L'objectif du TP est de présenter quelques algorithmes dichotomiques, qui font partie des algorithmes basés sur le principe « diviser pour régner » (ou divide and conquer en anglais). Ces algorithmes ont pour avantage d'être plus performants que les algorithmes naïfs. Leur force est de ramener la résolution d'un problème dépendant d'un entier n à la résolution de un ou plusieurs sous-problèmes identiques portant sur des entiers n' strictement inférieur à n. Ce TP est composé de 2 parties indépendantes. Dans la première, nous allons travailler sur la recherche dichotomique d'un élément dans une liste triée alors que dans la deuxième nous calculerons de grandes puissances d'un flottant.

Partie I : Recherche dichotomique dans un tableau trié

Le but de cette partie est de trouver si un entier `val` est présent dans une liste triée `L`.
On considère ici que `L` est composée de `n` entiers inférieurs à `M`.

1. Implémenter `L` pour `n=10**3` et `M=10**9` comme suit `L=[randint(0,M) for i in range(n)]` sans oublier d'importer la fonction `randint` via `from random import randint`.
   Trier ensuite la liste à l'aide de la méthode `L.sort()`.
2. Proposer une fonction `recherche_naive(L,val)` prenant en argument une liste triée `L` et une valeur `val` à rechercher dans celle-ci qui renvoie `True` si la valeur est présente et `False` sinon.
   On commence par réaliser un algorithme naïf qui parcourt l'ensemble des valeurs de la liste.
3. On peut réaliser quelques optimisations, en effet la liste étant triée, on peut sortir de la boucle de recherche dès que la valeur de la liste est strictement supérieure à la valeur recherchée.
   Mettre en place ces optimisations puis proposer un jeu de tests unitaires pour valider votre algorithme.
4. On propose à présent d'étudier la *recherche dichotomique*.
   La liste étant triée, l'idée est de tester si le terme du milieu de la liste est inférieur strictement, supérieur strictement ou égal à la valeur recherchée. S'il est égal, on a trouvé la valeur et l'algorithme renvoie `True`, sinon s'il est strictement inférieur (resp. supérieur), on cherche la valeur dans la demi-liste de droite (resp. de gauche). On itère jusqu'à ce que l'intervalle de recherche soit vide. On peut constater que la taille de ce dernier est divisée par deux à chaque itération.
   Proposer une fonction `recherche_dicho(L,val)` basée sur cette méthode en utilisant une boucle `while` de condition `ind_max-ind_min>=0` où `ind_min` et `ind_max` représentent respectivement l'indice minimal et maximal de l'intervalle de recherche.
5. Définir un jeu de tests unitaires pour valider votre algorithme.
6. On considère à présent une liste de taille plus significative, par exemple pour `n=10**6`. On s'intéresse aux temps de calcul des algorithmes ci-dessus. Le code suivant permet d'afficher le temps de calcul de votre `fonction` :

   ```
   from time import time
   start=time()
   fonction(L,val)
   end=time()
   print(end-start)
   ```

   Comparer les temps de calcul des recherches naïve, dichotomique et celle interne de python : `val in L`.
   On remarque que la méthode dichotomique est la plus efficace. Dans le pire des cas, si la valeur n'est pas dans la liste, l'algorithme naïf parcourt la boucle `for` entièrement et donc effectue un nombre de comparaisons proportionnel à n où n est la taille de la liste. Par ailleurs, l'algorithme dichotomique va diviser la taille de la liste en deux jusqu'à atteindre une liste vide. En notant k le nombre de fois où la boucle `while` est appelée, on a $2^k \leq n$ i.e. $k \leq \log_2(n)$. Le nombre de comparaisons dans le pire des cas est donc proportionnel à $\log_2(n)$ dans la méthode dichotomique.
   On dit que la complexité de l'algorithme naïf est linéaire alors que celle de l'algorithme dichotomique est logarithmique.

7. ***Pour les plus rapides :*** en modifiant les fonctions `recherche_naive` et `recherche_dicho` à l'aide d'un compteur, proposer deux nouvelles fonctions `complexite_dicho` et `complexite_naive` renvoyant le nombre de comparaisons réalisées lors de ces recherches.

8. ***Pour les plus rapides :*** on souhaite comparer `complexite_dicho` et `complexite_naive` en fonction de la taille de la liste en argument. Mettre en place une série d'instructions générant deux listes : `C_naive` et `C_dicho`, contenant les nombres de comparaisons dans le pire des cas lors de la recherche dans une liste de taille i pour i allant de 1 à 100.
   Représenter ces deux listes à l'aide des instructions suivantes :

```
import matplotlib.pyplot as plt
plt.clf()
plt.plot([i for i in range(1,10**2)], C_naive)
plt.plot([i for i in range(1,10**2)], C_dicho)
plt.axis('scaled')
plt.show()
```

9. ***Pour les plus rapides :*** Mettre en évidence la complexité logarithmique de l'algorithme dichotomique.

Partie II : Exponentiation rapide

Le but de cette partie est de calculer de manière efficace x^y où x est un flottant et y un entier naturel.

1. Proposer une fonction `puissance_naive(x,y)` prenant en argument un flottant `x` et un entier positif `y`, et qui renvoie la valeur de x^y. On commence par réaliser un algorithme naïf sans utiliser `**`.
2. On propose à présent d'étudier l'exponentiation rapide.

   Notons $\boxed{(y_k)_{k\in[\![0;n]\!]} \text{ les chiffres (0 ou 1) de l'écriture binaire de } y}$. Alors $y = \sum_{k=0}^{n} y_k 2^k$ d'où $x^y = \prod_{k=0}^{n} \left(x^{2^k}\right)^{y_k}$.

   En notant $a_k = x^{2^k}$, il se trouve que $\boxed{a_0 = x \text{ et que } a_{k+1} = a_k^2}$.

   Le principe de la méthode de l'exponentiation rapide consiste à calculer pas à pas le produit $\boxed{x^y = \prod_{k=0}^{n} a_k^{y_k}}$.

   A noter que l'on n'effectue aucune puissance dans ce produit car les y_k sont égaux à 0 ou 1.

   Illustrons ce produit sur un exemple, pour $y = 13$, comme $13 = \overline{1101}^2$, on a $y_3 = y_2 = y_0 = 1$ et $y_1 = 0$.
   D'où $x^{13} = x^{1\times2^0+0\times2^1+1\times2^2+1\times2^3} = (x)^1 \times (x^2)^0 \times (x^4)^1 \times (x^8)^1 = a_0^{y_0} \times a_1^{y_1} \times a_2^{y_2} \times a_3^{y_3} = a_0 \times a_2 \times a_3$.

   En vous basant sur cette méthode, proposer une fonction `exp_rapide(x,y)` ayant le même objectif que `puissance_naive(x,y)`. On pourra utiliser la fonction `bin` de python pour l'écriture binaire de y mais on s'interdira encore d'utiliser `**`.
3. Proposer un jeu de tests unitaires pour vos deux fonctions calculant x^y. Comparer ensuite les temps de calcul de vos algorithmes pour $x = 1 + 10^{-7}$ et $y = 10^7$. On pourra également comparer avec `x**y`.

   On remarque que l'exponentiation rapide est plus efficace. En effet, il y a $y-1$ produits à effectuer dans la méthode naïve ($x\times\cdots\times x$) alors qu'il y a moins de $2n$ produits dans l'exponentiation rapide ($a_{k+1} = a_k \times a_k$ et $x^y = a_0^{y_0} \times \cdots \times a_n^{y_n}$). Or $n+1$ correspond au nombre de chiffres dans l'écriture binaire de y, d'où $2^n \leq y$ i.e. $n \leq \log_2(y)$. Ce qui prouve que la complexité de la méthode naïve est linéaire alors que celle de l'exponentiation rapide est logarithmique.

4. ***Pour les plus rapides :*** en modifiant les fonctions `puissance_naive` et `exp_rapide` à l'aide d'un compteur, proposer deux nouvelles fonctions `puissance_naive_c` et `exp_rapide_c` renvoyant le nombre de produits réalisés lors de ces algorithmes.
5. ***Pour les plus rapides :*** on souhaite comparer `puissance_naive_c` et `exp_rapide_c` en fonction de la taille de y.
   Mettre en place une série d'instructions générant deux listes : `L_naive` et `L_exp`, contenant les nombres de produits lors du calcul de x^y pour y allant de 1 à 100. On prendra $x = 1 + 10^{-7}$.
   Représenter ces deux listes à l'aide d'un graphique puis mettre en évidence la complexité logarithmique de l'exponentiation rapide.