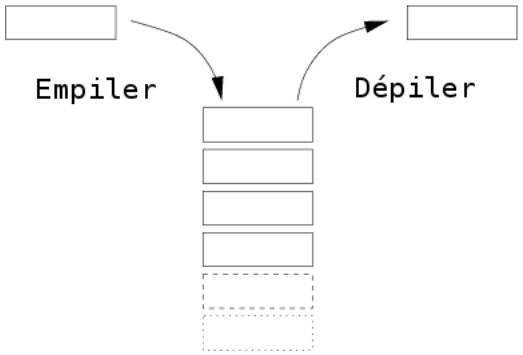
3.9 Les piles

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 ("empiler", 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* 3.10**

Dans la pile suivante

3	♣
2	♦
1	♥
0	♠

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

2	♦
1	♥
0	♠

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

3	✪
2	♦
1	♥
0	♠

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 3.18 *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 3.19 ***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 3.20 ***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* 3.11** 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
```

$F(4) = 4 * F(3)$ $F(3) = 3 * F(2)$ $F(2) = 2 * F(1)$ $F(1) = 1 * F(0)$ $F(0) = 1$ $F(1) = 1$ $F(2) = 2$ $F(3) = 6$ $F(4) = 24$	$F(4,1) = F(3,4)$ $F(3,4) = F(2,12)$ $F(2,12) = F(1,24)$ $F(1,24) = F(0,24)$ 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 3.21 *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 3.22 *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 3.23 *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 **complexité** 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 3.24 *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 **3.11**. *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 3.25 *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 3.26 *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 3.27 *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 3.28 *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 3.29

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 3.30 *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

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

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

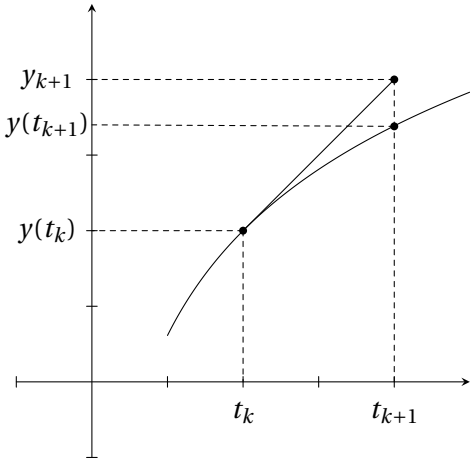
On note $y(t_k)$ les valeurs de la fonction exactes pour les instants choisis et y_k les valeurs approchées construites par itération.

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 3.31 *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

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

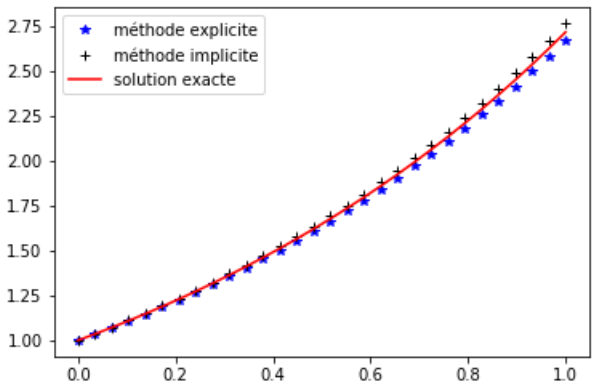
La condition initiale est $y(t_0) = y_0$; ensuite, $y_1 = y_0 + hf(t_1, y_1)$; puis, $y_2 = y_1 + hf(t_2, y_2)$ et ainsi de suite...
On constate que y_{k+1} est solution d'une équation et que sa valeur n'est donc pas obtenue explicitement. Un calcul supplémentaire est donc souvent nécessaire.

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 :

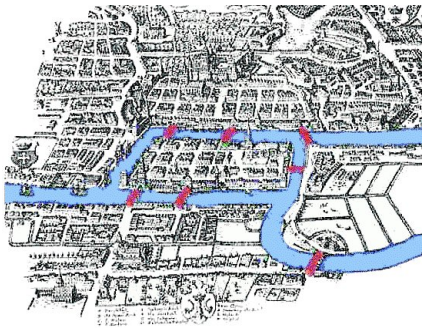
```
T=linspace(0,1,30)

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()
```

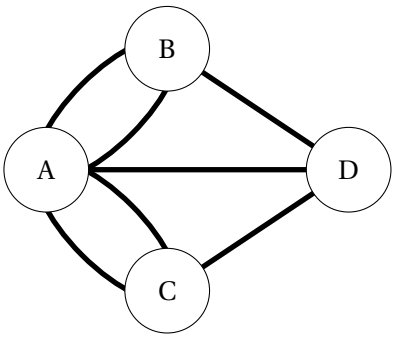


3.13 L'algorithme de Dijkstra

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.

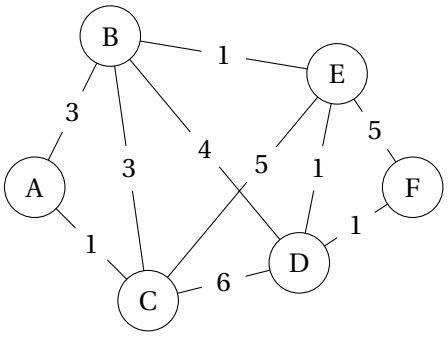
Dans le cas des ponts de Königsberg, le sommet A est de degré 5 et les sommets B, C et D sont de degré 3.

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 ∞ 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 ∞ 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 ∞.

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	∞	∞	∞	∞	∞	A
0	∞	3(A)	1(A)	∞	∞	∞	C
1	∞	3(A)	∞	7(C)	6(C)	∞	B
3	∞	∞)	∞	7(C)	4(B)	∞	E
4	∞	∞	∞	5(E)	∞	9(E)	D
5	∞	∞	∞	∞	∞	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 3.32 *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 3.33 *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
```