3 - Complexité

Question 3.1 – En combien (en ordre de grandeur, au pire des cas) d'opérations élémentaires vos fonctions de parcours se terminent-elles sur un graphe d'ordre *n* ayant *m* arêtes ?

Cas de la fonction itérative :

Le début de la fonction est l'allocation mémoire d'un parcours p et des listes M, N, L en O(1), ensuite vient la phase d'initialisation des variables int : premier, v et w en complexité constante également O(1).

```
parcours_t * DFS_Iteratif(graph_t * g, int r) {

    parcours_t * p = parcours_init(g->n); /* On initialise un nouveau parcours */

    liste * M = liste_construire(g->n); /* La liste d'exploration M (l'ensemble des sommets visités) */
    liste * N = liste_construire(g->n); /* La liste de fin d'exploration N (l'ensemble des sommets où s'achève l'exploration) */
    liste * L = liste_construire(g->n); /* La pile de parcours L */

    int premier = 0;
    int v, w;
```

Nous arrivons à la boucle principale, essayons de calculer le nombre de fois que la boucle s'effectue dans le pire des cas. Cette boucle se répète lorsque la pile est dépilée, autrement dit elle se répète pour chaque composante connexe du graphe. Dans le pire des cas, le graphe de taille n peut contenir n composantes connexes.

On considère donc que cette boucle se répète n fois. Désormais, étudions le nombre d'opérations élémentaires à l'intérieur même de cette boucle.

```
while(!liste_est_pleine(M)){ /* Tant qu'il reste des sommets non-explorés du graphe (V\M != VIDE --> M != PLEIN) */
    if(premier != 0){
        for(int i = 0; i < g->n; i++){
            if(!liste_contient_element(M,i)){
                r = i; /* Choix du plus petit sommet non-exploré du graphe */
                break;
            }
        }
    }
    premier = 1; /* Ce n'est plus la première racine */
    p->pere[r] = -1; /* Ce sommet n'a pas de père */

    liste_ajouter_fin(L,r);
    liste_ajouter_fin(M,r);

    while(!liste_est_vide(L)){ /* Tant que la pile L n'est pas vide */

        v = *liste_get_fin(L); /* Le sommet de la pile L est dans "v" */
        liste * voisins_v = voisins(g,v); /* On crée l'ensemble des voisins du sommet v */

        if(inclusion(voisins_v,M) == 1){ /* Si il n'y a plus aucun voisin de v à explorer */
            liste_supprimer_fin(L,&v);
            liste_ajouter_fin(N,v);
        }
        else{ /* Sinon il reste des voisins non explorés */
            liste * voisins_v_privee_M = difference(voisins_v,M); /* On crée l'ensemble des voisins de v qui n'ont pas déjà été explorés */
            w = minimum_liste(voisins_v_privee_M); /* On choisit le plus petit voisin non-exploré est dans "w" */
            p->pere[w] = v; /* le père de w vaut v */

            p->aretes[p->nb_Aretes].sommet1 = v; /* On ajoute l'arete (v,w) dans le parcours */
            p->aretes[p->nb_Aretes].sommet2 = w;
            p->nb_Aretes++;

            liste_ajouter_fin(L,w);
            liste_ajouter_fin(M,w);
            liste_detruire(&voisins_v_privee_M);
        }
        liste_detruire(&voisins_v);
    }
}
```

La première partie, contient une boucle en O(n), puis des opérations d'affectations, et des fonctions en complexité constante. Nous arrivons donc à une complexité en O(n²), en combinant la boucle principale et cette nouvelle boucle.

```
if(premier != 0){
    for(int i = 0; i < g->n; i++){
        if(!liste_contient_element(M,i)){
            r = i; /* Choix du plus petit sommet non-exploré du graphe */
            break;
        }
    }
}
premier = 1; /* Ce n'est plus la première racine */
p->pere[r] = -1; /* Ce sommet n'a pas de père */

liste_ajouter_fin(L,r);
liste_ajouter_fin(M,r);
```

La deuxième partie est une nouvelle boucle. L'itération de celle-ci est liée au nombre de voisins que peut atteindre de proche en proche une racine choisie auparavant. Dans le pire des cas, il y a une seule composante connexe de la taille du graphe n. Cependant remarquons, l'incompatibilité de la première boucle dans le pire des cas avec la seconde boucle dans le pire des cas : S'il y a n composantes connexes, il y a nécessairement n composantes connexes de taille n/n = 1. Et s'il y a une composante de taille n, alors il y a une seule composante connexe dans le graphe.

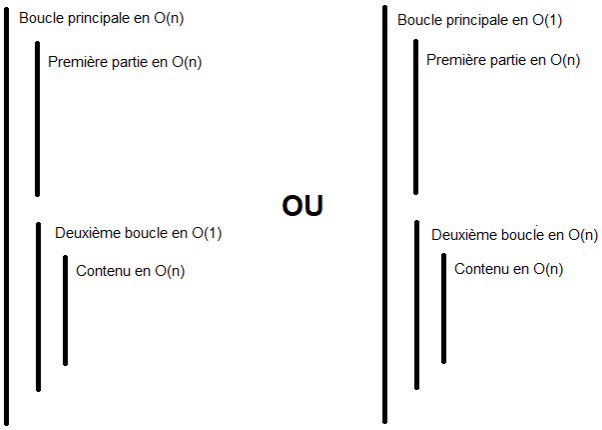
Dans tous les cas, comptons d'abord le nombre d'opérations de cette boucle.

```
while(!liste_est_vide(L)){ /* Tant que la pile L n'est pas vide */

    v = *liste_get_fin(L); /* Le sommet de la pile L est dans "v" */
    liste * voisins_v = voisins(g,v); /* On crée l'ensemble des voisins du sommet v */

    if(inclusion(voisins_v,M) == 1){ /* Si il n'y a plus aucun voisin de v à explorer */
        liste_supprimer_fin(L,&v);
        liste_ajouter_fin(N,v);
    }
    else{ /* Sinon il reste des voisins non explorés */
        liste * voisins_v_privee_M = difference(voisins_v,M); /* On crée l'ensemble des voisins de v qui n'ont pas déjà été explorés */
        w = minimum_liste(voisins_v_privee_M); /* On choisit le plus petit voisin non-exploré est dans "w" */
        p->pere[w] = v; /* le père de w vaut v */

        p->aretes[p->nb_Aretes].sommet1 = v; /* On ajoute l'arete (v,w) dans le parcours */
        p->aretes[p->nb_Aretes].sommet2 = w;
        p->nb_Aretes++;

        liste_ajouter_fin(L,w);
        liste_ajouter_fin(M,w);
        liste_detruire(&voisins_v_privee_M);
    }
    liste_detruire(&voisins_v);
}
```

- Jusqu'au *else*, à part les fonction *voisins* et *inclusion* qui sont en O(n), le reste est en complexité constante.

- Puis, dans le *else,* Les fonctions *difference*, et *minimum* sont en O(n). Le reste est en O(1).

Le contenu de la boucle *while(!liste_est_vide(L))* s'effectue donc en O(n). Comme nous l'avons dit précédemment, si la boucle principale est en O(n) alors cette boucle s'effectue seulement 1 fois, soit la boucle principale est en O(1) et cette boucle s'effectue n fois. Dans les deux cas, en combinant les complexités on obtient O(n²). Ci-dessous, un schéma qui résume le raisonnement.



Total : O(n) * O(n) + O(n) * O(1) * O(n)
= O(n^2) + O(n^2) = O(n^2)

Total : O(1) * (O(n) + O(n) * O(n))
= O(n^2) + O(n) = O(n^2)

En ce qui concerne la fin de la fonction, elle est hors de toute boucle et s'effectue en O(n), ce qui ne change rien à la complexité finale qui est quadratique.

Cas de la fonction récursive :

Phase d'initialisation et d'allocation mémoire en O(1) (comme la fonction itérative)

```
parcours_t *DFS_Recursif(graph_t *g, int r)
{

    parcours_t *p = parcours_init(g->n);

    liste *M = liste_construire(g->n); /* La liste d'exploration M */
    liste *N = liste_construire(g->n); /* La liste de fin d'exploration N */

    int premier = 0;
```

Comme la fonction itérative, ci-dessous la boucle qui se répète pour chaque composante connexe. Dans le pire des cas, cette boucle se répète n fois.

La boucle « for », à l'intérieur, est en O(n) dans le pire des cas. Le reste est constant jusqu'à la fonction *explorer*.

```
while (!liste_est_pleine(M))
{ /* Tant qu'il reste des sommets non-explorés du graphe (V\M != VIDE --> M != PLEIN) */
    if (premier != 0)
    {
        for (int i = 0; i < g->n; i++)
        {
            if (!liste_contient_element(M, i))
            {
                r = i; /* Choix du plus petit sommet non-exploré du graphe */
                break;
            }
        }
    }
    premier = 1;     /* Ce n'est plus la première racine */
    p->pere[r] = -1; /* Ce sommet n'a pas de père */

    explorer(g, r, M, N, p); /* On explore le sommet r (on rappelle la fonction pour chaque composante connexe) */
}
```

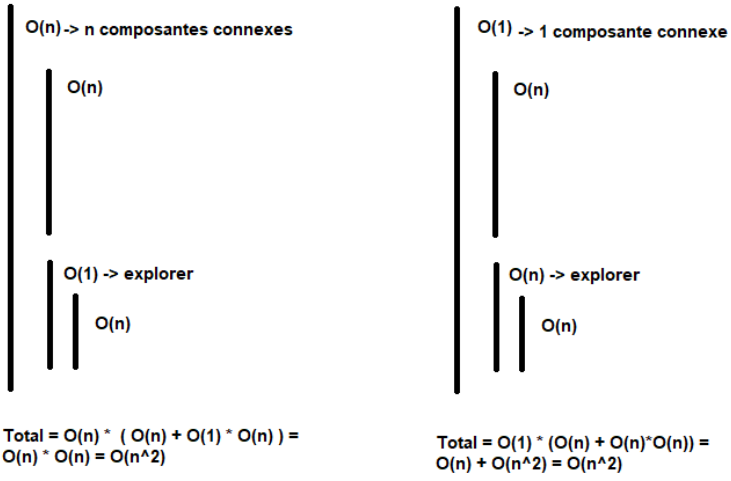
Calculons la complexité de la fonction « *explorer* ».

```c
void explorer(graph_t *g, int s, liste *M, liste *N, parcours_t *p)
{

    liste_ajouter_fin(M, s);

    for (int i = 0; i < g->n; i++)
    { /* On visite l'ensemble des voisins non-explorés de s */
        if (!liste_contient_element(M, i) && g->matrice_adj[s][i] >= 1)
        {
            p->pere[i] = s; /* le père de i vaut s */

            p->aretes[p->nb_Aretes].sommet1 = s; /* On ajoute l'arete (s,i) dans le parcours */
            p->aretes[p->nb_Aretes].sommet2 = i;
            p->nb_Aretes++;

            explorer(g, i, M, N, p); /* Appel récursif, on explore le voisin i de s */
        }
    }

    liste_ajouter_fin(N, s); /* Si il n'y a plus de voisins à explorer on ajoute s à la liste de fin d'exploration N */
}
```

La boucle *for* s'effectue n fois. A l'intérieur de la boucle, la condition est en complexité constante.

La fonction *explorer* est rappelée autant de fois qu'il y a de sommets dans la composante connexe en cours d'exploration, dans le pire des cas n fois. Donc en combinant la complexité de la boucle *for*, on obtient une complexité en $O(n^2)$ pour la fonction « explorer » dans le cas où il y a une seule composante connexe. Mais dans le cas, où il y a n composantes connexes de taille 1, *explorer* est de complexité O(n) car on rappelle explorer une seule fois.

Finalement, comme dans la fonction itérative, ci-dessous la complexité dans les deux cas :



La fin de la fonction, hors de toute boucle est en O(n). Ce qui ne change rien à la complexité finale qui est quadratique.

Question 3.2 – Combien (en ordre de grandeur, au pire des cas) d'octets vos fonctions occupent-elles en mémoires sur un graphe d'ordre n ayant m arêtes ?

Pour calculer l'espace en mémoire, intéressons aux endroits du code où on alloue de la mémoire.

Cas de la fonction itérative :

On alloue de la mémoire au début, pour initialiser le parcours, la liste M, la liste N, et la pile L. L'occupation en mémoire dépend de n qui est la taille du graphe, donc elle est en O(n).

Le prochain endroit où on alloue de la mémoire est pour la liste des voisins de v et pour la liste des voisins de v privée de M dans la boucle *while,* ces allocations sont aussi en O(n) mais sachant que la boucle *while* est en O(n) aussi, on occupe en ordre de grandeur $O(n^2)$.

Finalement, la complexité en espace est donc quadratique.

Cas de la fonction récursive :

Le calcul est rapide : on alloue de la mémoire au début mais nulle part d'autre. Donc la complexité en espace est linéaire, en O(n).

Question 3.3 – S'il faut choisir, recommanderiez-vous une écriture itérative, ou récursive de l'algorithme de parcours en profondeur d'abord ?

D'après nos calculs, la complexité en temps est la même pour nos deux algorithmes, mais la complexité en espace est favorable à l'algorithme de parcours récursif car la pile L n'existe pas dans cette version et on n'alloue pas de mémoire dans une boucle (les fonctions « voisins » et « difference »). Donc le choix se porterait plus vers l'écriture récursive.