Costa da Quinta, João Filipe
Algorithme - Tp Bonus 2

exo 1.1)

Divide-and-conquer

Imaginons que une liste de n éléments que nous devons trier, c'est le problème, le principe de 'Divide', est de tout simplement rendre ce problème plus petit en réduisant la taille de la liste récursivement, plutôt qu'avoir 1 grand problème, on a plein de petits problèmes plus simples à résoudre séparément, c'est ce qui veut dire 'conquer', ensuite, il faut 'combine' la solution de tous les petits problèmes pour avoir la solution au grand problème de base.

Si on met en lien avec l'algorithme get_major(), 'divide' serait l'appel à reduce_liste() qui nous transforme le grand problème en problème plus petit, 'combine' serait l'appel à is_major() pour vérifier que la solution au petit problème est bien la solution du grand problème de base et finalement 'conquer' est le fait de trouver le majoritaire de la plus petite liste A'=reduce_liste(A).

exo 1.3)

Le pire cas possible pour cet algorithme serait la liste A de taille n=2^x (x appartient à N), où A[i]=A[i+1] pour tout i (A est homogène), par exemple, une liste remplie de 1, en calculant A'=reduce(A), on aura len(A')=len(A)/2 car tout élément de A est égal à 1, sachant que A est de taille n=2^x, len(A') sera forcément un multiple de 2, donc paire, ce qui veut dire qu'il faudra refaire l'appel à recude. L'appel sera fait x+1 fois.

Soit A une liste de taille n= 10^x (x=1,2,3,4,5,6):

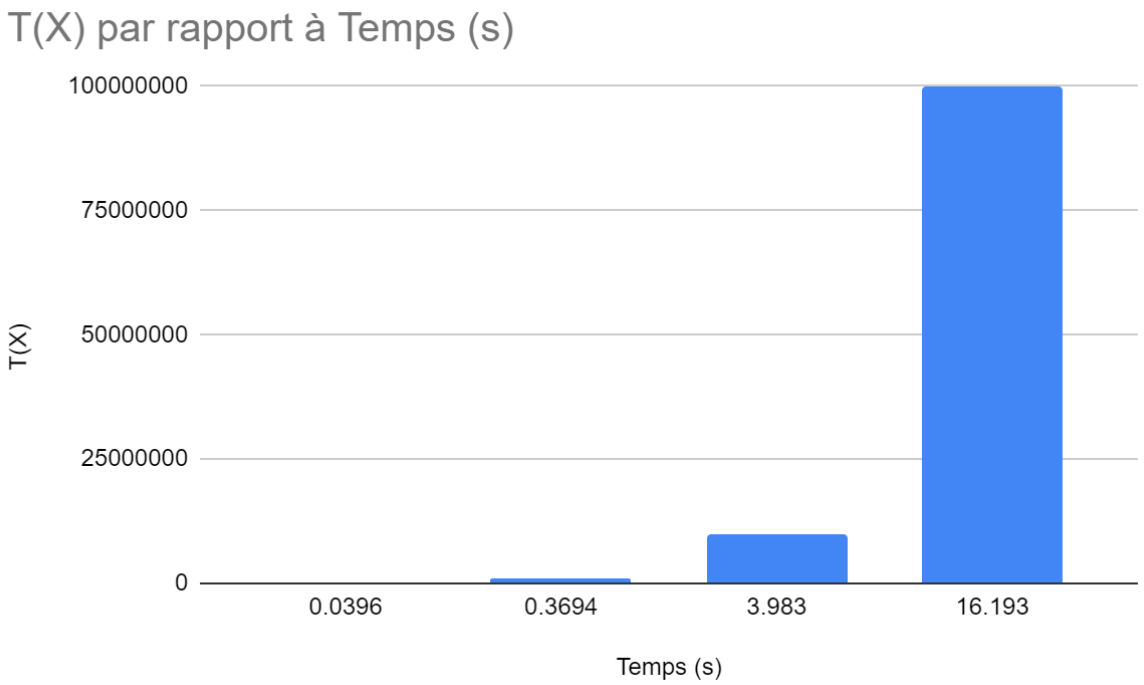
- n=10^1: J'ai eu [3,2,3,2,2,3,2,2,2,3] appels de reduce. (1ere valeur correspond au nombre d'appels reduce pour le premier teste)
- n=10^2: [3,2,3,4,2,4,4,2,3,2] appels
- n=10^3: [4,2,2,3,2,2,3,2,3,2] appels
- n=10^4: [2,3,2,2,4,7,2,2,3,2] appels
- n=10^5: [3,3,2,2,3,2,7,2,3,2] appels
- n=10^6: [4,5,4,2,2,2,2,2,2,3] appels

Le plus intéressant à voir,c'est que pour n=10^6 et pour n=10^2 le nombre d'appels reduce se ressemblent, alors que ca ne serait pas de tout le cas pour l'algorithme naïf, car avec reduce on ne traite pas la liste A, on traite une liste qui fait au plus, la moitié de la taille de A, et en plus est récursif.

Pour une liste sans élément majoritaire j'ai trouvé 2-4 appels à la fonction reduce, c'est donc le même coût que pour le cas avec élément majoritaire, par contre une liste plus grande avait plus souvent plus d'appels que les plus petites listes. Alors qu'avec l'algorithme naïf on aura le plus gros coût possible, car il parcourt toute la liste.

Comme expliqué avant, la longueur de A'=reduce(A), sera au plus égal à A/2, mais si les éléments sont pas homogènes, alors A' sera encore plus petit que la moitié de A, car plus d'éléments seront oublié vu qu'ils sont traités par couples, et gardés que s'ils ont la même valeur. Une liste pas homogène aura plus souvent des couples différents entre eux, et sont donc oubliés dés le premier appel à reduce.

exo 1.4)



C'est le graphique où t(x) représente la taille de la liste A, et le temps(s) représente la moyenne (10 appels chaque) en secondes nécessaire pour calculer le Majoritaire de A, vu que les résultats sont si différents, je n'ai pas pu mettre toutes les donnés dans le même graphique, mais j'envoie aussi un tableau avec tous les informations.

Ici on a le tableau avec plus de données, le temps pour les calculs T(10), T(100) étaient hyper petits dans l'ordre de 0.1*10^-6(s).

T(X)	Temps (s)
1000	0.0004
10000	0.0040
100000	0.0396
1000000	0.3694
10000000	3.983

100000000	16.193

Ces résultats m'ont beaucoup surpris, je m'attendais à des meilleurs temps de calcul pour algorithme get_major(), surtout que l'algorithme naïf a des temps de calcul moins importants, ces données peuvent être du à plusieures choses, déjà ma fonction get_major() n'est peut-être pas optimale, ou alors que les listes A ne sont pas parfaitement aléatoires, imaginons que les valeurs majoritaires sont plus au début de la liste, alors l'algorithme naïf a un avantage incroyable dû au fait qu'il a pas besoin de parcourir toute la liste, ou tout simplement dû à la puissance de calcul de mon ordinateur.

Dans le cas où la liste A n'a pas de valeur majoritaire nous gagnons beaucoup de temps avec l'algorithme get_major(), car même pour une grande liste, le nombre d'itérations reduce_liste() est très petit, alors que l'algorithme naïf doit parcourir toute la liste.

Imaginons la liste A=[1,2,1,2,....,1,2,1], avec len(A)=101, alors l'algorithme naïf va parcourir toute la liste, alors que get_major() va tout simplement prendre la dernière valeur et vérifier si c'est la valeur majoritaire. Si A=[1,2,1,2,...,1,2] avec len(A)=100 reduce_liste() est appelé qu'une fois, A'=[], et A aura pas de valeur majoritaire, alors que le naïf va encore parcourir toute la liste.