Bonjour à tous,

Nous allons nous retrouver en IPT l'an prochain.

Dans le but de retravailler les notions vues en première année, je vous transmets un polycopié qui est constitué de deux chapitres qui sont des chapitres de révisions du programme de première année.

Le premier chapitre est un chapitre de révisions sur
- la complexité
- les algorithmes récursifs
- les algorithmes de tri (Les 4 sont à maitrisés parfaitement !)
- il y a 12 exercices à chercher, une correction vous sera donnée sur cahier de prépa fin août.

Un premier devoir sur ces éléments est prévu le jour de la rentrée !

Le second chapitre est une reprise de votre cours sur les graphes. Vous devez maîtriser :
- les matrices d'adjacences, les listes d'adjacences et les dictionnaires pour représenter un graphe et savoir passer de l'un à l'autre.
- les algorithmes de parcours en largeur et en profondeur des graphes
- l'algorithme de Dijkstra sera repris en cours mais je vous conseille de le retravailler !
- Les exercices de ce chapitre seront à préparer au cours du mois de septembre.

Le dernier travail est de refaire vos DS de l'année et si possible vos TPs de première année sur les images, les tris, la récursivité, les graphes. S'entrainer devant un ordinateur c'est fondamental et en deuxième année nous n'avons que très peu de temps !

Je vous souhaite à tous de bonnes vacances. Pour toute question concernant un exercice, je suis disponible par mail : fb.pcstar.cezanne@gmail.com. N'hésitez pas si vous avez besoin !
Au plaisir de vous voir le 4 septembre !

MP

ANGLAIS : RENTREE 2023

Bonjour à tous, afin que la rentrée se fasse le plus sereinement possible, voici quelques indications. Si vous avez des questions, notamment pour les choses à absolument mettre en place pendant l'été (point numéro ❶), vous pouvez me contacter par mail : eaurambault@gmail.com

❶ LES OUTILS INDISPENSABLES POUR LA RENTREE

Afin d'entrainer efficacement votre mémoire tout au long de l'année, nous[1] travaillerons avec le logiciel «**Anki** », qui signifie « mémorisation » en japonais, et qui permet ***d'apprendre et de réviser*** quotidiennement des ***cartes-mémoire*** grâce à la ***répétition espacée***.

Pour cela, **vous devez télécharger Anki** ici : https://apps.ankiweb.net/

Vous pouvez l'installer sur votre **ordinateur** et/ou votre **téléphone**, ce qui est pratique pour réviser dès que l'occasion se présente. Seul bémol, l'application sur iPhone est payante (une trentaine d'euros). A vous de voir si cet investissement est nécessaire. Vous pourrez toujours travailler en ligne et sur ordinateur **gratuitement**.

Vous devrez aussi vous créer un **compte en ligne sur AnkiWeb** afin de synchroniser vos données sur tous vos appareils : https://ankiweb.net/account/register

Je vous fournirai certaines des ***cartes-mémoire*** à importer dans le logiciel tout au long de l'année en fonction de nos avancées dans le programme. Vous pourrez aussi créer les vôtres en fonction de vos besoins. Vous pouvez aussi télécharger des paquets de cartes « tout prêts » et disponibles en ligne.

Pour prendre en main le logiciel AVANT la rentrée :

- **Tutos de professeurs** qui vous expliquent très clairement et pas à pas le principe du logiciel pour créer vos propres cartes, entrainer votre mémoire et synchroniser votre compte.

https://www.youtube.com/watch?v=w5_1xeyWQeA

https://www.youtube.com/watch?v=BNDgxzIgARE

- **Tuto d'un élève de prépa, très enthousiaste** !

https://riche-de-temps.fr/anki-francais/

- **Le manuel utilisateur**, très complet : https://apps.ankiweb.net/docs/manual.fr.html

Pour ceux et celles d'entre vous qui utilisent "**Quizlet**" (système similaire à Anki mais moins efficace sur le processus de mémorisation à long terme), vous pouvez, si vous le souhaitez, importer vos paquets Quizlet dans Anki pour tout avoir au même endroit. Le tuto est là : https://greencoin.life/how-to/convert/quizlet-to-anki/

Vous pouvez commencer par importer un paquet de cartes tout prêt en ligne comme ***les verbes irréguliers*** par exemple et commencer à les revoir pour la rentrée.

Si vous avez des questions : eaurambault@gmail.com

[1] « **nous** » = vous, ainsi que Mme Lavainne en physique et moi-même en anglais.

❷ LES HABITUDES A NE PAS PERDRE PENDANT L'ETE

Vous le savez, le support principal pour les différentes épreuves des concours, à l'écrit comme à l'oral, est la presse. Pensez donc à **lire la presse** cet été … Voici une liste non exhaustive des principaux journaux ou magazines et de leurs sites internet :

- *The Guardian* www.theguardian.com/international (Site entièrement gratuit et ressource principale pour les sujets de concours)
- *The Financial Times* https://www.ft.com/ (Accès gratuit à un certain nombre d'articles; privilégiez les articles de la section « Opinion » : https://www.ft.com/opinion)
- BBC News https://www.bbc.com/news
- *The Economist* www.economist.com (Accès gratuit à un certain nombre d'articles).

- *The New York Times* www.nytimes.com (De très bons articles, calibrés pour le concours, dans la section "Opinion" notamment – accès gratuit à un certain nombre d'articles par mois)
- https://slate.com/
- *The Wall Street Journal* www.wsj.com
- CNN news website http://edition.cnn.com

Je suis certaine que vous saurez aussi allier plaisir et travail en regardant vos **séries** préférées en VO ou en écoutant de la **musique**, voilà deux sites qui vous permettront de le faire : https://fr.lyricstraining.com/en/
https://www.esolcourses.com/topics/learn-english-with-songs.html

D'autres sites qui proposent des ressources intéressantes :

1. Pour trouver des **informations sur des thèmes variés** en anglais :

News in Levels : Différents articles et vidéos sur des sujets d'actualité variés, et avec différents niveaux de difficulté – http://www.newsinlevels.com/

2. **Pour progresser au niveau de l'accent** :

- Pour comprendre les sons, une infographie audio et interactive : https://englishcpge.jimdofree.com/pronunciation/
- **Acapela** https://www.acapela-group.com/ – tapez et écoutez en choisissant un accent américain, britannique, etc.
- https://www.bbc.co.uk/learningenglish

Enfin, pour être efficace dès la rentrée et partir sur de bonnes bases, faites un **bilan** pour optimiser votre première année :

1. **Rangez** vos cours en fonction des thématiques/thèmes abordés.
2. **Centralisez** tous vos DS/DM et **créez des fiches récapitulatives** des erreurs à ne plus commettre (vous pouvez d'ailleurs commencer à vous créer des cartes dans Anki)
3. **Relisez** les fiches des colleurs et **faites une fiche spéciale pour les colles** où vous listez les erreurs à ne plus commettre au niveau de la grammaire, du lexique, de la prononciation, et de la méthode. FIXEZ VOUS DES OBJECTIFS.

See you soon !

INFORMATIQUE COMMUNE
PARTIE I

MP - PC*

Lycée Paul Cézanne

Florence Allègre

Année 2023-2024

PARTIE I : Révisions

Chapitre 1 : Algorithme de tri - Récursivité - Complexité

1 Les différents types de données

1.1 Structures indicées immuables (chaines, tuples)

Une structure de donnée est indicée lorsqu'on peut accéder à ses éléments individuels par l'intermédiaire de leur indice ; une structure de donnée est immuable lorsque ces éléments individuels ne peuvent être modifiés. Deux structures de ce type sont à connaître :

— les chaînes de caractère (le type str) qui sont des suites de caractères alphanumériques délimités par des guillemets simples ou doubles. Par exemple 'L informatique est une belle science' est une chaîne de caractère.

— les tuples (le type tuple) ou n-uplets qui sont des suites finies de valeurs séparées par des virgules (si besoin enclos par des parenthèses). Par exemple, (2, 3, 5, 7, 11) est un tuple.

Ces structures partagent les opérations suivantes :

— la fonction len permet de calculer leur longueur ;

Test
```
>>>len('L informatique est une belle science')
36

>>>len((2,5,7))
3
```

— on accède aux éléments individuels avec la syntaxe $[k]$ où k est un indice positif valide ;

Test
```
>>>'L informatique est une belle science'[3]
'n'

>>>'L informatique est une belle science'[35]
'e'

>>>'L informatique est une belle science'[36]
Traceback (most recent call last):
  File "<pyshell#97>", line 1, in <module>
    'L informatique est une belle science'[36]
IndexError: string index out of range
```

— on peut en calculer une tranche avec la syntaxe [i : j] qui extrait tous les éléments dont les indices sont compris entre i inclus et j exclus ;

Test
```
>>>'L informatique est une belle science'[0:6]
'L info'
```

— ces structures peuvent être concaténées avec l'opérateur + et répétées avec l'opérateur *.

Test

```
>>>'L informatique est une belle science'+' mais chronophage'
'L informatique est une belle science mais chronophage'
>>>(2,)*10
(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
```

(Notez au passage comment se définit un tuple de longueur 1).

1.2 Les Listes

L'implémentation des listes doit être familière pour vous. On rappelle qu'une liste est un séquence **modifiable**.

```python
#Création d'une liste entre crochets
L=[2,3,[4,5],True,'bonjour']

#Longeur d'une liste
print(len(L))

#Récuperer un élément de L en position i
i=2
print(L[i])

#parcourir une liste
for i in range(len(L)):
    print(L[i])
```

Test

```
>>>L
[2, 3, [4, 5], True, 'bonjour']

>>>len(L);L[2]
5
[4,5]

>>>for i in range(len(L)):
    print(L[i])
2
3
[4, 5]
True
bonjour
```

```python
#Extraction de l'élément i à l'élément j
i=2
j=5
L[i:j]

#Tester si un élément appartient à L
2 in L #(renvoie un booléen)

#Tester si un élément n'appartient à L
True not in L #(renvoie un booléen)

#Création d'une liste vide
Lvide=[]

#Ajouter un élément à la fin de la liste
L.append('Hello')

#Ajouter un élément en position i dans L
L.insert(2,'to')

#Enlever un élément à la fin de L
L.pop()

#Enlever un élément en position i à L
L.pop(3)

#Concacténer deux listes
Lprime=L+[3,5]
```

Test

```
>>>L[2:5]
[[4, 5], True, 'bonjour']

>>>2 in L
True
>>>2 not in L
False

>>>L.append('Hello')
>>>L
[2, 3, [4, 5], True, 'bonjour', 'Hello']

>>>L.insert(2,'to')
>>>L
[2, 3, 'to', [4, 5], True, 'bonjour', 'Hello']

>>>L.pop()
'Hello'
>>>L
[2, 3, 'to', [4, 5], True, 'bonjour']

>>>L.pop(3)
[4, 5]
>>>L
[2, 3, 'to', True, 'bonjour']

>>>Lprime
[2, 3, 'to', True, 'bonjour', 3, 5]
```

Tout comme pour les tuples et les chaînes de caractères, la fonction **len** renvoie la longueur d'une liste, on accède aux éléments par indice positif valide, on peut en calculer une tranche et réaliser une concaténation de deux listes par l'opérateur +.

Cependant, il faut prendre garde au fait que les deux syntaxes $L.append(x)$ et $L = L + [x]$, même si elles conduisent au même résultat apparent, **ne sont pas équivalentes**. En effet, la seconde version recrée une nouvelle liste augmentée d'un élément là où la première se contente de rajouter un élément à une liste existante. On s'en doute, la deuxième version peut se révéler beaucoup plus coûteuse pour une liste de grande taille. Pour cette raison, on utilisera avec la plus grande parcimonie l'opérateur de concaténation + pour les listes.

La principale différence avec les structures de données précédentes est que les listes sont des structures de données **mutables**, dans le sens où il est possible de modifier un élément individuel d'une liste sans avoir à recréer la liste dans son entièreté.

La copie d'une liste L se réalise par la méthode L.copy() ou par le calcul d'une tranche comprenant l'entièreté de la liste L[:].

Enfin, la méthode L.pop() supprime de la liste L son dernier élément et le renvoie en valeur de retour. Cette méthode modifie la liste en temps constant, contrairement à une syntaxe de type L = L[0 :len(L) ?1] qui recalculerait l'entièreté de la liste moins son dernier élément (et serait donc bien moins efficiente).

Remarque

Attention pour faire une liste de liste de zéros, que l'on veut modifier après il ne faut pas faire n'importe comment, regarder et essayer de comprendre la différence entre les deux approches :

```python
#Liste de listes de taille (3,4)
L1=3*[4*[0]]

#Liste de listes de taille (3,4)
L2=[[0 for k in range(4)] for k in range(3)]
```

Test

```
>>>L1
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>>L2
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>>L1[0][2]=2
>>>L1
[[0, 0, 2, 0], [0, 0, 2, 0], [0, 0, 2, 0]]
>>>L2[0][2]=3
>>>L2
[[0, 0, 3, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Retenir

On retiendra qu'il existe trois modes de création qui sont à connaitre :

- par *compréhension* en suivant la syntaxe [expr for x in s]
- par *duplication* en suivant la syntxe [expr]*n ;
- par .append(successifs)

Nous noterons que la création par duplication présente deux inconvénients : elle ne permet que de créer des listes de valeurs égales et surtout ; *elle se révèle incorrecte lorsque cette valeur est de type mutable comme le présente l'exemple précédent.* Ce mode de création est pour moi à éviter.

On retiendra les commandes les plus utiles pour une liste L déjà créée :

- L.append(a)# ajout en fin de liste de l'élément a
- L.pop()# retrait du dernier élément de L
- L[k]=b# modification de l'élément L[k]

1.3 Les Piles et les Files

Les listes Python sont une séquence d'éléments ordonnée, mutable et modifiable comme on l'a vu précédemment. Python permet d'ajouter des listes en temps constant (0(1), la taille de la liste n'a aucun effet sur le temps qu'il faut pour l'ajout), mais l'insertion au début d'une liste pet être bien plus lente puisqu'en $O(n)$, donc ce temps augmente à mesure que la liste s'agrandit. Ceci a donc conduit à introduire des séquences ordonnées et modifiables où l'ajout d'un élément au début ou à la fin est en temps constant, soit $O(1)$. Ces notions sont celles de Pile et File.

Les notions de PILE et de FILE sont des structures de données abstraites importantes en informatique.

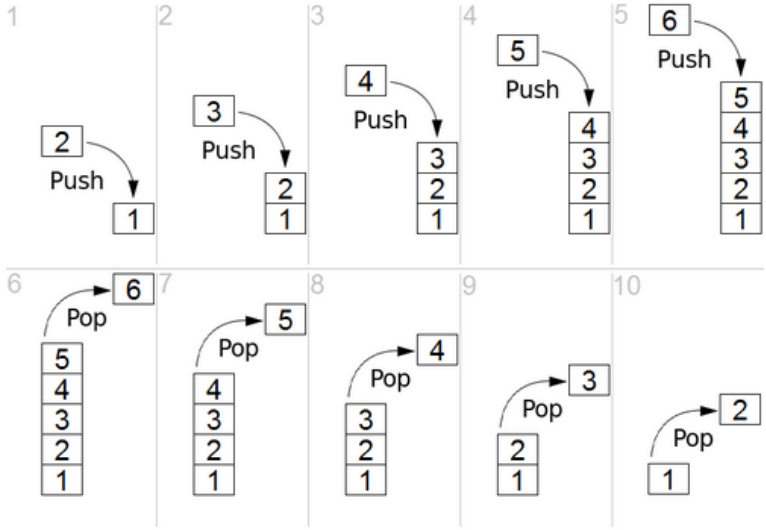
• **PILE (stack)**

La structure de PILE est celle d'une pile d'assiettes :

— Pour ranger les assiettes, on les empile les unes sur les autres.

— Lorsqu'on veut utiliser une assiette, c'est l'assiette qui a été empilée en dernier qui est utilisée.

On appelle cela la structure LIFO : Last In, First Out



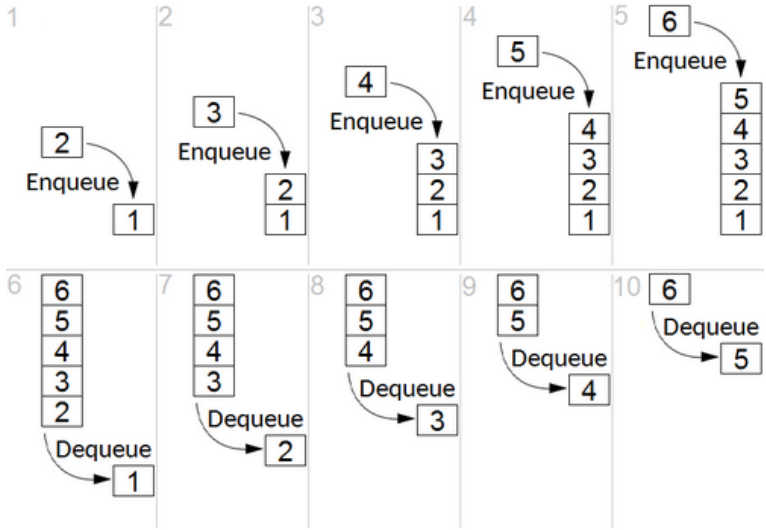
• **FILE (queue)**

La structure de FILE est celle d'une file d'attente à un guichet :

— les nouvelles personnes qui arrivent se rangent à la fin de la file d'attente

— La personne servie est celle qui est arrivée en premier dans la file

On appelle cela la structure FIFO : First In, First Out



En python, nous utiliserons la bibliothèque collections et sa sous bibliothèque deque() qui permet une implémentation des files et des piles à capacité illimitée ou limitée en temps constant. Voici les principales lignes de commandes :

```python
from collections import deque

# initialise un deque vide
qvide=deque()


#Créer une pile [a,b,1] à capacité illimitée
q1=deque(['a','b',1])

#Créer une file [b,4,d] à capacité limitée à 10
q2=deque(['b',4,'d'])


#Créer une pile [a,b,1] à capacité limité à 10
q1l=deque(['a','b',1],maxlen=10)

#Créer une file [b,4,d] à capacité limitée à 10
q2l=deque(['b',4,'d'],maxlen=10)
```

Test

```
>>>q1
deque(['a', 'b', 1])

>>>q2
deque(['b', 4, 'd'])

>>>q1l
deque(['a', 'b', 1], maxlen=10)

>>>q2l
deque(['b', 4, 'd'], maxlen=10)
```

```python


#rajouter un élément à la fin
(pour les piles)
q1.append('c')
q1l.append(7)
qvide.append(8)
qvide.append(10)


#rajouter un élément au début
(pour les files)
q2.appendleft(2)
q2l.appendleft('e')


#renvoie et retire le dernier élément
(pour les piles/files)
q1.pop()
q2.pop()
q2.pop()


#renvoie et retire le premier élément
si on le souhaite
qvide.popleft()

#renvoie la taille de la pile
len(q1)
len(q2)
len(q1l)
len(q2l)
```

Test

```
>>>q1
deque(['a', 'b', 1])

>>>q2
deque([2, 'b'])

>>>q1l
deque(['a', 'b', 1, 7], maxlen=10)

>>>q2l
deque(['e', 'b', 4, 'd'], maxlen=10)

>>>len(q1);len(q2);len(q1l);len(q2l)
3
2
4
4
```

1.4 Les dictionnaires

Un dictionnaire en python va aussi permettre, comme pour les listes, de rassembler des éléments **mais ceux-ci seront identifiés par une clé**, au lieu d'un indice. On peut faire l'analogie avec un dictionnaire de français où on accède à une définition avec un mot.

Contrairement aux listes qui sont délimitées par des crochets, on utilise des accolades pour les dictionnaires.

```python
#Créer un dictionnaire
G={}

#Ajouter un élément
G={}
G['voiture']=["moteur", 4]
G['velo']=["sans moteur", 2]
G[' tricyle ']=["sans moteur",3]

#parcourir un dictionnaire
for cle in G:
    for j in range(len(G[cle])):
        print(G[cle][j])
```

Test

```
>>> for cle in G:
        print(cle)

voiture
velo
 tricyle
>>> for cle in G:
        print(cle, ':')
        for j in range(len(G[cle])):
            print(G[cle][j])

voiture :
moteur
4
velo :
sans moteur
2
 tricyle :
sans moteur
3
```

Retenir

La création d'un dictionnaire se réalise donc en suivant la syntaxe $\{c_1 : v_1, \ldots, c_n : v_n\}$ où $c_1, \ldots c_n$ sont des clefs (nécessairement deux à deux distinctes) et $v_1, \ldots, v_n$ les valeurs qui leur sont associés. On notera que $\{\}$ crée un dictionnaire vide.

Si D est un dictionnaire et c une clé :

- l'expression c in D renvoie un booléen indiquant si la clé est est présente ou non dans le dictionnaire ;
- $D[c]$ renvoie la valeur associée à la clé si celle-ci est présente dans le dictionnaire (et provoque une erreur sinon) ;
- $D[c] = v$ crée une nouvelle association si la clé n'est pas présente dans le dictionnaire, et modifie l'association précédente sinon.
- $delD[c]$ supprime la clé c ainsi que sa valeur dans le dictionnaire D.

2 Complexités d'un algorithme

2.1 Définitions

Pour traiter un même problème, il existe souvent plusieurs algorithmes.

Quand on doit choisir parmi plusieurs algorithmes : l'un des critères est celui du **temps d'exécution.** On l'appelle aussi parfois le coût temporelle de l'algorithme (ce temps conditionne les ressources utilisées : machine sur laquelle on exécutera l'algorithme, consommation électrique...). Un autre critère est celui du **coût de la mémoire.**

Le temps d'exécution est un point important à étudier lorsque l'on implémente un algorithme pour savoir si il est utilisable en temps réel : sur une base de 10^9 opérations élémentaires effectuées en 1s, si on doit faire n^2 opérations pour un algorithme avec $n = 10^6$ cela demandera 2heures 48min !

Définition

La complexité temporelle : c'est le nombre d'opérations élémentaires ou fondamentales effectuées par l'algorithme lors d'une exécution.

La complexité spatiale : c'est le coût en mémoire (ou en espace) donné par le nombre d'emplacements mémoires occupés lors de l'exécution d'un programme.

2.2 Détermination de la complexité spatiale d'un algorithme

Nous rappelons deux informations importantes :

Retenir

1. Un nombre entier N est représentable par un nombre binaire de n bits avec $2^{n-1} \leqslant N < 2^n$.
   Par exemple, pour stocker des entiers naturels entre 0 et 1023, il faut 10 bits car $2^{10} = 1024$.
2. le stockage d'un flottant en simple précision nécessite 32 bits, et en double précision nécessite 64 bits.

Exemple

Pour stocker les valeurs d'une matrices de taille 10000×10000 , si ceux sont des flottants et que l'on souhaite les stocker en double précision, combien d'espace mémoire est necessaire ?

Exemple

Pour stocker les valeurs d'une image de taille 1500×2000 codés en RGB, avec chaque couleur codée avec un entier compris entre 0 et 255, combien d'espace mémoire est nécessaire ?

2.3 Détermination de la complexité temporelle d'un algorithme

Nous nous baserons sur le modèle de complexité suivant :

Retenir

1. Une affectation, une comparaison ou l'évaluation d'une opération arithmétique simple sera considérer comme unité de base dans laquelle on mesure le coût de l'algorithme.
2. Le coût des instructions p et q en séquence est la somme des coûts de l'instruction p et de l'instruction q.
3. Le coût d'un test $if\ p\ else\ q$ est inférieur ou égal au maximum des instructions p et q, plus un unité qui correspond au temps d'évaluation de l'expression b.
4. Le coût d'une boucle $for\ i\ in\ range(m)\ p$ est m fois le coût de l'instruction p si ce coût ne dépend pas de la valeur de i. Quand le coût du corps de la boucle dépend de la valeur du compteur i, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de i.
5. Le cas des boucles while est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu a priori. On parle alors :
   (a) **de complexité dans le meilleur des cas** : on donne une estimation la plus optimiste du temps d'exécution en fonction des données.
   (b) **de complexité dans le pire des cas** : on donne un majorant du temps d'exécution sur toutes les données possibles d'une même taille.

   Le complexité au pire de cas est la plus significative, mais dans certains cas, il peut être utile de connaître aussi la complexité dans le meilleur des cas, pour avoir une borne inférieure du temps d'exécution d'un algorithme.
   En particulier, si la complexité dans le meilleur des cas et dans le pire des cas sont du même ordre, cela signifie que le temps d'exécution de l'algorithme est relativement indépendant des données et ne dépend que de la taille du problème.

2.4 Notation de Landau $\mathcal{O}$

Retenir

Pour déterminer la complexité d'un algorithme, on peut déterminer de façon très précise le nombre d'opérations élémentaires, mais il est souvent inutile d'aller jusqu'à ce niveau de détail : on peut se contenter de dire que le nombre d'opérations élémentaires effectuées est proportionnel à n, n^2, ... On parle alors de complexité asymptotique.

Il y a plusieurs raisons à cela :

— Les différentes opérations élémentaires ne demandent pas toutes exactement le même temps de calcul

— le même programme peut être exécuté sur deux machines différentes avec l'une qui va deux fois plus vite que l'autre, ce qui nous intéresse n'est pas le temps d'exécution mais l'ordre de grandeur de ce temps en fonction des données.

— Ainsi l'élément à considérer est le terme dominant : dès que la taille n des données devient un peu importante, si un algorithme nécessite n^2+3n opérations élémentaires, le terme $3n$ est très vite négligeable devant le terme n^2. On dira que la complexité est en $\mathcal{O}(n^2)$, notation de Landau.

Un ordre de grandeur du temps d'exécution pour les différentes complexités en temps :

	Nom	Tps pour $n = 10^5$	Remarques
$\mathcal{O}(1)$	Temps cst	1ns	Indépdt des données, très rare
$\mathcal{O}(\ln(n))$	logarithmique	10ns	Exécution quasi instannée
$\mathcal{O}(n)$	linéaire	1ms	Tps d'éxecution supérieur à la minute pour $n \geqslant 10^{10}$
$\mathcal{O}(n^2)$	quadratique	15 min	Acceptable pour $n = 10^6$ au delà non
$\mathcal{O}(n^k)$	polynomiale	30 ans si $k = 3$	Taille donnée très vite critique
$\mathcal{O}(2^n)$	exponentielle	$> 10^{300000}$ ans	Limite entre pb traitable ou pas

3 Exemples

3.1 Somme des éléments d'un tableau T de taille n

Exemple

```python
def Somme(T):
    S=0
    for i in range(len(T)):
        S=S+T[i]
    return S
```

Quelle est la complexité temporelle de cette algorithme ?

3.2 Algorithme de recherche séquentielle

Un premier exemple naïf :

Exemple

```python
def recherche(x,L):
    for i in range(len(L)): #on parcourt la liste L
        if L[i]==x: #si on rencontre x, on renvoie True
            return True #et on sort de la fonction avec return
    return False #si x n'a pas été rencontré, alors on retourne False
```

Quelle est la complexité temporelle de cette algorithme ?

L'algorithme précédent de recherche dans un tableau peut être amélioré lorsque par exemple le tableau **est trié**. On peut alors effectuer une recherche par principe de dichotomie

Exemple

```python
def dichotomie_liste(x,L):
    debut=0 #ces deus indices vont évoluer au cours de l'algorithme
    fin=len(L)#pour /2 la longueur de la liste à parcourir
    while fin-debut>1:
        milieu=(debut+fin)//2
        if L[milieu]>x:
            #dans ce cas on a x dans la partie de gauche de la liste
            fin=milieu
        else:
            debut=milieu
            #dans ce cas on a x dans la partie de droite de la liste
    if L[debut]==x:
        return True, debut
    else:
        return False
```

Quelle est la complexité temporelle de cette algorithme ?

.

3.3 Pour un algorithme récursif

Factorielle n

Exemple

Ecrire un algorithme récursif et un algorithme itératif pour déterminer $n!$

```python
def fact_it (n):
    .
    .
    .
    .
    .
    .
    .
```

```python
def fact_rec(n):
    .
    .
    .
    .
    .
    .
    .
```

Déterminer la complexité de ces deux algortihmes.

Suite de Fibonacci

Nous rappelons la définition de la suite de Fibonacci :

$$u_0 = 1, u_1 = 1, \quad \forall n \geqslant 0,\ u_{n+2} = u_n + u_{n+1}$$

Exemple

Ecrire un algorithme itératif et une algorithme récursif pour déterminer u_n.

```python
def fibo_it (n):
    .
    .
    .
    .
    .
    .
    .
```

```python
def fibo_rec(n):
    .
    .
    .
    .
    .
    .
    .
```

Comparer la complexité de ces deux algortihmes.

4 Les algorithmes de tri et leur complexité temporelle

4.1 Le tri selection

Algorithme

Son principe du tri selection est assez simple. Etant donné un tableau composé de N éléments :

— Chercher dans le tableau l'élément maximal.

— Le déplacer en fin de tableau

— Recommencer avec le sous-tableau constitué des N-1 premiers éléments.

— etc....

Son nom provient du fait qu'à chaque étape on SELECTIONNE le plus grand élément du sous-tableau, avant de la déplacer en fin de ce sous-tableau.

On écrit donc une premier fonction **IndiceMax** qui prend comme argument une liste T et un indice i et qui cherche parmi les i premiers éléments de T, l'indice de l'élément maximal. Et ensuite, on écrit le tri selection.

```python
def IndiceMax(T,i):
    imax=i
    for k in range(i):
        if T[imax]<T[k]:
            imax=k
    return imax

def Tri_Selection(T):
    for i in range(len(T)-1,0,-1):
#On parcourt T dans le sens inverse
        iMax=IndiceMax(T,i)
        T[i],T[iMax]=T[iMax],T[i]#on effectue les échanges
    return T
```

La complexité du tri selection

4.2 Le tri insertion

Algorithme

1. On commence par une boucle for pour parcourir le tableau
2. Ensuite pour insérer l'élément i, noté x, à la bonne place, on doit regarder tous les éléments précédents tant qu'ils sont supérieurs à l'élément x et dès que ce n'est plus le cas, on place alors x à cette position trouvée.
3. Enfin, on retourne le tableau trié.
4. Cet algorithme de tri se fait en place est donc à la fin du tri la liste L est triée, ce n'est plus le tableau de départ si on l'appelle dans le shell, mais le tableau trié !

```python
def tri_insertion (T):
    for i in range(1,len(T)):
        x=T[i]
        j=i
        while j>0 and T[j-1]>x:
            T[j]=T[j-1]
            j=j-1
        T[j]=x
    return(T)
```

La complexité du tri insertion

4.3 Le tri rapide

Algorithme

Le tri rapide a été inventé par le scientifique C.A.R. Hoare en 1961.

Le tri rapide est une méthode de tri **récursive** s'appuyant sur le principe **de diviser pour régner**. La méthode est constituée de trois étapes :

1. la découpe du problème
2. le traitement des sous-problèmes
3. la reconstruction de la solution

Le principe est de choisir un pivot (en pratique nous prendrons le premier élément de la liste), autour duquel nous allons couper le tableau. Cela consiste donc à mettre tous les éléments plus petits que le pivot à sa gauche, et tous les éléments plus grands que le pivot à sa droite. Au fur et à mesure de la découpe ; le tableau conserve la forme suivante : $[\leqslant p | \, p \, | > p]$. On trie ensuite de manière récursive les sous tableaux de gauche et de droite : en recollant les morceaux, le tableau est trié.

Le tri rapide s'effectue en place, comme le tri insertion, mais il est plus difficile à implémenter. Il peut aussi s'effectuer en utilisant des tableaux intermédiaires (donc pas en place), mais alors nécessite plus d'espace mémoire et il est plus facile à implémenter.

Les étapes sont les suivantes :

— Choisir arbitrairement le premier élément dans le tableau T comme pivot

— Décomposer les autres éléments du tableau en deux sous-tableaux :

  — Tg constitué des éléments inférieurs à e.

  — Td constitué des éléments supérieurs à e.

- Après, on fait des appels récursifs sur les deux sous-tableaux Td et Tg, puis le tableau T s'obtient en concaténant les sous-tableaux Tg trié, suivi de e, suivi de Td trié. Voilà une première tentative

```python
def tri_rapide(T):
    if len(T)<=1:
        return T
    e=T[0]
    Tg=[]
    Td=[]
    for k in range(len(T)):
        if T[k]<=e:
            Tg.append(T[k])
        else:
            Td.append(T[k])
    return tri_rapide2(Tg)+tri_rapide2(Td)
```

Mais losrqu'on lance l'algorithme, un message nous dit qu'il y a trop d'appel récursifs. Pourquoi ?

Corriger l'algorithme pour qu'il fonctionne.

```python
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
```

La complexité du tri rapide

4.4 Le tri fusion

Algorithme

Comme pour le tri rapide, le tri fusion applique le principe de diviser pour régner. Il partage les éléments à trier en deux parties de même taille, sans chercher à comparer leurs éléments. Une fois les deux parties triées récursivement, il les fusionne, d'où le nom de tri fusion. Ainsi, on évite le pire des cas du tri rapide où les deux parties sont disproportionnées.

On va chercher à réaliser le tri fusion d'un tableau, en délimitant la portion à trier par deux indices $g = 0$ (inclus) et d =len(T) (exclu). Pour le partage il suffit de calculer l'indice médian $m = (g + d)//2$. On trie alors récursivement les deux parties délimités par g et m d'une part, puis m et d d'autre part. Il reste alors à fusionner. La fusion s'avère très difficile à réaliser en place, nous choisissons donc d'utiliser un second tableau.

La fonction fusion récursive

— La fonction fusion prend en arguments deux tableaux $T1$ et $T2$.

— On suppose que les tableaux $T1$ et $T2$ sont triés

— L'objectif est de les fusionner dans un nouveau tableau.

— Tout ceci de manière récursive sur la longueur des listes $T1$ et $T2$

```python
def fusion_rec(T1,T2):
    if len(T1)==0:
        return T2
    elif len(T2)==0:
        return T1
    if T1[0]<=T2[0]:
        return [T1[0]]+fusion_rec(T1[1:],T2)
    else:
        return [T2[0]]+fusion_rec(T1,T2[1:])
```

La deuxième partie récursive du tri fusion va être matérialisée par une fonction récursive tri_fusion_rec : on calcule le médian (entier) des indices $m = len(T)//2$ et on trie récursivement $T[0 : m]$ et $T[m, len(T)]$:

Ce qui donne :

```python
def tri_fusion_rec(T):
    if len(T)<=1:
        return T
    m=len(T)//2
    T1=T[0:m]
    T2=T[m:len(T)]
    return fusion_rec(tri_fusion_rec(T1), tri_fusion_rec(T2))
```

La complexité du tri fusion

5 Terminaison et correction d'un programme

La programmation est une activité complexe. Un programme manipule un nombre de données pouvant être important, dont les valeurs fluctuent durant l'exécution du programme. Il est important de pouvoir justifier rigoureusement qu'un programme fait bien ce qu'on lui demande et retourne bien le résultat cherché. C'est ce qu'on appelle **la correction de programme**. Domaine de recherche actif, difficile, basé sur la logique mathématique. Nous allons présenté un type de correction : la terminaison de boucle.

5.1 Terminaison

Boucle for

Celle ci se termine forcément après les n tests sur la boucle for.

Boucle while

Lors de l'utilisation d'une boucle while, il est important de s'assurer que la boucle ne tournera pas indéfiniment.

Un exemple :

```python
k=10
c=1
while k!=0:
    k=k+1
    c=c*2
return c
```

Ce programme ne s'arrêtera jamais, on devra forcer l'arrêt en appuyant simultanément sur les touches *Ctrl* et *C* dans l'interpréteur.

Le programme suivant sera certainement mieux adapté

```python
k=10
c=1
while k!=0:
    k=k-1
    c=c*2
return c
```

La variable k joue le rôle de compteur. Mathématiquement, il est garanti que l'on sorte de la boucle car :

— la valeur de k est un entier strictement positif

— elle décroît strictement après chaque itération

Comme il n'existe pas de suite infinie strictement décroissante d'entiers naturels, il ne peut y avoir qu'un nombre fini d'itérations.

Retenir

Il n'est pas nécessaire d'avoir une variable du type compteur, il suffit qu'une quantité vérifie bien ces deux propriétés : être un entier positif tout au long de l'algorithme et décroitre strictement après chaque itération. On appelle cette quantité un variant de boucle.

5.2 La correction de l'algorithme de recherche du maximum dans une liste

5.3 La correction du tri insertion

6 Exercices

Exercice 1

1. Ecrire une fonction python nommée **smul** à deux paramètres, un nombre et une liste de nombres, qui multiplie chaque élément de la liste par le nombre et renvoie une nouvelle liste : par exemple, **smul**$(2, [1, 2, 3]) \longrightarrow [2, 4, 6]$.
2. Ecrire une fonction python **vsom** qui prend en argument deux listes, avec la contrainte que si elles n'ont pas la même longueur, on renvoie False et sinon on renvoie la somme terme à terme de ces deux listes dans une nouvelle liste.

Exercice 2

On souhaite stocker pour une suite $(u_n)_n$ définie par u_0 et $\forall i \in \mathbb{N}$, $u_i = 2 * u_{i-1} + i$ les valeurs de $u_0, \ldots, u_n$ dans une liste.
Ecrire une fonction **suite** d'argument n et u_0 qui renvoie cette liste.
Pour $n = 10$ et $u_0 = 1$, vous devez obtenir : $suite(1, 10)[1, 3, 8, 19, 42, 89, 184, 375, 758, 1525, 3060]$.

Exercice 3

1. Soit l'entier $n = 1234$. Quel est le quotient, noté q dans la division euclidienne de n par 10 ? Quel est le reste ? Que se passe-t-il si on recommence la division par 10 de q ?
2. Ecrire la suite d'instructions calculant la somme des cubes des chiffres de l'entier 1234.
3. Ecrire une fonction python **somme**, d'argument N, en entier naturel, renvoyant la somme des cubes des chiffres constituant le nombre entier N.
4. Ecrire une fonction python **chaine** qui convertit l'entier N en une chaine de caractères qui contient ses chiffres sous forme de caractères.

Exercice 4

Soit n un entier naturel. on note D_n l'ensemble des entiers naturels compris entre 0 et $2^n - 1$. On appelle "point de $D_n \times D_n$" tout couple d'entiers $(x, y) \in D_n \times D_n$. Soient P et Q deux parties de $D_n \times D_n$. On cherche à calculer efficacement l'intersection des ensembles de points P et Q.

Un point de coordonnées $(x, y) \in D_n \times D_n$ est représenté en PYTHON par une liste de deux entiers naturels `[x,y]`. Un ensemble de points est représenté par une liste de points sans répétition, donc comme une liste de listes d'entiers naturels de longueur 2.

1. Ecrire une fonction `EGAL(a,b)` qui renvoie True sur deux listes sont identiques et False sinon.
2. Ecrire une fonction `membre(p,q)` qui renvoie `True` si le point `p` est dans l'ensemble représenté par la liste `q` et qui renvoie `False` dans le cas contraire.
3. Ecrire une fonction `intersection(l,q)` qui renvoie une liste représentant l'intersection des ensembles représentés par `l` et `q`. On implémentera l'algorithme qui consiste à itérer sur tous les points de `l` et à insérer dans le résultat ceux qui sont aussi dans `q`
4. Si la comparaison entre deux entiers naturels est prise comme opération élémentaire, quelle est la complexité de l'algorithme de la question précédente exprimée en fonction de la longueur de `p` et `q` ?

Exercice 5

Nous allons considérer les matrices comme des listes de listes. Nous allons redéfinir les fonctions classiques sur les matrices sans utiliser les commandes de numpy mais juste la structure de listes de listes.

1. Définir les matrices suivantes :

$$A_1 = \begin{pmatrix} 2 & 3 & 4 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}, \quad A_2 = \begin{pmatrix} 1 & 2 \\ 5 & 6 \\ 7 & 9 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 3 \\ 1 & 4 & 7 \end{pmatrix}, \quad b = \begin{pmatrix} 10 \\ 5 \\ 14 \end{pmatrix}$$

   Ces matrices vous serviront à faire des tests dans la suite sur vos fonctions !
2. Comment récupérer la taille d'une matrice ?
3. Ecrire une fonction **Echange**(A,i,j) qui échange les lignes i et j de A (en numérotation python) lorsque cela est possible et sinon renvoie un message d'erreur. On pourra utiliser la commande assert. Quelle est la complexité de cette fonction ?
4. Ecrire une fonction **Transvection**(A,i,j,μ) qui ajoute à la ligne i, μ fois la ligne j (en numérotation python) lorsque cela est possible et sinon renvoie un message d'erreur.
5. Ecrire une fonction **Transpose**(A) qui renvoie la matrice transposée de A.
6. Ecrire une fonction **Produit**(A,B) qui renvoie le produit de deux matrices lorsque cela est possible sinon renvoie un message d'erreur. Quelle est la complexité de cette fonction ?
7. On se donne une matrice $A \in \mathcal{M}_{n,p}(\mathbb{R})$ et $b \in \mathcal{M}_{n,1}(\mathbb{R})$. On souhaite alors résoudre le système linéaire ${}^tAAX = {}^tAb$. On remarque que ${}^tAA \in \mathcal{M}_2(\mathbb{R})$.

   Proposer une fonction **Solve**(A,b) qui réponde à la question, lorsque cela est possible, et vous la testerez avec les matrices A_2 et b. Vous devez trouvez

$$X = \begin{pmatrix} -12.54 \\ 11.3 \end{pmatrix}$$

Exercice 6

```python
def mystere1(n):
    N=n
    i=0
    T=[]
    while N>0:
        T.append(N%2)
        N=N//2
        i=i+1
    return T,i
```

1. Que fait cet algorithme ?
2. Déterminer sa complexité .
3. Montrer que pour tout i, on a $n = N_i \times 2^i + \sum_{j=0}^{i-1} T[j] \times 2^j$.
4. Montrer que l'algorithme fait bien ce qu'on a supposé à la question 1.

Exercice 7

1. Ecrire une fonction itérative pour calculer x^n avec x un nombre et n un entier. La fonction vérifiera avec assert que n est bien un entier.
2. Ecrire une fonction récursive pour calculer x^n avec x un nombre et n un entier. La fonction vérifiera avec assert que n est bien un entier.
3. Comparer la complexité de vos deux algorithmes.
4. Connaissez vous un algorithme de meilleur complexité. Si oui, implémentez le et justifiez sa complexité.

Exercice 8

On dispose des températures à Aix en Provence à 8h00 dans un dictionnaire :

```
temp ={ J1 : -5 , J2 : -3 ,J3 : 3 ,J4 :0 , J5 : -1 ,J6 :4 , J7 : -5 ,J8 :1 , J9 : -2}
```

1. Écrire une fonction moyenne qui prend en argument un dictionnaire d du type de celui défini ci-dessus et qui renvoie la valeur moyenne des températures.
2. Écrire une fonction froid(d,T0) qui prend en argument un dictionnaire d du type de celui défini ci-dessus et qui renvoie la liste des jours et le nombre de jours où la température a été inférieure ou égale à une certaine température T0.

Exercice 9

1. Ecrire une fonction en python **moyenne1(L)** pour calculer la moyenne des éléments dans une liste d'entiers. Votre algorithme sera de complexité linéaire. Vous justifierez avec soin ce dernier point.
2. Ecrire une fonction en python **occurrence(L)** ,L étant une liste d'entiers, renvoyant un dictionnaire qui a pour clé les valeurs distinctes contenues dans L et qui a chaque clé fait correspondre l'occurrence de la clé dans la liste L.
3. Ecrire une fonction en python **moyenne2(D)** pour calculer la moyenne des éléments du dictionnaire créée précédemment. Votre algorithme sera de complexité linéaire. Vous justifierez avec soin ce dernier point.
4. Comparer les résultats de **moyenne1(L)** et **moyenne2(occurrence(L))** sur une liste L de nombres entiers aléatoires. Comparer également la complexité de ces deux fonctions.

Exercice 10

Étant donné un dictionnaire python dont les clés sont les noms des élèves et les valeurs sont les listes des notes et qui renvoie un autre dictionnaire dont les clés sont le nom des élèves et les valeurs la moyenne de leur note. Par exemple :

```
nom_moy({Louise:[12,15,12],Gabriel:[15,17,16],Léon:[8,18,7]})
```

renverra

```
{Louise:13,Gabriel:16,Léon:11}.
```

Exercice 11

On appelle rotation d'un tableau t le fait de décaler tous les éléments d'une place vers la droite, à l'exception du dernier qui est placé en première place. Par exemple, la rotation du tableau $[1, 2, 3, 4]$ est le tableau $[4, 1, 2, 3]$.

1. Rédiger une fonction **rotation(t)** qui renvoie un nouveau tableau égal à la rotation du tableau initial.
2. Rédiger une fonction **rotation2(t)** qui modifie le tableau t pour le remplacer par sa rotation.
3. Rédiger une fonction **rotation3(k, t)** qui renvoie un nouveau tableau égal à k rotations du tableau t. Vous préciserez la complexité de votre algorithme.

Exercice 12

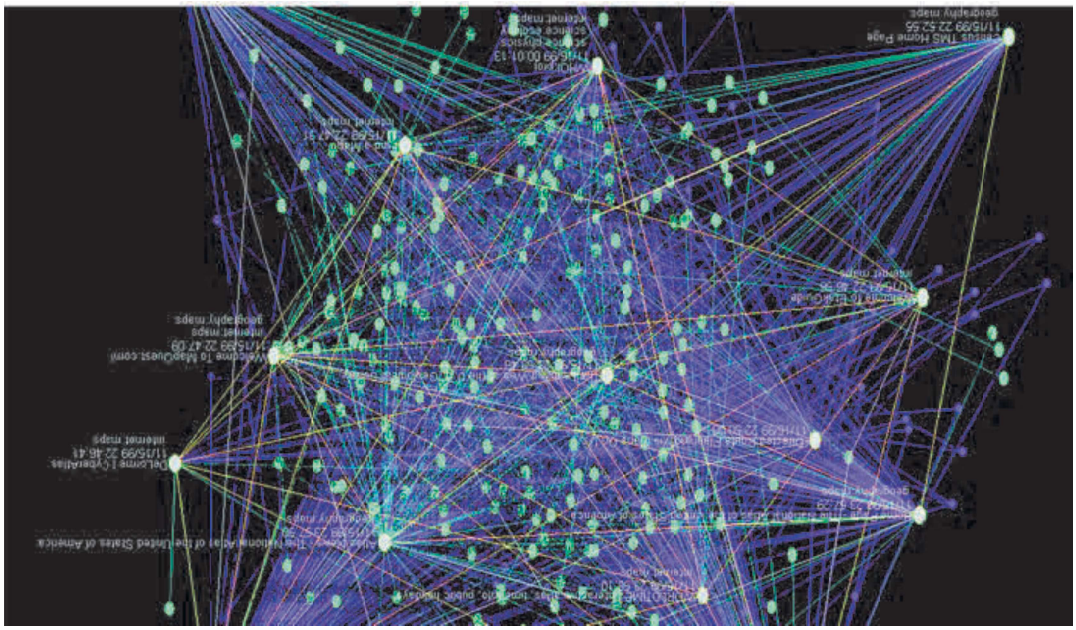
Un tableau non vide t de n entiers relatifs étant donné, on cherche la valeur maximale de la quantité

$$\Delta_k = (t[0] + t[1] + \cdots + t[k]) - (t[k+1] + \cdots + t[n-1])$$

lorsqu'on fait varier k dans $[|0; n-2|]$.

1. Rédiger une fonction **delta(k, t)** qui calcule la quantité Δ_k et en déduire une fonction **equilibre(t)** qui résout le problème posé. Évaluer la complexité temporelle de cette dernière fonction.
2. Trouver une fonction **equilibre2(t)** qui résout ce problème en temps linéaire.

Chapitre 2 : Les graphes



Imaginez un réseau informatique d'une entreprise de milliers d'employés. Ce réseau est formé des ordinateurs et des liens qu'il existe ou pas entre eux. Nous allons essayer de répondre aux questions suivantes :

— Est ce que deux employés donnés peuvent communiquer entre eux ?

— Est ce que tout employé peut communiquer avec un autre employé ?

— Si les liens entre les ordinateurs sont pondérés d'un coût, existe-t-il un meilleur chemin pour communiquer entre deux employés ?

— ....

7 Vocabulaire

7.1 Présentation des graphes

Définition

Un **graphe** est un type abstrait de données relationnel qui est constitué de **sommets** et de relations entre les sommets qu'on appelle **arcs**, s'ils sont orientés, ou **arêtes** s'ils ne sont pas orientés.

Définition

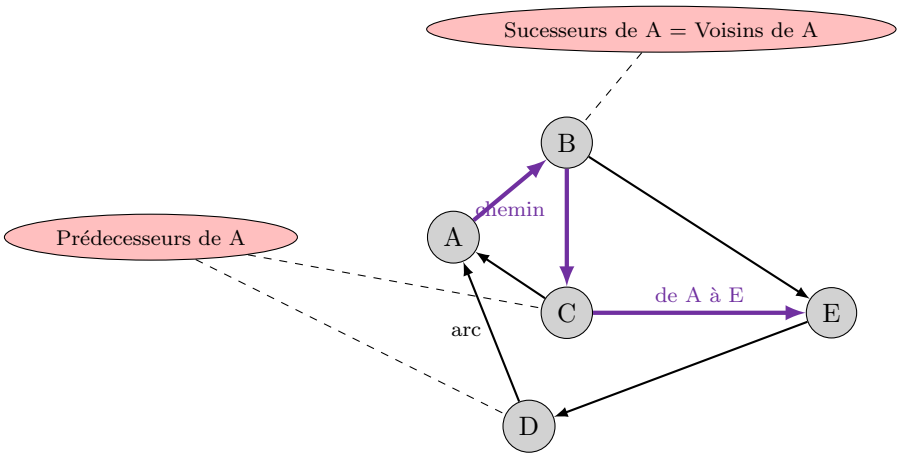
On considère un graphe quelconque.
On appelle **chaîne ou chemin** toute succession d'arêtes ou d'arcs dont l'extrémité de l'une (sauf la dernière) est l'origine de la suivante.
Le nombre d'arêtes qui composent une chaîne est appelé **longueur de la chaîne**.
On appelle **chaîne fermée ou chemin fermé** toute chaîne dont l'origine et l'extremité coincident.
On appelle **cycle** toute chaîne fermée dont les arêtes sont toutes distinctes.

Définition

L'ordre d'un graphe est le nombre de sommets du graphe.
Un sommet B est **adjacent** à A, ou B est voisin de A lorsque A est relié à B par une arête/ un arc.

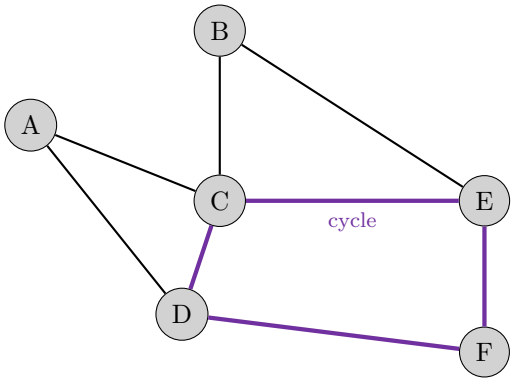
Voici quelques exemples de graphes :

1. **Un graphe orienté**



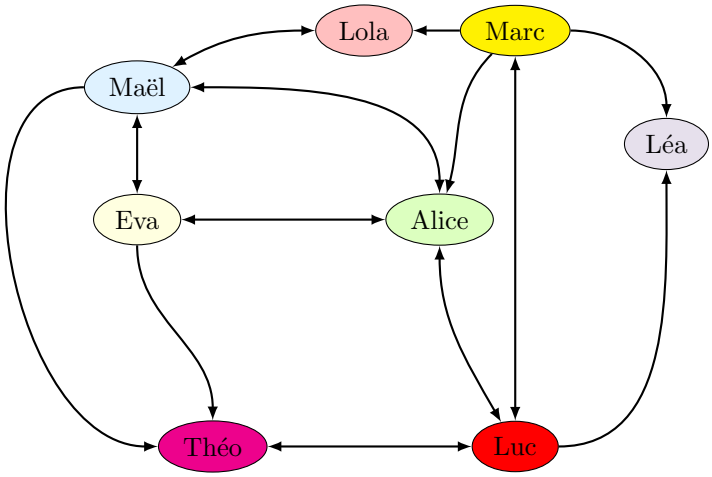
Les sommets du graphe sont A, B, C, D, E.
Les voisins de B sont C et E.

2. **Un graphe non orienté**

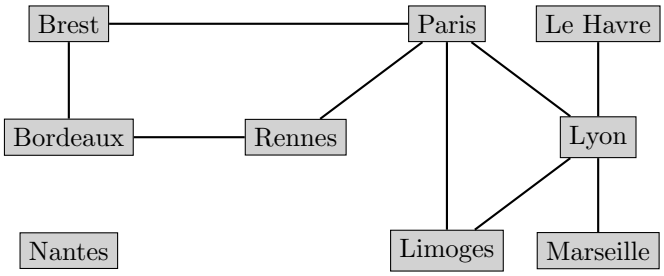


Les sommets du graphe sont A, B, C, D, E, F.
Les voisins de E sont C et F.

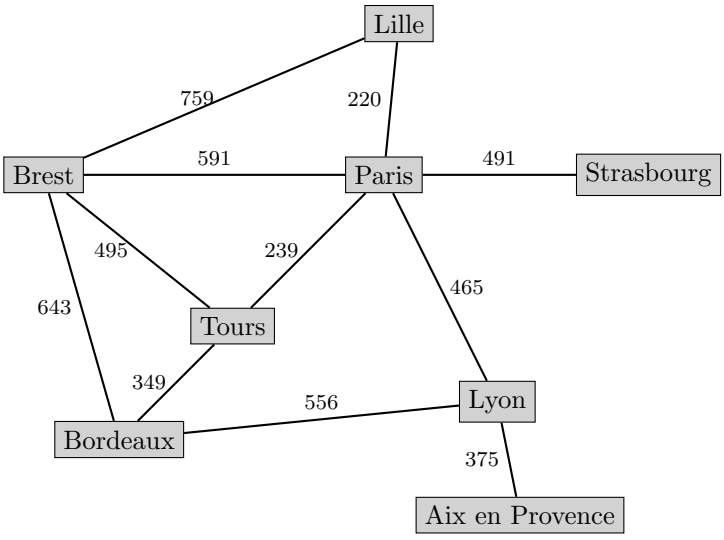
3. **Un réseau social « est ami avec »**
   C'est un graphe orienté, les relations sont représentées par des arcs fléchés.



4. **Liaisons aériennes**
   C'est un graphe non orienté, les relations sont représentées par des arêtes. (Nantes est un sommet isolé, il n'est relié à aucun autre sommet).



5. **Itinéraires routiers**
   Les arêtes de ce graphe pondéré portent les distances kilométriques séparant les villes.



7.2 Cycle, Connexité d'un graphe

Définition

On considère un graphe quelconque. On appelle **cycle** toute chaîne fermée (ou chemin fermé) dont les arêtes sont toutes distinctes.

Par exemple, dans le graphe du Réseau social

— Lola-Maël-Alice-Luc est une chaine de longueur 3

— Lola-Maël-Eva-Alice-Maël-Alice-Luc-Marc-Lola est une chaine fermée mais n'est pas un cycle car une arête se répète deux fois.

— Lola-Maël-Eva-Alice-Luc-Marc-Lola est un cycle.

Définition

Un graphe est dit **connexe** si pour tous sommets u et v il existe une chaîne qui les relie.

Par exemple,

— les graphes du Réseau social, des itinéraires routiers sont connexes.

— le graphe des liaisons aériennes n'est pas connexe.

7.3 Degré d'un sommet

Définition

• Pour un graphe non orienté, le **degré d'un sommet S** est égal au nombre d'arêtes sortant (ou entrant) de S. On note le $d(S)$.

• Pour un graphe orienté,

— le **demi-degré exterieur** de S est égal au nombre d'arcs sortants de S. On le note $d_+(S)$.

— le **demi-degré interieur** de S est égal au nombre d'arcs entrants de S. On le note $d_-(S)$.

Par exemple,

— Dans le graphe non orienté des liaisons aériennes, le degré de Paris est 4 : $d(Paris) = 4$

— Dans le graphe orienté du réseau social : $d_+(Lola) = 1$, $d_-(Lola) = 2$.

8 Représentation d'un graphe

On va utiliser une ou deux variables pour stocker l'ensemble des sommets S et des arêtes A.

Soit S l'ensemble des sommets d'un graphe G. On appelle A l'ensemble :

— Des arêtes d'un graphe non orienté tel que $A = \{\{s_i, s_j\}, s_i \in S, s_j \in S\}$

— Des arcs d'un graphe orienté tel que $A = \{(s_i, s_j), s_i \in S, s_j \in S\}$

On note $G = (S, A)$ le graphe composé de ces sommets et arêtes/arcs. On note $G = (S, A, \omega)$ un graphe pondéré tel que chaque arc/arête a un poids $\omega(s_i, s_j)$.

Remarque

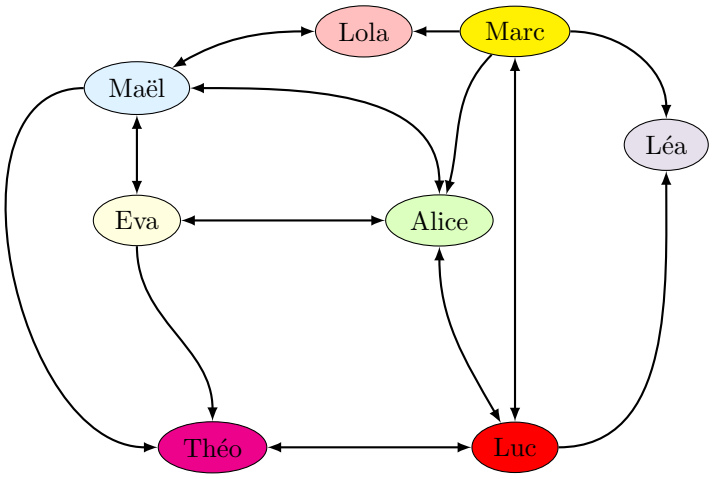
On notera la différence entre $\{(s_i, s_j)\}$ où il n'y a pas d'ordre (arête) et (s_i, s_j) où il y a une ordre (arc orienté).

8.1 Matrices d'adjacence avec un graphe non pondéré

Définition

La matrice d'adjacence est un tableau de dimension 2 dans lequel on indique la présence ou le poids d'un arc (ou d'une arête) à l'intersection de la ligne et le colonne dont les numéros permettent d'identifier les deux sommets ainsi reliés.

Reprenons l'exemple du réseau social, en numérotant les amis :



0 : Lola ; 1 : Marc ; 2 : Léa ; 3 : Luc ; 4 : Théo ; 5 : Eva ; 6 : Maël ; 7 : Alice.

On peut alors écrire la liste des sommets de ce graphe dans une liste et la matrice d'adjacence correspondante au graphe.

```python
#Réseau social
Sommet_G3=["Lola","Marc", "Léa", "Luc", "Théo",
"Eva","Maël","Alice"]

d3={'Lola' : 0, 'Marc' :1, 'Léa' :2 , 'Luc' :3,
'Théo':4,'Eva' :5,'Maël' :6,'Alice' :7}

M=[[0 for j in range(8)] for i in range(8)]
#Liens de Lola
M[0][6]=1# avec Maël
#Liens de Marc
M[1][0]=1; M[1][2]=1;M[1][3]=1;M[1][7]=1
#Liens de Léa : aucun
#Liens de Luc
M[3][1]=1;M[3][2]=1;M[3][4]=1;M[3][7]=1
#Liens de Théo
M[4][3]=1
#Liens de Eva
M[5][4]=1;M[5][6]=1;M[5][7]=1
#Liens de Maël
M[6][0]=1;M[6][4]=1;M[6][5]=1;M[6][7]=1
#Liens de Alice
M[7][3]=1;M[7][5]=1;M[7][6]=1

for i in range(len(M)):
    for j in range(len(M)):
        if M[i][j]!=0:
            print(S[i],"->",S[j])
```

Test

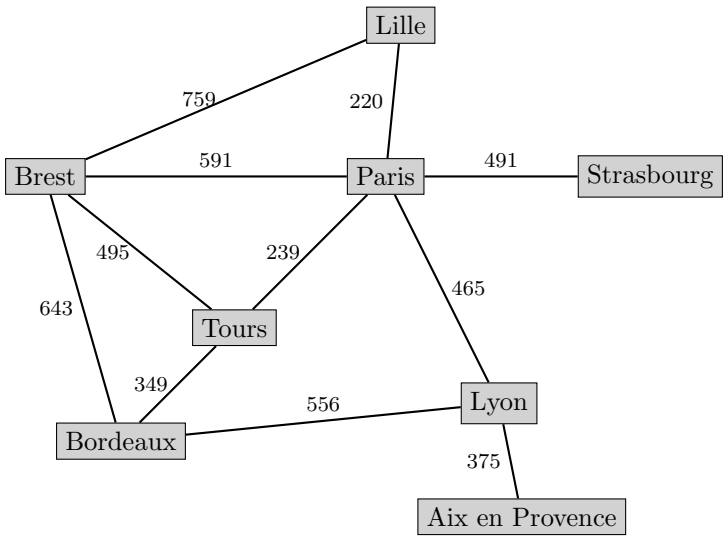
```
Lola -> Maël
Marc -> Lola
Marc -> Léa
Marc -> Luc
Marc -> Alice
Luc -> Marc
Luc -> Léa
Luc -> Théo
Luc -> Alice
Théo ->Luc
Eva -> Théo
Eva -> Maël
Eva -> Alice
Maël -> Lola
Maël -> Théo
Maël -> Eva
Maël -> Alice
Alice -> Luc
Alice -> Eva
Alice -> Maël
```

Remarque

Une matrice d'adjacence d'un graphe non orienté est **toujours symétrique**.

8.2 Matrices d'adjacence (ou matrices des distances) avec un graphe pondéré

Pour cela, on reprend l'exemple du graphe des **Itinéraires routiers**.
Les arêtes de ce graphe pondéré portent les distances kilométriques séparant les villes.



La matrice d'adjacence va tenir compte de la pondération. Ainsi si il y a 0 c'est qu'aucun lien n'existe et sinon on met la pondération.

Voilà les lignes de code pour la matrice d'adjacence des Itinéraires routiers. On numérote les villes par ordre alphabétique.

0 : Aix en Provence ; 1 : Bordeaux ; 2 : Brest ; 3 : Lille ; 4 : Lyon ; 5 : Paris ; 6 : Strasbourg ; 7 : Tours

```python
Sommet_G5=["Aix en Provence","Bordeaux",
"Brest"," Lille ","Lyon","Paris","Tours",
"Strasbourg"]

d5={"Aix en Provence":0,"Bordeaux":1,
"Brest":2,
" Lille ":3,"Lyon":4,"Paris":5,
"Tours":6,"Strasbourg":7}


R=[[0 for k in range(8)] for i in range(8)]

R[0][4]=R[4][0]=375
R[1][2]=R[2][1]=643
R[1][7]=R[7][1]=349
R[1][4]=R[4][1]=556
R[2][3]=R[3][2]=759
R[2][5]=R[5][2]=591
R[2][7]=R[7][2]=495
R[3][5]=R[5][3]=220
R[5][7]=R[7][5]=491
R[5][6]=R[6][5]=465
R[5][6]=R[6][5]=239
```

Test

```
>>>R
[[0, 0, 0, 0, 375, 0, 0, 0],
[0, 0, 643, 0, 556, 0, 0, 349],
[0, 643, 0, 759, 0, 591, 0, 495],
[0, 0, 759, 0, 0, 220, 0, 0],
[375, 556, 0, 0, 0, 0, 0, 0],
[0, 0, 591, 220, 0, 0, 239, 491],
[0, 0, 0, 0, 0, 239, 0, 0],
[0, 349, 495, 0, 0, 491, 0, 0]]
```

8.3 Liste d'adjacence à l'aide d'une liste

Définition

Lorsqu'il y a peu d'arc ou d'arêtes sur un graphe, il devient plus intéressant de mémoriser pour chaque sommet du graphe, seulement la liste des voisins (par leur numéro ou leur étiquette). Cette liste est appelée **la liste d'adjacence du graphe**.

On reprend l'exemple du graphe orienté du réseau social, sa liste d'adjacence correspondante sera

```python
#Réseau social
Sommet_G3=["Lola","Marc", "Léa", "Luc", "Théo","Eva","Maël","Alice"]
L_G3=[[6],
    [0,2,3,7],
    [],
    [1,2,4,7],
    [3],
    [4,6,7],
    [0,4,5,7],
    [3,5,6]]

for i in range(len(L_G3)):
    for j in L_G3[i]:
        print(Sommet_G3[i], "-->",Sommet_G3[j])
```

8.4 Liste d'adjacence à l'aide d'un dictionnaire

Définition

On peut également définir un graphe à l'aide d'un dictionnaire. Les clés sont les sommets et les valeurs les voisins des sommets. Ce dictionnaire est appelé **le dictionnaire d'adjacence** du graphe.

```python
Dic_G3=dict()
Dic_G3["Lola"]=["Maël"]
Dic_G3["Marc"]=["Lola","Léa","Luc","Alice"]
Dic_G3["Léa"]=[]
Dic_G3["Luc"]=["Marc","Léa","Théo","Alice"]
Dic_G3["Théo"]=["Luc"]
Dic_G3["Eva"]=["Théo","Maël","Alice"]
Dic_G3["Maël"]=["Lola","Théo","Eva","Alice"]
Dic_G3["Alice"]=["Luc","Eva","Maël"]
```

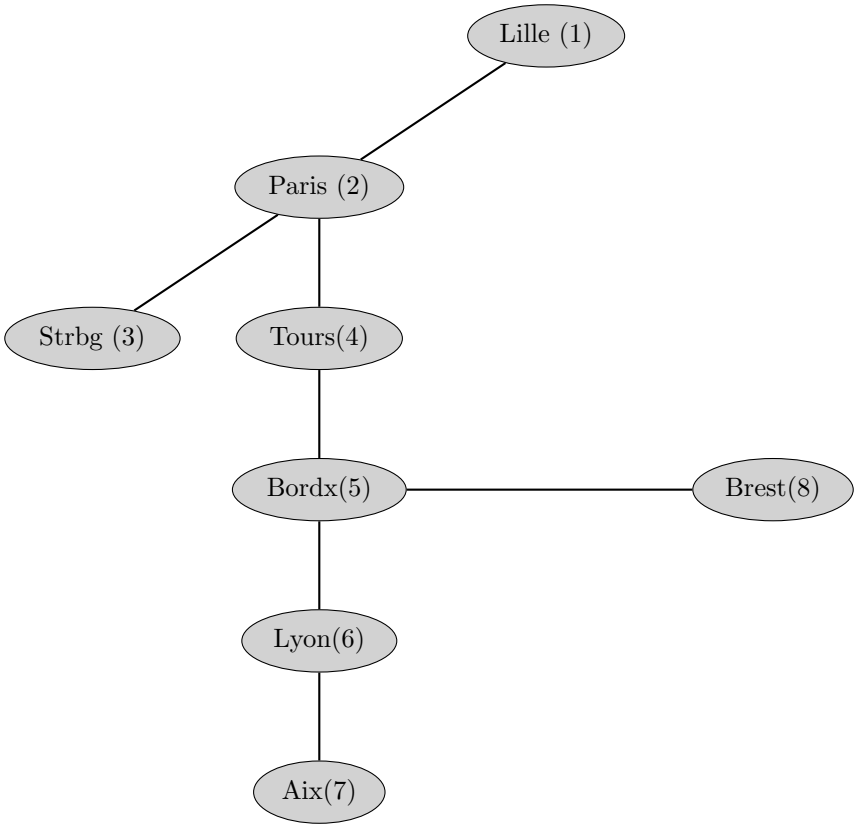
On peut également gérer les graphes pondérés comme celui des itinéraires routiers. On utilise des dictionnaires de dictionnaires.

```python
Dic_R={}
Dic_R['Aix en Provence']={'Lyon' : 375}
Dic_R['Bordeaux']={'Brest':643,'Tours':349,'Lyon':556}
Dic_R['Brest']={'Bordeaux':643,'Lille ':759,'Paris':591,'Tours':495}
Dic_R[' Lille ']={'Brest':759, 'Paris':220}
Dic_R['Lyon']={'Bordeaux': 556, 'Paris':465, 'Aix en Provence' : 375}
Dic_R['Paris']={'Brest':591,' Lille ':220,'Lyon':465,'Tours': 239,'Strasbourg':491}
Dic_R['Tours']={'Brest' : 495,'Bordeaux' : 643,'Paris' : 239}
Dic_R['Strasbourg']={'Paris': 491}
```

9 Parcours d'un graphe

9.1 Parcours en profondeur d'un graphe

Une fois choisi le sommet de départ, le parcours en profondeur (Depth-First-Search : DFS), consiste à avancer sur une branche tant que cela reste possible, ainsi à chaque étape on passe par le premier voisin qui se présente et qui n'a pas encore été parcouru avant de changer de branche. On commence en un sommet, on passe à un de ses enfants et ainsi de suite. S'il n'y a pas d'enfant, le programme revient au sommet précédent et passe à un autre enfant.



Une façon d'implémenter un parcours en profondeur est d'utiliser une pile pour stocker les sommets à parcourir. Nous présentons, ici, un algorithme itératif sur une liste d'adjacence ou un dictionnaire, **L**, représentant un graphe, à partir du sommet **S**.

```python
from collections import deque

def parcours_dfs(G,dep):
    visite =[]
    pile=deque()
    pile.append(dep)
    while len(pile)>0:
        sommet=pile.pop()
        if sommet not in visite:
            visite.append(sommet)
            for s in G[sommet]:
                if s not in visite :
                    pile.append(s)
    return visite
```

Test

```
>>>parcours_dfs(L_G3,0)
[0, 6, 7, 5, 4, 3, 2, 1]

>>>parcours_dfs(Dic_G3,'Lola')
['Lola', 'Maël', 'Alice', 'Eva',
'Théo', 'Luc', 'Léa', 'Marc']

>>>parcours_dfs(Dic_R,'Lille')
['Lille', 'Paris', 'Strasbourg',
'Tours', 'Bordeaux', 'Lyon',
'Aix en Provence', 'Brest']
```

Le test a été fait sur le réseau social et sur les itinéraires routiers. On retrouve bien la recherche en profondeur présentée au début.

On remarque que la lecture du résultat n'est pas toujours très agréable avec la liste d'adjacence, on peut alors écrire une fonction pour afficher le parcours :

```python
def affiche(chemin,S):
    ch=""
    if len(chemin)==0:
        return ch
    else:
        ch=S[chemin[0]]
        for i in range(1,len(chemin)):
            ch=ch+"-->"+S[chemin[i]]
        return ch
```

Test

```
>>> affiche(parcours_dfs(L_G3,0),Sommet_G3)
'Lola-->Maël-->Alice-->Eva-->Théo
-->Luc-->Léa-->Marc'
```

Nous pouvons également faire un programme récursif pour le parcours en profondeur :

```python
def parcours_rec(L,sommet,visite=[]):
    if sommet not in visite:
        visite.append(sommet)
    for voisin in L[sommet]:
        if voisin not in visite :
            parcours_rec(L,voisin, visite )
    return visite
```

Remarque

On remarque que la liste Visite est modifiée par les appels récursifs !

On peut également vouloir chercher s'il existe une chaîne entre deux sommets du graphe. Pour cela on adapte le parcours du graphe en cherchant à voir si en partant de 'dep' on peut atteindre 'arr'. Voici une façon d'adapter le parcours en profondeur :

```python
def parcours_dfs_dep_arr(G,dep,arr):
    visite=[]
    pile=deque()
    pile.append(dep)
    while len(pile)>0:
        sommet=pile.pop()
        if sommet not in visite:
            visite.append(sommet)
            if sommet==arr:
                return visite
            for s in G[sommet]:
                if s not in visite :
                    pile.append(s)
    return False
```

Test

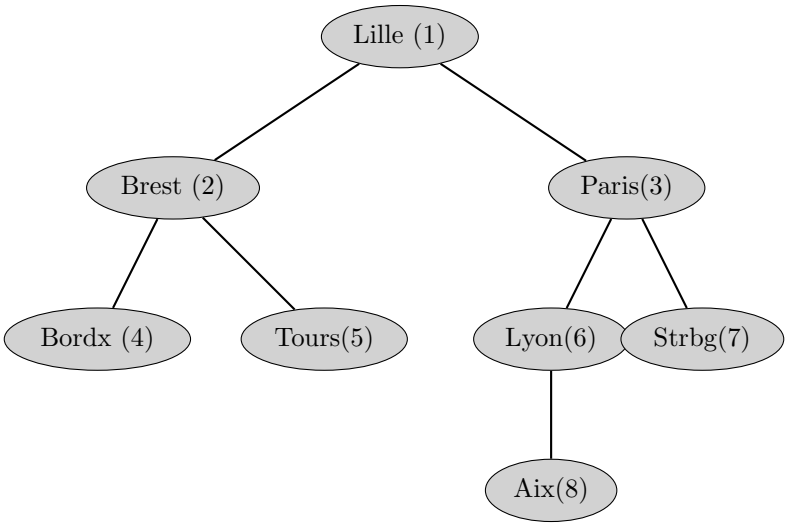
```
>>>parcours_dfs_dep_arr(Dic_G3,'Lola','Alice')
['Lola', 'Maël', 'Alice']

>>>parcours_dfs_dep_arr(Dic_G3,'Lola','Luc')
['Lola', 'Maël', 'Alice', 'Eva', 'Théo', 'Luc']

>>>parcours_dfs_dep_arr(Dic_G3,'Léa','Lola')
False

>>>parcours_dfs_dep_arr(Dic_R,'Lille',
'Aix en Provence')
[' Lille ', 'Paris', 'Strasbourg',
'Tours', 'Bordeaux', 'Lyon',
'Aix en Provence']
```

Remarque

On remarque que le parcours trouvé de Lille à Aix en Provence n'est peut être pas le plus court et qd il faut repasser par une ville déjà visité il ne nous le dit pas, ce n'est pas un bon moyen pour déterminer si il y a une chemin d'un sommet vers un autre !

9.2 Parcours en largeur d'un graphe

Le parcours en largeur consiste à examiner les sommets voisins puis les voisins des voisins et ainsi de suite... On commence en un sommet et on va voir tous ses enfants. Ensuite on visite tous les enfants de ces sommets. Il procède par éloignement progressif depuis le sommet de départ.



On emploie alors une file dans un algorithme itératif présenté ci dessous.

```python
def parcours_bfs(G,dep):
    visite =[]
    file =deque()
    file .appendleft(dep)
    while len( file )>0:
        sommet=file.pop()
        if sommet not in visite:
            visite .append(sommet)
            for voisin in G[sommet]:
                if voisin not in visite :
                    file .appendleft(voisin)
    return visite
```

Test

```
>>>parcours_bfs(Dic_G3,'Lola')
['Lola', 'Maël', 'Théo', 'Eva',
'Alice', 'Luc', 'Marc', 'Léa']

>>>parcours_bfs(Dic_R,'Lille')
['Lille', 'Brest', 'Paris',
'Bordeaux', 'Tours', 'Lyon',
'Strasbourg', 'Aix en Provence']
```

9.3 Recherche de connexité d'un graphe non orienté

Définition

Un graphe non orienté, est dit **connexe** si quelques soient les sommets u et v du graphe, il existe une chaine entre ces deux sommets.

Par exemple,

— le graphe non orienté des liaisons aeriennes est non connexe puisque le sommet Nantes n'a aucune arête de liaison avec un sommet du graphe.

— le graphe non orienté des itinéraires routiers est connexe puisque chaque sommet est au moins relié à un autre sommet.

Remarque

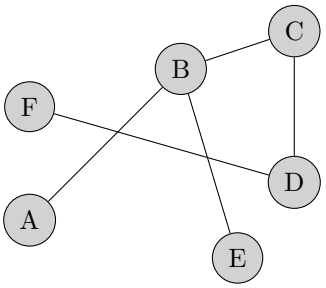
La notion de connexité d'un graphe est essentiel car elle permet de répondre à des questions très pratiques en réalité :

— A quel moment un réseau informatique satisfait-il à la propriété que tous les ordinateurs du réseau, pris deux à deux, puissent partager l'information ?

— A quel moment un réseau routier satisfait-il à la propriété que toutes les villes sont réliées les une aux autres ?

Pour tester si un graphe est connexe, on peut le parcourir et regarder si pour chaque sommet tous les autres sommets sont atteignables. On peut utiliser les fonctions de parcours des paragraphes précédents. On présente ici un test de connexité sur les graphes implémentés en liste d'adjacence ou à l'aide des dictionnaires.

Pour faire un test, on implémentera le graphe connexe suivant :




```python
def est_connexe_liste(L):
    for i in range(len(L)):
        for j in range(len(L)):
            if parcours_bfs_dep_arr(L,i,j)==False:
                return ('Le graphe n est pas connexe', 'pas de chemin entre ' ,i, 'et',j)
    return True

def est_connexe_dic(dic):
    for cle1 in dic:
        for cle2 in dic:
            if parcours_bfs_dep_arr(dic,cle1,cle2)==False:
                return ('Le graphe n est pas connexe', 'pas de chemin entre ' ,cle1, 'et',cle2)
    return True

Gtest=[[1],[0,2,4],[1,3],[2,5],[1],[3]]

Dictest={}
Dictest['A']=['B']
Dictest['B']=['A','C','E']
Dictest['C']=['B','D']
Dictest['D']=['C','F']
Dictest['E']=['B']
Dictest['F']=['D']
```

Test

```
>>>est_connexe_liste(Gtest)
True
>>>est_connexe_dic(Dictest)
True
>>>est_connexe_dic(Dic_G3)
('Le graphe n est pas connexe',
'pas de chemin entre ', 'Léa',
'et', 'Lola')
```

10 Le plus court chemin

10.1 Description de l'algorithme de Dijkstra

Certains problèmes consistent à chercher entre deux points donnés le parcours qui a une "longueur" (durée, coût, distance) minimum. L'algorithme de Dijkstra permet de résoudre ce type de problème dans les graphes pondérés connexes et à pondérations positives.

- **Les données du problème** :
  On considère un graphe connexe pondéré G et un sommet d'entrée E.
- **Objectif** :
  Les plus courtes distances du sommet E à chacun des sommets.
- **Traitement** :
  - Attribuer la marque ∞ à tous les sommets
  - Attribuer la marque 0 au sommet d'entrée E.
  - TANT QU' il reste des sommets non sélectionnés :
    * Sélectionner, parmi les sommets non-sélectionnés, le sommet A ayant la marque la plus petite.
    * Pour chaque sommet B adjacent à A et non déjà sélectionné :
      * Calculer p = (marque de A) + (poids de l'arête A-B).
      * SI p < marque de B
        ALORS remplacer marque de B par p
        SINON conserver marque de B.

    FIN TANT QUE

Au premier tour de boucle, c'est le sommet E qui est sélectionné (puisqu'il porte la marque 0 et les autres la marque ∞).

Si l'objectif est de calculer la distance du sommet E à un sommet S, on peut stopper l'algorithme dès que S est parmi les sélectionnés.

Si on veut connaître le chemin à suivre pour la plus courte distance, on ajoute à chaque étape une mémorisation du "parent" d'un sommet, ce qui donne :

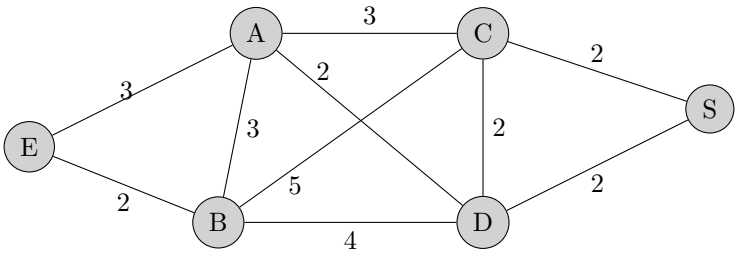
- **Les données du problème** :
  On considère un graphe connexe pondéré G et un sommet d'entrée E.
- **Objectif** :
  Les plus courtes distances du sommet E à chacun des sommets.
- **Traitement** :
  - Attribuer la marque ∞ à tous les sommets
  - Attribuer la marque 0 au sommet d'entrée E.
  - père(E) = /
  - TANT QU' il reste des sommets non sélectionnés :
    * Sélectionner, parmi les sommets non-sélectionnés, le sommet A ayant la marque la plus petite.
    * Pour chaque sommet B adjacent à A et non déjà sélectionné :
      * Calculer p = (marque de A) + (poids de l'arête A-B).
      * SI p < marque de B
        ALORS remplacer marque de B par p ; père(B)=A
        SINON /

    FIN TANT QUE

Ainsi, à la fin de l'algorithme, la marque du sommet d'arrivée est la longueur du plus court chemin et le chemin est obtenu en remontant la liste des père dans le graphe.

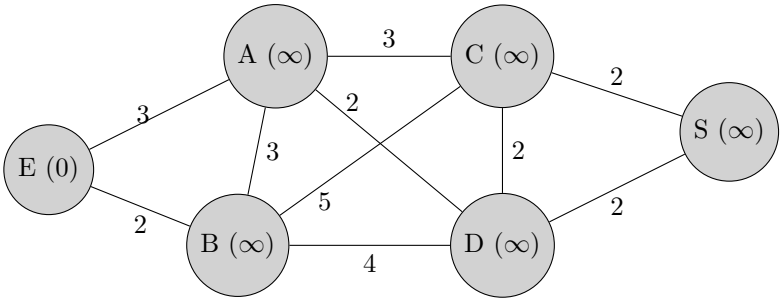
10.2 Un exemple

Nous allons décrire l'algorithme à travers un exemple en partant du graphe pondéré suivant.

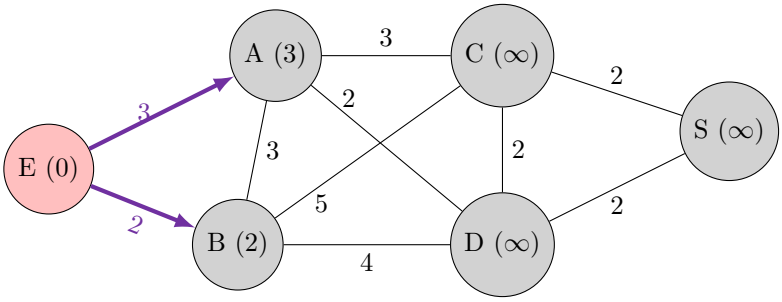


Nous cherchons le chemin le plus court entre les sommets E et S.

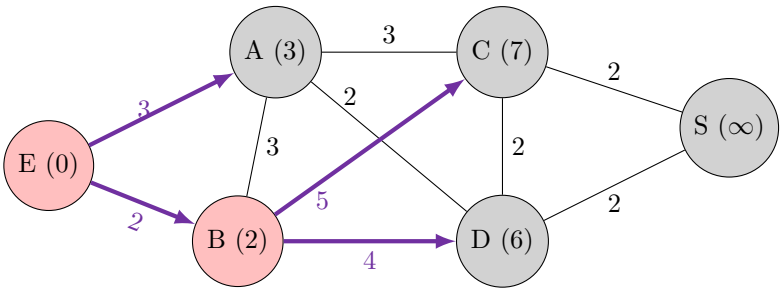
— Initialisation
On marque tous les sommets sauf celui de début à l'infini et celui du départ à 0.



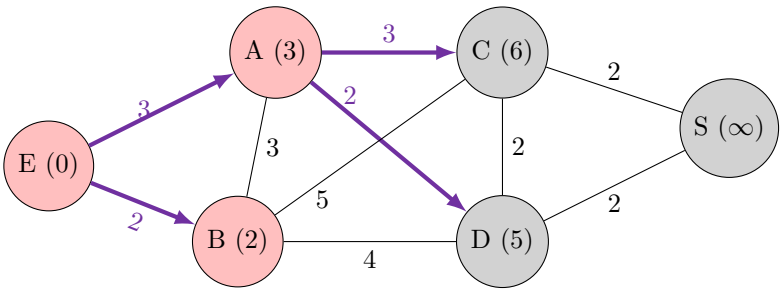
— On sélectionne E et on actualise la marque de ses voisins



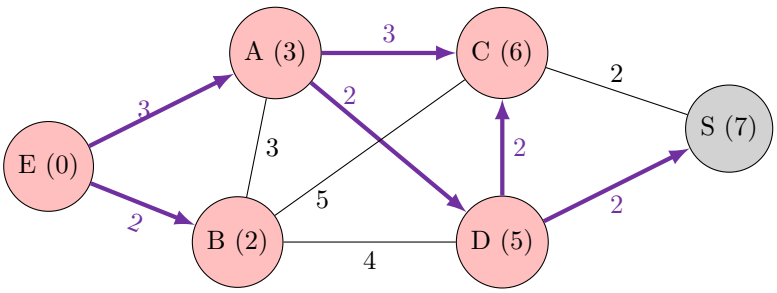
— Parmi les non selectionnés, on selectionne la marque la plus petite et on traite ses voisins non selectionnées.
La marque du sommet A est inchangée puisque marque(B)+poids(A-B)>marque(A).



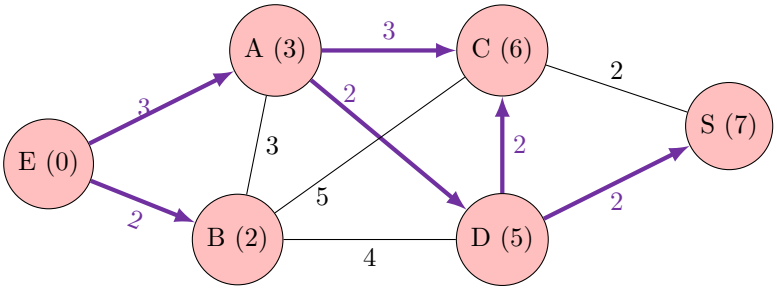
— Parmi les non selectionnés, on selectionne la marque la plus petite et on traite les voisins non selectionnés.
La marque de C est modifié car marque(A)+poids(A-C)<marque(C). La marque de D est modifié car marque(A)+poids(A-D)<marque(D). On modifie donc la couleur des arêtes (qui sert à enregistrer le père).



— Et on itère le procédé, jusqu'à arriver à S



— Dernière étape



Interprétation : La longueur minimale de E à S est 7 et en remontant les flèches violettes, on sait que le chemin correspondant est $E - A - D - S$.

10.3 Implémentation de l'algorithme de Dijkstra

```python
from math import inf
from collections import deque

Graphe ={
        'A':{'B':3, 'C':3, 'D':2, 'E':3},
        'B':{'A':3, 'C':5, 'D':4, 'E':2},
        'C':{'A':3, 'B':5, 'D':2, 'S':2},
        'D':{'A':2, 'B':4, 'C':2, 'S':2},
        'E':{'A':3, 'B':2},
        'S':{'C':2, 'D':2},
        }

def mini(listeSommets, marque):
        """
        Renvoie le sommet de listeSommets
        ayant la plus petite marque.
        """
        marquePlusPetite =inf
        for s in listeSommets:
                if marque[s] <marquePlusPetite:
                        marquePlusPetite =marque[s]
                        sommetPlusPetit =s
        return sommetPlusPetit


def dijkstra (graphe,depart,arrivee):
    marque={}
    for sommet in graphe:
        marque[sommet]=inf
    marque[depart]=0
    non_selectionnes=[sommet for sommet in graphe]
    pere={}
    pere[depart]=None
    while len(non_selectionnes)>0:
        s=mini(non_selectionnes,marque)
        if s==arrivee:
            break
        non_selectionnes.remove(s)
        VoisinsAvisiter =[sommet for sommet in graphe[s] if sommet in non_selectionnes]
        for sommet in VoisinsAvisiter:
            p=marque[s]+graphe[s][sommet]
            if p<marque[sommet]:
                marque[sommet]=p
                pere[sommet]=s
    return marque, pere
```

Test

```
>>> dijkstra (Graphe,'E','S')
({'A': 3, 'B': 2, 'C': 6, 'D': 5, 'E': 0, 'S': 7},
 {'E': None, 'A': 'E', 'B': 'E', 'C': 'A', 'D': 'A', 'S': 'D'})
```

Le premier dictionnaire est le dictionnaire des marques de chaque sommet à le fin de l'algorithme. Ainsi, le plus court chemin est la marque du sommet arrivée.

Le deuxième dictionnaire est la liste des pères qu'il faut remonter en partant du point arrivée pour trouver le chemin à parcourir.

On écrit alors une fonction qui affiche la distance et le chemin que nous testons sur le graphe de l'exemple et le graphe des liaisons routières du début du chapitre. .

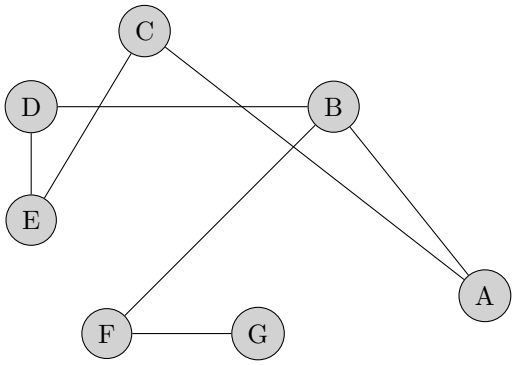
```python
def affiche_chemin_min(graphe,depart,arrivee):
    marque,pere=dijkstra(graphe,depart,arrivee)
    #on recupère la marque de tous les sommets
    #et la liste des père de chaque sommet
    #pour récuperer la distance du chemin, il suffit de prendre la marque de l'arrivée
    print('La distance de', depart, 'à', arrivee, 'est ', marque[arrivee])
    chemin=[arrivee]
    sommet=arrivee
    while pere[sommet]!=None:
        chemin=[pere[sommet]]+chemin
        sommet=pere[sommet]
    chemin=chemin
    print('Un chemin est ', chemin)

print(affiche_chemin_min(Graphe,'E','S'))

Dic_R={}
Dic_R['Aix en Provence']={'Lyon' : 375}
Dic_R['Bordeaux']={'Brest':643,'Tours':349,'Lyon':556}
Dic_R['Brest']={'Bordeaux':643,'Lille':759,'Paris':591,'Tours':495}
Dic_R['Lille']={'Brest':759, 'Paris':220}
Dic_R['Lyon']={'Bordeaux': 556, 'Paris':465, 'Aix en Provence':375}
Dic_R['Paris']={'Brest':591,'Lille':220,'Lyon':465,'Tours':239,'Strasbourg':491}
Dic_R['Tours']={'Brest' : 495,'Bordeaux' : 643,'Paris' : 239}
Dic_R['Strasbourg']={'Paris':491}

print(affiche_chemin_min(Dic_R,'Brest','Aix en Provence'))
```

Test

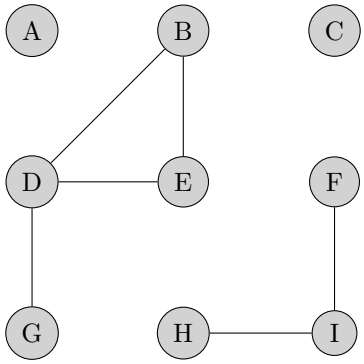
```
La distance de E à S est  7
Un chemin est ['E', 'A', 'D', 'S']

La distance de Brest à Aix en Provence est 1431
Un chemin est ['Brest', 'Paris', 'Lyon', 'Aix en Provence']
```

11 Exercices

Dans les exercices 1, 2, 3, 4 qui suivent, on testera les fonctions écrites sur les deux graphes ci-dessous.



Graphe 1



Graphe 2

Pour les matrices d'adjacence, le dictionnaire de correspondance sommet-indice est le suivant :

$$D1 = \{'A' : 0, 'B' : 1, 'C' : 2, 'D : 3, 'E' : 4, 'F' : 5, 'G' : 6\}$$
$$\text{D2} = \{\text{A}' : 0, \text{'B}' : 1, \text{'C}' : 2, \text{'D}' : 3, \text{'E}' : 4, \text{'F}' : 5, \text{'G}' : 6, \text{H}' : 7, \text{I}' : 8\}$$

Exercice 1

1. Ecrire M1 et M2 les matrices d'adjacences des graphes 1 et 2.
2. Ecrire Dico1 et Dico2 les dictionnaires d'adjacences des graphes 1 et 2.

Exercice 2

Recherche des voisins d'un sommet dans le cas du dictionnaire d'adjacence.
Soit un graphe G représenté par son dictionnaire d'adjacence.

1. Créer une fonction **Voisins_dic** (G, S) qui va renvoyer la liste des noms des sommets voisins du sommet S.
2. Testez votre fonction avec Dico1 et Dico2 qui représentent les graphes G_1 et G_2 (choisir un sommet de départ).

Exercice 3

Recherche des voisins d'un sommet dans le cas de la matrice d'adjacence.
Soit un graphe G représenté par le couple G = (M, D) où :

— M est la matrice d'adjacence,

— D est un dictionnaire avec pour clefs les noms des sommets et en valeur associée l'indice du sommet.

1. Créer une fonction **Voisins**(G, S) qui va renvoyer la liste des noms des sommets voisins du sommet S.
   On pourra créer un dictionnaire inversé ayant pour clé les numéros des sommets et pour valeur le nom du sommet correspondant.
2. Testez votre fonction avec G1 = (M1, D1) et G2 = (M2, D2) (choisir un sommet de départ).

Exercice 4

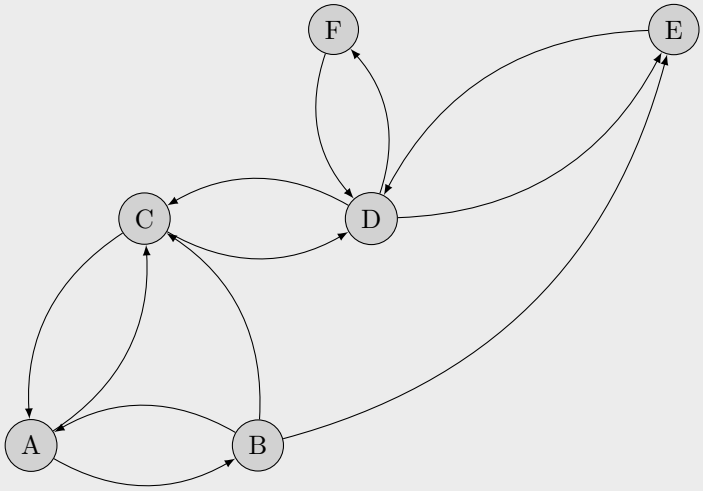
Dans cet exercice le graphe est représenté par son dictionnaire d'adjacence.

1. Ecrire une fonction **Parcours_Profondeur**(G,S) qui parcourt en profondeur le graphe G à partir du sommet S.
2. En déduire une fonction simple **est_connexe(G,S)** pour déterminer si le graphe est connexe. Vous ferez un test avec les deux graphes. Le premier est connexe, le second ne l'est pas.
   On pourra penser à comparer l'ordre du graphe à la longueur de la liste rendue par le parcours du graphe.
3. Ecrire une fonction simple **chemin(G,S1,S2)** pour déterminer s'il existe un chemin de S1 vers S2. Cette fonction renverra un booléen.

Exercice 5

Un groupe de randonneur veut s'organiser une randonnée dans les Alpes. Ils selectionnent les sommets qu'ils veulent faire. Ils étudient alors les chemins possibles entre ces différents sommets. Pour chaque sommet, ils relèvent deux informations la distance entre les deux sommets et le niveau (ou difficulté) de la randonnée entre les deux sommets.

Un exemple d'une telle situation est représentée dans le graphe suivant :



Les données entre les sommets sont les suivantes :

Sommet 1	Sommet 1	Distance	Difficulté
A	B	5 km	1
A	C	7 km	6
B	A	8 km	2
B	C	10 km	3
B	E	4 km	10
C	A	9 km	1
C	D	7 km	3
D	C	12 km	3
D	E	8 km	2
D	F	5 km	0
E	D	10 km	6
F	D	5 km	0

Un dictionnaire, nommé Randonnee1, représentant ce graphe est alors donné par

```python
Randonnee1={}
Randonnee1['A']={'B': (5,1),'C':(7,6)}
Randonnee1['B']={'A':(8,2), 'C':(10,3), 'E':(4,10)}
Randonnee1['C']={'A':(9,1),'D':(7,3)}
Randonnee1['D']={'C' : (12,3), 'E':(8,2), 'F':(5,0)}
Randonnee1['E']={'D':(10,6)}
Randonnee1['F']={'D':(5,0)}
```

ce qui donne lors d'un appel dans la console :

Test

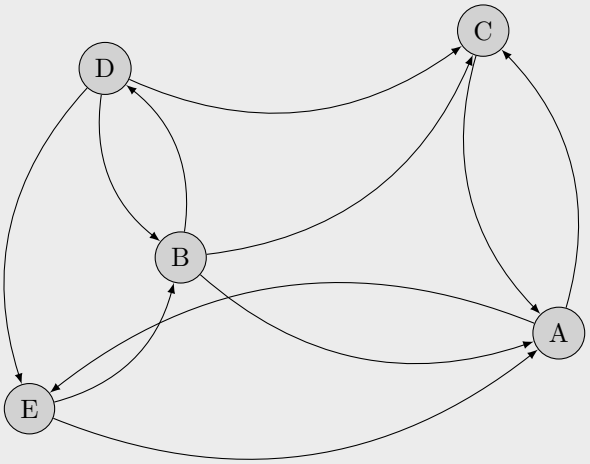
```
>>>Randonnee1
{'A': {'B': (5, 1), 'C': (7, 6)}, 'B': {'A': (8, 2), 'C': (10, 3), 'E': (4, 10)},
'C': {'A': (9, 1), 'D': (7, 3)}, 'D': {'C': (12, 3), 'E': (8, 2), 'F': (5, 0)},
'E': {'D': (10, 6)}, 'F': {'D': (5, 0)}}
```

1. Ecrire les lignes de commandes pour récupérer, à partir du dictionnaire Randonnee1, définissant le graphe de la randonnée présentée, la distance pour aller du sommet C au sommet D et la difficulté. Votre résultat dans la console devra ressembler à

Test

```
La distance pour aller du sommet C au sommet D est ...
La difficulté pour aller du sommet C au sommet D est ...
```

Voilà un autre graphe représentant une deuxième randonnée



Les données entre les sommets sont les suivantes :

Sommet 1	Sommet 1	Distance	Difficulté
A	C	15 km	2
A	E	20 km	6
B	A	15 km	2
B	C	7 km	3
B	D	8 km	7
C	A	10 km	4
D	B	9 km	3
D	C	9 km	4
D	E	10 km	2
E	A	5 km	6
E	B	4 km	5

2. Ecrire les lignes de commande en python pour créer le dictionnaire, noté Randonnee2, pour représenter ce nouveau graphe de randonnée. Il aura la même configuration que le dictionnaire Randonnee1. Dorénavant les graphes définissant une randonnée seront du type de Randonnee1, Randonnee2.
3. Un randonneur se demande à partir d'un sommet combien de sommets il peut atteindre en une journée (soit d'un sommet à l'autre directement, sans passer par un autre sommet) et quel est le sommet possédant le plus de sommet atteignable en une journée. Nous allons répondre à sa question. Pour clarifier, nous appellerons voisin direct d'un sommet les autres sommets atteignables en une journée.
   (a) Ecrire une fonction **Nombres_Voisins(G,sommet)** prenant en argument un graphe G d'une randonnée et un sommet du graphe et renvoyant son nombre de voisins directs.
   (b) Ecrire une fonction **Meilleur_Sommet(G)** prenant en argument un graphe d'une randonnée et renvoyant le sommet ayant le plus de voisins directs.
   (c) Déterminer la complexité de votre algorithme **Meilleur_Sommet**

4. Un autre randonneur se demande s'il est possible de faire une randonnée qui permet de visiter tous les sommets du graphe de la randonnée à partir de n'importe quel sommet au départ.
   (a) Donner une propriété que le graphe représentant la randonnée doit respecter pour satisfaire ce point.
   (b) Ecrire une fonction **parcours** d'argument un graphe, un sommet de départ et un sommet d'arrivée qui renvoie True s'il existe un chemin entre le sommet de départ et celui de l'arrivée et False sinon.
   (c) Ecrire une fonction **Randonnee_Possible** d'argument un graphe, représentant une randonnée, qui renvoie True s'il satisfait la condition demandée par le randonneur et sinon False.
5. Vous trouverez en annexe l'algorithme de Dijkstra implémenter pour un graphe ne contenant qu'une information entre deux sommets, du type *Graphe* donné également en annexe.
   (a) Ecrire la (ou les) ligne(s) à modifier, vous préciserez les numéros de lignes, dans l'algorithme de Dijkistra présenté en annexe pour qu'il détermine entre les sommets dep et arr d'un graphe de randonnée le chemin le plus court en distance.
   (b) Ecrire une fonction en python **distance(graphe, dep, arr)** qui rendra le chemin déterminer précédemment sous forme de liste avec les sommets à visiter et la distance totale parcourue.
6. (a) Ecrire la (ou les) ligne(s) à modifier, vous préciserez les numéros de lignes, dans l'algorithme de Dijkistra présenté en annexe pour qu'il détermine entre les sommets dep et arr d'un graphe de randonnée le chemin le plus facile.
   (b) Ecrire une fonction en python **difficulte(graphe, dep, arr)** qui déterminera entre deux sommets, le chemin le plus facile. Votre fonction devra rendre le chemin sous forme de liste avec les sommets à parcourir.

ANNEXE
L'algorithme de Djikistra

```python
from math import inf
from collections import deque

Graphe ={
        'A':{'B':3, 'C':3, 'D':2, 'E':3},
        'B':{'A':3, 'C':5, 'D':4, 'E':2},
        'C':{'A':3, 'B':5, 'D':2, 'S':2},
        'D':{'A':2, 'B':4, 'C':2, 'S':2},
        'E':{'A':3, 'B':2},
        'S':{'C':2, 'D':2},
        }

def mini(listeSommets, marque):
        """
        Renvoie le sommet de listeSommets
        ayant la plus petite marque.
        """
        marquePlusPetite =inf
        for s in listeSommets:
                if marque[s] <marquePlusPetite:
                        marquePlusPetite =marque[s]
                        sommetPlusPetit =s
        return sommetPlusPetit


def dijkstra (graphe, depart, arrivee):
        #  initialisation
        marque ={}#dictionnaire contenant les marques des sommets au début
        #tous à l'infini sauf le sommet départ
        for sommet in graphe:
          marque[sommet] =inf
        marque[depart] =0
        non_selectionnes =[sommet for sommet in graphe]
        pere = {}
        pere[depart] = None
        # boucle principale :
        while non_selectionnes:
                # sélection :
                s = mini(non_selectionnes, marque)
                if s == arrivee:
                break
                non_selectionnes.remove(s)
                # mise à jour des voisins du sommet sélectionne:
                VoisinsAVisiter = [sommet for sommet in graphe[s] if sommet in non_selectionnes]
                for sommet in VoisinsAVisiter:
                        p = marque[s] +graphe[s][sommet]
                        if p < marque[sommet]:
                                marque[sommet] =p
                                pere[sommet] =s

        return marque, pere
```