# Rapport NF16 TP3 - Gaba Auriane, Chahine Alissa

## Liste des structures et des fonctions supplémentaires

**void insererVoisin(sommet* s, int id_voisin);**
Afin de simplifier la fonction *ajouterArete*, nous avons décidé de créer une fonction qui crée un voisin à partir d'un indice passé en paramètre et qui insère ce voisin dans la liste de voisins du sommet s tout en conservant l'ordre croissant des indices. Ainsi, nous avons seulement à appeler la fonction *insererVoisin* pour les deux sommets entre lesquels nous voulons ajouter une arête afin d'ajouter l'autre sommet dans leur liste de voisins.

**int compte Voisins(sommet* s);**
Afin de simplifier la fonction *rechercherDegre*, nous avons créé une fonction qui retourne le nombre de voisins d'un sommet passé en paramètre. Ainsi, nous pouvons rechercher plus facilement le degré maximum de tous les sommets d'un graphe dans *rechercherDegre* en appelant la fonction *compteVoisins* pour chaque sommet et en retenant le résultat le plus élevé.

**void supprimerVoisin(sommet* s, int id);**
Afin de simplifier la fonction *supprimerSommet*, nous avons créé une fonction qui supprime le voisin (d'indice passé en paramètre) d'un sommet s. Ainsi, nous pouvons supprimer toutes les apparitions d'un sommet dans les voisins des autres sommets d'un graphe lors de l'utilisation de *supprimerSommet*.

**voisin* rechercherVoisin(sommet s, int id);**
Afin de simplifier la fonction *contientBoucle*, nous avons créé une fonction qui recherche le voisin (d'indice passé en paramètre) d'un sommet s. Si le sommet est bien dans la liste de voisins, la fonction renvoie un pointeur sur ce voisin, sinon elle renvoie NULL. Ainsi, nous pouvons vérifier si le voisin de même indice que le sommet existe ou non dans *contientBoucle*.

Complexité des fonctions

**graphe* creerGraphe();**
Le nombre d'itérations est constant donc la complexité de *creerGraphe* est **O(1)**.

**void creerSommet(graphe *g, int id);**
La complexité de cette fonction est donnée par le nombre d'itérations de la boucle *while* qui parcourt la liste des sommets déjà présents dans graphe. Soit *n* le nombre de sommets, la complexité est linéaire, et ainsi, *creerSommet* est de complexité **O(n)**.

**sommet* rechercherSommet(graphe g, int id);**
La complexité de cette fonction est donnée par le nombre d'itérations de la boucle *while* qui parcourt la liste des sommets du graphe. Soit *n* ce nombre de sommets, la complexité est linéaire, et ainsi, *rechercherSommet* est de complexité **O(n)**.

**void insererVoisin(sommet* s, int id_voisin)**
Soit *m* le nombre de voisins et *n* le nombre de sommets du graphe, on a toujours $m<=n$ et dans le pire des cas $m=n$. La complexité de la fonction est donnée par le nombre d'itérations de la boucle *while* qui parcourt la liste des voisins du sommet s du graphe. Donc, la complexité est linéaire, et ainsi, *insererVoisin* est de complexité **O(m) = O(n)**.

**void ajouterArete(graphe *g, int id1, int id2);**
Soit *n* le nombre de sommets, les fonctions *rechercherSommet* (O(n)) et *insererVoisin* (O(n)) interviennent toutes deux dans la fonction et donnent sa complexité. Donc, la complexité de *ajouterArete* est **O(n)**.

**graphe* construireGraphe(int N);**
La complexité va dépendre du nombre de sommets et d'arêtes saisis par l'utilisateur. Dans le pire des cas, chaque sommet est relié à tous les autres et à lui-même (boucle). Pour *n* sommets cela implique que le nombre d'arêtes sera $(n.(n+1))/2$ ce qui correspond à une complexité quadratique. Ainsi, la complexité de *construireGraphe* est **$O(n^2)$**.

**void afficherGraphe(graphe g);**
La complexité dépend de la boucle while qui parcourt et affiche chaque sommet et, pour chaque sommet, parcours tous ses voisins . On note *n* le nombre de sommets. Dans le pire des cas, de même que pour *construireGraphe*, le nombre de voisins est proportionnel à $n^2$ donc la complexité de *afficherGraphe* est **$O(n^2)$**.

**int compteVoisins(sommet* s);**
La complexité est donnée par la boucle while qui parcourt et compte tous les voisins d'un sommet. Soit *n* le nombre de sommets (qui est égal au nombre de voisins dans le pire des cas), la complexité est linéaire. Ainsi, la complexité de *compteVoisins* est **O(n)**.

**int rechercherDegre(graphe g);**
Pour chaque sommet, la boucle while appelle la fonction *compteVoisins* de complexité O(n) avec *n* le nombre de sommets. Encore une fois, pour le pire cas (tous les sommets sont reliés) on le nombre de voisins proportionnel à $n^2$. Ainsi, la complexité de *rechercherDegre* est **$O(n^2)$**.

**void supprimerVoisin(sommet* s, int id);**
Dans le pire cas, la boucle while parcourt toute la liste des voisins du sommet s. Soit *n* le nombre de sommets qui est égal au nombre de voisins dans ce cas, la complexité de *supprimerVoisin* est **O(n)**.

**void supprimerSommet(graphe *g, int id);**
La complexité est donnée par la boucle while qui parcourt chaque sommet et appelle la fonction *supprimerVoisin* de complexité O(n) avec *n* le nombre de sommets. On supprime donc le sommet d'indice id dans les voisins de tous les sommets. Ainsi, comme précédemment, la complexité de *supprimerSommet* est **$O(n^2)$**.

**voisin* rechercherVoisin(sommet s, int id);**
La complexité est donnée par la boucle *while* qui parcourt, dans le pire cas, tous les voisins du sommet s. Donc, soit *n* le nombre de sommets (égal au nombre de voisins dans le pire des cas), la complexité de *rechercherVoisin* est **O(n)**.

**int contientBoucle(graphe g);**
La complexité est donnée par la boucle *while* qui parcourt chaque sommet du graphe et appelle la fonction *rechercherVoisin* de complexité O(n) avec *n* le nombre de sommets. Ainsi, dans le pire des cas, la complexité de *contientBoucle* est **$O(n^2)$**.

**void fusionnerSommet(graphe *g, int idSommet1, int idSommet2);**
Après la fusion, on parcourt tous les voisins de tous les sommets pour y supprimer le sommet avec le plus grand indice. On note *n* le nombre de sommets du graphe. La complexité de la fonction est donnée par celle de *supprimerSommet* qui vaut $O(n^2)$ dans le pire des cas. Ainsi, la complexité de *fusionnerSommet* est **$O(n^2)$**.