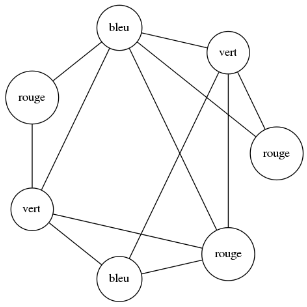
I Coloration d'un graphe par la méthode de Welsh et Powell

1. La liste des sommets ordonnées est `[1, 3, 5, 7, 4, 2, 6]` puis on exécute l'algorithme :
- On colore en bleu 1 puis 4 qui est le seul non lié à 1.
- On colore 3 en vert, 5 et 6 ne sont pas liés à 3 donc on colore 5 en vert aussi puisqu'il a le plus grand degré.
- On colore en rouge 7 puis 2 et enfin 6.



2.
```python
def degre(s,G) :
    return len(G[s])
```

3.
```python
def colorer(G,L) :
    S = trier(G)
    C = L[:]    # on copie L pour éviter de la modifier
    couleur = C.pop()
    dicoColore = {}
    while len(S) > 0 :
        s = S[0]
        S.remove(s)
        dicoColore[s] = couleur
        # on crée une liste annexe des sommets avec cette couleur
        M = [s]
        i = 0
        while i < len(S) :
            a = S[i]
            voisin = False
            # on vérifie que a n'est voisin avec aucun des sommets que l'on a
                déjà colorés de cette couleur
            for j in M :
                if a in G[j] :
                    voisin = True
            if not voisin :
                S.remove(a)
                dicoColore[a] = couleur
                M.append(a)
            # inutile d'incrémenter, la suppression d'un élément décale
            else :
                i += 1
        couleur = C.pop()
    return dicoColore
```

4. La fonction `trier` avec un tri par insertion

```python
def trier(G) :
    L = list(range(1,len(G)+1))
    for i in range(1,len(L)) :
        rg = i
        elt = L[i]
        while rg > 0 and degre(L[rg-1]) < degre(elt) :
            L[rg] = L[rg-1]
            rg -= 1
        L[rg] = elt
    return L
```

II Chasse au trésor sur un graphe

1. Le parcours est $1, 2, 4, 6, 4, 2, 1, 3, 5$ donc de longueur 8 alors qu'il y a plus court : $1, 3, 5, 3, 1, 2, 4, 6$ qui est de longueur 7.

2. a) Le parcours en largeur est un parcours par distance croissante donc qui permet de trouver le sommet le plus proche vérifiant une certaine condition.

b) Dans le code classique du parcours en profondeur, on arrête la recherche dès qu'on a trouvé une nouvelle pièce et on introduit un dictionnaire `origine` pour pouvoir mémoriser le chemin à faire : si `s` est un sommet visité, `origine[s]` est le sommet qui permet d'arriver au sommet `s`

```python
from collections import deque

def suivant(s,G,T) :
    file = deque()
    vus = {c:False for c in G}
    origine = {}
    file.append(s)
    trouve = False
    while not trouve :
        w = file.popleft()
        if not vus[w] :
            vus[w] = True
        for u in G[w] :
            if not vus[u] :
                file.append(u)
                if u not in origine :
                    origine[u] = w
        if T[w] :
            trouve = True
    chemin = [] # on reconstruit le chemin (à l'envers)
    while w != s :
        chemin.append(w)
        w = origine[w]
    return chemin[::-1]  # et on le retourne
```

3. On commence par compter le nombre de pièces pour savoir quand la récolte sera terminée.

```python
def recolte(s,G,T) :
    n = 0
    for c in T :
        if T[c] :
            n += 1
    trajet = [s]
    for k in range(n) :
        L = suivant(s,G,T)
        trajet += L
        s = trajet[-1] # le sommet dont il faut repartir
        T[s] = False # il n'y a plus de pièce sur ce sommet
    return trajet
```