Algorithmique:
Correction: Divide-and-Conquer 1

1 Element majoritaire

1.1 Divide-and-Conquer (1 points)

Le 'divide' correpond à la division des éléments par couples, le 'conquer' à leur comparaison, et le 'combine' à l'établissement de la liste qui contient les éléments appartenant à des couples 'doublons'.

1.2 Implémentation (3 points)

Se référer au fichier *majoritycorrection.py*

1.3 Nombre d'étapes (2 points)

Le nombre d'étapes nécessaires semble ne pas dépendre de la taille de la liste. Cela peut se comprendre de la manière suivante : dans le cas d'une liste mélangée homogénement, la probabilité, lorsqu'on prend un élément au hasard, que cet élément soit l'élément majoritaire, est par définition supérieure à 1/2. Or, dès qu'une liste, suite à un appel à reduce, est de longueur impaire, on vérifie si l'un de ses éléments est l'élément majoritaire, afin de l'éléminer et de ramener la longueur de la liste à un nombre pair. D'un autre côté, l'algorithme naïf fait uniquement cela : vérifier, pour chaque élément, s'il est majoritaire. Ainsi on comprend aisément que dans le cas de liste homogènes, l'algorithme Divide-and-Conquer ne présente pas d'avantage ; il est même sensiblement moins rapide.

Cependant, considérons maintenant une liste inhomogène : par exemple $A = [1, 2, 3, 4, 4, 4, 4]$. L'algorithme naïf va vérifier $n/2$ éléments avant de trouver l'élément majoritaire. Pour chacune de ces vérifications, il va compter le nombre d'occurences de l'élément considéré dans la liste, ce qui lui demandera n opérations. La complexité de l'algorithme naïf est donc $O(n/2 \cdot n) = O(n^2)$. L'algorithme Divide-and-Conquer ne souffre pas de ce défaut puisque les appels à reduce transforment la liste. On peut, enfin, considérer le cas où il n'y a pas d'élément majoritaire : la discussion est alors la même que ci-dessus.

2 Exponentiation

2.1 Algorithme naïf

L'algorithme le plus simple est implémenté au travers de la fonction `naive_pow(x, n)` dans le fichier *exponentiation.py*. Il consiste à multiplier n fois x par lui-même. Cela peut être fait par une fonction récursive ou avec une boucle : dans les deux cas, le nombre de multiplications à effectuer est évidemment de n.

2.2 Exponentiation rapide

Premièrement, notons que le nombre de bits nécessaire pour écrire n en base 2 vaut $\lfloor log_2 n \rfloor + 1$. On peut donc écrire n comme :

$$n = \sum_{i=0}^{\lfloor log_2 n \rfloor + 1} a_i \cdot 2^i,$$

avec a_i des coefficients binaires (0 ou 1), alors on a :

$$x^n = \prod_{i=0}^{\lfloor log_2 n \rfloor + 1} \left(x^{2^i}\right)^{a_i}.$$

La complexité en temps du calcul des x^{2^i} est de $O(log_2 n)$, et la combinaison des différents facteurs de l'expression ci-dessus également, ce qui donne une complexité en demps de $O(log_2 n)$.

L'algorithme divide-and-conquer découlant de cette vision des choses est le suivant :

```
fonction quick_exponant(x,n):
if n == 1:
     return x
    elif n%2 == 0:
        return quick_exponant(x*x, n/2)
    else:
     return x*quick_exponant(x*x, (n-1)/2)
```

Si n est pair, alors on 'descend' d'un bit, en appellant la procédure pour $n/2$, avec x^2 en input, tandis que si elle est impair, on se ramène au cas pair en multipliant x par le résultat de exponentiation-rapide(x^2, (n-1)/2). La fonction `fast_pow` du fichier *exponentiation.py* donne une implémentation de cet algorithme.