Rappel : Afin d'évaluer le temps d'exécution d'un algorithme, on pourra importer la fonction time du paquet time. La fonction time() ainsi importée renvoie le temps en seconde. En faisant la différence entre le temps après l'exécution et celui avant l'exécution, on peut déterminer le temps mis par l'algorithme en seconde.

0.1 Tri Fusion

Nous allons commencer par implémenter le tri fusion sur les listes afin de bien assimiler la technique.

1. Écrire une fonction merge(liste1,liste2) qui avec deux listes triées en entrée, retourne une liste triée des éléments des deux listes.
2. Écrire une fonction split(l) qui découpe une liste l en deux sous-listes de longueur égales (à un élément près si la longueur est impaire).
3. Écrire une fonction merge_sort(l) qui utilise le principe du « diviser pour régner » et les deux fonctions précédentes.
4. Écrire une fonction random_numbers(n,K) qui génère des listes aléatoires d'entiers de longueur n avec pour valeur maximale K
5. Écrire une fonction one_merge_time(l) qui prend en paramètre une liste l et retourne le temps nécessaire pour que le tri par fusion soit effectué.
6. Écrire une fonction merge_time(n,t) qui prend en paramètre un nombre entier n et un nombre entier t, génère n listes de taille t puis retourne le temps moyen d'un tri fusion pour ces listes. Utiliser les fonctions écrites précédemment

0.2 Exponentiation rapide

L'opération $x \longrightarrow x^n$ peut être très coûteuse si on l'implémente naïvement. Nous allons étudier une version « diviser pour régner » plus efficace, appelée « exponentiation rapide ».

Objectifs : Étant donné un exposant e, on souhaite calculer le plus efficacement possible x^e, c'est-à-dire en minimisant le nombre de multiplications, pour gagner en temps d'exécution.

En effet, pour calculer X^8, on peut :

1. Calculer $X \times X \times ... \times X$, ce qui fait 7 multiplications. La liste des exposants calculés est : 1, 2, 3, 4, 5, 6, 7, 8.
2. Calculer : $Y = X \times X$, ce qui correspond à X^2, puis $Z = Y \times Y$, ce qui correspond à X^4, enfin $T = Z \times Z$, ce qui donne le résultat. On obtient donc le résultat $T = X^8$en seulement trois multiplications. La liste des exposants est : 1, 2, 4, 8.

L'écart se creuse très rapidement, entre la méthode naïve et les méthodes plus évoluées. Ainsi, pour calculer X^{1024}, il faut 1023 multiplications, pour la méthode naïve et 10 multiplications pour la méthode dite exponentiation rapide.

0.2.1 Présentation

L'exponentiation rapide est une technique classique, utilisée en pratique, pour obtenir la puissance d'un nombre donné.

Cette exponentiation rapide itère, pour le calcul de Xe, les opérations suivantes :

1. l'élévation au carré
2. la multiplication par X dans un ordre dépendant de l'écriture en base 2 de e

0.2.2 L'algorithme

Plus précisément...

1. On écrit e en base 2,
2. On enlève le premier 1 de cette écriture,
3. On remplace alors, dans cette écriture, les 0 par des S, les 1 par des SX, on obtient ainsi un mot constitué des lettres S et X.
4. On part du nombre X Si la première lettre est S, on élève au carré, sinon, on multiplie par X
5. On recommence avec les lettres suivantes, jusqu'à la fin du mot.

0.2.3 Version Simple

1. Commençons par écrire une version naïve de l'exponentiation, puissance_naive telle que puissance_naive(x,n) calcule x^n en $O(n)$ multiplications.
2. Ecrire une fonction binaire(n) qui prend un entier n en entrée et renvoie son écriture en binaire sous forme de chaîne de caractères.
3. Ecrire une fonction exponentiation_rapide(x,n) qui calcule x^n en utilisant le principe de l'algorithme décrit ci-dessus.
4. Ecrire une fonction comparaison_temps(x,n,k) qui renvoit le temps moyen de k exponentiations de x puissance n pour la méthode naïve et la méthode d'exponentiation rapide.

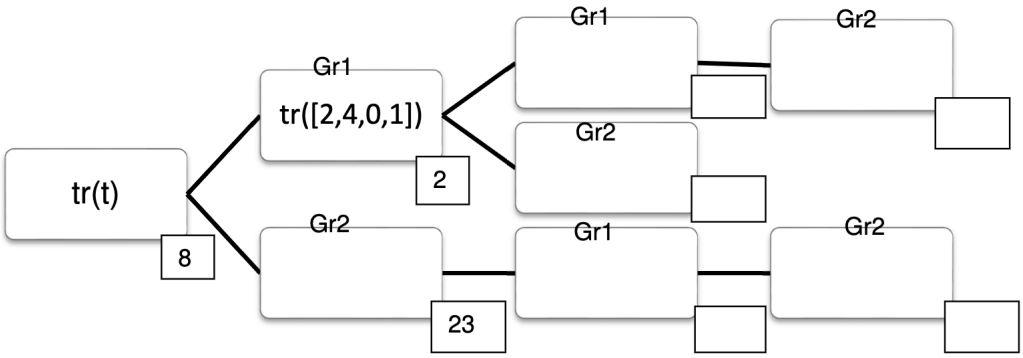
0.3 Quick Sort

0.3.1 L'algorithme

```python
def tri_rapide(lst):
    if len(lst) <= 1:
        return lst

    else:
        Gr1 = []
        Gr2 = []
        pivot = lst[-1]

        for elt in lst[:-1]:
            if elt <= pivot:
                Gr1.append(elt)
            else:
                Gr2.append(elt)

        # Résolution récursive
        Gr1 = tri_rapide(Gr1)
        Gr2 = tri_rapide(Gr2)

        # Renvoi des morceaux
        return Gr1 + [pivot] + Gr2
```

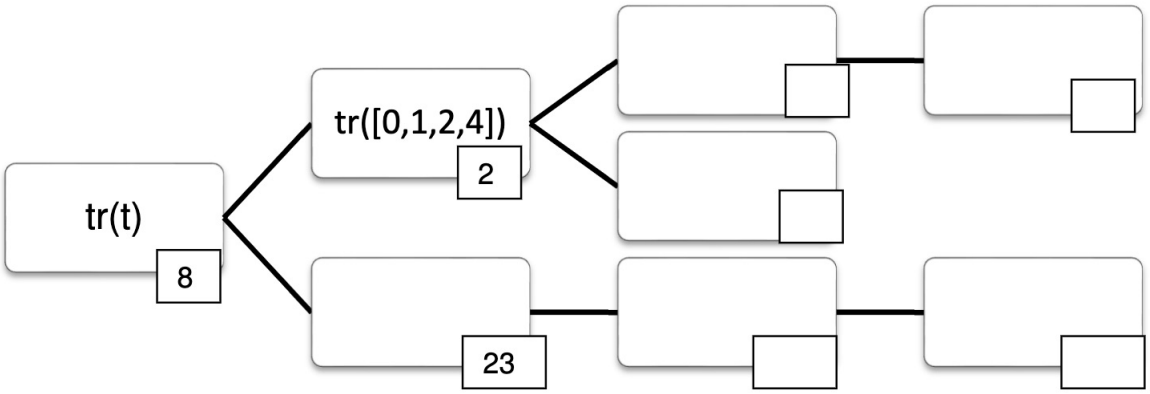
0.3.2 Compréhension de l'algorithme de tri rapide

1. Après avoir lu l'algorithme de tri rapide, dérouler étape par étape l'algorithme de tri rapide de la liste $t = [8,2,4,23,15,17,0,1]$.
On complètera pour cela l'arbre des appels récursifs dans lequel on utilisera la notation tr() pour évoquer l'appel à la fonction *tri_rapide*(). Le respect de l'ordre des valeurs dans les partitions est impératif dans la liste d'entrée de tr(). On indiquera dans le rectangle inférieur droit de chaque appel le pivot qui sera utilisé et qui est retiré des éléments à trier. Les listes vides (Gr1 ou Gr2) ne sont pas représentées.



2. Même question que précédemment mais avec la liste quasi-triée t=[0,1,2,4,8,15,17,23]

3. Même question que précédemment mais avec la liste quasi-triée t=[0,1,2,4,8,15,23,17] pour un pivot pris à la médiane de la liste.



4. Conclure sur le choix du pivot probablement le plus rapide sur des listes quasi-triées.

0.3.3 Implémentation

1. Implémenter la fonction dans Spyder.
2. Modifier la fonction tri_rapide afin de choisir aléatoirement le pivot (on pourra importer la fonction randint de la bibliothèque random et utiliser la fonction $randint(a,b)$, qui choisit aléatoirement un entier dans [a, b], ATTENTION le dernier indice est inclus)
3. Evaluer l'effet de cette modification d'algorithme avec une liste triée de 1000 éléments en mesurant le temps d'exécution de l'algorithme.
4. Comparer les temps nécessaires au tri rapide sur une liste quasi triée selon le choix du pivot (premier élément, médian, aléatoire et dernier élément).

position du pivot	1er indice	médian	aléatoire	dernier indice
durée (en s)	n=1000 →	n=1000 → n=10000→ n=10^6 →	n=1000→ n=10000→	n=1000 →