Algorithmique:
Divide-and-Conquer 1

Bonus : l'exercice 1 de cette série (élément majoritaire) fait l'objet d'un bonus. Seul les travaux respectant exactement les spécifications pour l'input et l'output seront pris en compte. Il en va de même pour le format du rendu (rapport dactylographié au format .pdf, script dans un fichier .py et noms des fichiers) et la version de Python (> 3.0) utilisée.

Pour résoudre un grand problème, on le divise en petits problèmes plus simples. Cette approche comprend trois étapes :

1. **Diviser** le problème en sous-problèmes moins complexes.
2. **Régner** sur les sous-problèmes en les résolvant.
3. **Combiner** les solutions pour obtenir une solution globale.

1 Element majoritaire

Cet exercice concerne le problème de l'élément majoritaire vu en cours. Dans ce problème, on dispose d'une liste A contenant n éléments. On dit que l'élément m est majoritaire dans A s'il constitue *plus* de la moitié des éléments de A. Ici, on veut implémenter un algorithme de type Divide-and-Conquer pour trouver, s'il existe, l'élément majoritaire d'une liste. Pour plus de détails sur la théorie, se référer au cours.

1.1 Divide-and-Conquer (1 points)

Expliquez brièvement quelles parties de la méthode vue en cours correspondent respectivement au 'divide', 'conquer' et 'combine'.

1.2 Implémentation (3 points)

Implémentez l'algorithme en Python. Vous devez définir une fonction `get_major(A)`, qui prend la liste `A` en argument et retourne la valeur de son élément majoritaire s'il y en a un, rien (`None`) sinon). `A` est une liste de nombres comprenant plus de deux éléments, et pouvant contenir ou non un élément majoritaire. Pour un code propre, veuillez également définir la fonction `is_major(A, el)`, qui retourne `True` si `el` est majoritaire dans `A` et `False` sinon, ainsi que la fonction `reduce_list(A)`, qui correspond à la procédure 'reduce' vu en cours pour ce problème. Vous trouverez sur Moodle un fichier `aidemajority.py` contenant l'en-tête de ces fonctions (à compléter) ainsi qu'une fonction générant des listes aléatoires ayant (ou non) des éléments majoritaires, et une implémentation d'un algorithme naïf pour la résolution de ce problème.

1.3 Nombre d'étapes (2 points)

En lancant l'algorithme pour une taille de liste $n \in \{10^1, 10^2, 10^3, 10^4, 10^5, 10^6\}$, et en comptant à chaque fois le nombre d'itérations nécessaires (nombre d'appels à `reduce`), qu'en déduisez-vous sur le gain de cet algorithme par rapport à l'algorithme naïf ? Faites également le test pour des cas où il n'y a pas d'élément majoritaires, ou encore dans des cas où les éléments ne sont pas mélangés de manière homogène.

1.4 Temps d'exécution (2 points)

La fonction `clock()` du module `time` permet d'obtenir le temps écoulé depuis l'importation du module (Unix) ou depuis le premier appel à la fonction (Windows). Ainsi, le code suivant stocke dans la variable `temps_total` le temps d'exécution de la fonction `ma_fonction` :

```
import time
begin = time.clock()
ma_fonction()
end = time.clock()
temps_total = end - begin
```

De cette manière, chronométrez votre algorithme en fonction de la taille de la liste. Comme les listes sont aléatoires et que le temps d'exécution dépend, dans le cas non-naïf, des éléments de la liste, faites des moyennes. Par exemple, le temps d'exécution $T(1000)$ pour une liste de taille 1000, peut être calculé en moyennant sur 10 essais :

```
lists = [generate_maj_list(1000) for i in range(10)]
begin = time.clock()
for i in range(10):
major = get_major(lists[i])
end = time.clock()
temps_moyen = (end - begin) / 10.
```

Produisez une courbe $T(n)$ pour l'algorithme non-naïf, avec par exemple $n \in \{10^1, 10^2, 10^3, 10^4, 10^5, 10^6\}$ (adaptez n à votre machine : il faut que la valeur de $T(n)$ soit de l'ordre de la secondes pour les grandes valeurs de n). Que concluez-vous sur la complexité en temps de l'algorithme ? Le gain est-il appréciable par rapport à la version naïve ? Encore une fois, refaites le test pour des listes où il n'y a pas d'élément majoritaire.

Cet exercice compte pour les bonus. Pour obtenir les points de bonus, vous devrez, pour le mercredi 17 octobre 10 :00 au plus tard, rendre les fichiers suivants (avec votre nom et votre prénom à la place de 'NomPrenom') :

- Un fichier NomPrenom.pdf répondant à toutes les questions de l'exercice.
- Un fichier python (NomPrenom.py) de votre implémentation de l'algorithme. N'envoyez pas le code nécessaire pour les questions 1.4 et 1.5. N'envoyez pas le bytecode (fichier .pyc).

Les fichiers doient être soumis sur Moodle dans le devoir 'Bonus 2'.

NB : Pour produire le graphique demandé à la question 1.5, vous pouvez utiliser la méthode de votre choix. Le librairie *matplotlib* permet néanmoins de créer rapidement des graphiques en python. **Les graphiques doivent avoir un titre ainsi que des axes nommés et avec des unités le cas échéant !**

2 Exponentiation

Nous nous intéressons dans cet exercice aux algorithmes capables de calculer des puissances entières quelconques de la forme x^n.

2.1 Algorithme naïf

Proposez l'algorithme le plus simple et le plus naïf pour le calcul d'une puissance entière (multiplier n fois x par lui-même). Combien d'opérations cela demande-t-il ?

2.2 Exponentiation rapide

En se basant sur l'observation que n peut être réécrit comme $n = \sum a_i \cdot 2^i$ avec a_i binaire (0 ou 1), trouver un algorithme qui fait passer le nombre d'opérations requise à l'ordre logarithmique.