Introduction

Ce polycopié reprend, uniquement, les **notions ajoutées** dans le programme de 1re année à la rentrée 2021 par rapport au programme que vous avez eu.

Sans précision, tout ce que vous avez vu en informatique dans l'ancien programme au cours de votre 1re et de votre 2e année est toujours au programme.

Passage 1re année → 2e année

Il faudra donc un peu suivre le cours sur les bases de données, même si vous étiez à l'aise.

Initiation aux bases de données.	**Disparition de l'algèbre relationnelle.** **Ajout du modèle entité-association. Ajout de quelques opérations SQL.**

Passage 2e année → 1re année

Fonctions récursives	Version récursive d'algorithmes dichotomiques. Fonctions produisant à l'aide de print successifs des figures alphanumériques. Dessins de fractales. Énumération des sous-listes ou des permutations d'une liste.On peut montrer le phénomène de dépassement de la taille de la pile
Tris	Tri à bulles. Notion de complexité quadratique. On propose des outils pour valider la correction de l'algorithme. Algorithmes quadratiques : tri par insertion, par sélection. Tri par partition-fusion. Tri rapide. Tri par comptage. On fait observer différentes caractéristiques (par exemple, stable ou non, en place ou non, comparatif ou non, etc).

Ingénierie numérique

A quitté le programme d'informatique, mais est passé dans les cours de physique, chimie et SII, et peut donc être posé aux écrits de ces matières et dans l'épreuve de modélisation (CCINP).

Méthodes des rectangles et des trapèzes pour le calcul approché d'une intégrale sur un segment.	Méthode des rectangles est passé dans les programmes de physique, chimie, et SII. **Méthode des trapèzes : a disparu.**
Méthode de dichotomie et Newton.	Ajout : utiliser la fonction bisect de la bibliothèque scipy.optimize (sa spécification étant fournie).
Méthode d'Euler	**Ajout :** Transformer une équation différentielle d'ordre n en un système différentiel de n équations d'ordre + utilisation de la fonction `odeint` ou `solve_ivp`
Pivot de Gauss	**A disparu.**

I Écriture d'un programme : Quelques ajouts

I.1 Instructions

Définitions

- Une **expression** est utilisée pour calculer une valeur ou pour appeler une procédure (une fonction qui renvoie la valeur `None`).
  Une expression est composée de noms, de nombres, de chaînes de caractères, d'éléments séparés par des signes de ponctuation, entourés de parenthèses, de crochets, d'accolades et d'opérateurs. Une expression a une valeur.
- Une **instruction** est un morceau de code minimal qui produit un effet. Une instruction est exécutée par une machine.
- Une **affectation** est le fait de donner une valeur à une variable.
  En Python, une expression et une affectation sont deux instructions particulières.

Exemple 1.
`3*2+5` ; `3==2` ; `"Merci"+"beaucoup"` sont des expressions.
L'affectation d'une variable, l'affichage d'une variable sont des instructions.
Le résultat d'une expression peut être affecté à une variable au cours d'une instruction.
L'affectation d'une variable, l'affirmation avec `assert`, le renvoi d'une fonction avec `return`, l'arrêt avec `break` ou encore l'important avec `import` sont des instructions.

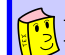
Exceptée les expressions, une instruction n'a pas de valeur.

Certaines instructions nécessitent une expression, comme avec le signe `=`, les mots assert, if, elif, while, qui sont suivis d'une expression, donc d'une valeur. Ces choix de conception du langage Python ont des conséquents à connaître.

Ces choix permettent d'éviter quelques erreurs de programmation qui pourraient passer inaperçues en première lecture. Si par exemple on écrit `assert x=1` ou `while x=0` ou `if (assert x > 0)`ou `return  x=0`, cela provoque une erreur de syntaxe et le programme s'arrête.
si par contre une instruction comme `x=1` avait une valeur, et si on écrivait `while x=1` par erreur à la place `while x==1`, on pourrait obtenir une boucle infinie (dans ce cas on finirait par s'en rendre compte). Mais si on écrivait `if (x=1)` au lieu de `if (x==1)`, l'exécution pourrait se poursuivre « normalement ».

Ces choix donnent également de la liberté dans l'écriture et permettent des raccourcis. Les mots `if, elif, while, asse` doivent être suivis d'une expression. Cette expression n'a pas forcément une valeur booléenne, mais quelle que soit sa valeur, celle-ci est interprétée comme une valeur booléenne.
Tout nombre nul et tout conteneur vide (`0 , 0.0 , "" , [] , () " {}`) est interprété comme une valeur `False`, toute autre valeur est interprétée comme une valeur `True`.
Par exemple on peut écrire `if x%2` plutôt que `if x%2 == 1` pour tester si un nombre `x` est impair, ou `while x` plutôt que `while x != 0` pour poursuivre une boucle tant que `x` n'est pas nul.

I.2 Effet de bord



Définition

On dit qu'une fonction a un **effet de bord** si son exécution modifie quelque chose en dehors de ce qui est défini dans le corps de la fonction, par exemple un de ses paramètres ou une variable globale définie dans le programme.

C'est un effet secondaire qui peut être désiré ou pas.

Exemple 2.

```python
def ajoute1(liste,x):
    liste.append(x)
def ajoute2(liste,x):
    liste = liste + [x]
    return liste
```

La fonction `ajoute1` présente un effet de bord : avec la première fonction, la liste passée en paramètre est modifiée par la méthode `append`. La fonction `ajoute2` ne présente pas d'effet de bord : l'affectation crée

une nouvelle liste (qui est une variable locale). Les deux fonctions peuvent être écrites et utilisées, mais en connaissance de cause pour éviter d'éventuel bug.

I.3 Spécification

Définitions

- Établir la **signature** consiste à :
  - la nommer avec un nom explicite ;
  - préciser ses arguments et leurs types ;
  - préciser le type des valeurs renvoyées par la fonction
- **Spécifier** un algorithme consiste à préciser sa signature et à lui ajouter :
  - les **pré-conditions** que ses arguments doivent satisfaire ;
  - les **post-conditions** portant sur ces valeurs.

  Elle est résumée dans la *docstring*, inscrite au début du corps de la fonction entre des triples guillemets.
  La fonction `help(nom_de_la_fonction)` affiche la docstring inscrite dans le code de la fonction `nom_de_la_fonction`

Exemple 3.

```
def permute(liste):
  """
  La fonction permute le premier et le dernier élément de liste
  Parameters
  ----------
  liste : TYPE list
    liste dont on souhaite inverser le premier et le dernier élément
  Returns
  -------
  copie : TYPE list
    liste identique à liste avec échange du premier et du dernier élément
  """
  copie=liste[:] # copie superficielle de liste
  copie[0],copie[-1]=copie[-1],copie[0]
  return copie
```

Les signatures et spécifications des fonctions devront toujours être précisées.

I.4 Annotations et commentaires

Un programme doit pouvoir être lu facilement par l'auteur mais aussi par quelqu'un qui découvre le programme. Il est important pour cela d'annoter certaines lignes de code ou des blocs d'instructions afin de préciser leur rôle.

Un choix de structure (liste ou dictionnaire par exemple) ou de méthode (itérative plutôt que récursive... par exemple) peut aussi être expliqué à l'aide d'un commentaire.

I.5 Assertion

Une spécification permet d'éclairer sur les données en entrées, le type des valeurs autorisées, la plage de valeurs acceptées, ... On peut ajouter des instructions qui vont arrêter le programme en cas de mauvaise utilisation.

Définition

Une **assertion** est l'affirmation qu'une propriété est vraie.
Elle est composée du mot `assert` suivi d'une expression dont la valeur est interprétée comme une valeur booléenne.
Si l'expression a la valeur `True` il ne se passe rien. Sinon le programme est interrompu et un message d'erreur s'affiche `AssertionError`.

Exemple 4.

```
def inverse(x):
    """
    Paramètre : x nombre non nul, de type int ou float
    Renvoi : inverse de x
    """
    assert x!=0
    return 1/x
>>> inverse(-3)
-0.3333333333
>>> inverse(0)
Traceback (most recent call last):
  File "<ipython-input-31-7538d73c586c>", line 1, in <module>
    inverse(0)
  File "C:\Users\sanstitre1.py", line 32, in inverse
AssertionError

def dichotomie(f,a,b,eps):
    """
    Paramètres : f fonction dont on cherche l'annulation sur l'intervalle [
   a,b] avec la précision eps
    a,b : bornes de l'intervalle, avec a<b, flottants
    eps : précision, flottant positif
    Renvoi : valeur approchée de x tel que f(x)=0
    """
    assert a<b
    assert f(a)*f(b)<0
    ...

def fonction(dico,k):
    """
    Paramètres : dico dictionnaire
    k une clé du dictionnaire
    Renvoi : ...
    """
    assert type(dico)==dict
    assert k in dico
    ...
```

I.6 Jeux de tests

Définition

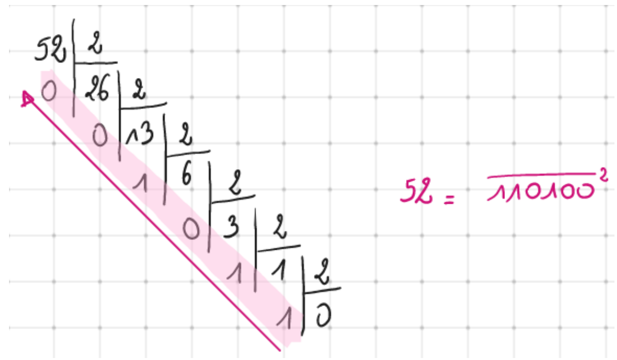
Un test est la donnée de valeurs particulières pour des paramètres d'entrée de la fonction, et la valeur attendue en retour.

L'objectif d'un jeu de tests est de valider le bon comportement de la fonction, notamment sur des cas limites. Cela ne permet pas de prouver que le programme est correct (cf § suivant), mais de trouver des cas où il pourrait ne pas fonctionner pour le corriger.

II Représentation des entiers signés en complément à deux

II.1 Rappel (très rapide) : Entiers naturels

La représentation en base deux d'une entier naturel est constitué des restes obtenus dans les divisions euclidiennes successives par 2 jusqu'à obtenir un quotient nul. On obtient l'écriture en base deux en lisant les restes de bas en haut.



L'écriture binaire $\overline{101101}^2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$

Sur n bits, on peut coder 2^n entiers, donc seuls les entiers naturels compris entre 0 et $2^n - 1$ peuvent être codés.

II.2 Représentation des entiers signés en complément à deux [Nouveauté]

À retenir

- La représentation en machine d'un **entier négatif** r, est l'**écriture binaire de la différence** $2^n - (-r)$. Cette représentation s'appelle le complément à 2^n, souvent abrégé en **complément à deux**.
- Le bit de poids fort d'un entier positif est toujours 0.
  Le bit de poids fort d'un entier négatif est toujours 1.
- Avec n bits, on peut représenter les nombres compris entre -2^{n-1} et $2^{n-1} - 1$.

Pratiquement, pour trouver la représentation d'un entier négatif r, on prend l'écriture binaire de $-r$ (qui est donc un entier positif), on inverse les bits de cette écriture et on ajoute 1. Inverser les bits et ajouter 1 sont des opérations simples pour la machine. Dans toutes ces écritures, la taille des mots, c'est-à-dire le nombre de bits utilisés, est fixée.

Exemple 5. Avec une taille de 6 bits, pour obtenir le codage de -12 :

- On code 12 en binaire sur 6 bits : 001100
- On inverse chaque bit : 110011
- On ajoute 1 : 110100, et on obtient la représentation de -12 sur 6 bits.

Ou on peut également calculer la représentation binaire de $2^6 - 12 = 52$ qui est 110100 (ouf !).

Quels entiers signés peut-on coder sur n bits ?

Avec un octet, soit huit bits, on peut écrire les entiers naturels n entre 0 et $2^8 - 1 = 255$, et donc représenter les entiers relatifs r entre $-2^7 = -128$ et $2^7 - 1 = 127$.

Les nombres n de 0 à 127 (de 0000 0000 à 0111 1111 en base deux) servent à représenter les entiers relatifs positifs ou nul r avec $n = r$, et les nombres n de 128 à 255 (1000 0000 à 1111 1111 en base deux) représentent les entiers relatifs négatifs r avec $n = 2^8 + r$ (c'est-à-dire $n = 2^8 - (-r)$).

Exercice 1

On utilise des mots de cinq bits pour coder les entiers relatifs en complément à deux.

Q1. Comment sont codés les nombres 0 ? 2 ? -2 ? -14 ? 12 ? -16 ?

Q2. Peut-on coder -32 ? 32 ?

Q3. Quels nombres sont codés par 11010 ? 01010 ? 01111 ? 11110 ?

Exercice 2 Codage d'un entier naturel

Q1. Écrire une fonction `convert_binaire(x)` qui prend en paramètre un entier naturel `x` en base dix, et renvoie l'écriture en base deux de ce nombre sous la forme d'une liste.

Par exemple :

```
>>> convert_binaire(8)
[1,0,0,0]
>>> convert_binaire(18)
[1,0,0,1,0]
```

Q2. Écrire une fonction `convert_bin_nbits(x,n)`qui prend en paramètre un entier naturel `x` en base dix, et renvoie l'écriture en base deux de ce nombre sur `n` bits sous la forme d'une liste.

On complètera, si besoin, par des 0 pour que la liste ait la bonne longueur.

On ajoutera une assertion pour vérifier que x est codable sur n bits.

Par exemple, sur 5 bits :

```
>>> convert_bin_nbits(8,5)
[0, 1, 0, 0, 0]
>>> convert_bin_nbits(18,5)
[1, 0, 0, 1, 0]
>>> convert_bin_nbits(32,5)
[1, 0, 0, 0, 0, 0]
>>> convert_bin_nbits(33,5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\nadin\OneDrive\Documents\Enseignements\
   PCPSI_Informatique_prog2022\pour les 52\pour52.py", line 16, in
   convert_bin_nbits
    assert x<=2**n , "non codable"
AssertionError: non codable
```

Exercice 3 Codage d'un entier relatif par complément à 2

Écrire la fonction `compl_a2(x,n)` qui convertit l'entier relatif `x` sur `n` bits.
On traitera bien évidemment les cas $x \geq 0$ et $x < 0$, on gèrera le cas où x n'est pas codable, et on veillera à renvoyer une liste de n éléments.
On pourra avantageusement utiliser une fonction définie dans l'exercice précédent.

III Dictionnaires [Nouveauté]

III.1 Dictionnaires

Les dictionnaires correspondent aux objets de type `dict` en Python. Ils forment une structure de données alternative aux listes. Ils permettent d'associer à un premier objet (appelé clef) un second objet (appelé valeur). Un tel couple clef/valeur est un élément du dictionnaire.

Le nombre d'éléments stockés est accessible par `len(dictionnaire)`.

III.1.a) Création d'un dictionnaire

Les dictionnaires sont définis par des accolades et on met dans l'ordre la clé et la valeur correspondante séparés par un double point : `dico={clé1:valeur1,clé2:valeur2,}`

```
1 >>> dico = {} # création d'un dictionnaire vide
2 >>> dico = {'tomates' : 3.5 , 'concombre' : 2} # création d'un dictionnaire de
    deux éléments
3 >>> dico
4 {'tomates' : 3.5 , 'concombre' : 2}
5 >>> dico['patates']=1.7 # la clé 'patates' n'existe pas, l'élément est ajouté
6 >>> dico
7 {'tomates' : 3.5 , 'concombre' : 2 , 'patates' : 1.7}
```

Les clés d'un dictionnaire ne sont pas forcément du même type. Nous pouvons utiliser des clés entières (type `int`), flottantes (type `float`), des chaînes de caractères (type `str`), ou bien des n-uplets (type `tuple`). tout objet non mutable peut être une clef valide.

```
1 >>> dico2={3 : [] , 'cle' : 42 , (15,15) : [1,2,3,4]}
```

On peut également créer un dictionnaire en compréhension comme pour les listes :

```
1 >>> dico3={x:x**2 for x in range(1,6)}
2 >>> dico3
3 {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

III.1.b) Manipulations d'un dictionnaire

i- Accès à un élément

On accède aussi à un élément avec la même notation entre crochet que pour les listes, mais en donnant cette fois la clef d'accès en argument : `dictionnaire[clé]`

```
1 >>> dico['tomates'] # accès à la valeur de la clé 'tomates'
2 3.5
3 >>> dico['aubergines'] # erreur quand on demande l'accès à une clé non présente
    dans le dictionnaire
4 Traceback (most recent call last):
5   File "<console>", line 1, in <module>
6 KeyError: 'aubergines'
7
8 >>> dico['tomates'] = 4 # la clé 'tomates' existe déjà, sa valeur est modifiée
9 >>> dico
10 {'tomates' : 4 , 'concombre' : 2 , 'patates' : 1.7}
```

ii- Test d'appartenance

Pour tester si une clé est présente dans le dictionnaire, on utilise `clé in dictionnaire` qui renvoie un booléen : `True` si la clé est présente, `False` sinon.

```
1 >>> 'tomates' in dico
2 True
3 >>> 'aubergines' in dico
4 False
```

iii- Parcours

Pour parcourir les clés d'un dictionnaire avec une boucle `for`, on utilise la syntaxe : `for cle in dictionnaire :`.

Par exemple, avec le dictionnaire précédent, on veut construire une liste `L` contenant les valeurs du dictionnaire `dico` :

```
L=[]
for cle in dico :
    L.append(dico[cle])
```

On peut aussi le faire en définissant la liste par compréhension.

```
L= [ dico[c] for c in dico]
```

On peut accéder à l'ensemble des clés à l'aide de la méthode `keys()`, et à la liste des couples `clé:valeur` par la méthode `items()`

```
>>> cles=dico.keys()
>>> cles
dict_keys(['tomates', 'concombre', 'patates'])
>>> couples=dico.items()
>>> couples
dict_items([('tomates', 3.5), ('concombre', 2), ('patates', 1.7)])
```

On obtient à chaque fois un objet de type `dict_keys` et `dict_items` respectivement, qui se comporteront de manière presque similaire à un objet de type list (on peut le parcourir, mais toutefois pas accéder à un élément par son indice).

III.1.c) Copies

Si une valeur est d'un type mutable, on peut alors la modifier sans réaffectation. Tous les exemples précédents montrent bien que les dictionnaires sont des objets mutables en Python. Se poseront donc les mêmes problèmes d'alias et de copie que pour les .

On pourra utiliser la méthode `copy()` pour réaliser une copie superficielle d'un dictionnaire, ou bien la fonction `deepcopy` de la bibliothèque `copy` pour réaliser une copie profonde de ce dictionnaire.

Commenter les scripts suivants.

```
>>> d={0:[]}
>>> e=d
>>> d[1]=""
>>> d[0].append(42)
>>> d
{0: [42], 1: ''}
>>> e
{0: [42], 1: ''}
```

```
>>> d={0:[]}
>>> e=d.copy()
>>> d[1]=""
>>> d[0].append(42)
>>> d
{0: [42], 1: ''}
>>> e
{0: [42]}
```

```
>>> from copy import deepcopy
>>> d={0:[]}
>>> e=deepcopy(d)
>>> d[1]=""
>>> d[0].append(42)
>>> d
{0: [42], 1: ''}
>>> e
{0: []}
```

III.2 Exercices

Exercice 4 Températures

On dispose des températures à Montélimar à 8h00 dans un dictionnaire :

```
temp={'J1':-10,'J2':-9,'J3':-4,'J4':0,'J5':-1,'J6':4,'J7':-5,'J8':1,'J9':-2}
```

Q1. 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.

Q2. 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 à une certaine température `T0`.

Exercice 5 Un dictionnaire

On donne le dictionnaire suivant contenant les notes des élèves sur 5 :
`notes_sur_5={'Eleve1':4,'Eleve2':3,'Eleve3':5,'Eleve4':2,'Eleve5':1,'Eleve6':5,'Eleve7':0}`
Écrire une suite d'instructions permettant de créer le dictionnaire `notes_sur_20` avec les mêmes clés que `notes_sur_5` mais où chaque note a été multipliée par ce qu'il faut pour faire une note sur 20.

Exercice 6 Préparation d'une soupe

Les stocks des ingrédients nécessaires à la réalisation d'une soupe indispensable pour affronter les premiers froids commencent à se vider et on vous charge d'aller acheter une certaine quantité de chaque ingrédient, afin de pouvoir continuer la production pendant le prochain mois. Le comptable étant particulièrement pointilleux, il vous donnera exactement la quantité d'argent dont vous avez besoin, pas une pièce de plus. Heureusement vous savez à l'avance le prix de chaque ingrédient et la quantité dont vous avez besoin.

Q1. Écrivez une fonction `prix_total` qui prend en argument deux listes (`prix_au_kg` et `masse_voulue`) qui représentent respectivement les prix au kilogramme de chaque ingrédient et la masse (en kg) nécessaire à la fabrication de la soupe, rangés bien sûr dans le même ordre pour qu'ils se correspondent.

Elle doit renvoyer la quantité totale d'argent à demander au comptable.

Q2. On veut résoudre le même problème, mais cette fois-ci `prix_au_kg` et `masse_voulue` sont des dictionnaires qui n'ont pas forcément le même nombre de clefs (`prix_au_kg` contient l'ensemble des articles proposés à la vente alors que les clefs de `masse_voulue` concernent seulement les articles nécessaires à la fabrication de la soupe).

Écrire la fonction `prix_total_avec_dico` qui permette de renvoyer la quantité totale d'argent à demander au comptable.

Exercice 7 Quelle heure est-il ?

Un site de voyage permet de calculer les temps de parcours entre deux villes sous forme d'un dictionnaire qui contient 4 clefs (`'jours', 'heures', 'minutes' et 'secondes'`) dont les valeurs associées représentent respectivement les durées en jours, heures, minutes et secondes pour le voyage, le tout de manière unique, c'est-à-dire que 27 heures vaut en fait 1 jour et 3 heures. Néanmoins, si vous faites plusieurs escales, vous voudriez bien connaître la durée totale que vous aurez passée dans les transports.

On va décomposer ce problème en plusieurs sous-problèmes plus simples à résoudre.

Q1. Écrire une fonction `decomposition(duree)` qui prend en argument une durée exprimée en secondes et renvoie un dictionnaire à valeurs entières dont les clefs sont `'jours', 'heures', 'minutes', 'secondes'`.

Q2. Écrire la fonction inverse `secondes(dico)` qui prend en argument un dictionnaire du type précédent pour renvoyer la valeur correspondante en secondes.

Q3. En utilisant les deux fonctions précédentes, écrire la fonction `addition(dico1,dico2)` qui va additionner correctement deux dictionnaire `dico1` et `dico2` correspondant à deux voyages successifs et renvoyer le dictionnaire du même type correspondant à la durée totale du voyage.

Q4. Écrire une fonction `affichage(dico)` qui affiche (à l'aide de `print`) le temps total passé en transport de manière un peu plus lisible pour le commun des mortels.

Par exemple, l'appel à la fonction renvoie :

```
>>> affichage({'jours': 3, 'heures': 22, 'minutes': 10, 'secondes': 54})|
Vous allez voyager un total de 3 jours, 22 heures, 10 minutes et 54 secondes
```

Q5. Enfin, écrire une fonction `temps_total(liste_de_durees)` qui prend en argument une liste (de taille arbitraire) de durées sous la forme des dictionnaires précédents et renvoie le temps total de parcours à l'aide de la fonction `affichage(dico)` précédente.

IV Algorithme glouton [Nouveauté]

Capacité exigible : Rendu de monnaie. Allocation de salles pour des cours. Sélection d'activité. On peut montrer par des exemples qu'un algorithme glouton ne fournit pas toujours une solution exacte ou optimale.

Objectifs :

— Comprendre le principe d'un algorithme glouton.

— Reconnaître une stratégie gloutonne dans un programme.

— Être capable d'écrire un algorithme glouton pour résoudre un problème.

— Savoir utiliser un algorithme glouton dans des situations diverses.

IV.1 Principe

Les **algorithmes gloutons** suivent une stratégie simple : **lorsqu'à chaque étape, un choix doit être fait, c'est le choix optimal à ce moment qui est fait.**

Un problème d'optimisation a deux caractéristiques : une fonction que l'on doit maximiser ou minimiser et une série de contraintes auxquelles il faut satisfaire. On peut essayer de résoudre ce type de problème en écrivant un algorithme qui énumère toutes les possibilités afin de trouver la meilleure. C'est un algorithme très simple mais souvent inutilisable à cause du coût. L'objectif d'un algorithme glouton est d'obtenir une solution rapidement, mais qui ne sera pas toujours la solution optimale.

À chaque étape exécutée par un algorithme, se présente un ensemble de choix et un algorithme glouton fait le meilleur choix parmi les propositions. **Un choix glouton est donc un choix localement optimal**. **La question est de savoir si en faisant une série de choix localement optimaux, on finit par aboutir à une solution optimale. C'est parfois le cas mais pas toujours.**

IV.2 Problème du rendu de monnaie

Prenons l'exemple classique du rendu de monnaie. On considère le système de monnaie de la zone euro, et nous avons les pièces suivantes (en centimes) : $S = (1, 2, 5, 10, 20, 50, 100, 200)$.

Le problème du rendu de monnaie consiste à trouver le nombre de pièces de chaque type à rendre pour arriver à la somme à rendre, en minimisant le nombre de pièces utilisées.

La contrainte est le fait que le montant total des pièces rendues est égal à la somme à rendre.

On cherche, par valeur décroissante en partant de la pièce qui a la plus forte valeur, la première pièce qui a une valeur inférieure ou égale à la somme à rendre r. On prend cette pièce, on retranche sa valeur v à r. On recommence en partant de la pièce prise en cherchant cella qui a une valeur inférieure ou égale à la nouvelle somme à rendre $r - v$. On prend cette pièce, et ainsi de suite jusqu'à arriver à une somme à rendre nulle.

Exercice 8 Mettre en œuvre l'algorithme à la main sur 8 centimes, 7 € ,12 €.

On écrit la fonction `monnaie(p,r)` en Python :

```
def monnaie(p,r):
    """
    Arguments :
    p : liste des pièces du système de monnaie
    r : somme à rendre
    Retour :
    sol : liste du nombre de pièces à rendre pour chaque pièce du système de
  monnaie
    """
    assert type(p)==list and type(r)==int
    assert r>=0
    n=len(p) # nbre de pièces
    i=n-1 # on part de la pièce la plus grosse
    sol=n*[0] # initialisation du nbre de pièces dans chaque type
    while r>0: # tant que la somme à rendre n'est pas nulle
        while p[i]>r:
            i=i-1 # tant que la pièce testée est plus grande que le montant à
  rendre, on passe à la suivante (plus petite)
        sol[i]=sol[i]+1 # on ajoute une pièce
        r=r-p[i] # on enlève le montant de la pièce ajoutée à la somme à rendre
    return sol
>>> S=[1,2,5,10,20,50,100,200]
```

```
>>> monnaie(S,8) # rendu de 8 centimes
[1,1,1,0,0,0,0,0,0]
>>> monnaie(S,700) # rendu de 7 E
[0, 0, 0, 0, 0, 0, 1, 3]
>>> monnaie(S,1200) # rendu de 12 E
[0, 0, 0, 0, 0, 0, 0, 6]
```

Dans presque tous les systèmes de monnaie, l'algorithme glouton est optimal. Pour rendre la monnaie, on rend la pièce (ou le billet) de valeur maximale, et on continue tant qu'il reste quelque chose à rendre.

IV.3 Exercices

Exercice 9 Rendu de monnaie (2)

La fonction `monnaie` écrite précédemment renvoie une liste pour représenter le nombre de pièces de chaque type à rendre.

Q1. Écrire une fonction qui renvoie un dictionnaire dont les clés sont les valeurs des pièces rendues et les valeurs associées aux clés sont le nombre de pièces correspondantes.

Q2. Tester la fonction avec le système de monnaie $(1, 2, 5, 10, 20, 50, 100)$ € sur 48 €.

Exercice 10 Rendu de monnaie (3)

Dans un système monétaire, on dispose de pièces de valeurs 1, 3, 4, 5, 8, 10 unités monétaires (um).

Q1. Quelle est la solution fournie par l'algorithme glouton ?

Q2. L'algorithme glouton fournit-il une solution optimale pour cette situation ?

Exercice 11 Rendu de monnaie (4)

On considère les valeurs des pièces et billets en euros `p=[1,2,5,10,20,50,100,200,500]`.
On représente par une liste `C` le nombre de chaque pièces/billets qu'un commerçant dispose dans sa caisse (par exemple : `[32,37,17,12,13,5,1,0,0]` signifie qu'il a 32 pièces de 1 €, 37 pièces de 2 €, etc).

Q1. Écrire une fonction `rendu_caisse(p,C,r)` qui détermine la façon de rendre la somme `r`, avec qu'il y a de disponible dans la caisse. On renverra une liste avec la quantité de chaque pièce/billet qu'il devra rendre.

Q2. Effectuer le rendu de 200€ avec la liste `C` citée plus haut.

Exercice 12 Station d'essence

Vous partez en vacances en voiture, avec le plein de carburant, et devez parcourir un long trajet. Votre véhicule peut parcourir une distance maximale $d_{\max}$ avec un plein.
La route que vous empruntez comporte n stations services, $s_0, s_1, ...s_{n-1}$, rangées dans l'ordre rencontré en suivant le parcours. La première est à une distance d_0 du départ, la deuxième à une distance d_1 de la première, ... Le point d'arrivée est à une distance d_n de la dernière station.
Nous supposons que les distances entre le départ et la première station, puis entre chaque station, et entre la dernière station et l'arrivée sont toutes inférieures à la distance $d_{\max}$ maximale. C'est une condition nécessaire et suffisante pour que l'automobiliste puisse effectuer le parcours.
Votre objectif est de vous arrêter pour faire le plein le moins souvent possible.
Un algorithme glouton consiste chaque fois que le plein est fait à parcourir le plus long trajet possible sans refaire le plein. Le plein est donc fait chaque fois à la dernière station avant de tomber en panne.

Q1. Écrire une fonction `trajet(d,dmax)` qui prend en argument deux listes : `d` des distances entre le départ et la première station, puis entre chaque station, et entre la dernière station et l'arrivée, et la distance maximale que peut parcourir la voiture avec le plein et qui renvoie la liste `stations` des stations auxquelles vous devez vous arrêter.

Q2. Tester la fonction avec la liste des distances suivantes : `distances=[58, 176, 176, 61, 123, 131, 181, 75, 105, 175, 186, 174, 120, 172, 199, 113, 75]` pour un trajet de 2300 km (somme des valeurs de la liste `distances`) avec une distance maximale de 600 km.

V Graphes [Nouveauté]

Vocabulaire des **graphes.**	Graphe orienté, graphe non orienté. Sommet (ou nœud) ; arc, arête. Boucle. Degré (entrant et sortant). Chemin d'un sommet à un autre. Cycle. Connexité dans les graphes non orientés. On présente l'implémentation des graphes à l'aide de listes d'adjacence (rassemblées par exemple dans une liste ou dans un dictionnaire) et de matrice d'adjacence.
Notations.	Graphe $G = (S, A)$, degrés $d(s)$ (pour un graphe non orienté), $d_+(s)$ et $d_-(s)$ (pour un graphe orienté).
Pondération d'un graphe. Étiquettes des arcs ou des arêtes d'un graphe.	On motive l'ajout d'information à un graphe par des exemples concrets.
Parcours d'un graphe.	On introduit à cette occasion les **piles et les files** ; on souligne les problèmes d'efficacité posés par l'implémentation des files par les listes de Python et l'avantage d'utiliser un module dédié tel que collections.deque. Détection de la présence de cycles ou de la connexité d'un graphe non orienté.
Recherche d'un plus court chemin dans un graphe pondéré avec des poids positifs.	Algorithme de Dijkstra. On peut se contenter d'un modèle de file de priorité naïf pour extraire l'élément minimum d'une collection. Sur des exemples, on s'appuie sur l'algorithme A* vu comme variante de celui de Dijkstra pour une première sensibilisation à la notion d'heuristique.

V.1 Définitions

Définitions : Graphe non orienté

- Un **graphe non orienté** est un couple $G = (S, A)$ dans lequel S est un ensemble non vide (dont les éléments sont appelés **sommets**) et A un ensemble de parties à deux éléments de S (dont les éléments sont appelés arêtes du **graphe**).
- Deux sommets sont **adjacents** ssi il existe une arête qui les relie.
- Le **degré** d'un sommet est égal au nombre d'arêtes dont il est extrémité. On le notera $d(s_i)$.
- Un graphe est **connexe** si deux sommets distincts sont toujours reliés par un chemin.
- L'**ordre** d'un graphe est le nombre de sommets.

Définitions : Chemin

- Un chemin d'un sommet s_0 à un sommet s_n est une séquence $(s_0, s_1, ..., s_n)$ où deux sommets consécutifs sont adjacents.
- La longueur d'un chemin est le nombre d'arêtes utilisées pour aller du sommet de départ au sommet d'arrivée.
- La distance entre deux sommets est la longueur minimale d'un chemin reliant ces deux sommets.
- Un cycle est un chemin tel que le sommet de départ et le sommet d'arrivée sont identiques.

Définitions : Graphe orienté

Un **graphe orienté** est un couple d'ensembles (S, A) avec $A \subset S^2$. Les éléments de A que l'on appelle alors des **arcs** sont des couples (s_i, s_j) de sommets. On pourra aussi les notes $s_i \to s_j$ et parler d'extrémités initiale et terminale. Les arêtes ont un sens de parcours.
On appelle **degré sortant** d'un sommet s_i le nombres d'arcs $(s_i, s_j) \in A$. Ce degré sortant est noté $d_+(s_i)$. Le degré entrant est défini de façon analogue et est noté $d_-(s_i)$.

Définitions : Graphe pondéré

Ajouter des poids ou des étiquettes aux arêtes d'un graphe apporte des informations supplémentaires.

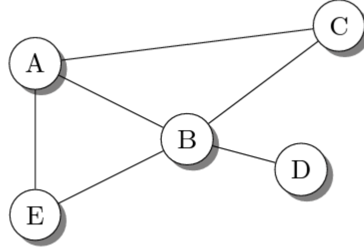
- Un graphe est **pondéré** si un nombre, un **poids**, est associé à chaque arête ou chaque arc.
- Un graphe est **étiqueté** si une **étiquette**, est associée à chaque arête ou chaque arc.

Dans un réseau routier par exemple, un poids peut être le nombre de kilomètres d'une route liant deux lieux, une étiquette peut être le nom de cette route.

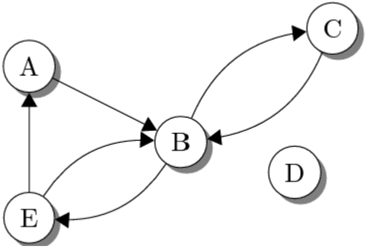
V.2 Représentation schématique

La manière la plus simple de représenter un graphe est de faire un dessin. Les sommets sont représentés par des points, les arêtes par des lignes, chacune reliant deux poids.

Exemple 6.



Ce graphe n'est pas orienté, et connexe.
Sommet A : $d(A) = 3$, ses voisins sont B, C, E.
$[ABD]$ est un chemin de longueur 2.
$[AEBD]$ est un chemin de longueur 3.
$[ACBEA]$ est un cycle



Ce graphe est orienté, et non connexe.
D est un sommet isolé.
Degré entrant de B : $d_-(B) = 3$
Degré sortant de B : $d_+(B) = 2$

V.3 Représentation et implémentations

V.3.a) Liste d'adjacence

On peut représenter un graphe non orienté en précisant pour chacun des sommets la liste de ses voisins. Ces listes s'appellent des **listes d'adjacence**. L'ordre d'écriture n'a pas d'importance.

Dans le cas de graphes orientés, on peut présenter des **listes de successeurs**.

Ces **listes d'adjacence peuvent être représentées en `Python` à l'aide** :

— **liste de listes** : chaque élément de la liste est une liste contenant un sommet et la liste de ces voisins.

```
# Pour le graphe non orienté :
G=[ ["A",["B","C","E"]] , ["B",["A","C","D","E"]] , ["C",["A","B"]] , ["D",["B"]] , ["E",["A","B"]] ]
# Pour le graphe orienté :
G =[ ["A",["B"]] , ["B",["C","E"]] , ["C",["B"]] , ["D",[]] , ["E",["A","B"]] ]
```

— **dictionnaire** : les clés sont les sommets et les valeurs correspondent aux clés sont les listes des voisins.

```
# Pour le graphe non orienté :
G = {"A":["B","C","E"] , "B":["A","C","D","E"] , "C":["A","B"] , "D":["B"] , "E":["A","B"] }
# Pour le graphe orienté :
G = {"A":["B"] , "B":["C","E"] , "C":["B"] , "D":[] , "E":["A","B"] }
```

Si le graphe est pondéré, on complète les listes d'adjacence avec les poids.

```
# Avec une liste de listes :
G=[ ["A",[("B",2),("E",5)]] , ["B",[("A",2),("C",1),("E",3)]] , ["C",[("B",1)]] , ["D",[]] , ["E",[("A",5),("B",3)]]]
# Avec un dictionnaire
G={"A":[("B",2),("E",5)] , "B":[("A",2),("C",1),("E",3)] , "C":[("B",1)] , "D":[] , "E":[("A",5),("B",3)]}
```

V.3.b) Matrice d'adjacence

En mathématiques, on peut associer à un graphe, une matrice carrée (n, n) ou un tableau, où n est le nombre de sommets. Les sommets sont numérotés de 0 à $n-1$. À l'intersection d'une ligne i et d'une colonne j le nombre représenté la présence ou non d'une arrête enter les sommets i et j : 1 pour la présence, 0 pour l'absence.

Le tableau correspondant au graphe non orienté dessiné précédemment est le suivant :

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	1	1
C	1	1	0	0	0
D	0	1	0	0	0
E	1	1	0	0	0

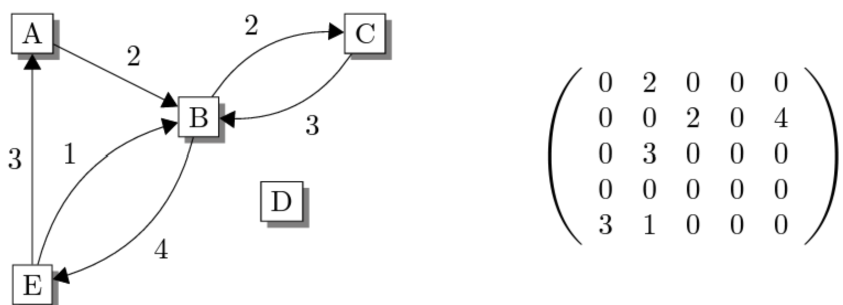
La diagonale ne contient que des 0 et est un axe de symétrie du tableau, c'est le cas pour les graphes non orientés.

Le même graphe peut être représenté par des matrices différentes qui dépendent de l'ordre des sommets qui est pris en compte.

La matrice du graphe non orienté est la suivante : $\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$

Dans le cas du graphe orienté précédent, on obtient la matrice d'adjacence suivante, qui n'est plus symétrique : $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$

Dans le cas de graphe pondéré ou étiqueté, on place les informations le long des arêtes. On peut utiliser la matrice en remplaçant les 1 par les poids par exemple.

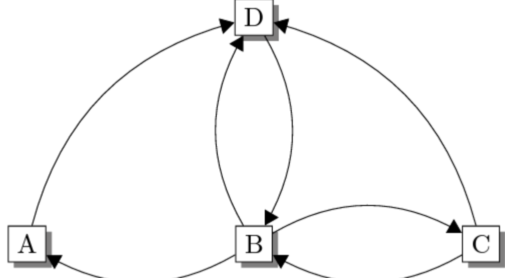


$$\begin{pmatrix} 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 4 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \end{pmatrix}$$

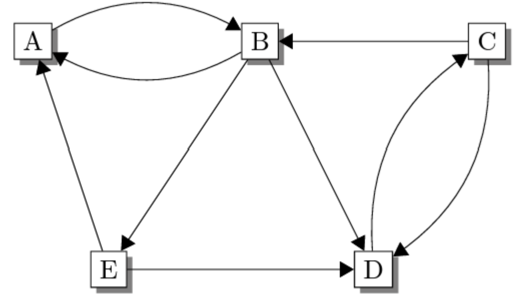
En `Python`, on pourra utiliser les **listes de listes** ou les tableaux `numpy` pour représenter une matrice d'adjacence.

V.4 Exercices

Exercice 13 Écrire la matrice d'adjacence associée au graphe ci-dessous.



Exercice 14 Écrire la liste de successeurs du graphe ci-dessous.



Exercice 15

Tracer des graphes associés aux matrices d'adjacence données.

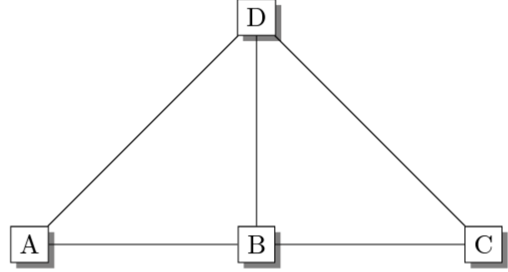
Q1. Un graphe non orienté avec $M_1 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$

Q2. Un graphe orienté avec $M_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$

Q3. Un graphe orienté avec $M_3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$

Exercice 16

On considère le graphe ci-dessous :



Q1. Écrire la définition de la liste `m` qui représente la matrice d'adjacence du graphe.

Q2. Écrire une fonction qui prend en paramètre une matrice `m` représentant un graphe et renvoie le nombre d'arêtes du graphe.

Q3. Écrire un dictionnaire en Python, où les clés sont les sommets, pour représenter ce graphe à l'aide des listes d'adjacence.

Q4. Écrire une fonction `sommets` qui prend en paramètres un sommet `s` et un graphe `g`, sous la forme d'un dictionnaire, et renvoie la liste des sommets liés par une arête au sommet `s`.

Q5. Écrire une fonction `aretes` qui prend en paramètre un graphe `g`, sous la forme d'un dictionnaire, et renvoie la liste des arêtes du graphe.

Exercice 17

On considère des graphes non orientés et non pondérés. Une variable `g` est définie pour représenter un graphe. Un sommet est représenté par un caractère comme `'A'`.

Q1. Écrire une fonction qui prend en paramètre un graphe `g` représenté par un dictionnaire des listes d'adjacence et renvoie la liste des sommets.

Q2. Écrire une fonction qui prend en paramètre un graphe `g` représenté par une liste des listes d'adjacence et renvoie la liste des sommets.

Q3. Écrire une fonction qui prend en paramètres un graphe `g` représenté par un dictionnaire des listes d'adjacence et un sommet `s` et renvoie le degré du sommet.

V.5 Piles et Files

Cf votre cours de 2e année pour les piles

Une structure linéaire sur un ensemble E est une suite d'éléments de E, chaque élément ayant une place bine précise dans cette suite qui est ordonnée. En python, une structure de **pile** et de **file** sont des structures linéaires qui peuvent être construites à l'aide de **listes** sur lesquelles on applique des **conditions d'utilisation restreintes.**

Définitions

- Une file est une structure linéaire pour laquelle on n'autorise que l'insertion d'un élément d'un côté et la suppression de l'autre, en suivant le principe « premier entré, premier sorti », anglais FIFO, acronyme de « First In First Out » .
- Une pile est une structure linéaire pour laquelle on n'autorise que l'insertion et la suppression d'un même côté, en suivant le principe « dernier entré, premier sorti », anglais LIFO, acronyme de « Last In First Out » .

Exemple 7. Les piles et les files sont des structures très présentes en informatique.
Une imprimante gère une file de documents en attente d'impression.

Les opérations de bases sur les fils sont semblables à celles codées pour les piles.

On peut également implémenté les files à l'aide d'une liste.

Le retrait du sommet de la pile se fait à l'aide de la méthode `pop` qui se fait en temps constant $\mathcal{O}(1)$. Par contre, le retrait du premier élément de la file se fait en utilisant la méthode `pop(0)` qui ne se fait pas en temps constant, mais en $\mathcal{O}(n)$ si n est la taille de la liste.

Il existe dans le module `collections` la structure `deque` (pour double-ended queue). Un objet de type `deque` a un comportement similaire à celui d'une liste mais **les ajouts et les retraits sont rapides à chaque extrémité**. On peut donc l'utiliser pour une file ou une pile.

```
from collections import deque
>>> pile=deque([1,2,3])
>>> pile.append(4) # ajout en fin de pile
>>> pile
deque([1, 2, 3, 4])
>>> pile.pop() # retrait en fin de pile
4
>>> pile
deque([1, 2, 3])

>>> file=deque([1,2,3])
>>> file.append(4)
>>> file
deque([1, 2, 3, 4]) # ajout en fin de file
>>> file.popleft() # retrait en début de file
1
>>> file
deque([2, 3, 4])
```

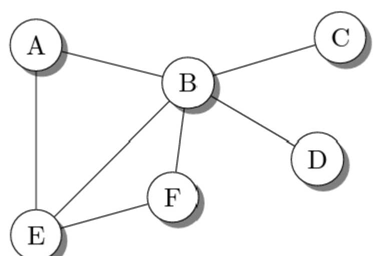
V.6 Parcours d'un graphe

V.6.a) Introduction

Nous disposons de deux types de manières de parcourir un graphe :

- **Parcours d'un graphe en profondeur** : à partir d'un sommet, on passe à un de ses voisins, puis à un voisin de ce voisin et ainsi de suite. S'il n'y a pas de voisin, on revient au sommet précédent et on passe à un autre de ses voisins.
- **Parcours d'un graphe en largeur** : à partir d'un sommet, on explore tous ses voisins immédiats. Puis à partir d'un voisin, on explore tous ses voisins immédiats sauf ceux déjà explorés. Et ainsi de suite.

Exemple 8.



— Parcours en profondeur à partir de A : On commence par un voisin de A : B (par ex., on aurait pu commencer par E), puis un voisin de B : C. C n'a pas de voisin, on remonte à B et on visite un autre voisin : D qui n'a pas de voisin. On remonte à B et on visite un autre voisin : E, puis un voisin de E : F.

— Parcours en largeur à partir de A : On commence par tous les voisins de A : B et E, puis on repart de B et on visite tous ses voisins restants : C, D et F.

V.6.b) Parcours en profondeur

i- Programme récursif

On utilise une liste `visite` pour stocker les sommets visités et une liste `attente` pour stocker les voisins d'un sommet non encore visités. On utilise un dictionnaire `marque` pour marquer les sommets visités, la clé est le sommet et sa valeur est `True`.

L'utilisation d'un dictionnaire est le plus efficace pour vérifier si un sommet a déjà été visité ou non, ce qui se fera en temps constant avec un dictionnaire, alors qu'avec une liste ce serait de complexité linéaire.

```
def parcours_prof_rec(graphe,sommet,visite,marque):
    if sommet not in marque: # le sommet n'a pas été encore visité
        visite.append(sommet) # on ajoute le sommet à la liste des sommets visités
        marque[sommet]=True # on marque le sommet en l'ajoutant dans le dictionnaire marque
    attente=[s for s in graphe[sommet] if s not in marque] # liste des voisins du sommet, qui n'ont pas été encore visités et ne sont donc pas présents dans le dictionnaire marque
    for s in attente: # pour chaque voisins de sommet
        parcours_prof_rec(graphe,s,visite,marque) # appel récursif : parcours en profondeur du graphe à partir de s
    return visite # liste des sommets visités
```

Avec le graphe défini par un dictionnaire des listes d'adjacence en exemple, on obtient, en partant de `'A'` :

```
>>> g = {'A':['B','E'],'B':['A','C','D','E','F'],'C':['B'],'D':['B'],'E':['A','B','F'],'F':['B','E']}
>>> parcours_prof_rec(g,'A',[],{})
['A', 'B', 'C', 'D', 'E', 'F']
```

ii- Programme itératif

On utilise une pile pour placer les sommets en attente.

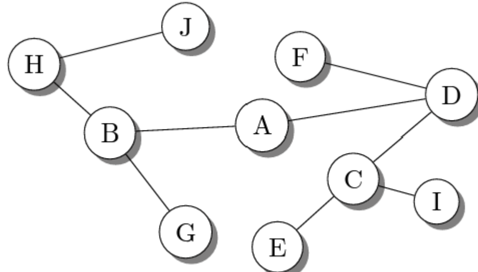
```
def parcours_prof_it(graphe,sommet):
    visite=[] # liste des voisins visités
    marque={} # dictionnaire des voisins visités
    attente=deque() # pile qui garde les sommets en attente
    attente.append(sommet) # on ajoute au sommet de la pile le sommet de départ
    while len(attente)>0: # tant que la pile n'est pas vide
        sommet=attente.pop() # on dépile le sommet de la pile
        if sommet not in marque: # si le sommet n'a pas déjà été visité
            visite.append(sommet) # on l'ajoute à la liste des sommets visités
            marque[sommet]=True # on l'ajoute au dictionnaire
            for s in graphe[sommet] : # pour les voisins de sommet
                if s not in marque: # le voisin s n'a pas déjà été visité
                    attente.append(s) # on le place dans la pile des sommets en attente d'être visité
    return visite
>>> parcours_prof_it(g,'A')
['A', 'E', 'F', 'B', 'D', 'C']
```

V.6.c) Parcours en largeur

Pour l'algorithme itératif du parcours en largeur, il suffit d'utiliser une file à la place d'une pile pour placer les sommets en attente.

```
def parcours_largeur_it(graphe,sommet):
    visite=[] # liste des voisins visités
    marque={} # dictionnaire des voisins visités
    attente=deque() # file qui garde les sommets en attente
    attente.append(sommet) # on ajoute au sommet de la file le sommet de départ
    while len(attente)>0: # tant que la file n'est pas vide
        sommet=attente.popleft() # on visite le sommet au début de la file
        if sommet not in marque: # si le sommet n'a pas déjà été visité
            visite.append(sommet) # on l'ajoute à la liste des sommets visités
            marque[sommet]=True # on l'ajoute au dictionnaire
            for s in graphe[sommet] : # pour les voisins de sommet
                if s not in marque: # le voisin s n'a pas déjà été visité
                    attente.append(s) # on le place dans la pile des sommets en
  attente d'être visité
    return visite
>>> parcours_largeur_it(g,'A')
['A', 'B', 'E', 'C', 'D', 'F']
```

Exercice 18 On considère le graphe ci-dessous :



Expliciter comment on parcourt ce graphe à partir du sommet initial A avec un parcours en largeur d'abord, puis avec un parcours en profondeur d'abord.

V.6.d) Recherche d'un cycle

Un parcours en largeur peut être utilisé pour déterminer la présence d'un cycle, c'est-à-dire d'un chemin où toutes les arêtes sont différentes, et qui se termine à son sommet de début.

Pour cela, on associe un niveau à tous les sommets, `None` pour initialiser. On commence le parcours en un sommet auquel on donne le niveau 0. Puis on visite ses voisins, auxquels on donne le niveau 1 et ainsi de suite.

Cette méthode utilise une file, qui est initialement vide, et on y place le sommet de départ choisi dont le niveau est 0. Tant que la file n'est pas vide :

- on retire l'élément en tête de file, soit S ;
- on visite ses voisins s et pour chaque voisin :
  - si son niveau est `None`, on lui donne le niveau de S augmenté de 1 et on le place à la suite dans la file ;
  - si son niveau est supérieur ou égal à celui de S, on a trouvé un cycle et on renvoie `True` ;
  - si son niveau est strictement inférieur à celui de S, on ne fait rien.

Si à la fin, la file est vide, on renvoie `False`, le graphe ne contient aucun cycle.

```
def cycle_largeur(graphe,sommet):
    niveaux={s:None for s in graphe} # initialisation niveau None pour tous les sommets du graphe
    niveaux[sommet]=0 # sommet de départ de niveau 0
    file=deque() # file vide
    file.append(sommet) # on y ajoute le sommet de départ
    while len(file)>0:
        S=file.popleft() # tête de liste
        for s in graphe[S]: # pour les voisins de S
            if niveaux[s]==None:
                niveaux[s]=niveaux[S]+1 # on augmente le niveau de s, à celui de S plus 1
                file.append(s) # on place s dans la file
            elif niveaux[s]>=>niveaux[S]: # le voisin est d'un niveau supérieur ou égal à celui de S, on a trouvé un cycle
                return True
    return False # la file n'est pas vide
```

V.7 Recherche d'un plus court chemin

On s'intéresse maintenant à un graphe pondéré avec des poids positifs. Le chemin le plus court est celui de coût, c'est-à-dire la somme des poids des arêtes, le plus faible.

V.7.a) Algorithme de Dijkstra

Cet algorithme, publié par Edsger Dijkstra en 1959, utilise un **parcours en largeur** et calcul le plus court chemin entre un sommet et chacun des autres sommets.

On suppose le graphe connexe et non orienté.

i- Principe

Si le plus court chemin entre deux sommets D et A passe par un sommet I, alors la partie de ce chemin entre D et I est le plus court chemin de D à I, et la partie entre I et A est le plus court chemin entre I et A. À chaque étape, on effectue donc le meilleur choix possible. C'est un algorithme glouton.

L'algorithme est semblable à celui d'un parcours en largeur d'abord, mais au lieu d'utiliser une file pour les sommets en attente, on utilise une fil de priorité. Cela signifie qu'on extrait le sommet ayant la priorité, dans ce cas c'est celui qui correspond à la distance minimale.

ii- Exemple

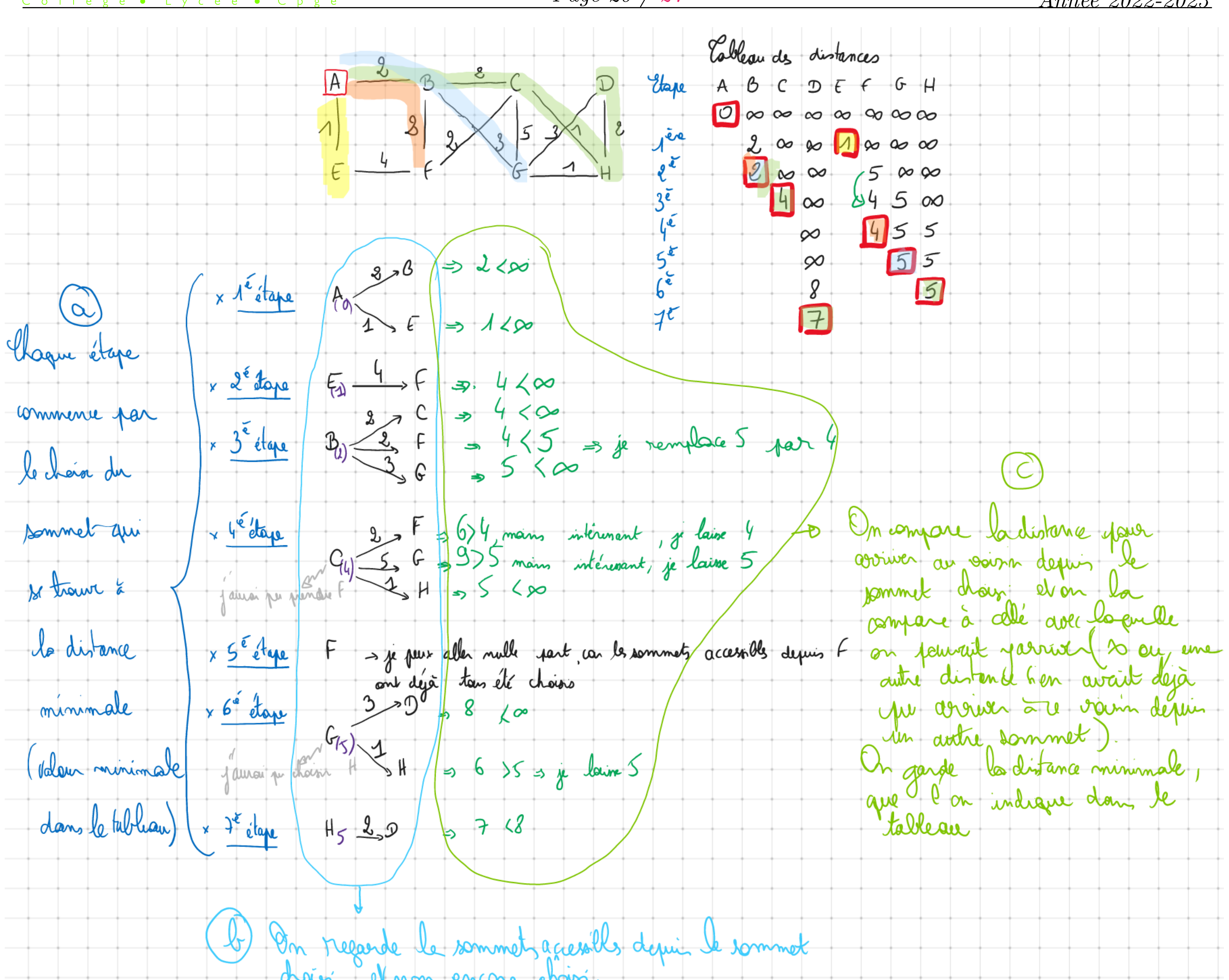
On considère le graphe représenté ci-dessous et on cherche le plus court chemin entre le sommet A et chacun des autres sommets du graphe. On affecte la valeur ∞ à chaque sommet, sauf au sommet A de départ, à qui on affecte la valeur 0.

À chaque étape :

(a) On choisit le sommet dont la distance depuis A dans le tableau est minimale.

(b) On regarde ses différents voisins encore accessibles (c'est-à-dire qui n'ont pas déjà été choisis).

(c) On compare la distance avec laquelle on arrive aux différents voisins depuis ce sommet, à la distance avec laquelle on avait pu y arriver jusque là (l'infini, ou une autre distance par un autre chemin). On garde la distance minimale avec laquelle on peut arriver à ce voisin.



iii- Implémentation en python

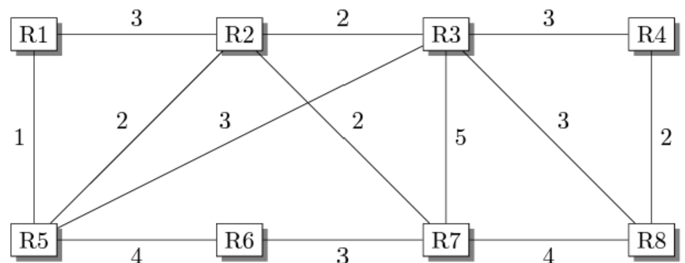
Pour représenter l'ensemble des sommets et leur distance, nous allons utiliser un dictionnaire dont les sommets sont les clés et les distances leurs valeurs.

À chaque étape, il faut comparer la distance à laquelle se trouve chaque sommet. Il faut donc commencer par écrire une fonction qui renvoie la clé de valeur minimale

```python
def minimum(dico):
    """
    dico : dictionnaire, dont au moins une valeur est différente de inf
    Renvoi :
    cle_min : la clé dont la valeur est la valeur minimale des valeurs
    """
    mini=float('inf') # initialisation de la valeur minimale
    for cle in dico :  # parcours des clés de dico
        if dico[cle]<mini: # clé de valeur <au minimum local
            mini=dico[cle] # on a trouvé un nouveau minimum local
            cle_min=cle # clé de valeur=minimum local actuel
    return cle_min
```

```
def Dijkstra(graphe,S):
    """
    graphe : dictionnaire des listes d'adjacence
    sommet : sommet de départ
    """
    res={} # dictionnaire des valeurs minimales de distance entre le sommet  S et chacun des autres sommets
    dist={cle:float('inf') for cle in graphe} # initialisation du dictionnaire des distances
    dist[S]=0 # sommet S de départ, distance=0
    while len(dist)>0: # tant que d n'est pas vide, il reste des sommets à visiter
        # (a)
        smin=minimum(dist) # sommet de distance minimale
        res[smin]=dist[smin] # on copie le sommet smin et sa distance dans le dictionnaire des résultats
        for k in range(len(graphe[smin])): # parcours des voisins de smin
            # (b)
            v,d=graphe[smin][k] # v : nom du voisin, d distance à laquelle il se trouve de smin
            if v not in res: # on ne s'intéresse qu'au voisin non déjà choisi
                # (c)
                dist[v]=min(dist[v],dist[smin]+d) # on choisit la distance minimale entre celle avec laquelle on aurait déjà pu arriver à v (dist[v]), et celle avec laquelle on arrive depuis smin (dist[smin]+d)
        res[smin]=dist[smin] # on copie le sommet smin et sa distance dans le dictionnaire des résultats
        del dist[smin] # on supprime smin de dist, qui contient les sommets qu'il reste à visiter
    return res
```

Exercice 19 Voici un réseau. On cherche les plus courtes distances entre chaque sommet et R8. Appliquer l'algorithme de Dijkstra « à la main » en construisant un tableau des distances.



iv- Complexité

L'étude précise est difficile, mais on peut tout de même identifier qu'il y a autant d'étapes que de sommet, donc n étapes. Pour chaque étape, on calcule des distances avec les autres sommets, au maximum n opérations, et on cherche le minimum parmi n sommets au maximum. La complexité est donc dans le pire des cas quadratique (en n^2).

V.7.b) Algorithme A^*

Peter E. Hart, Nils John Nilsson et Bertram Raphael ont proposé un algorithme de recherche d'un chemin nommé **algorithme** A^* qui fournit un chemin entre deux sommets donnés. C'est une extension de l'algorithme de Dijkstra.

Cet algorithme fournit l'une des meilleurs solutions rapidement. Il est utilisé en intelligence artificielle et dans des applications de jeux vidéos pour lesquels le plus important est la vitesse d'obtention d'une solution, même si elle n'est pas optimale.

L'algorithme utilise une évaluation heuristique sur chaque sommet afin de parvenir à trouver le meilleur chemin. Les sommets sont visités suivant l'ordre donné par cette évaluation.

Une méthode heuristique est une méthode de résolution utilisée pour obtenir une solution rapidement, pas forcément la meilleure, quand d'autres algorithmes ont une complexité en temps trop élevée. On n'explore pas toutes les possi-

bilités pour trouver la solution optimale, mais on les filtre à l'aide de données supplémentaires provenant de mesures, d'expériences, ou de statistiques.

Dans une recherche de distance minimale dans un graphe représentant un réseau routier, les valeurs heuristiques peuvent être les distances « à vol d'oiseau. »

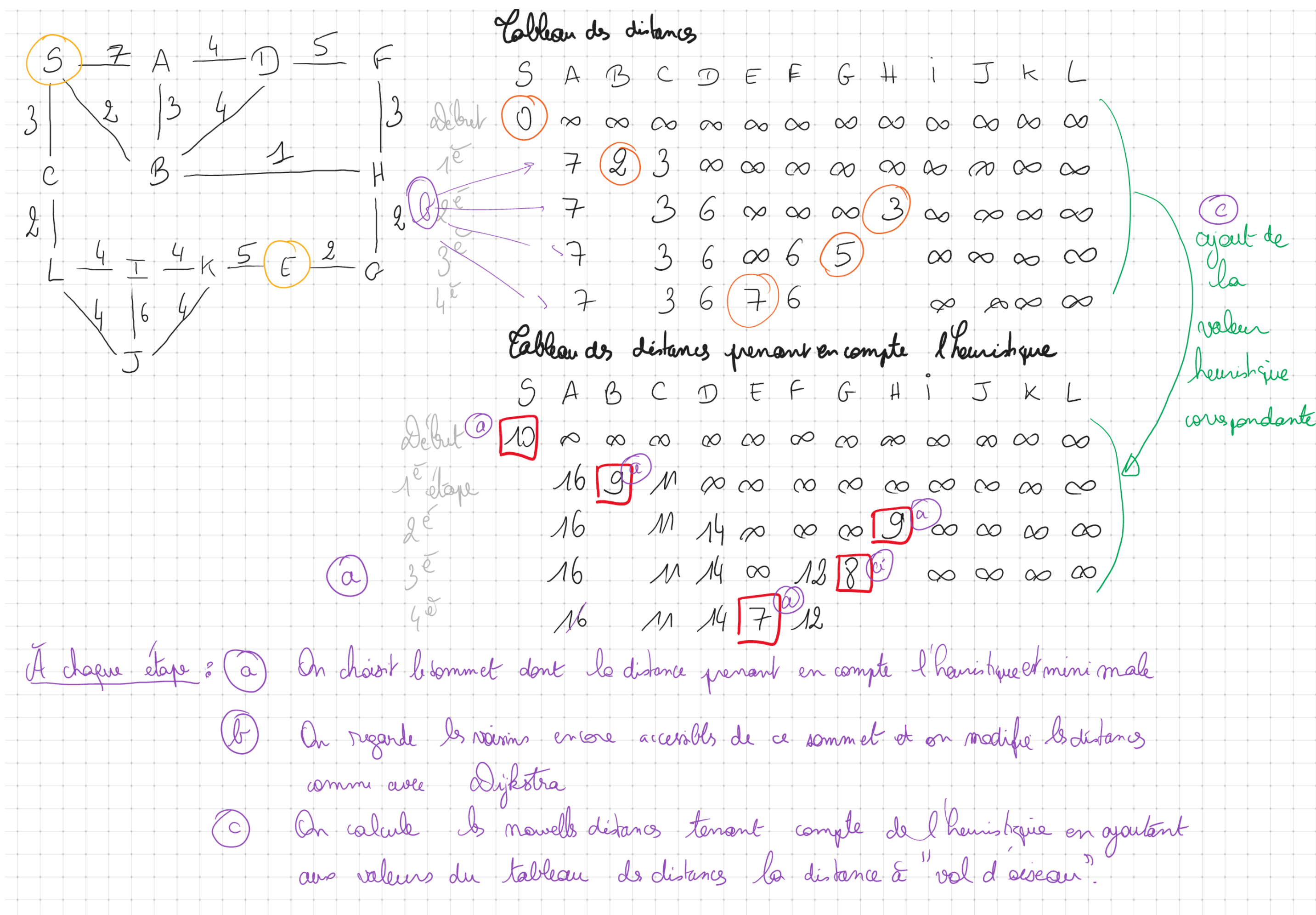
```
def A_etoile(graphe,deb,fin,h):
    """
    graphe : dictionnaire des listes d'adjacence
    deb : sommet de départ
    fin : sommet d'arrivée
    h : fonction heuristique
    """
    res={} # dictionnaire des valeurs minimales de distance entre le sommet  S et chacun des autres sommets
    dist={cle:float('inf') for cle in graphe} # initialisation du dictionnaire des distances
    dist[deb]=0 # sommet deb de départ, distance=0
    disth={cle:dist[cle]+h(cle) for cle in graphe} # dictionnaire des distances tenant compte de l'heuristique
    while fin in dist: # tant qu'on n'est pas arrivée au sommet fin
        smin=minimum(disth) # sommet de distance minimale pour démarrer une étape
        res[smin]=dist[smin] # on copie le sommet smin et sa distance dans le dictionnaire des résultats
        for k in range(len(graphe[smin])): # parcours des voisins de smin
            v,d=graphe[smin][k] # v : nom du voisin, d distance à laquelle il se trouve de smin
            if v not in res: # on ne s'intéresse qu'au voisin non déjà choisi
                dist[v]=min(dist[v],dist[smin]+d) # on choisit la distance minimale entre celle avec laquelle on aurait déjà pu arriver à v (dist[v]), et celle avec laquelle on arrive depuis smin (dist[smin]+d)
                disth[v]=dist[v]+h(v) # nouvelle distance minimale tenant compte de l'heuristique
        res[smin]=dist[smin] # on copie le sommet smin et sa distance dans le dictionnaire des résultats
        del dist[smin] # on supprime smin de dist, qui contient les sommets qu'il reste à visiter
        del disth[smin]
    return res
```

Pour utiliser cet algorithme, il faut définir la fonction heuristique.

```
def heuristique(s):
    return valeurs_heuristiques[s]
```

avec `valeurs_heuristiques` est le dictionnaire qui définit les valeurs heuristiques associé au graphe, elles peuvent être les distances « à vol d'oiseau » .

> **Exemple 9.** On étudie le graphe ci-dessous, et on souhaite déterminer le chemin le plus court pour aller du sommet `'S'` au sommet `'E'`.



Ce graphe est représenté par le dictionnaire des listes d'adjacences, et on choisit l'heuristique donnée par les valeurs heuristiques distance « à vol d'oiseau » jusqu'à `'E'` :

```
G={}
G['S']=[['A',7],['B',2],['C',3]]
G['A']=[['S',7],['B',3],['D',4]]
G['B']=[['S',2],['A',3],['D',1],['H',1]]
G['C']=[['S',3],['L',2]]
G['D']=[['A',4],['B',4],['F',5]]
G['E']=[['G',2],['K',5]]
G['F']=[['D',5],['H',3]]
G['G']=[['H',2],['E',2]]
G['H']=[['B',1],['F',3],['G',2]]
G['I']=[['J',6],['K',4],['L',4]]
G['J']=[['I',6],['K',4],['L',4]]
G['K']=[['I',4],['J',4],['E',5]]
G['L']=[['C',2],['I',4],['J',4]]

valeurs_heuristiques={'A':9,'B':7,'C':8,'D':8,'E':0,'F':6,'G':3,'H':6,'I':4
    , 'J':4 , 'K':3 , 'L':6 , 'S':10}
```

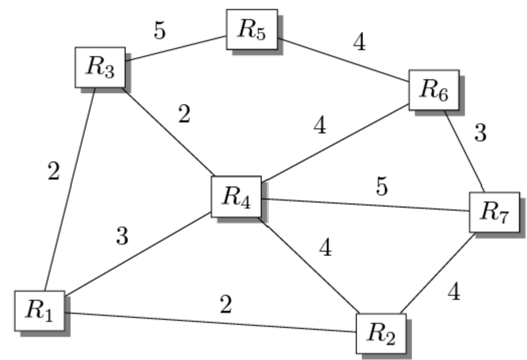
```
>>> A_etoile(G,'S','E',heuristique)
{'S': 0, 'B': 2, 'H': 3, 'G': 5, 'E': 7}

# Pour comparaison :
>>> Dijkstra(G,'S')
{'S': 0, 'B': 2, 'C': 3, 'D': 3, 'H': 3, 'A': 5, 'G': 5, 'L': 5, 'F': 6, 'E
    ': 7, 'I': 9, 'J': 9, 'K': 12}
```

Bien évidemment, **le résultat dépend de l'heuristique choisi**.

Exercice 20

On considère un graphe qui représente une partie d'un réseau constitué de sept appareils. On cherche la route la plus courte entre l'appareil R1 et l'appareil R6.



Ce réseau est défini par le dictionnaire `reseau` et l'heuristique est définie par le dictionnaire `heur` :

```
reseau={"R1" : [ ("R2",2) , ("R3",2) , ("R4",3) ] ,
        "R2" : [ ("R1",1) , ("R4",4) , ("R7",4)],
        "R3" : [ ("R1",2),("R4",2),("R5",5)],
        "R4" : [("R1",3),("R2",4),("R3",2),("R6",4),("R7",5) ],
        "R5" : [("R3",5),("R6",4)],
        "R6" : [("R4",4),("R5",4),('R7',3)],
        "R7" : [("R2",4),('R4',5),('R6',3)]
          }
heur = { "R1":6 , "R2":6 , "R3":6 , "R4":3 , "R5":3 ,  "R6":0 , "R7":3 }
```

Q1. Décrire les différentes étapes lors de l'exécution d'un programme utilisant l'algorithme de Dijkstra.

Q2. Comparer avec les étapes d'un programme utilisant l'algorithme A^*.

L'heuristique choisie est le nombre d'arêtes reliant les sommets multiplié par 3, où 3 est la longueur moyenne supposée d'une arête.

VI Matrices de pixels et images [Nouveauté]

Il est possible que la bibliothèque PIL, nécessaire à ce TP ne soit pas importée. En cas de problème, dans la console, recopier ces instructions :

```
pip uninstall PIL
pip install pip--upgrade
pip install Pillow
```

VI.1 Format des images

Le langage Python permet le traitement d'images de divers formats (.png,.bmp,.jpeg...). Quelque soit la nature de l'image (couleur ou noir et blanc), elle est vue comme un tableau représentant les $n \times p$ *pixels* de l'image de départ (en général une puissance de 2 comme $512, 1024$...). Une ligne du tableau correspond donc à une ligne de l'image (très fine souvent) et une colonne du tableau correspond à une colonne de l'image.

Chaque élément de ce tableau contient les informations propres au pixel qu'il représente. Ces informations sont de deux natures.

- Pour une image en noir et blanc, c'est un `1` pour un pixel blanc et un `0` pour un pixel noir.
- Pour une image en couleur, c'est une liste de trois nombres qui représentent respectivement les nuances des trois couleurs primaires dans l'ordre rouge (R), vert (V) et bleu (B). Chaque nombre est un entier entre 0 et 255. L'équilibre des trois nombres donne donc la couleur (au sens du mélange) du pixel en question.

> **Exemple 10.** La liste `[0,0,0]` code un pixel noir et `[255,255,255]` un pixel blanc. La liste `[255,0,0]` représente un pixel totalement rouge primaire, `[0,255,0]` un pixel totalement vert primaire et `[0,0,255]` un pixel totalement bleu primaire.

Chaque triplets de tels entiers représentant une couleur différente, il y a donc $255^3 = 16777216$ couleurs/nuances possibles.

En somme, pour une image couleur, tous éléments `T[i][j]` du tableau `T` représentant l'image est lui même une liste dont les éléments sont `T[i][j][k]` pour $k = 0, 1, 2$.

Pour toute la suite, on s'intéresse à des images couleurs pour lesquelles on va modifier les informations des pixels, modifier leurs positions etc.

VI.2 Ouverture et création d'une image

VI.2.a) L'importation de l'image

On considère une image `photo` au format `form` se trouvant à l'adresse `chemin` dans l'ordinateur. On importe `photo` dans le fichier `.py` qui réalise le traitement par les commandes.

```
import numpy as np # on importe la bibliothèque numpy
from PIL import Image # on importe la sous-bibliothèque Image
photo=Image.open("chemin/photo.form") # on ouvre dans Python la photo souhaitée
Tab=np.array(photo) #  on crée le tableau correspondant à l'image
```

Il est nécessaire d'identifier le chemin d'accès à `photo` (depuis l'explorateur de fichiers par exemple), de le copier et de le coller à l'endroit nécessaire. Certaine version de `Python` exige **le remplacement de tous les anti-slashs par des slashs**, ce que l'on fera en cas d'erreur de lecture.

Exercice 21

Dans la zone `Partage`, il y a une photo `3becs.jpg`. La copier dans votre zone personnelle dans le dossier dans lequel vous enregistrerez votre fichier `python` de ce TP.
Dans l'éditeur, réaliser l'ouverture de la photo des 3 becs, sous le nom `photo_becs`, puis créer le tableau correspondant, sous le nom `T_becs`.

VI.2.b) Les caractéristiques de l'image

On obtient les caractéristiques du tableau représentant l'image avec les instructions suivantes :

```
photo.size # taille de l'image en pixels
Tab.size #  nombre d'éléments dans le tableau
Tab.shape # dimensions du tableau
len(Tab) # nombre de lignes du tableau
len(Tab[0]) # nombre de colonnes du tableau
len(Tab[0][0]) # nombre d'éléments dans une des listes du tableau
```

VI.2.c) La création d'une image

Si on dispose d'un tableau `T` dont chaque élément est une liste de trois entiers entre 0 et 255, on peut créer l'image qui lui correspond par les lignes de code suivantes.

```
image=Image.fromarray(np.uint8(T))  # T est un tableau adéquat
image.show() # on affiche l'image
image.save("chemin/photo1.form") # on enregistre l'image sous le nom photo1
```

`uint8` représente l'encodage des données (le format des caractères du tableau).

VI.3 Transformations d'une photo

Pour copier un tableau `T` dans un tableau `T2` sans que les modifications de `T2` (resp. de `T`) entraînent celles de `T` (resp. de `T2`), on utilisera la fonction `deepcopy` disponible dans la bibliothèque `copy`.

VI.3.a) Composantes rouges

Exercice 22

Écrire une fonction `composanterouge(fichier)` qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg. La fonction crée une nouvelle image ne contenant que les composantes rouges, c'est-à-dire chaque triplet `[r,v,b]` est transformé en `[r,0,0]`, et l'affiche.

```
# En utilisant les facilités permises par les tableaux numpy :
def composanterouge(fichier):
    photo=Image.open(fichier) # j'ouvre le fichier dans la variable photo
    T=np.array(photo) # je convertis la photo en un tableau numpy
    T2=copy(T) # je copie T dans T2 pour modifier T2 sans modififer T
    T2[:,:,1]=0 # pour tous les éléments (i,j) du tableau, je mets le pixel vert à 0
    T2[:,:,2]=0 # pour tous les éléments (i,j) du tableau, je mets le pixel bleu à 0
    photo2=Image.fromarray(np.uint8(T2)) # création de l'image correspondant au tableau T2
    photo2.show() # affichage de la photo2

# En parcourant le tableau :
def composanterouge2(fichier):
    photo=Image.open(fichier)
    T=np.array(photo)
    dim=T.shape # dimensions du tableau (n_lignes, n_colonnes, 3)
    T2=np.zeros((dim[0],dim[1],3)) # le tableau où l'on stockera le résultat
    # parcours du tableau :
    for i in range(dim[0]):
        for j in range(dim[1]):
                                 # à compléter : on garde le pixel rouge
                                 # à compléter : pixel vert à 0
                                 # à compléter : pixel bleu à 0
    photo2=Image.fromarray(np.uint8(T2))
    photo2.show()
```

VI.3.b) Négatif

Exercice 23 On appelle *négatif* de tout triplet `[r,v,b]` le triplet `[255-r,255-v,255-b]`.
Écrire une fonction `negatif(fichier)` qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg. La fonction crée une nouvelle image négative de celle du fichier.

VI.3.c) Niveau de gris

Exercice 24

Q1. Écrire une fonction `gris1(fichier)` qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg. La fonction crée une nouvelle image en niveaux de gris et l'affiche.

Pour calculer l'intensité de gris, on calcule la moyenne m des intensités de rouge, de vert et de bleu. Un pixel (r, v, b) devient un pixel (m, m, m).

Q2. Écrire une fonction `gris2(fichier)` semblable à `gris1`, mais le niveau de gris est calculé par $m = 0,2126 \times r + 0,7152 \times v + 0,0722 \times b$.

Q3. Écrire une fonction `noir_blanc` qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg et un nombre entier naturel nommé `seuil`.

La fonction crée une nouvelle image en noir et blanc et l'affiche : si la valeur du niveau de gris du pixel, calculé selon la formule utilisée dans `gris2` est supérieure à la valeur `seuil`, le pixel devient blanc , sinon il est noir.

Exercice 25 Détection de contour

Écrire une fonction `DessineContours(fichier)` qui va créer une nouvelle image à partir de l'image du fichier `fichier` qui va dessiner les contours de l'image.

Pour cela, commencer par convertir l'image en niveau de gris (avec la fonction `gris2`) et créer le tableau `T` correspondant, puis créer un nouveau tableau `T1` telle que :

$$T1[i][j] = abs(T[i+1][j] - T[i-1][j]) + abs(T[i][j+1] - T[i][j-1])$$

La valeur d'un pixel sera d'autant plus grande que les variations entre les voisins de ce pixels seront grands. Cela veut dire dans ce cas, qu'on est à une frontière entre deux parties de l'image de niveaux de gris très différents.

VI.3.d) Atténuation

Exercice 26 Atténuation

Écrire une fonction `attenuation(fichier,alpha)` qui divise chaque valeur d'un triplet de couleur par un entier `alpha`.

VI.3.e) Modifications géométriques

Exercice 27 Symétries

Q1. Écrire une fonction `renverse(fichier)` qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg et crée une nouvelle image renversée par symétrie centrale.

Q2. Écrire des fonctions `miroir_vert(fichier)`qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg et crée une nouvelle image symétrique d'axe vertical.

Q3. Écrire des fonctions `miroir_horiz(fichier)`qui prend en paramètre le nom d'un fichier représentant une image en couleur au format jpg et crée une nouvelle image symétrique d'axe horizontal.

Q4. Faire apparaître les trois images transformées.

Exercice 28 Agrandissement

Agrandir une figure du plan revient à construire son image par une homothétie de rapport $k > 1$.
Par contre, si nous disposons d'une image de $n \times m$ pixels que nous voulons agrandir pour en faire une image de $kn \times km$ pixels avec $k > 1$, entier, l'idée de partir d'un point de l'image d'origine pour placer son homothétique dans la nouvelle image dans l'image d'arrivée est clairement insuffisante : il y a k^2nm pixels dans la nouvelle image pour nm dans l'image d'origine.
On procédera en réservant un tableau T_1 pour l'image d'arrivée et pour chacun des points ou pixels de T_1, on déterminera le point ou pixel de l'image de départ le plus proche de son antécédent géométrique, et on reportera la valeur moyenne de pixels avoisinants dans le pixel de T_1

Q1. Écrire une fonction `interpoler9(T,x,y)` qui prend en argument un tableau T représentant une image, `x` et `y` deux entiers, et renvoie, une liste de trois nombres, chacun correspondant, pour chaque composante, à la moyenne du contenu du pixel (x, y) et de ses voisins. On tiendra compte du cas où (x, y) est sur le bord de l'image.

Q2. Écrire une fonction `agrandir(fichier,k)` qui prend en argument le nom d'un fichier représentant une image en couleur au format jpg et crée une nouvelle image agrandie d'un facteur k.