute cette variable à l'état du système ;
- L'affectation de variable qui modifie la valeur de la variable ;
- La séquence d'instructions qui exécute plusieurs instructions à la suite l'une de l'autre ;
- L'instruction conditionnelle qui sert à n'exécuter une instruction que dans certains états ;
- Les structures itératives qui exécutent plusieurs fois la même instruction.

Séquences d'instructions

En algorithmique, on délimite les groupes d'instructions par des mots clés **Faire** et **Fin Faire** par exemple, ce qui permet de considérer ce groupe comme une seule instruction.

En PYTHON, on groupe les instructions à l'aide de l'indentation (décalage par rapport à la marge de gauche). Deux instructions de même profondeur logique doivent avoir la même indentation.

```
en-tête1 :
      instruction
      en-tête2 :
          instruction
          ...
          instruction
      instruction
instruction
```

Instructions conditionnelles

Test simple

Une instruction conditionnelle n'est éxécutée que si une condition (de type booléen) est vérifiée par l'état courant.

```
if condition :
    instruction ou bloc d'instructions
```

Test avec alternative

Suivant la valeur (vraie ou fausse) d'une condition, on peut dévier le flot d'instructions :

```
if condition :
    instruction ou bloc d'instructions
else :
    instruction ou bloc d'instructions
```

Test imbriqués

On peut scinder le flot d'instructions suivant plusieurs cas :

```
if condition1 :
    instruction ou bloc d'instructions
elif condition2 :
```

```
    instruction ou bloc d'instructions
        ...
elif condition n :
    instruction ou bloc d'instructions
else :
    instruction ou bloc d'instructions
```

Expressions conditionnelles

PYTHON offre la possibilité de former des expressions dont l'évaluation est soumise à une condition. La syntaxe est la suivante :

```
expression1 if condition else expression2
```

Structures itératives

Boucles inconditionnelles (boucles for)

Pour répéter un certain nombre de fois une instruction, on utilise la construction suivante :

```
for i in iterable :
    instruction ou bloc d'instructions
```

On rappelle qu'un itérable est un objet de type liste, t-uplet, chaîne de caractères ou ensemble.

Boucles conditionnelles (boucles while)

Lorsqu'on doit répéter un bloc d'instructions un certain nombre de fois bien déterminé, ou plus généralement lorsqu'on souhaite parcourir un iterable, on effectuera une boucle `for`. Mais lorsque le nombre d'itérations n'est pas connu à l'avance et que la décision d'arrêter la boucle ne peut s'exprimer que par un test, on choisira la boucle `while`.

Pour répéter un bloc d'instructions tant qu'une condition est réalisée, PYTHON propose la clause `while` :

```
while condition :
    bloc d'instruction si condition est vraie
```

Remarques :

• Si condition est fausse dès le départ, le bloc qui suit n'est jamais parcouru.

• Dans la plupart des cas, le bloc qui suit l'instruction d'en-tête `while` agit sur la condition, de sorte que celle-ci, vraie au départ, devient fausse et provoque la sortie de la boucle.

• Lorsqu'on utilise une boucle conditionnelle, la question de sa terminaison après un nombre fini d'itérations se pose systématiquement.

Entrées sorties

Affectation par l'utilisateur

Il existe une expression particulière, `input()`, qui attend que l'utilisateur saisisse une chaîne de caractères au clavier. Cette expression est alors affectée à une variable. Lorsque le résultat attendu est d'un type autre que chaîne de caractères, une conversion est nécessaire.

```
>>> nom=input('Entrez votre nom :')
>>> age=int(input('Entrez votre age :')
>>> tai=float(input('Entrez votre taille en m :')
```

Affichage d'une valeur

Inversement, l'instruction `print` affiche à l'écran les instructions qui lui sont données en argument. Elle ne modifie pas l'état, mais seulement l'aspect de l'écran. On s'en sert souvent pour afficher des chaînes de caractères, mais elle peut afficher des expressions quelconques.

Fonctions et modules

Dans un même programme, une opération, ou une séquence d'opérations peut intervenir à plusieurs reprises. En ce cas, il est intéressant de définir une fonction qui exécute cette instruction (ou ce bloc d'instructions). Elle pourra être appelée plusieurs fois dans un script ou même dans plusieurs scripts.

Ainsi, une fonction en PYTHON est comme un sous-programme à l'intérieur du programme principal. Il se constituera comme le programme principal d'un en-tête et du corps de la fonction. Il pourra aussi posséder ses propres variables, que l'on qualifiera de **variables locales**.

Fonctions

Définition d'une fonction

Une fonction est définie par la donnée de :

- son nom ;
- ses paramètres formels (les variables auxquelles seront appliquées les instructions) ;
- éventuellement une valeur de retour, communiquée au programme principal à la fin d'exécution.

La syntaxe PYTHON pour la définition d'une fonction est la suivante :

```
def nom_de_fonction(liste de paramètres formels):
    """la documentation"""
    instructions
    return (valeur de retour)
```

Le mot-clé `def` annonce la définition d'une fonction. Il doit être suivi par le nom de la fonction et une liste entre parenthèses de paramètres formels suivie de deux-points ' :'. On trouve ensuite une documen-

tation (optionnelle), puis le corps de la fonction. Le mot-clé return interrompt le déroulement de la fonction et renvoie la valeur précisée.

On peut également définir une fonction de façon plus succincte à l'aide du mot-clé **lambda** lorsqu'elle est définie par une expression simple, comme une fonction mathématique.

```
f=lambda x :3*x**2-6*x-12
```

est équivalent à :

```
def f(x):
    return(3*x**2-6*x-12)
```

Appel d'une fonction

Lorsqu'on applique une fonction à des valeurs, il suffit de taper le nom de la fonction et de passer les paramètres effectifs en arguments.

```
>>> f(2)
-12
```

Variables locales

Lorsqu'on définit une fonction, on peut avoir besoin de variables. Tout comme les paramètres formels, elles ne servent qu'à l'exécution de la suite d'instructions qui mènent à la valeur de retour.

Fonctions récursives

Dans les paragraphes précédents, nous avons pris soin de distinguer la définition d'une fonction de son appel. Toutefois, rien n'interdit d'appeler une fonction lors de sa définition. C'est ce que l'on appelle une fonction **récursive**. Un exemple classique est d'une programmation récursive de la fonction factorielle.

```
def facto(n):
    if n==0:
        return(1)
```

```
    else :
        return(n*facto(n-1))
```

Comme pour une boucle conditionnelle, la terminaison d'une fonction récursive pose question. Tout d'abord, il est primordial que la fonction récursive possède un ou plusieurs cas d'arrêt (ici, lorsque n est nul).

En outre, pour établir la terminaison, on utilisera souvent une suite strictement décroissante d'entiers naturels, associés à chaque appel récursif de la fonction (voir p. 24).

Modules et packages

Les instructions import, from

PYTHON possède un très grand nombre de fonctions et de constantes. Certaines, dites **natives** sont disponibles directement, mais la plupart sont définies dans des modules, eux-mêmes regroupés en packages. Tous ces modules ne sont pas chargés automatiquement au démarrage de PYTHON. Pour utiliser leurs fonctions, il faut d'abord les importer. On dispose de plusieurs manières pour utiliser une fonction appelée « fonction » d'un module nommé « module ».

« Un ordinateur vous permet de faire plus de bêtises, beaucoup plus rapidement, que n'importe quelle autre invention dans l'histoire de l'humanité. À l'exception notable des armes à feu et de la tequila.»

Mitch Ratcliffe, journaliste américain.

import module module.fonction	charge tous les noms (fonctions, constantes) du module on peut utiliser la fonction du module avec ce préfixe
import module as md md.fonction	charge tous les noms du module et lui donne un nom plus court (alias) on peut utiliser sous ce nom la fonction du module
from module import fonction from module import *	on peut directement utiliser la fonction sans préfixe on peut utiliser directement toutes les fonctions du module sans préfixe

Packages et modules usuels

Voici quelques packages et modules parmi les plus fréquemment utilisés.

math	module avec les fonctions et constantes mathématiques de base (sin, cos, exp, π . . .)
random	module permettant la génération de nombres aléatoires
numpy	package notamment utile pour le traitement des tableaux multidimensionnels comme par exemple des vecteurs, matrices, images
scipy	package utile pour algèbre linéaire, intégration, équations différentielles, traitement du signal et imagerie, statistiques
matplotlib	module indispensable pour la représentation graphique
time	permet d'accéder à l'heure de l'ordinateur et aux fonctions gérant le temps

profile	permet d'évaluer le temps d'exécution de chaque fonction dans un programme
tkinter	interface Python avec Tk qui permet de créer des objets graphiques
pygame	permet de créer une interface graphique avec Python

Fonctions natives

Voici un résumé de fonctions directement accessibles en PYTHON.

abs(x)	valeur absolue ou module d'un nombre
bin(x)	convertit en binaire
chr(x)	caractère de code donné entre 0 et 255
complex(x,y)	convertit un couple de flottants en nombre complexe
divmod(x,y)	quotient et reste de la division euclidienne
eval(ch)	évalue la valeur numérique d'une chaîne de caractères
float(x)	convertit en flottant
help(name)	interroge l'aide en ligne sur un nom
hex(n)	convertit en hexadécimal un nombre
input(ch)	affiche une chaîne et récupère une entrée clavier
int(x)	convertit en entier
len(x)	longueur d'un objet composé
list(x)	convertit un objet en liste
max(x)	maximum d'un objet composé
min(x)	minimum d'un objet composé
ord(ch)	code (entre 0 et 255) d'un seul caractère
pow(x,y)	élève à la puissance
print(ch)	affiche une chaîne à l'écran

range(n,m,p)	construit un intervalle de valeurs
round(x,p)	arrondit au nombre le plus proche à une précision donnée
set(x)	convertit un objet en ensemble
sorted(x)	crée une copie triée d'un iterable
str(ch)	convertit en chaîne de caractères
sum(x)	calcule la somme des élément d'une séquence de nombres
type(x)	type d'un objet

Fonctions du module math

factorial(n)	nombre factorielle de n
floor(x)	partie entière d'un réel
frexp(x)	couple mantisse et exposant d'un nombre
gcd(x,y)	PGCD de deux entiers
exp(x)	exponentielle d'un nombre
log(x)	logarithme néperien d'un nombre
log2(x)	logarithme en base 2 de x
log10(x)	logarithme décimal de x
pow(x,y)	élève à la puissance
sqrt(x)	racine carrée
acos(x)	arc cosinus d'un nombre
asin(x)	arc sinus d'un nombre
atan(x)	arc tangente d'un nombre
cos(x)	cosinus d'un nombre
sin(x)	sinus d'un nombre
tan(x)	angente d'un nombre
degrees(x)	convertit en degrés un angle exprimé en radians
radians(x)	convertit en radians un angle exprimé en degrés

cosh(x)	cosinus hyperbolique d'un nombre
sinh(x)	sinus hyperbolique d'un nombre
tanh(x)	tangente hyperbolique d'un nombre
pi	constante π
e	constante e

Spécification, terminaison, correction, complexité

Preuve de programme

Faire une preuve de programme consiste à vérifier que le programme s'applique dans tous les cas prévus et conduit en temps fini au résultat attendu.

Une preuve de programme repose sur plusieurs étapes décrites brièvement ci-après.

- Spécification
- Terminaison
- Correction
- Complexité

Spécification

Un programme ou une fonction en PYTHON ressemble à une fonction mathématique comprenant un ensemble de départ, un ensemble d'arrivée, et un procédé qui transforme tout élément de l'ensemble de départ en un élément de l'ensemble d'arrivée.

La spécification d'un programme ou d'une fonction consiste à décrire le problème qu'il doit résoudre, le calcul à effectuer, et dans quelles conditions il doit le faire. Plus précisément, la spécification indique :

- quelles sont les données en entrée (leur type notamment), qu'elles soient saisies par l'utilisateur ou passée en paramètre lors de l'appel d'une fonction ;

- quelles sont les valeurs attendues en retour, leur type et à quoi elles correspondent.

Terminaison

Lorsque le programme ne comporte que des boucles inconditionnelles, il est certain de se terminer après un nombre fini de passages en boucle.

Cependant, lorsque le programme utilise une boucle conditionnelle while, il y a un risque de voir la boucle se répéter indéfiniment ! Il est donc nécessaire, lors de l'écriture du programme de prouver la **terminaison** de la boucle, c'est-à-dire qu'après un certain nombre d'itérations, qui dépendra évidemment des données, mais qui est toujours **fini**, la condition ne sera plus vérifiée.

L'argument est souvent lié à la propriété suivante :

Toute suite $(N_k)_{k\in\mathbb{N}}$ d'entiers naturels décroissante est finie, c'est-à-dire stationnaire.

Correction

Sachant que la boucle se termine, encore faut-il garantir qu'elle retourne le résultat escompté. Une des manières les plus efficaces pour tester la correction d'un programme reposant sur une boucle while ou for est d'établir un **invariant de boucle**, c'est-à-dire une propriété mathématique qui est préservée tout au long de l'exécution de la boucle.

Cette démarche est à rapprocher du raisonnement par récurrence : en connaissant les valeurs initiales des variables et à leur évolution au cours d'une itération, on peut en déduire des propriétés valides quel que soit le nombre d'itérations. Plus précisément :

Un **invariant de boucle** est une propriété mathématique qui :

- est vérifiée avant d'entrer dans la boucle ;

- si elle est vérifiée avant une itération, elle est aussi vérifiée après celle-ci ;
- le fait qu'elle soit vérifiée en sortie de boucle, garantit que le programme est correct.

Complexité

Efficacité d'un programme

On dit d'un programme qu'il est efficace d'une part si son exécution est rapide et d'autre part, si les ressources mémoire qu'il mobilise sont limitées. Bien sûr, ces deux critères sont liés : il est naturel que plus un programme traite de données, plus le temps d'exécution est important.

Opérations élémentaires

Il est difficile d'évaluer le temps d'exécution d'un algorithme car le résultat dépend des capacités de la machine et du langage de programmation choisi.

Par contre on peut assez aisément compter le nombre d'opérations élémentaires (OPEL) effectuées par celui-ci. Il peut s'agir

- d'une opération algébrique élémentaire : addition, soustraction, multiplication, division ;
- d'un test de comparaison : égalité, inégalité ;
- d'une affectation d'une variable ;
- d'un affichage ou retour d'un résultat.

En outre, ceci est indépendant du langage de programmation et des capacités de la machine !

Complexité

Le nombre d'opérations à effectuer dans un programme augmente avec le nombre de données à traiter. Pour savoir si un programme est efficace, il est intéressant de connaître le rapport entre le nombre de données et le nombre d'opérations surtout lorsque le nombre de données devient important.

Soit n le nombre de données d'un programme et $f(n)$ une fonction de n. On dit que la complexité d'un programme est un $\mathcal{O}(f(n))$ si son coût (en terme de nombre d'OPEL) est inférieur à $C.f(n)$, où C est une constante indépendante de n.

Le tableau ci-dessous indique un ordre de grandeur des temps d'exécution d'un problème de taille $n = 10^6$ sur un ordinateur à un milliard d'opérations par seconde.

Complexité	Nom courant	Temps
$\mathcal{O}(1)$	temps constant	1 ns
$\mathcal{O}(\ln(n))$	logarithmique	10 ns
$\mathcal{O}(n)$	linéaire	1 ms
$\mathcal{O}(n^2)$	quadratique	15 min
$\mathcal{O}(n^k)$	polynomiale	30 ans pour $k = 3$
$\mathcal{O}(2^n)$	exponentielle	$10^{300\,000}$ milliards d'années

« Il y a deux façons de faire la conception d'un logiciel. Une façon est de le rendre si simple qu'il n'y a selon toute apparence aucun défaut. Et l'autre est de le faire si compliqué qu'il n'y a pas de défaut apparent. »

Tony Hoare, informaticien britannique.

3. Python pour l'ingénierie numérique et la simulation

Tableaux avec Numpy

PYTHON propose non pas des tableaux mais des listes : elles peuvent contenir des objets de types divers (on dit qu'elles sont **hétérogènes**) et de taille variable.

Le type `nd.array` de NUMPY est constitué de « vrais » tableaux : homogènes et de taille fixe. Ainsi, leur manipulation est optimisée et beaucoup plus efficace que celle des listes.

Pour des raisons de lisibilité, on suppose que NUMPY a été importé par :

```
from numpy import *
```

En important NUMPY avec l'alias `np`, tous les noms de fonctions devairent être préfixés par `np`.

Modes de construction de tableaux

On utilise surtout, des tableaux de dimension 1 (vecteurs) ou 2 (matrices), mais les tableaux peuvent être multidimensionnels. Leur forme ou dimension est un t-uplet d'entiers, appelée `shape` par NUMPY.

`array(L)`	convertit une liste PYTHON en un tableau
`empty((n,m))`	crée un tableau de n lignes, m colonnes sans initialiser les valeurs
`zeros((n,m))`	crée un tableau de n lignes, m colonnes avec tous ses coefficients nuls

ones((n,m))	crée un tableau de n lignes, m colonnes avec tous les coefficients égaux à 1
identity(n)	crée la matrice identité d'ordre n
arange(n,m,p)	analogue à range mais renvoie un tableau de type ndarray
linspace(x,y,n)	renvoie une subdivision régulière de $[x, y]$ en n éléments régulièrement espacés

Caractéristiques d'un tableau

Quelques fonctions permettent de trouver les caractéristiques d'un tableau.

Dimensions d'un tableau

shape(A)	indique le format du tableau A, sous la forme du t-uplet du nombre d'éléments dans chaque direction
alen(A)	donne la première dimension d'un tableau (la taille pour un vecteur, le nombre de lignes pour une matrice)
ndim(A)	donne la dimension du tableau, soit le nombre d'indices nécessaire au parcours du tableau (1 pour un vecteur, 2 pour une matrice)
size(A)	donne le nombre total d'éléments du tableau ($n \times p$ pour une matrice de type n lignes p colonnes

À noter également, la fonction reshape(A,(n,m)) permet de redimensionner un tableau A au format (n,m) pourvu que A possède $n \times m$ éléments.

Type de tableau

Les éléments d'un tableau NumPy sont tous du même type. On trouve notamment des tableaux de type : booléen, entier, flottant, complexe. On peut aussi former des tableaux de chaînes de caractères de longueur fixée.

Le type de tableau est choisi par la fonction `array` en fonction des coefficients lors de la conversion. Pour les autres procédés de construction, on peut imposer le type de tableau en ajoutant le paramètre optionnel dtype= float par exemple.

Lecture et écriture dans un tableau

Accès aux éléments et coupes

Les conventions pour l'accès aux éléments d'un tableau sont celles des listes (voir p 11). En particulier, pour un tableau A de dimension 2 à n lignes et m colonnes, les indices de ligne et de colonne varient de 0 à $n-1$ et de 0 à $m-1$ respectivement.

A[i , j]	accède à l'élément d'indice (i, j)
A[i, :] ou A[i]	retourne le vecteur $i^{\text{ième}}$ ligne
A[: , j]	retourne le vecteur $j^{\text{ième}}$ colonne
A[:i , :j]	retourne la sous-matrice formée des i premières lignes et des j premières colonnes
A[i1:i2,j1:j2]	retourne la sous-matrice constituée coefficients $a_{i,j}$ avec $i_1 \leqslant i < i_2$ et $j_1 \leqslant j < j2$

Modifications d'un tableau

Comme on peut accéder directement aux éléments et aux coupes, on peut également les remplacer.

A[i , j]=b	remplace l'élément d'indice (i, j) par b
A[i] =B	remplace la $i^{\text{ième}}$ ligne par le vecteur B
A[: , j]= B	remplace la $j^{\text{ième}}$ colonne par le vecteur B

Opérations sur les tableaux

Opérations algébriques

De façon générale, les opérations algébriques entre tableaux de type ndarray s'entendent coefficient par coefficient. Ainsi

A+B	additionne terme à terme des tableaux de même taille
A*B	multiplie terme à terme des tableaux de même taille
x*A	multiplie par x tous les coefficients de A
x+A	ajoute x à tous les coefficients de A

Attention : L'opérateur * multiplie terme à terme deux tableaux de même dimension, il ne s'agit pas du produit matriciel au sens mathématique du terme.

Statistiques élémentaires sur un tableau

Voici quelques opérations statistiques qu'on peut réaliser sur un tableau de nombres en dimension 1. Ces opérations s'appliquent également aux tableaux multidimensionnels. Il faut alors choisir un axe de calcul, en rajoutant l'option ,axis=$\cdots$. Par défaut, l'axe est 0 et on travaillera colonne par colonne.

sum(a)	retourne la somme des éléments de a
prod(a)	retourne le produit des éléments de a
max(a)	retourne le maximum des éléments de a
min(a)	retourne le minimum des éléments de a
mean(a)	retourne la moyenne des éléments de a
var(a)	retourne la variance des éléments de a

Fonctions universelles

Si a est un tableau unidimensionnel de nombres, et f est une fonction numérique, on aura souvent intérêt à appliquer la fonction f à tous les éléments de a. Les fonctions mathématiques de NUMPY s'appliquent directement à des vecteurs, on dit qu'elles sont **universelles**.

On retrouve ici les fonctions usuelles en version « universelle » :sin, cos, tan, arcsin, arccos, arctan, cosh, sinh, tanh, exp, log, log2, log10, etc.

Outils pour l'ingénierie numérique, la simulation

Calcul numérique matriciel avec linalg

Le module linalg de NUMPY est riche de nombreuses fonctions. En voici quelques unes. On suppose ici les modules importés avec :

```
from numpy import *
import numpy.linalg as la
```

Opérations basiques sur les matrices

dot(M,N)	réalise le produit matriciel au sens mathématique du terme
la.matrix_power(M,n)	réalise la puissance $n^{\text{ième}}$ d'une matrice carrée
a*M+b*N	réalise la combinaison linéaire des matrices M et N

Inverse, déterminant d'une matrice

la.inv(A)	retourne l'inverse d'une matrice carrée A
la.det(A)	calcule le déterminant d'une matrice carrée A

trace(A)	trace de la matrice carrée A
transpose(A)	transposée de A

Résolution d'un système d'équations linéaires

la.solve(A,B)	solution du système d'équations linéaires (S) $AX = B$

Réduction des matrices

la.eigvals(A)	retourne les valeurs propres de la matrice carrée A
la.eig(A)	retourne en plus les vecteurs propres (unitaires) associés

Analyse numérique avec Scipy

Résolution numérique d'équations avec optimize

Le module `optimize` de SciPy permet de calculer des valeurs approchées des zéros d'une fonction.

fsolve(f,a)	retourne une solution de l'équation $f(x) = 0$, proche de a.

Intégration numérique avec integrate

La fonction `quad` du module `integrate` de SciPy permet de calculer de façon approchée des intégrales.

quad(f,a,b)	retourne une valeur approchée de l'intégrale de f entre a et b (qui peuvent être infinis)

Équations différentielles avec integrate

On considère le problème de Cauchy suivant :

$$(C) \qquad \begin{cases} y'(t) &= f(y(t),t) \\ y(t_0) &= y_0 \end{cases}$$

où y est une fonction à valeurs réelles ou vectorielles.

odeint(f,y0,t)	retourne des valeurs approchées de la solution du problème de Cauchy (C).

Ici, t est un tableau de valeurs de la forme $t = [t_0, t_1, ..., t_n]$. La solution retournée correspond à la distribution des valeurs approchées de f aux différents instants $t_0, t_1, \ldots, t_n$.

Simulation numérique avec random

Le module random de NUMPY permet de réaliser des tirages aléatoires.

random()	choisit un réel aléatoire dans $[0, 1]$
uniform(x,y)	choisit un réel aléatoire dans $[x, y]$
randrange()	choisit un entier aléatoire dans range(n,m,p)
randint(n,m)	choisit un entier aléatoire dans $[\![n, m]\!]$
choice(x)	choisit un élément aléatoire dans une séquence

NUMPY propose également tout un jeu de distributions standards. Cette fois, si on veut plusieurs valeurs, on utilisera l'argument optionnel size $= \cdots$.

normal(m,s)	simule une loi normale de paramètre m, s
binomial(n,p)	simule une loi binomiale de paramètres n et p
geometric(p)	simule une loi geometrique de paramètre p
poisson(x)	simule une loi de poisson de paramètre x

Graphiques avec matplotlib

Le module matplotlib.pyplot permet de tracer des graphiques dans une fenêtre séparée. L'utilisation de base de cette fonction est plot (x,y) où x et y sont des tableaux d'abscisses et d'ordonnées, en fait, ce sont les coordonnées cartésiennes des points du graphe.

clf()	efface la fenêtre graphique
plot(x,y)	trace les points de coordonnées (x, y)
grid()	affiche la grille
axis([a,b,c,d]	fixe les axes de coordonnées de $[a, b] \times [c, d]$
xlabel(ch)	ch est l'étiquette des abscisses (chaîne de caractères)
savefig()	exporte et enregistre la figure au format donné dans l'extension précisé en argument (mafigure.pdf par exemple)

« La science est ce que nous comprenons suffisamment bien pour l'expliquer à un ordinateur. L'art, c'est tout ce que nous faisons d'autre. »

Donald Knuth, informaticien américain.

4. Bases de données et algèbre relationnelle

Algèbre relationnelle

On présente d'abord le vocabulaire de l'algèbre relationnelle avant de donner la description des commandes SQL à connaître ainsi que leur équivalent en algèbre relationnelle.

Vocabulaire d'algèbre relationnelle

On considère un ensemble fini $\mathcal{A}$, dont on appelle les éléments les **attributs**. On note D un ensemble et dom une application de $\mathcal{A}$ dans $\mathcal{P}(D)$.

Pour tout attribut $A \in \mathcal{A}$ on appelle **domaine** de A l'ensemble $dom(A)$.

On appelle **schéma relationnel** un n-uplet de la forme

$$S = (A_1, \ldots, A_n) \in \mathcal{A}^n$$

où les A_i sont distincts deux à deux.

On appelle **relation** ou **table** associée au schéma relationnel S un sous-ensemble de $dom(A_1) \times \cdots \times dom(A_n)$.

On note $R(S)$ une relation R si on souhaite préciser qu'elle est associée au schéma relationnel S.

Les opérations d'algèbre relationnelle

On peut distinger les opérations ensemblistes et les opérations relationnelles d'autre part.

Les opérations ensemblistes sont :

- l'**union** $R_1 \cup R_2$ de deux relations $R_1(S)$ et $R_2(S)$ est l'ensemble des éléments appartenant à R_1 ou à R_2,
- l'**intersection** $R_1 \cap R_2$ de deux relations $R_1(S)$ et $R_2(S)$ est l'ensemble des éléments appartenant à la fois à R_1 et à R_2,
- la **différence** $R_1 \setminus R_2$ de deux relations $R_1(S)$ et $R_2(S)$ est l'ensemble des éléments appartenant à R_1 mais pas à R_2.

Les opérations relationnelles sont :

- la **sélection** $\sigma_C(R_1)$ est l'ensemble des éléments de $R_1(S)$ satisfaisant la condition logique C,
- la **projection** $\pi_{A_1,\ldots,A_m}$ de la relation $R_1(S)$ est l'ensemble des éléments obtenus par projection ensembliste de $R1(S)$ sur $dom(A_1) \times \cdots \times dom(A_m)$,
- le **produit cartésien** $R_1 \times R_2$ de deux relations $R_1(S_1)$ et $R_2(S_1)$ est l'ensemble $\{(x_1, x_2), x_1 \in R_1, x_2 \in R_2\}$,
- la **jointure** $R_1[C]R_2$ de deux relations $R_1(S_1)$ et $R_2(S_1)$ est l'ensemble des éléments du produit $R_1 \times R_2$ satisfaisant la condition C,
- la **division** $R_1 \div R_2$ de deux relations $R_1(S_1)$ et $R_2(S_2)$ telles que $S_2 = (A_1, \ldots, A_m)$ et $S_1 = (A_1, \ldots, A_m, A_{m+1}, \ldots, A_n)$ est l'ensemble des éléments x images par la projection de R_1 sur $(A_{m+1}, \ldots, A_n)$ tels que le produit $\{x\} \times R_2$ soit inclus dans R_1.

Commandes SQL

Opérations SQL : opérations ensemblistes

UNION	permet d'effectuer l'**union** $R_1 \cup R_2$ des éléments de deux tables $R_1(S)$ et $R_2(S)$ de même schéma relationnel

INTERSECT	permet d'effectuer l'**intersection** $R_1 \cap R_2$ des éléments de deux tables $R_1(S)$ et $R_2(S)$ de même schéma relationnel
EXCEPT	permet d'effectuer la **différence** ensembliste $R_1 \setminus R_2$ des deux tables $R_1(S)$ et $R_2(S)$ de même schéma relationnel

On peut en fait effectuer en SQL ces opérations si les schémas relationnels des deux relations sont différents, à condition qu'elles aient le même nombre d'attributs et des domaines identiques. On dit alors que les schémas sont **identiques**.

Autres opérateurs à schéma relationnel fixé

Les instructions décrites ici sont les instructions de construction de table à partir d'une base de données et d'un schéma relationnel. On précise comment les faire correspondre à telle ou telle opération d'algèbre relationnelle.

SELECT	c'est l'instruction de début de construction de table. Suivie de * elle prendra tous les attributs possibles, suivie de $A_1, \ldots, A_m$ elle permet d'effectuer la **projection** $\pi_{A_1,\ldots,A_m}$ sur ces attributs
WHERE	suivie d'une condition logique C, elle permet d'effectuer la **sélection** σ_C ne conservant que les éléments satisfaisant C
AS	A as B permet d'effectuer le **renommage** $\rho_{A \leftarrow B}$ de l'attribut A en B

Pour créer des conditions en SQL, on peut utiliser <, >, =, ainsi que les connecteurs logiques AND, OR. Les opérateurs suivants font intervenir des relations de schémas différents.

Opérations complexes

JOIN	permet d'effectuer le **produit cartésien** $R_1 \times R_2$ de deux relations R_1 et R_2.

ON	située après un JOIN, et suivie d'une condition C, elle permet d'effectuer la **jointure** $R_1[C]R_2$ des deux tables considérées c'est à dire de ne garder dans le produit cartésien que les éléments satisfaisant C.
GROUP BY	suivie de $A_1, \ldots, A_m$ et si on a indiqué $A_1, \ldots, A_m, f(A_{m+1}, \ldots, A_n)$ comme attributs après le SELECT, cette instruction effectue l'**aggrégation** $\gamma(A_{m+1}, \ldots A_n) \rightarrow f(A_{m+1}, \ldots, A_n)$

La **division cartésienne** $R_1 \div R_2$ n'a pas de réalisation immédiate en SQL et nécessite une combinaison des commandes présentées ci-dessus pour être réalisée.

On signale également la commande DISTINCT qui, placée directement après SELECT, permet de ne garder qu'une seule occurence de chaque élément dans la table.

On présente enfin quelques unes des fonctions pouvant être utilisées dans les aggrégations.

Fonctions d'aggrégation

COUNT	compte les éléments. On peut l'utiliser suivie de DISTINCT pour éviter de compter plusieurs fois les mêmes éléments
MAX	donne le maximum des éléments
MIN	donne le minimum des éléments
SUM	donne la somme des éléments
AVG	donne la moyenne des éléments

Partie 2

Algorithmes de première année

LE SAVIEZ-VOUS ?

Informatique

Le mot *informatik* fut créé par l'ingénieur allemand Karl **Steinbuch** en 1957. C'est ce qu'on appelle un mot-valise c'est-à-dire obtenu comme contraction de deux mots ; dans ce cas, il s'agissait des mots *information* et *automatique*. À cette époque là, en effet, l'utilisation de l'informatique se diversifie et l'on commence à voir son rôle dans le traitement de l'information.

L'introduction dans notre langue du terme francisé *informatique* date de 1962. Elle est l'œuvre de Philippe **Dreyfus**, alors directeur du centre national de calcul électronique de la société Bull.

La terminaison en –ique est heureuse car sa similarité avec celle de noms de sciences proches comme *mathématiques* ou *physique* donne à cette discipline naissante ses lettres de noblesse.

1. Programmer avec les réels

Sommes et produits finis

Étant donné une suite $(\ell_k)_{k\in\mathbf{N}}$ de nombres réels, on peut calculer les sommes ou produits finis :

$$S(n,m) = \sum_{k=n}^{m} \ell_k \text{ et } P(n,m) = \prod_{k=n}^{m} \ell_k,$$

où n et m sont des entiers naturels tels que $n < m$.

Les algorithmes

```
Fonction Somme(L,n,m)
Donnees
L : liste de réels
S : réel
k,n,m : entiers
Debut
S <-- 0
Pour k variant de n à m
  Faire
  S <-- S + L[k]
  Fin Faire
Retourner(S)
Fin
```

```
Fonction Produit(L,n,m)
Donnees
L : liste de réels
P : réel
k,n,m : entiers
Debut
```

```
P <-- 1
Pour k variant de n à m
  Faire
  P <-- P * L[k]
  Fin Faire
Retourner(P)
Fin
```

Analyse de ces algorithmes :
Ces deux algorithmes étant semblables, nous ne traitons que de l'analyse de la fonction `Somme`.

■ **Spécification :** La fonction `Somme` prend en entrée une liste de nombres réels ainsi que deux indices `n` et `m`. En sortie, elle retourne la valeur de la somme : $L[n] + L[n+1] + \cdots + L[m]$.

■ **Correction :** Notons pour tout entier naturel k, $\ell_k = L[k]$ et S_k la valeur de la variable informatique `S` à l'entrée du passage en boucle inconditionnelle d'indice k. Vérifions qu'un invariant de boucle est la propriété :

$$(\mathcal{P}_k) \qquad S_k = \sum_{i=n}^{k-1} \ell_i$$

• **Initialisation :** Lorsque $k = n$, S_n est initialisé à 0, ce qui correspond bien à la somme sans terme : $\sum_{i=n}^{n-1} \ell_i$.

• **Hérédité :** Soit $k \geqslant n$ tel que $\mathcal{P}_k$. Dans la boucle d'indice k, l'instruction informatique :

```
S <-- S + L[k]
```

se traduit par la relation mathématique :

$$S_{k+1} = S_k + \ell_k.$$

Compte tenu que $\mathcal{P}_k$ est vrai par hypothèse, il en résulte que :

$$S_{k+1} = \sum_{i=n}^{k-1} \ell_i \ + \ell_k = \sum_{i=n}^{k} \ell_i.$$

La propriété $\mathcal{P}_{k+1}$ est vérifiée.

• **Conclusion :** Lors du dernier passage en boucle, S_m est égal à $\sum_{i=n}^{m-1} \ell_i$. À l'issue de ce dernier passage, la valeur retournée est

$$S_{m+1} = \sum_{i=n}^{m} \ell_i.$$

■ **Complexité :** Pour chaque passage en boucle, on peut distinguer 3 opérations élémentaires (OPEL) : une somme et deux affectations (celle de `S` et celle de `k`). En dehors de la boucle, il y a 2 OPEL : l'initialisation de `S` et le retour du résultat. Ainsi, la complexité de cet algorithme est de $3(m-n)+5$.

Les programmes en Python

```
def Somme(L,n,m):
    S=0
    for k in range(n,m+1):
        S=S+L[k]
    return(S)

def Produit(L,n,m):
    P=1
    for k in range(n,m+1):
        P=P*L[k]
    return(P)
```

Boîte à outils

`sum(L)`	cette fonction prédéfinie de PYTHON calcule la somme des éléments d'une liste
`prod(L)`	calcule leur produit, mais il s'agit d'une fonction de NUMPY.

Puissances d'un réel

Les puissances successives d'un réel a sont un cas particulier de calcul de produit fini. On peut donc utiliser un algorithme proche de la fonction `produit(L,n,m)`. Toutefois, il est possible d'obtenir ce même résultat avec une complexité bien meilleure.

Pour comprendre cet algorithme dit d'**exponentiation rapide** considérons les deux expressions de a^4 par exemple :

$$\begin{aligned} a^4 &= ((a \times a) \times a) \times a \\ a^4 &= (a \times a)^2 \end{aligned}$$

La première expression de a^4 nécessite trois multiplications, tandis que la deuxième n'en coûte que deux !

L'algorithme naïf

```
Fonction Puissance_naif(a,n)
Donnees
a,P : reels
k,n : entiers
Debut
P <-- 1
Pour k variant de 1 à n
    Faire
    P <-- P * a
    Fin Faire
Retourner(P)
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée un réel a et un entier naturel n et retourne la valeur de a^n.
- **Terminaison :** Cette boucle inconditionnelle se termine après n itérations.

■ **Correction :** On note pour tout entier naturel $k \geqslant 0$, P_k la valeur de la variable P après k passages en boucle. On vérifie qu'un invariant de boucle est ici :

$$(\mathcal{P}_k) \quad P_k = a^k$$

■ **Complexité :** Pour cet algorithme naïf, il y a 3 OPEL par passage dans la boucle. La complexité de l'algorithme est linéaire puisque le nombre d'OPEL s'évalue à $3n + 2$.

Le programme en Python

```
def Puissance_naif(a,n):
    P=1
    for k in range(n):
        P=P*a
    return(P)
```

L'algorithme rapide

```
Fonction Puissance_rapide(a,n)
Donnees
a, b, P : reels
k,n,m : entiers
Debut
P <-- 1
b <-- a
m <-- n
Tant que m>0
    Faire
    Si m%2=1
        Alors P <-- P * b
    b <-- b*b
    m <-- m//2
    Fin Faire
Retourner(P)
Fin
```

Notation : dans cet algorithme, nous avons noté `m%2` et `m//2` le reste et le quotient de la division euclidienne de m par 2.

Analyse de cet algorithme :

■ **Spécification :** C'est évidemment la même que celle de la précédente fonction `Puissance_naif`.

■ **Terminaison :** Notons pour tout entier $k \geqslant 0$, P_k,b_k, m_k les valeurs des variables informatiques `P`, `b`, `m` après k passages en boucles. On observe que pour tout entier k, m_{k+1} est le quotient dans la division euclidienne de m_k par 2. Ainsi :

$$m_{k+1} \leqslant \tfrac{1}{2} m_k$$

La suite d'entiers naturels $(m_k)_k$ est donc strictement décroissante. Elle est donc nécessairement finie.

■ **Correction :** Montrons qu'un invariant de boucle est ici la propriété

$$(\mathcal{P}_k) \qquad P_k \times b_k^{m_k} = a^n$$

• **Initialisation :** Lorsque $k = 0$, $P_0 = 1$, $b_0 = a$ et $m_0 = n$, de sorte que $P_0 \times b_0^{m_0} = 1 \times a^n = a^n$.

• **Hérédité :** Soit $k \geqslant 0$ tel que $\mathcal{P}_k$. À l'entrée du $k+1^{\text{ième}}$ passage en boucle, on distingue deux cas :

▸ Si m_k est pair alors $b_{k+1} = b_k^2$, $m_{k+1} = \frac{1}{2} m_k$, tandis que `P` est inchangé, $P_{k+1} = P_k$. Il en résulte que :

$$P_{k+1} \times b_{k+1}^{m_{k+1}} = P_k \times \left(b_k^2\right)^{\frac{1}{2} m_k} = P_k \times b_k^{m_k} = a^n.$$

▸ Si m_k est impair $P_{k+1} = P_k \times b_k$, $b_{k+1} = b_k^2$ et $m_{k+1} = m_k//2$, soit $m_k = 2m_{k+1} + 1$. En ce cas,

$$P_{k+1} \times b_{k+1}^{m_{k+1}} = P_k \times b_k \times \left(b_k^2\right)^{m_{k+1}} = P_k b_k^{2m_{k+1}+1} = P_k b_k^{m_k} = a^n.$$

Dans tous les cas, $\mathcal{P}_{k+1}$ est bien vérifiée.

• **Conclusion :** En sortie de boule, après N passages, on a $m_N = 0$. $\mathcal{P}_N$ donne alors

$$a^n = P_N \times b_N^0 = P_N$$

La valeur retournée est bien égale à a^n.

■ **Complexité :** Chaque passage en boucle nécessite ici 9 OPEL contre 3 dans l'algorithme naïf. Mais le nombre d'itérations est bien plus faible. En effet, comme nous l'avons observé, pour tout entier $k \geqslant 0$, $m_{k+1} \leqslant \frac{1}{2} m_k$, de sorte que

$$m_N \leqslant \frac{n}{2^N}$$

Par conséquent, m_N est nul dès que $2^N > n$, c'est-à-dire lorsque $N > \dfrac{\ln(n)}{\ln(2)} = \log_2(n)$.
Finalement, la complexité de cet algorithme rapide est logarithmique puisque le nombre d'OPEL s'évalue à $9N + 4$, où $N = \log_2(n)$.

Le programme en Python

```
def Puissance_rapide(a,n):
    P=1
    b=a
    m=n
    while m>0:
        if m%2==1:
            P=P*b
        b=b*b
        m=m//2
    return(P)
```

Boîte à outils

pow(x,y)	retourne la valeur de x à la puissance y, avec $x > 0$ et y des réels.
x**y	idem en version plus pratique

Évaluation polynomiale

Soit P un polynôme de degré n et $x \in \mathbf{R}$. On souhaite évaluer P en x, c'est-à-dire calculer la valeur de $P(x)$.

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Pour ce faire, il existe deux algorithmes « concurrents ».

L'algorithme naturel

Le premier algorithme, naturel, consiste à calculer les puissances successives de x et d'en former la combinaison linéaire. Autrement dit la suite (S_k) définie par :

$$\begin{cases} \bullet \quad S_0 = 0 \\ \bullet \quad S_{k+1} = S_k + a_k * x^k \end{cases}$$

vérifie $S_{n+1} = P(x)$.

```
Fonction Evaluer_polynome(A,x)
Donnees :
A : liste de réels (coefficients du polynôme)
x, S, puiss : réels
n, k : entiers,
Debut
n <-- Longueur(A)-1 # A = [a_0, a_1, ..., a_n]
puiss <-- 1
S <-- 0
Pour k variant de 0 à n
    Faire
    S <-- S + A[k] * puiss
    puiss <-- x * puiss
    Fin Faire
Retourner (S)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée la liste `A` des coefficients d'un polynôme P rangée suivant les puissances croissantes et un réel x. Elle renvoie la valeur de $P(x)$.

■ **Terminaison :** Il s'agit d'une boucle inconditionnelle. Elle termine après $n+1$ itérations.

■ **Correction :** Notons S_k et $puiss_k$ les valeurs des variables `S` et `puiss` à l'issue du $k^{\text{ième}}$ passage en boucle. On vérifie qu'un invariant de boucle est ici :

$$(\mathcal{P}_k) \qquad S_k = \sum_{i=0}^{k-1} A[i] \times x^i \text{ et } puiss_k = x^k$$

• **Initialisation :** Avant le premier passage en boucle, S_0 est initialisé à 0, ce qui correspond effectivement à la somme sans terme : $\sum_{i=0}^{-1} A[i] \times x^i$, tandis que $puiss_0$ est initialisé à 1.

• **Hérédité :** Soit $k \geqslant 0$ tel que $(\mathcal{P}_k)$. Lors du $k+1^{\text{ième}}$ passage en boucle (qui correspond à l'indice k), `S` puis `puis` sont actualisés :

$$\begin{aligned} S_{k+1} &= S_k + A[k] \times puiss_k = \sum_{i=0}^{k-1} A[i]x^i + A[k] \times x^k \\ &= \sum_{i=0}^{k} A[i]x^i \\ puiss_{k+1} &= x \times puiss_k = x^{k+1} \end{aligned}$$

Ainsi $(\mathcal{P}_{k+1})$ est vérifié.

• **Conclusion :** En sortie de boucle, à l'issue du $n+1^{\text{ième}}$ et dernier passage, on a :

$$S_{n+1} = \sum_{i=0}^{n} A[i]x^i = P(x).$$

■ **Complexité :** Le nombre d'OPEL de ce premier algorithme est de $6n+11$.

Le programme en Python

```
def Evaluer_polynome(A,x):
    n= len(A)
    puiss=1
    S=0
    for k in range(n):
        S=S+A[k]*puiss
        puiss=x*puiss
    return(S)
```

L'algorithme de Hörner

Le deuxième algorithme (dit **schéma de Hörner**) est basé sur l'idée que :

$$P(x) = a_0 + x(a_1 + x(a_2 + \cdots x(a_{n-1} + xa_n)\cdots))$$

En d'autres termes, la suite (u_k) définie par :

$$\begin{cases} \bullet \quad u_0 = a_n \\ \bullet \quad u_k = a_{n-k} + xu_{k-1} \end{cases}$$

vérifie $u_n = P(x)$.

```
Fonction Horner_polynome(A,x)
Donnees :
A : liste de réels (coefficients du polynôme)
x, S, puiss : réels
n, k : entiers,
Debut
n <-- Longueur(A)-1 # A = [a_0, a_1, ..., a_n]
u <-- A[n]
Pour i variant de n-1 à 0
    Faire
    u <-- A[i] + x * u
    Fin Faire
Retourner (u)
Fin
```

Analyse de cet algorithme :

■ **Terminaison :** La boucle inconditionnelle s'arrête après n itérations.

■ **Correction :** Notons u_k la valeur de la variable informatique `u` après k passages en boucle. Un invariant de boucle est :

$$(\mathcal{P}_k) \qquad u_k = a_{n-k} + a_{n-k+1}x^1 + \cdots + a_n x^k.$$

• **Initialisation :** Avant le premier passage en boucle, on a $u_0 = a_n x^0$.

• **Hérédité :** Soit k tel que $\mathcal{P}_k$ est vérifié. À l'issue du $k+1^{\text{ième}}$ passage en boucle (qui correspond à l'indice $i = n-k-1$) on a :

$$\begin{aligned} u_{k+1} &= a_{n-k-1} + x \times u_k \\ &= a_{n-k-1} + x \times \left(a_{n-k} + a_{n-k+1}x^1 + \cdots + a_n x^k\right) \\ &= a_{n-k-1} + a_{n-k}x^1 + a_{n-k+1}x^2 + \cdots + a_n x^{k+1} \end{aligned}$$

Ainsi, $(\mathcal{P}_{k+1})$ est bien vérifié.

• **Conclusion :** En sortie de boucle, après n passages en boucle, on a bien $u_n = P(x)$.

■ **Complexité :** Le coût de ce deuxième algorithme, s'élève à $4n+4$. En effet, il y a ici 4 opérations dans la boucle.

On peut démontrer que l'algorithme de Hörner est optimal, c'est l'algorithme d'évaluation d'un polynôme mettant en jeu le moins d'opérations élémentaires.

Le programme en Python

```
def Horner_polynome(A,x):
    n=len(A)-1
    u=A[n]
    for i in range(n-1,-1,-1):
        u= A[i]+ x*u
    return(u)
```

2. Programmer avec les entiers

Division euclidienne

L'algorithme

```
Fonction Division_euclidienne (a,b) :
Donnees
a,b,q,r, des entiers naturels, b>0
Debut
q <-- 0
r <-- a
Tant que r >= b
    Faire
    q <-- q+1
    r <-- r-b
    Fin Faire
Retourner (q,r)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend comme données deux entiers naturels a et b, avec $b > 0$ et calcule le quotient et le reste de la division euclidienne de a par b, c'est-à-dire l'unique couple (q, r) d'entiers tel que :

- $a = b \times q + r$
- $0 \leqslant r < b$

■ **Terminaison de boucle :** Notons pour $k \in \mathbb{N}$, r_k et q_k les valeurs des variables q et r après k passages dans la boucle et $\mathcal{P}_k$ la propriété

$$(\mathcal{P}_k) \qquad r_k \geqslant 0 \text{ et } a = b \times q_k + r_k.$$

La suite (r_k) est une suite d'entiers naturels strictement décroissante, elle est donc nécessairement finie. Par conséquent, il existe k tel que $r_k < b$. On note n le plus petit entier avec cette propriété.

■ **Correction :** Une récurrence facile permet de montrer que pour tout $k \in [\![0, n]\!]$ $\mathcal{P}_k$ est vraie.

De plus, à l'issue de la boucle, on a *à la fois* la condition d'arrêt $r_n < b$ et l'invariant $\mathcal{P}_n$, ce qui revient exactement à dire que $a = b \times q_n + r_n$ et $0 \leqslant r_n < b$

■ **Complexité :** Le nombre d'opérations élémentaires est de l'ordre de $\lfloor \frac{a}{b} \rfloor$: $N = 3 + 3 \times \lfloor \frac{a}{b} \rfloor$.

Le programme en Python

```
def Division_euclidienne(a,b):
    q=0
    r=a
    while r>=b:
        q=q+1
        r=r-b
    return(q,r)
```

Boîte à outils

Lorsque a et b sont des entiers, le quotient et le reste de la division euclidienne de a par b s'obtiennent directement en **Python** :

`a%b`	reste de la division euclidienne de a par b.
`a//b`	quotient de la division euclidienne de a par b.
`divmode(a,b)`	couple quotient,reste de la division euclidienne de a par b.

Test de primalité simple

L'algorithme

```
Fonction Est_premier(n)
Donnees
prem : booléen
k,n : entiers
Debut
prem <-- Vrai
Pour k variant de 2 à n-1
    Faire
    Si k divise n
        Alors prem <-- Faux
    Fin Si
    Fin Faire
Retourner(prem)
Fin
```


Analyse de cet algorithme :

- **Spécification :** La fonction prend en entrée un entier naturel `n` supérieur à 2 et indique en retour `Vrai` s'il est premier, `Faux` sinon.
- **Correction :** Notons $prem_k$ la valeur de la variable `prem` à l'issue de la boucle d'indice k. Un invariant de boucle est « $prem_k$ équivaut à n n'est divisible par aucun entier compris entre 2 et k ».
- **Complexité :** La complexité de cet algorithme est de l'ordre de $3n - 4$.

On peut surtout noter le manque d'efficacité de cet algorithme, qui, pour conclure que $n = 1024$ n'est pas premier, devra effectuer 3068 OPEL, alors que dès la première itération, il apparaît que n étant pair, il n'est pas premier !

L'utilisation d'une boucle conditionnelle permet d'améliorer ceci :

```
Fonction Est_premier2(n)
Donnees
prem : booléen
k,n : entiers
```

```
Debut
k <-- 2
prem <-- Vrai
Tant que (Prem est Vrai et k<n)
    Faire
    Si k divise n
        Alors prem <-- Faux
    Fin Si
    k <-- k+1
    Fin Faire
Retourner(prem)
Fin
```

Le programme en Python

```
def Est_premier2(n):
    k=2
    prem=True
    while (prem and k<n):
        if n%k == 0:
            prem=False
        k=k+1
    return (prem)
```

On peut également utiliser une boucle conditionnelle en utilisant l'instruction `return` pour interrompre la boucle.

```
def Est_premier(n):
    for k in range(2,n):
        if n%k == 0 :
            return(False)
    return (True):
```

Décomposition primaire d'un entier

Soit $n \in \mathbf{N}$, $n \geqslant 2$. Il existe alors des nombres premiers $p_1, \ldots, p_N$, il existe des entiers naturels non nuls $\alpha_1, \ldots, \alpha_N$ tels que :

$$n = p_1^{\alpha_1} \times p_2^{\alpha_2} \times \cdots \times p_N^{\alpha_N}.$$

Cette écriture, unique à l'ordre des facteurs près, s'appelle la **décomposition primaire de** n.

L'algorithme

```
Fonction Decomposition_primaire(n)
Donnees
n, i, r, q : entiers
diviseurs : liste des diviseurs premiers
Debut
q <-- n
diviseurs <-- []
Tant que q > 1
    Faire # trouver le plus petit diviseur de q
    i <-- 2
    r <-- q % i
    Tant que r > 0 et i < q
        Faire
        i <-- i+1
        r <-- q % i
        Fin Faire
    diviseurs <-- diviseurs+[i]
    q <-- q // i
    Fin faire
Retourner(diviseurs)
```


Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée un entier naturel n et retourne la liste de ses diviseurs premiers.

■ **Terminaison de boucle :** Notons pour $k \in \mathbb{N}$, q_k les valeurs de la variable informatique q après k passages dans la boucle principale. Comme i est initialisé à 2 en début de boucle et est incrémenté à chaque passage dans la boucle secondaire, cette deuxième boucle se termine après $q_k - 2$ passages au maximum. D'autre part, la valeur de

q étant divisée par 2 au minimum à chaque passage en boucle principale, on a $q_{k+1} < q_k$. Cette suite strictement décroissante d'entiers positifs et donc nécessairement finie.

■ **Correction :** Notons $diviseurs_k$ la valeur de la variable diviseurs après k passages en boucle principale. Notons $\mathcal{P}_k$ la propriété $diviseur_k$ est la liste des k plus petits diviseurs premiers de n.

• **Initialisation :** Avant d'entrer dans la boucle principale, diviseurs est la liste vide. Autrement dit, $diviseurs_0$ est la liste des 0 plus petits diviseurs premiers de n.

• **Hérédité :** Soit $k \in \mathbb{N}$ tel que $\mathcal{P}_k$ est vraie. Au cours du $k+1^{\text{ième}}$ passage, on détermine le plus petit diviseur supérieur à 2 de q_k. Comme il est plus petit diviseur de q_k, il est nécessairement premier et il divise n. C'est donc le plus petit diviseur premier de n en dehors de la liste $diviseurs_k$, soit le $k+1^{\text{ième}}$ diviseur premier de n dans l'ordre croissant. Ainsi, en le rajoutant à la liste $diviseurs_k$ on obtient bien la liste des $k+1$ plus petits diviseurs premiers de n.

• **Conclusion :** En sortie de boucle, on a q=1. La liste diviseurs contient bien la liste de tous les diviseurs premiers de n rangés dans l'ordre croissant.

■ **Complexité :** La complexité de cet algorithme est un problème très difficile qui dépasse largement le cadre de cet ouvrage !

Le programme en Python

```
def Decomposition_primaire(n):
    q=n
    diviseurs=[]
    while q>1:
        i=2
        r=q%i
        while (r>0 and i<q):
            i=i+1
            r=q%i
        diviseurs=diviseurs+[i]
        q=q//i
    return(diviseurs)
```

Calcul du PGCD de deux entiers

Soit $(a, b) \in \mathbb{N}^2$. L'algorithme d'Euclide permet de déterminer le plus grand entier, diviseur commun à a et b, noté $PGCD(a, b)$. Il repose sur le fait suivant :

Lemme d'Euclide : Lorsqu'on effectue la division euclidienne de a par b, $a = bq + r$, avec $0 \leqslant r < b$, on a :

$$PGCD(a, b) = PGCD(b, r)$$

On répète ces divisions successives, tant que le reste obtenu est non nul.

L'algorithme

```
Fonction PGCD(a,b)
Donnees a,b, u, v, w : entiers
Debut
u <-- max(a,b)
v <-- min(a,b)
w <-- u%v # reste de la DE de u par v
Tant que w > 0
    Faire
    u <--- v
    v <--- w
    w <--- u%v # reste de la DE de u par v
    Fin Faire
Retourner(v)
Fin
```

Analyse de cet algorithme :

- **Spécification :** La fonction PGCD(a,b) prend en entrée deux entiers naturels a et b et retourne leur PGCD.
- **Terminaison :** Notons u_k, v_k et w_k les valeurs des variables u, v et w après k passages dans la boucle. On a par construction :

$$\begin{cases} u_k = v_{k-1} \\ v_k = w_{k-1} \\ w_k \text{ est le reste de la DE de } u_k \text{ par } v_k \end{cases}.$$

En particulier $w_k < w_{k-1}$. La suite (w_k) est une suite strictement décroissante d'entiers naturels. Elle est donc nécessairement finie.

■ **Correction :** Démontrons qu'un invariant de boucle est ici la propriété :

$$(\mathcal{P}_k) \quad PGCD(a,b) = PGCD(u_k, v_k).$$

- **Initialisation :** c'est vrai avant l'entrée dans la boucle.
- **Hérédité :** Soit $k \in \mathbf{N}$ tel que $\mathcal{P}_k$. Le lemme d'Euclide montre que :

$$\begin{aligned} PGCD(a,b) &= PGCD(u_k, v_k) = PGCD(v_k, w_k) \\ &= PGCD(u_{k+1}, v_{k+1}) \end{aligned}$$

- **Conclusion :** En sortie de boucle, on a $w_n = 0$ et par suite

$$PGCD(a,b) = PGCD(u_n, v_n) = PGCD(v_n, 0) = v_n.$$

■ **Complexité :** Remarquons tout d'abord que w_{k+2} étant le reste de la division euclidienne de u_{k+2} par v_{k+2}, on a nécessairement $u_{k+2} \geqslant v_{k+2} \geqslant w_{k+2}$. Il en résulte que $u_{k+2} \geqslant 2w_{k+2}$. Comme u_{k+2} n'est autre que w_k, il s'ensuit que $w_{k+2} < \frac{1}{2}w_k$. Ainsi $w_{2k} < \frac{\alpha}{2^k}$, où $\alpha = \max\{a,b\}$. Par conséquent, dès que $k \geqslant \log_2(N)$, $w_{2k} = 0$. Finalement, le nombre de divisions euclidiennes (d'itérations) dans l'algorithme d'Euclide est donc $N = \log_2(\alpha)$. Par suite, la complexité de l'algorithme d'Euclide s'évalue à $4N + 7$, où $N = \log_2(\alpha)$.

Le programme en Python

```
def PGCD(a,b):
    if a>b:
        u=a
        v=b
```

```
    else :
        u=b
        v=a
    w=u%v # reste de la DE de a par b
    while w>0 :
        u=v
        v=w
        w=u%v # reste de la DE de u par v
    return(v)
```


Boîte à outils

gcd(a,b)	fonction du module **math** qui retourne le PGCD de a et b

Conversion d'un entier naturel en base b

Soit $b \in \mathbb{N}$, $b \geqslant 2$. Tout entier naturel $n \in \mathbb{N}^\star$ peut s'écrire de manière unique sous la forme :

$$n = \sum_{k=0}^{p} a_k b^k = a_p\, b^p + a_{p-1}\, b^{p-1} + \cdots + a_1\, b^1 + a_0\, b^0,$$

où $a_0, a_1, \ldots, a_p \in [\![0, b-1]\!]$ et $a_p \neq 0$. Lorsque b est différent de 10, on notera la base en indice : $n = (a_p a_{p-1} \ldots a_1 a_0)_b$. Effectuons la division euclidienne de n par b, il vient :

$$n = b\left(a_p b^{p-1} + \cdots + a_1\right) + a_0.$$

Ainsi, le reste de la division euclidienne de n par b est a_0 et le quotient est le nombre représenté en base b par $(a_p \cdots a_1)_b$. En itérant ce procédé, on obtient :

Méthode des divisions euclidiennes successives

On effectue les divisions euclidiennes successives par b jusqu'à l'obtention d'un quotient nul. On note a_0, a_1, …, a_{p-1}, a_p les restes

obtenus successivement. Alors n admet la représentation en base b :

$$\boxed{n = (a_p \cdots a_1 a_0)_b}$$

L'algorithme

```
Fonction Conversion(n,b)
Debut
chiffres <-- "0123456789ABCDE"
L <-- ""   #chaîne de caractère vide
q <-- n
Tant que q ≠ 0
  Faire
  r <-- q%b   #reste de la DE de q par b
  q <-- q//b  #quotient de la DE de q par b
  L <-- chiffres[r] + L
  Fin faire
Retourner(L)
fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction Conversion(n,b) prend en entrée deux entiers naturels $n \geqslant 1$ et $b \in [\![2, 16]\!]$. Elle retourne en sortie l'écriture de l'entier n en base b sous la forme d'une chaîne de caractères.

■ **Terminaison :** Notons r_k, q_k et L_k les valeurs des variables r, q et L après k passages dans la boucle. Lors du $k+1^{\text{ième}}$ passage en boucle, l'affectation q <– q//b se traduit par l'égalité $q_{k+1} = q_k//b$. En particulier $q_{k+1} < q_k$. La suite (q_k) est une suite strictement décroissante d'entiers naturels. Elle est donc nécessairement finie.

■ **Correction :** Écrivons $n = a_p\, b^p + a_{p-1}\, b^{p-1} + \cdots + a_1\, b^1 + a_0\, b^0$. Notons pour $k \geqslant 0$, $\mathcal{P}_k$ la propriété :

$$(\mathcal{P}_k) \quad q_k = (a_p \cdots a_k)_b \quad \text{et} \quad L_k = \text{"}a_{k-1} \ldots a_1 a_0\text{"}.$$

• **Initialisation :** avant d'entrer en boucle ($k = 0$), $q_0 = n = (a_p \cdots a_0)_b$ et L est vide.

• **Hérédité :** Soit $k \geqslant 0$ tel que $\mathcal{P}_k$. À l'entrée du $k+1^{\text{ième}}$ passage en boucle, on a

$$\begin{aligned} q_k &= (a_p \cdots a_k)_b \\ L_k &= "a_{k-1} \ldots a_1 a_0" \end{aligned}$$

On effectue alors la division euclidienne de q_k par b : d'après $\mathcal{P}_k$, le reste est a_k et le quotient est $(a_p \cdots a_{k+1})_b$. Ainsi, à l'issue du $k+1^{\text{ième}}$ passage, on a

$$\begin{aligned} q_{k+1} &= (a_p \cdots a_{k+1})_b \\ L_{k+1} &= "a_k" + "a_{k-1} \ldots a_1 a_0" = "a_k \ldots a_1 a_0" \end{aligned}$$

• **Conclusion :** En sortie de boucle, on a $q_{p+1} = 0$ et par suite

$$L_{p+1} = "a_p \ldots a_1 a_0".$$

■ **Complexité :** Le nombre de passages en boucle est égal à $p+1$, c'est-à-dire au nombre de chiffres de l'écriture en base b de n. Si tel est le cas, cela signifie que l'entier n vérifie l'encadrement $(01_p0 \ldots 0_0)_b \leqslant n < (1_{p+1}0 \ldots 0_0)_b$, soit :

$$b^p \leqslant n < b^{p+1}.$$

On en déduit successivement que $p \leqslant \log_b(n) < p+1$, puis que $p = \lfloor \log_b(n) \rfloor = \left\lfloor \dfrac{\ln(n)}{\ln(b)} \right\rfloor$.
Finalement, le nombre d'OPEL dans cet algorithme de conversion s'évalue à $6p + 10$, où p est de l'ordre de $\ln(n)$.

Le programme en Python

```
def Conversion(n,b):
    assert b<17,"b doit être inférieur à 16"
    chiffres ="0123456789ABCDE"
```

```
    L=""   #chaîne de caractère vide
    q=n
    while q!=0:
        r=q%b   #reste de la DE de q par b
        q=q//b  #quotient de la DE de q par b
        L=chiffres[r]+L
    return(L)
```

Boîte à outils

bin(n)	cette fonction prédéfinie retourne l'écriture de l'entier n en binaire
oct(n)	cette fonction prédéfinie retourne l'écriture de l'entier n en octal
hex(n)	cette fonction prédéfinie retourne l'écriture de l'entier n en hexadecimal

Conversion de la base b vers la base 10

Inversement, si $n \in \mathbf{N}$ est donné par son écriture en base $b \geqslant 2$, $n = (a_p \dots a_1 a_0)_b$, on peut retrouver son écriture décimale en calculant la somme :

$$n = a_p\, b^p + a_{p-1}\, b^{p-1} + \cdots + a_1\, b^1 + a_0\, b^0.$$

L'algorithme

```
Fonction Ecriture_decimale(L,b)
Données
L : chaîne de caractères
chiffres : liste de caractères
b,B,n,p,k,val : entiers
```

```
Debut
chiffres <-- ["0","1","2","3","4","5","6","7","8",
                ↳ "9","A","B","C","D","E"]
p <-- Longueur(L)
B <--1
n <-- 0
Pour k variant de p-1 à 0
    Faire
    #rechercher L[k] dans les chiffres
    val=0
    Tant que L[k] ≠ chiffres[val]
        Faire
        val <-- val+1
        Fin faire
    n <-- n + val * B
    B <-- b * B
    Fin Si
    Fin faire
Retourner(n)
fin
```


Analyse de cet algorithme :

■ **Spécification :** La fonction `Ecriture_decimale` prend en entrée un entier naturel $b \in [\![2, 16]\!]$ et une chaîne L formée de chiffres pour la base b. Elle retourne en sortie l'entier naturel n qui a L comme écriture en base b.

■ **Terminaison :** Les deux boucles sont indexées par des entiers k et val qui ne peuvent prendre qu'un nombre fini de valeurs. Elles se terminent donc.

■ **Correction :** Notons pour $i \geqslant 0$, n_i la valeur de la variable informatique n après i passages dans la boucle principale. Un invariant de boucle est la propriété :

$$(\mathcal{P}_i) \quad n_i = a_{i-1}\, b^{i-1} + \cdots + a_0\, b^0.$$

En sortie de boucle, après p passages, on a bien $n_p = a_{p-1}\, b^{p-1} + \cdots + a_0\, b^0 = n$.

■ **Complexité :** Le nombre de passages en boucle principale est égal à p et le nombre d'itérations dans la boucle secondaire est au maximum égal à b. Ainsi, le nombre d'OPEL dans cet algorithme s'évalue à $4pb + 5p + 5$, où p est la longueur de la liste L.

Le programme en Python

```
def Ecriture_decimale(L,b):
    chiffres=["0","1","2","3","4","5","6","7","8",
                        ↳ "9","A","B","C","D","E"]
    p=len(L)
    B=1
    n=0
    for k in range(p-1,-1,-1):
        val=0
        while L[k]!=chiffres[val]:
            val=val+1
        n=n+val*B
        B=b*B
    return(n)
```

Boîte à outils

int(ch,b)	fonction native de PYTHON, retourne l'entier représenté en base b par la chaîne ch.

« Aujourd'hui, la programmation est devenue une course entre le développeur, qui s'efforce de produire de meilleures applications à l'épreuve des imbéciles et l'univers, qui s'efforce de produire de meilleurs imbéciles. Jusque-là, c'est l'univers qui gagne. »

Rich Cook, écrivain américain.

3. Recherche d'élément dans une liste

Présence d'un élément dans une liste

La manière la plus simple de rechercher une valeur consiste tout simplement à comparer l'un après l'autre les éléments de la liste avec la valeur recherchée jusqu'à ce qu'on la trouve.

L'algorithme

```
Fonction Appartient(a,L)
Donnees
L : liste de réels
a : valeur recherchée
k, n: entiers
trouve : booléen
Debut
n <-- Longueur(L)
trouve <-- faux
Pour k variant de 0 à n-1
    Faire
    Si L[k] = a
      Alors trouve <-- vrai
    Fin Si
    Fin Faire
Retourner (trouve)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction Appartient (a,L) prend en argument un réel a et une liste de réels L. Elle retourne vrai ou faux suivant que a appartient à L ou pas.

■ **Terminaison :** La terminaison ne pose ici aucun problème puisqu'il s'agit d'une boucle inconditionnelle.

■ **Correction :** Un invariant de boucle est ici « Après k passages dans la boucle trouve est vrai si l'élément a a été trouvé et faux sinon ».

■ **Complexité :** Pour chaque passage en boucle, il y a un test et éventuellement une affectation. Compte tenu de l'intialisation et du retour de résultat, le nombre d'OPEL est donc $3n + 3$, où n est la longueur de la liste.

On peut noter ici que l'efficacité n'est pas optimale puisque si l'élément a a été trouvé dès le premier élément de la liste, il faudra parcourir l'ensemble de la liste pour retourner le résultat !

L'utilisation d'une boucle conditionnelle permet de remédier à ce défaut :

```
Fonction Appartient2(a,L)
Donnees
L : liste de réels
a : valeur recherchée
k, n: entiers
trouve : booléen
Debut
n <-- Longueur (L)
k <-- 0
Tant que k < n et L[k] ≠ a
    Faire
    k <-- k+1
    Fin Faire
Si k < n
    Alors trouve <-- vrai
    Sinon trouve <-- faux
Fin Si
Retourner (trouve)
Fin
```

Le programme en Python

```
def Appartient2(a,L):
    trouve=False
    n=len(L)
    k=0
    while (k<n and L[k] != a):
        k=k+1
    if k<n:
      trouve=True
    return(trouve)
```

Boîte à outils

`s in L`	retourne le booléen égal à True lorsque l'élément s apparaît dans la liste L

Recherche dichotomique d'un élément

Étant donné une liste L de n réels, nous avons déjà obtenu un algorithme pour rechercher la présence d'un élément a dans cette liste. Cet algorithme séquentiel avait une complexité linéaire en n. Lorsque la liste est triée par ordre croissant, nous pouvons faire bien mieux !

L'algorithme

```
Fonction Appartient3(a,L)
Donnees
L : liste triee par ordre croissant
N,m,g,d : entiers
a : réel recherché
trouve : booléen
Debut
n <-- Longueur (L)
trouve <-- faux
Imin <--0
```

```
Imax <--n-1
Tant que Imin <= Imax et Non (trouve)
    Faire
    m <--(Imin+Imax)//2 # m est le plus gd entier
        ↳inférieur au milieu entre Imin et Imax
    Si L[m] = a
        Alors trouve <-- vrai
        Sinon Si L[m] < a Alors Imin <-- m+1
        Sinon Si L[m] > a Alors Imax <-- m-1
    Fin Si
    Fin Faire
Retourner (trouve)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction Appartient3(a,L) prend en argument un réel a et une liste de réels L. Elle retourne vrai ou faux suivant que a appartient à L ou pas.

■ **Terminaison :** Notons pour $k \in \mathbb{N}$, m_k, $Imin_k$ et $Imax_k$ les valeurs des variables m, $Imin$ et $Imax$ à l'issue du $k^{\text{ième}}$ passage en boucle. On a pour tout entier k que

$$Imin_k \leqslant m_k \leqslant Imax_k$$

De plus, la suite $(Imin_k)$ est croissante, la suite $(Imax_k)$ est décroissante. Comme

$$Imax_{k+1} - Imin_{k+1} \leqslant \tfrac{1}{2}\big[Imax_k - Imin_k\big],$$

il en résulte que ces deux suites d'entiers naturels sont donc adjacentes et par conséquent finies.

■ **Correction :** Un invariant de boucle est ici « Après k passages dans la boucle trouve est faux ou a appartient à la sous-liste des termes d'indices compris entre $Imin_k$ et $Imax_k$. ».

■ **Complexité :** Comme la longueur de la liste dans laquelle on effectue la recherche est divisée par 2 à chaque nouvelle étape, on a après k étapes

$$Imax_k - Imin_k \leqslant \frac{n}{2^k}.$$

Par suite, si $k \geqslant \log_2(n)$, on a $Imax_k - Imin_k \leqslant 1$. On effectuera donc au maximum une étape supplémentaire. Finalement, la complexité de cet algorithme est donc de l'ordre de $\ln(n)$, soit bien meilleure que les algorithmes de recherche séquentielle étudiés précédemment.

Le programme en Python

```
def Appartient3(a,L):
    trouve=False
    n=len(L)
    imin=0
    imax=n-1
    while (imin<=imax and not(trouve)):
        m=(imin+imax)//2
        if L[m]==a :
            trouve=True
        elif L[m]<a :
            imin=m+1
        elif L[m]>a :
            imax=m-1
    return(trouve)
```


Première occurrence d'un élément

Un élément, présent au sein d'une liste peut y figurer à plusieurs reprises. On peut alors s'intéresser au rang d'apparition d'un élément pour la première fois, ce que l'on appelle la première occurrence d'un élément.

L'algorithme

```
Fonction Premiere_occurrence(a,L)
Donnees
L : liste de réels
a : valeur recherchée
k, n : entiers
```

```
Debut
n <-- Longueur (L)
k <-- 0
Tant que k < n et L[k] ≠ a
    Faire
    k <-- k+1
    Fin Faire
Si k < n
    Alors Retourner (k)
    Sinon Retourner (Néant)
Fin Si
Fin
```

Analyse de cet algorithme :

- **Spécification :** En entrée, Premiere_occurrence(a,L) prend un réel a et une liste de réels L. Elle retourne le plus petit indice k tel que $L[k] = a$ si a appartient à L ou rien sinon.
- **Terminaison :** L'algorithme se termine après n passages en boucles au maximum.
- **Correction :** Un invariant de boucle est « Après k passages dans la boucle, l'élément a n'a pas encore été trouvé ».
- **Complexité :** Pour chaque passage en boucle, il y a deux tests, une affectation et une addition. Compte tenu de l'intialisation et du retour de résultat, le nombre d'OPEL est donc $4n + 4$, où n est la longueur de la liste.

Le programme en Python

```
def Premiere_occurrence(a,L):
    n=len(L)
    k=0
    while (k<n and L[k] != a):
        k=k+1
    if k<n:
        return (k)
    else:
        return(None)
```

Nombre d'occurrences d'un élément

Un élément a, présent au sein d'une liste L, peut y figurer à plusieurs reprises. On va déterminer le nombre d'apparitions d'un élément.

L'algorithme

```
Fonction Nombre_occurrence(a,L)
Donnees
L : liste de réels
a : valeur réelle recherchée
k, n, occur: entiers
Debut
n <-- Longueur(L)
occur <-- 0
Pour k variant de 0 à n-1
    Faire
    Si L[k]= a
      Alors occur <-- occur + 1
    Fin Si
    Fin Faire
Retourner (occur)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée une liste L de nombres réels, a un nombre réel et retourne le nombre d'occurrences de a dans L.

■ **Terminaison :** Cette boucle inconditionnelle se termine après n itérations.

■ **Correction :** Notons $(occur_k)$ la suite des valeurs de la variable informatique occur. Ainsi, pour tout entier $k \in \mathbb{N}$, $occur_k$ est la valeur de occur lors de l'entrée en boucle d'indice k. Un invariant de boucle est ici la propriété $(\mathcal{P}_k)$:

« $occur_k$ est le nombre d'occurrences de a dans $\{L[0], \ldots, L[k-1]\}$ ».

• **Initialisation :** Lorsque $k = 0$, $occur_0$ est initialisé à 0, ce qui correspond bien évidemment au nombre d'apparitions de a dans la liste vide.

• **Hérédité :** Soit $k \geqslant 0$ tel que $\mathcal{P}_k$. Dans la boucle d'indice k, on teste si $L[k]$ est égal à a. Si tel est le cas, alors l'affectation occur <– occur + 1 se traduit par $occur_{k+1} = occur_k + 1$. Ainsi, $occur_{k+1}$ est le nombre d'occurrences de a dans $\{L[0], \ldots, L[k]\}$.

• **Conclusion :** Lorsqu'on sort définitivement de la boucle, $occur_n$ est le nombre d'occurrences de a dans $\{L[0], \ldots, L[n-1]\}$.

■ **Complexité :** Pour chaque passage en boucle, il y a 4 OPEL. Compte tenu de l'intialisation et du retour de résultat, le nombre d'OPEL est donc $4n + 3$.

Le programme en Python

```
def Nombre_occurrence(a,L):
    occur= 0
    n=len(L)
    for k in range(n):
        if L[k]==a:
            occur=occur+1
    return(occur)
```

«L'homme est encore le plus extraordinaire des ordinateurs. »

John F. Kennedy, Ancien président des États-Unis.

4. Statistiques sur une liste de réels

Maximum d'une liste

On suppose que nous disposons d'une liste de n nombres réels. Pour déterminer le maximum de tous les éléments, on introduit une variable qui prend la valeur maximale des premiers éléments. Nous rédigeons cet algorithme sous la forme d'une fonction, nommée Maximum(L) :

L'algorithme

```
Fonction Maximum(L)
Donnees
L : une liste de réels
k,n : des entiers
max : un réel
Debut
maxi <-- L[0]
n <-- Longueur(L)
Pour k variant de 0 à n-1
    Faire
    Si L[k] > maxi
        Alors maxi <-- L[k]
    Fin Si
    Fin Faire
Retourner(maxi)
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste L de n nombres réels et retourne la valeur maximale de cette liste.
- **Terminaison :** La terminaison ne pose ici aucun problème car nous avons une boucle inconditionnelle. La boucle s'arrête après n

itérations.

■ **Correction :** Notons pour tout entier $k \in \mathbb{N}$, m_k la valeur de la variable informatique maxi à l'entrée du passage en boucle d'indice k (c'est donc le $k+1^{\text{ième}}$ passage) et notons

$$(\mathcal{P}_k) \quad \text{la propriété : } m_{k-1} = \max\{L[0], \ldots, L[k-1]\}$$

• **Initialisation :** maxi est initialisé à L[0]. Donc $m_0 = L[0]$, $\mathcal{P}_1$ est vérifié.

• **Hérédité :** Soit $k \geqslant 1$ tel que $\mathcal{P}_k$ est vraie :

$$m_{k-1} = \max\{L[0], \ldots, L[k-1]\}.$$

En ce cas, si $L[k]$ est strictement supérieur à maxi, on remplace la valeur de maxi par celle de L[k], sinon, on ne fait rien. Ainsi, à l'issue du $k+1^{\text{ième}}$ passage en boucle, on aura bien :

$$m_k = \max\{L[0], \ldots, L[k]\}.$$

• **Conclusion :** Lorsque l'algorithme se termine, la propriété $\mathcal{P}$ est vraie pour k=n-1, la valeur maxi coïncide avec le maximum $\max\{L[0], \ldots, L[n-1]\}$.

■ **Complexité :** Pour chaque passage en boucle, il y a 3 OPEL. Compte tenu de l'intialisation et du retour de résultat, le nombre d'OPEL est donc $3n+2$.

Le programme en Python

```
def Maximum (L):
    n=len(L)
    maxi=L[0]
    for k in range(n):
        if L[k]>maxi :
            maxi=L[k]
    return(maxi)
```

Boîte à outils

max(L)	retourne le maximum d'une liste de réels
min(L)	retourne le minimum d'une liste de réels

Moyenne d'une liste

Étant donné une liste de n nombres réels, on peut condenser l'information en remplaçant cette série de n nombres par sa valeur moyenne.

Soit $(x_i)_{i\in[\![0,n-1]\!]}$ une liste de réels. La valeur moyenne de la liste (x_i) est donnée par :

$$m = \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

L'algorithme

```
Fonction Valeur_moyenne(L)
Donnees
L : une liste de réels
k, n : des entiers
somme : un réel
Debut
n <-- longueur(L)
somme <-- 0
Pour k variant de 0 à n-1
    Faire
    somme <-- somme + L[k]
    Fin Faire
Retourner (somme/n)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée une liste L de n nombres réels et retourne la valeur moyenne de cette liste.

■ **Terminaison :** La terminaison ne pose ici aucun problème car nous sommes en présence d'une boucle inconditionnelle. La boucle s'arrête après n itérations.

■ **Correction :** Notons pour tout $k \in \mathbb{N}$, $somme_k$ la valeur de la variable somme à l'entrée de la boucle d'indice k (le $k+1^{\text{ième}}$ passage) et notons

$$\mathcal{P}_k \quad \text{la propriété : } somme_k = L[0] + \cdots + L[k-1]$$

• **Initialisation :** Avant le premier passage en boucle, somme est initialisé à 0, ce qui revient à dire que $somme_0 = 0$.

• **Hérédité :** Soit $k \geqslant 0$ et supposons que

$$somme_k = L[0] + \cdots + L[k-1].$$

En ce cas, lors du $k+1^{\text{ième}}$ passage, l'affectation

$$\text{somme} <- \text{somme} + \text{L[k]}$$

se traduit par

$$\begin{aligned} somme_{k+1} &= somme_k + L[k] \\ &= L[0] + \cdots + L[k-1] + L[k]. \end{aligned}$$

• **Conclusion :** Lorsque l'algorithme se termine, la propriété $\mathcal{P}$ est vraie pour k=n-1, c'est-à-dire la valeur somme est égale à la somme $L[0] + \cdots + L[n-1]$. En divisant ce nombre par n, on obtient bien la valeur moyenne de cette liste.

■ **Complexité :** Pour chaque passage en boucle, il y a 3 OPEL. Compte tenu de l'initialisation, du calcul et du retour de résultat, le nombre d'OPEL est donc $3n+4$.

Le programme en Python

```
def Moyenne(L):
    n=len(L)
    somme=0
    for k in range(n):
        somme = somme + L[k]
    return(somme/n)
```

Boîte à outils

`mean(L)`	fonction de NUMPY, retourne la valeur moyenne d'une liste

Variance d'une liste

On peut donc représenter une distribution de valeurs par sa moyenne en première approximation. Ce faisant, on commet bien entendu une erreur. La variance mesure précisément la dispersion des valeurs autour de la moyenne. C'est donc une mesure d'erreur.

Soit $(x_i)_{i\in[\![0,n-1]\!]}$ une liste de moyenne m. La variance de la liste (x_i) est donnée par deux expressions :

Formule de Huygens :

$$V = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - m)^2 = \frac{1}{n}\sum_{i=0}^{n-1} x_i^2 \ - m^2$$

Remarque : en pratique, on préfère utiliser la deuxième expression. C'est également ce point de vue que nous utiliserons.

L'algorithme

```
Fonction Variance(L)
Donnees
```

```
L : liste de réels
k, n : entiers
smoy, svar : réels
Debut
n <-- Longueur(L)
smoy <-- 0
svar <-- 0
Pour k variant de 0 à n-1
    Faire
    smoy <-- smoy + L[k]
    svar <-- svar + L[k]*L[k]
    Fin Faire
Retourner ( (svar/n) - (smoy/n)*(smoy/n) )
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée une liste L de n nombres réels et retourne la variance de cette liste.

■ **Terminaison :** La terminaison ne pose ici aucun problème car nous avons une boucle inconditionnelle. La boucle s'arrête après n itérations.

■ **Correction :** Notons pour tout $k \in \mathbf{N}$, $smoy_k$ (resp $svar_k$) la valeur de la variable smoy (resp svar) à l'entrée de la boucle d'indice k (soit également à l'issue du $k^{\text{ième}}$ passage en boucle). On définit $\mathcal{P}_k$ la propriété :

$$smoy_k = L[0] + \cdots + L[k] \quad \text{et} \quad svar_k = L[0]^2 + \cdots + L[k]^2$$

• **Initialisation :** Lors du passage dans la boucle d'indice 0, $somme_0$ vaut 0.

• **Hérédité :** Soit $k \geqslant 0$ et supposons que $\mathcal{P}_k$ est vraie. En ce cas, lors du $k+1^{\text{ième}}$ passage,

$$\begin{aligned} smoy_{k+1} &= smoy_k + L[k+1] \\ &= L[0] + \cdots + L[k] + L[k+1] \quad \text{et} \\ svar_{k+1} &= svar_k + L[k+1]^2 \\ &= L[0]^2 + \cdots + L[k]^2 + L[k+1]^2 \end{aligned}$$

• **Conclusion :** Lorsque l'algorithme se termine, la propriété $\mathcal{P}$ est vraie pour k=n, ie. la valeur smoy est égale à la somme $L[1]+\cdots+L[n]$ et la valeur de svar est égale à la somme $L[0]^2+\cdots+L[n]^2$. À l'aide de la **Formule de Huygens**, on en déduit la variance.

■ **Complexité :** Pour chaque passage en boucle, il y a 6 OPEL. Compte tenu de l'intialisation et du calcul et du retour de résultat, le nombre d'OPEL est donc $6n+10$.

Le programme en Python

```
def Variance(L):
    n=len(L)
    smoy=0
    svar=0
    for k in range(n):
        smoy = smoy + L[k]
        svar=svar+L[k]**2
    return(svar/n-(smoy/n)**2)
```

Boîte à outils

var(L)	fonction de NUMPY qui retourne la variance d'une liste
std(L)	celle-ci retourne l'écart-type (racine carrée de la variance)

Médiane d'une liste

Étant donné une liste de n nombres réels distincts, la médiane est la valeur qui partage en deux l'effectif de la liste.

Lorsque $(x_i)_{i\in[\![0,n-1]\!]}$ une liste strictement croissante de réels. La valeur médiane de la liste (x_i) est donnée par :

$$\begin{cases} - & x_p & \text{, si } n=2p+1 \text{ est un nombre impair} \\ - & \frac{1}{2}(x_{p-1}+x_p) & \text{, si } n=2p \text{ est un nombre pair.} \end{cases}$$

L'algorithme qui suit calcule la médiane d'une liste qui n'est pas nécessairement triée.

L'algorithme

```
Fonction Mediane(L)
LP <-- L
LG <-- [] #Liste vide
Tant que Longueur(LP)-Longueur(LG) > 0
    Faire #trouver l'indice du maximum de LP
    imax=0
    Pour k variant de 0 à Longueur(LP)-1
        Faire
        Si LP[k] > LP[imax]
            Alors imax <-- k
        Fin Si
        Fin Faire
    #Enlever le terme LP[imax] de LP
    LP <-- LP \ LP[imax]
    #Rajouter le terme LP[imax] à LG
    LG <-- LG + LP[imax]
    Fin Faire
Si Longueur(LP) = Longueur(LG)
    Alors Retourner (Maximum(LP)+LG[0])*0.5)
    Sinon Retourner (LG[0])
Fin Si
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste L de n nombres réels distincts et retourne la valeur médiane de cette liste.
- **Terminaison :** À chaque passage dans la boucle conditionnelle, on enlève un terme de la liste LP pour l'ajouter à la listeLG. Ainsi, la différence Longueur(LP)-Longueur(LG) diminue de 2. Les valeurs successives de cette expression forment donc une suite strictement décroissante d'entiers strictement positifs. Elle est donc nécessairement finie.

■ **Correction :** Notons pour tout $i \in \mathbb{N}$, LG_i et LP_i les valeurs des variables LP et LG à l'entrée du $i+1^{\text{ième}}$ passage) dans la boucle inconditionnelle. Notons $\mathcal{P}_i$ la propriété :

$$LG_i \text{ est la suite ordonnée des } i \text{ plus grandes valeurs de } L$$

• **Initialisation :** Avant le premier passage en boucle, LG est vide, elle est formée des 0 plus grandes valeurs de la suite L.

• **Hérédité :** Soit $i \geqslant 0$ et supposons que $\mathcal{P}_i$ est vérifié. En ce cas, lors du $i+1^{\text{ième}}$ passage, on repère la valeur maximale de LP_i, on l'enlève et on la rajoute à LG_i (par la gauche). Ainsi, LG_{i+1} est bien constituée des $i+1$ plus grandes valeurs de L et elle est rangée par ordre croissant.

• **Conclusion :** Lorsque l'algorithme se termine, la propriété $\mathcal{P}$ est vraie. En outre, $Longueur(LP_i) - Longueur(LG_i) \leqslant 0$. Compte tenu du fait que cette suite décroît de 2 en 2, il y a possibilités :

- Lorsque $Longueur(LP_i) - Longueur(LG_i) = -1$, c'est-à-dire lorsque $Longueur(LG_i) = Longueur(LP_i) + 1$. En ce cas, la médiane est la plus petite valeur de LG_i.
- Lorsque $Longueur(LP_i) - Longueur(LG_i) = 0$, c'est-à-dire lorsque $Longueur(LG_i) = Longueur(LP_i)$. En ce cas, la médiane est la moyenne entre la plus petite valeur de LG_i et la plus grande valeur de LP_i.

■ **Complexité :** La boucle inconditionnelle peut être indexée par Longueur(LP) qui diminue de n à $\frac{n}{2}$ à pas de 1. À chaque passage dans cette boucle, on réalise un parcours de la liste LP. Ainsi, la complexité de cet algorithme est de l'ordre de n^2.

Remarque : l'utilisation d'algorithme de tri ou de la récursivité permet d'améliorer grandement cet algorithme. Voir l'algorithme de recherche rapide de la médiane, page 141.

Le programme en Python

```
def Mediane(L):
    LP=L;LG=[]
```

```
    while len(LP)-len(LG)>0:
        imax=0
        for k in range(len(LP)):
            if LP[k]>LP[imax]:
                imax=k
        LG=[LP[imax]]+LG
        LP=LP[:imax]+LP[imax+1:]
    if len(LP)-len(LG)==0:
        return ((Maximum(LP)+LG[0])*0.5)
    elif len(LP)-len(LG) == -1:
        return (LG[0])
```

Boîte à outils

median(a)	fonction de NUMPY retourne la valeur médiane d'une liste de réels

« En mathématiques, on ne comprend pas les choses, on s'y habitue. »

John Von Neumann, mathématicien américain.

5. Recherche d'une sous-séquence

Recherche d'une séquence dans une liste

Nous allons écrire un algorithme qui détecte la présence d'une séquence $a_0, a_1, \ldots, a_{r-1}$ de r réels dans une liste (principale) de n réels $L_0, L_1, \ldots, L_{n-1}$.
Pour ce faire, l'algorithme parcourt séquentiellement la liste L et compte, à partir de la position courante dans la liste L, j, le nombre de termes consécutifs qui coïncident dans les deux listes. Ce nombre de succès consécutifs est stocké dans une variable dénommée k. Si k atteint la longueur totale de la séquence a, c'est que toute la séquence apparaît dans la liste L.

L'algorithme

```
Fonction Contenu(a,L)
Donnees
a, L : listes de réels
j, k, r, n : entiers
trouve : booléen
Debut
n <-- Longueur (L)
r <-- Longueur (a)
trouve <-- faux
Pour j variant de 0 à n - r
    Faire
    k <-- 0
    Tant que k < r et a[k] = L[j+k]
        Faire
        k <-- k + 1
        Fin Faire
    Si k = r
        Alors trouve <-- Vrai
```

```
    Fin Si
    Fin Faire
Retourner(trouve)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en argument deux listes de réels, a et L de longueurs respectives r et n. Elle retourne vrai ou faux suivant que la séquence a apparaît dans la liste L ou pas.

■ **Terminaison :** Pour j compris entre 0 et $n-r-1$, la boucle conditionnelle se termine après r itérations au maximum. La boucle inconditionnelle se termine quant à elle après $n-r$ itérations.

■ **Correction :** Pour j compris entre 0 et $n-r-1$, notons $succes_j$ la valeur de la variable informatique k à la fin du $j+1^{\text{ième}}$ passage dans la boucle inconditionnelle.

Un invariant de cette boucle est :
« $\mathcal{P}_j$ $succes_j$ est égal au nombre de coïncidences consécutives entre entre les termes de la liste L à partir du rang j et la séquence a ».
Lorsque $succes_j$ est égal à r, cela signifie que $(L_j, \ldots, L_{j+r-1})$ coïncide avec $(a_0, a_1, \ldots, a_{r-1})$. La séquence a a été détectée.

■ **Complexité :** Pour chaque passage en boucle conditionnelle, il y a 4 OPEL (deux comparaisons, une addition et une affectation). Ainsi, chaque passage en boucle inconditionnelle représente au maximum $4r+3$ OPEL. Finalement, compte tenu des initialisations et du retour du résultat, le nombre maximal d'OPEL est $(n-r+1)(4r+3)+4$, où n et r sont les longueurs des deux listes.

Le programme en Python

```
def Contenu(a,L):
    n=len(L)
    r=len(a)
    trouve = False
    for j in range (n-r+1):
        k=0
```

```
        while(k<r and a[k]==L[j+k]):
                k=k+1
        if k==r:
            trouve = True
    return (trouve)
```

Remarque : On peut améliorer l'efficacité de cet algorithme en quittant la boucle inconditionnelle dès que la séquence a a été détectée.

```
def Contenu2(a,L):
    n=len(L)
    r=len(a)
    for j in range (n-r+1):
        k=0
        while(k<r and a[k]==L[j+k]):
                k=k+1
        if k==r:
            return(True)
    return(False)
```


Présence d'un mot dans une chaîne

L'algorithme précédent s'applique également aux chaînes de caractères. Il s'agit alors de détecter la présence d'un mot dans un texte.

L'algorithme

```
Fonction Recherche(mot,texte)
Donnees
texte, mot : chaînes de caractères,
j, k, r, n : entiers
trouve : booléen
Debut
n <-- Longueur (texte)
r <-- Longueur (mot)
trouve <-- faux
Pour j variant de 0 à n - r
```

```
    Faire
    k <-- 0
    Tant que k < r et mot[k] = texte[j+k]
        Faire
        k <-- k+1
        Fin Faire
    Si k = r
        Alors trouve <-- Vrai
    Fin Si
    Fin Faire
Retourner(trouve)
Fin
```

Analyse de cet algorithme :

- **Spécification :** La fonction Recherche (mot , texte) prend en argument deux chaînes de caractères, mot et texte. Elle retourne vrai ou faux suivant que la chaîne de caractères mot apparaît dans la chaîne texte ou pas.
- **Terminaison :** Pour j compris entre 0 et $n - r - 1$, la boucle conditionnelle se termine après r itérations au maximum. La boucle inconditionnelle se termine quant à elle après $n - r$ itérations.
- **Correction :** Pour j compris entre 0 et $n-r-1$, notons $succes_j$ la valeur de la variable informatique k à la fin du $j + 1^{\text{ième}}$ passage dans la boucle inconditionnelle.

Un invariant de cette boucle est :

« $\mathcal{P}_j$ $succes_j$ est égal au nombre de coïncidences consécutives entre entre les caractères de la chaîne texte à partir du rang j et la chaîne mot ».

Lorsque $succes_j$ est égal à r, cela signifie que mot apparaît dans la chaîne texte.

- **Complexité :** Pour chaque passage en boucle conditionnelle, il y a 4 OPEL (deux comparaisons, une addition et une affectation). Ainsi, chaque passage en boucle inconditionnelle représente au maximum $4r + 3$ OPEL. Finalement, compte tenu des initialisations et du

retour du résultat, le nombre maximal d'opérations élémentaires est $(n-r+1)(4r+3)+4$, où n et r sont les longueurs des deux listes.

Le programme en Python

```
def Rechercher(mot,texte):
    n=len(texte)
    r=len(mot)
    trouve = False
    for j in range (n-r+1):
        k=0
        while(k<r and mot[k]==texte[j+k]):
                k=k+1
        if k==r:
            trouve = True
    return(trouve)
```

Remarque : On peut améliorer l'efficacité de ce programme en quittant la boucle inconditionnelle dès que la chaîne mot est détectée.

```
def Rechercher(mot,texte):
    n=len(texte)
    r=len(mot)
    for j in range (n-r+1):
        k=0
        while(k<r and mot[k]==texte[j+k]):
                k=k+1
        if k==r:
            return(True)
    return(False)
```

Boîte à outils

mot in texte	retourne directement le booléen qui est vrai lorsque mot apparaît comme sous-chaîne de texte

6. Programmer avec les suites et les fonctions

Suite récurrente $u_{n+1} = f(u_n)$

Soit $f : I \to I$ une fonction définie et à valeurs dans un intervalle I de $\mathbf{R}$ et $a \in I$. Il existe une suite $(u_n)_{n\in\mathbf{N}}$ unique définie par les relations :

$$(S)\begin{cases} \bullet \quad u_0 = a \\ \bullet \quad \forall k \in \mathbf{N},\ u_{n+1} = f(u_n) \end{cases}$$

De nombreuses méthodes d'approximations numériques reviennent à déterminer le terme de rang n d'une telle suite.

L'algorithme

```
Fonction Suite_recurrente(f,a,n)
Donnees
f : fonction
a,u : reels
n,k : entiers
Debut
u <-- a
Pour k variant de 0 à n-1
    Faire
    u <-- f(u)
    Fin Faire
Retourner(u)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction `Suite_recurrente` prend en entrée une fonction itératrice `f`, une valeur initiale `a` et un entier `n` et

retourne en sortie la valeur du terme de rang n de la suite (u_n).

■ **Terminaison :** La terminaison de cette boucle inconditionnelle ne pose pas de problème.

■ **Correction :** Un invariant de boucle est ici « La valeur de la variable `u` après k passages en boucle est le terme de rang k de la suite définie par (S). »

■ **Complexité :** Évaluer le nombre d'opérations élémentaires d'un tel algorithme suppose de connaître la complexité de l'évaluation de f en u. En supposant que celle-ci est d'une OPEL, la complexité de l'algorithme s'évalue à $3n + 2$ OPEL.

Le programme en Python

```
def Suite_recurrente(f,a,n):
    u=a
    for k in range(n):
        u=f(u)
    return(u)
```

Pour l'étude de la limite d'une telle suite, voir la **méthode du point fixe**, p 97.

Suite récurrente $u_{n+2} = f(u_{n+1}, u_n)$

Il arrive qu'une suite soit définie par une relation de récurrence mettant en jeu trois générations de termes. En ce cas, il est nécessaire de connaître deux valeurs initiales pour la déterminer. Le schéma est le suivant :

$$(S_2)\begin{cases} \bullet & u_0 = a, u_1 = b \\ \bullet & \forall k \in \mathbf{N},\ u_{n+2} = f(u_{n+1}, u_n) \end{cases}$$

L'algorithme

```
Fonction Suite_recurrente2(f,a,b,n)
Donnees
f : fonction
a,b,u,v,w : réels
n,k : entiers
Debut
u <-- b
v <-- a
Pour k variant de 0 à n-1
    Faire
    w <-- f(u,v)
    v <-- u
    u <-- w
    Fin Faire
Retourner(v)
Fin
```

Analyse de cet algorithme :

- **Spécification :** La fonction `Suite_reccurente2` prend en entrée une fonction itératrice `f`, deux valeurs initiales `a`, `b` et un entier `n`. Elle retourne en sortie la valeur du terme de rang n de la suite à récurrence d'ordre 2, (u_n).
- **Terminaison :** La terminaison de cette boucle inconditionnelle ne pose pas de problème.
- **Correction :** Un invariant de boucle est ici « La valeur de la variable `u` après k passages en boucle est le terme de rang k de la suite définie par (S_2). »
- **Complexité :** Comme pour l'algorithme précédent, l'évaluation du nombre d'opérations élémentaires de cet algorithme suppose la connaissance de la complexité de l'évaluation de f. En supposant que celle-ci est d'une seule OPEL, la complexité de l'algorithme s'évalue alors à $5n + 3$ OPEL.

Le programme en Python

```
def Suite_recurrente2(f,a,b,n):
    v=a
    u=b
    for k in range(n):
        v,u= u,f(u,v)
    return(v)
```

Dans ce script, nous avons utilisé la possibilité offerte en PYTHON de réaliser une affectation simultanée de plusieurs variables.

Valeur maximale d'une fonction

Soit $f : [a, b] \to \mathbf{R}$ une fonction continue sur un segment. On sait que f possède une valeur maximale M sur $[a, b]$. Pour déterminer une valeur approchée de M, nous utilisons la méthode dite du **balayage**. Elle consiste à «discrétiser» l'intervalle $[a, b]$ en introduisant la subdivision régulière :

$$a_k = a + k\,\frac{b-a}{n},\ \text{pour } k \in [\![0, n]\!].$$

Une valeur approchée de M sera obtenue comme maximum de la liste de valeurs $\big(f(a_0), f(a_1), \ldots, f(a_n)\big)$.

L'algorithme

```
Fonction Valeur_maximale(f,a,b,n)
Donnees
f : fonctions
a,b,u,max,h: réels
k,n : entiers
Debut
u <-- a
v <-- f(a)
max <-- v
h <-- (b-a)/n
```

```
Pour k variant de 0 à n
    Faire
    u <-- u + h
    v <-- f(u)
    Si v > max
        Alors max <-- v
    Fin Si
    Fin Faire
Retourner(max)
Fin
```

Analyse de cet algorithme :

- **Spécification :** La fonction `Valeur_maximale` prend en entrée une fonction `f`, des réels `a` et `b` ainsi qu'un entier naturel `n`. Elle retourne une valeur approchée de la valeur maximale de f sur le segment $[a, b]$.
- **Terminaison :** Cette boucle inconditionnelle se termine après $n+1$ itérations.
- **Correction :** Notons max_k la valeur de la variable informatique `max` après k passages en boucle. Un invariant de boucle est la propriété :

$$(\mathcal{P}_k) \qquad max_k = \max\{f(a_0), f(a_1), \ldots, f(a_k)\}$$

- **Complexité :** en comptant pour une seule OPEL l'évaluation de f, la complexité de cet algorithme s'évalue à $7n + 14$.

Le programme en Python

```
def Valeur_maximale(f,a,b,n):
    u=a
    v=f(a)
    max=v
    h=(b-a)/n
    for k in range(n+1):
        u=u+h
        v=f(u)
```

```
        if v>max :
            max=v
    return(max)
```

Boîte à outils

Le module optimize de SCIPY propose la fonction minimize_scalar

```
opt=minimize_scalar(f,bounds=(a,b),method='bounded')
```

opt.fun	retourne la valeur minimale de f entre les bornes a et b
opt.x	retourne l'abscisse de ce minimum

«Son ordinateur était éteint parce que des octets avaient refroidi l'énergie de l'alphabet. »

Philippe Jaffieux, écrivain et poète français.

7. Résolution numérique des équations non linéaires

On considère une fonction $f : I \to \mathbf{R}$ une fonction suffisamment régulière. On suppose que f s'annule exactement une fois dans I, au point c. On cherche à résoudre numériquement l'équation

$$f(x) = 0 \qquad (E)$$

c'est-à-dire à obtenir une valeur approchée de c.

Recherche d'un zéro par dichotomie

On considère une fonction $f : I \to \mathbf{R}$ une fonction strictement monotone sur un intervalle I. On suppose qu'il existe un couple $(a, b) \in I^2$ tel que $a < b$ et $f(a) \times f(b) \leqslant 0$.

D'après le **théorème des valeurs intermédiaires** appliqué à une fonction injective (car strictement monotone), il existe un unique élément $c \in [a, b]$ tel que $f(c) = 0$.

Une démonstration de ce théorème repose sur la construction de suites adjacentes par dichotomie.
Plus précisément, on construit par récurrence des suites (u_n) et (v_n) vérifiant pour tout entier $n \in \mathbf{N}$:

(C_n)
- $u_0 \leqslant u_1 \leqslant \cdots \leqslant u_n \leqslant v_n \leqslant \cdots \leqslant v_1 \leqslant v_0$
- $v_n - u_n = \dfrac{b-a}{2^n}$
- $f(u_n) \times f(a) \geqslant 0$, $f(v_n) \times f(b) \geqslant 0$.

On montre alors que les suites (u_n) et (v_n) sont adjacentes et convergent

toutes deux vers la solution c de l'équation (E). De plus, pour tout entier $n \in \mathbf{N}$, on a

$$u_n \leqslant c \leqslant v_n$$

L'avantage de cette méthode de résolution est qu'elle est constructive, c'est-à-dire qu'elle fournit un algorithme qui permet d'approcher la solution de l'équation $f(x) = 0$ mais avec également un bon contrôle de l'erreur commise.

L'algorithme

```
Fonction Zero_dichotomie(f,a,b,eps)
Donnees
f : fonction;
a,b,u,v,w,eps : reels
Debut
u <-- a
v <-- b
Tant que v-u > 2*eps
    Faire
        w <-- (u+v)/2
        Si f(w)*f(a) >= 0
            Alors u <-- w
            Sinon v <-- w
        Fin Si
    Fin Faire
Retourner (w)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction `Zero_dichotomie` prend en argument une fonction f et deux réels a,b et une précision eps strictement positive. Elle retourne une valeur approchée à eps près de la solution de l'équation (E) comprise entre a et b.

■ **Terminaison :** Notons pour tout entier $n \in \mathbf{N}$, u_n, v_n les valeurs des variables informatiques u, v à l'issue du $n^{\text{ième}}$ passage dans la boucle conditionnelle. Comme nous allons le démontrer, un invariant de boucle est alors la propriété (C_n) définie précédemment. En

particulier, les suites (u_n) et (v_n) vérifient :

$$v_n - u_n \leqslant \frac{b-a}{2^n} \xrightarrow[n\to\infty]{} 0.$$

Par définition de la notion de limite, il existe un entier n_0 tel que $v_{n_0} - u_{n_0} \leqslant 2\,eps$. La boucle se termine après n_0 itérations.

■ **Correction :** Vérifions que (C_n) est un invariant de boucle.

• **Initialisation :** avant le premier passage en boucle, $u_0 = a$ et $v_0 = b$. On a bien $u_0 \leqslant v_0$, $v_0 - u_0 = b - a$ et $f(u_0) * f(a) \geqslant 0$, $f(b) \times f(v_0) \geqslant 0$.

• **Hérédité :** Soit $n \in \mathbb{N}$ tel que (C_n). Au cours du $n^{\text{ième}}$ passage en boucle, on définit $w_n = \frac{u_n+v_n}{2}$.

▸ Si $f(w_n)$ a le même signe que $f(a)$, on pose alors $u_{n+1} = w_n$ et on ne change pas la valeur de v, donc $v_{n+1} = v_n$.

▸ Sinon $f(w_n)$ a le même signe que $f(b)$, on pose alors $v_{n+1} = w_n$ et on ne change pas la valeur de u, donc $u_{n+1} = u_n$.

Dans les deux cas, on a donc :

$$u_n \leqslant u_{n+1} \leqslant v_{n+1} \leqslant v_n \text{ et } v_{n+1} - u_{n+1} = \frac{v_n-u_n}{2}$$
$$f(a) \times f(u_{n+1}) \geqslant 0 \text{ et } f(b) \times f(v_{n+1}) \geqslant 0.$$

Compte tenu que (C_n) est vérifié, on en déduit aisément (C_{n+1}).

• **Conclusion :** en sortie de boucle, on a C_n est vérifié et $v_{n_0} - u_{n_0} \leqslant 2\,eps$. Comme c est compris entre u_{n_0} et v_{n_0}, w_{n_0} est une valeur approchée de c à eps près.

■ **Complexité :** Pour estimer la complexité de cet algorithme, convenons de compter l'évaluation de f en une valeur comme une OPEL, ce qui n'est pas le cas en général. Avec cette convention, chaque passage en boucle revient à 11 OPEL. Pour que $v - u$ soit inférieur à $2eps$, il suffit que n soit supérieur à $N = \lfloor \log_2 \left(\frac{b-a}{eps}\right) \rfloor$. Finalement, la complexité s'évalue à $11N + 3$, où N est de l'ordre de $\ln(eps)$.

Le programme en Python

```
def Zero_dichotomie(f,a,b,eps):
    assert f(a)*f(b) <=0,"f doit changer de signe"
    u=a
    v=b
    while v-u > 2*eps:
        w=(u+v)/2
        if f(a)*f(w) >= 0:
            u=w
        else:
            v=w
    return(w)
```


Remarques :

• l'instruction assert f(a)*f(b) <=0,"f doit changer de signe" évite l'exécution de la fonction `Zero_dichotomie` lorsque la fonction $f(a) \times f(b) \leqslant 0$ n'est pas satisfaite.

• En revanche, l'hypothèse que f est strictement monotone sur I sert uniquement à garantir que l'équation (E) $f(x) = 0$ possède une unique solution sur I. Lorsque ce n'est pas le cas, la fonction `Zero_dichotomie (f,a,b,eps)` va retourner une valeur approchée d'une solution de cette équation comprise entre a et b.

Méthode du point fixe

Soit $g : [a, b] \to [a, b]$ une fonction lipschitzienne de constante $k \in]0, 1[$, c'est-à-dire telle que

$$\forall (x, y) \in [a, b]^2,\ |g(x) - g(y)| \leqslant k|x - y|$$

Alors la suite (u_n) définie par

$$\begin{cases} \bullet\ u_0 = a \\ \bullet\ \forall n \in \mathbf{N},\ u_{n+1} = g(u_n) \end{cases}$$

converge vers l'unique point fixe c de g sur $[a, b]$, c'est-à-dire l'unique solution de l'équation

$$g(x) = x. \qquad (F)$$

De plus

$$\boxed{\forall n \in \mathbf{N}, \quad |u_n - c| \leqslant k^n (b - a)}$$

Autrement dit, u_n est une valeur approchée du point fixe de g sur $[a, b]$. L'intérêt de cette méthode est d'être simple à programmer. En revanche, les conditions de convergence de la méthode ne sont pas aisées à vérifier : sur quel intervalle $[a, b]$ l'appliquer, quelle constante k peut-on choisir ?

L'algorithme

```
Fonction Point_fixe(g,a,N,eps)
Donnees
g : fonction
n,N : entiers
a,u,eps : reels
Debut
n <-- 0
u <-- a
Tant que n < N et |g(u)-u| > eps
    Faire
    n <-- n+1
    u <-- g(u)
    Fin Faire
Si |g(u)-u| ⩽ eps
    Alors Retourner (u)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction prend en argument une fonction g, un réel a, un nombre d'itérations maximal N et une précision eps. Elle retourne une valeur approchée d'un point fixe de g si l'algorithme converge, rien sinon.

■ **Terminaison :** La boucle conditionnelle se terminera après N itérations au maximum.

■ **Correction :** Un invariant de boucle est ici « après n itérations, la valeur de la variable informatique u est le terme de rang n de la suite récurrente (u_n)».
En sortie de boucle, si $|g(u_n) - u_n)| \leqslant eps$, alors u_n est une valeur approchée de c. Sinon, l'algorithme n'a pas convergé et la fonction ne retourne rien.

■ **Complexité :** Dans le pire des cas, si l'algorithme ne converge pas, on effectue $9N + 7$ OPEL. Lorsque au contraire, l'algorithme converge, pour que $|g(u_n) - u_n|$ soit inférieur à eps, il suffit que $k^n(b-a)$ soit inférieur à eps. Le plus petit entier N qui convienne alors est de l'ordre de $\ln(|eps|)$.

Le programme en Python

```
def Point_fixe(g,a,N,eps):
    n=0
    u=a
    while (n<N and abs(g(u)-u)>eps):
        n=n+1
        u=g(u)
    if abs(g(u)-u)<=eps:
        return(u)
```

Remarque : la méthode du point fixe peut être utilisée pour résoudre l'équation (E). En effet, si on pose $g(x) = x \pm f(x)$, on obtient l'équivalence :

$$(F) \qquad \forall x \in I,\ g(x) = x \iff f(x) = 0 \qquad (E)$$

Ainsi, on s'est ramené par changement de fonction à un problème de point fixe.

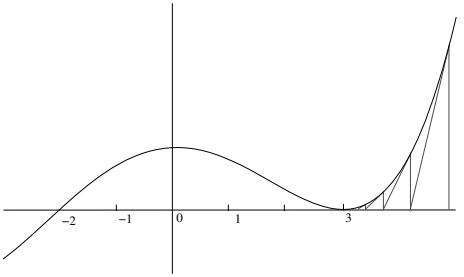
Méthode de Newton

Intuitivement, l'idée de base dans la méthode de Newton est de remplacer f par sa tangente. Plus précisément, partant d'un élément $u_0 = a$, on définit pour tout $n \in \mathbf{N}$, u_{n+1} comme étant l'abscisse du point du plan, intersection de la tangente au graphe de f au point u_n avec l'axe des abscisses (voir figure ci-dessous).

Comme l'équation de la tangente au point u_n s'écrit $y = f(u_n) + f'(u_n)(x - u_n)$, u_{n+1} est donc donné par :

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$$

Illustration



Admettons que cette suite récurrente soit bien définie (il faut pour cela que pour tout n , u_n appartienne à I et que $f'(u_n)$ soit non nul). En ce cas, si (u_n) converge vers $c \in I$, alors par unicité de la limite :

$$c = c - \frac{f(c)}{f'(c)}, \quad \text{soit } f(c) = 0$$

Ainsi, la limite c est solution de (E). Pour garantir que la suite soit bien définie et converge vers la solution c de (E), le résultat fondamental est le suivant :

Théorème de Newton —. Soit $c \in I$ et $f : I \to \mathbf{R}$ une fonction de classe $\mathcal{C}^2$. On suppose que f s'annule en c et que f' et f'' ne s'annulent pas dans I.

Pour tout $a \in I$ tel que $f(a)f''(a) \geqslant 0$, on considère la suite $(u_n)_{n\in\mathbf{N}}$ définie par $u_0 = a$ et la relation de récurrence :

$$\forall n \in \mathbf{N},\ u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}.$$

La suite $(u_n)_{n\in\mathbf{N}}$ est bien définie et converge vers c.

Soit $\mathcal{V}$ un voisinage contenant les points d'itérations. Posons $M = \sup_{\mathcal{V}} |f''(x)|$ et $m = \inf_{\mathcal{V}} |f'(x)|$. La formule de Taylor dans $\mathcal{V}$ donne l'estimation :

$$|u_n - c| \leqslant \left(\frac{M}{2m}\right)^{2^n-1} \times |a-c|^{2^n}.$$

L'algorithme

```
Fonction Zero_newton(f,df,a,N,eps)
Donnees
f,df : fonctions df est la dérivée de f
a,u,v, eps : reels
n,N : entiers
Debut
n <-- 0
u <-- a
v <-- u - f(u)/df(u)
Tant que (n < N et |v-u| > eps)
  Faire
  n <-- n+1
  u <-- v
  v <-- u - f(u)/df(u)
  Fin Faire
Si |v-u| <= eps
    Alors Retourner(v)
Fin
```

Analyse de cet algorithme :

- **Spécification :** La fonction `Zero_newton` prend en argument une fonction f ainsi que sa dérivée df un réel a un nombre maximal d'itérations N et une précision eps strictement positive. Elle retourne une valeur approchée à eps près de la solution de l'équation $f(x) = 0$ obtenue par la méthode de Newton à partir de la valeur initiale si toutefois cette méthode converge, rien sinon.
- **Terminaison :** La boucle conditionnelle se terminera après N itérations au maximum.
- **Correction :** Un invariant de boucle est ici « après n itérations, la valeur de la variable informatique u est le terme de rang n de la suite récurrente (u_n) de la méthode de Newton».
En sortie de boucle, si $|u_{n+1} - u_n| \leqslant eps$, alors u_n est une valeur approchée de c. Sinon, l'algorithme n'a pas convergé et la fonction ne retourne rien.
- **Complexité :** Dans le pire des cas, si l'algorithme ne converge pas, on effectue $12N + 11$ OPEL. Lorsque au contraire, l'algorithme converge, pour que $|u_{n+1} - u_n|$ soit inférieur à eps, il suffit que

$$\left(\frac{M}{2m}\right)^{2^n-1} \times |a - c|^{2^n}$$

soit inférieur à eps. Le plus petit entier N qui convienne alors est de l'ordre de $\ln\left(|\ln(|eps|)|\right)$.

Le programme en Python

```
def Zero_newton(f,df,a,N,eps):
    n=0
    u=a
    v=u-f(u)/df(u)
    while (n<N and abs(v-u)>eps):
        n=n+1
        u=v
        v=u-f(u)/df(u)
    if abs(v-u)<=eps:
        return(v)
```

Méthode de Lagrange

La méthode de Lagrange est une variante de la méthode de Newton. Soit $f \in \mathcal{C}^2([a,b], \mathbf{R})$ ayant une convexité déterminée (f'' de signe constant dans $[a,b]$). Pour calculer un zéro c de f par la méthode de Newton, la suite (u_n) peut être définie de la manière suivante :

$$\begin{cases} \bullet & u_0 \text{ voisin de } c \\ \bullet & \forall n \in \mathbf{N}, \quad f'(u_n)\,(u_{n+1} - u_n) = -f(u_n) \end{cases}$$

Dans certaines situations, la dérivée de f peut être très compliquée voire impossible à calculer.

En remplaçant la dérivée par un accroissement fini entre u_n et un point b, nous obtenons la **méthode de Lagrange** :

$$\forall n \in \mathbf{N} \quad \frac{f(u_n) - f(b)}{u_n - b}\,(u_{n+1} - u_n) = -f(u_n)$$

$$\text{soit} \quad u_{n+1} = \frac{u_n f(b) - b f(u_n)}{f(b) - f(u_n)}$$

D'où le schéma suivant :

$$\begin{cases} \bullet & u_0 = a \text{ voisin de } c \\ \bullet & \forall n \in \mathbf{N}, \quad u_{n+1} = \dfrac{u_n f(b) - b f(u_n)}{f(b) - f(u_n)} \end{cases}$$

Lorsque la valeur initiale a est bien choisie, la suite (u_n) ainsi définie converge vers c.

L'avantage de cette méthode est qu'elle ne nécessite pas le calcul de la dérivée f. L'inconvénient est que nous perdons la convergence quadratique.

L'algorithme

```
Fonction Zero_lagrange(f,a,b,N,eps)
Donnees
f : fonction
u,v, eps : réels
```

```
n,N : entiers
Debut
n <-- 0
u <-- a
v <-- (u*f(b)-b*f(u))/(f(b)-f(u))
Tant que (n<N et |v-u|> eps)
    Faire
    n <-- n+1
    u <-- v
    v <- (u*f(b)-b*f(u))/(f(b)-f(u))
    Fin Faire
Si |v-u| ⩽ eps
    Alors Retourner(v)
Fin
```

Analyse de cet algorithme :

- **Spécification :** La fonction `Zero_lagrange` prend en argument une fonction f, deux réels a et b, un nombre maximal d'itérations N et une précision eps strictement positive. Elle retourne une valeur approchée à eps près de la solution de l'équation $f(x) = 0$ obtenue par la méthode de Lagrange à partir de la valeur initiale a si toutefois cette méthode converge, rien sinon.

- **Terminaison :** La boucle conditionnelle se terminera après N itérations au maximum.

- **Correction :** Un invariant de boucle est ici « après n itérations, la valeur de la variable informatique u est le terme de rang n de la suite récurrente (u_n) de la méthode de Lagrange ».
  En sortie de boucle, si $|u_{n+1} - u_n| \leqslant eps$, alors u_n est une valeur approchée de c. Sinon, l'algorithme n'a pas convergé et la fonction ne retourne rien.

- **Complexité :** dans le pire des cas, lorsque l'algorithme ne converge pas, on effectue $12N + 11$ OPEL.

Le programme en Python

```
def Zero_lagrange(f,a,b,N,eps):
    n=0
    u=a
    v=(u*f(b)-b*f(u))/(f(b)-f(u))
    while (n<N and abs(v-u)>eps):
        n=n+1
        u = v
        v = (u*f(b)-b*f(u))/(f(b)-f(u))
    if abs(v-u)<=eps:
        return(v)
```

Boîte à outils

Le module **optimize** de SCIPY propose la fonction fsolve qui dispose de multiples options.

fsolve(f,x0)	retourne la solution obtenue en initiant la recherche au voisinage de x0 ou un message d'erreur le cas échéant

« La révolution informatique fait gagner un temps fou aux hommes, mais ils le passent avec leur ordinateur ! »

Isaac Asimov, écrivain américain de science-fiction.

8. Intégration numérique

Le théorème fondamental du calcul intégral affirme que toute fonction $f : I \to \mathsf{R}$, continue sur un intervalle possède une primitive F sur I. On peut alors calculer l'intégrale de f sur un segment inclus dans I :

$$\int_a^b f(t)\,dt = \Big[F(t)\Big]_a^b = F(b) - F(a)$$

Cependant l'existence de primitives sur f est un résultat théorique qui ne donne pas de formule permettant d'obtenir F à partir de f.

L'interprétation « géométrique » de l'intégrale comme étant l'aire de la région plane comprise entre le graphe de f et l'axe des abscisses permet d'obtenir des valeurs approchées de $\int_a^b f(t)\,dt$.

Méthode des rectangles

Le principe est de subdiviser le segment $[a, b]$ en n intervalles délimités par les points :

$$a_k = a + k\,\frac{b-a}{n}, \text{ pour } k \in [\![0, n]\!].$$

La longueur $h = \frac{b-a}{n}$ est appelée **le pas de la subdivision**.

Sur l'intervalle $[a_k, a_{k+1}]$, on remplace la fonction f par la fonction constante égale à $f(a_k)$ et l'intégrale $\int_a^b f(t)\,dt$ est approchée par la somme, notée S_n, des aires des n rectangles, comme on le visualise sur la figure ci-après.

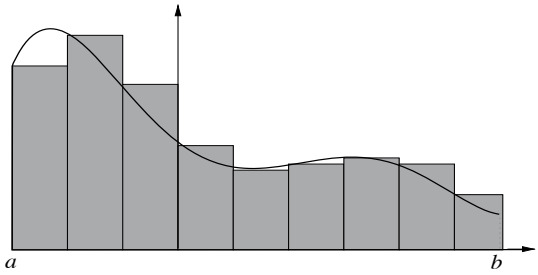
Ainsi :

$$\int_a^b f(t)\,dt \approx \sum_{k=0}^{n-1} f(a_k) \times h = h \sum_{k=0}^{n-1} f(a_k)$$

Plus précisément, si $f : [a,b] \to \mathbf{R}$ est une fonction de classe $\mathcal{C}^1$, nous avons :

$$\forall n \in \mathbf{N}^\star, \quad \left| S_n - \int_a^b f(t)\,dt \right| \leqslant \sup_{a \leqslant x \leqslant b} |f'(x)| \, \frac{(b-a)^2}{2n}$$

Illustration :



L'algorithme

```
Fonction Integrale_rectangle(a,b,f,n)
Donnees
f : fonction;
a,b,h, S : reels
k,n : entiers
Debut
h <-- (b-a) / n
S <-- 0
Pour k variant de 0 à n-1
    Faire
    S <-- S + f(a+k*h)
    Fin Faire
Retourner (h*S)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction `Integrale_rectangle` prend en argument deux réels a et b, une fonction f définie sur $[a, b]$ et un entier n. Elle retourne une valeur approchée de l'intégrale de a à b de f obtenue par la méthode des rectangles en subdivisant le segment $[a, b]$ en n.

■ **Terminaison :** La terminaison de la boucle inconditionnelle ne pose pas de problème.

■ **Correction :** Un invariant de boucle est ici « après k itérations, la valeur de la variable informatique S est égale à la somme $f(a_0) + \cdots + f(a_{k-1})$ ». En sortie de boucle, c'est-à-dire après n itérations, $S * h$ est précisément égale à S_n.

■ **Complexité :** À chaque passage en boucle, il y a 6 OPEL. Compte tenu des initialisations et du retour de résultat, la complexité de cet algorithme s'évalue à $6n + 6$ OPEL.

Le programme en Python

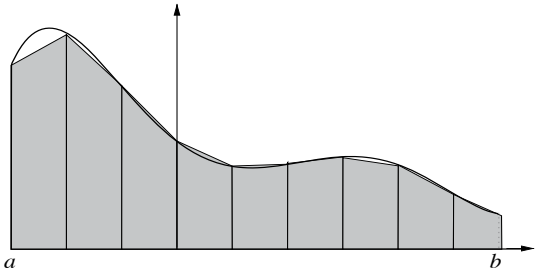
```
def Integrale_rectangle(a,b,f,n):
    h=(b-a)/n
    S=0
    for k in range(n):
        S= S+f(a+k*h)
    return(S*h)
```

Méthode des trapèzes

L'idée consiste à remplacer les rectangles utilisés dans la construction précédente par des trapèzes qui s'appuient sur le graphe de f. Avec les notations précédentes, on pose :

$$T_n(f) = \frac{b-a}{n} \sum_{i=0}^{n-1} \frac{f(a_i) + f(a_{i+1})}{2}$$

Illustration



Clairement, pour une subdivision de même pas, l'approximation est bien meilleure.

Effectivement, on peut démontrer le résultat suivant :

Si f est de classe $\mathcal{C}^2$ sur $[a,b]$, alors

$$\boxed{\left|T_n(f) - \int_a^b f(t)\,dt\right| \leqslant M\frac{(b-a)^3}{12n^2},}$$

où $M = \sup_{[a,b]} |f''|$.

L'algorithme

```
Fonction Integrale_trapezes(a,b,f,n)
Donnees
f : fonction;
a,b,h,T: reels
k,n : entiers
Debut
h <-- (b-a) / n
T <-- (f(a)+f(b))/2
Pour k variant de 1 à n-1
  Faire
      T <-- T + f(a+k*h)
  Fin Faire
Retourner (h*T)
Fin
```

Analyse de cet algorithme :

- **Spécification :** La fonction `Integrale_trapeze` prend en argument deux réels a et b, une fonction f définie sur $[a, b]$ et un entier n. Elle retourne une valeur approchée de l'intégrale de a à b de f obtenue par la méthode des trapèzes en subdivisant le segment $[a, b]$ en n.

- **Terminaison :** La terminaison de la boucle inconditionnelle ne pose pas de problème.

- **Correction :** Un invariant de boucle est ici après k itérations, la valeur de la variable informatique T est égale à la somme :

$$T = \frac{f(a) + f(b)}{2} + \sum_{i=1}^{k-1} f(a_i).$$

En sortie de boucle, c'est-à-dire après n itérations, T vaut donc

$$\begin{aligned} T &= \frac{f(a) + f(b)}{2} + \sum_{i=1}^{n-1} f(a_i) = \frac{1}{2}\sum_{i=0}^{n-1} f(a_i) + \frac{1}{2}\sum_{i=1}^{n} f(a_i) \\ &= \frac{1}{2}\sum_{i=0}^{n-1}[f(a_i) + f(a_{i+1})] \end{aligned}$$

$T * h$ est bien égal à T_n.

- **Complexité :** À chacun des $n - 1$ passages en boucle, il y a 6 OPEL. Compte tenu des initialisations et du retour de résultat, la complexité de cet algorithme s'évalue à $6n + 4$ OPEL.

Le programme en Python

```
def Integrale_trapeze(f,a,b,n):
    h=(b-a)/n
    T=(f(a)+f(b))/2
    for k in range(1,n):
        T=T+f(a+k*h)
    return h*T
```

Boîte à outils

NUMPY dispose d'une fonction trapz qui permet de calculer la valeur approchée d'une intégrale directement à partir de la liste des valeurs de f.

Si $t = [t_0, t_1, \ldots, , t_n]$ et y est la distribution de valeurs de f correspondantes, $y = [f(t_0), f(t_1), \ldots, f(t_n)]$.

`trapz(y,x)`	retourne la valeur approchée de l'intégrale de f entre t_0 et t_n calculée par la méthode des trapèzes

« Comme la Hongrie, le monde informatique a une langue qui lui est propre. Mais il y a une différence. Si vous restez assez longtemps avec des Hongrois, vous finirez bien par comprendre de quoi ils parlent. »

Dave Barry, journaliste américain.

9. Résolution numérique des équations différentielles

Hormis quelques cas d'école, on ne sait pas déterminer d'expression analytique exacte pour les solutions d'équations différentielles. On s'intéresse alors à des fonctions qui *approchent* ces solutions, obtenues par la **méthode d'Euler**.

Équation différentielle scalaire

On souhaite résoudre le problème de Cauchy suivant :

$$\left\{ \begin{array}{l} y' = f(t, y(t)) \text{ pour tout } t \in [a, b] \\ y(a) = y_0 \end{array} \right. \qquad (C_1)$$

Comme précédemment, on effectue une subdivision régulière de $[a, b]$ de pas $h = \frac{b-a}{n}$.

$$t_k = a + k\,\frac{b-a}{n}, \text{ pour } k \in [\![0, n]\!].$$

Il s'agit de déterminer une approximation de y_k de la valeur exacte $y(t_k)$. Or d'après le théorème fondamental du calcul intégral, on a pour $k \in [\![0, n-1]\!]$:

$$\begin{aligned} y(t_{k+1}) - y(t_k) &= \int_{t_k}^{t_{k+1}} y'(u)\,du \\ &= \int_{t_k}^{t_{k+1}} f(u, y(u))\,du \\ &\approx hf(t_k, y(t_k)) \end{aligned}$$

La méthode d'Euler consiste à poser que $y_{k+1} - y_k$ est exactement égal à $hf(t_k, y_k)$.

Graphiquement, cela revient à remplacer le graphe de y par des fonctions affines sur chaque sous-intervalle de la subdivision. D'un point de vue algorithmique, on calculera les approximations y_0, y_1, y_2 ... de proche en proche à l'aide des relations :

$$\begin{cases} y_0 = y(a) \\ \forall k \in [\![0, n-1]\!],\ y_{k+1} = y_k + hf(t_k, y_k) \end{cases}$$

L'algorithme

```
Fonction Euler_scalaire(a,b,y0,f,n)
Donnees
f : fonction de deux variables
a,b,h : réels
n,k : entiers
t,y : listes de réels
Debut
h <-- (b-a)/n
t <-- [0 pour k variant de 0 à n]
y <-- [0 pour k variant de 0 à n]
t[0] <-- a
y[0] <-- y0
Pour k variant de 0 à n-1
    Faire
    t[k+1] <-- t[k] + h
    y[k+1] <-- y[k] +  h * f (t[k], y[k])
    Fin Faire
Retourner (t,y)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction `Euler_scalaire` prend en entrée les bornes `a` et `b` d'un intervalle, une valeur réelle initiale `y_0`, une fonction de deux variables `f` et un entier `n`. Elle retourne deux listes :

- une subdivision régulière $(t_k)_{k\in[\![0,n]\!]}$ du segment $[a,b]$ en n sous-intervalles ;
- les valeurs approchées $(y[k])_{k\in[\![0,n]\!]}$ de la solution du problème de Cauchy (C_1) aux points de la subdivision.

■ **Terminaison :** La terminaison de la boucle inconditionnelle ne pose pas de problème.

■ **Correction :** Un invariant de boucle est « après k itérations, les $k+1$ premiers termes des listes t et y correspondent à ceux des suites (t_n) et (y_n)».

■ **Complexité :** Chaque passage en boucle génère 7 OPEL. Compte tenu des initialisations et du retour du résultat, le nombre d'OPEL s'évalue à $7n+8$.

Le programme en Python

```
def Euler_scalaire(a,b,y0,f,n):
    h= (b-a)/n
    t=(n+1)*[0]
    y=(n+1)*[0]
    t[0]= a
    y[0]= y0
    for k in range (n):
        t[k+1] = t[k] + h
        y[k+1] = y[k] + h * f (t[k], y[k])
    return (t,y)
```

Équation différentielle vectorielle

Très souvent, l'état du système est décrit par plusieurs variables. Dans ce cas, la loi d'évolution prend la forme d'un système d'équations différentielles linéaires :

$$\left\{\begin{array}{rcl} y_1'(t) & = & F_1(t,y_1(t),y_2(t)) \\ y_2'(t) & = & F_2(t,y_1(t),y_2(t)) \end{array}\right.$$

Pour traiter ces systèmes d'équations différentielles, on considère que la fonction F est vectorielle. Plus précisément, on considère les deux inconnues y_1 et y_2 comme les coordonnées d'une fonction vectorielle :

$$\begin{array}{rccc} Y & : & I & \to & \mathbb{R}^2 \\ & & t & \mapsto & [y_1(t), y_2(t)] \end{array}.$$

De même, on pose $F = [F_1, F_2]$. Ainsi, la loi d'évolution prend la forme bien connue $Y'(t) = F(t, Y(t))$. Finalement, nous pouvons réécrire le système d'équations différentielles scalaires sous forme d'une équation différentielle vectorielle :

$$\begin{cases} Y' = F(t, Y(t)) \text{ pour tout } t \in [a, b] \\ Y(a) = Y_0 \end{cases} \qquad (C_2)$$

L'algorithme

```
Fonction Euler_vectoriel(a,b,Y0,F,n)
Donnees
F : fonction de deux variables
a,b,h : réels
n,k : entiers
t : listes de réels
Y : listes de vecteurs
Y0 : vecteur
Debut
h <-- (b-a)/n
t <--  [0 pour k variant de 0 à n]
Y <-- [[0 pour k variant de 0 à n],
    ↳[0 pour k variant de 0 à n]]
t[0] <-- a
Y[0] <-- Y0
Pour k variant de 0 à n-1
    Faire
    t[k+1] <-- t[k] + h
    Y[k+1] <-- Y[k] + h * F (t[k], Y[k])
    Fin Faire
Retourner (t,Y)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction `Euler_vectoriel` prend en entrée les bornes `a` et `b` d'un intervalle, un couple de valeurs réelles initiales `Y_0`, une fonction de deux variables `F` et un entier `n`. Elle retourne deux listes :

• une subdivision régulière $(t_k)_{k\in[\![0,n]\!]}$ du segment $[a, b]$ en n sous-intervalles ;

• les valeurs approchées $(Y[k])_{k\in[\![0,n]\!]}$ de la solution du problème de Cauchy (C_2) aux points de la subdivision.

■ **Terminaison :** La terminaison de la boucle inconditionnelle ne pose pas de problème.

■ **Correction :** Un invariant de boucle est « après k itérations, les $k + 1$ premiers termes des listes t et Y correspondent à ceux des suites (t_n) et (Y_n) ».

■ **Complexité :** Chaque passage en boucle génère 7 OPEL. Compte tenu des initialisations et du retour du résultat, le nombre d'OPEL s'évalue à $7n + 8$.

Le programme en Python

Si les algorithmes de la méthode d'Euler sont presque identiques entre versions scalaire et vectorielle, leurs implémentations en PYTHON diffèrent plus notablement.

En effet, comme `Y[k]` et `F(t[k],Y[k])` sont des listes (couples), le symbole d'addition $+$ serait interprété comme concaténation de listes, de même que la multiplication `h * F(t[k],Y[k])`.

Pour contourner cette difficulté, on peut faire ces opérations « coordonnées par coordonnées », ou bien utiliser le format `array` de NUMPY.

```
def Euler_vectoriel(a,b,Y0,F,n):
    h= (b-a)/n
    t=[0 for k in range(n+1)]
    Y=[[0,0] for k in range(n+1)]
    t[0]= a
    Y[0][:]= Y0[:]
    for k in range (n):
        t[k+1] = t[k] + h
        Y[k+1][0] = Y[k][0] + h*F(t[k],Y[k])[0]
        Y[k+1][1] = Y[k][1] + h*F(t[k],Y[k])[1]
    return (t,Y)
```

Le programme avec Numpy

```
def Euler1(a,b,Y0,F,n):
    h= (b-a)/n
    p=len(Y0)
    t=np.linspace(a,b,n+1)
    Y=np.zeros(shape=(p,n+1),dtype=float)
    Y[:,0]= Y0[:,0]
    for k in range(n):
        Y[:,k+1]= Y[:,k]+h*F(t[k],Y[:,k])
    return (t,Y)
```

Remarques :

• Ici, les vecteurs ou listes de vecteurs sont des tableaux de type ndarray. Il en va ainsi de Y, Y0, t. De même la fonction F passée en paramètre est une fonction de deux variables : un réel et un tableau (p,1). Elle retourne également un tableau (p,1).

• Cette dernière version permet de traiter des équations différentielles vectorielles dans $\mathbb{R}^p$, soit des systèmes de p équations différentielles.

Boîte à outils

Pour résoudre le problème de Cauchy

$$(C) \qquad \left\{ \begin{array}{rcl} y'(t) & = & f(y(t),t) \\ y(t_0) & = & y_0 \end{array} \right.$$

où y est une fonction à valeurs réelles ou vectorielles, on utilise la fonction odeint du module scipy.integrate

`odeint(f,y0,t)` retourne la solution du problème de Cauchy (C)

Ici, t est un tableau de valeurs de la forme $t = [t_0, t_1, ..., t_n]$. La solution retournée correspond à la distribution des valeurs approchées de f aux différents instants $t_0, t_1, \ldots, t_n$.

Équation différentielle d'ordre supérieur

Une équation différentielle scalaire d'ordre supérieur peut se ramener à une équation différentielle d'ordre 1, à valeur vectorielle.
Prenons l'exemple d'une équation différentielle scalaire d'ordre 2 résolue en y'' de la forme :

$$y'' = f(t, y(t), y'(t))$$

Nous allons introduire l'inconnue $Y = \begin{pmatrix} y \\ y' \end{pmatrix}$ de sorte que cette équation puisse s'écrire (matriciellement) sous la forme :

$$\begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}' = \begin{pmatrix} y'(t) \\ f(t, y(t), y'(t)) \end{pmatrix}$$

Ainsi, en définissant la fonction vectorielle

$$F(t, y_0, y_1) = [y_1, f(t, y_0, y_1)],$$

on se ramène à résoudre l'équation différentielle d'ordre 1 :

$$Y'(t) = F(t, Y(t))$$

Plus précisément, on résout numériquement le problème de Cauchy d'ordre 2 :

$$\begin{cases} y'' = f(t, y(t), y'(t)) \text{ pour tout } t \in [a, b] \\ y(a) = d_0, y'(a) = d_1 \end{cases} \qquad (C_3)$$

L'algorithme

```
Fonction Euler_ordre2(a,b,d0,d1,f,n)
Donnees
a,b,d0,d1,h : réels
t, Y0,Y1 : listes de réels
k,n : entiers
f : fonction
Debut
h <-- (b-a)/n
t <--  [0 pour k variant de 0 à n]
Y0 <-- [0 pour k variant de 0 à n]
Y1 <-- [0 pour k variant de 0 à n]
t[0] <-- a
Y0[0] <-- d0
Y1[0] <-- d1
Pour k variant de 0 à n-1
    Faire
    t[k+1] <-- t[k] + h
    Y0[k+1] <-- Y0[k] + h * Y1[k]
    Y1[k+1] <-- Y1[k] +  h * f(t[k],Y0[k],Y1[k])
    Fin Faire
Retourner (t,Y0)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction `Euler2` prend en entrée les bornes `a` et `b` d'un intervalle, un couple de valeurs réelles initiales `d0,d1`, une fonction de trois variables `f` et un entier `n`. Elle retourne deux listes :

• une subdivision régulière $(t_k)_{k \in [\![0,n]\!]}$ du segment $[a, b]$ en n sous-intervalles ;

• les valeurs approchées $(Y0[k])_{k\in[\![0,n]\!]}$ de la solution du problème de Cauchy (C_3) aux points de la subdivision.

■ **Terminaison :** La terminaison de la boucle inconditionnelle ne pose pas de problème.

■ **Correction :** Un invariant de boucle est « après k itérations, les $k+1$ premiers termes des listes t et $Y0$ correspondent à ceux des suites (t_n) et $(Y0_n)$ obtenu par la méthode d'Euler ».

■ **Complexité :** Chaque passage en boucle génère 9 OPEL. Compte tenu des initialisations et du retour du résultat, le nombre d'OPEL s'évalue à $9n+10$.

Le programme en Python

```
def Euler_ordre2(a,b,d0,d1,f,n):
    h= (b-a)/n
    t=[0 for k in range(n+1)]
    Y0=[0 for k in range(n+1)]
    Y1=[0 for k in range(n+1)]
    t[0]= a
    Y0[0]=d0
    Y1[0]=d1
    for k in range (n):
        t[k+1] = t[k] + h
        Y0[k+1] = Y0[k] + h*Y1[k]
        Y1[k+1] = Y1[k] + h*f(t[k],Y0[k],Y1[k])
    return (t,Y0)
```

« L'ordinateur est né pour résoudre des problèmes qui n'existaient pas auparavant. »

Bill Gates, informaticien américain.

10. Résolution numérique d'un système linéaire

On s'intéresse dans ce chapitre à la résolution de systèmes (S) de n équations linéaires à n inconnues de la forme :

$$\left\{\begin{array}{lllllllll} a_{0,0}x_0 & + & a_{0,1}x_1 & + & a_{0,2}x_2 & +\cdots+ & a_{0,n-1}x_{n-1} & = & b_0 \\ a_{1,0}x_0 & + & a_{1,1}x_1 & + & a_{1,2}x_2 & +\cdots+ & a_{0,n-1}x_{n-1} & = & b_1 \\ a_{2,0}x_0 & + & a_{2,1}x_1 & + & a_{2,2}x_2 & +\cdots+ & a_{0,n-1}x_{n-1} & = & b_2 \\ \vdots & & & & \vdots & & & & \vdots \\ a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & a_{n-1,2}x_2 & +\cdots+ & a_{n-1,n-1}x_{n-1} & = & b_{n-1} \end{array}\right.$$

Un tel système peut être représenté par l'équation matricielle :

$$AX = B$$

où A est la matrice carrée d'ordre n :

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & a_{2,n-1} \\ \vdots & & & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} \end{pmatrix}$$

et les matrices colonnes X et b sont données par :

$$X = {}^t[x_0\ x_1\ x_2\ \cdots\ x_{n-1}] \text{ et } B = {}^t[b_0\ b_1\ b_2\ \cdots\ b_{n-1}].$$

Lorsque (S) est de Cramer, c'est-à-dire ayant une unique solution, la **méthode de Gauss** permet de se ramener à la résolution d'un système triangulaire inversible (T), c'est-à-dire de la forme :

$$\begin{cases} \boldsymbol{a_{0,0}}x_0 + a_{0,1}x_1 + a_{0,2}x_2 + \cdots + a_{0,n-1}x_{n-1} = b_0 \\ \boldsymbol{a_{1,1}}x_1 + a_{1,2}x_2 + \cdots + a_{1,n-1}x_{n-1} = b_1 \\ \boldsymbol{a_{2,2}}x_2 + \cdots + a_{2,n-1}x_{n-1} = b_2 \\ \ddots \quad \vdots \\ \boldsymbol{a_{n-1,n-1}}x_{n-1} = b_{n-1} \end{cases}$$

où les coefficients diagonaux $\boldsymbol{a_{0,0}}, \ldots, \boldsymbol{a_{n-1,n-1}}$ sont non nuls.

Résolution d'un système triangulaire

On résout un tel système par remontée. Cela consiste à résoudre la dernière équation, puis de *remonter* à l'avant-dernière en substituant à x_{n-1} sa valeur. Ainsi, l'avant-dernière équation est une équation du premier degré en x_{n-2} que l'on résout, puis on *remonte* à la $(n-2)^{\text{ième}}$ équation en substituant aux inconnues x_{n-1} et x_{n-2} les valeurs précédemment obtenues et ainsi de suite.

L'algorithme

```
Fonction Solution_remontee(A,B):
Donnees
A = A[i][j] : matrice carrée
X =X[i][0], B=B[i][0] : matrices colonnes
somme : réel
i,j,n : entiers
Debut
# resout par remontee un SEL triangulaire AX=b
n <-- Longueur (A) # nombre de lignes,
X <-- [[0] pour i variant de 0 à n-1]
Pour i variant de n-1 à 0
    Faire # x_i = (b_i - a_{i,i+1}x_{i+1} + ... + a_{i,n-1}x_{n-1})/a_{i,i}
    somme <--0
    Pour j variant de i+1 à n-1
        Faire
        somme <-- somme + A[i][j] * X[j][0]
        Fin Faire
    X[i][0] <-- (B[i][0] - somme)/A[i][i]
```

```
    Fin Faire
Retourner (X)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** La fonction `Solution_remontee` prend en entrée une matrice triangulaire inversible A, sous forme d'une liste de listes et une matrice colonne B sous forme d'une liste. Elle retourne en sortie l'unique matrice colonne X solution du système linéaire $AX = B$, elle-même sous forme d'une liste.

■ **Terminaison :** La terminaison des deux boucles inconditionnelles ne pose pas de problème.

■ **Correction :** Notons pour $i \in [\![0, n-1]\!]$, x_i la $i+1^{\text{ième}}$ coordonnée du vecteur colonne solution, $X = {}^t[x_0 \cdots x_{n-1}]$. Un invariant de boucle est :

$$(\mathcal{P}_i) \quad X[n-1][0] = x_{n-1}, \ldots, X[i][0] = x_i.$$

• **Initialisation :** Lors du premier passage en boucle, $i = n-1$. Par conséquent la variable somme est nulle et

$$X[n-1][0] = B[n-1]/A[n-1][n-1]$$

coïncide bien avec x_{n-1}.

• **Hérédité :** Soit i tel que $\mathcal{P}_i$ est vraie. Au prochain passage en boucle, la variable somme prend la valeur

$$somme = \sum_{j=i}^{n-1} A[i-1][j]X[j][0].$$

Dans cette somme, n'interviennent que les $X[j][0]$ pour $j \geqslant i$. D'après $\mathcal{P}_i$, on a $somme = \sum_{j=i}^{n-1} A_{i-1,j}x_j$. Par conséquent,

$$X[i-1][0] = \big(b_{i-1} - \sum_{j=i}^{n-1} A_{i-1,j}x_j\big)/A_{i-1,i-1}$$

Ce qui coïncide également avec x_{i-1}.

- **Conclusion :** En sortie de la boucle principale, on a bien

$$X[n-1][0] = x_{n-1}, \ldots, X[0][0] = x_0.$$

La liste retournée est donc celle des coordonnées du vecteur solution.

■ **Complexité :** Le nombre de passages en boucle principale est égal à n, mais le nombre d'OPEL à chaque passage dépend de i. Ainsi, dans la boucle principale, le nombre d'OPEL est égal à :

$$\sum_{i=0}^{n-1}[4 + 4 \times (n-i-1)] = 4n + 4\frac{n(n-1)}{2} = 2n^2 + 2n.$$

Compte tenu des initialisations et du retour de résultat, le nombre d'OPEL est de $2n^2 + 2n + 3$.

Le programme en Python

```
def Solution_remontee(A,B):
    # resout par remontee un SEL triangulaire
    n= len(A)
    X= [[0] for i in range(n)]
    for i in range (n-1,-1,-1):
        somme=0
        for j in range (i+1,n):
            somme=somme + A[i][j] * X[j][0]
        X[i][0]=(B[i][0]-somme)/A[i][i]
    return (X)
```

Boîte à outils

Le module **linalg** de SCIPY propose la fonction solve_triangular qui résout les systèmes triangulaires -supérieurs par défaut)

solve_triangular(A,B)	si A est une matrice triangulaire, retourne la solution du système $AX = B$, ou bien un message d'erreur le cas échéant

Opérations élémentaires sur les lignes d'un système ou d'une matrice

Le théorème de Gauss montre que tout système de Cramer est équivalent à un système triangulaire inversible. En outre, la démonstration de ce théorème fournit un algorithme que nous détaillerons dans la prochaine section.

Les outils de base pour mettre en œuvre cette méthode sont les opérations élémentaires sur les lignes d'un système (ou d'une matrice A). Ces opérations consistent à :

- Rechercher dans une colonne fixée de A, disons celle d'indice i, le coefficient $A_{j,i}$ qui a la plus grande valeur absolue parmi tous les coefficients $A_{k,i}$ avec $k \geqslant i$. Ce coefficient sera appelé **pivot**.
- Échanger deux lignes (L_i et L_j) de la matrice A.
- Ajouter à la ligne L_i un multiple d'une autre ligne, L_j.

Les algorithmes

```
Fonction Chercher_pivot(A,i)
Donnees
A : matrice
i,j,k,n : entiers
Debut
n <-- Longueur(A) # nombre de lignes de A
j <-- i
Pour k variant de i+1 à n-1
    Faire
    Si |A[k][i]| > |A[j][i]|
        Alors j <-- k
    Fin Si
Retourner(j)
Fin
```

```
Fonction Echanger_ligne(A,i,j)
Donnees
A : matrice
i,j,k,n : entiers
```

```
temp : réel
Debut
n <-- Longueur(A[0]) # nombre de colonnes de A
Pour k variant de 0 à n-1
    Faire
    temp <-- A[i][k]
    A[i][k] <-- A[j][k]
    A[j][k] <-- temp
    Fin Faire
Fin
```

```
Fonction Ajouter_ligne(A,i,j,mu)
Donnees
A : matrice
i,j,k,n : entiers
mu : réel
Debut
n <-- Longueur(A[0]) # nombre de colonnes de A
Pour k variant de 0 à n-1
    Faire
    A[i][k] <-- A[i][k] + mu * A[j][k]
    Fin Faire
Fin
```

Remarque : Les deux dernières fonctions `Echanger_ligne` d'une part et `Ajouter_ligne` d'autre part, nne retournent rien.
Elles modifient la matrice A « sur place ».

Analyse de ces algorithmes :

- **Spécification :**

• La fonction `Chercher_pivot` prend en entrée une matrice A et un numéro de colonne i et retourne le numéro de la ligne j du pivot.

• La fonction `Echanger_ligne` prend en entrée une matrice A et deux numéros de lignes i et j. Elle échange dans la matrice A les lignes L_i et L_j.

• La fonction `Ajouter_ligne` prend en entrée une matrice A, deux numéros de lignes i,j et un réel mu. Elle remplace dans la matrice A la ligne L_i par $L_i + mu \cdot L_j$.

■ **Terminaison :** Les terminaisons de ces boucles inconditionnelles ne posent pas de problème.

■ **Complexité :**

• Le nombre d'itérations dans la fonction `Chercher_pivot` est de $n-i-1$. À chaque passage, il y a 5 OPEL, donc la complexité de cet algorithme est de $5(n-i)-2$.

• Le nombre d'OPEL de la fonction `Echanger_ligne` est de $4n+1$.

• Le nombre d'OPEL de la fonction `Ajouter_ligne` est également de $4n+1$.

Les programmes en Python

```
def Chercher_pivot (A,i):
    n=len(A)
    j=i
    for k in range (i+1,n):
        if abs(A[k][i]) > abs (A[j][i]):
            j = k
    return j
```

```
def Echanger_ligne (A,i,j):
    n=len(A[0])# le nombre de colonnes
    for k in range(n):
    A[i][k],A[j][k] = A[j][k], A[i][k]
```

```
def Ajouter_ligne (A, i, j, mu):
    n=len(A[0])# le nombre de colonnes
    for k in range (n):
        A[i][k] = A[i][k] + mu * A[j] [k]
```

Résolution d'un système par méthode de Gauss

D'après le théorème de Gauss, tout système de Cramer est équivalent à un système triangulaire inversible, c'est-à-dire ayant tous ses

coefficients diagonaux non nuls. En effet, considérons le système de Cramer :

$$\begin{cases} a_{0,0}^{(0)}x_0 + a_{0,1}^{(0)}x_1 + \cdots + a_{0,n-1}^{(0)}x_{n-1} = b_0^{(0)} \\ a_{1,0}^{(0)}x_0 + a_{1,1}^{(0)}x_1 + \cdots + a_{0,n-1}^{(0)}x_{n-1} = b_1^{(0)} \\ \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \qquad\quad \vdots \\ a_{n-1,0}^{(0)}x_0 + a_{n-1,1}^{(0)}x_1 + \cdots + a_{n-1,n-1}^{(0)}x_{n-1} = b_{n-1}^{(0)} \end{cases}$$

Notons $A^{(0)}$ la matrice carrée d'ordre n constituée des coefficients de (S) et la matrice colonne $B^{(0)}$ formée des seconds membres. On a donc

$$(S) \iff A^{(0)}X = B^{(0)}$$

[1] Chercher un pivot : On observe que tous les coefficients de x_0 ne sont pas nuls (sans quoi (S) n'aurait pas une unique solution). On recherche parmi ces coefficients celui qui a la plus grande valeur absolue, on l'appelle le **pivot**.

[2] Échanger deux lignes : On échange si nécessaire la ligne d'indice 0 avec celle du pivot pour ramener le pivot en haut à droite de la matrice. On obtient

$$\begin{cases} \boldsymbol{\tilde{a}_{0,0}^{(0)}}x_0 + \tilde{a}_{0,1}^{(0)}x_1 + \cdots + \tilde{a}_{0,n-1}^{(0)}x_{n-1} = \tilde{b}_0^{(0)} \quad (L_0) \\ \tilde{a}_{1,0}^{(0)}x_0 + \tilde{a}_{1,1}^{(0)}x_1 + \cdots + \tilde{a}_{0,n-1}^{(0)}x_{n-1} = \tilde{b}_1^{(0)} \quad (L_1) \\ \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \qquad\quad \vdots \\ \tilde{a}_{n-1,0}^{(0)}x_0 + \tilde{a}_{n-1,1}^{(0)}x_1 + \cdots + \tilde{a}_{n-1,n-1}^{(0)}x_{n-1} = \tilde{b}_{n-1}^{(0)} \ (L_{n-1}) \end{cases}$$

où le pivot $\boldsymbol{\tilde{a}_{0,0}^{(0)}}$ est en première position.

[2] Ajouter aux lignes un multiple de celle du pivot : À présent, nous allons éliminer l'inconnue x_0 des $n-1$ dernières lignes du système en leur ajoutant le bon multiple de L_0. Plus précisément,

on effectue les opérations élémentaires :

$$\forall i \in [\![1, n-1]\!], \quad L_i \leftarrow L_i - \frac{\tilde{a}_{i,0}^{(0)}}{\tilde{\boldsymbol{a}}_{\mathbf{0,0}}^{\mathbf{(0)}}} \cdot L_0$$

On obtient alors le système $A^{(1)}X = B^{(1)}$:

$$\begin{cases} \boldsymbol{a}_{\mathbf{0,0}}^{\mathbf{(1)}} x_0 + a_{0,1}^{(1)} x_1 + \cdots + a_{0,n-1}^{(1)} x_{n-1} = b_0^{(1)} \\ a_{1,1}^{(1)} x_1 + \cdots + a_{0,n-1}^{(1)} x_{n-1} = b_1^{(1)} \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ a_{n-1,1}^{(1)} x_1 + \cdots + a_{n-1,n-1}^{(1)} x_{n-1} = b_{n-1}^{(1)} \end{cases}$$

À la fin de ce premier cycle, nous avons placé un pivot en première position et « vidé » la colonne en dessous. Ensuite, il s'agit de reprendre le cycle à l'étape **[1]** au sous-système constitué des $n-1$ dernières équations. On obtient ainsi successivement :

$$\begin{aligned} (S) &\iff A^{(0)}X = B^{(0)} \\ &\iff A^{(1)}X = B^{(1)} \\ &\vdots \\ &\iff A^{(n-1)}X = B^{(n-1)} \end{aligned}$$

Ce dernier système ayant au moins $n-1$ pivots sur la diagonale, il en a forcément exactement n : il s'agit d'un système triangulaire inversible.

L'algorithme

```
Fonction Solution_Gauss(A,B)
Donnees
A,vA : matrices carrées sous forme de liste
B,vB : matrices colonnes sous forme de liste
i,j,k,n : entiers
mu : réel
Debut
```

```
n <-- Longueur(A)
vA <-- A #on travaille sur des copies de A
vB <-- B # et de B que l'on pourra modifier
# trigonalisation du système
Pour i variant de 0 à n-1
    Faire
    j <-- Chercher_pivot(vA,i)
    Si j < i
        Alors # remonter la ligne du pivot
        Echanger_ligne(vA,i,j)
        Echanger_ligne(vB,i,j)
    Fin Si
    Pour k variant de i+1 à n-1
        Faire # échelonner sous vA[i][i]
        mu= -vA[k][i]/vA[i][i]
        Ajouter_ligne(vA,k,i,mu)
        Ajouter_ligne(vB,k,i,mu)
# résolution par remontée
Retourner Solution_remontee(vA,vB))
Fin
```

Analyse de cet algorithme :

- **Spécification :** La fonction `Solution_Gauss` prend en entrée une matrice carrée inversible A=A[i][j] ainsi qu'une matrice colonne B=B[i][0]. Elle retourne l'unique solution du système d'équations linéaires représenté matriciellement par l'équation $AX = B$, sous la forme d'une matrice colonne X=X[i][0].

- **Terminaison :** L'algorithme ne mettant en jeu que des boucles inconditionnelles, sa terminaison est indubitable.

- **Correction :** Notons pour $i \geqslant 0$, $vA^{(i)}$ et $vB^{i)}$ les valeurs des variables informatiques vA et vB après i passages dans la boucle principale. Un invariant de boucle est alors :

$$\text{« } vA^{(i)} = A^{(i)} \text{ et } vB^{(i)} = B^{(i)} \text{ ».}$$

Conformément à ce qui a été expliqué précédemment, on sait qu'en sortie de boucle, après n étapes, $A^{(n)}$ est une matrice triangulaire supérieure à coefficients diagonaux tous non nuls et que

$$AX = B \iff A^{(n)}X = B^{(n)}$$

On peut donc appliquer la fonction `Solution_remontee` pour obtenir la solution du système initial.

■ **Complexité :** La boucle principale compte n itérations. À chacune de ces itérations, on effectue une boucle secondaire qui compte $n-i-1$ itérations. Finalement, chaque passage dans cette deuxième boucle fait deux appels à la fonction `Ajouter_ligne` dont on sait qu'elle coûte $4n+1$ OPEL !

Au final, le nombre d'OPEL est de l'ordre de n^3.

Le programme en Python

```
def Solution_Gauss(A,B):
    """Résolution de AX=B """
    n = len(A)
    vA = [A[i][:] for i in range(n)]
    vB = [B[i][:] for i in range(n)]
    # trigonalisation du système
    for i in range(n):
        j = Chercher_pivot(vA,i)
        if j > i:
            Echanger_ligne(vA,i,j)
            Echanger_ligne(vB,i,j)
        for k in range (i+1,n):
            mu=-vA[k][i]/float(vA[i][i])
            Ajouter_ligne (vA,k,i,mu)
            Ajouter_ligne (vB,k,i,mu)
    # resolution par remontee
    return (Solution_remontee(vA,vB))
```

Boîte à outils

Le module **linalg** de SCIPY propose la fonction solve qui résout les systèmes d'équations linéaires :

$$(S) \qquad AX = B$$

solve(A,B)	retourne la solution du système $AX = B$, ou bien un message d'erreur le cas échéant

« Le nombre de femmes que j'aurais eues si je n'avais pas un PYTHON chez moi, c'est fou. L'embarras du choix, c'est l'angoisse. »

Romain Gary, écrivain français.

Partie 3

Algorithmes de deuxième année

LE SAVIEZ-VOUS ?

Algorithme

Le mot *algorithme*, anagramme du mot *logarithme*, semble tout droit tiré de vocables grecs. Il n'en est rien puisqu'il provient du nom latinisé puis francisé du mathématicien persan **Al-Khwarizmi** qui vivait à Bagdad au début du IX^{e} siècle. On doit à ce savant une méthode de résolution complète des équations du second degré, en quelque sorte un algorithme ; il est aussi l'auteur d'un ouvrage expliquant l'utilisation des chiffres arabes, ceux que nous utilisons. Son ouvrage fut traduit en latin au Moyen Âge et son nom transformé en *Algorithmus*.

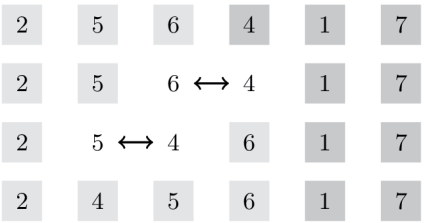
Le premier algorithme connu est celui de la division euclidienne élaboré par **Euclide** au III^{e} siècle avant notre ère. De nombreux autres sont apparus en mathématiques mais le développement de l'informatique leur a donné de nos jours un rôle fondamental dans la gestion de l'activité humaine.

11. Algorithmes de tri

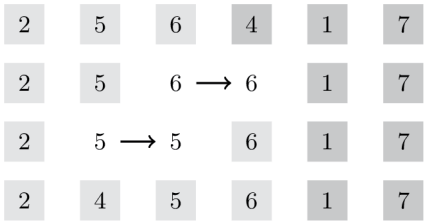
Dans tout le chapitre, on décrit des algorithmes pour trier des listes de nombres. On spécifie ici le type entier pour les éléments des listes considérées, mais les fonctions PYTHON écrites fonctionnent aussi bien avec des listes contenant des entiers et/ou des flottants.

Tri par insertion

Le principe de ce tri est le suivant. On découpe la liste L considérée en deux sous-listes, la première triée et la deuxième contenant le reste des éléments de L. On considère le premier élément de la partie non-triée, et on l'insère dans la partie triée en comparant successivement aux élément à sa gauche, comme dans les diagrammes suivants :



En répétant ce procédé on obtient une liste triée une fois inséré le dernier terme de la liste. On peut utiliser moins d'affectations en stockant la valeur que l'on insère et en se contentant de faire des copies successives des élements vers la droite. Ainsi on obtient cette fois-ci les diagrammes :



À la dernière étape on a recopié le 4 stocké à la place du dernier élément déplacé.

L'algorithme

```
Fonction TriInsertion(L)
Donnees
L : liste d'entiers,
a, i, j : entier
Debut
    Pour i variant de 1 à len(L)
    Faire
        a <-- L[i]
        j <-- i
        Tant Que j > 0 et L[j-1] > a
        Faire
            L[j] <-- L[j-1]
            j<--j-1
        Fin Faire
        L[j] <-- a
    Fin Faire
    Retourner L
Fin
```

Analyse de cet algorithme :

■ **Spécification** : Cette fonction prend en entrée une liste d'entiers L et retourne la liste triée contenant les mêmes éléments que L. À noter que l'ordre initial des éléments de L est perdu dans la transposition PYTHON de cet algorithme.

■ **Terminaison :** La suite des valeurs de j dans la boucle Tant Que est à valeurs entières et strictement décroissante. Cette boucle termine donc forcément, ne serait-ce qu'à cause de la condition $j > 0$.

■ **Correction :** On a pour invariant de boucle : "la liste formée des éléments de L d'indice strictement inférieur à i est triée".

■ **Complexité :** On note n la longueur de la liste L. Dans le pire des cas, la liste est constituée de valeurs décroissantes. On a alors i passages dans la boucle Tant Que et comme on a 7 OPEL dans cette boucle ainsi que 3 OPEL dans la boucle Pour on obtient un total de

$$\sum_{i=1}^{n} 3 + 7i = 3n + 7\frac{n(n+1)}{2} = \frac{7n^2 + 13n}{2}$$

OPEL. Cet algorithme est donc de complexité quadratique en n.

Le programme en Python

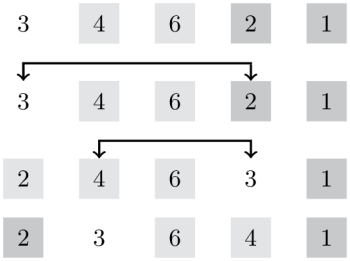
```
def TriInsertion(L):
    for i in range(1,len(L)):
        a=L[i]
        j=i
        while j>0 and L[j-1] > a:
            L[j]=L[j-1]
            j=j-1
        L[j]=a
    return L
```

Tri rapide

Ce tri est un tri récursif. Il utilise une étape intermédiaire, appelée **partition**, que nous détaillons tout d'abord. Cette étape a pour but, étant donné un élément de la liste L appelé **pivot**, de répartir avant et après le pivot les éléments de L respectivement inférieurs et strictement supérieurs au pivot.

Pour faire cela, on commence par échanger le pivot avec le premier élément de L. On parcourt ensuite les éléments de L. Si on trouve un élément inférieur ou égal au pivot, on échange le pivot avec cet élément puis le pivot avec l'élément précédemment à droite du pivot.

On peut représenter un exemple de cette opération avec les diagrammes suivants (on a déjà considérés le 4 et le 6, que l'on a laissés à leur place car ils sont strictement supérieurs au pivot 3) :



On donne maintenant l'algorithme de la partition :

L'algorithme

```
Fonction Partition(L,p)
Donnees
L : liste d'entiers,
p,i, j : entiers
Debut
    L[0],L[p] <-- L[p],L[0]
    i <-- 0
    Pour j variant de 1 à len(L)
    Faire
        Si L[j]<=L[i]
        Alors
            L[i], L[j] <-- L[j], L[i]
            L[j], L[i+1] <-- L[i+1], L[j]
            i <-- i+1
        Fin Si
    Fin Faire
    Retourner (L,i)
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste d'entiers L et un indice p appelé position du pivot. Elle retourne la liste obtenue en plaçant avant le pivot L[p] tous ceux qui lui sont inférieurs et après lui tous ceux qui lui sont strictement supérieurs. On retourne de plus l'indice correspondant à la position dans cette nouvelle liste du pivot.
- **Terminaison :** On a ici une boucle inconditionnelle terminant en len(L)-1 itérations.
- **Correction :** On a l'invariant de boucle : "les éléments d'indice inférieur à i sont inférieurs ou égaux à L[i] et L[i] est égal au pivot".
- **Complexité :** On note n la longueur de la liste. On a au pire 3 + 7n OPEL dans cet algorithme.

Le programme en Python

```
def Partition(L,p):
    L[0], L[p] = L[p], L[0]
    i = 0
    for j in range(1,len(L)):
        if L[j] <= L[i]:
            L[i], L[j] = L[j], L[i]
            L[j], L[i+1] = L[i+1], L[j]
            i = i+1
    return(L,i)
```

On peut maintenant détailler l'algorithme du tri rapide.

L'idée est choisir un pivot, effectuer la partition correspondant, puis recommencer l'opération sur les deux listes formées par les éléments situés d'une part avant et d'autre part après le pivot.

Comme tout algorithme récursif il est primordial de préciser le **cas d'arrêt** de l'algorithme. On s'arrête ici si la liste considérée contient un ou aucun élément.

On présente sur les diagrammes suivants une réalisation de cet algorithme. Les pivots choisis sont en gris les éléments fixés en foncé.

Choix du premier pivot	3	4	6	2	1
Première partition	2	1	3	4	6
Choix des pivots	2	1	3	4	6
Partitions	1	2	3	4	6

L'algorithme s'arrête ici car les listes sur lesquelles on devrait refaire l'opération sont de longueur inférieure ou égale à 1.

Nous choisissons ici la position du pivot au hasard dans la partie de liste considérée. On effectuera ce tirage au hasard en python à l'aide de la fonction rd.randint() de la bibliothèque random. On pourrait aussi prendre systématiquement le premier ou le dernier élément.

L'algorithme

```
Fonction TriRapide(L)
Donnees
L,L1,L2 : liste d'entiers,
p, k : entiers
Debut
    Si len(L)<=1
    Alors
        Retourner(L)
    Sinon
        p <-- entier au hasard entre 0 et len(L)
        L,k <-- Partition(L,p)
        L1 <-- TriRapide(L[0:k])
        L2 <-- TriRapide(L[k+1:])
        Retourner L1+[L[k]]+L2
    Fin Si
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste d'entiers et retourne la liste triée contenant les mêmes éléments que L.
- **Terminaison :** Cette fonction est récursive, et à chaque application elle se relance deux fois avec des listes de longueurs strictement

inférieures à celle de la liste donnée en entrée. L'arbre d'exécution de cette fonction récursive est donc de hauteur au plus len(L).

■ **Correction :** On démontre la correction de cet algorithme à l'aide d'une récurrence forte.

■ **Complexité :** On note n la longueur de L. Dans le pire des cas, la partition donne une des listes L1 ou L2 vide et l'autre de longueur len(L)-1. Si c'est le cas à chaque étape, on applique alors n-1 fois Partition. Comme celle-ci est de complexité linéaire en n on obtient une complexité totale quadratique en n dans le pire des cas.
On peut cependant montrer que la complexité en moyenne de cet algorithme est de l'ordre de $n\ln(n)$.

Le programme en Python

```
def TriRapide(L):
    if len(L)<=1:
        return(L)
    p = rd.randint(0,len(L)-1)
    L, k = Partition(L,p)
    L1 = TriRapide(L[0:k])
    L2 = TriRapide(L[k+1:])
    return L1+[L[k]]+L2
```

Il est également possible de réaliser cet algorithme **en place**, c'est-à-dire en ne créant pas de listes intermédiaires en machine, mais en faisant toutes les modifications dans la liste d'origine. On garde les mêmes complexités que dans le cas traité précédemment.

```
def Partition2(L,p,a,b):
   L[a], L[p] = L[p], L[a]
   i = a
   for j in range(a+1,b+1):
       if L[j] <= L[i]:
            L[i], L[j] = L[j], L[i]
            L[j], L[i+1] = L[i+1], L[j]
            i = i+1
   return(i)
```

```
def TriRapide2(L,a,b):
    if b-a>0:
        p = rd.randint(a,b)
        k = Partition2(L,p,a,b)
        TriRapide2(L,a,k-1)
        TriRapide2(L,k+1,b)
```

Recherche rapide de la médiane

En s'inspirant de ce qui a été fait pour le tri rapide, on peut construire un algorithme déterminant l'élément de rang k dans une liste L quelconque en temps linéaire. En particulier si le rang k en question est la moitié de la longueur de la liste, on peut trouver la médiane d'une liste en temps linéaire.

Comme pour le tri rapide, le principe est d'appliquer une partition avec un pivot garantissant une répartition suffisamment bonne des éléments de L autour de lui.

On commence par décrire une fonction déterminant ce bon pivot.

L'algorithme

```
Fonction MedianeLocale(L,a,b)
Donnees
L : liste d'entiers
a, b : entiers
Debut
    Retourner TriInsertion(L[a:b+1])[(b-a)//2)]
Fin

Fonction MedianeDesMedianes(L)
Donnees
L, L1 : listes d'entiers,
i : entiers
Debut
    Si len(L) < 5
```

```
    Alors
        Retourner(MedianeLocale(L,0,len(L)-1))
    Sinon
        L1 <-- [ ]
        Pour i variant de 0 à len(L)//5
        Faire
            L1 <-- L1+[MedianeLocale(L,5*i,5*i+4)]
        Fin Faire
        Retourner(MedianeDesMedianes(L1))
    Fin Si
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste d'entiers L et retourne un nombre dont la position relative dans L n'est pas trop éloignée de la mediane.
- **Terminaison :** À chaque appel de la fonction, la longueur de la liste considérée est divisée par 5, elle finira donc bien par être inférieure à 5.
- **Correction :** On découpe la liste considérée en tronçons de longueur 5 sur lesquels on calcule la médiane. On recommence alors l'opération sur la liste de médianes obtenue. On peut montrer que l'on garantit ainsi l'obtention d'un pivot permettant au pire de partitionner la liste en deux parties contenant au moins 30 % de la liste.

Le programme en Python

```
def MedianeLocale(L,a,b):
    return TriInsertion(L[a:b+1])[(b-a)//2]

def MedianeDesMedianes(L):
    if len(L)<5:
        return MedianeLocale(L,0,len(L)-1)
    L1=[]
    for i in range(0,len(L)//5):
        L1=L1+[MedianeLocale(L,5*i,5*i+4)]
    return MedianeDesMedianes(L1)
```

On peut maintenant écrire l'algorithme de recherche d'un élément de rang fixé. On utilisera pour cela les fonctions MedianeDesMedianes et Partition2.

L'algorithme

```
Fonction ElementRangk(L,a,b,k)
Donnees
L, : liste d'entiers,
a, b, k, i, m : entiers
Debut
    m <-- MedianeDesMedianes(L[a:b+1])
    i <-- 0
    Tant Que L[i] != m
    Faire
        i <-- i + 1
    Fin Faire
    p <-- Partition2(L, i ,a, b,)
    Si k == p
    Alors
        Retourner(L[k])
    Fin SI
    Si k < p
    Alors
        Retourner(ElementRangk(L,a,p-1,k))
    Sinon
        Retourner(ElementRangk(L,p+1,b,k))
    Fin Si
Fin
```

Analyse de cet algorithme :

- **Spécification** : Cette fonction prend en entrée une liste d'entiers L, des indices a, b et k et retourne l'élément de rang k dans L.
- **Terminaison** : Cette fonction est récursive, et à chaque application elle se relance avec de nouveaux indices a et b tels que la nouvelle valeur de $b-a$ soit strictement inférieure à l'ancienne. Comme $b-a$ est entier, l'algorithme termine donc bien.

■ **Correction :** La définition de la fonction Partition2 assure que l'élément de rang k est toujours entre les a et b utilisés dans les appels à la fonction, ce qui garantit la correction de l'algorithme.

■ **Complexité :** C'est ici délicat. On admettra que la complexité obtenue est linéaire en la longueur de la liste L.

Tri Fusion

Ce tri est un également un tri récursif, comme le tri rapide. Il utilise également une étape intermédiaire, appelée cette fois-ci **fusion**, que nous détaillons en premier. Cette étape a pour but, étant données deux listes triées, de construire la liste triée formées par les éléments des deux listes.

Pour faire cela, on utilise deux indices parcourant chaque liste, et suivant la comparaison des éléments correspondant on ajoute à une nouvelle liste le plus petit des deux. On augmente alors de un l'indice dans la liste dans laquelle on a choisi l'élément. En répétant cette opération, on finit par vider une des deux listes. Il ne reste plus qu'à ajouter les éléments restant à parcourir dans l'autre liste.

L'algorithme

```
Fonction Fusion(L1,L2)
Donnees
L, L1, L2 : listes d'entiers,
i, j : entiers
Debut
    L <-- [ ]
    i <-- 0
    j <-- 0
    Tant Que i < len(L1) et j < len(L2)
    Faire
        Si L1[i] < L2[j]
        Alors
            L <-- L + [L1[i]]
            i <-- i + 1
```

```
        Sinon
            L <-- L + [L2[j]]
            j <-- j + 1
        Fin SI
    Retourner(L+L1[i:]+L2[j:])
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée deux listes d'entiers L1 et L2 triées. Il retourne la liste triée obtenue en regroupant les éléments de L1 et L2 en une seule liste.

■ **Terminaison :** La suite formée par les valeurs successives de i+j à chaque itération de la boucle Tant Que est à valeurs entières et strictement croissante. Par conséquent, si la boucle ne s'arrête pas auparavant elle prendra la valeur 2min(len(L1),len(L2)). Or i et j étant positifs si i+j vaut 2min(len(L1),len(L2)) c'est que i est supérieur à len(L1) ou que j est supérieur ou égal à len(L2), donc que la boucle s'arrête si elle ne l'a pas déjà fait.

■ **Correction :** Un invariant de boucle est donné par : "Les éléments de L constituent une liste triée et sont tous inférieurs à ceux d'indices supérieurs à i dans L1 et supérieurs à j dans L2". Donc à la sortie de la boucle, comme L1[i :] ou L2[j :] est vide, il suffit de rajouter les deux pour que L contienne tous les éléments de L1 et de L2 et soit triée.

■ **Complexité :** On note $n1$ et $n2$ les longueurs des listes L1 et L2. On a au pire, c'est-à-dire dans le cas où on effectue $n1 + n2 - 1$ passages dans la boucle $4+5(n2+n1-1)$ OPEL dans cet algorithme.

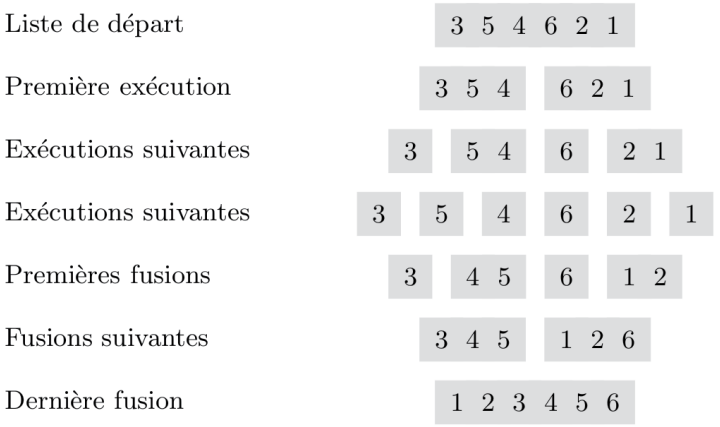
Le programme en Python

```
def Fusion(L1,L2):
    L=[]
    i = 0
    j = 0
    while i < len(L1) and j < len(L2):
        if L1[i] < L2[j]:
```

```
            L = L+[L1[i]]
            i = i+1
        else :
            L = L+[L2[j]]
            j=j+1
    return L+L1[i:]+L2[j:]
```

On peut maintenant décrire l'algorithme du tri fusion. L'idée est de dire que l'on peut trier une liste en triant la première moitié de la liste ainsi que la deuxième puis de fusionner les deux listes triées obtenues. Comme dans le cas du tri rapide, le cas d'arrêt de cet algorithme est le cas d'une liste de longueur inférieure ou égale à 1, où il n'y a rien à faire.

On peut illustrer cet algorithme en donnant le diagramme suivant :



L'algorithme

```
Fonction TriFusion(L)
Donnees
L,L1,L2 : liste d'entiers,
p, k : entiers
Debut
    Si len(L)<=1
```

```
    Alors
        Retourner(L)
    Sinon
        L1 = TriFusion(L[0,len(L)//2])
        L2 = TriFusion(L[len(L)//2:])
        Retourner Fusion(L1,L2)
    Fin Si
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée une liste d'entiers et retourne la liste triée contenant les mêmes éléments que L.

■ **Terminaison :** Cette fonction est récursive, et à chaque application elle se relance deux fois avec des listes de longueurs la moitié de celle de L. L'arbre d'exécution de cette fonction récursive est donc de hauteur au plus $\log_2(len(L))$.

■ **Correction :** On démontre la correction de cet algorithme à l'aide d'une récurrence forte sur la longueur de la liste.

■ **Complexité :** Si on note K_n la complexité de cet algorithme pour une liste de longueur n dans le pire des cas, on voit aisément que l'on a la relation de récurrence suivante pour tout entier naturel n :

$$K_{2n} = 2K_n + \mathcal{O}(n).$$

On en déduit que K_{2^n} est de l'ordre de $n\ln(n)$ puis par encadrement dans le cas général on obtient que la complexité totale est de l'ordre de $n\ln(n)$ dans le pire des cas.

Le programme en Python

```
def TriFusion(L):
    if len(L)<=1:
        return(L)
    L1 = TriFusion(L[0,len(L)//2])
    L2 = TriFusion(L[len(L)//2:])
    return Fusion(L1,L2)
```

Pour terminer sur le tri fusion on donne des fonction PYTHON permettant d'appliquer cet algorithme **en place**, en ne créant pas de listes intermédiaires en machine, mais en faisant toutes les modifications dans la liste d'origine. Contrairement au tri rapide on perd beaucoup en complexité en faisant ce changement, et on n'obtient qu'un algorithme quadratique en la longueur de la liste.

```
def FusionEnPlace(L,a,b,c,d):
    i=a
    j=c
    while i<=j-1 and j<=d:
        if L[i]<=L[j]:
            i+=1
        else:
            k=j
            while k>i:
                L[k-1],L[k]=L[k],L[k-1]
                k=k-1
            i=i+1
            j=j+1
    return L

def TriFusion(L,a,b):
    if b-a<=0:
        return L
    else:
        return FusionEnPlace(TriFusion(
        ↳ TriFusion(L,a,(a+b)//2),(a+b)//2+1,b),
        ↳ a,(a+b)//2,(a+b)//2+1,b)
```

Autres tris

On présente dans cette partie trois autres algorithmes de tri. S'ils ne sont pas explicitement au programme, ils restent incontournables dans le domaine.

Le premier est le **tri sélection**. Il utilise une fonction calculant l'indice

du minimum d'une liste (où le premier indice en lequel on le trouve). L'idée de l'algorithme est alors de construire la liste triée en ajoutant le minimum de ce qui reste à trier à chaque étape.

L'algorithme

```
Fonction IndiceMinimum(L,a,b)
Donnees
L : liste d'entiers,
a, b, m, i : entiers
Debut
    m <-- a
    Pour i variant de a+1 à b+1
    Faire
        Si L[i] < L[m]
        Alors
            m <-- i
        Fin Si
    Fin Faire
    Retourner(m)
Fin

Fonction TriSelection(L)
Donnees
L : liste d'entiers,
i, m : entier
Debut
    Pour i de 0 \'a len(L)
    Faire
        m <-- IndiceMinimum(L, i, len(L)-1)
        L[i], L[m] <-- L[m], L[i]
    Fin Faire
    Retourner(L)
Fin
```


Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste d'entiers L et retourne la liste triée contenant les mêmes éléments que L.

■ **Terminaison :** La boucle inconditionnelle termine après $len(L)$ itérations.

■ **Correction :** On a pour invariant de boucle : "la liste formée des éléments de L d'indice strictement inférieur à i est triée et tous ses éléments sont inférieurs ou égaux aux éléments de L d'indice supérieur ou égal à i".

■ **Complexité :** On note n la longueur de la liste L. L'appel de la fonction IndiceMinimum avec des indices a et b effectue au pire $1 + 2(b - a)$ OPEL. Dans TriSelection on appelle cette fonction avec les indices a allant de 0 à len(L)-1, ce qui donne au pire un total de

$$\sum_{i=0}^{n-1} 4 + 2(n - i) = 4n + n(n + 1) = n^2 + 5n$$

OPEL. Cet algorithme est donc de complexité quadratique en n.

Le programme en Python

```
def IndiceMinimum(L,a,b):
    m = a
    for i in range(a+1, b+1):
        if L[i] < L[m]:
            m = i
    return m

def TriSelection(L):
    for i in range(0,len(L)):
        m = IndiceMinimum(L, i, len(L)-1)
        L[i], L[m] = L[m], L[i]
    return(L)
```

On présente ensuite le **tri bulles**. L'idée est de balayer la liste en échangeant deux éléments côte-à-côte si ils ne sont pas bien ordonnés. On répète les balayages jusqu'à ne plus effectuer d'échange, ce qui signifie que la liste est triée.

L'algorithme

```
Fonction TriBulles(L)
Donnees
L : liste d'entiers,
b : booléen
i,m : entier
Debut
    b <-- Vrai
    m <-- len(L)-1
    Tant Que b
    Faire
        b <-- Faux
        Pour i variant de 0 à m
        Faire
            Si L[i] > L[i+1]
            Alors
                L[i], L[i+1] <-- L[i+1], L[i]
                b <-- Vrai
                m <-- i+1
             Fin Si
        Fin Faire
    Fin Faire
    Retourner(L)
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste d'entiers L et retourne la liste triée contenant les mêmes éléments que L.
- **Terminaison :** On peut montrer que tant que la liste n'est pas triée la suite formée par les valeurs successives de la variables m à chaque itération de la boucle Tant Que est strictement décroissante et à valeurs entières. Si m prend la valeur 1 alors la liste est triée car le dernier échange a eu lieu entre les indices 1 et 2. On en conclut que la liste devient bien triée au bout d'au plus len(L) itérations, et que la boucle Tant Que termine donc.
- **Correction :** On a l'invariant de boucle suivant pour la boucle Tant Que : "en début de boucle, les éléments de L d'indice supérieur

ou égal à la valeur de m forment une liste triée et sont tous supérieurs ou égaux aux éléments d'indices strictement inférieurs à m"

■ **Complexité :** On note n la longueur de L. Dans le pire des cas, la liste L est triée dans le sens inverse. L'algorithme effectue alors n passages dans la boucle Tant Que, m prend successivement les valeurs de $n-1$ à 1, et hormis lors du dernier passage, le test du Si est toujours vérifié. On en déduit que l'on effectue :

$$3 + n + \sum_{m=1}^{n-1} 6m + n = 3 + 2n + 3n(n-1) = 3n^2 - n + 3$$

OPEL au total. Cet algorithme est donc de complexité quadratique en n.

Le programme en Python

```
def TriBulles(L):
    b = True
    m = len(L)-1
    while b:
        b = False
        for i in range(0,m):
            if L[i] > L[i+1]:
                L[i], L[i+1] = L[i+1], L[i]
                b = True
                m = i+1
    return L
```

On termine enfin par un algorithme de complexité linéaire en la longueur de la liste, mais nécessitant de connaître un majorant des éléments de la liste, et que celle-ci ne contienne que des entiers naturels. L'idée est alors simplement de dénombrer les apparitions de chaque entier pouvant être présent dans la liste, puis de reconstruire la liste triée en les énumérant.

L'algorithme

```
Fonction TriEnumeration(L,M)
Donnees
L, L1, L2 : listes d'entiers,
i, j : entiers
Debut
    L1 <-- [0 pour i variant de 0 à M+1]
    Pour i variant de 0 à len(L)
    Faire
        L1[L[i]] <-- L1[L[i]]+1
    Fin Faire
    L2 <-- [ ]
    Pour i variant de 0 à M+1
    Faire
        L2 <-- L2 + [i pour j de 0 à L1[i]]
    Fin Faire
    Retourner(L2)
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste d'entiers naturels L et un entier naturel M majorant les éléments de L et retourne la liste triée contenant les mêmes éléments que L.
- **Terminaison :** On a ici deux boucles inconditionnelles de $len(L)$ et $M+1$ itérations respectivement.
- **Correction :** La première boucle terminée, chaque entrée de L1 contient le nombre d'apparitions de l'indice lui correspondant dans L. Il suffit donc bien de construire $L2$ comme dans la deuxième boucle Pour pour obtenir la liste triée des éléments de L.
- **Complexité :** On note n la longueur de L. On effectue $M+1+2n+1+n$ OPEL au total. À M fixé, l'algorithme est donc linéaire en n.

Le programme en Python

```
def TriEnumeration(L,M):
    L1 = [0 for i in range(0, M+1)]
    for i in range(0, len(L)):
        L1[L[i]] = L1[L[i]] + 1
    L2 = []
    for i in range(0, M+1):
        L2 = L2 + [i for j in range(0,L1[i])]
    return L2
```

Comparatifs des différents tris

Pour clore ce chapitre, on donne une table de comparaison des algorithmes rencontrés ici, et en particulier leur ordre de complexité dans le pire et dans le meilleur des cas ainsi que leur ordre de complexité en moyenne. On a laissé de côté le tri par énumération qui ne marche pas dans un cas général.

On note à nouveau n la longueur de la liste considérée.

	Complexité dans le meilleur des cas	Complexité dans le pire des cas	Complexité en moyenne
tri insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
tri rapide	$\mathcal{O}(n\ln(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(n\ln(n))$
tri fusion	$\mathcal{O}(n\ln(n))$	$\mathcal{O}(n\ln(n))$	$\mathcal{O}(n\ln(n))$
tri sélection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
tri bulles	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$

Boîte à outils

Lorsque L est une liste de nombres, on peut directement obtenir la liste triée en **Python** :

`sorted(L)`	trie la liste L

12. Traitement numérique des images

Représentation des images

En machine, les images sont représentées par des matrices de nombres ou de listes de nombres, chacune des entrées de la matrice codant un pixel. Quand l'image est en niveaux de gris, les entrées sont des entiers compris entre 0 et 255 (pouvant donc être écrits sur 8 bits), le 0 correspondant à un pixel noir et le 255 à un pixel blanc. Quand elle est en couleur, l'entrée est un triplet [R,G,B] d'entiers entre 0 et 255 codant chacun l'intensité du pixel en rouge, vert et bleu respectivement.

Pour récupérer la matrice correspondant à une image ainsi que créer une image à partir d'une matrice, on utilisera les modules **Numpy** (pour les matrices) et PIL, et plus particulièrement le sous-module Image de PIL.

Ainsi, une fois ces modules importés les instructions :

```
im=Image.open('image.bmp')
M=np.array(im)
```

permettent de récupérer dans M la matrice correspondant à l'image du fichier image.bmp et les instructions

```
im = Image.fromarray(M)
im.save('image.bmp')
```

permettent de sauvegarder l'image correspondant à la matrice M dans un fichier image.bmp. À noter que les nombres apparaissant dans M doivent être codés en uint8, ce qui s'assure en créant la matrice avec np.array(,np.uint8).

Terminons en précisant qu'une image im dans PYTHON peut être affichée à l'écran avec l'instruction

```
im.show()
```

Premières manipulations

On donne ici quelques exemples de manipulations d'images. Dans l'ordre, on écrit des fonctions permettant d'appliquer une symétrie horizontale, une symétrie verticale, une rotation d'un quart de tour. On prendra en entrée la matrice M correspondant à l'image.

```
def SymetrieHorizontale(M):
    m = len(M)
    n = len(M[0])
    M2 = np.array([[[0,0,0] for i in range(0,n)]
    ↳ for m in range(0,m)],np.uint8)
    for i in range(0,m):
        for j in range(0,n):
            M2[i,j] = M[m-1-i,j]
    return(M2)
```

```
def SymetrieVerticale(M):
    m = len(M)
    n = len(M[0])
    M2 = np.array([[[0,0,0] for i in range(0,n)]
    ↳ for m in range(0,m)],np.uint8)
    for i in range(0,m):
        for j in range(0,n):
            M2[i,j] = M[i,n-1-j]
    return(M2)
```

```
def RotationQuartDeTour(M):
    m = len(M)
    n = len(M[0])
    M2 = np.array([[[0,0,0] for i in range(0,n)]
    ↳ for m in range(0,n)],np.uint8)
    for i in range(0,n):
        for j in range(0,m):
```

```
                M2[i,j] = M[j,n-1-i]
    return(M2)
```

On peut également construire une fonction redonnant une image en niveaux de gris correspondant à une image prise en couleurs. On prendra en entrée a et b et c tels qu'un pixel codé par $[R, G, B]$ soit remplacé par

$$\left\lfloor \frac{1}{a+b+c}(aR + bG + cB) \right\rfloor$$

en niveaux de gris. En pratique, le choix a = b = c = 1 ne donne pas le meilleur rendu. On pourra préférer $a = 0.2126$, $b = 0.7152$ et $c = 0.0722$ (norme Rec. 709).

```
def NiveauxDeGris(M,a,b,c):
    m = len(M)
    n = len(M[0])
    M2 = np.array([[0 for j in range(0,n)]
    ↳ for i in range(0,m)],np.uint8)
    for i in range(0,m):
        for j in range(0,n):
            M2[i,j] = np.uint8((a*M[i,j][0]+
            ↳ b*M[i,j][1]+c*M[i,j][2])/(a+b+c))
    return(M2)
```

Recherche naïve de contours

Pour rechercher les contours existant dans une image, on peut utiliser une première approche simple en marquant les points en lesquels les changements des valeurs codant les pixels sont importants. Pour cela, on calcule la somme des valeurs absolues des différences des entiers de $p_{i,j}$ avec tous les pixels l'environnant. On marquera alors le pixel en noir si le résultat dépasse un seuil que l'on s'est fixé à l'avance.

Pour simplifier le parcours, on ne considèrera que les pixels intérieurs à l'image considérée.

Le programme en Python

```
def Contours(M,S):
    m=len(M)
    n=len(M[0])
    M2=np.zeros((m,n),np.uint8)
    for i in range(1,m-1):
        print(i)
        for j in range(1,n-1):
            G=0
            for a in range(0,3):
                for b in range(0,3):
                    for k in range(0,3):
                        G=G+abs(M[i,j][k]
                        ↳ -M[i-1+a,j-1+b][k])
            if G>S:
                M2[i,j]=255
    return(M2)
```

Traitement par convolution

Le procédé de traitement d'image proposé ici permet d'effectuer de nombreuses modifications différentes sur des images, tout en adoptant une démarche générale.

Le principe est d'utiliser un **noyau de convolution**, c'est-à-dire une matrice que nous prendrons dans notre cas comme carrée d'ordre 3, donc de la forme :

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{pmatrix}$$

et de remplacer chaque pixel $p_{i,j}$ par :

$$\begin{array}{lccc} & a_1p_{i-1,j-1} & +a_2p_{i-1,j} & +a_3p_{i_1,j+1} \\ + & a_4p_{i,j-1} & +a_5p_{i,j} & +a_6p_{i,j+1} \\ + & a_7p_{i+1,j-1} & +a_8p_{i+1,j} & +a_9p_{i+1,j+1} \end{array}$$

Suivant le noyau de convolution choisi, l'effet obtenu sur l'image peut être très varié. Commençons par écrire des fonctions PYTHON appliquant ce procédé. On utilisera une fonction intermédiaire modifiant simplement un pixel étant donné la matrice de convolution N. On prendra N à coefficients entiers et on utilisera un éventuel diviseur d après le calcul effectué avec N de façon à faire un maximum de calculs avec des entiers.

Il faut se fixer une convention pour le travail sur le bord de l'image. On utilisera ici les pixels du bord opposé pour le calcul, on pourrait aussi simplement enlever les pixels du bord.

Le programme en Python

```
def ConvolutionLocale(i,j,M,N,d):
    ↳ for a in range(0,len(M))],np.uint8)
    S=np.array([0,0,0])
    for a in range(0,3):
        for b in range(0,3):
            S=S+(M[i-1+a,j-1+b]*N[a,b])/d
    return [max(min(S[0],255),0),
    ↳ max(min(S[1],255),0),max(min(S[2],255),0)]
```

```
def Convolution(M,N,d):
    m=len(M)
    n=len(M[0])
    M=np.array([M[-1]]
    ↳ +[M[k] for k in range(0,len(M))])
    M=np.array([[M[k][-1]]
    ↳ +list(M[k]) for k in range(0,len(M))])
    M=np.array([M[k] for k in range(0,len(M))]
    ↳+[M[0]])
    M=np.array([list(M[k])+[M[k][0]]
    ↳ for k in range(0,len(M))])
    M2 = np.array([[[0,0,0] for i in range(0,n)]
    ↳ for j in range(0,m)])
    for i in range(1,m+1):
        print(i)
        for j in range(1, n+1):
```

```
                M2[i-1,j-1] =
                ↳ ConvolutionLocale(i,j,M,N,d)
    return M2
```

On peut alors, suivant l'effet voulu, prendre tel ou tel noyau. On donne ici quelques exemples classiques :

- **Floutage ou lissage**

Pour flouter une image ou réduire un éventuel bruit on peut prendre pour noyau la matrice :

$$\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

- **Détection de contours**

Pour faire ressortir les contours d'une image on peut prendre pour noyau la matrice :

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

- **Filtrage Gaussien**

Pour essayer de faire disparaître un bruit naturel d'une image, on peut prendre le noyau gaussien :

$$\frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

« Les ordinateurs sont inutiles, ils ne nous donnent que des réponses. »

Pablo Picasso, artiste peintre espagnol.

13. Codage et transmission

Algorithmes de cryptage élémentaires

Pour transmettre un message en évitant que quelqu'un l'interceptant ne puisse en tirer profit, on utilise des méthodes de cryptage. Dans l'idéal, seul quelqu'un connaissant la méthode utilisée et une clé de décryptage pourra récupérer le message d'origine à partir du message codé. De nombreuses méthodes ont été développées dans ce but, certaines très savantes. Nous nous contentons ici d'en présenter une, relativement élémentaire, le **chiffrement de Vigenère**. Dans la suite, on utilisera le rang dans l'alphabet au sens de PYTHON (A sera de rang 0).

Le principe est le suivant. On souhaite transmettre une chaîne de caractères et pour la crypter nous allons définir une règle de substitution associant à chaque caractère, suivant sa position dans le message, un autre caractère. Pour simplifier nous nous contenterons de prendre des messages écrits en majuscules et nous ne traiterons que les lettres du message.

Pour fixer la règle de substitution, on fixe une clé qui sera un mot. On note C ce mot et N sa longueur. On note R_i le rang dans l'alphabet de la j-ième lettre de C et r_i le rang dans l'alphabet de la lettre en i-ième position dans le message. Alors la lettre écrite dans le message chiffré en i-ième position sera la lettre de rang le reste de la division par 26 de $r_i + R_j$ si j est le reste de la division de i par N.

Un exemple sera plus parlant ! Si on prend C = "CAUCHY" et si on souhaite coder "SCHWARZ" alors :

- Le S sera codé par la lettre de rang 18 (le rang de S dans l'alphabet) plus 2 (le rang de C dans l'alphabet), soit 20.

- Le C sera codé par la lettre de rang 2 (le rang de C dans l'alphabet) plus 0 (le rang de A dans l'alphabet), soit 2.
- Le H sera codé par la lettre de rang 7 (le rang de H dans l'alphabet) plus 20 (le rang de U dans l'alphabet), soit 1 (on prend le reste de la division de 27 par 26).
- Le W sera codé par la lettre de rang 22 (le rang de W dans l'alphabet) plus 2 (le rang de C dans l'alphabet), soit 24.
- Le A sera codé par la lettre de rang 0 (le rang de A dans l'alphabet) plus 7 (le rang de H dans l'alphabet), soit 7.
- Le R sera codé par la lettre de rang 17 (le rang de R dans l'alphabet) plus 24 (le rang de Y dans l'alphabet), soit 15 (on prend le reste de la division de 41 par 26)
- Le Z sera codé par la lettre de rang 25 (le rang de Z dans l'alphabet) plus 2 (le rang de C dans l'alphabet) soit 1 (on prend le reste de la division de 28 par 26).

Le message chiffré sera donc finalement "UCBYHPB".

On donne l'algorithme de chiffrement et les fonctions PYTHON pour le chiffrement et le déchiffrement. Le déchiffrement se faisant simplement en soustrayant le rang de la lettre de la clé au lieu de l'ajouter.

L'algorithme

```
Fonction ChiffrementVigenere(M,C)
Donnees
M, C, M1 : chaînes de caractères,
i, j, N : entiers
Debut
    N <-- len(C)
    M1 <-- ""
    Pour i variant de 0 à len(M)
    Faire
        M1 <-- M1 + Caractere(
        ↳ (Rang(M[i])+Rang(C[i%N]))%26)
    Fin Faire
```

```
    Retourner M1
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée deux chaînes de caractères M et C et retourne le chiffrement de Vigenère de M en utilisant comme clé C.
- **Terminaison :** La boucle inconditionnelle se termine après len(M) itérations.
- **Complexité :** On obtient une complexité linéaire en la longueur de M.

Le programme en Python

On utilise ici les fonctions ord() et chr() de PYTHON. La fonction ord() associe un entier à un caractère (avec 65 pour le caractère "A") et la fonction chr() est le procédé réciproque de ord().

```
def ChiffrementVigenere(M,C):
    N = len(C)
    M1 = ""
    for i in range(0, len(M)):
        M1 = M1 + chr((((ord(M[i]) +
                        ↳ ord(C[i%N]))-130)%26)+65)
    return M1
```

Sommes de contrôle, codes correcteurs

Toujours dans le cadre de la transmission d'information, on considère maintenant un problème un peut différent. Désormais, on souhaite déterminer si le message transmis a été altéré et, dans l'idéal, on souhaite pouvoir retrouver le message initial si des erreurs sont apparues.

De façon à pouvoir repérer l'apparition d'une erreur, on va agrandir un peu le message en lui ajoutant une information sur sa structure. Si

le message est modifié on pourra repérer une incohérence entre cette information et la nouvelle structure obtenue.

On se limitera à un exemple très simple de somme de contrôle : le bit de parité.

On transmettra ici un message en binaire sous la forme d'une chaîne de caractères. Si le message est $b_0...b_n$ on appelle bit de parité de ce message le bit $b_{n+1} \in \{0,1\}$ tel que la somme $S_n = \sum_{i=0}^{n+1} b_i$ soit paire. Le nouveau message est alors $b_0...b_{n+1}$ et on pourra détecter une erreur dans le message reçu si la somme S_n est impaire (on détecte plus généralement s'il y a un nombre impair d'erreurs).

Le programme en Python

On donne ici des fonctions donnant le message obtenu en ajoutant le bit de parité ainsi qu'une fonction testant la somme de contrôle.

```
def BitDeParite(M):
    S=0
    for i in range(0,len(M)):
        S=S+int(M[i])
    return M+str(S%2)

def ControleParite(M):
    S=0
    for i in range(0,len(M)):
        S=S+int(M[i])
    if S%2 == 0:
        return True
    else:
        return False
```

En utilisant ces fonctions, nous allons maintenant décrire une méthode permettant non seulement de détecter mais également de corriger une (et une seule !) erreur. Nous décrivons ici une implémentation de ce que l'on appelle le **code de Hamming(4,7)**. Les codes de Hamming forment une famille de codes correcteurs d'erreur linéaires que l'on

peut construire et décrire de façon plus sophistiquée que ce que nous faisons dans cet exemple simple.

Nous considérons la transmission d'un message constitué de quatre bits $b_1b_2b_3b_4$. Le message qui sera transmis sera constitué de sept bits. Nous allons en effet rajouter des bits de parité p_1, p_2 et p_3 correspondant respectivement aux messages $b_1b_2b_4$, $b_1b_3b_4$ et $b_2b_3b_4$. Une erreur sur un des bits sera indiquée par le fait qu'un des bit de parité au moins ne correspondra pas. De plus, la configuration des bits de parité erronés précise la position de l'erreur ! On envoie en pratique le message $p_1p_2b_1p_3b_2b_3b_4$ car nous verrons que cette configuration permet d'automatiser le décodage.

Prenons un exemple : si nous recevons le message '1111100' alors le bit de parité p_1 est faux, le bit de parité $p2$ est correct et le bit de parité p_3 est correct. Dans ce cas la seule possibilité est que l'erreur (qui doit être unique !) a eu lieu sur p_1 !

Pour décoder efficacement le message on peut recourir à une matrice de contrôle pour ce code. On définit la matrice H par :

$$\boxed{H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.}$$

On appelle **syndrôme** le produit de cette matrice par la colonne correspondant au message reçu. Ainsi, toujours avec l'exemple précédent, le syndrôme sera :

$$s = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

On peut remarquer que H a été construite de manière à ce que ce produit donne comme résultat la colonne $\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}$.

Le positionnement des bits de parité dans le message fait que le syndrôme, interprété en binaire, donne le numéro du bit sur lequel une erreur a été commise. Ainsi ici, $(001)_2$ code 1 en binaire et l'erreur a bien eu lieu sur le premier bit transmis, p_1.

On peut maintenant écrire des fonctions de codage et décodage pour le code de Hamming(7,4).

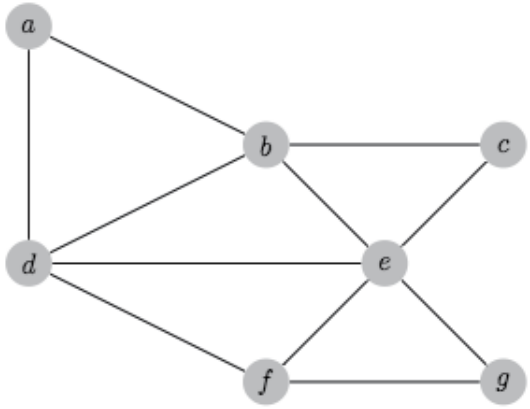
```
def codage(M):
        p1 = BitDeParite(M[0]+M[1]+M[3])[-1]
        p2 = BitDeParite(M[0]+M[2]+M[3])[-1]
        p3 = BitDeParite(M[1]+M[2]+M[3])[-1]
        return p1+p2+M[0]+p3+M[1:]

def decodage(M):
    H = np.array([[0,0,0,1,1,1,1],
    ↳ [0,1,1,1,0,0,1,1],[1,0,1,0,1,0,1]])
    X = np.array([[int(M[i])] for i in range(7)])
    Y = np.dot(H,X)
    M1 = ""
    for i in range(0,3):
        M1 = M1 + str(Y[i,0] % 2)
    a = int(M1,2)
    if a == 0:
        return M
    M2=""
    for i in range(0,7):
        if a == i + 1:
            M2 = M2 + str((int(M[i])+1)%2)
        else:
            M2 = M2 + M[i]
    return M2
```

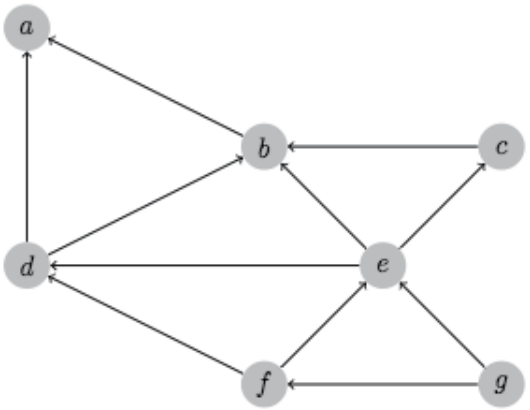
14. Graphes

Représentation d'un graphe

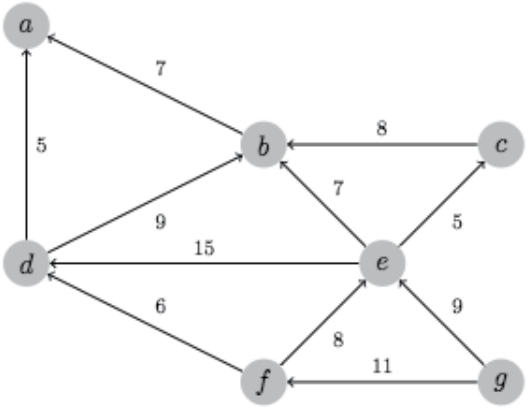
Un **graphe** est la donnée d'un couple $(\mathcal{S}, \mathcal{A})$ où $\mathcal{S}$ est un ensemble dont les éléments sont appelés **sommets** du graphe et où $\mathcal{A}$ est un ensemble de paires d'élements de $\mathcal{A}$ dont les éléments sont appelés les **arêtes** du graphe. On peut représenter un tel objet comme sur la figure suivante :



On appelle **graphe orienté** la donnée d'un couple $(\mathcal{S}, \mathcal{A})$ où $\mathcal{S}$ est un ensemble dont les éléments sont appelés **sommets** du graphe et où $\mathcal{A}$ est un ensemble de couples d'éléments de $\mathcal{A}$ dont les éléments sont appelés les **arcs** du graphe. On obtient alors une figure comme :



On dit qu'un graphe, orienté ou non, est **pondéré** si à chaque arête ou arc du graphe est associé un nombre réel (généralement positif) appelé **poids** de cet arc. Ce qui peut se représenter par :



Pour pouvoir travailler avec de tels objets, on associe à un graphe une matrice, appelée **matrice d'adjacence**. À chaque sommet du graphe on associe un indice i. Dans la matrice, on indique la présence ou non d'un arc ou d'une arête entre deux sommets i et j en affectant à a_{ij} la valeur 1 ou 0. Si le graphe est pondéré, on affecte plutôt à a_{ij} le poids de l'arc partant de i et arrivant à j. Si un tel arc n'existe pas, on pourra affecter ∞ à a_{ij}.

Dans le cas des graphes des trois figures précédentes, on obtient les matrices suivantes :

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 7 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty & \infty \\ 5 & 9 & \infty & \infty & \infty & \infty & \infty \\ \infty & 7 & 5 & 15 & \infty & \infty & \infty \\ \infty & \infty & \infty & 6 & 8 & \infty & \infty \\ \infty & \infty & \infty & \infty & 9 & 11 & \infty \end{pmatrix}$$

Algorithme de plus court chemin

Un problème classique, étant donné un graphe, est de chercher à déterminer le plus court chemin joignant deux sommets donnés. Dans le cas d'un graphe pondéré, la longueur du chemin consistera en la somme des poids des arcs ou arêtes parcourus. On présente ici l'**algorithme de Dijkstra**, permettant de résoudre ce problème.

On décrira l'algorithme dans le cas d'un graphe orienté pondéré. Expliquons rapidement le principe de l'algorithme. Supposons que l'on cherche le plus court chemin joignant g à a. On va construire une table dans laquelle à chaque sommet seront associés la longueur

du plus court chemin trouvé jusqu'ici menant à ce sommet depuis g et le dernier sommet emprûnté pour ce chemin.

• On initialise la table à $(0, g)$ pour g et (∞, x) pour le sommet x. Le sommet g est donc considéré comme traité.

• On détermine le voisin non traité d'un des sommets traités le plus proche de g en comparant les sommes pour chaque sommet non traité s_1 et chaque sommet traité s_2 de la distance entre s_1 et s_2 et de la distance stockée dans la table pour s_2.

• On associe au sommet s_1 le plus proche trouvé dans l'étape précédente la somme minimale obtenue et le sommet $s2$ pour lequel elle a été obtenue.

• On détermine pour chaque sommet s_i déjà traité avant ce nouveau sommet s_1 si la somme de la longueur de l'arc le joignant à s_1 et de la distance stockée dans la table pour s_1 est inférieure à la distance stockée précédemment pour s_i. Si c'est le cas on écrit la somme dans la table et y associe le sommet s_1.

• On recommence jusqu'à obtenir comme sommet le plus proche le sommet cible, c'est-à-dire a ici, ou jusqu'à avoir traité tous les sommets du graphe.

Pour écrire cet algorithme on utilisera ici plusieurs sous-fonctions.

• La fonction Dist déterminera la distance au sommet initial d'un sommet s donné, obtenue en sommant les distances obtenue précédemment et la distance au sommet s correspondant.

• La fonction PlusProche utilisera la fonction Dist pour déterminer le sommet pour lequel on obtiendra la plus petite distance cumulée.

• La fonction Enleve permettra d'enlever un sommet de la liste des sommets restant à traiter.

• La fonction Chemin donnera le chemin obtenu une fois la table terminée.

• La fonction Dijkstra appliquera l'algorithme.

On ne détaillera pas les algorithmes pour Enleve et Chemin, qui sont très simples. On peut tout du moins remarquer qu'ils sont linéaires en la taille de leurs entrées.

L'algorithme

```
Fonction Dist(A,Table,s)
Donnees
A, Table : listes de listes
s, i : entier
Debut
    d <-- [infini, 0]
    Pour i variant de 0 à len(A)
    Faire
        Si Table[i][1] + A[i][s] < d[0]
        Alors
            d <-- [Table[i][1] + A[i][s], i]
        Fin Si
    Fin Faire
    Retourner(d)
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée A une liste de listes décrivant la matrice d'adjacence du graphe, Table, une liste de listes contenant la table courante pour l'algorithme de Dijkstra et s l'indice du sommet considéré. Elle retourne la somme de distance minimale obtenue pour le sommet s ainsi que l'indice du sommet pour lequel elle l'a été.
- **Terminaison :** La boucle inconditionnelle se termine après len(A) itérations, c'est-à-dire en autant d'itérations que de sommets du graphe.
- **Complexité :** On obtient une complexité linéaire par rapport au nombre de sommets du graphe.

L'algorithme

```
Fonction PlusProche(A,Table,L)
Donnees
A, Table : listes de listes
L : liste d'entiers
i : entier
mini : triplet de nombres
```

```
Debut
    mini <-- L[0], infini, L[0]
    Pour i variant de 0 à len(L)
    Faire
        [d, p] <-- Dist(A,Table, L[i])
        Si d < mini[1]
        Alors
            mini <-- [L[i], d, p]
        FIn Si
    Fin Faire
    Retourner(mini)
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée A une liste de listes décrivant la matrice d'adjacence du graphe, Table, une liste de listes contenant la table courante pour l'algorithme de Dijkstra, L une liste d'entiers donnant la liste des sommets restant à traiter. Elle donne comme résultat un triplet donnant le voisin pour lequel le nouveau sommet a été trouvé, la somme de distance correspondant, et le nouveau sommet.
- **Terminaison :** La boucle inconditionnelle termine en len(L) itérations, c'est-à-dire en autant d'itérations que de sommets du graphe restant à traiter.
- **Complexité :** En utilisant la complexité obtenue pour Dist, on obtient une complexité au pire linéaire en le produit du nombre de sommets du graphe et le nombre de sommets du graphe restant à traiter.

L'algorithme

```
Fonction Dijkstra(A,a,b)
Donnees
A, Table : listes de listes,
a, b, i, j : entiers
L : liste d'entiers,
mini : triplet de nombres
```

```
Debut
    n <-- len(A)
    Table <-- [[a, infini] pour i de 0 à n]
    Table[a] <-- [a, 0]
    L <--[k pour k de 0 à a]+[k pour k de a+1 à n]
    Pour i variant de 1 à n
    Faire
        mini <-- PlusProche(A, Table, L)
        Table(mini[0]] <-- [mini[2], mini[1]]
        Pour j variant de 0 à n
        Faire
            Si Table[mini[0]][1] + A[mini[0]][j]
            ↳ < Table[j][1]
            Alors
                Table[j][1] <-- Table[mini[0]][1]+
                ↳ A[mini[0]][j]
                Table[j][0] <-- mini[0]
            Fin Si
        Fin Faire
        Si mini[0] == b
        Alors
            Retourner(Chemin(Table,a,b))
        L <-- Enleve(L, mini[0])
    Fin Faire
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée A une liste de listes décrivant la matrice d'adjacence du graphe, ainsi que les indices a et b des sommets pris comme points de départ et d'arrivée du chemin cherché. Elle donne comme résultat la longueur du chemin le plus court reliant les deux sommets choisis ainsi que la liste des indices des sommets parcourus sur ce chemin.
- **Terminaison :** Les boucles inconditionnelles se terminent en n itérations, c'est-à-dire en autant d'itérations que de sommets du graphe.
- **Correction :** On a l'invariant de boucle : « Les distances stockées dans la table sont les distances minimales au sommet initial pour le

sous-graphe des sommets déjà traités ».

■ **Complexité :** On ne rentrera pas dans le détail, mais on obtient un algorithme polynomial en le nombre de sommets et le nombre d'arcs du graphe.

Le programme en Python

On utilise ici l'infini de PYTHON, donné par float("inf"). On écrira donc au préalable dans le programme inf = float("inf").

```
def Dist(A,Table,s):
    d=inf, 0
    for i in range(0,len(A)):
        if Table[i][1]+A[i][s] < d[0]:
            d = Table[i][1]+A[i][s], i
    return d
```

```
def PlusProche(A,Table,L):
    mini=L[0],inf,L[0]
    for i in L:
        d, p = Dist(A,Table,i)
        if d<mini[1]:
            mini = i, d, p
    return mini
```

```
def Enleve(L,i):
    L1=[]
    for j in range(0,len(L)):
        if L[j] != i:
            L1=L1+[L[j]]
    return L1
```

```
def Chemin(Table,a,b):
    L=[Table[b][0],b]
    while L[0] != a:
        L=[Table[L[0]][0]]+L
    return Table[b][1],L
```

```
def Dijkstra(A,a,b):
    n = len(A)
    Table = [[a,inf] for i in range(0,n)]
    Table[a] = [a,0]
    L = [k for k in range(0,a)]+
                    ↳ [k for k in range(a+1,n)]
    for i in range(1,n):
        mini = PlusProche(A,Table,L)
        Table[mini[0]]=[mini[2],mini[1]]
        for i in range(0,n):
            if Table[mini[0]][1]+A[mini[0]][i]
                                    ↳ < Table[i][1]:
                Table[i][1] = Table[mini[0]][0]+
                                    ↳ A[i][mini[0]]
                Table[i][0] = mini[0]
        if mini[0]==b:
            return Chemin(Table,a,b)
        L=Enleve(L,mini[0])
```

Le problème du voyageur de commerce

On présente un autre problème voisin de celui que nous avons traité dans le paragraphe précédent. Cette fois-ci il s'agit de trouver le plus court chemin dans le graphe permettant de passer par tous les sommets. Pour résoudre ce problème nous allons utiliser un algorithme de **colonie de fourmis**.

On note dans la suite n le nombre de sommets du graphe pour lequel on cherche à résoudre le problème. Comme l'algorithme que l'on utilisera nécessite de faire des calculs avec les entrées de la matrice d'adjacence, on ne prendra plus ∞ pour les arêtes n'existant pas mais un grand nombre (la somme de tous les poids du graphe convient, en général).

Le principe d'un tel algorithme est d'effectuer plusieurs simulations de mouvements d'un ensemble de mobiles le long du graphe (les

fourmis), ces mobiles marquant à chaque fois d'une manière ou d'une autre leur chemin dans le graphe, ce qui influencera le chemin choisi par les mobiles de la vague suivante. On créera pour cela une matrice carrée de même ordre que la matrice d'adjacence du graphe, dans laquelle on associera à chaque arête du graphe un nouveau poids que l'on appellera **phéromone**.

Pour être plus précis, on répétera les opérations suivantes :

- On place N fourmis au hasard sur les n sommets.
- À chaque fourmi on associe la liste des sommets par lesquels elle passe. Initialement elle ne contient donc que le sommet de départ de la fourmi.
- En tenant compte des phéromones, on fait effectuer $n-1$ déplacements à chaque fourmi, en s'assurant qu'elle ne passe pas deux fois par le même sommet (elle a donc parcouru tout le graphe en passant exactement une fois par chaque sommet).
- On augmente les phéromones sur chaque arête empruntée par la fourmi, d'une intensité d'autant plus faible que le parcours de la fourmi aura été long.

Une fois un certain nombre d'itérations effectuées, on garde le meilleur parcours réussi par les fourmis.

Il nous reste à quantifier les choses. On note A la matrice d'adjacence du graphe et B la matrice des phéromones. On initialisera les phéromones à 0.01 sur chaque arête.

La probabilité qu'une fourmi située en le sommet i emprunte une arête menant au sommet j sera donnée par l'expression :

$$p_{i,j} = \frac{b_{i,j}^{\alpha} a_{i,j}^{-\beta}}{\sum_k b_{i,k}^{\alpha} a_{i,k}^{-\beta}}$$

où la somme au dénominateur a lieu sur les indices des sommets non encore parcourus par la fourmi. On fixera la probabilité d'un sommet déjà traversé à 0.

À la fin d'un trajet, on actualisera la matrice B en calculant les nouvelles valeurs des $b_{i,j}$ données par :

$$b_{i,j} \leftarrow r.b_{i,j} + \sum_f \frac{1}{L_f}$$

où la somme a lieu sur l'ensemble des fourmis ayant parcouru l'arête menant de i à j et où L_f est la longueur du parcours de la fourmi f. La constante r est un réel positif au choix.

On obtient au final un algorithme dépendant de paramètres plus ou moins ad hoc, α, β et r. En pratique, $\alpha = \beta = 1$ et $r = 0,5$ donnent de bons résultats.

On découpe l'algorithme en plusieurs sous-fonctions :

L'algorithme

```
Fonction PositionsInitiales(n,N)
Donnees
n, N, i : entiers,
L : liste de liste d'entiers.
Debut
    L <-- [ ]
    Pour i variant de 0 à N
    Faire
        L <-- L + [[entier au hasard dans [[0,n-1]]]
    Fin Faire
    Retourner L
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée deux entiers : le nombre de sommets n et le nombre de fourmis de chaque vague N. Elle retourne la liste des positions initiales des fourmis, sous forme de liste de listes.
- **Complexité :** Cette fonction est de complexité linéaire en N.

Pour programmer le tirage au sort, on utilise la fonction **randint** du module random qui donne un entier tiré au hasard de façon uniforme entre les deux bornes données en entrée.

Le programme en Python

```
def PositionsInitiales(n,N):
    L = []
    for i in range(0,N):
        L = L + [[rd.randint(0,n-1)]]
    return L
```

L'algorithme

```
Fonction ListeProbabilités(i,LS,A,B,n,alpha,beta):
Donnees
i, j, k, n : entier,
LS : liste d'entiers,
A, B : matrices de flottants,
LP : liste de flottants.
alpha, beta : flottants
Debut
    LP <-- [0 pour j variant de 0 à n]
    S <-- 0
    Pour k variant de 0 à n
    Faire
        Si k n'est pas dans LS
        Alors
            S <-- S + B[i][k]**alpha
            ↳ * A[i][k]**(-beta)
        Fin Si
    Fin Faire
    Pour j variant de 0 à n
    Faire
        Si j n'est pas dans LS
        Alors
            LP[j] <-- B[i][j]**alpha
            ↳ * A[i][j]**alpha/S
        Fin Si
    Fin Faire
    Retourner LP
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée l'indice i du sommet sur lequel est la fourmi, la liste LS des sommets parcourus par la fourmi en cours, la matrice A d'adjacence du graphique et la matrice B des phéromones, le nombre n de sommets et les constantes alpha et beta correspondant à α et β. Elle retourne la liste des probabilités que la fourmi choisisse chaque sommet du graphe comme prochaine étape.

■ **Complexité :** Cette fonction est de complexité linéaire par rapport au produit n fois la longueur de LS.

Le programme en Python

```
def ListeProbabilites(i,LS,A,B,n,alpha,beta):
    LP = [0 for j in range(0,n)]
    S = 0
    for k in range(0, n):
        if k not in LS:
            S = S+B[i][k]**alpha*A[i][k]**(-beta)
    for j in range(0,n):
        if j not in LS:
            LP[j] = B[i][j]**alpha *A[i][j]**alpha
    return LP
```

L'algorithme

```
Fonction ChoixSommet(LP)
Donnees
LP : liste de flottants
S : flottant
i : entier
Debut
    a <-- nombre au hasard dans [0,1]
    S <-- LP[0]
    i <-- 0
    Tant Que S < a
    Faire
        i <-- i+1
```

```
        S <-- S + LP[i]
    Fin Faire
    Retourner i
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée la liste LP des probabilités d'atteindre chacun des sommets. Elle retourne le sommet choisi.
- **Terminaison :** Comme la somme des éléments de LP est égale à 1, S prendra bien une valeur supérieure ou égale à a.
- **Correction :** l'événement $(i = k)$ est égal à l'événement

$$(a \leq \sum_{j=0}^{i-1} LP[i]) \cap (a > \sum_{j=0}^{i-1} LP[i]).$$

Il a donc pour probabilité LP[i].

- **Complexité :** Cette fonction est de complexité linéaire en n.

Pour programmer le tirage au sort, on utilise la fonction rand du module random qui donne un entier tiré au hasard de façon uniforme sur $[0, 1]$.

Le programme en Python

```
def ChoixSommet(LP):
    a = rd.random()
    S = LP[0]
    i = 0
    while S < a:
        i = i+1
        S = S+LP[i]
    return i
```

L'algorithme

```
Fonction TourComplet(A,B,n,N,alpha,beta)
Donnees
A, B : matrices de flottants
LP : liste de flottants
S : flottant
n, N, i, j, f : entier
alpha, beta : flottants
Debut
        L <-- PositionsInitiales(n,N)
        Pour i variant de 0 à n-1
        Faire
            Pour f variant de 0 à N
            Faire
                LP <-- ListeProbabilités(L[f][-1],
                    ↳ L[f], A, B, n, alpha, beta)
                j <-- ChoixSommet(LP)
                L[f] <-- L[f] + [j]
            Fin Faire
        Fin Faire
        Retourner L
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée la matrice A d'adjacence du graphique et la matrice B des phéromones, le nombre n de sommets, le nombre N de fourmis et les constantes alpha et beta correspondant à α et β. Elle retourne la liste des probabilités que la fourmi choisisse chaque sommet du graphe comme prochaine étape.
- **Complexité :** Cette fonction est de complexité de l'ordre de $n^3.N$.

Le programme en Python

```
def TourComplet(A,B,n,N,alpha,beta):
    L = PositionsInitiales(n,N)
    for i in range(0,n-1):
        for f in range(0,N):
            LP = ListeProbabilites(L[f][-1], L[f],
```

```
                                    ↳ A, B, n, alpha, beta)
            j = ChoixSommet(LP)
            L[f] = L[f] + [j]
    return L
```

L'algorithme

```
Fonction ActualisePheromones(L,A,B,n,N,r)
Donnees
A, B : matrices de flottants
L : liste de flottants
n, N, i, j, f : entier
r, S : flottants
Debut
    Pour i variant de 0 à n
            Faire
                Pour j variant de 0 à n
                Faire
                    B[i][j] =<-- r * B[i][j]
    Pour f variant de 0 à N
    Faire
        S <-- 0
        Pour i variant de 0 à n-1
        Faire
            S <-- S + A[L[f][i]][L[f][i+1]]
        Fin Faire
        Pour i variant de 0 à n-1
        Faire
            B[L[f][i]][L[f][i+1]] <--
                    ↳ B[L[f][i]][L[f][i+1]] + 1/S
        Fin Faire
    Fin Faire
    Retourner B
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée la liste des listes des sommets parcourus par les fourmis, la matrice A d'adjacence du graphique et la matrice B des phéromones, le nombre n de sommets,

le nombre N de fourmis et la constante r. Elle retourne la matrice des phéromones mises à jour.

- **Complexité :** Cette fonction est de complexité de l'ordre de $n^2.N$.

Le programme en Python

```
def ActualisePheromones(L,A,B,n,N,r):
    for i in range(0,n):
        for j in range(0,n):
            B[i][j] = r*B[i][j]
    for f in range(0,N):
        S = 0
        for i in range(0,n-1):
            S = S + A[L[f][i]][L[f][i+1]]
        for i in range(0,n-1):
            B[L[f][i]][L[f][i+1]] =
                ↳ B[L[f][i]][L[f][i+1]] + 1/S
    return B
```

On peut enfin donner la fonction déterminant un chemin le plus court :

L'algorithme

```
Fonction CheminLePlusCourt(A,N,NV, r,alpha,beta)
Donnees
A, B : matrices de flottants
L, Ll : liste de flottants
n, N, i, j, f, NV: entier
r, alpha, beta, S, lmin : flottants
Debut
    n <-- len(A)
    B <-- [[0.01 pour i variant de 0 à n]
    ↳ pour j variant de 0 à n]
    lmin <-- infini
    Pour i variant de 0 à NV
    Faire
        L <-- TourComplet(A,B,n,N,alpha,beta)
        B <-- ActualisePheromones(L,A,B,n,N,r)
        Pour f de 0 à N
```

```
        Faire
            S <-- 0
            Pour i de 0 à n-1
            Faire
                S <-- S + A[L[f][i]][L[f][i+1]]
            Fin Faire
            Si S < lmin
            Alors
                lmin <-- S
                Ll <-- L[f]
            Fin Si
    Fin Faire
    Retourner lmin, Ll
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée la matrice A d'adjacence du graphique, le nombre N de fourmis de chaque vague, le nombre NV de vagues de fourmis, la constante r et les deux constantes alpha et beta correspondant à α et β. Elle retourne la longueur du plus court chemin trouvé faisant passer par tous les sommets ainsi que la liste ordonnée des sommets par lesquels passe ce chemin.
- **Complexité :** La complexité est ici de l'ordre de $NV.n^3.N$.

Le programme en Python

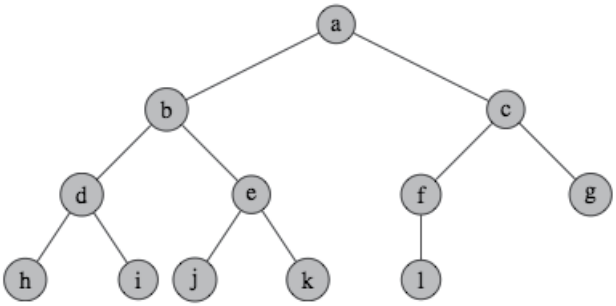
```
def CheminLePlusCourt(A,N,NV, r,alpha,beta):
    n = len(A)
    B = [[0.01 for i in range(0,n)]
    ↳ for j in range(0,n)]
    lmin = float('inf')
    for i in range(0,NV):
        L = TourComplet(A,B,n,N,alpha,beta)
        B = ActualisePheromones(L,A,B,n,N,r)
        for f in range(0,N):
            S = 0
            for i in range(0,n-1):
                S = S + A[L[f][i]][L[f][i+1]]
```

```
            if S < lmin :
                lmin = S
                Ll=L[f]
    return lmin, Ll
```

Tri par tas

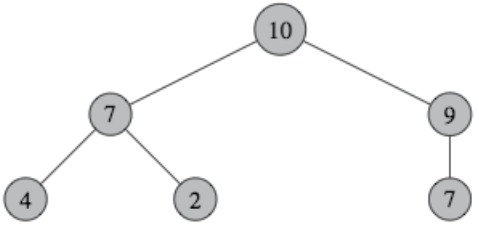
On termine ce chapitre en présentant un nouvel algorithme de tri. Il tire parti de la structure de **tas**, qui est une structure de graphe très particulière.

On appelle **arbre binaire** un graphe de la forme suivante :



Chaque sommet à part la **racine** a a un "père" (situé au-dessus dans le diagramme) et chaque sommet à part ceux situés en bas du diagramme a deux "fils" (situés au-dessus dans le diagramme).

On appelle **tas** un arbre binaire dans lequel on associe à chaque sommet un poids supérieur ou égal à ceux de ses fils. On considère une liste d'entiers L. Le tri par tas de L se fait en deux étapes. Tout d'abord on modifie L de façon à ce qu'elle représente un tas, c'est-à-dire si la construction de l'arbre obtenu en rangeant de bas en haut et de gauche à droite les éléments de la liste donne un tas. C'est ainsi le cas avec la liste [10,7,9,4,2,7] qui donne le tas de diagramme suivant :



Dans un deuxième temps, on récupère la liste triée. Pour cela on sépare notre liste entre une première partie L1 correspondant à un tas et une deuxième L2 donnant la liste triée. On répète alors le processus suivant :

- On échange le premier élément de L1 avec le dernier de L1. La partie de liste L2 gagne ainsi un élément alors que L1 en perd un.
- On fait « descendre » l'élément désormais en première position dans L1 de façon à ce qu'elle corresponde toujours à un tas.

Une fois L1 vide, L est triée.

L'algorithme

```
Fonction MiseEnTas(L)
Donnees
L : liste d'entiers
i : entier
Debut
    Pour i variant de 0 à len(L)
    Faire
        k<--i
        Tant Que k>0 et L[(k-1)//2] < L[k]
        Faire
            L[(k-1)//2],L[k] <-- L[k],L[(k-1)//2]
            k <-- (k-1)//2
        Fin Faire
    Fin Faire
    Retourner L
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste d'entiers L et retourne une liste contenant les mêmes éléments que L mais représentant un tas.
- **Terminaison :** La boucle Tant Que termine car la suite des valeurs prises par la variable k est à valeurs entières et strictement décroissante.
- **Correction :** On a l'invariant de boucle : « la liste formée par les éléments de L d'indices inférieurs ou égaux à i représente un tas ».
- **Complexité :** On note n la longueur de la liste L. Dans le pire des cas, on a besoin de l'ordre de $\ln(i)$ passages dans la boucle TantQue, on obtient donc une complexité dans le pire des cas de l'ordre de $n\ln(n)$. Dans le meilleur des cas, elle est linéaire en n.

Le programme en Python

```
def MiseEnTas(L):
    for i in range(0,len(L)):
        k = i
        while k > 0 and L[(k-1)//2] < L[k]:
            L[(k-1)//2],L[k] = L[k], L[(k-1)//2]
            k = (k-1)//2
    return L
```

L'algorithme

```
Fonction TriParTas(L)
Donnees
L : liste d'entiers
i, j, n : entier
test : booléen
Debut
    n <-- len(L)
    L <-- MiseEnTas(L)
    Pour i variant de 0 à n-1
    Faire
        L[0],L[n-1-i] <-- L[n-1-i],L[0]
```

```
        j <-- 0
        test <-- Vrai
        Tant Que test
        Faire
            Si 2*j+1 < n-1-i et
            ↳ L[2*j+1] >= L[2*j+2] et
            ↳ L[2*j+1] > L[j]
            Alors
                L[j], L[2*j+1] <-- L[j], L[2*j+1]
                j <-- 2*j +1
            Sinon Si 2*j+2 <n-1-i et
            ↳ L[2*j + 2] > L[2*j + 1]
            ↳ et L[2*j+2] > L[j]
            Alors
                L[j], L[2*j+2] <-- L[j], L[2*j+2]
                j <-- 2*j +2
            Sinon
                test <-- Faux
            Fin Si
        Fin Faire
    Fin Faire
    Retourner L
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une liste d'entiers L et retourne la liste triée contenant les mêmes éléments que L.
- **Terminaison :** La boucle Tant Que termine car la suite des valeurs prises par la variable k est à valeurs entières et strictement croissante.
- **Correction :** On a l'invariant de boucle : « la liste formée par les éléments de L d'indices supérieurs ou égaux à i est triée ».
- **Complexité :** On note n la longueur de la liste L. Dans le pire des cas, on a besoin de l'ordre de $\ln(n-i)$ passages dans la boucle TantQue, on obtient donc une complexité dans le pire des cas de l'ordre de $n\ln(n)$. Dans le meilleur des cas, elle est linéaire en n.

Le programme en Python

```
def TriParTas(L):
    n = len(L)
    L = MiseEnTas(L)
    for i in range(0,n-1):
        L[0], L[n-i-1] = L[n-i-1],L[0]
        j = 0
        test = True
        while test:
            if 2*j+1 < n-1-i and
            ↳ L[2*j+1] >= L[2*j+2]
            ↳ and L[2*j+1]> L[j]:
                L[j],L[2*j+1] = L[2*j+1], L[j]
                j = 2* j +1
            elif 2*j+2 < n-1-i and
            ↳ L[2*j+2] > L[2*j+1] and
            ↳ L[2*j+2] > L[j]:
                L[j],L[2*j+2] = L[2*j+2], L[j]
                j = 2* j +2
            else:
                test = False
    return L
```

« Si on attend d'une machine d'être infaillible, elle ne peut être intelligente. »

Alan Turing, mathématicien anglais.

15. Arithmétique

Le crible d'Erathostène

Soit N un entier naturel positif. On cherche à déterminer l'ensemble constitué de tous les nombres premiers inférieurs ou égaux à N.

L'algorithme

```
Fonction Crible(N)
Donnees
N,n,p,i : entiers,
LEntiers,LPremiers: listes d'entiers
Debut
LEntiers <-- [n pour n de 2 a N+1]
LPremiers <-- []
Tant que LEntiers !=[]
    Faire
    p <-- LEntiers[0]
    LEntiers <-- LEntiers[1:len(LEntiers)]
    LPremiers <-- LPremiers + [p]
    n <-- len(LEntiers)
    Pour i variant de 0 à n
        Faire
        Si LEntiers[n-1-i]%p = 0 Alors
                LEntiers <-- LEntiers[0:n-1-i]+
                    ↳LEntiers[n-i:len(LEntiers)]
        Fin Si
        Fin Faire
Retourner LPremiers
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée un entier naturel non nul N et retourne la liste des nombres premiers inférieurs ou égaux à N.
- **Terminaison :** À chaque passage dans la boucle, la liste LEntiers perd au moins son premier élément. Elle sera donc vide en au pire $N-1$ passages dans la boucle.
- **Correction :** On a pour invariant de boucle : « Les éléments de la liste LEntiers ne sont pas divisibles par les entiers inférieurs strictement au premier élément de la liste LEntiers ». En particulier, ce premier élément est bien un nombre premier.
- **Complexité :** On peut majorer le nombre de passages dans la boucle Tant Que par N. Pour chaque passage dans cette boucle, on effectue une boucle Pour parcourant les éléments de LEntiers effectuant une comparaison et un nombre d'affectations borné par la longueur de LEntiers. On obtient donc une complexité quadratique en N.

Le programme en Python

```
def Crible(N):
    LEntiers=[n for n in range(2,N+1)]
    LPremiers=[]
    while LEntiers!=[]:
        p=LEntiers[0]
        LEntiers=LEntiers[1:len(LEntiers)]
        LPremiers+=[p]
        n=len(LEntiers)
        for i in range(0,n):
            if LEntiers[n-1-i]%p==0:
                LEntiers=LEntiers[0:n-1-i]+
                ↳ LEntiers[n-i:len(LEntiers)]
    return LPremiers
```

Calcul de la fonction φ de Euler

L'algorithme

```
Fonction Phi(N)
Donnees
N, Position, n, l, i  : entier,
LEntiers : liste d'entiers
Debut
LEntiers <-- [n pour n de 2 a N-1]
Position <-- 0
Tant que Position<len(LEntiers)
    Faire
    n <-- LEntiers[Position]
    Si N%n=0 Alors
        Faire
        l <-- len(LEntiers)
        Pour i de 0 a l-Position
            Faire
            Si LEntiers[l-1-i]%n=0 Alors
                Faire
                LEntiers <-- LEntiers[0:l-1-i]+
                ↳ LEntiers[l-i:len(LEntiers)]
                Fin Faire
            Fin Si
            Fin Faire
        Fin Faire
    Sinon
        Position <-- Position+1
    Fin Si
    Fin Faire
Retourner len(LEntiers)+1
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée un entier naturel non nul N et retourne le nombre d'entiers naturels inférieurs à N et premiers avec N.

■ **Terminaison :** À chaque passage dans la boucle Tant Que, la liste LEntiers perd au moins un élément ou Position augmente de 1. Par conséquent, la suite des valeurs prises par la longueur de LEntiers moins Position est à valeurs entières et strictement décroissante. Elle prendra donc bien la valeur 0.

■ **Correction :** On a pour invariant de boucle : « Les éléments de la liste LEntiers d'indice inférieur à Position sont premiers avec N ».

■ **Complexité :** Dans le pire des cas, le nombre de passages dans la boucle Tant Que sera égal à $N-2$. Au i-ième passage dans cette boucle, la boucle Pour de l'ordre de la longueur de LEntiers moins Position OPEL. On peut donc majorer ce nombre d'OPEL par N-i ce qui en sommant pour i allant de 1 à $N-2$ donne une complexité globale quadratique en N.

Le programme en Python

```
def Phi(N):
    LEntiers=[n for n in range(2,N)]
    Position=0
    while Position<len(LEntiers):
        n=LEntiers[Position]
        if N%n==0:
            l=len(LEntiers)
            for i in range(0,l-Position):
                if LEntiers[l-1-i]%n==0:
                    LEntiers=LEntiers[0:l-1-i]+
                    ↳ LEntiers[l-i:len(LEntiers)]
        else:
            Position=Position+1
    return len(LEntiers)+1
```

Algorithme d'Euclide étendu

L'algorithme

```
Fonction EuclideEtendu(a,b)
Donnees
a,b : entiers,
r0,u0,v0,r1,u1,v1q,r : entiers
uIntermediaire, vIntermediare : entiers
Debut
r0,u0,v0 <-- a,1,0
r1,u1,v1 <-- b,0,1
Tant que r1 ≠ 0
    Faire
    q,r <-- r0//r1, r0%r1
    uIntermediaire <-- u0-q*u1
    vIntermediaire <-- v0-q*v1
    r0,u0,v0 <-- r1,u1,v1
    r1,u1,v1 <-- r,uIntermediaire,vIntermediaire
    Fin Faire
Retourner (r0,u0,v0)
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée deux entiers naturels a et b et retourne des entiers naturels $r0$, $u0$ et $v0$ tels que $r0$ soit le pgcd de a et de b et tels que l'on ait $r0 = u0.a + v0.b$.
- **Terminaison :** La suite des valeurs prises par la variable $r0$ est à valeurs entières et strictement décroissante à partir du deuxième terme. La variable $r0$ prendra donc bien la valeur 0.
- **Correction :** Un invariant de boucle est la propriété « r0 = u0.a + v0.b et r1 = u1.a + v1.b ».
- **Complexité :** On peut prouver par récurrence que le pire des cas est celui où a et b sont deux termes consécutifs de la suite de Fibonacci. Si on note le $n$ième terme de la suite de Fibonacci F_n alors le nombre d'itération de la boucle est $n - 1$ si $a = F_n$ et $b = F_{n-1}$.

Or F_n est de l'ordre de e^n, on a donc une complexité globale de l'algorithme logarithmique en $\max(a,b)$.

Le programme en Python

```
def EuclideEtendu(a,b):
    r0,u0,v0=a,1,0
    r1,u1,v1=b,0,1
    while r1!=0:
        q,r=r0//r1,r0%r1
        uIntermediaire=u0-q*u1
        vIntermediaire=v0-q*v1
        r0,u0,v0=r1,u1,v1
        r1,u1,v1=r,uIntermediaire,vIntermediaire
    return r0,u0,v0
```

Chiffrement RSA

On va écrire ici une fonction permettant de chiffrer et de déchiffrer en utilisant l'algorithme RSA. On rappelle qu'étant donné deux nombres premiers p et q distincts, on obtient la **clé publique** du chiffrement (n,e) en prenant $n = pq$ et e premier avec $\varphi(n)$. La clé privée est alors d tel que de soit congru à 1 modulo $\varphi(n)$. On va de plus écrire une fonction calculant d étant donné p et q. La sécurité du chiffrement repose en effet sur la difficulté de retrouver p et q étant donné n si ceux-ci sont suffisamment grands.
On donne enfin une fonction permettant de chiffrer un texte. On utilise la fonction ord de PYTHON donnant un entier associé à un caractère. On récupère le caractère avec chr.

L'algorithme

```
Fonction Chiffrement(a,b)
Donnees
a,b : entiers,
r0,u0,v0,r1,u1,v1q,r : entiers
uIntermediaire, vIntermediaire : entiers
```

```
Debut
r0,u0,v0 <-- a,1,0
r1,u1,v1 <-- b,0,1
Tant que r1 ≠ 0
    Faire
    q,r <-- r0//r1, r0%r1
    ulntermediaire <-- u0-q*u1
    vlntermediaire <-- v0-q*v1
    r0,u0,v0 <-- r1,u1,v1
    r1,u1,v1 <-- r,ulntermediaire,vlntermediaire
    Fin Faire
Retourner (r0,u0,v0)
Fin
```

Le programme en Python

```
def CalculCle(p,q,e):
    r,u,v=EuclideEtendu((p-1)*(q-1),e)
    if v<0:
        v=v+(p-1)*(q-1)
    return v

def Chiffrement(n,e,a):
    return a**e%n

def ChiffrementTexte(n,e,T):
        NT=''
        for c in T:
                NT=NT+chr(Chiffrement(n,e,ord(c)))
        return NT
```

Pour décoder, il suffit d'utiliser les mêmes fonctions que celles utilisées dans le codage mais avec la clé d à la place de e.

16. Autour des polynômes

Représentation et manipulation de polynômes

Les polynômes n'ont pas de représentation naturelle en PYTHON. On représente ici un polynôme

$$P(x) = \sum_{i=0}^{d} a_i X^i$$

par le tableau de ses coefficients sous forme de flottants, $[a_0, \ldots, a_d]$ avec a_d non nul ou $d = 0$. Ainsi, le polynôme $X + 2X^3$ pourra être représenté par le tableau [0.,1.,0.,2.].

De façon à pouvoir programmer l'algorithme d'Euclide étendu pour de tels polynômes, nous commençons par définir quelques fonctions permettant d'effectuer les opérations algébriques de base. Nous nous contentons ici de donner les fonctions PYTHON. On peut juste préciser que la fonction Nettoie sert à éliminer les premiers termes de la liste représentant un polynôme si ceux-ci sont nuls ou quasi-nuls. De tels termes pourront en effet apparaître suite aux calculs effectués dans la division euclidienne.

```
def Nettoie(P):
    while (abs(P[-1])>10**-15 and len(P)>1):
        P=P[0:len(P)-1]
    return P
def Degre(P):
    if len(P)>1:
        return len(P)-1
    elif abs(P[0])>10**-15:
        return 0
    else:
        return -np.Inf
```

```
def Somme(P1,P2):
    P=[]
    m=min(len(P1),len(P2))
    M=max(len(P1),len(P2))
    for i in range(0,m):
        P=P+[P1[i]+P2[i]]
    P=P+P1[m:M]
    P=P+P2[m:M]
    return P
def Produit(P1,P2):
    P=[0 for i in range(0,len(P1)+len(P2)-1)]
    for i in range(0,len(P1)):
        for j in range(0,len(P2)):
            P[i+j]=P[i+j]+P1[i]*P2[j]
    return P
```

Division euclidienne

L'algorithme

```
Fonction DivisionEuclidienne(P1,P2)
Donnees
P1,P2 : polynômes
R, Q, Pint : polynômes
Coeff : flottant
Debut
R<--P1
Q<--0
Tant que deg(R)>=deg(P2)
    Faire
    Coeff <-- CoeffDominant(R)/CoeffDominant(P2)
    Pint <--- Coeff*X^(deg(R) - deg(P2))
    R <-- R+P2*Pint
    Q <-- Q+Coeff*X^(deg(R) - deg(P2))
    Fin Faire
Retourner (Q,R)
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée deux polynômes P1 et P2 et retourne le quotient Q et le reste R de la division de P1 par P2.

■ **Terminaison :** La suite des degrés des polynômes stockés dans la variable R est à valeurs entières et strictement décroissante. Elle prendra donc la valeur 0.

■ **Correction :** On a pour invariant de boucle : « $P1 = Q.P2 + R$ ». Comme le dernier R est de degré strictement inférieur à celui de P2, il s'agit bien du reste de la division euclidienne, et Q est donc bien le quotient.

■ **Complexité :** À chaque passage dans la boucle Tant Que, le degré de R diminue du degré de P2. On obtient donc un nombre total d'opérations linéaire en $deg(P1) - deg(P2)$.

```
def DivisionEuclidienne(P1,P2):
    P1,P2=Nettoie(P1),Nettoie(P2)
    R=P1[:]
    Q=[0.0 for i in range(0,len(P1)-len(P2)+1)]
    while Degre(R)>=Degre(P2):
        Coeff=R[-1]/P2[-1]
        Pint=[0 for i in range(0,len(R)-len(P2))]
        Pint=Pint+[-1*Coeff]
        R=Somme(R,Produit(P2,Pint))
        Q[len(R)-len(P2)]=Coeff
        R=Nettoie(R)
    return Q,R
```

L'algorithme pour les polynômes

Il ne reste plus qu'à transcrire l'algorithme utilisé pour les entiers relatifs avec ces nouveaux outils. Pour la description de l'algorithme, on garde les notations des opérations utilisées en PYTHON avec les entiers.

L'algorithme

```
Fonction EuclideEtendu(P1,P2)
Donnees
P1,P2 : polynômes,
R0,U0,V0,R1,U1,V1,Q,R : polynômes
UIntermediaire, VIntermediaire : polynômes
Debut
R0,U0,V0 <-- P1,1,0
R1,U1,V1 <-- P2,0,1
Tant que degré(R1) > 0
    Faire
    Q,R <-- R0//R1, R0%R1
    UIntermediaire <-- U0-Q*U1
    VIntermediaire <-- V0-Q*V1
    R0,U0,V0 <-- R1,U1,V1
    R1,U1,V1 <-- R,UIntermediaire,VIntermediaire
    Fin Faire
Retourner (R0,U0,V0)
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée deux polynômes $P1$ et $P2$ et retourne trois listes polynômes $R0$, $U0$ et $V0$ tels que $R0$ est le pgcd des polynômes $P1$ et $P2$ et $U0.P1 + V0.P2 = R0$.
- **Terminaison :** La suite des degrés des polynômes stockés dans $R1$ est une suite d'entiers strictement décroissante à partir du rang 1, elle prendra donc bien une valeur négative ou nulle.
- **Correction :** On a pour invariant de boucle : « $R0 = U0.P1 + V0.P2$ et $R1 = U1.P1 + V1.P2$ ». Comme le dernier $R0$ est le dernier reste de degré strictement positif, il contient bien le pgcd des polynômes considérés.
- **Complexité :** Le nombre de passages en boucle est de l'ordre du maximum des degrés de $P1$ et $P2$. Pour déterminer la complexité de l'algorithme, il faut ensuite estimer les degrés des polynômes successifs auxquels on applique la division euclidienne. Finalement, on peut montrer que l'algorithme est linéaire en $deg(P1).deg(P2)$.

Le programme en Python

```
def EuclideEtendu(P1,P2):
    R0,U0,V0=P1,[1.0],[0.0]
    R1,U1,V1=P2,[0.0],[1.0]
    while len(R1)>1:
        Q,R=DivisionEuclidienne(R0,R1)
        UIntermediaire=Nettoie(
        ↳ Somme(U0,Produit(Produit([-1.0],Q),U1)))
        VIntermediaire=Nettoie(
        ↳ Somme(V0,Produit(Produit([-1.0],Q),V1)))
        R0,U0,V0=R1,U1,V1
        R1,U1,V1=R,UIntermediaire,VIntermediaire
    return R0,U0,V0
```

Boîte à outils

Le module numpy.polynomial fournit une classe de polynômes appelée Polynomial. Après avoir importée celle-ci, on dispose des fonctions et méthodes suivantes :

`Polynomial(L)`	retourne le polynôme dont les coefficients sont dans L
`P()`	applique le polynôme P à un ou des éléments
`P.coef[]`	donne le coefficient de degré précisé
`P.degree()`	donne le degré de P
`P.roots()`	donne les racines de P
`P.deriv()`	donne le polynôme dérivé de P
`P // Q`	donne le quotient de P par Q
`P % Q`	donne le reste de la division de P par Q

17. Gauss-Legendre

Polynômes de Legendre

Pour tout entier naturel k on appelle **k-ième polynôme de Legendre** le polynôme :

$$\boxed{P_k(X) = \frac{1}{2^k k!}\frac{d^k}{dx^k}((X^2-1)^k)}$$

Ces polynômes forment une suite de polynômes orthogonaux pour le produit scalaire défini sur l'ensemble des fonctions continues sur $[-1,1]$ par :

$$< f, g >= \int_{-1}^{1} fg$$

La suite (P_k) des polynômes de Legendre vérifie pour tout $k \geqslant 2$:

$$\boxed{P_k(X) = \frac{2k-1}{k}XP_{k-1}(X) - \frac{k-1}{k}P_{k-2}(X)}$$

Cette relation de récurrence, nous fournit une méthode efficace pour calculer les polynômes de Legendre ainsi que leurs polynômes dérivés en PYTHON.

Nous ne détaillons pas l'algorithme pour les polynômes dérivés, qui est le même mais en appliquant la relation de récurrence vérifiée par la suite (Q_k) des polynômes dérivés des polynômes de Legendre :

$$\boxed{Q_k(X) = \frac{2k-1}{k}P_{k-1}(X) + \frac{2k-1}{k}Q_{k-1}(X) - \frac{k-1}{k}Q_{k-2}(X)}$$

L'algorithme

```
Fonction CalculPolynomeLegendre(n)
Donnees
n : entiers,
P1,P2,Q : polynômes,
k : entier
Debut
    P1 <-- [1.0]
    P2 <-- [0.,1.]
Si k = 0 alors Retourner P1 FinSi
Pour k variant de 2 à n
    Faire
    Q <-- P2
    P2 <-- (2*k-1)/k*X*P1 - (k-1)/k*P2
    P1 <-- Q
    Fin Faire
Retourner P2
Fin
```

Le programme en Python

```
def CalculPolynomeLegendre(n):
    P1=[1.]
    P2=[0.,1.]
    if k==0:
        return P1
    for k in range(2,n+1):
        Q=P2[:]
        P2=Somme(Produit([0.,(2*k-1)/k],P1),
                 ↳ Produit([-(k-1)/k],P2))
        P1=Q[:]
    return P2
```

```
def CalculPolynomeLegendreD(n):
    P1=[0.]
    P2=[1.]
    if k==0:
        return P1
    for k in range(2,n+1):
```

```
        Q=P2[:]
        P2=Somme(CalculPolynomeLegendre(k-2),
        ↳ Somme(Produit([0.,(2*k-1)/k],P1),
        ↳ Produit([-(k-1)/k],P2)))
        P1=Q[:]
    return P2
```

Méthode d'intégration numérique

Comme on ne présente pas à proprement parler de nouvel algorithme ici, mais plutôt des applications des algorithmes classiques dejà présentés auparavant, on se contentera ici de donner les programmes en PYTHON.

Le principe de la méthode d'intégration de Gauss-Legendre est d'approcher une intégrale $\int_{-1}^{1} f(x)\mathrm{d}x$ par une somme de la forme

$$\sum_{k=1}^{n} w_k f(x_k)$$

où les x_k sont les racines de P_k et où les w_k sont les réels tels que pour tout polynôme Q de degré inférieur ou égal à $2n-1$ on ait :

$$\int_{-1}^{1} Q(t)\mathrm{d}x = \sum_{k=1}^{n} w_k Q(x_k).$$

Pour déterminer les racines x_k, on peut utiliser la méthode de Newton. On prend comme comme valeur initiale $\cos\left(\frac{k-0{,}25}{n+0{,}5}\pi\right)$ pour approcher x_k. On obtient la fonction suivante :

```
def RacApprochee(k,i,epsilon):
    u0=np.cos(np.pi*(i-0.25)/(k+0.5))
    L=CalcPprime(u0,k)
    u1=u0-L[0]/float(L[1])
    while abs(u1-u0)>epsilon:
        L=CalcPprime(u1,k)
        u0,u1=u1,u1-L[0]/float(L[1])
    return u1
```

Les w_k peuvent eux être déterminés en résolvant le système formé par les équations :

$$\int_{-1}^{1} x^p \mathrm{d}x = \sum_{k=1}^{n} w_k x_k^p$$

pour p allant de 0 à n. Ce système est bien de Cramer car il est défini par une matrice de Van Der Monde. En réutilisant la fonction Solution_Gauss (voir p 129), on obtient :

```
def CalcCoeff(n):
    a=-1
    b=1
    L=[RacApprochee(n,i,0.01)
    ↳ for i in range(1,n+1)]
    LY=[(a+b)/2.0+x*(b-a)/2.0 for x in L]
    M=np.array([[y**k for y in LY]
    ↳ for k in range(0,n)])
    B=np.array([(b**k-a**k)/float(k)
    ↳ for k in range(1,n+1)])
    X=Solution_Gauss(M,B)
    return [[x[0] for x in X],LY]
```

En utilisant finalement un changement de variable, on peut écrire une fonction permettant de calculer une valeur approchée d'une intégrale $\int_a^b f(t)\mathrm{d}t$ par la méthode détaillée ci-dessus :

```
def CalcLegendre(f,a,b,n,N):
    L=CalcCoeff(n)
    S=0
    for i in range(0,N):
        print S,L
        g=lambda x :
        ↳ f(a+(b-a)*(2*i+1)/float(2.0*N)+
        ↳ x*(b-a)/(2.0*N))
        Stemp=sum([float(L[0][k])*g(L[1][k])
        ↳ for k in range(0,len(L[0]))])
        S=S+(b-a)/(2.0*N)*Stemp
    return S
```

Boîte à outils

Le module numpy.scipy.legendre contient la fonction suivante :

`leggauss(d)`	retourne deux tableaux donnant les points et les poids à utiliser dans la méthode pour un résultat exact jusqu'au degré $2d + 1$

« Si les gens ne croient pas que les mathématiques sont simples, c'est uniquement parce qu'ils ne réalisent pas à quel point la vie est compliquée. »

John Von Neumann, mathématicien américain.

18. Probabilités

Simulation des lois classiques

On utilisera ici la fonction random du module du même nom donnant comme résultat un nombre « aléatoire » de l'intervalle $[0, 1[$. Les seules propriétés à connaître sur ce nombre sont d'une part que la probabilité d'obtenir un élément d'un intervalle de bornes α et β avec $0 \leqslant \alpha \leqslant \beta \leqslant 1$ est $\beta - \alpha$ et d'autre part que les appels à cette fonction sont indépendants les uns des autres.

On donne dans ce paragraphe des programmes simulant les lois uniforme, binomiale et géométrique.

L'algorithme

```
Fonction SimulationUniforme(a,b)
Donnees
a,b : entiers
Debut
    t <-- random()
    t <-- t*(b-a+1)+a
    t <-- PartieEntiere(t)
Retourner t
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée deux entiers naturels a et b tels que l'on ait $a < b$ et retourne un entier de l'ensemble $\{a, \ldots, b\}$ avec la même probabilité pour chaque résultat possible.
- **Correction :** la probabilité que l'on obtienne comme résultat k est la probabilité que le nombre obtenu avec random appartienne à $\left[\frac{k-a}{b-a+1}, \frac{k+1-a}{b-a+1}\right[$. Cette probabilité est donc bien égale à $\frac{1}{b-a+1}$.

Le programme en Python

```
def SimulationUniforme(a,b):
    t=rd.random()
    t=t*(b-a+1)+a
    t=int(t)
    return(t)
```

L'algorithme

```
Fonction SimulationBinomiale(n,p)
Donnees
n : entier
p : flottant
Debut
    S=0
    Pour i variant de 1 à n+1
        t <-- random()
        Si t < p
        Alors S <-- S+1
        Fin Si
Retourner S
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée un entier naturel n et un flottant p appartenant à $[0, 1]$. Elle retourne un entier naturel inférieur ou égal à n tel que la probabilité de chaque résultat possible suive la loi binomiale de paramètres n et p.

- **Correction :** À chaque passage dans la boucle Pour, la probabilité que l'on ajoute 1 à la variable S est égale à p car c'est la probabilité que random() donne un nombre de l'intervalle $[0, p]$. À la sortie de la boucle, S contient donc la somme de n variables aléatoires prenant la valeur 1 avec la probabilité p et la valeur 0 avec la probabilité $1 - p$. Comme la somme de n variables aléatoires indépendantes de même loi de Bernoulli de paramètre p suit la loi binomiale de paramètres n et p, le résultat de la fonction suit bien la loi attendue.

■ **Complexité :** Lors de l'exécution de cet algorithme, on effectue $3n$ OPEL et n appels à la fonction random().

Le programme en Python

```
def SimulationBinomiale(n,p):
    S=0
    for i in range(n+1):
        t=rd.random()
        if t < p:
            S=S+1
    return (S)
```

L'algorithme

```
Fonction SimulationGeometrique(p)
Donnees
p : flottant
Debut
    N <-- 0
    Tant Que random() >p
        Faire
        N <-- N+1
        Fin Faire
Retourner N
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée un flottant p élément de $[0, 1]$. Elle retourne un entier naturel non nul tel que la probabilité de chaque résultat possible suive la loi géométrique de paramètre p.

■ **Correction :** Soit $k \in \mathbb{N}^*$. la probabilité que la variable N soit égale à k à la fin de la boucle Tant Que est la probabilité que l'on ait obtenu $k-1$ fois un résultat de random appartenant à $[p, 1]$ puis un résultat appartenant à $[0, p]$. Par indépendance, la probabilité de retourner k est donc égale à $(1-p)^{k-1} \times p$, ce qui donne bien un résultat suivant la loi géométrique de paramètre p.

■ **Complexité :** Lors de l'exécution de cet algorithme, si on fait n passages dans la boucle on effectue $3n - 2$ OPEL et n appels à la fonction random().

Le programme en Python

```
def SimulationGeometrique(p):
    N=0
    while rd.random() >= p:
        N=N+1
    return(N)
```

Comparaison des lois binomiale et de Poisson

Pour comparer le comportement de variables aléatoires suivant des lois de la forme $\mathcal{P}(\lambda)$ et $\mathcal{B}(n, \lambda/n)$ on va simuler un nombre N de tirages suivant la loi binomiale pour un n et calculer la proportion obtenue pour chaque valeur. On tracera alors la fonction de répartition théorique correspondant à la loi de Poisson ainsi que la fonction de répartition empirique obtenue pour la loi binomiale pour comparer. On utilisera pour ce faire une fonction SommeCumulee calculant les sommes partielles correspondant à une liste donnée.

L'algorithme

```
Fonction ComparaisonBinomialePoisson(λ,n,N)
Donnees
n, N, i, a : entiers
λ : flottant
LB, LP : listes de flottants
Debut
    LB <-- [0 pour i variant de 0 à n+1]
    Pour i variant de 0 à N
    Faire
        a <-- SimulationBinomiale(n,l/n)
        LB[a] <-- LY[a] + 1/N
    Fin Faire
```

```
    plot(SommeCumulee(LY),'r')
    LP <-- [exp(-λ)]
    Pour i variant de 0 à n
    Faire
        LP <-- LP + [LP[-1]/(i+1)*λ]
    Fin Faire
    Tracer(SommeCumulee(LP),'b')
Fin
```

Analyse de cet algorithme :

■ **Spécification :** Cette fonction prend en entrée un flottant λ et deux entiers n et N. Elle trace la fonction de répartition empirique pour la loi binomiale de paramètres n et p (en rouge) et la fonction de répartition théorique pour la loi de Poisson de paramètre λ (en bleu).

■ **Correction :** La première boucle Pour simule N tirages de la loi binomiale et en sortie de boucle on récupère dans LB les proportions obtenues pour chaque résultat possible. En faisant la somme cumulée des valeurs de LB on obtient bien la fonction de répartition empirique pour la loi binomiale. La liste LP se construit par la relation de récurrence $LP[k+1] = LP[k]/(k+1)*\lambda$. Comme $LP[0]$ vaut $\exp(-\lambda)$, on montre (par récurrence) que pour tout k de $\{0,\ldots,n\}$:

$$LP[n] = \frac{\lambda^n}{n!}$$

ce qui donne bien par somme cumulée la fonction de répartition théorique de la loi de Poisson de paramètre λ.

Le programme en Python

```
def SommeCumulee(L):
    for i in range(1,len(L)):
        L[i]=L[i]+L[i-1]
    return(L)
```

```
def ComparaisonBinomialePoisson(l,N,n):
    LY=[0 for i in range(0,n+1)]
    for i in range(N):
```

```
        a=SimulationBinomiale(n,l/n)
        LY[a]=LY[a]+1/N
    plt.plot(SommeCumulee(LY),'r')
    LP=[np.exp(-l)]
    for i in range(0,n):
        LP=LP+[LP[-1]/(i+1)*l]
    plt.plot(SommeCumulee(LP),'b')
    plt.show()
```

Illustration de la loi faible des grands nombres

Soit $(X_n)_{n\in\mathbf{N}^*}$ des variables aléaoires mutuellement indépendantes définies sur un même espace probabilisé et admettant une espérance et une variance. Soit $\epsilon > 0$. On a :

Loi faible des grands nombres

$$\lim_{n\to+\infty} P\left(\left|\frac{X_1+\cdots+X_n}{n} - E(X)\right| \geq \epsilon\right) = 0$$

Pour illustrer la loi faible des grands nombres, on tracera entre deux entiers n_1 et n_2 la proportion de tirages obtenus sur N tirages dans l'intervalle donné par le théorème pour un epsilon choisi. On utilisera ici la loi de Bernoulli de paramètre p, mais on pourrait adapter très facilement à toute loi voulue.

L'algorithme

```
Fonction IllustrationLFGN(n1,n2,N,p,epsilon)
Donnees
n1, n2, N, S, i, j, k : entiers
LS : liste d'entiers
p, epsilon : flottants
Debut
    LS<--[0 pour i variant de 0 à n2-n1]
    Pour i variant de n1 à n2+1
    Faire
```

```
        Pour j variant de 0 à N
        Faire
            S <--0
            Pour k variant de 0 à i
            Faire
                Si random()<p
                Alors S<--S+1
                Fin Si
            Fin Faire
            Si |S/i-p| >= epsilon
            Alors LS[i-n1]<--LS[i-n1]+1/N
            Fin Si
        Fin Faire
    Fin Faire
    Tracer (LS)
Fin
```

Le programme en Python

```
def IllustrationLFGN(n1,n2,N,p,epsilon):
    LS=[0 for l in range(n1,n2+1)]
    for i in range(n1,n2+1):
        for j in range(0,N):
            S=0
            for k in range(0,i):
                if rd.random()<p:
                    S=S+1
            if abs(S/i-p)>=epsilon:
                LS[i-n1]=LS[i-n1]+1/N
    plt.plot(LS)
    plt.show()
```

Boîte à outils

Le module numpy.random fournit les fonctions de simulation suivantes :

`random_integers(a,b)`	donne un résultat suivant la loi uniforme sur ⟦a, b⟧
`binomial(n,p)`	donne un résultat suivant la loi binomiale de paramètres n et p
`geometric(p)`	donne un résultat suivant la loi géométrique de paramètre p
`poisson(lambda)`	donne un résultat suivant la loi de Poisson de paramètre λ

De plus, dans chaque cas, on peut rajouter un troisième paramètre indiquant une longueur l entière. Le résultat sera alors un ndarray de longueur l dont chaque coefficient a été obtenu par une simulation (indépendante des autres) de la loi précisée.

« J'ai toujours rêvé que mon ordinateur soit aussi simple à utiliser que mon téléphone. Ce rêve est devenu réalitéă : je ne sais plus comment utiliser mon téléphone. »

Bjarne Stroustrup, informaticien danois.

19. Calculs matriciels

Quelques opérations de base sur les matrices

Commençons par programmer quelques opérations élémentaires sur les matrices que nous utiliserons par la suite. Nous pourrions évidemment recourir aux fonctions du module NUMPY, mais il n'est pas inutile de savoir programmer des opérations les plus courantes.

On donne ici des fonctions PYTHON permettant de calculer :

- La matrice identité ;
- La trace d'une matrice carrée ;
- La transposée d'une matrice ;
- Le produit de deux matrices.

Les matrices sont ici données sous forme de listes de listes.

```
def Identite(n):
    I=[[0 for j in range(n)] for i in range(n)]
    for i in range(0,n):
        I[i][i] = 1
    return I

def Trace(A):
    n = len(A)
    T=0
    for i in range(n):
        T=T+A[i][i]
    return T

def Transposee(A):
    m = len(A)
    n = len(A[0])
    B=[[0 for j in range(m)] for i in range(n)]
```

```
    for i in range(n):
        for j in range(m):
            B[i][j] = A[j][i]
    return(B)

def Produit(A,B):
    m = len(A)
    n = len(A[0])
    p = len(B)
    q = len(B[0])
    if n != p:
        print('erreur :
        ↳ les matrices ne sont pas compatibles')
    else:
        M=[[0 for j in range(q)]
        ↳ for i in range(m)]
        coeff=0
        for i in range(m):
            for j in range(q):
                for k in range(n):
                    coeff = coeff+A[i][k]*B[k][j]
                M[i][j] = coeff
    return M
```

Valeur propre de plus grand module

Pour toute matrice A carrée d'ordre n telle que les valeurs propres λ_i de A vérifient $|\lambda_1| > |\lambda_i|$ pour tout $i > 1$, on a :

$$\lim_{n\to+\infty} \frac{\mathrm{Tr}(A^{n+1})}{\mathrm{Tr}(A^n)} = \lambda_1$$

On va ici tracer les quotients des traces de deux puissances successives d'une matrice de manière à identifier une valeur approchée de la plus grande valeur propre de celle-ci. On donnera de plus comme résultat la dernière valeur obtenue.

L'algorithme

```
Fonction VPMAppr(A,N)
Donnees
A, M1, M2  : matrices
N, i : entiers,
L : liste de flottants
Debut
    M1 <-- A
    M2 <-- A^2
    L <-- [Tr(M2)/Tr(M1)]
    Pour i variant de 0 à N+1
    Faire
        M1 <-- M1.A
        M2 <-- M1.A
        L <-- L + [Tr(M2)/Tr(M1)]
    Fin Faire
    Tracer(L)
Retourner L[-1]
Fin
```

Le programme en Python

```
def VPMAppr(A,N):
    M1=A
    M2=Produit(A,A)
    L=[Trace(M2)/Trace(M1)]
    for i in range(N+1):
        M1=Produit(M1,A)
        M2=Produit(M1,A)
        L=L+[Trace(M2)/Trace(M1)]
    plt.plot(L)
    plt.show()
    return L[-1]
```

Exponentielle de matrice

On donne ici une fonction qui retourne une valeur approchée de l'exponentielle d'une matrice A, en calculant les sommes partielles

$$\sum_{k=0}^{N} \frac{A^k}{k!}.$$

L'algorithme

```
Fonction ExponentielleM(A,N)
Donnees
A, M, E : matrices
i : entier
Debut
    M <-- Identite(len(A))
    E <-- M
    Pour i variant de 1 à N+1
    Faire
        M <-- M*A/i
        E <-- E + M
    Fin Faire
Retourner E
Fin
```

Analyse de cet algorithme :

- **Spécification :** Cette fonction prend en entrée une matrice A et un entier N. Elle retourne la somme partielle de rang N de l'exponenielle de A :

$$\sum_{k=0}^{N} \frac{A^k}{k!}$$

- **Correction :** Si on note M_k les valeurs successives stockées dans la variable M, on a $M_0 = I_n$ et pour tout entier k inférieur à N on a la relation $M_{k+1} = \frac{1}{k+1} M.A$ qui implique par une récurrence immédiate l'égalité $M_k = \frac{1}{k!} A^k$ pour tout entier k inférieur à N.

Comme E contient la somme des M_k successifs, la fonction retourne bien la somme voulue.

Le programme en Python

```
def ExponentielleM(A,N):
    M=Identite(len(A))
    E=M
    for i in range(1,N+1):
        M=Produit(M,A)
        for i in range(0,len(A)):
            for j in range(0,len(A)):
                M[i][j] = M[i][j] / i
        E=E+M
    return(E)
```

Boîte à outils

Le module numpy contient de très nombreux outils pour travailler avec des matrices, de type array. On peut notamment retenir les fonctions suivantes :

`array()`	donne un objet, de type array correspondant à l'entrée (liste ou liste de listes)
`zeros((n,p))`	donne la matrice nulle à n lignes et p colonnes
`identity(n)`	donne la matrice identité d'ordre n
`trace(A)`	donne la trace de la matrice A
`transpose(A)`	donne la transposée de la matrice A
`dot(A,B)`	donne le produit des matrices A et B

Pour l'exponentielle de matrice on aura cependant recours à une fonction de la bibliothèque scipy.linalg :

`expm(A)`	donne l'exponentielle de la matrice A

« Mesurer la progression du développement d'un logiciel à l'aune de ses lignes de code revient à mesurer la progression de la construction d'un avion à l'aune de son poids. »

Bill Gates, informaticien américain.

20. Piles

On présente ici une structure de données classique en informatique. Nous mettrons en oeuvre cette structure en PYTHON en créant une classe, ce qui donnera un exemple simple de programmation orienté objet.

Création de la classe Pile

On appelle **pile** une collection ordonnée d'éléments $(a_0, \ldots, a_n)$ dont on appelle a_n le **sommet**. On s'autorise à effectuer les opérations suivantes sur une pile :

- Tester si la pile est vide ;
- Prendre le sommet de la pile ;
- Retirer l'élément au sommet de la pile, dans ce cas a_{n-1} devient donc le nouveau sommet de la pile ;
- Ajouter un élément a_{n+1} qui devient le nouveau sommet de la pile.

On peut se représenter un tel objet sous la forme :

3
12
67
12

avec ici le sommet égal à 3.

Cette structure de données est particulièrement adaptée aux problèmes pour lesquels les dernières informations stockées seront les premières à devoir être traitées, puisqu'on ne peut accéder de manière directe

qu'au sommet de la pile. On dit pour cela qu'il s'agit d'une structure **last in first out**.

Pour définir ces objets en PYTHON, on utilise ici la commande class, qui permet de créer une structure de données et un ensemble d'opérations ou méthodes possibles avec ceux-ci :

Le programme en Python

```
class Pile :
    def __init__(self):
        self.elements=[]

    def EstVide(self):
        if self.elements==[]:
            return True
        else :
            return False

    def Ajouter(self,x):
        self.elements = self.elements+[x]

    def Retirer(self):
        if self.EstVide():
            print('Impossible de retirer :
            ↳ la pile est vide.')
        else :
            n=len(self.elements)
            self.elements=self.elements[:n-1]
    def Sommet(self):
        if self.EstVide():
            print("Il n'y a pas de sommet :
            ↳ la pile est vide.")
        else :
            return self.elements[-1]
```

On peut alors construire une pile vide en écrivant :

```
P = Pile()
```

et lui ajouter des éléments en utilisant la méthode Ajouter. Comme pour toutes les méthodes, la syntaxe est de la forme :

objet.methode(arguments)

Ainsi,

```
P.Ajouter(12)
P.Ajouter(67)
P.Ajouter(12)
P.Ajouter(3)
```

crée la pile que nous avons représentée précédemment.

Quelques fonctions utilisant les piles

Nous présentons ici quelques fonctions simples typiques du travail avec les piles. Dans l'ordre, nous programmons :

- Une fonction renversant une pile donnée ;
- Une fonction ou plutôt une procédure d'affichage d'une pile ;
- Une fonction calculant la hauteur d'une pile donnée ;
- Une fonction superposant deux piles données.

Dans chaque cas, on fait bien attention à ce que les piles d'origine soient inchangées à la fin de la fonction. En effet, la structure sous-jacente que nous avons prise pour nos piles est la structure de liste PYTHON.

Le programme en Python

```
def Renverser(P):
    P1 = Pile()
    P2 = Pile()
    while not(P.EstVide()):
        P2.Ajouter(P.Sommet())
        P1.Ajouter(P.Sommet())
        P.Retirer()
    while not(P1.EstVide()):
        P.Ajouter(P1.Sommet())
```

```
        P1.Retirer()
    return P2

def Affichage(P):
    P1=Renverser(P)
    while not(P1.EstVide()):
        print(P1.Sommet())
        P1.Retirer()

def Hauteur(P):
    P1 = Pile()
    H = 0
    while not(P.EstVide()):
        H = H + 1
        P1.Ajouter(P.Sommet())
        P.Retirer()
    while not(P1.EstVide()):
        P.Ajouter(P1.Sommet())
        P1.Retirer()
    return H

def Superposer(P1,P2):
    P3 = Pile()
    P4 = Pile()
    while not(P1.EstVide()):
        P3.Ajouter(P1.Sommet())
        P4.Ajouter(P1.Sommet())
        P1.Retirer()
    while not(P4.EstVide()):
        P1.Ajouter(P4.Sommet())
        P4.Retirer()
    while not(P2.EstVide()):
        P3.Ajouter(P2.Sommet())
        P4.Ajouter(P2.Sommet())
        P2.Retirer()
    while not(P4.EstVide()):
        P2.Ajouter(P4.Sommet())
        P4.Retirer()
    return P3
```

Index

LE SAVIEZ-VOUS ?

Python

Que vient faire le nom d'un serpent pour nommer un langage de programmation ? Un groupe d'humoristes britanniques anima une émission télévisuelle à la BBC sous le titre de *Monty Python's Flying Circus* entre 1969 et 1974. Celle-ci recueillit un tel succès qu'elle fut reprise par de nombreuses chaînes étrangères.

Le mathématicien hollandais Guido **van Rossum**, né en 1956, s'est enthousiasmé pour cette émission. Son autre passion fut l'informatique : dès 1986, il commence à concevoir des langages de programmation. Trois ans plus tard, il en développe un nouveau qu'il nomme par humour du nom de la troupe de comiques. La première version publique voit le jour en 1991.

Très philanthrope, van Rossum a souhaité laisser PYTHON en licence libre et en *open data* ; il continue régulièrement à améliorer son œuvre. Les grandes qualités de ce langage, en particulier sa simplicité et sa lisibilité, ont permis son développement au point qu'il devenu l'un des plus répandus dans le monde.

Index des commandes

Index général

Clique ici pour plus de livres : https://t.me/formations8

Clique ici pour plus de livres : https://t.me/formations8

PRÉPAS SCIENCES

Une vision claire des savoirs, une mobilisation rapide des connaissances et des compétences sont les atouts indispensables à la réussite en prépa.
Ce formulaire d'Informatique Pour Tous (IPT) répond parfaitement à ces exigences.
Il présente de façon synthétique l'intégralité du programme d'IPT des classes préparatoires aux grandes écoles scientifiques première et deuxième années.

- On y trouvera d'abord une base de connaissances théoriques indispensables en algorithmique, en programmation et pour les bases de données.
- Tous les algorithmes et scripts Python des programmes de première et deuxième années, classés par thème, sont ensuite détaillés et analysés.
- Finalement, deux index très précis (un index général et un index des commandes Python) permettent de trouver rapidement la notion cherchée.

Conçu pour appréhender tout le programme d'IPT en un clin d'œil, ce formulaire est le compagnon idéal pour la préparation des devoirs, les séances de travaux pratiques d'informatique, une ressource précieuse pour les TIPE mais surtout pour accompagner les révisions avant les concours.
Il complète intelligemment les ouvrages de la collection Prépas Sciences, qui permettent une acquisition solide des connaissances.

9 782340 026346

www.editions-ellipses.fr