TP D'INFORMATIQUE 4
Algorithmes dichotomiques

1 Recherche dichotomique

On a déjà vu qu'il était possible de rechercher si un élément appartient à une liste en coût linéaire : il suffit de parcourir la liste. L'objet de cette partie est de déterminer un algorithme plus efficace lorsque la liste considérée est déjà triée. On va pour cela adopter une approche **dichotomique** : couper le problème en deux autant de fois que nécessaire.

1. On prend l'exemple d'une liste **triée** L contenant 15 entiers. On cherche à déterminer si L contient un entier donné n en accédant au moins de cases possibles de la liste dans le pire cas. On note que parcourir L de gauche à droite jusqu'à trouver n nécessite 15 accès dans le pire cas, lorsque n n'est pas présent dans L. Proposer une méthode pour tester si n est dans L ne nécessitant jamais plus de 4 accès à une case de L.

   Représenter cette méthode par un **arbre de décision** : on écrit l'indice de la première case lue, puis on trace deux branches, l'une pour la suite de l'algorithme dans le cas où n est strictement inférieur au contenu de cette case, l'autre dans le cas où n est strictement supérieur (le cas où n est égal au contenu de la case n'a pas besoin d'une troisième branche puisque l'algorithme s'arrête immédiatement). Chaque branche continue ainsi avec l'indice de la nouvelle case lue, et ainsi de suite, jusqu'au moment où on est certain que l'algorithme se termine.
2. En déduire une fonction python `recherche_dichotomique` prenant en argument une liste L supposée triée et un élément x, et testant si x appartient à L en généralisant la méthode précédente.

   Indication : On utilisera une boucle `while` ainsi que deux variables `g` et `d` contenant les deux indices entre lesquels il est encore possible de trouver x dans L.
3. Plus les programmes écrits sont élaborés, plus il est important de procéder à des tests suffisamment exhaustifs pour s'assurer que le résultat renvoyé est correct dans tous les cas. Un tel ensemble de tests, couvrant les différentes possibilités d'erreurs, est appelé un **jeu de tests**. Recopier le jeu de tests suivant et corriger les éventuelles erreurs détectées :

```
L = [2,4,5,7,11]

assert recherche_dichotomique(L,4), "Un élément de la liste n'est pas reconnu"

assert not recherche_dichotomique(L,6), "Un élément hors de la liste est reconnu"

assert recherche_dichotomique(L,2), "Le premier élément n'est pas reconnu"

assert recherche_dichotomique(L,11), "Le dernier élément n'est pas reconnu"

assert not recherche_dichotomique(L,0), "Un élément inférieur au minimum est reconnu"

assert not recherche_dichotomique(L,12), "Un élément supérieur au maximum est reconnu"

assert not recherche_dichotomique([],12), "Un élément est reconnu dans la liste vide"
```

4. Écrire une variante `indice_dicho` de la fonction précédente renvoyant un indice où x apparaît dans L, ou `None` s'il n'en existe pas. Adapter le jeu de test pour vérifier cette fonction.
5. Dans le cas d'une liste de longueur $2^n - 1$, où $n \in \mathbb{N}$, combien de cases sont lues lors d'une recherche dichotomique dans le pire cas ?

   Remarque : on en déduit que le temps de calcul d'une recherche dichotomique est globalement proportionnel à $\log_2(l)$, où l est la longueur de la liste. On dit que cette fonction a un **coût logarithmique**.

2 Exponentiation rapide

L'objectif de cette partie est de calculer efficacement la puissance entière d'un flottant uniquement à l'aide de multiplications, sans utiliser l'opérateur préexistant `**`.

1. Écrire une fonction `puissance` prenant en argument un flottant x et un entier positif k, et renvoyant x^k au moyen de k multiplication successives.
2. Compléter la fonction précédente en incluant le cas où k est négatif. On pourra utiliser l'instruction `assert x != 0` dans ce cas pour vérifier que x est non nul (et interrompre la fonction avec une erreur sinon).
3. Par principe même, la méthode précédente nécessite k multiplications. Déterminer une méthode permettant de calculer x^{16} au moyen de seulement 4 multiplications.
4. En observant $x^{2n+1} = x^n * x^n * x$, déterminer une méthode permettant de calculer x^{21} en 6 multiplications.
5. Pour traduire cette méthode en fonction Python, nous allons utiliser 3 variables y, n, r, une boucle `while`, et nous baser sur l'**invariant de boucle** suivant :

   $$y^n * r = x^k$$

   Un invariant de boucle est une proposition qui doit rester vraie au début et à la fin de chaque itération de la boucle.

   (a) Quelles doivent être les valeurs initiales de y et n pour que l'invariant soit initialement vrai, sachant qu'on souhaite initialiser r à 1 ?

   (b) À chaque itération, on souhaite remplacer n par $n//2$ (le quotient dans la division euclidienne de n par 2). Comment doivent évoluer y et r pour que l'invariant reste vrai en fin d'itération ?
6. En déduire la fonction `puissance_rapide` prenant en argument un flottant x et un entier k et renvoyant x^k avec un nombre restreint de multiplications.
7. Écrire une fonction `nombre_mult` gardant la structure de la fonction précédente mais renvoyant plutôt le nombre de multiplications utilisées. En prenant $x = 1$, déterminer le plus petit k tel que le calcul de x^k par la fonction `puissance_rapide` nécessite au moins 40 multiplications.

   Remarque : là encore, diviser la taille du problème par 2 à chaque étape permet d'obtenir un coût logarithmique (en k).

3 Calcul de frontière par dichotomie

L'objectif de cette partie est d'adapter le principe de recherche dichotomique rencontré dans la partie I à un problème voisin : la recherche d'un plus petit élément plus grand qu'une borne donnée.

1. Ëcrire une fonction `frontiere` prenant en argument une liste L, supposée triée, et un élément x, et renvoyant le plus petit indice i tel que x est inférieur ou égal à `L[i]`, ou `len(L)` si un tel indice n'existe pas.

   Par exemple :
   - `frontiere([1,3,4,6],4)` doit renvoyer 2 ;
   - `frontiere([1,3,4,6],7)` doit renvoyer 4.

   La fonction devra avoir une structure dichotomique, de façon à être en coût logarithmique (en la longueur de L).
2. Concevoir un jeu de test permettant de vérifier tous les cas limites pour votre fonction.