Partie 1

Diviser pour regner :l'algorithme d'exponentiation

$$y = x^n$$

L'exponentiation consiste à trouver une méthode pour calculer x à la puissance n, SANS utiliser l'opérateur *puissance*. L'idée est de se rappocher de l'algorithme utilisé par le processeur d'un ordinateur, qui n'utilise que les 3 opérateurs de base pour effectuer les calculs (+,-,*).

1.1 Programme récursif

```
def exp2(n,x):
    """
    n : entier
    x : reel
    exp2 : reel
    """
    if n==0 : return 1
    else : return exp2(n-1,x)*x
```

Question : montrer que la complexité est en O(n).

1.2 Exponentiation rapide : application de la méthode diviser pour regner

Comme de nombreux algorithmes utilisant cette méthode, celui-ci fait des appels recursifs. Mais à la différence du précédent, l'appel recursif se fait avec un paramètre que l'on divise par 2 (le paramètre n). C'est ce qui fait que le nombre d'appels récursifs est plus réduit.

On retrouve l'étape 3 évoquée en introduction (la combinaison des sous problèmes) lorsque l'on réalise l'opération : `return y*y` ou bien `return x*y*y`.

```
def exp3(n,x):
    """
    programme plus efficace que le precedent car
    le nombre d'operations est log2(n)
    """
    if n== 0 : return 1
    else :
        y = exp3(n//2,x)  # on prend la valeur inferieure de n/2
        if n%2==0:
            return y*y
        else : return x*y*y
```

Prenons pour exemple n = 8 :

- Dans la phase de descente : exp3(8,x) appelle exp3(4,x) appelle exp3(2,x) qui appelle exp3(1,x) puis exp3(0,x).
- Puis dans la phase de remontée :
  - exp3(0,x) retourne 1

- exp3(1,x) retourne ...
- exp3(2,x) retourne ...
- exp3(4,x) retourne ...
- exp3(8,x) retourne ...

On peut illustrer la construction du resultat à l'aide de l'arbre suivant :

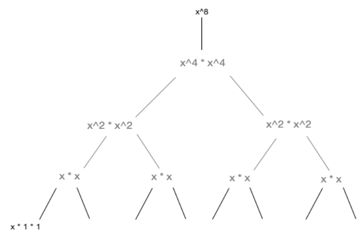

FIGURE 1 – illustration de l'exponentiation rapide

Question : On cherche à représenter le tracé des appels recursifs pour cette fonction.

a. Quelles sont les branches parcourues (les repasser sur le schéma) ?

b. L'arbre est-il parcouru en entier ?

c. Quelle est la complexité temporelle de l'algorithme de l'exponentiation rapide ?

Partie 2

Etude du tri fusion sur la liste `L = [1,10,8,4,3,6]`

Question 1 : Compléter la séquence avec l'ordre des branches parcourues et les sous-listes à chaque noeud, jusqu'à ce que tout le sous-arbre gauche soit "divisée".

Question 2 : Compléter le script de la fonction **`fusion`** qui trie une liste par fusion de manière recursive. Bien observer le schéma ci-dessous qui trace la suite des appels recursifs lors de la division de la liste :

```python
def fusion(L):
    if len(L) <= ... :
        return L
    m = len(L)//2
    gauche = ...
    droite = ...
    return interclassement(gauche,droite)
```

La **remontée** dans la pile d'appels commence lorsque l'on arrive à une sous-liste d'un seul élément pour *droite*.

La **fonction `interclassement`** prend deux listes en paramètres, L1 et L2, et retourne une seule liste avec les éléments de L1 et L2, mais classés.

Question 3 : Décrire les étapes de la remontée (interclassement) jusqu'à ce que la liste L soit triée.

La **complexité** de la fonction `interclassement` est O(n), où n est égal à la taille de chaque sous-liste.

Question 4 : Expliquer pourquoi, lors de la fusion, la complexité au rang n est donnée par la formule de recurence :

$$T(n) = T(\frac{n}{2}) + n$$

Question 5 : En déduire la complexité du tri fusion.

Partie 3

Annexe

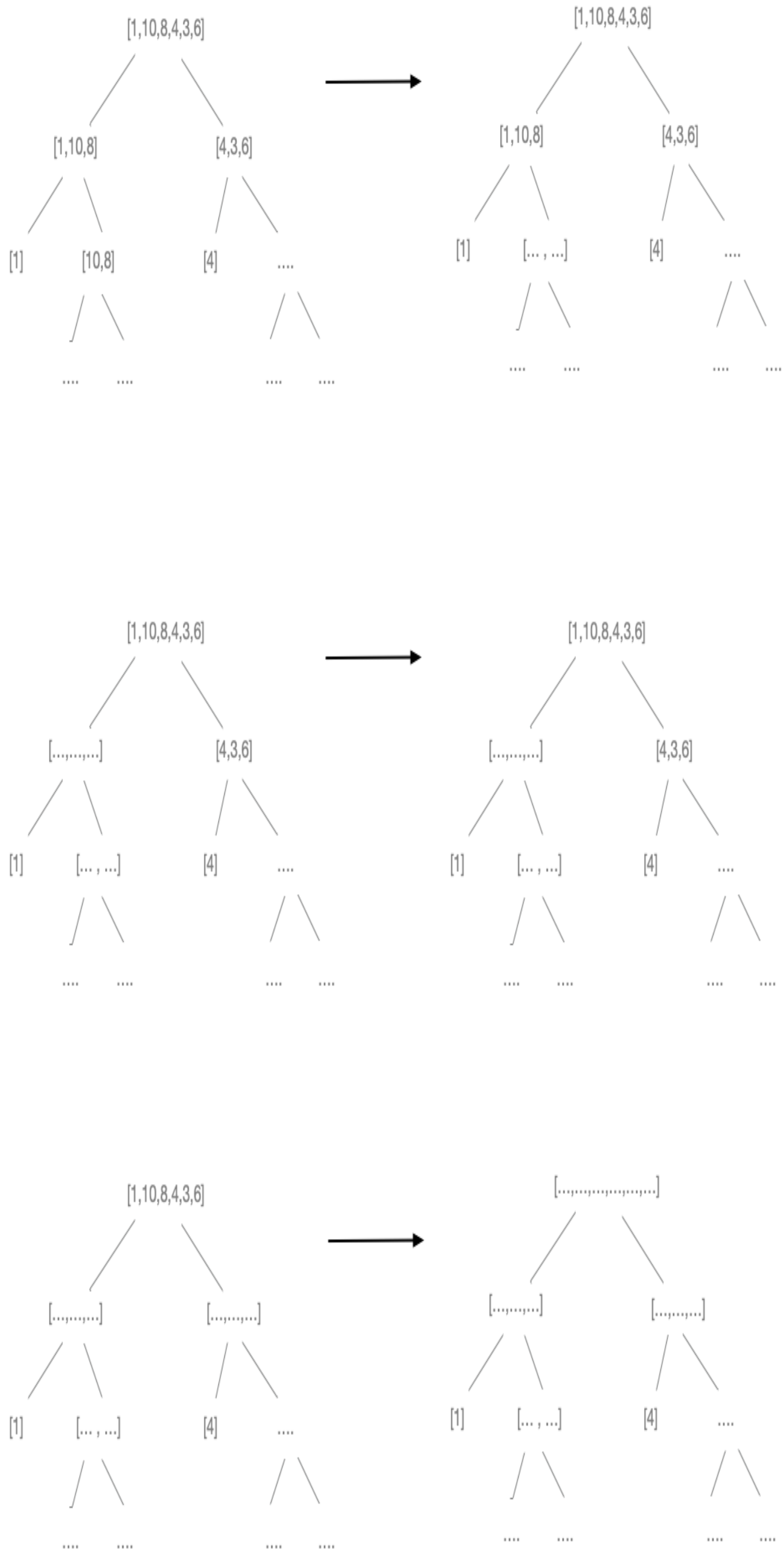

Figure 2 – les étapes du tri fusion