# TABLE DES MATIÈRES

CHAPITRE 1

# INTRODUCTION

## 1.1 Les pères de l'informatique.

Durant la deuxième guerre mondiale le mathématicien anglais Alan TURING intégra les services secrets britanniques avec pour mission de déchiffrer les messages codés utilisés par les allemands. En effet, la marine allemande gagnait la bataille de l'atlantique grâce à sa flotte de sous-marins U-Boot et aux renseignements qui leur étaient envoyés.

Les allemands utilisaient une machine, dénommée Enigma, pour coder leurs messages. Alan TURING, en opposition à de nombreux scientifiques de l'époque, pensait que seule une machine pouvait comprendre une autre machine. Il inventa donc sa propre machine et réussit!

Le décryptage des messages allemands envoyés à leurs sous-marins est considéré par les historiens comme un élément clé de la victoire des alliés.

Alan TURING voulut aller plus loin et créer une machine capable de résoudre tous les problèmes, et surtout capable d'imiter la pensée humaine, selon un "algorithme", qu'on nommera ensuite machine de Turing. Considérons l'expérience qui consiste à poser une série de questions à une personne qui ne doit répondre que par "oui" ou par "non". On peut imaginer connaître parfaitement cette personne à l'aide d'une séquence finie de questions. Cette personne serait donc représentée par une série d'instructions accompagnées de leurs réponses binaires!

Ce concept de programmation initié par la machine de Turing est alors utilisé par les premiers concepteurs d'ordinateurs.

Dans le même temps, le mathématicien americano-hongrois John VON NEUMANN, qui participa au projet Manhattan et travailla ainsi à la découverte de la bombe atomique, rapporta un certain nombre de travaux de l'époque sur l'informatique. Sa publication conduit à la création d'un modèle de calculateur, qu'il attribuait lui-même à Alan TURING, portant le nom d'architecture de von Neumann et utilisé dans la quasi totalité des ordinateurs aujourd'hui.

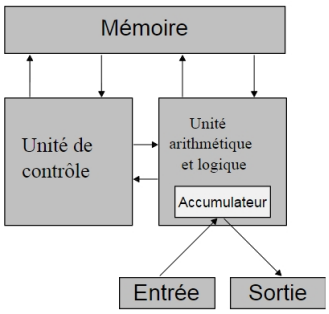



FIGURE 1.1 – Schéma de l'architecture de von Neumann

## 1.2 La mémoire

Une mémoire est caractérisée par sa capacité, son temps d'accès aux données et son débit.

Les données sont enregistrées dans une mémoire sous forme binaire, l'information élémentaire est appelée *bit* (Binary Digit). Le nombre de bits que peut contenir une mémoire définit sa **capacité**. Toutefois, les capacités ne sont pas données en nombre de bits, mais en nombre d'octets, c'est-à-dire en groupement de huit bits. On définit ensuite les multiples suivants :

| | | | | |
|---|---|---|---|---|
| 1 kilooctet (ko) | = | 1000 octets | = | $10^3$ octets |
| 1 mégaoctet (Mo) | = | 1000 ko | = | $10^6$ octets |
| 1 gigaoctet (Go) | = | 1000 Mo | = | $10^9$ octets |
| 1 téraoctets (To) | = | 1000 Go | = | $10^{12}$ octets |

*Remarque.* Dans les débuts de l'informatique, les préfixes "kilo", "méga", ... ont été utilisés de manière erronée pour désigner des puissances de 2. Plus précisément, comme $1024 \simeq 1000$, le kilooctet était utilisé pour désigner 1024 soit $2^{10}$ octets ; le mégaoctet pour $2^{20}$ octets... Il faut se méfier de cet usage qui perdure même s'il va à l'encontre de la norme.

Le **temps d'accès** est le temps mis par la machine entre l'instant de lancement de la commande d'écriture et l'instant où elle est réalisée.

Le **débit** est le nombre de bits écrits ou lus en une seconde.
Il s'exprime en **bit/s**. Parfois, lors de la transmission de données, le débit est donné en **bauds**, ce qui représente le nombre d'unité de signal par unité de temps. On a la formule :

$$\text{débit binaire} = \text{débit en bauds} \times \text{nombre de bits par baud}$$

Lorsque chaque bit est codé par un signal électrique, le débit binaire est égal au débit en bauds.

On peut classer les mémoires en deux catégories : les mémoires vives et les mémoires de masse.

Les mémoires **vives** ou **RAM** pour Random Access Memory sont usuellement plus rapides et de plus faible capacité (4 à 8 Go) que les mémoires de masse et surtout volatiles car elles perdent leur contenu dès qu'elles sont hors tension. Une RAM est constituée de composants électroniques.

Les mémoires **de masse** ou **ROM** pour Read Only Memory sont plus lentes mais de plus grande capacité (jusqu'à plusieurs To) et surtout n'ont pas besoin de courant pour garder l'information. Un disque dur classique (HDD) est composé de plateaux tournants et de têtes de lecture.

La rapidité d'un ordinateur dépend de la rapidité en lecture et en écriture des mémoires. Or, les mémoires les plus rapides sont aussi les plus chères et ne conservent pas les données après extinction de l'alimentation. C'est pour cela qu'un ordinateur utilise toujours les deux types de mémoires précédents : l'un pour stocker

les informations à long terme, l'autre pour réaliser des opérations sur les données. Ainsi ces deux mémoires échangent en permanence des informations.

Le **disque dur SSD** ((Solid State Drive) est un nouveau type de disque dur qui n'utilise pas la même technologie que le disque dur classique. Ce disque dur utilise une mémoire **flash**, il est muni de composants électroniques pour stocker les données, sauf qu'à la différence de la RAM, les données restent inscrites sur le disque dur même si on éteint l'ordinateur ; comme les clés USB ou les cartes SD. Cette mémoire flash allie donc rapidité des mémoires vives au stockage des données hors tension des mémoires mortes.

## 1.3 Les niveaux de langages de programmation

Pour communiquer avec un ordinateur, il existe plusieurs niveaux de langages. Le langage de plus *haut niveau* est le langage de l'homme, et donc incompréhensible pour la machine. Le langage de plus *bas niveau* est celui qui peut être interprété par le processeur, mais incompréhensible pour l'homme.

Pour illustrer ces niveaux de langage, utilisons l'algorithme de Syracuse, qui crée une suite de nombres se terminant toujours par 4, 2 et 1. Nous écrirons cet algorithme à l'aide de différents langages, du plus haut niveau jusqu'au plus bas.

### 1.3.1 Le langage humain.

C'est le langage courant, du plus haut niveau possible. Lorsqu'un algorithme est écrit dans ce langage, on parle de *pseudo code*.

« Je prends un nombre entier,
tant qu'il est supérieur à un,
s'il est pair je le divise par deux,
s'il est impair je le multiplie par trois et j'ajoute un. »

FIGURE 1.2 – Le langage humain ou pseudo code

### 1.3.2 Les langages de haut niveau.

Un langage évolué de programmation est une notation conventionnelle destinée à formuler des algorithmes et produire des programmes informatiques qui les appliquent. D'une manière similaire à une langue naturelle, un langage évolué de programmation est fait d'un *alphabet, un vocabulaire, des règles de grammaire, et des significations*. Il existe de nombreux langages de programmation, voici deux exemples :

```c
int main()
{
        while (N > 1)
        {
                if (N % 2 == 0)
                {
                        N = N / 2;
                } else {
                        N = 3 * _N + 1;
                }
        }
        mainendloop: goto mainendloop;
}
```

```python
def syracuse(n):
    while n>1:
        if n%2==0:
            n=n/2
            print(n)
        else:
            n=3*n+1
            print(n)
    return n
```

FIGURE 1.3 – Langage C et langage python

### 1.3.3 L'assembleur.

Un langage d'assemblage ou *langage assembleur* est un *langage de bas niveau* qui représente le langage machine sous une forme lisible par un humain. Les combinaisons de bits du langage machine sont représentées par des symboles dits *mnémoniques*, faciles à retenir. Le programme assembleur convertit ces mnémoniques en langage machine en vue de créer par exemple un fichier objet ou un fichier exécutable.

```
000002a2 <main>:
 2a2:    14 be           out     0x34, r1 ; 52
 2a4:    1e c0           rjmp    .+60            ; 0x2e2
<main+0x40>
 2a6:    80 91 00 01     lds     r24, 0x0100
 2aa:    90 91 01 01     lds     r25, 0x0101
 2ae:    80 fd           sbrc    r24, 0
 2b0:    0a c0           rjmp    .+20            ; 0x2c6
<main+0x24>
 2b2:    80 91 00 01     lds     r24, 0x0100
 2b6:    90 91 01 01     lds     r25, 0x0101
 2ba:    62 e0           ldi     r22, 0x02       ; 2
 2bc:    70 e0           ldi     r23, 0x00       ; 0
 2be:    0e 94 78 01     call    0x2f0   ; 0x2f0 <__divmodhi4>
 2c2:    cb 01           movw    r24, r22
 2c4:    0a c0           rjmp    .+20            ; 0x2da
<main+0x38>
 2c6:    20 91 00 01     lds     r18, 0x0100
 2ca:    30 91 01 01     lds     r19, 0x0101
 2ce:    c9 01           movw    r24, r18
 2d0:    88 0f           add     r24, r24
 2d2:    99 1f           adc     r25, r25
 2d4:    82 0f           add     r24, r18
 2d6:    93 1f           adc     r25, r19
 2d8:    01 96           adiw    r24, 0x01       ; 1
 2da:    90 93 01 01     sts     0x0101, r25
 2de:    80 93 00 01     sts     0x0100, r24
 2e2:    80 91 00 01     lds     r24, 0x0100
 2e6:    90 91 01 01     lds     r25, 0x0101
 2ea:    02 97           sbiw    r24, 0x02       ; 2
 2ec:    e4 f6           brge    .-72            ; 0x2a6
<main+0x4>
 2ee:    ff cf           rjmp    .-2             ; 0x2ee
<main+0x4c>
```

FIGURE 1.4 – Un extrait de l'algorithme de Syracuse en assembleur

### 1.3.4 Le code machine.

Le langage machine, ou *code machine*, est la suite de bits qui est interprétée par le processeur d'un ordinateur exécutant un programme informatique. C'est le langage natif d'un processeur, c'est-à-dire le seul qu'il puisse traiter. Il est composé d'instructions et de données à traiter codées en binaire.

```
64656620 73797261 63757365 286E293A
0A202020 20776869 6C65206E 3E313A0A
20202020 20202020 6966206E 25323D3D
303A0A20 20202020 20202020 2020206E
3D6E2F32 0A202020 20202020 20202020
20707269 6E74286E 290A2020 20202020
2020656C 73653A0A 20202020 20202020
20202020 6E3D332A 6E2B310A 20202020
20202020 20202020 7072696E 74286E29
0A202020 20726574 75726E20 6E
```

FIGURE 1.5 – Exemple de fichier HEX, contenant du code machine

Chaque processeur possède son propre langage machine, dont un code machine qui ne peut s'exécuter que sur la machine pour laquelle il a été préparé. Le code machine est aujourd'hui généré automatiquement, généralement par le *compilateur* d'un langage de programmation.

### 1.3.5 Un historique des langages de programmation.

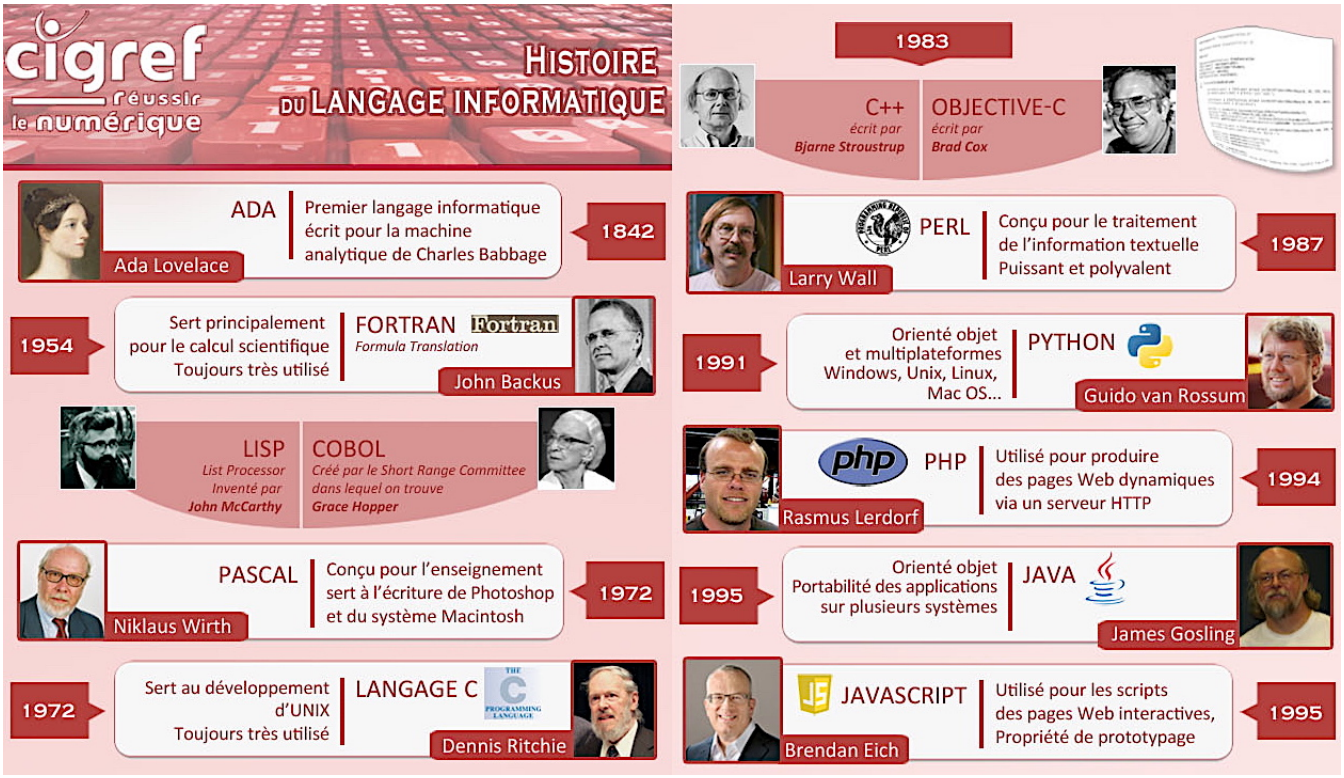



FIGURE 1.6 – Historique des langages de programmation

## 1.4 Le code compilé ou interprété.

Le langage de programmation est mis en œuvre par un traducteur automatique, soit par un *compilateur* comme pour le langage C, soit par un *interpréteur* comme pour le langage Python. Un compilateur est un programme informatique qui transforme dans un premier temps un code source écrit dans un langage de programmation donné en un code cible qui pourra être directement exécuté par un ordinateur, à savoir un programme en langage machine ou en code intermédiaire, tandis que l'interpréteur réalise cette traduction "à la volée".

CHAPITRE 2

# LA REPRÉSENTATION DES NOMBRES.

## 2.1 Représentation d'un nombre entier naturel

On peut donner l'écriture générale d'un nombre dans une base quelconque :

$$N_{(B)} = \sum_{k=0}^{n-1} a_k B^k = a_{n-1}B^{n-1} + \cdots + a_k B^k + \cdots + a_1 B^1 + a_0 B^0$$

où :
- $n$ est le nombre de chiffres composant $N$
- $a_k$ est le chiffre de rang $k$
- $B$ est la base

On trouve des bases $B$ de type décimale (Base 10), octale (Base 8), binaire (Base 2) et hexadécimale (Base 16).

***Exemple* 2.1** Ecriture d'un nombre naturel en base 10 :

$$\boxed{247_{(10)} = 2 \times 10^2 + 4 \times 10^1 + 7 \times 10^0}$$

où 2 est appelé "digit de poids fort" ou "Most Significant Digit" (MSD) et 7 "digit de poids faible" ou "Least Significant Digit" (LSD).

⋆ **En binaire : la base 2 :** Chaque bit prend une valeur 0 ou 1, le mot de 8 bits est l'octet, le mot de 4 bits est le quartet.

***Exemple* 2.2** Ecriture d'un nombre naturel en base 2 :

| | | | |
|---|---|---|---|
| $1111\ 0111_{(2)}$ | | = | $1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ |
| ↑ | ↑ | = | $128 + 64 + 32 + 16 + 0 + 4 + 2 + 1$ |
| MSB | LSB | = | $247_{(10)}$ |

*Remarque.* Phénomène d'**overflow** : Si on ajoute 10 à 247, on dépasse la capacité de 8 bits. En effet, on obtiendrait $247 + 10 = 257_{(10)} = 1 \times 2^8 + 1 \times 2^0 = 1\ 0000\ 0001_{(2)}$ ; mais comme seuls les 8 derniers bits sont gardés, on obtient 0000 0001, c'est-à-dire $1_{(10)}$.

⋆ **En hexadécimal : la base 16 :** On utilise un codage alphanumérique, qui fait correspondre à chaque nombre décimal compris entre 0 et 15 (donc codé sur 4 bits en base 2), un chiffre ou une lettre, suivant la table de correspondance ci-dessous.

| $N_{(10)}$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $N_{(16)}$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| $N_{(2)}$ | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

***Exemple* 2.3** Ecriture d'un nombre naturel en base 16

$$\boxed{\text{F7}_{(16)} = \$\text{F7} = 15 \times 16^1 + 7 \times 16^0 = 240 + 7 = 247_{(10)}}$$

***Exemple* 2.4** Conversion d'un nombre en base 2 en base 16.

$$\boxed{111100010111_{(2)} = \underbrace{1111}_{\text{F}}\ \underbrace{0001}_{1}\ \underbrace{0111}_{7} = \text{F17}_{(16)}}$$

⋆ **Le code BCD :** Le code BCD ("Binary Coded Decimal" ou "décimal codé binaire") permet de coder des nombres d'une façon relativement proche de la représentation usuelle (en base 10). Chaque chiffre (compris entre 0 et 9) du nombre est codé en binaire naturel (donc sur 4 bits). Bien que gourmand en mémoire, le code BCD est encore utilisé pour coder l'heure et la date dans le BIOS des PC par exemple. Ce code est plus fréquent en électronique, pour représenter les nombres sur des afficheurs (montres, calculatrices...)

***Exemple* 2.5** Codage BCD.

$$\boxed{2089_{(10)} = \left(0010\ 0000\ 1000\ 1001\right)_{(BCD)}}$$

Coder maintenant $2089_{(10)}$ en base 2 et comparer.

## 2.2 Représentation d'un nombre entier relatif.

**Représentation signée :** un entier relatif est un entier pouvant être négatif. Il faut donc coder le nombre de telle façon que l'on puisse savoir s'il s'agit d'un nombre positif ou d'un nombre négatif, et il faut de plus que les règles d'addition soient conservées. On utilise pour cela la méthode dite du *complément à 2* pour obtenir l'opposé d'un nombre positif ou nombre négatif CPL2.

***Exemple* 2.6**

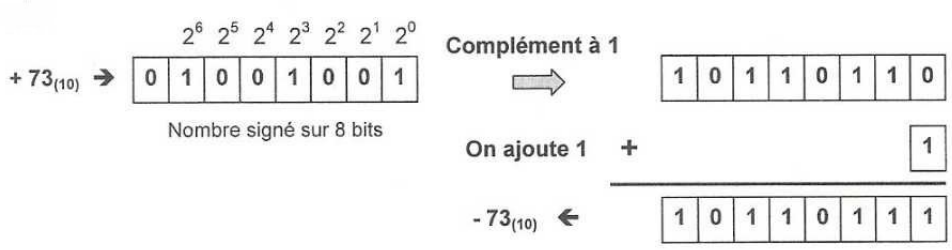



FIGURE 2.1 – Méthode du complément à 2

En effet, si

$$N = 0\times 2^7 + 1\times 2^6 + 0\times 2^5 + 0\times 2^4 + 1\times 2^3 + 0\times 2^2 + 0\times 2^1 + 1\times 2^0$$

on note $M$ le complément à 1, alors

$$M = 1\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 1\times 2^1 + 0\times 2^0$$

ainsi

$$N + M = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 2^8 - 1 = 256 - 1$$

et le complément à 2 est alors $M + 1 = 2^8 - N = 256 - N$ ; sur 8 bits, il représente donc bien $-N$.

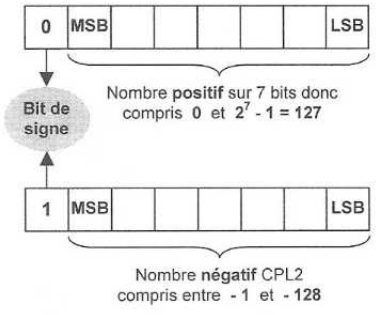



FIGURE 2.2 – Représentation d'un entier relatif en binaire signé

**Sur 8 bits :** On pourra coder un nombre de $-128$ à 127.

| Valeur en entier naturel | 0 à 127 | 128 à 255 |
|---|---|---|
| Valeur en entier relatif | 0 à 127 | $-128$ à $-1$ |

**Sur 16 bits :** On pourra coder un nombre de $-32768$ à 32767.

| Valeur en entier naturel | 0 à 32767 | 32768 à 65535 |
|---|---|---|
| Valeur en entier relatif | 0 à 32767 | $-32768$ à $-1$ |

D'une manière générale, sur $n$ bits, les entiers relatifs positifs commence par 0 et le plus grand entier relatif positif sera $2^{n-1} - 1$. Pour coder son opposé :

- ◇ on représente la valeur en base 2 sur $n-1$ bits,
- ◇ on complémente chaque bit (on inverse chaque bit),
- ◇ on ajoute 1,

***Exemple* 2.7** Coder en binaire sur 4 bits les nombres $0, 1, 2, \ldots 7$ ; les opposés $-1, -2, \ldots, -7$ ; puis $-8$.

Effectuer les opérations suivantes en binaire : addition de 3 et de 2 ; soustraction de 3 et de 2 ; addition de 55 et 42 ; soustraction de 55 et 42.

## 2.3 Représentation d'un nombre réel.

### 2.3.1 Les nombres à virgule fixe.

En décimal, on peut écrire facilement les nombres à virgule, en posant $0,1 = 10^{-1}$ ; $0,01 = 10^{-2}$... ; ainsi $3,625 = 3 \times 10^0 + 6 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3}$.

De la même façon, en binaire, 0,1 peut être noté $2^{-1}$ ; ainsi

$$11,101_{(2)} = 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 2 + 1 + \frac{1}{2} + 0 + \frac{1}{8} = 3,625_{(10)}$$

Cette notation étant comprise, reste le problème de la conversion des nombres. Si passer de l'écriture binaire à l'écriture décimale est assez simple, il est plus difficile de passer de l'écriture décimale à l'écriture binaire.

Une première possibilité consiste à coder de manière indépendante les parties entière et décimale.

Par convention, on détermine combien de bits représentent la part entière, et la part décimale du nombre. Par exemple, 8 bits pour la part entière, et 8 bits pour la part décimale.

***Exemple* 2.8** Le nombre 146,25 peut s'écrire *:*

| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | $2^{-5}$ | $2^{-6}$ | $2^{-7}$ | $2^{-8}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Partie entière = 146 | | | | | | | | Partie décimale = 0,25 | | | | | | | |

***Exemple* 2.9** Utiliser la méthode de virgule fixe pour coder 9,375.

Ce codage n'est malheureusement pas optimal, car il ne permet pas de coder des parties décimales très petites, par exemple, avec 8 bits pour la partie décimale, comme $2^{-8} = 0,00390625$; on peut donc pas coder 0,00001.

### 2.3.2 Les nombres à virgule flottante.

Si on veut coder le nombre $1101,1010_{(2)}$ on peut écrire 1101 1010 et décider de décaler de 4 chiffres pour avoir la virgule. Mais pour $0,000\ 000\ 000\ 111_{(2)}$, 8 bits ne suffiraient pas pour coder la partie décimale, car les 8 premières décimales sont des 0.

Pour optimiser le codage, il suffit de supprimer les 0 inutiles et de garder les chiffres caractéristiques, ici 111, qu'on appelle **mantisse**, et on nomme **exposant** le nombre de chiffres avant la mantisse, ici 10 ; on peut écrire de manière scientifique

$$0,000\ 000\ 000\ 111 = 1 \times 2^{-10} + 1 \times 2^{-11} + 1 \times 2^{-12} = 1,11 \times 2^{-10}\text{ ; l'exposant vaut } -10 \text{ et}$$

$$1101,1010 = 1,1011\ 0100 \times 2^3\text{ ; l'exposant vaut 3}$$

L'exposant doit donc pouvoir être positif (lorsque la virgule se situe après le début de la mantisse) ou négatif (lorsque la virgule se situe avant le début de la mantisse).

La norme IEEE 754 définit ainsi la façon de coder un nombre réel. Cette norme se propose de coder le nombre sur 32 bits (simple précision) ou sur 64 bits (double précision) et définit trois composantes :

- S qui représente le **signe** du nombre,
- E qui représente **l'exposant**,
- M qui représente **la mantisse**.

**Sur 32 bits (simple précision) :**

- ◇ 1 bit pour le signe : S = 0 pour un nombre positif, S = 1 pour un nombre négatif.
- ◇ Les 8 bits suivants pour l'exposant E.
  L'exposant peut être positif ou négatif. La représentation des nombres signés (complément à 2) rendant la comparaison entre les nombres flottants difficile, on préfère décaler l'exposant, c'est-à-dire lui soustraire $127_{(10)}$. Cet exposant varie donc entre $-126$ et 127.
  $255-127=128$ est réservé pour coder $\pm\infty$ (si la mantisse est nulle) ou un NaN, Not a Number (si la mantisse est non nulle) ; $0-127=-127$ est réservé pour coder 0 (avec la mantisse nulle).
- ◇ Les 23 suivants pour la mantisse M.
  Cette mantisse est le nombre décimal qui intervient dans l'écriture scientifique d'un nombre ; le premier chiffre n'est jamais nul (sauf pour le nombre 0), donc en binaire le premier chiffre est 1 ; il est implicite, donc non codé. On rajoute ce 1 devant les 23 bits pour reconstituer la mantisse.

Ainsi le codage se fait sous la forme suivante :

SEEEEEEEEMMMMMMMMMMMMMMMMMMMMMMM

Le nombre pourra ainsi s'écrire :

$$\boxed{(-1)^{S}\times(1M)\times 2^{E-127}}$$

**Sur 64 bits (double précision) :**

- ◇ 1 bit pour le signe S, S = 0 pour un nombre positif, S = 1 pour un nombre est négatif,
- ◇ les 11 bits suivants pour l'exposant E, valeur à laquelle on doit soustraire $1023_{10}$, l'exposant variant donc de $-1022$ à 1023,
- ◇ les 52 suivants pour la mantisse M (plus 1 bit implicite).

Ainsi le codage se fait sous la forme suivante :

SEEEEEEEEEEEMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM

Le nombre pourra ainsi s'écrire :

$$\boxed{(-1)^{S}\times(1M)\times 2^{E-1023}}$$

***Exemple* 2.10**

a) Soit

0100 0000 1011 1000 0000 0000 0000 0000

codé comme un réel à virgule flottante sur 32 bits. On obtient :

- ◇ S = 0, donc le nombre est positif ;
- ◇ E = 1000 0001$_{(2)}$, donc $129_{(10)}$, auquel on soustrait 127, donc E = 2 ;
- ◇ La mantisse vaut **1**011 1000 0000 0000 0000 0000

La représentation du nombre en notation scientifique binaire est donc :

$$(-1)^{0}\times 1{,}0111\times 2^{2}$$

En écriture décimale, cela donne :

$$(-1)^{0}\left(1\times 2^{0}+0\times 2^{-1}+1\times 2^{-2}+1\times 2^{-3}+1\times 2^{-4}\right)\times 2^{2}$$

$$=(1+0{,}25+0{,}125+0{,}0625)\times 4$$

$$=4+1+0{,}5+0{,}25$$

$$=5{,}75$$

b) Encoder en norme IEEE 754 avec simple précision le nombre 16,5.

### 2.3.3 Limites de la représentation des réels.

Même si on peut choisir une précision de plus en plus forte, la représentation des nombres reste limitée. Lorsque le nombre de bits utilisés est insuffisant, cela conduit au phénomène d'*overflow*.

Un manque de précision peut également conduire à un résultat faux.

***Exemple* 2.11** Peut-on coder exactement le nombre décimal $0,1$ ?

Voici un test en python où les nombres sont codés en double précision :

```
>>> 0.1+0.1+0.1==0.3
False
```

***Exemple* 2.12** Considérons l'équation

$$x^2 + 1,4x + 0,49 = 0$$

Le discriminant vaut $\Delta = 1,4^2 - 4 \times 1 \times 0,49 = 0$ et on conclut que l'équation possède une unique solution; cependant, lorsqu'on calcule ce discriminant avec des flottants, on obtient

-2.22044604925e-16

qui est strictement négatif!!!

***Exemple* 2.13** Pour calculer $\pi$ Archimède utilise des polygones à $2^n$ côtés inscrits dans le cercle unité et dont il calcule le périmètre. Lorsque $n$ devient très grand le demi-périmètre tend vers $\pi$.

Voici un programme Python correspondant ...

```
import numpy as np

#la bibliothèque numpy contient les fonctions mathématiques usuelles comme racine carrée
                                                ("square root") ; on importe la
                                               bibliothèque sous l'alias np


a=np.sqrt(2)
n=4
while a>0.0000000001:
  a=np.sqrt(2-2*np.sqrt(1-a*a/4))
  n=n*2
  P=(n/2)*a
  print(P)
```

... et le résultat!!!

| | | | |
|---|---|---|---|
| 3.06146745892 | 3.14158772528 | 3.14159260738 | 3.14245127249 |
| 3.12144515226 | 3.1415914215 | 3.14159291094 | 3.14245127249 |
| 3.13654849055 | 3.14159234561 | 3.1415941252 | 3.16227766017 |
| 3.14033115695 | 3.14159257655 | 3.1415965537 | 3.16227766017 |
| 3.14127725093 | 3.14159263346 | 3.1415965537 | 3.46410161514 |
| 3.14151380114 | 3.14159265481 | 3.14167426502 | 4.0 |
| 3.14157294037 | 3.14159264532 | 3.14182968189 | 0.0 |

## 2.4 Représentation d'un caractère : le code ASCII

La mémoire de l'ordinateur conserve toutes les données sous forme numérique, codées en binaire. Il n'existe pas de méthode pour stocker directement les caractères. Chaque caractère possède donc son équivalent en code numérique : c'est le **code ASCII** (American Standard Code for Information Interchange).

Le code ASCII standard permet de coder des caractères alphanumériques de l'alphabet latin sur 7 bits ; c'est-à-dire 128 caractères disponibles, de 0 à 127. Par exemple, le caractère "a" est associé à "01100001" et "A" est associé à "01000001". Les sept bits utiles sont précédés d'un "0" car les ordinateurs travaillent sur des multiples de huit bits (multiples d'un octet).

La norme ASCII permet ainsi à toutes sortes de machine de stocker, analyser et communiquer de l'information textuelle. En particulier, la quasi totalité des ordinateurs personnels utilisent l'encodage ASCII.

Certains caractères n'ont pas vocation à être affichés, mais correspondent à des commandes de contrôle. Par exemple le code "00001010" permet d'aller à la ligne.

Cependant, le code ASCII standard est limité car il ne peut pas encoder certains caractères, comme les caractères accentués par exemple. Le code a été créé pour encoder des textes en anglais !

Le codage ASCII peut donc être complété par des codes utilisant le huitième bit ; les caractères de 128 à 255 pour les accents, par exemple. Mais ces codes diffèrent d'un pays à l'autre ! Pour permettre une communication internationales ces normes complémentaires ont été unifiées en un code appelé **Unicode**.

Le problème de l'Unicode est qu'il peut comporter un million de caractères, couvrant 100 écritures. Généralement, en Unicode, un caractère prend 2 octets ; autrement dit, un texte prend deux fois plus de place qu'en ASCII ! De plus, si on prend un texte en français, la grande majorité des caractères utilisent seulement le code ASCII.

Dans la pratique, le codage le plus couramment utilisé est l'**UTF-8**. Cette norme est une extension du code ASCII, utilisant le huitième bit : chaque caractère est codé par une séquence d'octets.

Le principe est simple : tout caractère ASCII se code de la même manière en UTF-8 ; et dès qu'on a besoin d'un caractère UNICODE (non ASCII), on utilise un caractère spécial signalant "attention, le caractère suivant est en Unicode". Par exemple, pour le texte "polynôme du second degré", seuls le "ô" et le "é" ne sont pas dans la table ASCII. On écrit en UTF-8 :

polynÃ´me du second degré

ANNEXE - TABLE ASCII.

| Code décimal | Caractère ASCII | Description | Décimal | Caractère | Décimal | Caractère | Décimal | Caractère |
|---|---|---|---|---|---|---|---|---|
| 0 | NUL | Null | 32 | Space | 64 | @ | 96 | ‘ |
| 1 | SOH | Start of heading | 33 | ! | 65 | A | 97 | a |
| 2 | STX | Start of text | 34 | ” | 66 | B | 98 | b |
| 3 | ETX | End of text | 35 | # | 67 | C | 99 | c |
| 4 | EOT | End of transmission | 36 | $ | 68 | D | 100 | d |
| 5 | ENQ | Enquiry | 37 | % | 69 | E | 101 | e |
| 6 | ACQ | Acknowledge | 38 | & | 70 | F | 102 | f |
| 7 | BEL | Bell | 39 | ’ | 71 | G | 103 | g |
| 8 | BS | Backsapce | 40 | ( | 72 | H | 104 | h |
| 9 | TAB | horizontal tab | 41 | ) | 73 | I | 105 | i |
| 10 | LF | New line feed, new line | 42 | * | 74 | J | 106 | j |
| 11 | VT | Vertical tab | 43 | + | 75 | K | 107 | k |
| 12 | FF | NP form feed, new page | 44 | ‘ | 76 | L | 108 | l |
| 13 | CR | Carriage return | 45 | - | 77 | M | 109 | m |
| 14 | SO | Shift out | 46 | . | 78 | N | 110 | n |
| 15 | SI | Shift in | 47 | / | 79 | O | 111 | o |
| 16 | DLE | Data link espace | 48 | 0 | 80 | P | 112 | p |
| 17 | DC1 | Device control 1 | 49 | 1 | 81 | Q | 113 | q |
| 18 | DC2 | Device control 2 | 50 | 2 | 82 | R | 114 | r |
| 19 | DC3 | Device control 3 | 51 | 3 | 83 | S | 115 | s |
| 20 | DC4 | Device control 4 | 52 | 4 | 84 | T | 116 | t |
| 21 | NAK | Negative acknowledge | 53 | 5 | 85 | U | 117 | u |
| 22 | SYN | Synchronous idle | 54 | 6 | 86 | V | 118 | v |
| 23 | ETB | End of trans. block | 55 | 7 | 87 | W | 119 | w |
| 24 | CAN | Cancel | 56 | 8 | 88 | X | 120 | x |
| 25 | EM | End of medium | 57 | 9 | 89 | Y | 121 | y |
| 26 | SUB | Substitute | 58 | : | 90 | Z | 122 | z |
| 27 | ESC | Escape | 59 | ; | 91 | [ | 123 | { |
| 28 | FS | File separator | 60 | < | 92 | \ | 124 | \| |
| 29 | GS | Group separator | 61 | = | 93 | ] | 125 | } |
| 30 | RS | Record separator | 62 | > | 94 | ^ | 126 | ~ |
| 31 | US | Unit separator | 63 | ? | 95 | _ | 127 | DEL |

FIGURE 2.3 – Table ASCII

CHAPITRE 3

# ALGORITHMIQUE ET PROGRAMMATION

## 3.1 Introduction.

Un **algorithme** est une suite finie d'opérations élémentaires permettant d'effectuer un calcul ou de résoudre un problème donné de manière automatique. En mathématiques, nous connaissons, par exemple, l'algorithme de résolution d'une équation du second degré ou l'algorithme du pivot de Gauss ; on peut retrouver le principe dans d'autres domaines, par exemple, la réalisation d'une recette de cuisine peut être considérée comme un algorithme.

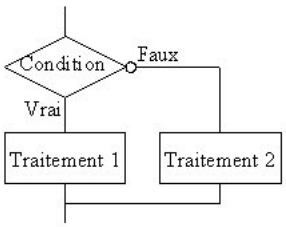
L'**algorithmique** désigne l'ensemble des règles et des techniques qui sont impliquées dans la définition et la conception d'algorithme.

Un **programme** est une suites d'instructions pouvant être exécutées par un ordinateur. Le programme est donc la traduction en langage compréhensible par la machine d'un algorithme.

Pour rédiger un algorithme, il existe deux modes de représentations :

- **l'organigramme** ou algorithme graphique :

  L'organigramme ci-contre représente une structure alternative. Si la condition 1 est vérifiée, alors on effectue le traitement 1 ; sinon, on effectue le traitement 2.




- **le pseudo-code** ou algorithme textuel.
  Le peudo-code est une façon de décrire un algorithme sans référence à un langage de programmation particulier. Il ressemble cependant à un langage de programmation, mais sans les problèmes de syntaxe. C'est la représentation que nous allons choisir.

  ***Exemple* 3.1** On considère l'algorithme de Syracuse qui consiste à demander un nombre entier non nul ; si celui-ci est pair, on le divise par deux, s'il est impair, on le multiplie par 3 et on ajoute 1. En répétant ainsi l'opération on obtient une suite de nombres, appelée suite de Syracuse.
  La conjecture de Syracuse est l'hypothèse mathématique selon laquelle la suite se termine toujours par 4, 2 et 1.

En pseudo-code, on peut écrire :

**Variable** $n$ un Entier
**Début**
**Ecrire** ('Entrez la valeur de $n$')
**Lire** $n$
**Tant que** $n > 1$
**Si** $n$ est pair
$n \longleftarrow n/2$
**Sinon**
$n \longleftarrow 3 \times n + 1$
**Ecrire** $n$
**Fin**

En langage *Python* cela se traduit par :

```
n=input('Entrer une valeur entière : ')
n=int(n)
while n>1:
    if n%2==0:
        n=n/2
    else:
        n=3*n+1
print(n)
```

On remarque que *Python* interprète **dynamiquement** le type de variables ; c'est-à-dire qu'il peut reconnaître le type de variable et donner l'information à l'ordinateur.

## 3.2 Affectation des variables.

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier) ; il peut aussi s'agir de résultats obtenus par le programme, intermédiaires ou définitifs. Ces données peuvent être de plusieurs types : des nombres (entiers de type `int`, réels ou flottants de type `float`, complexes de type `complex`), du texte ou chaîne de caractères (string de type `str`), des booléens (de type `bool`), des listes (de type `list`)... Toujours est-il que dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une **variable**. Une variable est en quelque sorte une **boîte**, que le programme (l'ordinateur) va repérer par une **étiquette**. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette. Dans l'ordinateur, physiquement, il y a un emplacement de mémoire dédié à chaque variable, repéré par une adresse binaire.

**Exercice 3.1** *Ecrire un algorithme permettant d'échanger les contenus de deux variables A et B*

**Exercice 3.2** *Ecrire un algorithme permettant de transférer à B la valeur de A, à C la valeur de B et à A la valeur initiale de C.*

**Incrémentations.** Dans un programme, on a souvent besoin d'*incrémenter* une variable `a`, c'est-à-dire remplacer `a` par `a+1` par exemple. L'instruction `a=a+1` peut être remplacer par `a+=1`. Voici un tableau de correspondances :

| | | |
|---|---|---|
| `a+=b` | `a=a+b` | `incrémentation de b unités` |
| `a-=b` | `a=a-b` | `décrémentation de b unités` |
| `a*=b` | `a=a*b` | `multiplication de a par b` |
| `a/=b` | `a=a/b` | `division de a par b` |
| `a//=2` | `a=a//2` | `division entière de a par 2` |
| `a%=2` | `a=a%2` | `reste de la division de a par 2` |

## 3.3 Fonctions.

Un programme peut être écrit sous forme de script, il s'agit alors d'une suite d'instructions et commandes destinées à effectuer des opérations conduisant au résultat souhaité. On peut également définir une **fonction** (au sens informatique) dont le but est d'appeler un certain nombre d'**arguments** pour fournir en sortie le résultat.

Par exemple, voici une fonction qui permet le calcul du discriminant d'un polynôme du second degré :

```
def delta(a,b,c):
    return b**2-4*a*c
```

Le **nom** de la fonction est `delta`, les **arguments** `a`, `b` et `c` sont entre parenthèses, les instructions nécessaires à cette fonction sont indentées après le " : ".
L'instruction `return` donne et garde en mémoire le résultat de la fonction.
L'instruction `print` permet uniquement l'affichage du résultat, et alors, la fonction `delta` ne peut pas intervenir dans un autre programme.
Il est fortement conseillé d'ajouter des **commentaires** à sa fonction pour qu'elle soit compréhensible par tout utilisateur :

```
def delta(a,b,c):
  '''
  * entrée : float ; a, b et c sont des réels
  * sortie : float ; la fonction renvoie le discriminant
  '''
    return b**2-4*a*c
```

On peut également utiliser des **annotations** pour préciser le type des paramètres attendus et type de la variable retournée :

```
def delta(a:float,b:float,c:float)->float:
    return b**2-4*a*c
```

La **signature** de la fonction regroupe le nom de la fonction, le type de ses arguments et le type de la variable renvoyée. Pour la fonction `delta`, la signature est :

`delta(float,float,float) -> float`

Les variables *a*, *b* et *c* qui interviennent dans la définition de la fonction sont **locales**.
Par exemple :

```
>>> a=2
>>> delta(1,2,1)
0
>>> print(a)
2
```

On peut décider de rendre une variable **globale** en le spécifiant :

```
def f():
    global y
    y=12*x
    return y
```

Voici le résultat de l'exécution de ce programme :

```
>>> x=7
>>> y=1
>>> f()
84
>>> y
84
```

⚠ Dans la mesure du possible on évitera l'utilisation de variables globales !

La **portée** d'une variable provient directement de l'endroit où elle a été créée. Une variable locale créée dans une fonction n'aura plus de visibilité lorsqu'on aura quitté l'espace de cette fonction. Il existe trois portées :

- ◇ **L** : la portée locale, qui est explorée en premier. Cet espace de nom est effacé dès qu'on quitte le fonction.
- ◇ **G** : la portée globale, explorée ensuite. Cet espace contient les nom globaux du module courant.
- ◇ **I** : la portée interne, explorée en dernier. Cet espace contient les noms, variables et fonctions intégrés à *Python* comme : `sqrt`, `abs`, `len`...

## 3.4 Instructions conditionnelles.

Il existe des instructions pour permettre à la machine de suivre une liste d'instructions ou une autre. Une telle procédure s'appelle **test** ou une **structure alternative** ou encore **structure conditionnelle**.

Il existe essentiellement deux formes d'instructions conditionnelles :

**Si** booléen **Alors**
    Instructions
**FinSi**

**Si** booléen **Alors**
    Instructions 1
**Sinon**
    Instructions 2
**FinSi**

On rappelle qu'un **booléen** est une expression qui ne prend que deux valeurs : VRAI ou FAUX.

***Exemple* 3.2**

**Si** $x \geqslant 0$ **Alors**
    **Ecrire** $\sqrt{x}$
**FinSi**

***Exemple* 3.3**

**Si** $x > 0$ **Alors**
    **Ecrire** ('le nombre',$x$,'est strictement positif')
**Sinon**
    **Ecrire** ('le nombre',$x$,'est négatif')
**FinSi**

En langage Python, en utilisant une fonction, cela se traduit par :

```
def signe(x):
    if x>0:
        return 'le nombre',x,'est strictement positif'
    else:
        return 'le nombre',x,'est negatif'
```

**Exercice 3.3** *Ecrire un algorithme qui demande deux nombres et les redonne dans l'ordre croissant.*

**Exercice 3.4** *Ecrire un algorithme qui demande deux nombres et qui donne le signe (au sens large) de leur produit ; sans calculer le produit des deux nombres.*

Une instruction conditionnelle peut graphiquement se représenter comme un arbre de probabilités à deux branches. Lorsque deux branches ne suffisent pas à faire l'inventaire de tous les résultats possibles, on a recours aux **tests imbriqués**.

***Exemple* 3.4** Voici un algorithme qui donne l'état de l'eau en fonction de la température de celle-ci :

**Variable** $Temp$ **en Entier**
**Début**
**Écrire** "Entrez la température de l'eau :"
**Lire** $Temp$
**Si** $Temp <= 0$ **Alors**
    **Écrire** " C'est de la glace"
**Sinon**
    **Si** $Temp < 100$ **Alors**
        **Écrire** " C'est du liquide"
    **Sinon**
        **Écrire** " C'est de la vapeur"
    **FinSi**
**FinSi**
**Fin**

**Exercice 3.5** *Etant donnés trois nombres a, b et c ; écrire une fonction python qui retourne leur minimum.*

Lors de l'utilisation de tests imbriqués on peut fusionner le **Sinon** suivi du **Si** en un seul **SinonSi**.

***Exemple* 3.5** Suivant les valeurs de $x$, on souhaite déterminer l'expression de la fonction

$$f : x \mapsto |x| - 2|x-1| + |x-2|$$

**Définition** de la fonction $f(x)$
**Si** $x < 0$ **Alors**
    $y = 0$
**SinonSi** $(x \geqslant 0)$ et $(x < 1)$ **Alors**
    $y = 2x$
**FinSi**
**SinonSi** $(x \geqslant 1)$ et $(x < 2)$ **Alors**
    $y = -2x + 4$
**FinSi**
**Sinon**
    $y = 0$
**FinSi**
**Ecrire** $y$
**Fin**

```
def f(x):
    if x<0:
        y=0
    elif (x>=0)&(x<1):
        y=2*x
    elif (x>=1)&(x<2):
        y=-2*x+4
    else:
        y=0
    return y
```

## 3.5 Instructions itératives.

Lorsqu'une séquence d'instructions doit être répétée "en boucle", on a recours à une **structure itérative**. Il existe deux sortes de telles structures :

- ◇ la boucle **inconditionnelle**, ou boucle **Pour** en pseudo-code, ou encore boucle `for` en langage *Python*; cette boucle suppose que l'on connaisse le nombre d'itérations au départ;
- ◇ la boucle **conditionnelle**, ou boucle **TantQue** en pseudo-code, ou encore boucle `while` en langage *Python*; **TantQue** est suivi d'un booléen : si celui-ci est VRAI alors le programme "entre" dans la boucle et suit les instructions jusqu'au **FinTantQue** et il retourne ensuite sur la ligne **TantQue**; la boucle s'arrête lorsque le booléen prend la valeur FAUX.

Comparons l'utilisation de ces deux structures sur l'exemple suivant :

***Exemple* 3.6** On souhaite calculer la somme des entiers naturels jusqu'à $N$, où $N$ est une valeur entrée par l'utilisateur.

**Variables** $N$, $k$ et $somme$ en **Numérique**
**Début**
**Ecrire** ('Entrez la valeur de $N$ : ')
**Lire** $N$
$somme \leftarrow 0$
**Pour** $k$ **de** 1 **à** $N$
  $somme \leftarrow somme + k$
**Suivant**
**Ecrire** ('La somme des $N$ entiers naturels est',$somme$)
**Fin**

**Variables** $N$, $k$ et $somme$ en **Numérique**
**Début**
**Ecrire** ('Entrez la valeur de $N$ : ')
**Lire** $N$
$somme \leftarrow 0$
$k \leftarrow 0$
**TantQue** $k \leqslant N$
  $somme \leftarrow somme + k$
  $k \leftarrow k + 1$
**FinTantQue**
**Ecrire** ('La somme des $N$ entiers naturels est',$somme$)
**Fin**

```
'''somme des entiers naturels
boucle for et boucle while'''
def somme_entiers(n):
    somme=0
    for k in range(1,n+1):
        somme=somme+k
    return somme
```

```
def somme_entiers(n):
    somme=0
    k=0
    while k<=n:
        somme=somme+k
        k=k+1
    return somme
```

*Remarques.*

- ◇ Les boucles `for` et `while` peuvent donc être utilisées indifféremment dans la plupart des situations itératives.
- ◇ La boucle `for` nécessite de connaître le nombre d'itérations, mais elle a l'avantage d'éviter de programmer la progression de la variable.
- ◇ ⚠ Avec une boucle `while` il faut s'assurer que le booléen puisse être VRAI, car dans le cas contraire le programme n'entrera jamais dans la boucle! Il faut également s'assurer que le booléen puisse être FAUX, car sinon le programme ne sort jamais de la boucle!!! On parle de boucle **infinie**.
- ◇ Dans l'instruction `for k in range(1,n)` la variable `k` va prendre toutes les valeurs comprises entre 1 et $n-1$ ⚠ avec un pas de 1. Pour changer le pas on peut utiliser une boucle `while` ou `range(1,n,2)` pour un pas de 2, `range(1,n,3)` pour un pas de 3... Mais dans une boucle `for` il est interdit de changer la valeur de la variable qui sert de compteur à l'intérieur de la boucle!

**Exercice 3.6** *Adapter l'un des algorithmes précédents pour calculer la somme des n premiers carrés d'entiers, la somme des n premiers inverses de carrés d'entier, la factorielle de n.*

**Exercice 3.7** ***Le numéro gagnant.***

*Le nombre 7 est gagnant. Ecrire un algorithme qui demande à l'utilisateur de choisir un nombre entre 1 et 10 et qui donne comme réponse "recommence!" si le nombre choisi est différent de 7 et jusqu'au moment ou le numéro gagnant est choisi. Là le jeu s'arrête et l'ordinateur affiche "c'est gagné!".*

**Exercice 3.8** *Ecrire une fonction python* `somme_liste` *qui prend en argument une liste de nombres et qui retourne la somme des termes de cette liste.*

**Exercice 3.9** *Ecrire une fonction python* `maxi_liste` *qui prend en argument une liste de nombres et qui retourne le maximum de cette liste.*

**Exercice 3.10** *Donner le résultat à l'issue des programmes suivants :*

```
resultat = ""
for c in "Bonsoir" :
    resultat = resultat + c
    print (resultat)
```

```
resultat = ""
for c in "Bonsoir" :
    resultat = resultat + c
print (resultat)
```

Un même programme peut utiliser plusieurs boucles ; celles-ci peuvent être **imbriquées** ou **successives** ; le résultat ne sera évidemment pas le même.

**Exercice 3.11** *Donner le résultat à l'issue des deux boucles suivantes :*

```
for i in range (1,5):
    print('il est passe par ici')
    for j in range (1,3):
        print('il repassera par la')
```

```
for i in range (1,5):
    print('il est passe par ici')
for j in range (1,3):
    print('il repassera par la')
```

**Exercice 3.12** *Ecrire un programme capable de donner la table de multiplication.*

## 3.6 Types de variables.

### 3.6.1. Types numériques

En mathématiques, les nombres appartiennent à des ensembles tels que $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ ou $\mathbb{C}$. En Python (comme dans beaucoup de langages informatiques) les nombres seront du type :

**int (integer)** correspond à un entier (de $\mathbb{N}$ ou $\mathbb{Z}$) stocké sur une partie limitée de la mémoire de l'ordinateur. Il ne peut donc pas être aussi petit (négatif) ou aussi grand que l'on veut. Pour une machine (dépend du microprocesseur et du système d'exploitation) de 32 bits (4 octets), un entier sera compris entre $-2^{31}$ et $2^{31}-1$ (soit entre −2 147 483 648 et 2 147 483 647). Pour une machine de 64 bits, un entier sera compris entre $-2^{63}$ et $2^{63}-1$

**long (long integer)** le type entier long est équivalent au type entier sauf qu'il n'est pas limité à nombre d'octets (ou de bits prédéterminé). Á l'extrême, un nombre peut prendre la quasi totalité de la mémoire de l'ordinateur. Il est clair alors que ce nombre peut être considérable.

**float** le type flottant permet de représenter des nombres à virgule. Il est codé en mémoire sur 32 bits (simple précision) ou sur 64 bits (double précision). Il est représenté sous la forme

$$\text{nombre} = (-1)^{S}1\text{mantisse} \times 2^{\text{exposant}}$$

Sur 64 bits la mantisse est écrite avec 52 chiffres binaires (51 plus le 1 implicite), et l'exposant avec 11 chiffres ; il y a 1 bit de signe. La précision maximale est donc de $2^{-52}$, soit environ $2 \times 10^{-16}$. Des valeurs spéciales permettent de représenter $-\infty$, $+\infty$ et `Nan`, *not a number*, souvent issu d'une forme indéterminée.

**complex** le type complexe correspond à une structure naturellement composée de deux flottants (partie réelle et imaginaire) sur $2\times 8 = 16$ octets. Par exemple, le nombre complexe $z = 2+3i$ s'écrit en Python sous la forme `2+3j`, ou `complex(2,3)`.

**bool** le type booléen correspond à l'algèbre booléenne et ne prend que deux valeurs : True/False. Il est codé sur un bit.

### 3.6.2. Listes, chaînes, tuples et dictionnaires

Il est souvent utile de collecter plusieurs données sous un même nom, et d'y accéder à travers un indice numérique. Sous Python on dispose de telles structures appelées **liste** (dans d'autres langages on dira **tableau**), au format `list` ; ou **chaîne de caractères**, au format `str`.

Si $n$ est la longueur (`len`) de la liste ou la chaîne, alors les indices des éléments sont compris entre 0 et $n-1$.

Ainsi, `L` = $[1, a, 'Bob']$ est une liste contenant trois valeurs : celle d'indice 0 est le nombre 1 (`L[0]=1`), celle d'indice 1 est la valeur de la variable `a` (`L[1]=a`) et celle d'indice 2 est la chaîne de caractères (ou mot) `'Bob'` (`L[2]='Bob'`).

L'appel `L[3]` renvoie le message : `list index out of range`.

Pour récupérer la dernière valeur d'une liste, on peut utiliser l'instruction `L[len(L)-1]`, ou plus simplement `L[-1]`.

De la même façon, `s='code123'` est une chaîne de caractères contenant 7 valeurs :

`s[0]='c'`, `s[6]=s[-1]='3'`, sont au format `str` ; et `int(s[5])=2` est un nombre entier.

La liste vide s'écrit [ ] et la chaîne vide s'écrit '', ou "".

*Opérations sur les listes* :

| Liste | Opération | Chaîne |
|---|---|---|
| `len(L)` | donne la longueur | `len(s)` |
| `L[i]` | renvoie le $i+1$ ième élément | `s[i]` |
| `a in L` | vérifie si l'élément se trouve dans `L` ou dans `s` | `'a' in s` |
| `L+M` | concatène (juxtapose) les listes ou les chaînes | `s+t` |
| `L.append(a)` | ajoute `a` à la fin de la liste ou de la chaîne | `s=s+'a'` |
| `L.insert(i,a)` | ajoute l'élément `a` à la position d'indice `i` | Ø |
| `del L[i]` | supprime l'élément d'indice `i` | Ø |
| `L.reverse()` | remplace la liste par celle d'ordre inverse | Ø |
| `L.index(a)` | retourne l'indice du premier `a` de la liste ou de la chaîne | `s.index('a')` |
| `list.sort(L)` | retourne la liste triée | Ø |

⚠ L'exemple suivant montre une exception dans l'affectation des variables concernant les listes (mais pas les chaînes!) :

```
>>>  L=[1,2,3]
M=L
>>> M.append(4)
>>> print(M)
[1, 2, 3, 4]
>>> print(L)
[1, 2, 3, 4]

>>>  s='abc'
t=s
>>> t=t+'d'
>>> print(t)
abcd
>>> print(s)
abc
```

*Création rapide d'une liste. Copie d'une liste.*

Il est possible de créer rapidement une liste dont on connaît la progression des éléments. Par exemple la commande `T = [k for k in range(5)]` renvoie la liste $[0,1,2,3,4]$.

Dans le script suivant, quelle est la différence entre les variables `U` et `V` ?

```
T=[k for k in range(5)]
U=T
V=[k for k in T]
```

**Exercice 3.13** *Ecrire une fonction Python* `addition_listes(L,T)` *qui prend en arguments deux listes* `L` *et* `T` *de la même taille et qui retourne la liste constituée de la somme des termes de* `L` *et* `T`.

*De même écrire une fonction* `multiplication_listes(L,T)` *qui retourne la liste constituée du produit des termes de* `L` *et* `T`.

**Exercice 3.14** *Une liste* `V` *contient des valeurs dont les poids sont contenus dans une liste* `P`. *Ecrire une fonction Python* `moyenne_pondérée(V,P)` *qui retourne la moyenne des valeurs de la série* `V` *pondérées par les poids de* `P`.

*Modifier la fonction précédente pour créer une fonction* `variance(V,P)`.

*Extraction des éléments d'une liste ou d'une chaîne.*

Il est possible d'extraire une partie d'une liste, ou d'une chaîne en indiquant les indices des éléments à conserver.

```
>>>L=[1,2,3,4,5,6]
>>>L[1:4]
[2, 3, 4]
>>>L[3:]
[4, 5, 6]
>>>L[:2]
[1, 2]

>>> s='code123'
>>>s[4:]
'123'
>>>s[:4]
'code'
```

Si la liste `L` contient des caractères ASCII, il est possible de retourner la valeur décimale de chaque caractère par la commande `ord('string')` et la commande réciproque est `chr(int)` :

```
>>>L=['a','&','%','0']
>>>ord(L[0])
97
>>>ord('&')
38
>>>chr(98)
'b'
>>>chr(37)
'%'
```

**Exercice 3.15** *Boucle for et boucle while : extraction des admis à un concours* `admis(L,b)`...

**Exercice 3.16** *Avec les nombres binaires, une opération possible est le Ou Exclusif. Cette opération consiste à additionner les nombres bit à bit avec la table de vérité suivante :*

$$0+0=0 \quad ; \quad 1+0=1 \quad ; \quad 0+1=1 \quad ; \quad 0+1=1 \quad ; \quad 1+1=0$$

*Par exemple, si* `A=10101010 et B=11001100` *alors :* `A OuEx B = 01100110`.

*En python cette opération est codée :* `A ^ B`.

*Ecrire une fonction* `OuEx(A:list,B:list)-> list` *qui réalise cette opération.*

*Par exemple :* `OuEx([1,0,1,0,1,0,1,0],[1,1,0,0,1,1,0,0]) → [0,1,1,0,0,1,1,0]`

**Tuple.** Un tuple (uplet) est une liste immuable ; on écrit un tuple avec des parenthèses, par exemple `a` = (1,2,3) est un tuple.

Quand il y a un seul élément, il est suivi d'une virgule : `a` = (1,) est un tuple, mais `a` = (1) est un entier.

Un tuple ne peut pas être modifié, mais on peut concaténer car cela revient à créer un nouveau tuple.

Par exemple :

```
>>>a=(1,2,3)
>>>b=(2,2,1)
>>>a+b
(1,2,3,2,2,1)
```

**Dictionnaires.** Un dictionnaire, au format `dict`, est un structure complexe de données, modifiable comme une liste, mais dans laquelle on accède à un objet donné, non par pas un index (qui est forcément un nombre entier), mais par une clé (qui peut être aussi une chaîne de caractères ou tout autre objet). Le dictionnaire est défini comme un ensemble non ordonné de couples (clé, valeur), sous la forme

`{clé_1 : valeur_1,...,clé_n : valeur_n}`

Voici, par exemple, un dictionnaire `ecoles` qui recense les écoles intégrées par les étudiants :

`ecoles={'Baptiste':'Centrale Nantes'}`

On peut ensuite ajouter des étudiants :

`ecoles['Chloé']='EIVP'`, `ecoles['Emilien']='ENTPE'`, `ecoles['Raphaël']='UTC'`

On accède ensuite aux données en se servant de la clé et non plus d'un index :

`ecoles['Chloé']` renvoie `'EIVP'`

Comme les listes et les chaînes, les dictionnaires sont des objets itérables ; l'itération se faisant sur les clés.

Ainsi l'appel
```
for a in ecoles :
    print(a)
```
renvoie la liste des clés.

L'année suivante, deux autres étudiants, Colyne et Alexandre ont intégré l'UTC.

**Exercice 3.17** *Ecrire une fonction python* `etudiants(ecole:str)->list` *qui prend une chaîne de caractères correspondant à une école en argument et qui renvoie la liste des étudiants qui ont intégré cette école.*

Par exemple, `etudiants('UTC')` ⟶ `['Raphaël','Colyne','Alexandre']`

*Ecrire ensuite une fonction* `max_ecole` *qui renvoie le nom de l'école ayant accueilli le plus d'étudiants de Chaptal.*

### 3.6.3. La bibliothèque Numpy.

Il est impossible de donner une liste exhaustive de toutes les bibliothèques relatives à Python. Toutefois, pour les applications scientifiques, la bibliothèque Numpy apparaît incontournable.
L'importation d'une bibliothèque se fait grâce à la commande `import`. Pour accéder à une fonction particulière de la bibliothèque, on peut taper
`nom_de_la_bibliothèque.nom_de_la_fonction()`.
Par exemple pour la fonction `sin()` :

```
import numpy
numpy.sin(3.141592654)
-4.1020685703470686e-10
```

Pour éviter de retaper le nom complet de la bibliothèque (qui dans certain peut être assez long), on peut modifier le nom de la bibliothèque en utilisant le terme `as`. Par exemple, ici on renomme `numpy` par `np` :

```
import numpy as np
np.sqrt(1024)
32.0
```

On peut se passer complètement du rappel du nom de la bibliothèque, en spécifiant directement les fonctions qu'on désire importer avec le terme `from` :

```
from numpy import log
log(1)
0.0
log(0)
-inf
```

On peut même importer directement toutes les fonctions d'une bibliothèque avec le symbole * :

```
from numpy import *
cos(pi)
-1.0
```

A priori, la dernière solution semble la plus simple car elle évite des écritures répétitives.

La listes des fonctions numériques définies dans `numpy` est extrêmement importante.
On peut retrouver les fonctions les plus classiques : `exp()`, `log()` (pour le logarithme népérien), `cos()`, `sin()`, `tan()`, `arccos()`, `arcsin()`, `arctan()`.

On peut également programmer une fonction non usuelle, mais dont connaît l'expression.
On considère la fonction $f$ définie par $f(x) = \dfrac{1}{1+x^2}$ ; on souhaite affecter cette fonction à la variable `f`.
Voici deux solutions :

```
f=lambda x:1/(1+x**2)
>>> f(1)
0.5
```

```
def f(x):
  return 1/(1+x**2)
>>> f(1)
0.5
```

## 3.7 Conception d'un algorithme.

Pour résoudre un problème non élémentaire, la démarche algorithmique consiste à décomposer le problème en sous-problèmes plus simples, jusqu'à l'obtention de sous-problèmes élémentaires.

***Exemple* 3.7** On dispose d'une planche, d'un marteau et d'un clou. Objectif : le clou est planté complètement dans la planche.

Ce problème peut se décomposer en deux sous-problèmes ; (**1**) : programmer la machine pour qu'elle se saisisse du marteau et du clou et (**2**) : programmer la machine pour qu'elle enfonce le clou. Là encore, le sous-problème (**2**) peut se décomposer en plusieurs sous-problèmes (**2.1**), (**2.2**)... : à chaque (**2.k**), taper un coup de marteau sur le clou, jusqu'à ce qu'il soit complètement enfoncé.

Pour s'assurer que l'algorithme créé fonctionne correctement, il est indispensable de respecter trois étapes de conception avec les éléments suivants :

- **Invariant de boucle** : Un invariant de boucle est une propriété qui :
  - est vérifiée avant d'entrer dans la boucle ;
  - si elle est vérifiée avant une itération, alors elle est aussi vérifiée après celle-ci ;
  - est vérifiée à la sortie de la boucle, ce qui garantit que l'algorithme résout bien le problème.

  Dans notre exemple l'invariant de boucle est : "le clou est planté dans la planche". En effet, on commence à utiliser les coups de marteau que si le clou est déjà positionné (légèrement planté) et, le clou doit rester planté après chaque coup de marteau, pour que le suivant fasse progresser la situation.
- **Terminaison** : Lorsque l'algorithme utilise une boucle, il est essentiel de définir une **condition d'arrêt** pour éviter que la boucle se répète infiniment. La terminaison de la boucle assure alors que le problème sera résolu au bout d'un nombre fini d'itérations.
  Dans notre exemple la condition d'arrêt est "la tête touche la planche". La terminaison est la distance

entre la tête du clou et la planche ; elle assure que le problème sera résolu en un nombre fini de coups car la longueur du clou, donc la distance entre la tête et la planche, est finie et celle-ci décroît à chaque coup de marteau, donc converge vers 0.

- ◇ **Initialisation.** Elle doit instaurer l'invariant. Dans notre exemple, l'initialisation est "planter légèrement le cou à la main".

***Exemple* 3.8** Revenons au problème du calcul de la somme des $n$ premiers entiers naturels. Voici un programme possible :

```
def somme_entiers(n):
    somme=0
    k=0
    while k<=n:
        somme+=k
        k+=1
    return somme
```

- ◇ L'invariant de boucle est la propriété $S_k = \sum_{i=0}^{k} i$ ; en effet, initialement : $S_0 = \sum_{i=0}^{0} i = 0$ ; pour $k$ donné si $S_k = \sum_{i=0}^{k} i$, alors, après la $k+1$ ième itération, on obtient $S_{k+1} = \sum_{i=0}^{k} i + k + 1 = \sum_{i=0}^{k+1} i$ ; et, à la sortie de la boucle, on obtient $S_n = \sum_{i=0}^{n} i$.
- ◇ La condition d'arrêt est $k \leqslant n$ et la terminaison est $n - k$ ce qui assure que la boucle s'arrête après $n$ itérations.
- ◇ L'initialisation est $S_0 = 0$.

**Exercice 3.18** *L'algorithme ci-dessous, dit "algorithme d'Euclide" permet d'effectuer la division entière de A par B. Prouver ce résultat.*

```
A=input('Entrez la valeur de A :')
B=input('Entrez la valeur de B :')
a=int(A)
b=int(B)
r=a
q=0
while r>=b:
    r=r-b
    q=q+1
print ('Le quotient est',q,'et le reste est ',r)
```

## 3.8 Complexité d'un algorithme.

Il existe souvent plusieurs algorithmes capables de traiter le même problème. On choisira donc celui qui est le plus **efficace**, c'est-à-dire celui qui a une exécution rapide et qui mobilise le moins de ressources mémoire. La rapidité d'exécution est évidemment liée au nombre de données traitées, mais aussi au nombre d'opérations effectuées avec ces données. La relation entre le nombre $n$ de données et le nombre d'opérations s'appelle la **complexité** de l'algorithme.

Voici un tableau qui récapitule les complexités les plus fréquentes :

| **Complexité** | **Nom courant** | **Description** |
|---|---|---|
| $\mathcal{O}(1)$ | temps constant | le temps d'exécution ne dépend pas du nombre de données à traiter |
| $\mathcal{O}(\ln n)$ | logarithmique | l'exécution est quasi instantanée |
| $\mathcal{O}(n)$ | linéaire | le nombre d'opérations est proportionnel au nombre de données; l'exécution est rapide jusqu'à des données de taille comparable à la mémoire vive |
| $\mathcal{O}(n \ln n)$ | semi-linéaire | complexité un peu moins bonne que la précédente mais qui reste très intéressante |
| $\mathcal{O}(n^2)$ | quadratique | complexité acceptable pour des données de taille raisonnable |
| $\mathcal{O}(n^k)$ | polynomiale | il n'est pas rare de rencontrer des complexité en $\mathcal{O}(n^3)$ ou $\mathcal{O}(n^4)$ |
| $\mathcal{O}(2^n)$ | exponentielle | un algorithme d'une telle complexité est impraticable sauf pour de très petites données |

***Exemple* 3.9** Soit $P$ un polynôme de degré $n$ et $a \in \mathbb{R}$ : $P = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n$

On souhaite calculer $P(a)$ ; pour cela on va utiliser deux algorithmes et évaluer leur complexité pour comparer leur efficacité.

On suppose que les cœfficients du polynôme sont enregistrés dans une liste $\texttt{P} = [\texttt{a}_0, \texttt{a}_1, ..., \texttt{a}_n]$

```
def Eval1(P,a):
    somme=P[0]
    for k in range(1,len(P)):
        somme=somme+P[k]*a**k
    return somme
```

```
def Eval2(P,a):
    somme=P[0]
    b=1
    for k in range(1,len(P)):
        b=b*a
        somme+=b*P[k]
    return somme
```

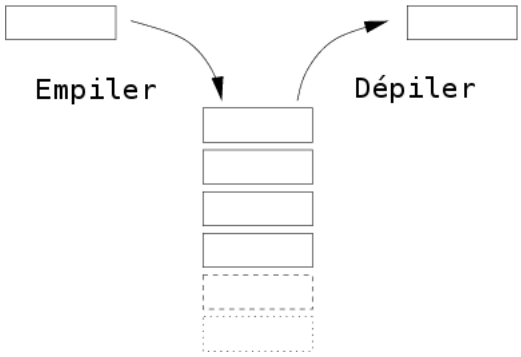
## 3.9 Les piles

Nous avons déjà rencontré plusieurs types de données structurées : les chaînes de caractères (`string`), les listes (`list`), les dictionnaires (`dict`) et les matrices (`array`).

Nous allons ici étudier un nouveau type de données structurées, les **piles**, que nous implémenterons en Python à l'aide de listes.

Par définition, une pile peut être considérée comme une liste dont a limité les accès : on ne peut insérer ("empliler", fonction `push`) un élément qu'à une seule extrémité, appelée le **sommet** de la pile ; on ne peut retirer ("dépiler", fonction `pop`) un élément que s'il est au sommet et on ne peut voir un élément que s'il est au sommet.

Si on a besoin d'accéder à un élément qui n'est pas au sommet, on doit retirer un par un les éléments qui au dessus, en partant du sommet.




*Exemple 1.* Dans la pile suivante

| | |
|---|---|
| 3 | 4 |
| 2 | 3 |
| 1 | 2 |
| 0 | 1 |

seul l'élément 4 (d'indice 3) peut être vu (fonction `top(p)`) ; seul l'élément 4 peut être retiré (fonction `pop(p)`), ce qui donnerait :

| | |
|---|---|
| 2 | 3 |
| 1 | 2 |
| 0 | 1 |

On peut ajouter l'élément 5 au sommet de la pile par une fonction `push(5,p)` pour obtenir :

| | |
|---|---|
| 3 | 5 |
| 2 | 3 |
| 1 | 2 |
| 0 | 1 |

Pour manipuler les piles, nous allons introduire la **programmation modulaire**. Ainsi toutes les fonctions utiles à la manipulation des piles seront enregistrées dans un fichier (module), que nous nommerons `pilepy`. Pour pouvoir les utiliser, il suffira d'importer ce module par l'instruction : `import pilepy as pp`. Chaque fonction pourra alors être appelée sous le nom `pp.nomDeLaFonction`.

**Exercice 3.19** *Construire les fonctions suivantes, permettant les opérations élémentaires sur les piles. Ces fonctions seront enregistrées dans un fichier nommé* `pilepy`

| | |
|---|---|
| `newStack()` | sans argument, crée une pile sous la forme d'une liste vide |
| `isEmpty(p)` | prend en argument une pile et renvoie `True` ou `False` suivant que la pile est vide ou non |
| `top(p)` | prend en argument une pile non vide et renvoie l'élément au sommet de la pile |
| `pop(p)` | prend en argument une pile non vide ; renvoie et supprime l'élément au sommet |
| `push(x,p)` | prend en arguments un élément et une pile, et insère l'élément au sommet de la pile |

**Exercice 3.20** ***File d'attente***

Il existe une autre structure de données, qui peut également considérée comme une liste dont on a limité l'accès : la **file**.

Dans une file, on ne peut insérer ("enfiler") un élément qu'à une extrémité, appelée **queue** ; on ne peut voir et retirer ("défiler") un élément que s'il est situé à l'autre extrémité appelé **tête**.

Ce type de structure correspond par exemple à des situations où on a besoin de mémoriser temporairement des actions en attente d'être traitées dans l'ordre.

*Construire les fonctions suivantes, permettant les opérations élémentaires sur les files :*

| | |
|---|---|
| `creer_file()` | sans argument, crée une file sous la forme d'une liste vide |
| `voir(f)` | prend en argument une file non vide et renvoie l'élément en tête de file |
| `defiler(f)` | prend en argument une file non vide ; renvoie et supprime l'élément en tête de file |
| `enfiler(x,f)` | prend en arguments un élément et une file, et insère l'élément en queue file |

**Exercice 3.21** ***Expressions bien parenthésées***

*On veut construire à l'aide d'une pile un vérificateur de parenthésage.*

*Ecrire une fonction qui prend en argument une chaîne de caractères contenant une expression parenthésée, la parcourt de gauche à droite de la façon suivante : lorsqu'elle rencontre une parenthèse ouvrante, elle empile la parenthèse fermante correspondante ; lorsqu'elle arrive à une fermeture, lorsque c'est possible, elle dépile.*

*Cette fonction doit renvoyer* `True` *si l'expression est bien parenthésée et* `False` *dans le cas contraire.*

## 3.10 La récursivité

La **récursivité** est un moyen de répéter un bloc d'instructions sans utiliser les instructions `while` et `for`. Pour cela, on programme une fonction qui va s'appeler elle-même.

### 3.10.1. Un exemple de fonction récursive

***Exemple* 3.10** La fonction récursive suivante crée une suite de nombres. Reconnaître cette suite :

```
def suite(n): #n est un entier naturel
    if n==0:
        return 1
    elif n==1:
        return 1
    return suite(n-1)+suite(n-2)
```

*Remarque.* ⚠ comme pour la boucle `while`, il ne faut pas oublier de prévoir une condition d'arrêt. Cependant, le nombre maximal d'appels récursifs est de l'ordre de 1000 par défaut, donc contrairement à la boucle `while`, même sans condition d'arrêt, un programme récursif s'arrête avec comme message d'erreur :

`Runtime Error : maximum recursion depth exeeded in comparison`

### 3.10.2. Pile d'exécution et récursivité terminale

Lors de l'exécution d'un algorithme récursif, les appels récursifs successifs sont stockés dans une pile, c'est la **pile d'exécution**.

La pile d'exécution est un emplacement mémoire destiné à stocker les paramètres, les variables locales ainsi que les adresses mémoires de retour des fonctions en cours d'exécution. On peut différencier deux types de fonctions récursives : celles pour lesquelles il n'y a pas de traitement entre l'appel récursif et le retour de la fonction, et celles pour lesquelles il y a des opérations entre l'appel et le retour.

Dans le deuxième cas, l'ordinateur empile dans la pile d'exécution les appels récursifs sans traiter les opérations, puis, lorsque la condition d'arrêt est vérifiée, la pile est dépilée, les opérations étant exécutées successivement.

Pour illustrer cette différence, utilisons l'exemple du calcul de la factorielle de 4 :

Voici deux fonctions, une de chaque type ; et les schémas d'exécutions associés lors des appels `factorielle1(4)` et `factorielle2(4)` :

```
def factorielle1(n):
    if n==0:
      return 1
    return n*factorielle1(n-1)
 #il y a 1 opération entre l'appel et
                      le retour
```

```
def factorielle2(n,f=1):
    if n==0:
      return f
    return factorielle2(n-1,f*n)
#il n'y a pas d'opération entre l'appel
                      et le retour
```

$F(4) = 4 * F(3)$
$\quad F(3) = 3 * F(2)$
$\quad\quad F(2) = 2 * F(1)$
$\quad\quad\quad F(1) = 1 * F(0)$
$\quad\quad\quad\quad F(0) = 1$
$\quad\quad\quad F(1) = 1$
$\quad\quad F(2) = 2$
$\quad F(3) = 6$
$F(4) = 24$

$F(4,1) = F(3,4)$
$\quad F(3,4) = F(2,12)$
$\quad\quad F(2,12) = F(1,24)$
$\quad\quad\quad F(1,24) = F(0,24)$
$\quad\quad\quad\quad 24$

Dans le deuxième cas l'opération emplilage/dépilage est plus performant. Ce type de fonction récursive avec retour direct de l'appel récursif s'appelle **récursivité terminale**.

**Exercice 3.22** *Ecrire une fonction récursive non terminale* `puissance(a,n)`*, prenant en argument un nombre* `a` *et un entier naturel* `n`*, et qui calcule* $a^n$.

*Transformer la fonction précédente en une fonction récursive terminale.*

**Exercice 3.23** *Calculer le* $n^{ième}$ *terme des suites* $(u_n)$ *et* $(v_n)$ *définie par* $u_0 = 1$, $v_0 = -1$ *et* $\forall n \in \mathbb{N}$ *:*

$$\begin{cases} u_{n+1} = 2u_n + v_n \\ v_{n+1} = u_n - 2v_n \end{cases}$$

**Exercice 3.24** *Programmer une fonction récursive terminale donnant les termes de la suite de Fibonacci.*

### 3.10.3. Preuve de terminaison et complexité

Comme pour les instructions itératives, la **terminaison** est assurée par une **condition d'arrêt**. Ainsi on prouvera la terminaison en exhibant une suite d'entiers naturels strictement décroissante.

Par exemple, pour la fonction `factorielle(n)`, si on note $(u_p)$ la suite des arguments la fonction, on a $u_0 = n$, $u_1 = n-1$, et de manière générale, $u_{p+1} = u_p - 1$ ; donc la suite $(u_p)$ est bien une suite d'entiers naturels strictement décroissante. Elle prend donc un nombre fini de valeurs ce qui prouve que la fonction `factorielle(n)` se termine.

Pour simplifier, on considère le nombre d'appels à la fonction récursive pour estimer la **compléxité** en temps. Lorsqu'il s'agit d'une récursivité simple et que la complexité d'une étape est constante, la complexité est estimée à $\mathcal{O}(n)$.

**Exercice 3.25** *Dans sa version terminale, la fonction récursive donnant les termes de la suite de Fibonacci appelle n fois la fonction ; sa complexité est donc en* $\mathcal{O}(n)$.

*Estimer la complexité de la fonction présentée dans l'exemple* **3.13**. *Comparer les résultats.*

## 3.11 Les algorithmes de tri

Dans ce paragraphe nous nous intéressons à des algorithmes capables de trier une liste, ou un tableau à une dimension, contenant des nombres, ou tout type d'objets dont l'ensemble est muni d'une relation d'ordre. Par exemple, on peut trier une liste contenant des chaînes de caractères à l'aide de l'ordre lexicographique.

Il existe des méthodes de tri déjà implémentées en langage Python, comme la méthode `sort` :

```
In [1]: L=[35,22,56,29]
In [2]: L.sort()

In[3]: L
Out[3]: [22, 29, 35, 56]
```

```
In[1]: T=['Ile et Vilaine','Côtes d Armor',
          'Morbihan','Finistère']
In[2]: T.sort()
In[3]: T
Out[6]: ['Côtes d Armor', 'Finistère',
          'Ile et Vilaine', 'Morbihan']
```

Nous pouvons remarquer que cette méthode est une fonction qui modifie la liste prise en argument et qui ne renvoie rien.

Notre objectif étant de comprendre et d'implémenter quelques algorithmes de tri, nous nous interdisons donc d'utiliser les méthodes de tri Python.

Il est également possible d'écrire une fonction qui ne modifie pas la liste de départ et qui retourne la liste triée. Par exemple

$$\texttt{L} = [35, 22, 56, 29]$$

$$\texttt{tri(L)} \quad \text{retourne} \quad [22, 29, 35, 56]$$

$$\texttt{L} = [35, 22, 56, 29]$$

### 3.11.1. Tri par sélection

Pour trier une liste L comportant $n$ éléments par **sélection**, voici la méthode :
En partant de la position `i = 0`, on recherche le plus petit élément parmi les éléments d'indice `i+1` à `n-1` et on l'échange avec `L[i]`.
Par exemple, pour trier [12,3,17,9,4,16], on obtient successivement :

$$[12,3,17,9,4,16] \quad [12,|\boxed{3},17,9,4,16] \quad [3,12,|17,9,\boxed{4},16] \quad [3,4,17,|\boxed{9},12,16]$$
$$[3,4,9,17,|\boxed{12},16] \quad [3,4,9,12,|17,\boxed{16}] \quad [[3,4,9,12,17,|\boxed{16}] \quad [3,4,9,12,16,17]$$

**Exercice 3.26** *Compléter la fonction suivante pour qu'elle retourne la liste triée par sélection :*

```
def tri_selection(L):
    for i in range(len(L)-1):
    ...
        for j in range (i+1,len(L)):
             # recherche du minimum dans le tableau restant
             # comparer ce minimum à L[i]
    return L
```

*Evaluer la complexité temporelle de cet algorithme.*

### 3.11.2. Tris par insertion

Pour trier une liste L par **insertion**, voici la méthode :
On prend le premier élément et on le met à l'indice `i = 0` ; puis on insère les autres éléments dans la partie déjà triée en plaçant chaque nouvel élément à la bonne place.
Cela donne :

$$[12,|3,17,9,4,16] \quad [3,12,|17,9,4,16] \quad [3,12,17,|9,4,16]$$
$$[3,9,12,17,|4,16] \quad [3,4,9,12,17,|16] \quad [3,4,9,12,16,17]$$

**Exercice 3.27** *Ecrire une fonction python prenant en argument une liste et qui retourne cette liste triée par insertion en complétant le script suivant.*
*Evaluer la complexité temporelle de cet algorithme. Comparer à la méthode par sélection.*

```
def tri_insertion(L):
    for i in range(1,len(L)):#on traite les éléments restants
        ...                  #on mémorise l'élément à traiter
        ...                  #variable créée pour trouver la bonne place
        while ...            #tant que la bonne place n'est pas trouvée
            ...              #on cherche la bonne place
        ...                  #on insère l'élément à sa place
        ...                  #on supprime le doublons
    return L
```

### 3.11.3. Tris à bulles

On donne l'algorithme suivant :

```
def tri_a_bulles(L):
      for i in range(len(L)-1):
           for j in range(len(L)-1,i,-1):
                if L[j]<L[j-1]:
                    L[j],L[j-1]=L[j-1],L[j]
     return L
```

**Exercice 3.28** *Prouver que cet algorithme retourne la liste triée.*
*Evaluer sa complexité temporelle.*

### 3.11.4. **Le tri rapide**

Le tri **rapide** (ou "quicksort") est un tri récursif dans lequel on divise le problème initial en deux sous-problèmes suivant le principe de « diviser pour mieux régner ». Concrètement, il s'agit de passer du tri d'une liste comportant $n$ éléments aux tris de deux sous-listes de tailles strictement inférieures à $n$.

Pour cela on choisit un élément `e` de la liste qu'on appelle **pivot**, on le retire de la liste et on crée deux sous-listes, l'une contenant les éléments strictement inférieurs à `e`, et l'autre les éléments supérieurs à `e`.

On trie récursivement les deux sous-listes et on regroupe le tout.

Par exemple, pour trier `[12,3,17,9,4,16]`, on obtient successivement :

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | [3,9,4] | | [12] | | [17,16] | | | choix de 12 comme pivot |
| | | [] | [3] | [9,4] | [12] | [16] | [17] | [] | | choix de 3 comme pivot à gauche et de 17 à droite |
| [] | [3] | [] | [4] | [9] | [] | [12] | [16] | [17] | [] | choix de 9 comme pivot |

**Exercice 3.29** *Ecrire une fonction python prenant en argument une liste et qui retourne une nouvelle liste triée par la méthode du tri rapide.*

Evaluons la complexité de cette méthode de tri. Soit $n \in \mathbb{N}$ le nombre d'éléments de la liste à triée, notons $c_n$ la complexité temporelle. Par construction, nous avons $c_0 = 0$ et $c_1 = 0$.

- ◇ *Dans le pire des cas :*
  On suppose ici qu'à chaque exécution de la fonction, tous les éléments se trouvent du même côté du pivot. (Ce qui est le cas si la liste est déjà triée par exemple !)
  Il y $n-1$ comparaisons et le tri de deux sous-listes, une de longueur 0 et l'autre de longueur $n-1$. On a donc $c_n = c_{n-1} + n - 1$ ; ce qui donne
  $$c_n = \frac{n(n-1)}{2} = \mathcal{O}\big(n^2\big)$$
- ◇ *Dans le meilleur des cas :*
  On suppose maintenant qu'à chaque exécution de la fonction, les éléments se répartissent équitablement de part et d'autre du pivot.
  Il y a toujours $n-1$ comparaisons et le tri de deux sous-listes, une de longueur $\lfloor \frac{n}{2} \rfloor$ et l'autre de longueur $\lfloor \frac{n-1}{2} \rfloor$. On a donc :
  $$c_n = c(n) = c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + n - 1$$
  - ⋆ Calculons $c(n)$ pour $n = 2^p - 1$ : en posant $u_p = c\big(2^p - 1\big)$, il vient : $u_p = 2u_{p-1} + 2^p - 2$.
    On démontre alors, par récurrence, que $u_p = (p-2)2^p + 2$.
  - ⋆ Désormais, soit $n \in \mathbb{N}$ ; il existe un unique $p \in \mathbb{N}$ tel que $2^p - 1 \leqslant n \leqslant 2^{p+1} - 1$ ; on a donc :
    $$u_p \leqslant c_n \leqslant u_{p+1} \;\Leftrightarrow\; (p-2)2^p + 2 \leqslant c_n \leqslant (p-1)2^{p+1} + 2$$
    pour $n$ au voisinage de $+\infty$, on a $(p-2)2^p + 2 \sim p2^p \sim n\log_2(n)$ ; avec $\log_2(n) = \frac{\ln(n)}{\ln(2)}$ ;
    ainsi $(p-2)2^p + 2 = \mathcal{O}\big(n\ln(n)\big)$ ; de même $(p-1)2^{p+1} + 2 = \mathcal{O}\big(n\ln(n)\big)$

  Finalement, on obtient :
  $$c_n = \mathcal{O}\big(n\ln(n)\big)$$

### 3.11.5 Le tri fusion

Le tri **fusion** est également un tri récursif dans lequel on divise le problème initial en deux sous-problèmes. Partant d'une liste de $n$ éléments, on la divise en deux listes contenant environ $\frac{n}{2}$ données. La méthode consiste à trier la première moitié de la liste, puis la deuxième et de fusionner les deux listes triées en une seule liste triée.

Pour programmer cette méthode nous utiliserons une fonction auxiliaire `fusion(L1,L2)` qui prend en argument deux listes `L1` et `L2` déjà triées et qui retourne la liste fusionnée.

Pour comprendre cet algorithme récursif, observons la pile d'exécution :

- ◇ *Empilage :*

`tri_fusion([12,3,17,9,4,16])`

`tri_fusion([12,3,17])` `tri_fusion([9,4,16])`

`tri_fusion([12])` `tri_fusion([3,17])` `tri_fusion([9])` `tri_fusion([4,16])`

`tri_fusion([12])` `tri_fusion([3])` `tri_fusion([17])` `tri_fusion([9])` `tri_fusion([4])` `tri_fusion([16])`

- ◇ *Dépilage :*

[12] [3] [17] [9] [4] [16]

[3,12] [9,17] [4,16]

[3,9,12,17] [4,16]

[3,4,9,12,16,17]

La fonction `fusion(L1,L2)` utilise une variable `L` de type liste qui doit contenir la fusion des deux listes `L1` et `L2`. Pour comprendre le fonctionnement de cette fonction, observons l'évolution de la variable `L` sur un exemple : prenons `L1` = [3,9,12,17] et `L2` = [4,16].

`L` = [] `L` = [3] `L` = [3,4] `L` = [3,4,9] `L` = [3,4,9,12] `L` = [3,4,9,12,16] `L` = [3,4,9,12,16,17]

La fonction compare `L1[0]` avec `L2[0]` et insère `L1[0]` dans `L`; ensuite elle compare `L1[1]` avec `L2[0]` et insère `L2[0]` dans `L` ... jusqu'à avoir parcouru les deux listes. Lorsque tous les éléments d'une listes ont été choisis, il suffit de compléter `L` avec les éléments restants de l'autre liste.

**Exercice 3.30**

1. *Proposer une fonction* `fusion(L1,L2)` *dont le fonctionnement est décrit ci-dessus.*
2. *Ecrire alors une fonction* `tri_fusion(T)` *qui prend en argument une liste* `T` *et qui retourne une liste contenant les éléments de* `T` *triés par la méthode de fusion.*

### 3.11.6. Comparaison des différents tris

Nous avons étudié cinq tris différents, trois tris quadratiques (par sélection, par insertion et à bulles) et deux tris récursifs (rapide et fusion).

Les trois tris itératifs modifient la liste prise en argument pour la retourner triée, alors que les deux tris récursifs créent une nouvelle liste sans modifier la liste de départ. Une autre différence entre ces tris est la complexité temporelle ; voici un tableau résumant la situation :

| | meilleur des cas | pire des cas |
|---|---|---|
| tri par sélection | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ |
| tri par insertion | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ |
| tri à bulles | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ |
| tri rapide | $\mathcal{O}(n\ln(n))$ | $\mathcal{O}(n^2)$ |
| tri fusion | $\mathcal{O}(n\ln(n))$ | $\mathcal{O}(n\ln(n))$ |

La complexité du tri fusion est est en $\mathcal{O}\big(n\ln(n)\big)$ dans tous les cas, donc semble moins risqué que le tri rapide. En pratique, c'est cependant le tri rapide qui est le plus utilisé, car son pire des cas est usuellement rare et on constate expérimentalement qu'il est meilleur que le tri fusion en moyenne. Pour cela il suffit de calculer les temps moyens d'exécution des algorithmes sur les listes construites aléatoirement et de les comparer.

## 3.12 La méthode d'Euler

Pour certaines équations différentielles, nous savons déterminer l'expression de la fonction solution, c'est-à-dire que nous savons résoudre analytiquement ces équations.

Malheureusement, il existe des équations différentielles pour lesquelles il n'est pas possible de déterminer l'expression de la solution. On a alors recours à des programmes qui recherchent une solution numérique approchée.

L'objectif de ce paragraphe est d'étudier une méthode numérique : la méthode d'EULER.

Pour illustrer la méthode nous utiliserons l'équation différentielle $\mathscr{E}$ :

$$\begin{cases} y' - 2ty = 1 \\ y(0) = 0 \end{cases}$$

### 3.12.1. Les limites de la méthode analytique.

**Exercice 3.31** *Déterminer l'expression $y_h(t)$ de la solution de l'équation homogène : $y' - 2ty = 0$.*

Désormais on recherche une solution particulière par la méthode de variation de la constante, c'est-à-dire sous la forme $y_p(t) = k(t)\mathrm{e}^{t^2}$, où $k$ est une fonction à déterminer.

*En utilisant l'équation $\mathscr{E}$, donner une expression de la fonction $k$. En déduire l'unique solution du problème de Cauchy. Quel commentaire peut-on faire ?*

### 3.12.2. La méthode d'Euler explicite.

D'une manière générale, une équation différentielle d'ordre 1 s'écrit :

$$y'(t) + a(t)y(t) = b(t) \text{ ou encore } \frac{\mathrm{d}y}{\mathrm{d}t}(t) = -a(t)y(t) + b(t) = f\big(t, y(t)\big)$$

Dans notre exemple, $f\big(t, y(t)\big) = 2ty(t) + 1$.

On souhaite résoudre numériquement cette équation sur un intervalle de temps $[0, T_{max}]$. Le temps sera représenté numériquement par une liste de $N$ instants régulièrement espacés que l'on écrira

$$\mathtt{T} = [\mathtt{t_0} = 0, \mathtt{t_1}, ..., \mathtt{t_k}, \mathtt{t_{k+1}}, ..., \mathtt{t_{N-1}} = \mathtt{Tmax}]$$

On note $h = t_{k+1} - t_k$ le pas de temps, ainsi $t_k = t_0 + kh = kh$.

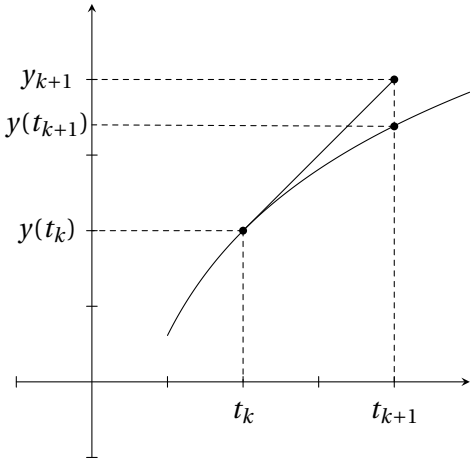
On note $y(t_k)$ les valeurs de la fonction exactes pour les instants choisis et $y_k$ les valeurs approchées construites par itération.

On obtient ces valeurs approchées par un développement limité à l'ordre 1 :

$$y(t_{k+1}) = y(t_k + h) = y(t_k) + hy'(t_k) + o(h)$$
$$\simeq y(t_k) + hy'(t_k)$$

La stratégie itérative mise en œuvre dans la méthode d'Euler explicite consiste, à partir d'une condition initiale, à rechercher une valeur approchée de la valeur $y(t_{k+1})$ avec la formule :

$$y_{k+1} = y_k + hy'(t_k) = y_k + hf(t_k, y_k)$$




La condition initiale est $y(t_0) = y_0$ ; ensuite, $y_1 = y_0 + hf(t_0, y_0)$ ; puis, $y_2 = y_1 + hf(t_1, y_1)$ et ainsi de suite...

**Exercice 3.32** *Ecrire une fonction* `init_T(Tmax,N)` *prenant pour arguments la durée* `Tmax` *de l'étude et le nombre* `N` *d'instants et retournant la liste* `T`.

Pour la suite, dans notre exemple, $f(t, y) = 2ty + 1$ et $y(0) = y_0 = 0$.
*Ecrire une fonction* `f(t,y)` *prenant en arguments les valeurs de* `t` *et de* `y` *et retournant l'expression de la fonction* $f(t, y)$.
Cette fonction sera modifiée à chaque résolution d'une autre équation différentielle.

*Ecrire enfin une fonction* `solve_euler(T)` *prenant en argument la liste* `T` *des instants* $t_k$ *de la résolution numérique et retournant la liste* `S` *des valeurs* $y_k$*, valeurs approchées de la solution de l'équation différentielle calculées aux instant,* $t_k$ *par la méthode d'Euler.*

D'après les calculs précédents, $y(1) = \mathrm{e}\displaystyle\int_0^1 \mathrm{e}^{-x^2}\,\mathrm{d}x$. Le calcul de l'intégrale peut également être effectué à l'aide des sommes de Riemann, par la méthode des trapèzes par exemple.
Comparons les résultats :

```
In[1]: solve_euler(init_T(1,1000))[-1]
Out[1]: 2.026937538968067

In[2]: exp(1)*somme_trapezes(lambda x:exp(-x**2),0,1,1000)
Out[2]: 2.0300783026120328
```

### 3.12.3. La méthode d'Euler implicite.

Le principe de la méthode d'Euler implicite est similaire au précédent, sauf que l'on utilise cette fois le taux d'accroissement pour approcher le nombre dérivé au point $t_{k+1}$, par la formule :

$$y'(t_{k+1}) = \frac{\mathrm{d}y}{\mathrm{d}t}(t_{k+1}) \simeq \frac{y(t_{k+1}) - y(t_k)}{t_{k+1} - t_k} \simeq \frac{y_{k+1} - y_k}{h}$$

On obtient ainsi, de manière implicite, la valeur de $y_{k+1}$ par la formule

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1})$$

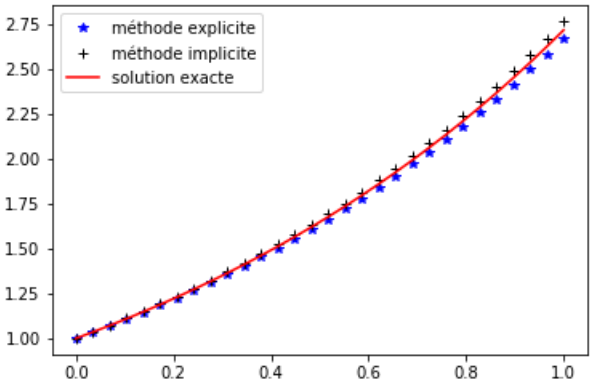
La condition initiale est $y(t_0) = y_0$ ; ensuite, $y_1 = y_0 + hf(t_1, y_1)$ ; puis, $y_2 = y_1 + hf(t_2, y_2)$ et ainsi de suite...
On constate que $y_{k+1}$ est solution d'une équation et que sa valeur n'est donc pas obtenue explicitement. Un calcul supplémentaire est donc souvent nécessaire.

Pour comparer ces deux méthodes, utilisons le problème de Cauchy suivant :

$$\begin{cases} y' = y \\ y(0) = 1 \end{cases}$$

Par la méthode explicite nous obtenons une suite $(y_k)$ telle que $y_{k+1} = y_k + hy_k = (1+h)y_k$.
Par la méthode implicite nous obtenons une suite $(z_k)$ telle que $z_{k+1} = z_k + hz_{k+1}$ ; dans cette situation, nous pouvons en déduire l'expression de $z_{k+1}$ par la formule $z_{k+1} = \dfrac{1}{1-h} z_k$.
Nous pouvons ensuite obtenir le graphique suivant :

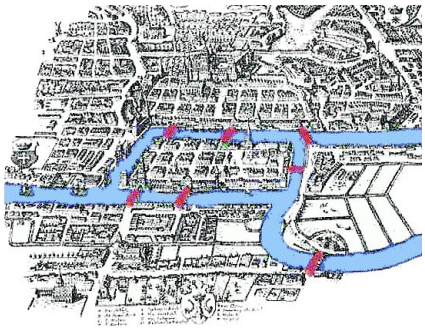
```
T=linspace(0,1,30)

F=solve_euler_exp(init_T(1,30))
G=solve_euler_imp(init_T(1,30))

plot(T,F,'b*',label='méthode explicite')
plot(T,G,'k+',label='méthode implicite')
plot(T,exp(T),'r',label='solution exacte
                        ')

legend()
```

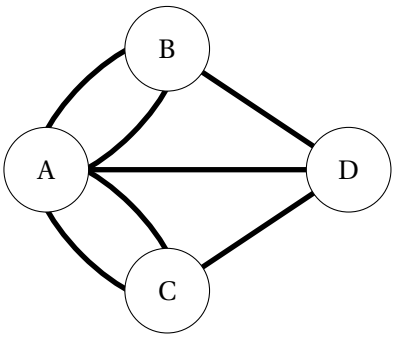



## 3.13 L'algorithme de Dijkstra

La théorie des graphes débute avec les travaux d'EULER au XVIII$^e$ siècle et trouve son origine dans l'étude de certains problèmes, tels que celui des ponts de Königsberg (actuellement Kaliningrad) : les habitants de Königsberg se demandaient s'il était possible, en partant d'un quartier quelconque de la ville, de traverser tous les ponts sans passer deux fois par le même et de revenir à leur point de départ.

Voici une représentation de Königsberg avec ses quatre quartiers et ses sept ponts ; ainsi que sa modélisation sous la forme d'un graphe :






En 1736, Euler démontre qu'une telle promenade n'existe pas en caractérisant les graphes que l'on appelle aujourd'hui **eulériens**.

Par théorème, un graphe simple connexe est eulérien si et seulement si pour tout sommet du graphe, son degré est pair.

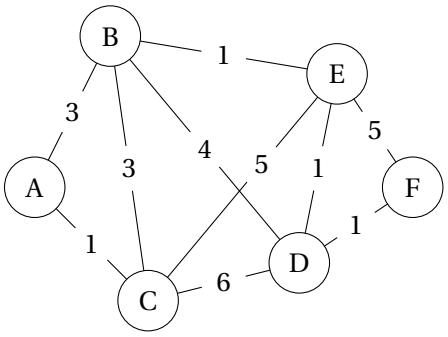
Dans le cas des ponts de Königsberg, le sommet A est de degré 5 et les sommets B, C et D sont de degré 3.

La théorie des graphes s'est alors développée dans diverses disciplines telles que la chimie, la biologie, les sciences sociales. Depuis le début du XX$^e$ siècle, elle constitue une branche à part entière des mathématiques, grâce aux travaux de König, Menger, Cayley puis de Berge et d'Erdös.

De manière générale, un graphe permet de représenter la structure, les connexions d'un ensemble complexe en exprimant les relations entre ses éléments : réseau de communication, réseaux routiers (recherche du plus court chemin), interaction de diverses espèces animales, circuits électriques,...

Les graphes constituent donc une méthode de pensée qui permet de modéliser une grande variété de problèmes en se ramenant à l'étude de sommets et d'arcs. Les derniers travaux en théorie des graphes sont souvent effectués par des informaticiens, du fait de l'importance qu'y revêt l'aspect algorithmique.

Il existe des graphes **pondérés**, c'est-à-dire dont les arêtes sont associées à des valeurs numériques. On définit alors la **matrice d'adjacence** $M = (m_{i,j})$ où $m_{i,j}$ est égal à la valeur portée par l'arête reliant les sommets $i$ et $j$ si ces sommets sont adjacents, ou $\infty$ sinon. Voici un exemple de graphe pondéré et de sa matrice d'adjacence :




$$M = \begin{pmatrix} 0 & 3 & 1 & \infty & \infty & \infty \\ 3 & 0 & 3 & 4 & 1 & \infty \\ 1 & 3 & 0 & 6 & 5 & \infty \\ \infty & 4 & 6 & 0 & 1 & 1 \\ \infty & 1 & 5 & 1 & 0 & 5 \\ \infty & \infty & \infty & 1 & 5 & 0 \end{pmatrix}$$

On peut implémenter cette matrice en Python sous la forme d'une variable de type `array` :

```
from numpy import*

M=array([[0,3,1,inf,inf,inf],[3,0,3,4,1,inf],[1,3,0,6,5,inf],[inf,4,6,0,1,1],
         [inf,1,5,1,0,5],[inf,inf,inf,1,5,0]])
```

Le graphe précédent n'est pas **orienté**, c'est-à-dire que lorsque deux sommets sont voisins, comme A et B par exemple, on peut indifféremment aller de A vers B ou de B vers A. La matrice d'adjacence est donc symétrique.

Le graphe est **simple**, car il y a au plus une arête entre deux sommets ; il est **connexe** car de chaque sommet il part au moins une arête, autrement dit, aucun sommet n'est isolé.

L'algorithme de **Dijkstra** sert à résoudre le problème du plus court chemin. Il permet, par exemple, de déterminer le plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. Il s'applique à un graphe connexe, simple et non orienté dont le poids lié aux arêtes est positif ou nul. L'algorithme porte le nom de son inventeur, l'informaticien néerlandais Edsger DIJKSTRA et a été publié en 1959.

Par exemple, on recherche dans le graphe précédent, le plus court chemin permettant de joindre les sommets A et F.

On construit un tableau, dont la première ligne contient les sommets du graphe : on initialise en affectant la valeur 0 à A et $\infty$ aux autres sommets :

La ligne suivante donne les voisins de A en précisant les poids associés à la ville d'origine, par exemple 3(A) dans colonne du sommet B signifie qu'on atteint B avec un chemin de longueur 3 en venant de A ; la ville choisie sera celle de poids minimum, donc C.

Une fois un sommet choisi son poids devient $\infty$.

Et on continue ainsi, en choisissant toujours la ville de poids minimum. Voici le résultat :

| distance totale | A | B | C | D | E | F | villes visitées |
|---|---|---|---|---|---|---|---|
| | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | A |
| 0 | $\infty$ | 3(A) | 1(A) | $\infty$ | $\infty$ | $\infty$ | C |
| 1 | $\infty$ | 3(A) | $\infty$ | 7(C) | 6(C) | $\infty$ | B |
| 3 | $\infty$ | $\infty$) | $\infty$ | 7(C) | 4(B) | $\infty$ | E |
| 4 | $\infty$ | $\infty$ | $\infty$ | 5(E) | $\infty$ | 9(E) | D |
| 5 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 6(D) | F |

Le plus court chemin mesure donc $d$(A,F) = 6 avec le parcours suivant : A - B - E - D - F
Ce tableau nous donne même tous les chemins les plus courts en partant de A, à savoir :
$d$(A,B) = 3 avec A - B ; $d$(A,C) = 1 avec A - C ; $d$(A,D) = 5 avec A - B - E - D et $d$(A,E) = 4 avec A - B - E.

**Exercice 3.33** *Appliquer l'algorithme de Dijsktra pour déterminer le plus court chemin de F à B.*

Implémentons l'algorithme de Dijskstra.

Nous allons écrire une fonction `dijkstra(villes,depart,arrivee,M)` qui prend en arguments une liste contenant toutes les villes du graphe, la ville de départ et la ville d'arrivée, ainsi que la matrice d'adjacence du graphe.

- *Choix des variables*
  La fonction retourne deux variables : une variable numérique `dist_choix` contenant la valeur du chemin le plus court et une variable `chemin_choix` contenant la liste des villes correspondant aux parcours le plus court.
  En variables auxiliaires, nous utiliserons la variable `visit` pour stocker la liste des villes visitées au cours de l'algorithme, la variable `dist` pour stocker les distances de chaque parcourt (l'évolution de cette variable correspond à l'évolution des lignes du tableau), la variable `choix` contenant l'indice de la ville choisie à chaque étape, et la variable `chemins` permettant de mémoriser, à chaque étape, la ville visitée, la ville précédente et la longueur du parcours, ce qui permettra de reconstituer tous les parcours et en particulier celui qui nous intéresse.
  L'algorithme fonctionne suivant deux principes très classiques :

- *Principe de relâchement :*
  On note $s$ le sommet de départ. Soit $u$ un sommet quelconque du graphe, on suppose à ce stade que la distance minimum obtenue pour se rendre de $s$ à $u$ est $d(u)$.
  Le principe de **relâchement** (ou **relaxation**) consiste à savoir s'il est possible d'améliorer $d(u)$ en passant par un autre sommet $v$ du graphe. En pseudo-code, celà s'écrit :

  **Variables :** $G$ un graphe, $u$ et $v$ deux sommets de $G$, $d(u)$ et $dist(u,v)$ deux nombres
  **Début**
  **Pour** $v$ dans $G$
  **si** $d(u) > d(v) + dist(u,v)$
  $d(u) \leftarrow d(v) + dist(u,v)$
  **Fin**

  Il faudra alors garder en mémoire que le sommet précédent $u$ devient $v$.

- *Principe de sélection :*
  A chaque étape de l'algorithme, on choisit le sommet $u$ si : $u$ n'a pas encore été sélectionné et si $d(u)$ est minimum parmi tous les sommets non encore sélectionnés. Le principe de sélection de la solution optimale à chaque étape s'appelle l'algorithme **glouton**.

**Exercice 3.34** *Compléter le schéma du programme suivant :*

```
def dijkstra(villes,depart,arrivee,M):
    '''
    * villes est la liste contenant les villes,
    * depart est la variable contenant le nom de la ville de départ,
    * arrivee est la variable contenant la ville d'arrivée,
    * M est la matrice d'adjacence du graphe'''

    '''tableau des distances initialisé en attribuant un poids infini aux
    villes autres que depart, qui reçoit le poids nul'''
    dist=[inf]*len(villes)
    dist[villes.index(depart)]=0

    '''liste des villes visitées'''
    visit=[]
    '''indice de la ville choisie à chaque étape'''
    choix=villes.index(depart)
    '''liste de tous les chemins parcourus'''
    chemins=[]

    '''exploration des sommets jusqu'à la ville d'arrivée'''

    while ... :
          '''la ville choisie est celle de poids minimum, elle est stockée dans la
                                                    variable visit'''
         dist_choix=
         choix=
             ...

          '''relâchement'''
          for k in range(len(M)):
            '''si la ville n'est pas déjà visitée'''
               if ... :
                 '''on relâche la ville choisie au tour précédent'''
                    '''on garde en mémoire le sommet visité, le sommet précédent
                       et le poids de l'arête'''

           '''la ville est déjà visitée, son poids devient infini'''
          dist[choix]=inf


    '''construction du chemin le plus court'''
    '''tous les chemins possibles peuvent être reconstruits à l'aide de la variable
                                                  chemin'''
    '''on repère la ville d'arrivée, dernière choisie, et on récupère le sommet
                                                  précédent'''
    '''on recommence avec le sommet que l'on vient de trouver'''
    v=villes[choix]
    chemin_choix=[v]

    while v!=depart:
         ....

    chemin_choix.reverse()

    return dist_choix,chemin_choix
```

# TRAVAUX PRATIQUES 1

# AUTOUR DES LISTES

Voici les instructions éventuellement utiles dans ce TP :

| | |
|---|---|
| `len(L)` | donne la longueur de `L` |
| `L=[]` | `L` est la liste vide |
| `L[i]` | donne la valeur de l'élément de la liste d'indice `i` |
| `L+M` | concatène (juxtapose) les listes `L` et `M` |
| `L.append(a)` | ajoute `a` à la fin de la liste |
| `L.insert(i,a)` | ajoute l'élément `a` à la $i^e$ position |
| `del L[i]` | supprime et retourne le $i^e$ élément |
| `for k in range(i,j)` | boucle **pour** `k` allant **de** `i` **à** `j` (exclus) |
| `while` | boucle **tant que** |

Le **tableau unidimensionnel** (ou **liste**) a l'avantage de regrouper un grand nombre de valeurs en une seule variable. Chaque valeur est alors est alors repérée par un numéro appelé **indice**.

Par exemple, dans un tableau `T` comportant 12 valeurs, la première valeur sera désignée par `T[0]`, la deuxième par `T[1]`,..., la dernière par `T[11]`.

1. **Recherches dans une liste**

   (a) Quel est le rôle de la fonction suivante ?

```
def fonction1(L): #L est une liste de nombres
    a=L[0]
    for k in range (len(L)):
        if L[k]>=a:
            a=L[k]
    return a
```

   (b) Modifier la fonction précédente pour obtenir le minimum de la liste L.

   (c) Dorénavant on suppose que L est une liste contenant des nombres deux à deux distincts.
   Ecrire une fonction `MAXmax(L:list)->tuple` qui renvoie le maximum et le second maximum de la liste L. Par exemple : `MAXmax([2,6,7,1,9,4]→ (9,7)`

2. **Calcul de la moyenne arithmétique**

   (a) Quel est le rôle des programmes suivants ?

```
def fonction2(n):
    '''
    * entrée : int
    * sortie : list     '''
    L=[]
    for k in range(n):
        L.append(k)
    return L
```

```
def fonction3(n:int)->int:
    S=0
    for k in range(n):
        S=S+k
    return S
```

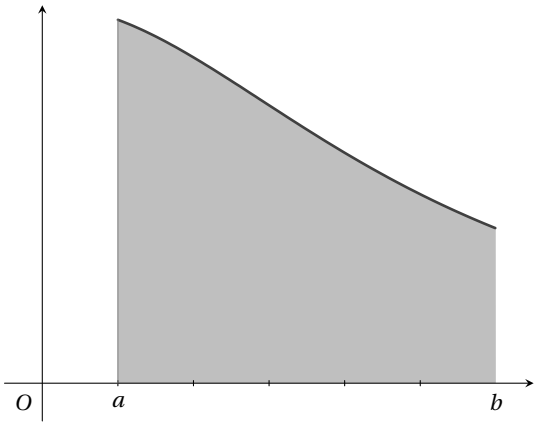
(b) Soit L une liste contenant des nombres. Ecrire une fonction `moyenne(L:list)-> float` qui renvoie la moyenne des valeurs de la liste L.

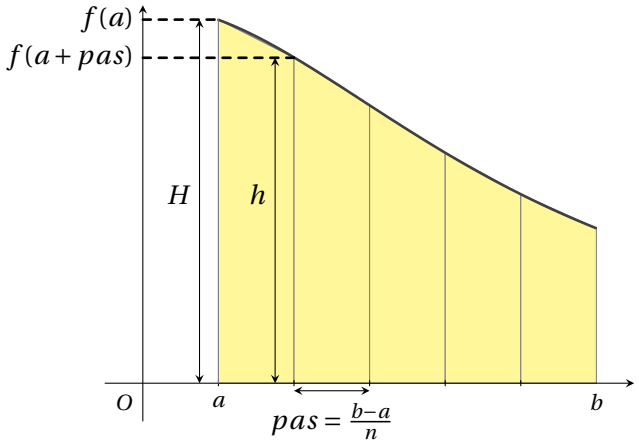
3. **Calcul de la valeur moyenne par la méthode des trapèzes**

*Méthode des trapèzes*

Par définition, l'intégrale d'une fonction continue et positive sur $[a,b]$ est l'aire (grise ci-dessous) de la surface délimitée par la courbe, l'axe des abscisses et le droites verticales d'équation $x=a$ et $x=b$. La **méthode des trapèzes** consiste à approcher l'aire sous la courbe par la somme des aires de trapèzes.

Pour cela on définit une **subdivision** à pas constant :
Soit $n \in \mathbb{N}^*$, on définit le **pas** : $pas = \frac{b-a}{n}$ et $a_k = a + k \times pas$




L'aire du premier trapèze jaune est obtenue par la formule $\frac{H+h}{2} \times pas$, où $H = f(a)$ et $h = f(a+pas)$.
La formule qui donne une approximation de l'intégrale par la somme d'aires de trapèzes est donc :

$$\sum_{k=0}^{n-1} \frac{f(a_k)+f(a_{k+1})}{2} \times pas$$

(a) Ecrire une fonction `f(x)` qui pour la variable $x$ retourne la valeur $\frac{4}{1+x^2}$.

(b) Ecrire une fonction `trapezes(f,a,b,n)` qui prend en argument une fonction $f$, deux réels $a$ et $b$, et un entier $n$ et qui renvoie la valeur approchée de l'intégrale obtenue par la méthode des trapèzes.

(c) Quelle valeur approchée obtient-on avec l'appel `trapezes(f,0,1,1000)` ?

*Calcul de la valeur moyenne*

On effectue une série de mesures sur une certaine quantité $Q$, durant une période donnée $T$.
Ces mesures sont stockées dans une liste `mesures`.
La valeur moyenne de ces mesures est définie par :

$$Q_{moy} = \frac{1}{T}\int_0^T Q(t)\,\mathrm{d}t$$

Définir une fonction `valeur_moyenne(mesures,T)`, prenant en arguments la liste `mesures` et la période $T$ et qui calcule la valeur moyenne de ces mesures obtenue par la méthode des trapèzes.

Pour comparer les deux calculs de moyenne, on prend `L=[k for k in range(1,101)]` ; comparer les appels `moyenne(L)` et `valeur_moyenne(L,100)`.

# TRAVAUX PRATIQUES 2

## RECHERCHES DANS UNE CHAÎNE

Voici les instructions éventuellement utiles dans ce TP :

| | |
|---|---|
| `len(S)` | donne la longueur de la chaîne de caractères `S` |
| `S = ''` | `S` est la chaîne vide |
| `S[i]` | donne le caractère d'indice `i` dans la chaîne |
| `S[i:j]` | donne la sous-chaîne des caractères d'indice `i` inclus à `j` exclus |
| `S+T` | concatène (juxtapose) les chaînes `S` et `T` |
| `len(L)` | donne la longueur de `L` |
| `L=[]` | `L` est la liste vide |
| `L[i]` | donne la valeur de l'élément de la liste d'indice `i` |
| `L+M` | concatène (juxtapose) les listes `L` et `M` |
| `L.append(a)` | ajoute `a` à la fin de la liste |
| `L.insert(i,a)` | ajoute l'élément `a` à la $i^e$ position |
| `del L[i]` | supprime et retourne le $i^e$ élément |
| `for k in range(i,j)` | boucle **pour** `k` allant **de** `i` **à** `j` (exclus) |
| `while` | boucle **tant que** |

1. **Rechercher un caractère dans une chaîne.**

   Ecrire une fonction `cherche(c,S)` qui prend en arguments un caractère `c` et une chaîne de caractères `S` ; si le caractère est présent dans la chaîne, cette fonction revoie `True` et la liste des indices de ce caractère ; sinon la fonction revoie False.

   Par exemple, si on considère la chaîne `S='je vois et je comprends'`, alors :

   `chercher('e',S)` → `(True,[1,8,12,19])` ; `chercher('a',S)` → `(False,[])`

2. **Rechercher un mot dans un texte.**

   Soit une chaîne de caractères `S='TGACTGGTCACT'`, on appelle sous-chaîne de caractères de `S` une suite de caractères incluse dans `S`. Par exemple, `'TGG'` est une sous-chaîne de `S` mais `'TAG'` n'est pas une sous-chaîne de `S`.

   L'objectif est de rechercher une sous-chaîne de caractères `M` de longueur $m$ appelée motif dans une chaîne de caractères `S` de longueur $n$.

   Il s'agit d'une problématique classique en informatique, qui répond aux besoins de nombreuses applications. On trouve plus de 100 algorithmes différents pour cette même tâche, les plus célèbres datant des années 1970 ; mais plus de la moitié ont moins de 10 ans.

   *Principe de l'algorithme naïf* : On parcourt la chaîne. À chaque étape, on regarde si on a trouvé le bon motif. Si ce n'est pas le cas, on recommence avec l'élément suivant de la chaîne de caractères.

Cet algorithme a une complexité en $O(nm)$ avec $n$, la taille de la chaîne de caractère et $m$, la taille du motif.

Écrire une fonction `rechercher(M,S)` qui à une sous-chaîne de caractères `M` et une chaîne de caractères `S` renvoie `False` si `M` n'est pas dans `S`, et `True` et la position de la première lettre de la chaîne de caractères `M` si `M` est présente dans `S`. Par exemple :

`rechercher('TGG',S)` → `(True,4)` ; `rechercher('TAG',S)` → `False`

# TRAVAUX PRATIQUES 3

## CRYPTOGRAPHIE

Voici les instructions éventuellement utiles dans ce TP :

| | |
|---|---|
| `len(L) ; len(S)` | donne la longueur de `L`; de la chaîne `S` |
| `L=[] ; S=''` | `L` est la liste vide; `S` est la chaîne vide |
| `L[i] ; S[i]` | donne la valeur de l'élément d'indice `i` dans la liste `L` ou la chaîne `S` |
| `M+N` | concatène (juxtapose) les listes ou les chaînes`M` et `N` |
| `L.append(a)` | ajoute `a` à la fin de la liste |
| `S+'a'` | ajoute `a` à la fin de la chaîne |
| `M.index(a)` | donne la première occurrence de `a` dans la liste ou la chaîne `M` |
| `chr(i)` | donne le caractère ASCII d'indice `i` |
| `ord('c')` | donne l'indice dans la table ASCII du caractère `c` |
| `n%p` | donne le reste de la division de `n` par `p` |
| `D={clé1:valeur1}` | `D` est le dictionnaire contenant une seule `valeur1` associé à une `clé1` |
| `D[clé]` | donne la valeur de l'élément du dictionnaire `D` associé à la `clé` |
| `D[clé2]=valeur2` | ajoute la `valeur2` associée à la `clé2` dans le dictionnaire `D` |

1. **Création d'un alphabet**

   La table ASCII contient les lettres de l'alphabet sous forme de chaînes de caractères.

   Utiliser cette table pour définir une fonction `alphabet()`, sans argument, qui renvoie une liste contenant les lettres de l'alphabet en minuscule sous la forme de chaînes de caractères. Le premier élément de cette liste contiendra le caractère *Space*, à l'indice 32 dans la table ASCII, permettant de créer l'espace entre deux mots.

2. **Codage Jules César**

   Le principe du codage est mono-alphabétique. On cherche à coder un message. Pour cela un nombre $N$ est choisi. Chaque lettre du message d'origine est alors remplacée par la lettre de l'alphabet qui est située $N$ places plus loin dans l'alphabet. Si en faisant cette manipulation, on "sort" de l'alphabet, alors on reprend l'alphabet au début. Par ailleurs, l'alphabet est complété par une lettre supplémentaire qui permettra de coder les espaces entre les mots.

   Par exemple, si le nombre $N$ choisi au départ est 5, et que le message à coder est :

   ```
   'je vois et je comprends'
   ```

   alors le message codé est :

   ```
   'oje  tnxejyeojehtruwjsix'
   ```

   (a) Ecrire une fonction `codage_cesar(message,N)`, prenant en arguments un message et un nombre et retournant le message codé en suivant le principe ci-dessus.

Ecrire une fonction `decodage_cesar(message_code,N)` prenant en arguments un message codé par la méthode César et un nombre, qui retourne le message décodé. Tester son fonctionnement sur le message suivant codé avec $N = 10$ :

$$'\texttt{obyzojaobcojsmsjocjbojaozybo}'$$

(b) Trouver, quitte à tester toutes les valeurs de $N$ possibles, le message caché derrière le message crypté :

$$'\texttt{buhhqwuphusguipuipsgneidwgqbbuphdciptuhphncdcnbuxh}'$$

3. **Codage Blaise de Vigenère**

Le principe du codage est cette fois poly-alphabétique. Un mot, appelé **clef**, est choisi au départ. Chaque lettre du message à coder sera remplacée par la lettre de l'alphabet qui est située $N$ places plus loin dans l'alphabet suivant le principe suivant :

- ⋆ pour la première lettre à coder, $N$ est le numéro correspondant à la première lettre de la clef,
- ⋆ pour la deuxième lettre à coder, $N$ est le numéro correspondant à la deuxième lettre de la clef,
- ⋆ pour la troisième lettre à coder, $N$ est le numéro correspondant à la troisième lettre de la clef...

Dès que l'on arrive à la dernière lettre de la clef, on revient au début de ce mot-clé.

En quelques sortes, tout se passe comme si on "ajoutait" les lettres du mot-clé à celles du message à coder.

(a) Ecrire une fonction `codage_vigenere(message,clef)` qui retourne le message codé suivant ce principe.

Tester le fonctionnement de la fonction avec la clef $'\texttt{python}'$ et le message suivant :

```
'je vois et je comprends'
```

(b) Ecrire une fonction `decodage_vigenere(message_code,clef)`, prenant en argument un message codé par la méthode Vigenère et la clef utilisée pour le codage, qui retourne le message décodé.

Tester le fonctionnement de la fonction avec la clef $'\texttt{informatique}'$ et le message :

```
'kwkbmronnqyfwff  wmnhwuzemsf  rvozxhgfbwwiw'
```

4. **Nombre d'apparition des lettres dans un texte**

Pour décoder un message codé avec le codage Vigenère, sans connaître la clé, il est nécessaire de faire une étude statistique sur le message codé, en comptant le nombre d'apparition de chaque lettre.

Pour réaliser ce comptage, nous allons utiliser un dictionnaire.

(a) On initialise une variable `alpha` au format dictionnaire par : `alpha={' ',0}`.

La clé est l'espace et son effectif est initialisé à 0. Compléter la variable `alpha` pour qu'elle contienne comme clés, les lettres de l'alphabet associés aux valeurs 0 :

```
>>> alpha
{' ': 0, 'a': 0, 'b': 0, 'c': 0, 'd': 0, 'e': 0, 'f': 0, 'g': 0, 'h': 0, 'i': 0
, 'j': 0, 'k': 0, 'l': 0, 'm': 0, 'n': 0, 'o': 0, 'p': 0, 'q': 0, 'r': 0, 's':
0, 't': 0, 'u': 0, 'v': 0, 'w': 0, 'x': 0, 'y': 0, 'z': 0}
```

(b) Soit `S` une chaîne de caractères. On suppose, pour simplifier, que `S` ne contient que des lettres minuscules et non accentuées.

Ecrire une fonction `compte(S:str)->dict` qui prend un argument un texte au format d'une chaîne de caractères et qui renvoie la variable `alpha` qui donne le nombre d'apparition de chaque lettre dans le texte. Par exemple :

```
S='demain des l aube a l heure ou blanchit la campagne je partirai vois tu je sais que
tu m attends j irai par la foret j irai par la montagne je ne puis demeurer loin de toi
plus longtemps je marcherai les yeux fixes sur mes pensees sans rien voir au dehors sans
entendre aucun bruit seul inconnu le dos courbe les mains croisees triste et le jour pour
```

```
moi sera comme la nuit je ne regarderai ni l or du soir qui tombe ni les voiles au loin
descendant vers harfleur et quand j arriverai je mettrai sur ta tombe un bouquet de houx
vert et de bruyere en fleur'
>>> compte(S)
{' ': 116, 'a': 37, 'b': 8, 'c': 9, 'd': 15, 'e': 69, 'f': 4, 'g': 4, 'h': 6, '
i': 34, 'j': 10, 'k': 0, 'l': 21, 'm': 15, 'n': 30, 'o': 25, 'p': 9, 'q': 4, 'r
': 41, 's': 33, 't': 26, 'u': 33, 'v': 6, 'w': 0, 'x': 3, 'y': 2, 'z': 0}
```

# TRAVAUX PRATIQUES 4

# ALGORITHMES DICHOTOMIQUES

Voici les instructions éventuellement utiles dans ce TP :

| | |
|---|---|
| `len(L)` | donne la longueur de `L` |
| `L=[]` | `L` est la liste vide |
| `L[i]` | donne la valeur de l'élément de la liste d'indice `i` |
| `L+M` | concatène (juxtapose) les listes `L` et `M` |
| `L.append(a)` | ajoute `a` à la fin de la liste |
| `L.insert(i,a)` | ajoute l'élément `a` à la $i^e$ position |
| `del L[i]` | supprime et retourne le $i^e$ élément |
| `for k in range(i,j)` | boucle **pour** `k` allant **de** `i` (inclus) **à** `j` (exclus) |
| `while` | boucle **tant que** |
| `randrange(a,b)` | renvoie un nombre entier `n` aléatoire tel que $a \leqslant n < b$ |
| `randint(a,b)` | renvoie un nombre entier `n` aléatoire tel que $a \leqslant n \leqslant b$ |
| `L.sort()` | renvoie la liste `L` triée dans l'ordre croissant |

1. **Recherche dichotomique dans une liste triée**

   (a) Que renvoie les instructions suivantes ?

```
L=[]
n=1000
for k in range(n):
    L.append(randint(0,1000))
L.sort()
```

   (b) Expliquer le rôle et le fonctionnement de la fonction suivante :

```
def fonction(L,x):
    for k in range(len(L)):
        if L[k]==x:
            return True,k
    return False
```

   (c) Si on décide de rechercher un mot dans un dictionnaire qui comporte 40 000 mots avec la méthode précédente, au pire des cas il faudra 40 000 tours de boucle !

   Or, le dictionnaire est trié par ordre alphabétique ; une autre méthode consiste à comparer le mot à chercher avec celui se trouvant au milieu du dictionnaire. Si le mot à chercher est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dans la première moitié du dictionnaire. Sinon, on le cherche dans la deuxième moitié.

Et on recommence...

Voici ce que cela donne en terme de nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire.

- ⋆ Au départ, on cherche le mot parmi 40 000.
- ⋆ Après le test n°1, on ne le cherche plus que parmi 20 000.
- ⋆ Après le test n°2, on ne le cherche plus que parmi 10 000.
- ⋆ Après le test n°3, on ne le cherche plus que parmi 5 000.
- ⋆ et ainsi de suite ...
- ⋆ Après le test n°15, on ne le cherche plus que parmi 2.
- ⋆ Après le test n°16, on ne le cherche plus que parmi 1.

Et là, on sait que le mot n'existe pas : on a obtenu notre réponse en 16 opérations contre 40 000 précédemment.

Pour un tableau comportant $n$ éléments, le programme s'arrête lorsque $\frac{n}{2^k} = 1$, où $k$ est le nombre d'étapes.

On obtient $k = \frac{\ln n}{\ln 2}$ ; et la complexité du programme est de l'ordre de $\ln n$. (On écrit $\log(n)$ en informatique.)

⚠ la recherche dichotomique ne peut s'effectuer que sur des éléments préalablement triés.

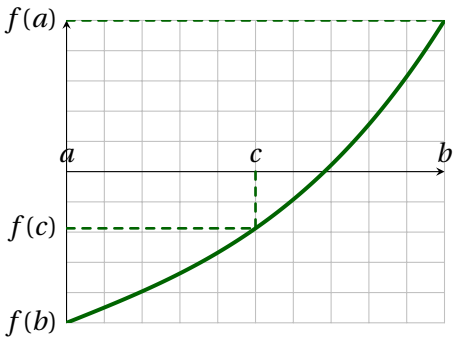
Ecrire une fonction `reccherche_dichotomie(L,x)` qui prend en argument une liste triée et un élément à chercher par dichotomie dans la liste. Cette fonction renvoie `True` et l'indice de `x` si celui-ci est dans la liste, et `False` sinon.

2. **Résolution dichotomique de l'équation** $f(x) = 0$

L'objectif de cette question est de programmer une méthodes numériques qui permet la résolution de l'équation $f(x) = 0$.

Pour illustrer cette méthode, on considère la fonction $f$ définie sur $\mathbb{R}$ par

$$f(x) = x^3 + x - 1$$




(a) Déterminer $f'(x)$, $\forall x \in \mathbb{R}$.

(b) Etudier les variations de la fonction $f$ sur $\mathbb{R}$.

(c) Démontrer que l'équation $f(x) = 0$ admet une unique solution et localiser celle-ci entre deux entiers consécutifs $a$ et $b$.

(d) On pose $c = \frac{a+b}{2}$.
Calculer $f(c)$ ; que peut-on en déduire ?

(e) Créer les fonctions python `f` et `fprime`.

(f) Ecrire une fonction `Dichotomie(f,p,a,b)`, prenant en argument la fonction $f$, un entier $p$ et les bornes $a$ et $b$ de l'intervalle sur lequel on applique la méthode de dichotomie. Cette fonction doit retourner une valeur approchée à $10^{-p}$ près de la solution de l'équation $f(x) = 0$ .

# TRAVAUX PRATIQUES 5

## TRIS QUADRATIQUES

Voici les instructions éventuellement utiles dans ce TP :

| | |
|---|---|
| `len(L)` | donne la longueur de `L` |
| `L=[]` | `L` est la liste vide |
| `L[i]` | donne la valeur de l'élément de la liste d'indice `i` |
| `L+M` | concatène (juxtapose) les listes `L` et `M` |
| `L.append(a)` | ajoute `a` à la fin de la liste |
| `L.insert(i,a)` | ajoute l'élément `a` à la $i^e$ position |
| `del L[i]` | supprime et retourne le $i^e$ élément |
| `for k in range(i,j)` | boucle **pour** `k` allant **de** `i` (inclus) **à** `j` (exclus) |
| `while` | boucle **tant que** |
| `randrange(a,b)` | renvoie un nombre entier `n` aléatoire tel que $a \leqslant n < b$ |
| `randint(a,b)` | renvoie un nombre entier `n` aléatoire tel que $a \leqslant n \leqslant b$ |

1. **Tri par sélection**

   Pour trier un tableau `T` comportant $n$ éléments par **sélection**, voici la méthode :

   En partant de la position `i` $= 0$, on recherche le plus petit élément parmi les éléments d'indice `i` $+1$ à `n` $-1$ et on l'échange avec `T[i]`.

   Par exemple, pour trier $[12,3,17,9,4,16]$, on obtient successivement :

   $$[12,3,17,9,4,16] \quad [12,|\boxed{3}|,17,9,4,16] \quad [3,12,|17,9,|\boxed{4}|,16] \quad [3,4,17,|\boxed{9}|,12,16]$$
   $$[3,4,9,17,|\boxed{12}|,16] \quad [3,4,9,12,|17,|\boxed{16}|] \quad [[3,4,9,12,17,|\boxed{16}|] \quad [3,4,9,12,16,17]$$

   Compléter la fonction suivante pour qu'elle retourne le tableau trié par sélection :

```
def tri_selection(T):
    for i in range(len(T)-1):
    ...
        for j in range (i+1,len(T)):
            # recherche du minimum dans le tableau restant
            # comparer ce minimum à T[i]
    return T
```

2. **Tri par insertion**

   Pour trier un tableau `T` par **insertion**, voici la méthode :

On prend le premier élément et on le met à l'indice `i = 0` ; puis on insère les autres éléments dans la partie déjà triée en plaçant chaque nouvel élément à la bonne place.

Cela donne :

$$[12,|3,17,9,4,16] \quad [3,12,|17,9,4,16] \quad [3,12,17,|9,4,16]$$

$$[3,9,12,17,|4,16] \quad [3,4,9,12,17,|16] \quad [3,4,9,12,16,17]$$

Ecrire une fonction python qui permette de trier le tableau par insertion, en complétant le script suivant :

```
def tri_insertion(T):
    for i in range(1,len(T)):#on traite les éléments restants
        ...                  #on mémorise l'élément à traiter
        ...                  #variable créée pour trouver la bonne place
        while ...            #tant que la bonne place n'est pas trouvée
              ...            #on cherche la bonne place
        ...                  #on insère l'élément à sa place
        ...                  #on supprime le doublons
    return T
```

Evaluer la complexité temporelle des deux algorithmes de tri.

3. **Tri à bulles**

   Soit `T` un tableau de valeurs, décrire l'évolution de la variable `T` au cours de l'algorithme suivant :

```
def tri_a_bulles(T):
    for i in range(len(T)-1):
        for j in range(len(T)-1,i,-1):
            if T[j]<T[j-1]:
                T[j],T[j-1]=T[j-1],T[j]
    return T
```

# TRAVAUX PRATIQUES 6

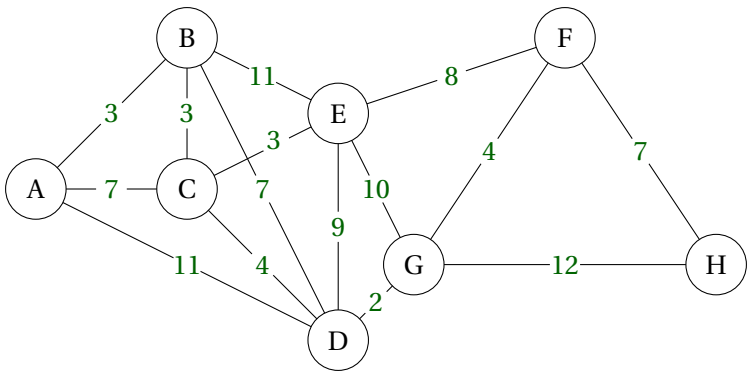
## GRAPHES

| | |
|---|---|
| `len(L)` | donne la longueur de `L` |
| `L=[]` | `L` est la liste vide |
| `L[i]` | donne la valeur de l'élément de la liste d'indice `i` |
| `L+T` | concatène (juxtapose) les listes `L` et `T` |
| `L.append(a)` | ajoute `a` à la fin de la liste |
| `L.insert(i,a)` | ajoute l'élément `a` à la $i^e$ position |
| `del L[i]` | supprime le $i^e$ élément |
| `for k in range(i,j)` | boucle **pour** `k` allant **de** `i` (inclus) **à** `j` (exclus) |
| `while` | boucle **tant que** |
| `M=array([L1,L2,...])` | `M` est la matrice constituée des lignes `L1`,`L2`... |
| `len(M)` | donne le nombre de lignes de `M` |
| `M[i][j]` | donne l'élément situé à la ligne `i` et à la colonne `j` |

On peut implémenter une matrice en Python par l'instruction `array` à importer de la bibliothèque `numpy` :

```
from numpy import array
```

On considère la graphe pondéré suivant :




1. Donner une instruction python qui permet de construire la *matrice d'adjacence* $M = \left(m_{i,j}\right)$ où $m_{i,j}$ est égal à la valeur portée par l'arête reliant les sommets d'indices $i$ et $j$ si ces sommets sont adjacents, ou $\infty$ sinon.

Pour cela, on considère la liste des sommets : `S=['A','B','C','D','E','F','G','H']` ; les indices des sommets dans la matrice `M` doivent correspondre aux indices des sommets dans la liste `S`.

On importera $\infty$ de la bibliothèque `numpy` par : `from numpy import inf`.

2. Ecrire une fonction python `voisins(M:array,S:list,S0:str)->list`, prenant en arguments la matrice $M$ d'adjacence d'un graphe, la liste $S$ de ses sommets, et un sommet $S_0$ de ce graphe. Cette fonction renvoie la liste des voisins du sommet $S_0$.

   Par exemple : `voisins(M,S,'A')` $\longrightarrow$ `['B','C','D']`.

3. Ecrire une fonction `degre(M,S,S0)`, de mêmes arguments que la fonction précédente, qui renvoie le nombre de voisins du sommet $S_0$ ; c'est-à-dire le nombre d'arêtes issues de $S_0$.

   Par exemple, `degre(M,S,'A')` $\longrightarrow$ `3`.

4. Pour optimiser l'étude de certaines situations, il est parfois imoprtant de trier les sommets par ordre croissant de degré.

   Ecrire alors une fonction `triSommets(M,S)` qui revoie la liste des sommets triée par ordre croissant des degrés.

   Par exemple, `triSommets(M,S)` $\longrightarrow$ `['H','A','F','B','C','G','D','E']`.

5. Ecrire une fonction `longChemin(M,S,L)` qui prend un arguments une matrice $M$, la liste $S$ de ces sommets et une liste $L$ contenant des sommets (adjacents ou non). Cette fonction renvoie la longueur de ce chemin s'il est réalisable ou $\infty$ sinon.

   Par exemple, `longChemin(M,S,['A','C','E','F'])` $\longrightarrow$ `18`.

# TRAVAUX PRATIQUES 7

## ALGORITHME DE GAUSS-JORDAN

On peut implémenter une matrice en Python par l'instruction `np.array` après avoir importé la bibliothèque `numpy` par l'instruction : `import numpy as np`. Vous trouverez en annexe des instructions utiles pour manipuler les matrices.

Pour ce TP, nous allons opter pour une **programmation modulaire**; ainsi toutes les fonctions auxiliaires seront enregistrées dans un fichier que nous appellerons `gausspy`.

Le fichier principal qui contiendra l'algorithme de Gauss-Jordan sera nommé `gauss` et nous importerons le module par l'instruction `import gausspy as gp`; ainsi toutes les fonctions contenues dans `gausspy` pourrons être appelées sous la forme `gp.NomDeLaFonction`.

Nombreux sont les logiciels permettant de résoudre un système d'équations linéaires.

On se propose ici de programmer l'algorithme de **Gauss-Jordan** de résolution d'un système linéaire de $n$ équations à $n$ inconnues.

On considère le système linéaire de $n$ équations à $n$ inconnues suivant :

$$(\Sigma)\ \begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \quad (L_1) \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \quad (L_2) \\ \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \quad (L_n) \end{cases}$$

On note $A$ la matrice des cœfficients associée à $(\Sigma)$, $B$ la matrice du second membre et $X$ la matrice des inconnues. Le système $(\Sigma)$ s'écrit alors : $AX = B$ ou encore :

$$(A|B) = \begin{pmatrix} a_{11} & \cdots & a_{1j} & \cdots & a_{1n} & b_1 \\ \vdots & & \vdots & & \vdots & \vdots \\ a_{i1} & \cdots & a_{ij} & \cdots & a_{in} & b_i \\ \vdots & & \vdots & & \vdots & \vdots \\ a_{n1} & \cdots & a_{nj} & \cdots & a_{nn} & b_n \end{pmatrix}$$

On appelle cette dernière matrice la **matrice augmentée** du système $(\Sigma)$.

Afin de simplifier la mise en œuvre de la méthode de Gauss, nous allons faire l'hypothèse que la matrice $A$ est inversible, c'est-à-dire que le système est de Cramer.

Il admet ainsi une unique solution donnée par $X = A^{-1}B$.

1. *Création de la matrice augmentée*

   Ecrire une fonction python `Augmente(A,B)` qui prend en arguments une matrice $A$ carrée de taille $n$ et une matrice colonne $B$ à $n$ lignes. Cette fonction retourne la matrice augmentée $(A|B)$ associé au système à résoudre.

2. Ecrire une fonction `EchangeLigne(M,i,j)` qui prend en arguments une matrice $M$ et deux entiers $i$ et $j$. Cette fonction doit retourner la matrice dans laquelle les lignes $i$ et $j$ ont été échangées.
3. *Recherche du pivot*

   L'hypothèse d'inversibilité de la matrice $A$ assure l'existence d'un pivot non nul sur chaque colonne $i$. On se propose d'écrire une fonction `Pivot(M,i)` qui prend en arguments une matrice $M$ et un entier $i$. Quitte à utiliser `EchangeLigne(M,i,j)`, cette fonction devra retourner la matrice dans laquelle le pivot se trouve au bon endroit.

   Il faut donc se poser la question du choix du pivot. Ce choix doit être fait avec l'objectif de minimiser les erreurs d'arrondis.

   Pour illustrer l'incidence du choix du pivot sur les erreurs d'arrondis, étudions l'exemple suivant :

$$\begin{cases} 10^{-4}x_1 + x_2 &= 1 \\ x_1 + x_2 &= 2 \end{cases}$$

   ⋆ La résolution exacte donne :

$$x_1 = \frac{-1}{10^{-4}-1} \simeq 1,00010001... \text{ et } x_2 = \frac{2\times 10^{-4}-1}{10^{-4}-1} \simeq 0,99989998...$$

   Si on utilise des flottants à trois chiffres significatifs, on obtient $x_1 \simeq 1$ et $x_2 \simeq 1$.

   ⋆ Résolution avec $10^{-4}$ comme pivot : le système devient : $\begin{cases} 10^{-4}x_1 + x_2 &= 1 \\ (1-10^4)x_2 &= 2-10^4 \end{cases}$

   donc $x_2 = \frac{2-10^4}{1-10^4} = \frac{9998}{9999} = \frac{9,998\times 10^3}{9,999\times 10^3}$ ; avec 3 chiffres significatifs, cela donne $x_2 = 1$ et, en reportant dans la première équation, on obtient $x_1 = 0$!!!

   ⋆ Résolution avec 1 comme pivot : le système devient : $\begin{cases} x_1 + x_2 &= 2 \\ (1-10^{-4})x_2 &= 1-2\times 10^{-4} \end{cases}$

   donc $x_2 = \frac{1-2\times 10^{-4}}{1-10^{-4}} = \frac{0,9998}{0,9999} = \frac{9,998\times 10^{-3}}{9,999\times 10^{-3}}$ ; avec trois chiffres significatifs, cela donne $x_2 = 1$ et, en reportant dans la première équation, $x_1 = 1$.

   Cet exemple montre que le choix de petits pivots peut engendrer des erreurs d'arrondis catastrophiques (comme obtenir $x_1 = 0$ ici!).

   Pour éviter une trop grand instabilité de l'algorithme, nous allons donc choisir comme pivot sur la colonne $i$ le cœfficient $a_{ij}$ ayant la plus grande valeur absolue.

   Ecrire alors la fonction `Pivot(M,i)` qui retourne la matrice dans laquelle le pivot est au bon endroit.
4. *Echelonnement de la matrice*

   Ecrire une fonction `ElimineD(M,i,j,l)` qui prend en arguments une matrice $M$, deux entiers $i$ et $j$ et un réel $\ell$. Cette fonction doit retourner une matrice dans laquelle la ligne $L_j$ est remplacée par la ligne $L_j - \ell \times L_i$ pour $j > i$. Cette fonction pourra être utilisée pour échelonner la matrice augmentée $(A|B)$.
5. *Normalisation de la matrice échelonnée*

   Une fois la matrice échelonnée, il faut "remonter" pour calculer les $x_i$. Pour commencer on normalise la matrice en opérant sur les lignes afin que tous les cœfficients diagonaux valent 1.

   Ecrire alors une fonction `Normalise(M)` qui effectue cette opération.
6. *Remontée*

   La dernière ligne donne la valeur de $x_n$. On utilise le pivot valant 1 sur la dernière ligne et la dernière colonne pour éliminer $x_n$ dans les lignes $L_{n-1}, L_{n-2}, ..., L_1$.

   La ligne $L_{n-1}$ donne alors la valeur de $x_{n-1}$ ; puis on élimine $x_{n-1}$ dans les lignes $L_{n-2}, ..., L_1$. Et ainsi de suite...

   Ecrire alors une fonction `ElimineM(M,i,j,l)` qui prend en arguments une matrice $M$, deux entiers $i$ et $j$ et un réel $\ell$. Cette fonction doit retourner une matrice dans laquelle la ligne $L_j$ est remplacée par la ligne $L_j - \ell \times L_i$ pour $j < i$. Cette fonction doit permettre la remontée dans la matrice augmentée et le calcul de chaque valeur de $x_i$. On obtiendra ainsi une matrice augmentée $(A|B)$ dans laquelle la matrice $A$ est diagonale.

7. *Obtention du résultat*

   La matrice augmentée $(A|B)$ est maintenant telle que la matrice $A$ est diagonale et que tous les cœfficients sont égaux à 1. Il suffit d'extraire le second membre de cette matrice pour obtenir la solution.

   Ecrire alors une fonction `SecondMembre(M)` qui permette d'extraire, sous la forme d'une liste, la dernière colonne d'un matrice $M$.

Toutes les fonctions précédente sont enregistrées dans le module `gausspy` et pourront donc être utilisées dans l'algorithme de Gauss.

8. *Algorithme de Gauss*

   En utilisant correctement les fonctions du module `gausspy` , écrire l'algorithme qui pour une matrice $A$ carrée de taille $n$ et une matrice colonne $B$ à $n$ lignes donne la solution du système de Cramer correspondant.

9. *Application.*

   Utiliser cet algorithme pour résoudre le système :

$$\begin{cases} x+y+z=6 \\ -2x-y+z=-1 \\ 3x+2y-2z=1 \end{cases}$$

10. *Approfondissement.*

    Ecrire une fonction `Comat(M,i,j)` qui prend en argument une matrice carrée sous la forme d'un tableau (`array`) et deux entiers `i` et `j`. Cette fonction doit retourner la matrice obtenue à partir de `M` dans laquelle on a retiré la ligne d'indice `i` ligne et la colonne d'indice `j`.

    Programmer alors une fonction **récursive** `Det(M)` qui prend un argument une matrice carrée sous la forme d'un tableau et qui retourne le **déterminant** de cette matrice.

## ANNEXE

Une matrice peut être considérée comme un tableau (`array`) à deux dimensions. La bibliothèque `numpy` est spécialisée dans la manipulation de ces tableaux.

1. *Création d'une matrice.*

   Au préalable, il ne faut pas oublier d'importer le module `numpy`.

```
import numpy as np
A=np.array([[1,2,3],[4,5,6]])
print (A)
print(type(A)) #type de structure
print(A.shape) #nombre de lignes et
                    colonnes
```

```
[[1. 2. 3.]
 [4. 5. 6.]]

<type 'numpy.ndarray'>
(2, 3)
```

   Il est possible de créer une matrice à partir d'une liste de valeurs ; voici quelques exemples :

```
A=np.arange(0,10).reshape(2,5)
print (A)
```

```
[[0.1. 2. 3. 4.]
 [5. 6. 7. 8. 9.]]
```

```
B=np.array([1,2,3,4,5,6])
C=B.reshape(3,2)
print (C)
```

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]
```

```
L=[1.2,1.3,1.4,1.5]
D=np.asarray(L).reshape(2,2)
print (D)
```

```
[[ 1.2  1.3]
 [ 1.4  1.5]]
```

Redimensionnement d'une matrice :

```
A=np.array([[1,2,3],[4,5,6]])
print (np.resize(A,new_shape=(3,2)))
```

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]
```

2. *Opérations sur les matrices.*

On peut additionner deux matrices de mêmes dimensions et multiplier une matrice par un scalaire :

```
A=np.arange(1,7).reshape(2,3)
B=np.array([[6,5,4],[3,2,1]])
print (A+B)
print (2*A)
```

```
[[7. 7. 7.]
 [7. 7. 7.]]
 [[ 2.  4.  6.]
 [ 8. 10. 12.]]
```

On peut accoler un vecteur `B` en tant que nouvelle ligne (`axis` = 0) ou en tant que nouvelle colonne (`axis` = 1) :

```
A=array([[1,3],[-2,2]])
B=array([[4,5]])
print (append(A,B,axis=0))
```

```
  [[ 1.  3.]
   [-2.  2.]
   [ 4.  5.]]
```

```
A=np.array([[1,3],[-2,2]])
B=np.array([[4],[5]])
print (np.append(A,B,axis=1) )
```

```
[[ 1.  3.  4.]
 [-2.  2.  5.]]
```

Insertion de `B` en tant que nouvelle ligne (`axis` = 0) à la position d'indice 1 :

```
A=np.array([[1,3],[-2,2]])
B=np.array([[4,5]])
print (np.insert(A,1,B,axis=0))
```

```
  [[ 1.  3.]
   [4.  5.]
   [ -2.  2.]]
```

Suppression de la ligne (`axis` = 0) via son indice (n°1) :

```
A=np.array([[1,3],[4.,5.],[-2,2]])
print (np.delete(A,1,axis=0))
```

```
  [[ 1.  3.]
  [-2. 2.]]
```

3. *Extractions des valeurs d'une matrice.*

```
A=np.array([[1,2,3],[4,5,6],[7,8,9]])
print (A)
```

```
  [[1. 2. 3.]
   [4. 5. 6.]
   [7. 8. 9.]]
```

Extraction d'une valeur de la matrice via son indice `A[i][j]` ou `A[i,j]` renvoie le terme situé à la ligne d'indice `i` et à la colonne d'indicie `j`.

```
print (A[0][0])
print (A[2][1])
print (A[len(A)-1][len(A)-1])
```

```
1.
8.
9.
```

Extraction d'une partie de la matrice, de la ligne 0 à la ligne 2 (non incluse) et de la colonne 0 à la colonne 2 (non incluse) :

```
print (A[0:2,0:2])
```

```
[[1. 2.]
 [4. 5.]]
```

Extraction des valeurs d'une ligne ou d'une colonne et retournées sous la forme d'une matrice ligne :

```
print (A[2,0:len(A)]) #dernière ligne
#ou plus simplement :
print (A[0]) #première ligne
print (A[0:len(A),2])#dernière colonne
```

```
[7. 8. 9.]

[1. 2. 3.]
[3. 6. 9.]
```

TRAVAUX PRATIQUES 8

# LA RÉCURSIVITÉ

## La factorielle

1. Programmer une fonction récursive `factorielle(n)` non terminale qui prend en argument un entier naturel $n$ et qui renvoie la factorielle de $n$. Evaluer la complexité de cette fonction.
2. Modifier la fonction précédente pour qu'elle devienne une fonction récursive terminale. Evaluer la complexité de cette dernière fonction et commenter le résultat.
3. **Intérêt de la dichotomie.**

   Le 31 décembre 2009, Fabrice BELLARD publie un nouveau record avec presque 2 700 milliards de décimales du nombre $\pi$. Et ceci avec un ordinateur personnel classique!

   Pour optimiser la complexité des calculs, il utilise le principe de dichotomie.

   Voici le principe : pour calculer le produit de tous les entiers compris entre $a$ et $b$, on effectue les produits des entiers entre $a$ et $c$, puis entre $c$ et $b$, où $c$ est situé au milieu; puis on effectue le produit des deux résultats. Par exemple, $\texttt{prod}(6,18) = \texttt{prod}(6,12) \times \texttt{prod}(13,18)$.

   Programmer la fonction récursive `prod(a,b)` qui effectue le produit de tous les entiers compris entre $a$ et $b$. La factorielle de $n$ s'obtiendra alors par l'appel `prod(1,n)`.

## Motifs

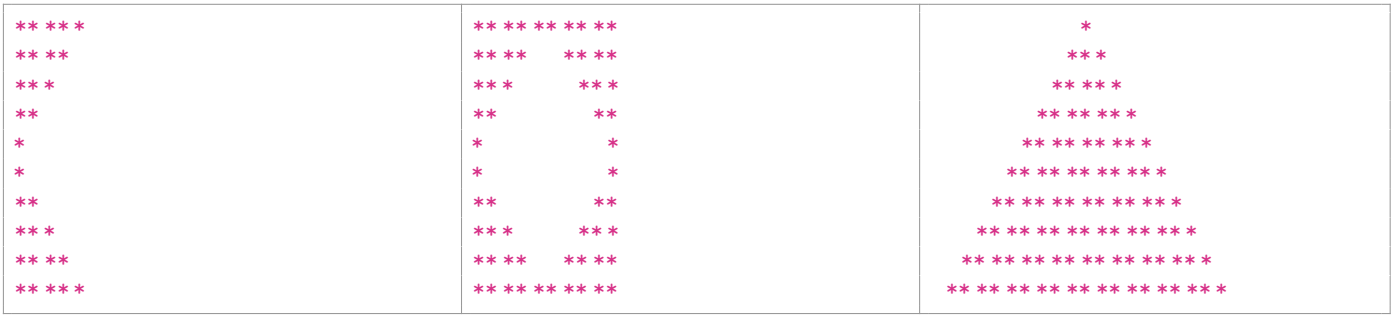
1. Qu'affiche l'algorithme suivant :

```
def fonc1(n):
    if n>0:
        print('*'*n)
        fonc1(n-1)
```

2. Modifier la fonction précédente pour qu'elle affiche :

```
>>> fonc2(5)
*
**
***
****
*****
```

3. Utiliser le principe précédent pour afficher les motifs suivants :



TRAVAUX PRATIQUES 9

# DYNAMIQUE GRAVITATIONNELLE

Modéliser les interactions physiques entre un grand nombre de constituants mène à l'écriture de systèmes différentiels pour lesquels, en dehors de quelques situations particulières, il n'existe aucune solution analytique. Les problèmes de dynamique gravitationnelle et de dynamique moléculaire en sont deux exemples. Afin d'analyser le comportement temporel de tels systèmes, l'informatique peut apporter une aide substantielle en permettant leur simulation numérique. L'objet de ce TP, inspiré d'un sujet de CENTRALE, est l'étude de solutions algorithmiques en vue de simuler une dynamique gravitationnelle afin, par exemple, de prédire une éclipse ou le passage d'une comète.

Soit $y$ une fonction de classe $\mathscr{C}^2$ sur $\mathbb{R}$ et $t_{\min}$ et $t_{\max}$ deux réels tels que $t_{\min} < t_{\max}$.
On note $I$ l'intervalle $[t_{\min}, t_{\max}]$. On s'intéresse à une équation différentielle du second ordre de la forme :

$$\forall t \in T \qquad y''(t) = f(y(t)) \tag{1}$$

où $f$ est une fonction donnée, continue sur $\mathbb{R}$. De nombreux systèmes physiques peuvent être décrits par une équation de ce type.

On suppose connues les valeurs $y_0 = y(t_{\min})$ et $z_0 = y'(t_{\min})$. On suppose également que le système physique étudié est conservatif. Ce qui entraine l'existence d'une quantité indépendante du temps (énergie, quantité de mouvement,...), notée $E$, qui vérifie l'équation (2) où $g' = -f$ :

$$\forall t \in I \qquad \frac{1}{2}y'(t)^2 + g(y(t)) = E \tag{2}$$

## Mise en forme du problème

Pour résoudre numériquement l'équation différentielle (1), on introduit la fonction $z : I \to \mathbb{R}$ définie par $\forall t \in I,\ z(t) = y'(t)$.

**1.** Montrer que l'équation (1) peut se mettre sous la forme d'un système différentiel du premier ordre en $z(t)$ et $y(t)$, noté $(S)$.

**2.** Soit $n$ un entier naturel strictement supérieur à 1 et $J_n = [\![0, n-1]\!]$.
On pose $h = \frac{t_{\max} - t_{\min}}{n-1}$ et $\forall i \in J_n,\ t_i = t_{\min} + ih$. Montrer que, pour tout entier $i \in [\![0, n-2]\!]$,

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} z(t)\,\mathrm{d}t \quad \text{et} \quad z(t_{i+1}) = z(t_i) + \int_{t_i}^{t_{i+1}} f(y(t))\,\mathrm{d}t \tag{3}$$

La suite du problème exploite les notations introduites dans cette partie et présente deux méthode numériques dans lesquelles les intégrales précédentes sont remplacées par une valeur approchée.

## Schéma d'Euler explicite

Dans le schéma d'Euler explicite, chaque terme sous le signe intégral est remplacé par sa valeur prise en la borne inférieure.

**3.** Dans ce schéma, montrer que les équations (3) permettent de définir deux suites $(y_i)_{i\in J_n}$ et $(z_i)_{i\in J_n}$, où $y_i$ et $z_i$ sont des valeurs approchées de $y(t_i)$ et de $z(t_i)$. Donner les relations de récurrence permettant de déterminer les valeurs de $y_i$ et $z_i$ connaissant $y_0$ et $z_0$.

**4.** Pour illustrer cette méthode, on considère l'équation différentielle

$$\forall t \in I,\ y''(t) = -\omega^2 y(t) \tag{4}$$

dans laquelle $\omega$ est un nombre réel.

Ecrire l'équation de conservation (2) correspondante à l'équation différentielle (4).
En portant les valeurs de $y_i$ et $z_i$ sur l'axe des abscisses et l'axe des ordonnées respectivement, quelle serait l'allure du graphe qui respecte la conservation de $E$ ?
La mise en œuvre de la méthode d'Euler explicite génère le résultat graphique donné **Figure 1** à gauche.

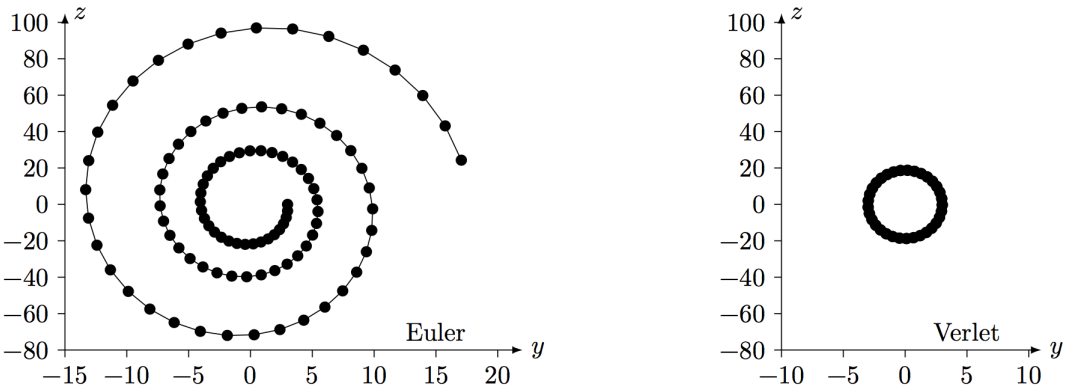



**Figure 1**

En quoi ce graphe confirme-t-il que le schéma numérique ne conserve par $E$ ? Pouvez-vous justifier son allure ?

**5.** Déterminer les suites $(y_i)_{i\in J_n}$ et $(z_i)_{i\in J_n}$ obtenues à la question **3.** qui correspondent à l'équation différentielle (4).
Ecrire alors une fonction `euler` qui reçoit en arguments les paramètres qui vous semblent pertinents et qui renvoie deux listes de nombres correspondant aux valeurs associées aux suites $(y_i)_{i\in J_n}$ et $(z_i)_{i\in J_n}$.
Pour illustrer cette méthode, on choisir les valeurs numériques suivantes :

$$y_0 = 3,\ z_0 = 0,\ t_{\min} = 0,\ t_{\max} = 3,\ \omega = 2\pi \text{ et } n = 100$$

La courbe pourra être obtenue par :

```
import matplotlib.pyplot as plt
import numpy as np
plt.close()
Y=euler(3,0,2*np.pi,0,3,100)[0]
Z=euler(3,0,2*np.pi,0,3,100)[1]
plt.plot(Y,Z,'ko',linestyle='-')
ax = plt.gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
plt.xlim(-15, 20)
plt.ylim(-80,120)
plt.show()
```

## Schéma de Verlet

Le physicien français Loup Verlet a proposé en 1967 un schéma numérique d'intégration d'une équation de la forme (5) dans lequel, en notant $f_i = f(y_i)$ et $f_{i+1} = f(y_{i+1})$, les relations de récurrence s'écrivent

$$y_{i+1} = y_i + hz_i + \frac{h^2}{2} f_i \quad \text{et} \quad z_{i+1} = z_i + \frac{h}{2}\left(f_i + f_{i+1}\right) \tag{5}$$

On reprend l'exemple de l'oscillateur harmonique de la question **4.** et on compare les résultats obtenus à l'aide des schémas d'Euler et de Verlet.

**6.** Ecrire une fonction `verlet` qui reçoit en arguments les paramètres qui vous semblent pertinents et qui renvoie deux listes de nombres correspondant aux valeurs associées aux suites $\left(y_i\right)_{i\in J_n}$ et $\left(z_i\right)_{i\in J_n}$.

**7.** La mise en œuvre du schéma de Verlet avec les mêmes paramètres que ceux utilisés au **5.** donne le résultat de la **Figure 1** à droite. Interpréter l'allure de ce graphe.
Que peut-on en conclure sur le schéma de Verlet ?

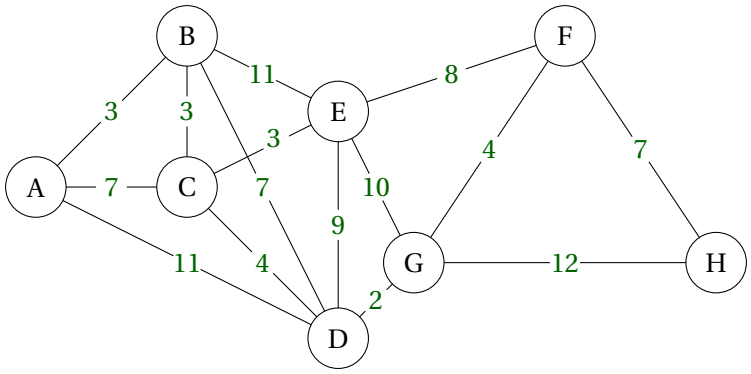
# TRAVAUX PRATIQUES 10

## GRAPHES BIS

On peut implémenter une matrice en Python par l'instruction `array` à importer de la bibliothèque `numpy` :

```
from numpy import array
```

On considère la graphe pondéré suivant :




1. Donner une instruction python qui permet de construire la *matrice d'adjacence* $M = (m_{i,j})$ où $m_{i,j}$ est égal à la valeur portée par l'arête reliant les sommets d'indices $i$ et $j$ si ces sommets sont adjacents, ou $\infty$ sinon.

   Pour cela, on considère la liste des sommets : `S=['A','B','C','D','E','F','G','H']` ; les indices des sommets dans la matrice `M` doivent correspondre aux indices des sommets dans la liste `S`.

   On importera $\infty$ de la bibliothèque `numpy` par : `from numpy import inf`.

2. Ecrire une fonction python `voisins(M:array,S:list,S0:str)->list`, prenant en arguments la matrice $M$ d'adjacence d'un graphe, la liste $S$ de ses sommets, et un sommet $S_0$ de ce graphe. Cette fonction renvoie la liste des voisins du sommet $S_0$.

   Par exemple : `voisins(M,S,'A')` $\longrightarrow$ `['B','C','D']`.

3. Ecrire une fonction `degre(M,S,S0)`, de mêmes arguments que la fonction précédente, qui renvoie le nombre de voisins du sommet $S_0$ ; c'est-à-dire le nombre d'arêtes issues de $S_0$.

   Par exemple, `degre(M,S,'A')` $\longrightarrow$ `3`.

4. Pour optimiser l'étude de certaines situations, il est parfois imoprtant de trier les sommets par ordre croissant de degré.

   Ecrire alors une fonction `triSommets(M,S)` qui revoie la liste des sommets triée par ordre croissant des degrés.

   Par exemple, `triSommets(M,S)` $\longrightarrow$ `['H','A','F','B','C','G','D','E']`.

5. Ecrire une fonction `longChemin(M,S,L)` qui prend un arguments une matrice $M$, la liste $S$ de ces sommets et une liste $L$ contenant des sommets (adjacents ou non). Cette fonction renvoie la longueur de ce chemin s'il est réalisable ou $\infty$ sinon.

   Par exemple, `longChemin(M,S,['A','C','E','F'])` $\longrightarrow$ `18`.

6. Ecrire une fonction `dijkstra(villes,départ,arrivée,matrice)` qui prend en argument une liste (`list`) de villes, le nom (`str`) de la ville de départ, le nom (`str`) de la ville d'arrivée et la matrice (`array`) d'adjacence du graphe ; cette fonction renvoie le plus court chemin entre les deux villes choisies.

   Par exemple : `dijkstra(S,'A','H',M)` $\longrightarrow$ `(23.0,['A','B','D','G','F','H'])`

TRAVAUX PRATIQUES 11

# INTERPOLATION DE LAGRANGE

## Opérations sur les polynômes

On considère un polynôme de degré $n$ sous la forme $P(X) = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$.

En Python, un polynôme est implémenté sour la forme d'une liste : `P = [a_0,a_1,...,a_n]` ; par exemple, le polynôme $P(X) = X^2 + 2X + 3$ sera implémenté par : `P = [3,2,1]`.

1. **Création d'un polynôme**

   Dans les algorithmes suivants, nous aurons besoin d'initialiser un polynôme d'un degré donné. Pour cela, nous allons écrire une fonction `Creer(n)` qui prend en argument un entier naturel $n$ et qui revoie le polynôme de degré $n$ dont tous les cœfficients sont nuls.

   Par exemple : `Creer(4)` $\longrightarrow$ `[0,0,0,0]`.

2. **Degré d'un polynôme**

   En fonction des opérations que nous devons effectuer sur les polynômes, nous devons accéder facilement à son degré. Pour cela nous allons écrire une fonction `degre(P)` qui prend en argument une liste `P` correspondant à un polynôme et qui revoie le degré du polynôme considéré.

   Par exemple : `degre([3,2,1])` $\longrightarrow$ `2`.

3. **Multiplication par un scalaire**

   Dans l'espace vectoriel des polynômes, il est fréquent de multiplier un polynôme par un scalaire. Pour cela nous allons écrire une fonction `Scalaire(P,s)` qui prend en arguments une liste `P` et un flottant `s` et qui revoie la liste correspondant au produit du polynôme $P$ par le scalaire $s$.

   Par exemple, `Scalaire([3,2,1],4)` $\longrightarrow$ `[12,8,4]`.

4. **Addition de deux polynômes**

   Dans l'espace vectoriel des polynômes, la deuxième opération incontournable est l'addition de deux polynômes. Il est possible bien sûr d'additionner deux polynômes de degrés différents ; mais pour les listes il peut être préférable d'additionner terme à terme deux listes de même taille. Par exemple, si $P(X) = X^2 + 2X + 3$ et $Q(X) = X - 4$, alors on pourra créer et utiliser les listes `[3,2,1]` et `[-4,1,0]` pour effectuer l'addition.

   Ecrire alors une fonction `Additionner(P,Q)` qui prend en argument deux listes représentant deux polynôme $P$ et $Q$ et qui renvoie la liste correspondant à la somme de $P$ et de $Q$.

   Par exemple : `Additionner([3,2,1],[-4,1])` $\longrightarrow$ `[-1,3,1]`.

   Ce qui correspond bien à l'addition $\left(X^2 + 2X + 3\right) + \left(X - 4\right) = X^2 + 3X - 1$.

5. **Produit de deux polynômes**

Soient $P(X)=\sum_{i=0}^{p} a_i X^i$ et $Q(X)=\sum_{j=0}^{q} b_j X^j$ deux polynômes de degrés respectifs $p$ et $q$, alors le produit $PQ$ est un polynôme de degré $p+q$ et le cœfficients de $X^k$ est donné par $\sum_{i=0}^{k} a_i b_{k-i}$.

Ecrire alors une fonction `Multiplier(P,Q)` qui prend en argument deux listes représentant deux polynôme $P$ et $Q$ et qui renvoie la liste correspondant au produit de $P$ et de $Q$.

Par exemple : `Multilpier([3,2,1],[-4,1])` $\longrightarrow$ `[-12,-5,-2,1]`.

Ce qui correspond bien à la multiplication : $\big(X^2+2X+3\big)\big(X-4\big)=X^3-2X^2-5X-12$.

6. **Intégrer un polynôme**

   Soit $P(X)=\sum_{i=0}^{p} a_i X^i$ un polynôme de degré $p$. A l'instar du polynôme dérivé on peut définir le polynôme intégré comme $\sum_{i=0}^{p} \frac{a_i}{i+1} X^{i+1}$ ; il s'agit d'un polynôme de degré $p+1$ qui coïncide avec la primitive de $P(x)$ qui s'annule en 0.

   Ecrire alors une fonction `Integrer(P)` qui prend en argument une liste représentant un polynôme $P$ et qui renvoie la liste correspondant au polynôme intégré de $P$.

   Par exemple : `Integrer([3,2,1])` $\longrightarrow$ `[0,3.0,1.0,0.3333333333333333]`.

   Ce qui correspond bien à la l'intégration de $X^2+2X+3$ qui donne $\frac{1}{3}X^3+X^2+3X$.

7. **Evaluer un polynôme**

   Ecrire une fonction `Evaluer(P,x)` qui prend en arguments une liste `P` correspondant à un polynôme $P$ et un flottant `x` ; cette fonction renvoie la valeur de $P(x)$. On cherchera à écrire une fonction de complexité linéaire.

   Par exemple : `Evaluer([3,2,1],2)` $\longrightarrow$ `11`. Ce qui correspond bien au calcul $P(2)=2^2+2\times 2+3$.

## Polynômes interpolateurs de Lagrange

On considère une liste $A=\big[a_0,a_1,...,a_n\big]$ constituée de $n+1$ réels deux à deux distincts appelés **nœuds**. On définit alors le polynôme **de Lagrange** d'indice $i$ par :

$$L_i(X)=\prod_{j=0,j\neq i}^{n}\frac{X-a_j}{a_i-a_j}=\frac{X-a_0}{a_i-a_0}\times\cdots\times\frac{X-a_{i-1}}{a_i-a_{i-1}}\times\frac{X-a_{i+1}}{a_i-a_{i+1}}\times\cdots\times\frac{X-a_n}{a_i-a_n}$$

Il s'agit d'un polynôme de degré $n$ qui vérifie : $\begin{cases} L_i(a_j)=0 & \text{si } j\neq i \\ L_i(a_i)=1 \end{cases}$

8. **Polynômes de Lagrange**

   En utilisant les fonctions définies précédemment, écrire une fonction `Lagrange(A,i)` qui prend en arguments une liste `A` de nœuds et un entier `i` ; cette fonction renvoie le polynôme de Lagrange d'indice $i$.
   Par exemple :

   `Lagrange([1,2,3],0)` $\longrightarrow$ `[3.0,-2.5,0.5]`
   `Lagrange([1,2,3],1)` $\longrightarrow$ `[-3.0,4.0,-1.0]`
   `Lagrange([1,2,3],2)` $\longrightarrow$ `[1.0,-1.5,0.5]`

Considérons $n+1$ points $\big((a_0,b_0),(a_1,b_1),...,(a_n,b_n)\big)$ tels que les réels $(a_i)_{0\leqslant i\leqslant n}$ sont deux à deux distincts. Alors il existe un unique polynôme de degré au plus $n$ qui passe exactement par ces $n$ points ; il s'agit du polynôme **interpolateur de Lagrange** défini par :

$$L(X)=\sum_{i=0}^{n} b_i L_i(X)=\sum_{i=0}^{n} b_i \prod_{j=0,j\neq i}^{n}\frac{X-a_j}{a_i-a_j}$$

9. **Polynôme interpolateur de Lagrange**

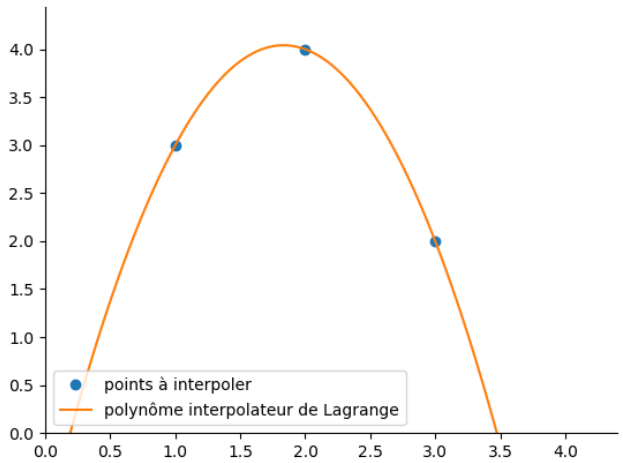
   On considère deux listes, la liste $A = [a_0, a_1, ..., a_n]$ des abscisses des points (nœuds) et la liste $B = [b_0, b_1, ..., b_n]$ des ordonnées des points.

   En utilisant les fonctions définies précédemment, écrire une fonction `Interpoler(A,B)` qui prend en arguments deux liste `A` et `B`, et qui renvoie le polynôme interpolateur de Lagrange. Par exemple :

   `Interpoler([1,2,3],[3,4,2]` $\longrightarrow$ `[-1.0,5.5,-1.5]`

   Pour visualiser le résultat précédent, on peut réaliser le graphique suivant :

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

A=[1,2,3]
B=[3,4,2]

plt.close()
plt.plot(A,B,'o',label='points à interpoler')
X=np.linspace(0,4,100)
Y=[Evaluer(Interpoler(A,B),x) for x in X]
plt.plot(X,Y,label='polynôme interpolateur de
                        Lagrange')
plt.legend()
plt.show()
```




## Application : interpolation polynomiale d'une fonction

Dans cette partie, nous considérons la fonction définie sur $\mathbb{R}$ par $f(x) = \dfrac{4}{1+x^2}$.

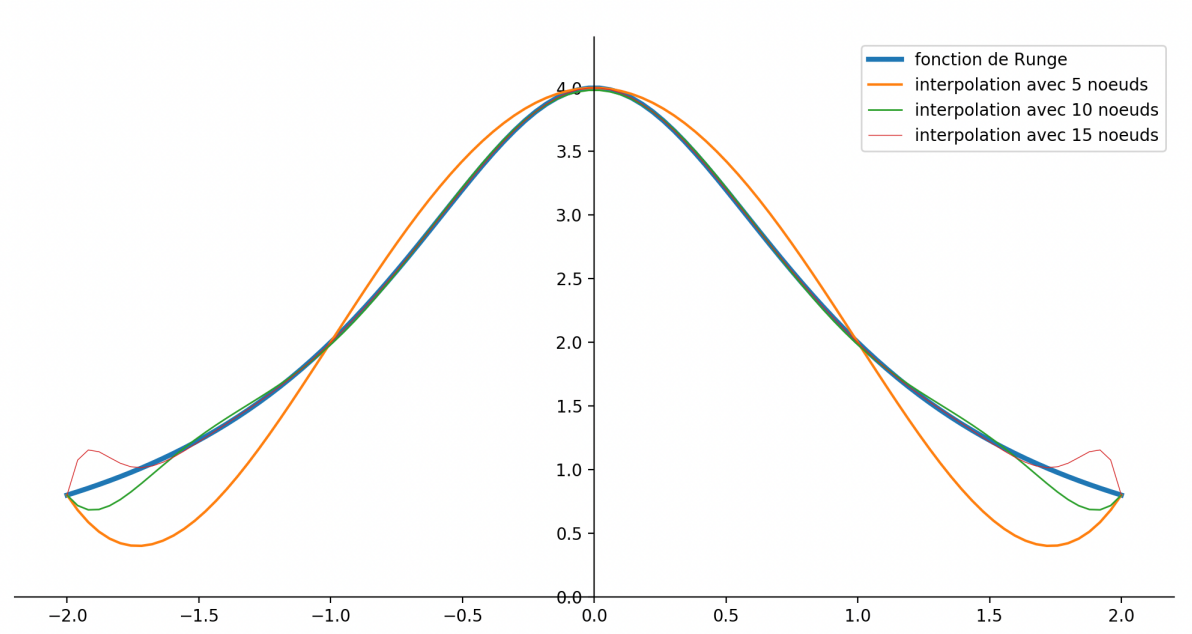
Dans un premier temps nous allons construire des polynômes interpolateurs de Lagrange qui permettent d'approcher cette fonction $f$ et constater le **phénomène de Runge**. Dans un deuxième temps nous utilisons cette interpolation polynomiale pour calculer une intégrale.

Pour obtenir une interpolation polynomiale de la fonction $f$ sur l'intervalle $[-2,2]$ par exemple, il suffit de définir un nombre $n$ de nœuds répartis dans cet intervalle et calculer les images de ces nœuds par $f$.

Par exemple si nous choisissons 5 nœuds équirépartis, nous obtenons $A = [-2,-1,0,1,2]$ et $B = [0.8,2,4,2,0.8]$.

Le polynôme interpolateur de Lagrange correspondant aux listes $A$ et $B$ donnera alors l'approximation ploynomiale de $f$ à 5 nœuds.

On peut alors penser que plus le nombre de nœuds augmentent et plus l'approximation est bonne ; mais en 1901, le mathématicien allemand Carl RUNGE découvrit un résultat contraire à l'intuition : il existe des configurations où l'écart maximal entre la fonction et son interpolation peut augmenter avec $n$, comme le montre le graphique suivant :




10. Donner une séries d'instructions Python qui permettent de réaliser un graphique mettant en évidence le phénomène de Runge.

11. **Calcul d'une intégrale**

    Dans cette question on s'intéresse à l'intégrale $\displaystyle\int_0^1 f(x)\,\mathrm{d}x = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$.

    L'objectif est de calculer cette intégrale au moyen de polynômes. En utilisant la technique d'interpolation polynomiale de $f$ et les fonctions définies dans ce TP, donner une série d'instructions qui permettent ce calcul.

    Donner le script de la fonction qui permet de calculer l'intégrale précédente par la méthode des trapèzes. Comparer les résultats.