TP7 - Recherche dichotomique

Il y a des situations où un problème de dimension n se trouve réduit à un problème de dimension $\lfloor \frac{n}{2} \rfloor$ à l'itération suivante. Par exemple, si on veut trouver un nombre entier choisi par l'ordinateur entre 0 et 100 , on testera la valeur du milieu 50, puis, suivant que le nombre cherché est plus petit ou plus grand que 50, on cherchera le nombre dans l'intervalle $[\![0, 49]\!]$ ou dans l'intervalle $[\![51, 100]\!]$.

Le principe est donc de diviser le problème en deux sous-problèmes. On dit que les algorithmes dichotomiques appartiennent à la famille des algorithmes *diviser pour régner*.

On montrera en cours (au semestre 2) qu'un algorithme dichotomique a un coût logarithmique quand une approche naïve a un coût linéaire.

I Recherche dichotomique dans un tableau trié

But : On considère un tableau trié d'entiers `tab` de taille `n` et `x` un entier.
On souhaite écrire une fonction qui renvoie `True` si `x` est dans `tab` et `False` sinon.
Remarque : on n'utilisera pas l'instruction `x in tab`.

I.1 Méthode naïve

Q1. CORRIGER la fonction `recherche_naive`, qui passe en revue tous les éléments du tableau.

```
def recherche_naive(tab, x):
    """tab : tableau d'entiers
       x : entier
       Sortie : booleen
    """
       for elt in tab:
           if elt == x:
               return True
           else:
               return False
```

Dans le pire cas, combien y a-t-il de comparaisons si le tableau est de taille n ?
On dit que *le coût est linéaire (cf cours complexité).*

I.2 Algorithme dichotomique

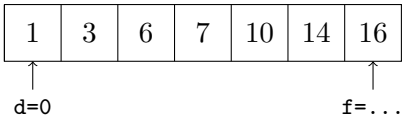
Le tableau étant trié, l'idée est de tester si l'élément du milieu du tableau est inférieur strictement, supérieur strictement ou égal à la valeur recherchée. S'il est égal, on a trouvé la valeur et l'algorithme renvoie `True`, sinon s'il est strictement inférieur, on cherche la valeur dans le demi-tableau de gauche, et sinon dans le demi-tableau de droite.
On itère la recherhce jusqu'à ce que le tableau de recherche soit vide.
On constate que l'intervalle de recherche est divisé par deux à chaque itération, ce qui permet à l'algorithme de converger rapidement.

Q2. Compléter l'analyse de l'algorithme :

On note :

- `d` : l'indice du début du tableau de recherche ;
- `f` : l'indice de fin du tableau de recherche.

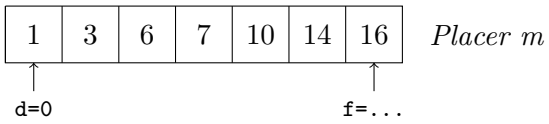
1. Initialisation



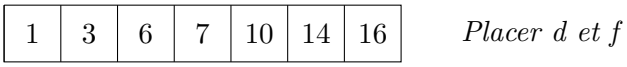
`d =`

`f =`

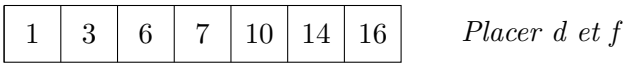
2. Etape 1 :
   a) On calcule l'indice de l'élément du milieu du tableau de recherche.
   `m =`



   b) Si `x` vaut `tab[m]`, alors la fonction renvoie ...
   c) Si `x` est plus petit que `tab[m]`, alors on poursuit la recherche dans le sous-tableau de ...
   `d =`
   `f =`
   Donc on a modifié ...

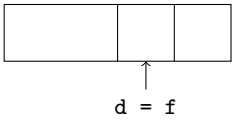


   d) Si `x` est plus grand que `tab[m]`, alors on poursuit la recherche dans le sous-tableau de ...
   `d =`
   `f =`
   Donc on a modifié ...



3. On poursuit la recherche, donc je itère l'étape 1.
4. Quand doit-on s'arrêter ?
   A chaque itération, soit `d` augmente strictement, soit `f` diminue strictement. Donc à chaque itération la longueur de l'intervalle $[\![$`d`,`f`$]\!]$ diminue strictement.

Regardonc le cas où `d` = `f`.



Alors `m = d = f`. Il y a 3 cas à voir :

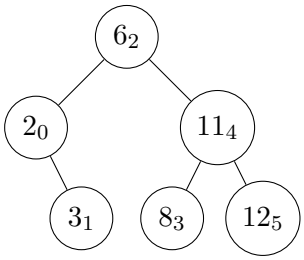
- Si `x` vaut `tab[m]`, alors on s'arrête ;
- Si `x` est plus petit que `tab[m]` (donc on doit s'arrêter), `f = m -1 = d-1`, donc `f<d`.
- Si `x` est plus grand que `tab[m]` (donc on doit s'arrêter), `d = m +1 = f+1`, donc `d>f`.

Donc on s'arrête quand `f<d`
Remarque : Si le tableau est vide, alors à l'initialisation on a : `d=0` et `f =0-1=-1`. Donc `f<d`, donc l'algorithme s'arrête.
Q3. Ecrire la fonction `recherche_dicho(tab, x)`

On verra plus tard dans l'année une version récursive de la recherche dichotomique.

I.3 Jeu de tests

On peut représenter la recherche dichotomique sous la forme d'un arbre binaire où les noeuds sont les éventuels milieux successifs utilisés.
Par exemple, si `tab = [2, 3, 6, 8, 11, 12]` :



Ainsi, pour un tableau de taille 6, on atteint un élément de celui-ci en effectuant au plus 3 itérations.
On met en place un jeu de tests pour tester la fonction `recherche_dicho`. On prouvera la correction de l'algorithme en cours.
Pour cela, on s'intéresse :

- aux valeurs limites :

```
t = [2, 3, 6, 8, 11, 12]
assert recherche_dicho([], 5) == ...
assert recherche_dicho(t, 2) == ...
assert recherche_dicho(t, 12) == ....
```

- à la valeur après une itération
  `assert ....`
- à une valeur après le maximum d'itération
  `assert ...`

I.4 Complexité

La complexité sera étudiée en cours au semestre 2. Nous verrons que la complexité est logarithmique, c'est-à-dire que le nombre d'opérations pour rechercher une valeur dans un tableau de taille n est de l'ordre de $O(\ln n)$.

Q4. En reprenant la fonction `recherche_dicho`, écrire une fonction `nb_iteration_max(tab)` qui a pour paramètre un tableau d'entiers `tab` trié , et qui renvoie le nombre maximal d'itérations lorsqu'on recherche une valeur quelconque par dichotomie.
Ainsi, on doit passer le test :
`assert nb_iteration_max([2, 3, 6, 8, 11, 12, 13, 14]) == 4`

Q5. On considere la fonction suivante :

```
def tableau_trie(n):
    return [k for k in range(n)]
```

Sans ordinateur, que vaut `tableau_trie(5)` ?
On dit que la liste a été définie par compréhension.

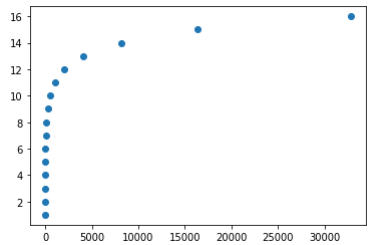
Q6. Ecrire une liste `taille`$= [1, 2, 2^2, \cdots, 2^{15}]$ par compréhension.
Q7. Ecrire une liste `nbIt` ayant 16 éléments avec :
pour `i` $\in [\![0, 15]\!]$, `nbIt[i]` est le nombre maximal d'itérations lorsqu'on fait une recherche dichotomique dans un tableau trié de taille `taille[i]`

Ainsi, on a deux listes de même dimension :

i	0	1	2	$\cdots$	15
taille	2^0	2^1	2^2	...	2^{15}
nbIt	...	...	...	...	...

Q8. Tracer le nombre d'itérations en fonction de la taille du tableau en utilisant la commande suivante de `matplotlib.pyplot` :

```
scatter(taille, nbIt) #nuage de points
```



Q9. Tracer sur le graphique précédent deux fonctions du type $K \log$ avec $K \in \mathbb{R}_+^*$ qui encadrent le nombre d'itérations.

On peut citer comme autre algorithme dichotomique l'exponentiation rapide qui sera abordé avec les fonctions récursives.
Q10. Ecrire une fonction `indice(tab, x)` qui a pour paramètre un tableau `tab` trié et une valeur `x` et qui renvoie l'indice où on doit insérer `x` dans le tableau `tab` pour que celui-ci reste trié.
La fonction doit donc passer les tests suivants :

```
t = [2, 3, 6, 8, 11, 12]

assert indice(t, 1) == 0
assert indice(t, 13) == 6
assert indice(t, 12) == 5
assert indice(t, 5) == 2
```