

# Des tris, des couleurs et des distances

INFORMATIQUE COMMUNE - Devoir n°4 - Olivier Reynet

## Consignes :

1. utiliser une copie différente pour chaque partie du sujet,
2. écrire son nom sur chaque copie,
3. écrire de manière lisible et intelligible,
4. préparer une réponse au brouillon avant de la reporter sur la feuille.

## A Trier conformément à un certain ordre

Cette section a pour objectif d'élaborer un code qui trie une liste conformément à l'ordre d'une autre liste.

■ **Exemple 1 — Trier conformément à.** Soit  $\kappa = [0, 1, 2, 3, 4, 5]$  et  $L = [3, 5, 2, 2, 4, 5]$ . On souhaite trier dans l'ordre décroissant  $L$  et faire subir les mêmes modifications à la liste  $\kappa$ .

Le tri décroissant de  $L$  résulte en  $L = [5, 5, 4, 3, 2, 2]$ , ce qui induit que  $\kappa$  devient  $[1, 5, 4, 0, 2, 3]$ .

Dans ce but, on se propose d'utiliser le tri par insertion.

- A1.** On suppose qu'on dispose d'une liste **partiellement** triée  $L$  dans l'ordre décroissant jusqu'à l'indice  $i-1$ . Écrire une fonction de signature `insert_elem(L, i)` qui insère l'élément d'indice  $i$  à la bonne place dans  $L[0 : i+1]$ . Cette fonction agit en place et ne renvoie donc rien. Par exemple, si  $L = [5, 2, 1, 4, 0, 7]$ , alors `insert_elem(L, 3)` modifie  $L$  ainsi :  $[5, 4, 2, 1, 0, 7]$ .
- A2.** Compléter le code de la fonction signature `tri_ins(L)` qui effectue le tri par insertion en place de la liste  $L$ .

```
def insertion_sort(t):  
    for i in range(1, len(t)):  
        # à compléter
```

- A3.** En vous inspirant du code précédent, écrire une fonction de signature `trier_conforme(L, K)` qui trie  $K$  conformément au tri de  $L$  dans l'ordre décroissant. Au début de la fonction, on garantira par une assertion que les deux listes passées en paramètre sont de même taille.
- A4.** Quelle est la complexité de la fonction `trier_conforme` dans le pire et le meilleur des cas?
- A5.** En vous inspirant des questions précédentes, proposer une méthode pour trier conformément  $K$  selon  $L$  avec une complexité en  $O(n \log n)$  dans tous les cas.
- A6.** Écrire un code qui implémente cette méthode et qui renvoie les deux listes. On décomposera ce code en deux fonctions récursives dont les signatures sont :

- `tc(L: list[int], K: list[int]) -> tuple(list[int], list[int])`,
- `f(L1: list[int], L2: list[int], K1: list[int], K2: list[int]) -> tuple(list[int], list[int])`.

## B Coloration de graphe

■ **Définition 1 — Coloration valide.** Une coloration d'un graphe est valide lorsque deux sommets adjacents n'ont jamais la même couleur.

■ **Définition 2 — Nombre chromatique.** Le nombre chromatique d'un graphe  $G$  est le plus petit nombre de couleurs nécessaires pour obtenir une coloration valide de ce graphe. On le note généralement  $\chi(G)$ .

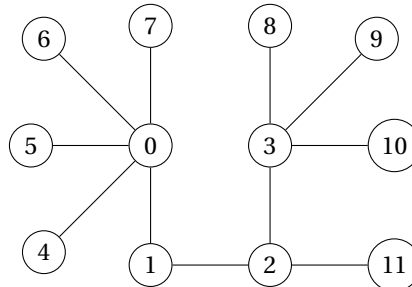


FIGURE 1 – Graphe  $g$

On se donne un graphe  $g$  (cf. figure 1) ainsi qu'une liste de couleur utilisables pour colorer un graphe.

```
colors = ["pink", "turquoise", "yellow", "blue", "magenta", "orange", "lime"]
```

- B7.** En utilisant les listes Python, représenter par une liste d'adjacence le graphe  $g$  de la figure 1.
- B8.** Quel est le nombre chromatique du graphe 1? Trouver ce nombre par le dessin et dessiner le graphe ainsi coloré.

L'algorithme de Welsh-Powell (cf. algorithme 1) est un algorithme de coloration de graphe glouton. Il traite les sommets dans l'ordre décroissant de leur degré. **Pour chaque sommet, dans l'ordre décroissant des degrés, il choisit une nouvelle couleur et l'affecte aux autres sommets du graphe si cela est possible, c'est-à-dire s'ils ne sont ni déjà colorés et ni directement adjacents.**

---

#### Algorithme 1 Welsh-Powell

---

```

1: Fonction WP( $g$ , couleurs)
2:    $n \leftarrow$  l'ordre du graphe  $g$ 
3:   sommets  $\leftarrow$  la liste des sommets du graphe triée dans l'ordre des degrés décroissants
4:   cmap  $\leftarrow$  une liste à  $n$  éléments initialisés à None ▷ les couleurs de chaque sommet
5:   tant que sommets et couleurs ne sont pas vides répéter
6:      $s \leftarrow$  le sommet de degré le plus fort et le retirer de la liste
7:      $c \leftarrow$  une couleur de la liste couleurs et la retirer de la liste
8:     cmap[ $s$ ]  $\leftarrow$   $c$  ▷ Colorer le sommet  $s$  en  $c$ 
9:     colorables  $\leftarrow$  liste des sommets colorables en  $c$  parmi les sommets restants dans l'ordre
10:    pour chaque  $v$  de colorables répéter
11:      cmap[ $v$ ]  $\leftarrow$   $c$  ▷ Colorer le sommet  $v$  en  $c$ 
12:      supprimer  $v$  de la liste sommets
13:  renvoyer cmap

```

---

- B9.** Écrire une fonction de signature `degres( $g$ )` dont le paramètre est un graphe sous la forme d'une liste d'adjacence. Cette fonction renvoie la liste des degrés des sommets du graphe. Par exemple, `[[1], [0, 2], [1, 3], [2]]` renvoie `[1, 2, 2, 1]`.
- B10.** Écrire une fonction de signature `remove( $L$ : list[int],  $e$ )`  $\rightarrow$  list[int] qui renvoie une nouvelle liste copie de  $L$  mais dans laquelle toutes les occurrences de l'élément de valeur  $e$  ont été supprimées.

- B11.** Écrire une fonction de signature `colorables(g, s) → list[bool]` qui prend comme paramètre le graphe `g` sous la forme d'une liste d'adjacence et `s` un sommet. Cette fonction renvoie une liste de booléens de la taille de `g` initialisées à `True`, sauf pour `s` et les voisins de `s`. Par exemple, `colorables([[1], [0, 2], [1, 3], [2]], 1)` renvoie `[False, False, False, True]`
- B12.** Écrire une fonction de signature `meme_couleur(g, s, sommets)` qui renvoie la liste des sommets de `g` colorables de la **même** couleur que `s` parmi les sommets de la liste `sommets`. On utilisera la liste générée par la fonction `colorables`. Par exemple, `meme_couleur([[1], [0, 2], [1, 3], [2]], 0, [2, 3])` renvoie `[2]`.
- B13.** Écrire une fonction de prototype `welsh_powell(g, colors)` qui implémente l'algorithme de Welsh-Powell. Cette fonction renvoie une liste `cmap` telle que `cmap[i]` est la couleur du sommet `i`. S'appuyer sur les fonctions précédentes.
- B14.** Appliquer à la main l'algorithme de Welsh-Powell sur le graphe `g` de la figure 1. Que pouvez-vous en conclure?
- B15.** Comment peut-on qualifier cet algorithme?

## C Parcours et vérification

On souhaite vérifier que l'algorithme de coloration précédent a bien généré une coloration valide, c'est-à-dire que chaque sommet possède une couleur distincte de celle de ses voisins.

- C16.** On suppose que le graphe coloré est connexe et non orienté. Proposer un algorithme fondé sur le parcours en largeur pour effectuer cette vérification. Donner des explications sur la méthode en langage naturel, sans coder en Python.
- C17.** Compléter le code ci-dessous afin d'implémenter cette vérification. Sur la copie, bien mettre en exergue les `#X` complétés.

```
def coloration_valide(g, color_map):
    n = len(g)
    F = []
    D = [False for _ in range(n)]
    F.append(0)
    D[0] = #1
    while len(F) > 0:
        s = #2
        for #3 in #4:
            if #5:
                return False
            if not D[v]:
                D[v] = True
                F.append(v)
    return #6
```

On considère maintenant le graphe ci-dessous, donné sous la forme d'une matrice d'adjacence :

```
G = [[0, 3, 0, 0, 7, 0, 0, 0],
      [3, 0, 3, 0, 0, 7, 0, 0],
      [0, 3, 0, 2, 0, 0, 0, 0],
      [0, 0, 2, 0, 0, 0, 4, 0],
      [7, 0, 0, 0, 0, 4, 0, 1],
      [0, 7, 0, 0, 4, 0, 2, 0],
      [0, 0, 0, 4, 0, 2, 0, 0],
      [0, 0, 0, 0, 1, 0, 0, 0]]
```

- C18.** Dessiner `G`.
- C19.** Comment peut-on qualifier ce graphe?
- C20.** Convertir `G` en une liste d'adjacence.

**C21.** Écrire une fonction de signature `mapj_to_ladj(g)` qui convertit un graphe de type `G` donné par sa matrice d'adjacence en une liste d'adjacence.

On souhaite calculer les plus courts chemins entre des sommets de même couleur. Dans ce but, on applique l'algorithme de Dijkstra.

**C22.** Pourquoi peut-on appliquer l'algorithme de Dijkstra au graphe `g`?

**C23.** On applique l'algorithme à la main au départ du sommet 2 en construisant le tableau des distances à chaque itération de l'algorithme de Dijkstra. Compléter le tableau ci-dessous.

$\Delta$	0	1	2	3	4	5	6	7	Prochain
{2}	$\infty$	3	0	2	$\infty$	$\infty$	$\infty$	$\infty$	
{2,...}	...	...	0	...	...	...	...	...	
{2,...}	...	...	0	...	...	...	...	...	
{2,...}	...	...	0	...	...	...	...	...	
{2,...}	...	...	0	...	...	...	...	...	
{2,...}	...	...	0	...	...	...	...	...	
{2,...}	...	...	0	...	...	...	...	...	
{2,...}	...	...	0	...	...	...	...	...	

**C24.** L'algorithme de Dijkstra permet d'obtenir la distance du chemin le plus court entre 2 et 7. Quel est ce chemin? Combien coûte-t-il? Quelle structure de données pourrait-on utiliser pour trouver ce chemin par l'algorithme de Dijkstra?

On dispose du code qui implémente l'algorithme de Dijkstra et qui produit le tableau des plus courtes distances.

**C25.** Écrire une fonction de signature `s_pp_meme_couleur(depart, d, color_map)` qui renvoie le sommet de même couleur le plus proche de `depart` ainsi que la distance à ce sommet. Le paramètre `d` est le tableau final des distances issu de l'algorithme de Dijkstra invoqué à partir de `depart`. Le paramètre `color_map` est le résultat de la coloration de ce graphe, une liste de couleur propre à chaque sommet : `color_map[i]` renvoie la couleur du sommet `i`.

**C26.** La liste des distances `d` est relative à un graphe comptant 10000 sommets. Chaque distance est codée sur 64 bits. Donner la taille de l'espace mémoire nécessaire pour stocker cette liste en Ko.