1- Choix de représentation

Nous avons décidé de symboliser l'élément de séparation des nombres par un "/" dans le data de l'élément en question.

A noter que dans ce rapport, nous parlons d'un élément comme étant un groupement de 5 chiffres qui représentent une partie d'un nombre. Un nombre complet est donc en ensemble (une chaîne) d'éléments.

2- Fonctions de service

Nous avons créés quelques fonctions de service qui ne sont pas demandées dans l'énoncé, mais qui facilitent la réalisation des fonctions à implémenter.

is_list_empty(List *list) :
Cette fonction permet de déterminer si une liste est vide. Elle prend donc la liste en paramètre et renvoie un booléen.
Nous avons donc importé la librairie stdbool.h pour pouvoir utiliser simplement les booléens.
La complexité de cette méthode est de $\Theta(1)$ car il ne s'agit ici que d'une comparaison (tête_de_liste = queue_de_liste).

cutNumber(char *str, int n) :
Cette fonction est celle que l'on utilise pour découper un nombre représenté dans une chaîne de caractères (premier argument) en une suite d'éléments. Elle renvoie donc un pointeur sur un Element.
Le second argument (n) est un entier que l'on utilise pour symboliser la décalage par rapport au début de la chaîne. Comme nous avons choisi d'implémenter cette fonction de manière récursive, il nous faut préciser à quelle position de la chaîne de caractères nous nous plaçons.
Lors du premier appel de la fonction, on met donc n=0.
La complexité dépend de la taille de la chaîne str, c'est à dire le nombre de chiffres du nombre qu'elle représente. Posons m comme étant ce nombre, on a donc une complexité en $\Theta(m)$.

goToEnd(Element *e) :
Cette fonction récursive renvoie un pointeur sur le dernier élément constituant une chaîne d'élément. On a donc l'argument (e) qui est un pointeur sur le premier élément de la chaîne d'éléments que l'on considère.
La complexité dépend directement du nombre d'éléments de la chaîne considérée. Si on pose n comme étant ce nombre, on a donc une complexité en $\Theta(n)$.

freeElements(Element *e) :
C'est une procédure récursive qui ne revoie rien, elle effectue juste des free sur tous les éléments jusqu'à ce qu'il n'y en ai plus (en partant bien entendu de la fin de la liste).
La complexité dépend encore du nombre d'éléments de la chaîne d'éléments. Si on pose n comme étant ce nombre, on a donc une complexité en $\Theta(n)$.

goToNumber(Element *e, int p) :
L'objectif est de se rendre à la fin du Pième (second argument) nombre représenté par les éléments. Le premier argument (e) est un pointeur sur le premier élément à considérer. C'est à dire qu'on va compter les nombres représentés à partir de cet élément.
La complexité ici dans le pire des cas (si p est le dernier élément de la chaîne d'éléments) est de O(n), avec n étant le nombre d'éléments présents dans la chaîne. Dans le meilleur des cas (p est le premier élément de la chaîne), cela dépend du nombre d'éléments, noté m, qui représentent le nombre. On a donc une complexité en $\Omega(m)$.

countElements(Element *e) :
On considère ici un élément comme étant un nombre, contrairement au reste, où un élément est un groupement de 5 chiffres.
Cette procédure a pour but de compter le nombre de nombres représentés dans une chaîne d'éléments, à partir du pointeur sur un élément passé en argument.
On compte donc le nombre de séparateurs "/" qui représentent la séparation entre 2 nombres.

La complexité dépend de la longueur de la chaîne d'éléments (groupement de 5 chiffres), notée n. On a donc une complexité en Θ(n).

countElementsOfNumber(Element *e) :
On retourne ici le nombre d'éléments qui composent un nombre, à partir d'un pointeur sur un élément de ce nombre (l'argument e).
On appelle donc toujours cette méthode sur le premier élément d'un nombre.
La complexité dépend du nombre d'éléments qui composent le nombre, noté n, commençant par l'élément e. C'est donc une complexité en Θ(n).

reconstruct(Element *e) :
C'est la méthode inverse de **cutNumber()**, mais elle est faite de façon itérative. Elle permet donc de renvoyer un pointeur sur une chaine de caractère qui reconstitue le nombre à partir d'un élément e (passé en argument). *On prend donc le premier nombre à partir de l'élément e.*
La complexité de cette procédure dépend donc du nombre d'éléments qui constituent le premier nombre, noté n, à partir de e. On a donc Θ(n).

3- Fonctions à implémenter

initialize(List *list) :
Cette procédure prend une liste (vide au départ) et l'initialise (en lui donnant un élément de tête et de queue, qui est donc le même). Elle ne renvoie rien.
Comme ce ne sont que des opérations élémentaires, on a une complexité en Θ(1).

Les 3 prochaines méthodes ne renvoient rien.

insert_empty_list(List *list, char *str) :
On insère le nombre représenté par str dans la liste (initialement vide). On utilise notre fonction **cutNumber()** pour découper ce dernier en éléments. Cette procédure ne renvoie rien.
On définit également la tête et la queue de liste. Pour la queue, on utilise notre méthode **goToEnd()**.
La complexité dépend donc de celle de **cutNumber()** (Θ(m), m le nombre de chiffres que la chaîne str représente) et de **goToEnd()** (Θ(n), n le nombre d'éléments de la chaîne qui constitue le nombre). Comme le nombre d'éléments (groupements de 5 chiffres) dépend directement du nombre de chiffres (m) du nombre représenté dans str, la complexité aussi. Donc on a Θ(m).

insert_begining_list(List *list, char *str) :
Sur le même principe que pour **insert_empty_list()**, on place un nombre découpé à l'aide de **cutNumber()** au début, et on va à la fin de la chaîne d'élément ainsi générée pour placer la suite comme étant l'ancienne tête de liste.
La complexité est la même que pour la méthode précédente (Θ(m), m le nombre de chiffres que la chaîne str représente), car nous dépendons directement des deux mêmes méthodes.

insert_end_list (List *list, char *str) :
Le principe est le même que pour les 2 précédentes méthodes, ainsi que la complexité (Θ(m), m le nombre de chiffres que la chaîne str représente).

insert_after_position(List *list, char *str, int p) :
Encore le même principe, sauf que nous renvoyons 0 si l'insertion s'est bien passée, -1 sinon.
Nous passons utilisons 2 variables qui représentent deux éléments voisins (le next du premier pointe sur le second). Cela nous permet d'insérer le nombre découpé (avec **cutNumber()**) entre ces 2 éléments.
La complexité dépend du nombre, noté n, de chiffres dans str, ainsi que de la position p à laquelle on veut insérer le nombre. On a donc une complexité en Θ(n+p).

removeElement (List *list, int p) :
Nous avons renommé cette méthode pour une question de redondance de nom de méthode (remove était déjà utilisé).
Pour supprimer le Pième nombre de la liste, on commence par se rendre à la position p dans la liste (méthode **goToNumber()**), puis on fait pointer la fin de l'élément représentant le nombre P-1 sur le début de celui représentant l'élément P+1. On effectue ensuite un **freeElements()** sur l'élément qui est maintenant isolé de la liste, afin de le supprimer définitivement.
La complexité dépend donc des procédures **freeElements()** (Θ(n), n le nombre d'éléments de la chaîne d'éléments à supprimer) ainsi que de **goToNumber()** (Θ(m), m le nombre de chiffres du nombre considéré).
On donc une complexité en Θ(n+m).

compare(char *str1, char *str2) :
On compare deux chaînes de caractères représentants des nombres pour savoir si l'une est plus grande que l'autre. On retourne un entier qui décrit ce résultat.
On ne peut pas reconvertir les chaînes en entier, car le sujet est que l'on traite des entiers trop grands pour êtres représentés. On compare donc les longueurs des chaînes, et si elles sont égales, ce sont les chiffres qui sont comparés un à un.
La complexité dépend donc de la longueur de la plus grande des deux chaînes, noté n. On a donc Θ(n).

display(List *list) :
Procédure itérative permettant d'afficher le contenu de la liste.
La complexité dépend du nombre d'éléments de la liste, noté n. On a donc Θ(n).

sort(List *list) :
Nous avons choisi d'implémenter un algorithme de tri à bulles. La complexité d'un tel algorithme est normalement en O(n^2). Cependant ici, nous ne pouvons pas directement deux nombres, il nous faut d'abord les reconstituer en chaînes de caractères (**reconstruct()**), puis les comparer (**compare()**).
Pour inverser le i-ème avec le i+1-ème si ils ne sont pas triés, nous supprimons (**removeElement()**) le i+1-ème et nous l'insérons à la position i (**insert_after_position()**).
Pour la complexité, on a donc n*n, n le nombre de nombres représentés dans la liste, pour le tri par bulle classique, mais nous multiplions par le maximum entre les complexité des méthodes énoncées ci-dessus. Ce maximum dans le pire des cas, correspond au nombre d'éléments dans la liste, noté m, multiplié par le nombre de chiffres dans le plus grand nombre, noté p.
On obtient donc une complexité en n*n*m*p.

destruct(List *list) :
Nous utilisons ici la méthode **freeElements()** pour libérer tous les éléments de la liste. Et nous faisons un dernier free pour libérer la liste.
La complexité est donc celle de la méthode citée précédemment. On a donc Θ(n), n étant le nombre d'éléments présents dans la liste.

4- Améliorations et optimisations envisageables

- Optimiser la fonction **sort()** afin de réduire sa complexité.
- Implémenter la méthode bonus : **sum()**