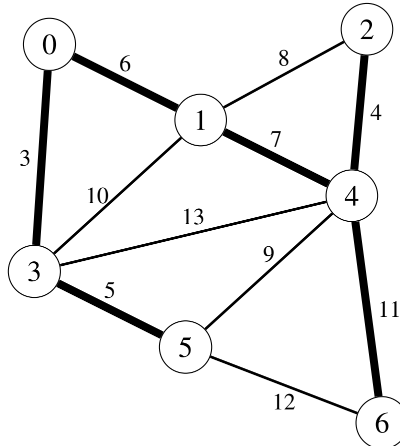
# Arbre couvrant minimal

## Corrigé

### MPSI - PCSI

**Q1)** Arbre de poids minimal pour Graphe 2 :




**Q2)** a)

```python
            if s<c[0] :
                L.append((c[1],s,c[0]))
```

b) On compte une opération en #1, puis la ligne # 5 est exécutée $2e$ fois (chaque arête non orientée fait passer exactement 2 fois dans la boucle interne), et exactement la moitié des tests sont vrais donc #6 est exécuté $e$ fois :

$$C_e = 1 + 2e + e = 3e + 1 = \boxed{O(e)}.$$

**Q3)** a)

- Étape (0) : CC = $\{0:0,\ 1:1,\ 2:2,\ 3:3,\ 4:4,\ 5:5,\ 6:6\}$,
- Étape (1) : CC = $\{0:0,\ 1:1,\ 2:2,\ 3:0,\ 4:4,\ 5:5,\ 6:6\}$,
- Étape (2) : CC = $\{0:0,\ 1:1,\ 2:2,\ 3:0,\ 4:2,\ 5:5,\ 6:6\}$,
- Étape (3) : CC = $\{0:0,\ 1:1,\ 2:2,\ 3:0,\ 4:2,\ 5:0,\ 6:6\}$,
- Étape (4) : CC = $\{0:0,\ 1:0,\ 2:2,\ 3:0,\ 4:2,\ 5:0,\ 6:6\}$,
- Étape (5) : CC = $\{0:0,\ 1:0,\ 2:0,\ 3:0,\ 4:0,\ 5:0,\ 6:6\}$,
- Étape (6) : CC = $\{0:0,\ 1:0,\ 2:0,\ 3:0,\ 4:0,\ 5:0,\ 6:0\}$.

b)

```python
def init_comp(G : dict) -> dict :
    """ Renvoie le dictionnaire associant à chaque sommet du graphe lui-même. """
    CC={}
    for s in G :
        CC[s]=s
    return(CC)
```

c)

```python
def même_comp(CC : dict, s1 : int, s2 : int) -> bool :
    """ Renvoie True si s1 et s2 appartiennent à la même composante connexe et False sinon. """
    return(CC[s1]==CC[s2])
```

d)

```python
def fusionne_comp(CC : dict, s1 : int, s2 : int) -> None :
    """ Fusionne les composantes connexes de s1 et s2. """
    if CC[s2]<CC[s1] :
        anc,nvx=CC[s1],CC[s2]
    else :
```

```
6            anc,nvx=CC[s2],CC[s1]
7        for s in CC :
8            if CC[s]==anc :
9                CC[s]=nvx
```

e) `init_comp` : $C_v = 1 + v = O(v)$.
`même_comp` : $C_v = 1 = O(1)$.
`fusionne_comp` (pire cas) : $C_v = 3 + v \times 2 = O(v)$.

**Q4)** a) `insère([2,4,6,8],3)` :

| $i$ | $k_i$ | $L_i$ |
|---|---|---|
| 0 | 4 | [2, 4, 6, 8, 3] |
| 1 | 3 | [2, 4, 6, 3, 8] |
| 2 | 2 | [2, 4, 3, 6, 8] |
| 3 | 1 | [2, 3, 4, 6, 8] |

`insère([2,4,6,8],1)` :

| $i$ | $k_i$ | $L_i$ |
|---|---|---|
| 0 | 4 | [2, 4, 6, 8, 1] |
| 1 | 3 | [2, 4, 6, 1, 8] |
| 2 | 2 | [2, 4, 1, 6, 8] |
| 3 | 1 | [2, 1, 4, 6, 8] |
| 3 | 0 | [1, 2, 4, 6, 8] |

`insère([2,4,6,8],9)` :

| $i$ | $k_i$ | $L_i$ |
|---|---|---|
| 0 | 4 | [2, 4, 6, 8, 9] |

b)

```
1   def insère(L : list, x) -> None :
2       """
3       Modifie la liste L en insérant x de manière à ce que la liste obtenue
4       soit toujours triée dans l'ordre croissant.
5       Entrée
6           L : liste triée dans l'ordre croissant dont tous les éléments sont d'un même type
7           x : valeur du même type
8       Sortie
9           Aucune : x est inséré dans L par effet de bord
10      Exemples :
11      >>>L=[2,4,6,8]
12      >>>insère(L,3)
13      >>>L
14      [2,3,4,6,8]
15      >>>L=[2,4,6,8]
16      >>>insère(L,1)
17      >>>L
18      [1,2,4,6,8]
19      >>>L=[2,4,6,8]
20      >>>insère(L,9)
21      >>>L
22      [2,4,6,8,9]
23      """
24
25      assert type(L)==list
26      for e in L :
27          assert type(e)==type(x)
```

c) On suppose par l'absurde que la boucle while ne se termine pas. Alors $(k_i)_{i\in\mathbb{N}}$ est une suite d'entiers strictement décroissante (on a $k_{i+1} = k_i - 1$ et $k_0 = \text{len}(L)$) et minorée (par 0 puisque la boucle ne s'arrête pas), ce qui est absurde.

d) On a $\boxed{k_i = n - i}$.
Initialisation : la propriété est vraie au rang 0 car $L_0 = [\ell_0, \dots, \ell_{n-1}, x]$ juste avant d'entrer dans la boucle.
Hérédité : supposons la propriété vraie à un rang $i$ pour lequel l'itération $i+1$ a lieu. Alors en utilisant l'hypothèse de récurrence et le fait que $k_i = n - i$, en effectuant #5 on obtient

$$L_{i+1} = [\ell_0, \dots, \ell_{n-i-2}, x, \ell_{n-i-1}, \dots, \ell_{n-1}]$$

ce qui est bien l'hypothèse au rang $i + 1$.

e) Notons $d$ la dernière itération.
Comme on quitte la boucle, la condition $k_d > 0$ and $L_d[k_d - 1] > L_d[k_d]$ est fausse.
$1^e$ cas : $k_d > 0$ est faux.
C'est alors que $k_d = 0$, et donc $d = n$, et en utilisant la propriété prouvée à la question précédente pour $i = d$ :

$$L_d = [x, \ell_0, \dots, \ell_{n-1}].$$

Or l'itération $d - 1$ ayant eu lieu, c'est que $L_{d-1}[k_{d-1} - 1] > L_{d-1}[k_{d-1}]$, ie. $\underbrace{L_{n-1}[0]}_{\ell_0} > \underbrace{L_{n-1}[1]}_{x}$. On a donc

$$x < \ell_0 \leqslant \cdots \leqslant \ell_{n-1}$$

donc la liste $L_d$, qui est renvoyée, est bien rangée dans l'ordre croissant.
$2^e$ cas : $L_d[k_d - 1] > L_d[k_d]$ est faux.
C'est alors que $\ell_{n-d-1} \leqslant x$, et en utilisant que la condition de l'itération $d - 1$ était vraie on montre comme au premier cas que $\ell_{n-d} > x$. On a donc

$$\ell_0 \leqslant \cdots \leqslant \ell_{n-d-1} \leqslant x < \ell_{n-d} \leqslant \ell_{n-1}$$

donc la liste $L_d$ est bien rangée dans l'ordre croissant.

f) On note $n$ la longueur de la liste L avant l'appel. On compte 2 opérations avant le while, puis répétés au maximum $n$ fois 9 opérations (on compte les tests, les affectations, et les soustractions), et enfin pour sortir de la boucle au plus 3 opérations. Ainsi

$$C_n \leqslant 2 + 9n + 3 = 5 + 9n.$$

g)
```
1        insère(Ltriée,a)
```

On compte 1 opération avant la boucle, puis en utilisant que la liste Ltriée gagne un élément à chaque passage dans la boucle, on compte un appel à `insère` sur une liste de taille $i$ pour le $i^e$ passage dans la boucle `for`, donc

$$C_e \leqslant 1 + \sum_{i=1}^{e}(5 + 9i) = 1 + 5e + 9\frac{e(e+1)}{2} = O(e^2).$$

h) Dans la liste L les éléments sont des triplets dont le premier élément est le poids de l'arête, le tri se fait par ordre lexicographique en comparant d'abord le premier élément du triplet, donc dans l'ordre croissant du poids de l'arête (en cas d'égalité de poids, les arêtes sont ordonnées suivant le premier sommet, et en cas d'égalité suivant le deuxième).

**Q5)** a)

```python
def Kruskal(G : dict) -> list :
    """ Renvoie la liste des arêtes de l'arbre couvrant minimal
    du graphe de dictionnaire d'adjacence G. """
    L=liste_arêtes(G)
    Lt=tri_croissant(L)
    CC=init_comp(G)
    ACM=[]
    for a in Lt :
        if not(même_comp(CC,a[1],a[2])) :
            ACM.append(a)
            fusionne_comp(CC,a[1],a[2])
    return ACM
```

b) On compte $O(e)$ opérations en #4, $O(e^2)$ opérations en #5, $O(v)$ opérations en #6, puis dans le prie cas $O(1) + 1 + O(v)$ opérations répétées pour chacune des $e$ arrêtes de Lt. Ainsi, la complexité est en $\boxed{O(e^2 + ev)}$.

c)

```python
def construit_ACM(L : list) -> dict :
    """ Renvoie le dictionnaire d'adjacence associé à la liste d'arêtes L. """
    ACM={}
    for a in L :
        if a[1] not in ACM :
            ACM[a[1]]=[]
        if a[2] not in ACM :
            ACM[a[2]]=[]
        ACM[a[1]].append((a[2],a[0]))
        ACM[a[2]].append((a[1],a[0]))
    return(ACM)
```

**Q6)** a)

```python
def init_comp(G : dict) -> dict :
    CC={s : None for s in G}
    return(CC)
def même_comp(CC : dict, s1 : int, s2 : int) -> bool :
    while CC[s1]!=None :
        s1=CC[s1]
    while CC[s2]!=None :
        s2=CC[s2]
    return(s1==s2)
def fusionne_comp(CC : dict, s1 : int, s2 : int) -> None :
    while CC[s1]!=None :
        s1=CC[s1]
    while CC[s2]!=None :
        s2=CC[s2]
    if s1<s2 :
        CC[s2]=s1
    else :
        CC[s1]=s2
```

b)

```python
def même_comp(CC : dict, s1 : int, s2 : int) -> bool :
    L1=[]
    while CC[s1]!=None :
        L1.append(s1)
        s1=CC[s1]
    for s in L1 :
        CC[s]=s1
    L2=[]
    while CC[s2]!=None :
        L2.append(s2)
        s2=CC[s2]
    for s in L2 :
```

```
13            CC[s]=s2
14        return(s1==s2)
```