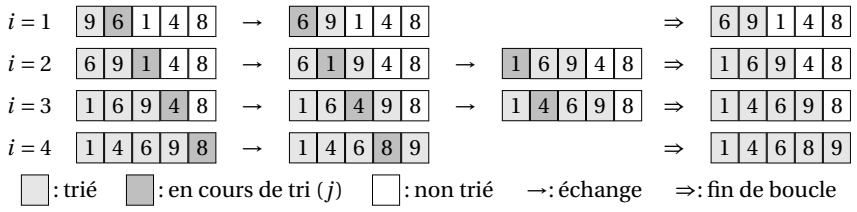
Informatique de Tronc Commun, MP2, D.S. 1

1 Un problème d'anagrammes

1.1 Tri par insertion

Le tri par insertion est le tri du joueur de carte, qui classe ses cartes au fur et à mesure qu'il les reçoit.

Pour l'implémenter, on peut utiliser un algorithme itératif, fondé sur une double boucle. Le schéma suivant qu'on doit lire de gauche à droite, ligne après ligne, indique, sur un exemple, comment on peut mettre en place ces deux boucles pour trier la liste `L=[9,6,1,4,8]` en rangeant les entiers dans l'ordre croissant.



Ce schéma fait comprendre que i est l'indice [1], dans la liste, de la valeur qu'on veut *insérer dans le paquet déjà ordonné des valeurs précédentes* et ce placement de la valeur `L[i]` au bon endroit parmi les précédentes va se faire par déplacements successifs vers la gauche (appelés échanges sur la figure). Ces déplacements feront intervenir une autre boucle (celle qui se déroule horizontalement sur le schéma et dont la fin est indiquée par le ⇒ du schéma).

On va décomposer l'algorithme en deux fonctions comme suit :

```
def tri_Insertion(L):
    for i in range(1,len(L)):
        insere(i,L)
# reste alors  à définir la fonction insere
def insere(i : int,L : list):
  """ i est un indice valide pour L, et L[0:i] est supposée déjà triée
la fonction modifie L pour que la tranche L[0: i+1]  soit triée"""
```

Question 1 Compléter le code de la fonction `insere` proposée.

Question 2 Complexité :

a) Quelle est la complexité, dans le pire cas, de la fonction `tri_Insertion` en terme de nombre de comparaisons (on ne comptera pas les affectations).

b) Quelle est sa complexité si on l'applique à une liste déjà triée ?

Python dispose de deux fonctions de tri de listes, appelées respectivement `sort` et `sorted` dont voici un exemple de fonctionnement :

```
>>> L=[1,3,2]

>>> M=sorted(L)
[1, 2, 3]

>>> L
[1, 3, 2]
```

1. toujours en comptant à partir de 0

```
>>>M
[1,2,3]

>>>L.sort()
>>> L
[1,2,3]
```

Question 3 Le comportement (en terme de valeur de retour/modification de la valeur d'entrée) de votre fonction `tri_Insertion` est-il proche de celui de `sorted` ou de `sort` ?

1.2 Type ensemble

En Python, on dispose d'un type `set` (ensemble en français) qui est une implémentation informatique de la notion d'ensemble et que l'on rentre avec des accolades, sauf pour l'ensemble vide défini par `set()`.

Dans un ensemble, ni l'ordre des données, ni les répétitions ne comptent, par exemple :

```
>>> S={5,4,1,1,4}

>>> S
{1, 4, 5}
```

On peut partir d'un ensemble vide et ajouter des éléments avec `add`

```
>>> S=set()
>>>S.add(17)
>>>print(S)
>>> S.add(17)
>>>print(S)
```

On verrait que les deux `print` donne la même chose : `{17}` .

Question 4 La documentation de Python explique que les *ensembles* sont fabriqués comme cas particuliers de *dictionnaires* avec des clefs sans valeur.

a) Avec vos connaissances sur les *dictionnaires* donner la complexité moyenne de l'ajout, de la suppression, et de la recherche d'un élément dans un ensemble.

b) Comment s'appelle la structure informatique utilisée par les dictionnaires permettant d'obtenir cette complexité moyenne ?

1.3 A l'attaque des anagrammes

Définition – Un mot w est une anagramme d'un mot v s'il existe une permutation des lettres qui transforme w en v.

Étant donné un ensemble de n mots de longueur au plus k, on veut détecter toutes les classes d'anagrammes, autrement dit on veut écrire une fonction `Anagrammes_Presentes` dont voici un exemple d'entrées-sorties :

• **entrée :** `"le chien marche vers sa niche, et trouve une limace de chine nue, pleine de malice, qui lui fait du charme! "`

• **sortie :** `[['limace', 'malice'], ['chien', 'niche', 'chine'], ['marche', 'charme'], ['nue', 'une']]`

On dispose de la commande Python `ord` qui prend en entrée une chaîne constituée de 1 caractère et renvoie le code ascii entre 0 et 255 de ce caractère : ces caractères ne sont pas forcément des lettres, mais on ne s'intéresse qu'aux lettres, accentuées ou non :

• pour les lettres Majuscules : code entre 65 pour A et 90 pour Z,

• pour les lettres minuscules : code entre 97 pour a et 122 pour z,

• pour les lettres accentuées dans l'ascii étendu (déjà beaucoup de langues...) : code entre 192 et 255.

Ainsi :

```
>>> ord("c")
99

>>> ord("é")
233
```

Question 5 Traitement du texte en entrée :

a) Ecrire une fonction `teste_lettre(a)` où a est une chaine de 1 caractère, qui renvoie `True` ou `False` suivant que ce caractère correspond à une lettre ou pas.

b) Ecrire une fonction `decoupe_en_mots(phrase : str) --> list` qui prend en entrée une chaîne de caractères comme dans l'exemple ci-dessus et retourne la liste des mots qui apparaissent dans cette phrase.

Par exemple

```
decoupe_en_mots("une petite phrase, avec une ponctuation, phrase bête.")
['une', 'petite', 'phrase', 'avec', 'une', 'ponctuation', 'phrase', 'bête']
```

c) Que fait la fonction suivante ?

```
def mystere(L:list):
    return list(set(L))
```

Donner une valeur de retour possible de `mystere(['une','petite','une','grande','petite'])`.

Question 6 Dictionnaire de signatures des mots du texte entré.

Dans ce qui suit, pour simplifier le tri des lettres, on supposera que le texte est en minuscules sans accents.

a) Sachant que Python compare les caractères correctement entre eux i.e. que par exemple `'a'<'b'` renvoie `True`, écrire, à l'aide de la fonction `Tri_Insertion` faite plus haut, une fonction `signature`, qui prend en argument un mot et renvoie le mot formé par les caractères de ce mot ordonnés dans l'ordre alphabétique croissant. Par exemple

```
 >>> signature('bazar')
'aabrz'
```

b) A l'aide de ce qui précède : écrire une fonction `dico_signature(phrase)` qui prend en entrée une chaîne de caractères `phrase` comme à la question 5 et renvoie un dictionnaire dont les clefs sont les signatures des différents mots de `phrase` et pour chaque signature la valeur associée est la liste des mots de la phrase ayant cette signature.

Par exemple :

```
>>> phrase="le chien marche vers sa niche, et trouve une limace
de chine nue, pleine de malice,  qui lui fait du charme! "
>>> dico_signature(phrase)
{'iqu': ['qui'], 'cehin': ['chine', 'niche', 'chien'], 'enu': ['nue', 'une'],
'aceilm': ['malice', 'limace'], 'ersv': ['vers'], 'eortuv': ['trouve'],
 'acehmr': ['marche', 'charme'],
 'as': ['sa'], 'eeilnp': ['pleine'], 'ilu': ['lui'], 'afit': ['fait'], 'du': ['du'],
 'et': ['et'], 'el': ['le'], 'de': ['de']}
```

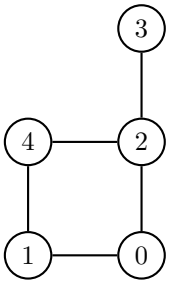
Question 7 Conclusion : A l'aide de ce qui précède, écrire la fonction `Anagrammes_Presentes` annoncée au début de ce paragraphe 1.3.

2 Parcours de graphes

2.1 Représentation des graphes

Une première façon de se donner un graphe non orienté est de se donner son ensemble de sommets S et son ensemble d'arêtes A où chaque arête est une paire de sommets. Informatiquement, on codera plutôt S comme une liste Python, et A comme une liste de couples.

Par exemple, pour l'exemple ci-dessous `S=[0,1,2,3,4]` et `A=[[0,1],[1,4],[2,4],[0,2],[2,3]]`.



Question 8 Ecrire une fonction `DA(S,A)` qui prend en argument des listes `S` et `A` comme ci-dessus et renvoie un dictionnaire d'adjacence, dont les clefs sont les éléments de `S` et la valeur pour chaque clef est une liste formée des sommets voisins du sommet correspondant à la clef.

Sur l'exemple ci-dessus :

```
>>> DA(S,A)
{0: [1, 2], 1: [0, 4], 2: [4, 0, 3], 3: [2], 4: [1, 2]}
```

A partir de maintenant, dans tout ce qui suit, on supposera (*) que les sommets du graphe sont toujours numérotés par l'ensemble $[\![0, n-1]\!]$.

Question 9 Ecrire une fonction `LA(S,A)` semblable à la précédente sauf qu'elle renvoie une `liste Python` `L` telle que `L[i]` est une liste des voisins du sommet i. Sur l'exemple ci-dessus :

```
>>> LA(S,A)
[[1, 2], [0, 4], [4, 0, 3], [2], [1, 2]]
```

Question 10 Du point de vue de l'efficacité, pour les graphes vérifiant la condition **(*)** ci-dessus, que faut-il préférer entre `DA` et `LA` ?

Question 11 On veut écrire une fonction `MA(D)` qui prend en argument une liste d'adjacence `L` comme celle fabriquée par la fonction `LA` précédente et renvoie la matrice d'adjacence du graphe non orienté correspondant.

a) Quelle instruction d'importation doit-on exécuter préalablement pour utiliser la commande : `M=np.zeros((n,n), dtype=int)` et que fait cette commande ?

b) A l'aide de la commande précédente, écrire la fonction `MA` demandée qui, sur l'exemple précédent, renverra :

```
array([[0, 1, 1, 0, 0],
       [1, 0, 0, 0, 1],
       [1, 0, 0, 1, 1],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 0, 0]])
```

2.2 Une fonction de parcours

On considère la fonction récursive suivante :

```
def PR(M,i,marque):
   """ M: matrice d'adjacence,
   i: sommet à partir duquel lancer l'exploration,
   marque : une liste de sommets, vide lors de l'appel principal"""
   marque.append(i)
   n=len(M)
   for j in range(n):
       if M[i,j]!=0 and j not in marque :
           print('appel récursif avec j=',j)
           PR(M,j,marque)
   print("traitement final de",i)
```

Question 12 On considère la matrice d'adjacence `M` de la question 11b). On exécute `PR(M,0,[])`. Donner la suite des affichages obtenus.

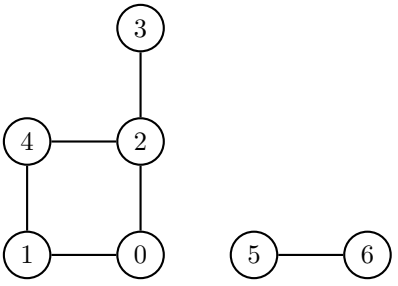
Question 13 On comprend sur l'exemple précédent que si le graphe est connexe, chaque sommet `i` ne sera « traité » qu'une seule fois. Comment s'appelle ce type de parcours de graphe : en profondeur ou en largeur (justifier) ?

Question 14 On considère maintenant la fonction suivante :

```
def PT(M):
    n=len(M)
    marque=[]
    for i in range(n):
        if i not in marque :
            print('appel principal avec i=',i)
            PR(M,i,marque)
```

a) Qu'affichera `PT(M)` avec la matrice `M` précédente ?

b) On considère maintenant le graphe suivant dont on appelle `M2` la matrice d'adjacence. Comment obtenir cette matrice `M2` de manière économique à partir des objets et fonctions déjà définis ?



c) Qu'affichera `PT(M2)` ?

Question 15 Evaluer la complexité de `PR` appliqué à un graphe connexe ayant n sommets.