TP 19bis - Recherche par dichotomie dans une liste triée

Vous avez déjà joué au jeu *"Trouve un nombre entre 1 et 10"* et proposé 5. Si la réponse est *"Plus grand"* vous proposez 7 puis si l'on vous répond *"Plus petit"* vous proposez 6...

Vous avez effectué une **recherche par dichotomie** !

Faire une recherche dichotomique, c'est chercher une valeur dans un tableau en prenant le milieu de l'ensemble des solutions possibles (qui sont donc **triées**) pour éliminer la moitié des possibilités à chaque étape.

Création d'une liste

En vous inspirant des TP 18 sur les tuples et 19 sur les listes, écrivez une fonction nommée `liste_entiers_au_hasard` paramétrée par un entier `n`, qui renvoie une liste de n d'entiers choisis aléatoirement entre 1 et n.

```
liste_entiers_au_hasard(10)
# doit renvoyer par exemple [2, 0, 6, 6, 1, 7, 6, 4, 7, 2]
```

In [1]:

```python
# à compléter
from random import randint

def liste_entiers_au_hasard(n):
    return [randint(0,n) for x in range(n)]
```

In [2]:

```python
# Vérification
liste_entiers_au_hasard(10)
```

Out[2]:

```
[5, 5, 5, 0, 1, 9, 1, 2, 2, 10]
```

Tri d'une liste

Nous étudierons plus tard cette année différents algorithmes de tri, mais il existe une méthode `sort` permettant de trier une liste. Vérifiez la méthode avec le code ci-dessous :

In [3]:

```python
# Génération d'une liste avec notre fonction précédente
maliste = liste_entiers_au_hasard(20)

# Affichage de la liste
print ("Liste non triée :\t",maliste)

# Tri de la liste
maliste.sort()

# Affichage de la liste
print ("Liste triée :\t\t",maliste)
```

```
Liste non triée :        [17, 0, 2, 13, 0, 1, 6, 13, 17, 4, 3, 6, 15, 0, 7, 10, 6, 1, 5, 12]
Liste triée :            [0, 0, 0, 1, 1, 2, 3, 4, 5, 6, 6, 6, 7, 10, 12, 13, 13, 15, 17, 17]
```

Suppression des doublons

Lors du TP 19 sur les listes (Exercice 6) vous avez développé la fonction `doublons` qui supprime les éléments en double. Nous pourrions l'utiliser ici pour notre liste, cependant elle ne serait pas assez performante sur des listes de grande taille.

La ligne de code suivant permet d'extraire les valeurs uniques d'une liste (`set`) puis d'en générer une nouvelle liste (`list`)

In [4]:

```python
list(set(maliste))
```

Out[4]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 10, 12, 13, 15, 17]
```

Recherche par méthode brute

Implémentation de la méthode brute

La première méthode qui nous viendra naturellement pour rechercher un élément dans une liste est la méthode brute.
Elle consiste à parcourir la liste dans l'ordre dans une boucle et comparer chaque élément à l'élément recherché.
Si l'élément en cours est l'élément recherché alors la fonction s'arrête et renvoie `True`.

Ecrivez ci-dessous une fonction `recherche_brute` qui prend en paramètres un élément `elt` et une liste `liste` et renvoie `True` si l'élément est trouvé ou `False` sinon.

Votre fonction devra parcourir la liste élément par élément et les comparer au paramètre `elt`.

In [5]:

```python
# à compléter
def recherche_brute(elt,liste):
    for e in liste:
        if e==elt :
            return True
    return False
```

In [6]:

```python
# Vérification
recherche_brute(19,[0, 2, 4, 5, 6, 8, 9, 10, 11, 12, 13, 15, 17, 19, 20])
```

Out[6]:

```
True
```

Performance de la méthode brute

- Pour mesurer le temps d'exécution d'une fonction, on peut utiliser le module `timeit`, les lignes suivantes permettront d'effectuer cette mesure en secondes et de la stocker dans la variable `temps` :

```python
from timeit import default_timer as timer
debut_chrono = timer()
#code à mesurer
fin_chrono = timer()
duree = fin_chrono - debut_chrono
```

Testez ce code suivant en mesurant la durée de génération d'une liste de 100 000 entiers choisis aléatoirement.

In [7]:

```python
# à compléter
from timeit import default_timer as timer
debut_chrono = timer()
maliste=liste_entiers_au_hasard(100000)
fin_chrono = timer()
duree = fin_chrono - debut_chrono

duree
```

Out[7]:

```
0.22379999900000058
```

- Ecrivez maintenant un programme qui :
  1. Génère une liste de 100 000 entiers au hasard
  2. Trie la liste
  3. Supprime les doublons de la liste
  4. Recherche l'entier 100 000 dans la liste avec la fonction `recherche_brute` (pire des cas)
  5. Affichera si l'entier à été trouvé et en quelle durée

Note : Vous pouvez multiplier les durées par 1000 pour avoir un résultat en millisecondes qui sera plus lisible

Attention, sauvegardez votre notebook avant de vous lancer dans l'écriture d'une fonction qui traite un gros volume de données...

In [14]:

```python
# à compléter
from timeit import default_timer as timer

elt=2000000

# Génération de la liste
maliste=liste_entiers_au_hasard(elt)
maliste.sort()
maliste=list(set(maliste))

# Démarrage du chrono
debut_chrono = timer()
# Recherche
trouve=recherche_brute(elt,maliste)
# Fin du chrono et calcul de la durée
fin_chrono = timer()
duree = fin_chrono - debut_chrono

# Affichage du résultat
if trouve:
    print("L'entier " + str(elt) + " a été trouvé dans la liste.")
else:
    print("L'entier " + str(elt) + " n'a pas été trouvé dans la liste.")

# Affichange de la durée
print("Durée : " + str(duree*1000) + " ms")
```

```
L'entier 2000000 a été trouvé dans la liste.
Durée : 181.69999900000278 ms
```

Recherche par dichotomie

Implémentation de la méthode

Imaginons que nous recherchions une personne dans un annuaire classé par ordre alphabétique

- On ouvre l'annuaire au centre
- La personne est soit sur cette page, soit avant soit après
- En supposant qu'elle soit avant (on la recherchera donc dans la première moitié de l'annuaire)
- On ouvre la page du milieu de cette première moitié...
- On recommence jusqu'à obtenir la bonne page ou conclure que la personne n'est pas dans l'annuaire

Soit `liste` une liste triée et `elt` l'élément à chercher, l'algorithme qui implémente cette méthode est le suivant :

```
debut ← 0
fin ← Indice du dernier élément de la liste
Tant que debut <= fin faire
    milieu=(debut+fin)//2
    Si liste[milieu] = elt alors
        retourner True
    Sinon
        Si liste[milieu] < elt alors
            debut ← milieu+1
        Sinon
            fin ← milieu-1
retourner False
```

A l'aide de cet algorithme, ecrivez ci-dessous une fonction `recherche_dicho` qui prend en paramètres un élément `elt` et une liste `liste` et renvoie `True` si l'élément est trouvé ou `False` sinon.

In [9]:

```python
# à compléter
def recherche_dicho(elt,liste):
    debut=0
    fin=len(liste)-1

    while debut<=fin:
        milieu=(debut+fin)//2
        if liste[milieu] == elt:
            return True
        else:
            if liste[milieu] < elt :
                debut=milieu+1
            else:
                fin=milieu-1
    return False
```

In [10]:

```python
# Vérification
recherche_dicho(19,[0, 2, 4, 5, 6, 8, 9, 10, 11, 12, 13, 15, 17, 19, 20])
```

Out[10]:

```
True
```

Performance de la recherche dichotomique

- Ecrivez maintenant un programme qui :
  1. Génère une liste de 100 000 entiers au hasard
  2. Trie la liste
  3. Supprime les doublons de la liste
  4. Recherche l'entier 100 000 dans la liste avec la fonction `recherche_dicho` (pire des cas)
  5. Affichera si l'entier à été trouvé et en quelle durée

Note : Vous pouvez multiplier les durées par 1000 pour avoir un résultat en millisecondes qui sera plus lisible

In [15]:

```python
# à compléter
from timeit import default_timer as timer

elt=2000000

# Génération de la liste
maliste=liste_entiers_au_hasard(elt)
maliste.sort()
maliste=list(set(maliste))

# Démarrage du chrono
debut_chrono = timer()
# Recherche
trouve=recherche_dicho(elt,maliste)
# Fin du chrono et calcul de la durée
fin_chrono = timer()
duree = fin_chrono - debut_chrono

# Affichage du résultat
if trouve:
    print("L'entier " + str(elt) + " a été trouvé dans la liste.")
else:
    print("L'entier " + str(elt) + " n'a pas été trouvé dans la liste.")

# Affichange de la durée
print("Durée : " + str(duree*1000) + " ms")
```

```
L'entier 2000000 a été trouvé dans la liste.
Durée : 0.0 ms
```

Comparatif entre les deux méthodes

- Ecrivez maintenant un programme qui :
  1. Génère une liste de 100 000 entiers au hasard
  2. Trie la liste
  3. Supprime les doublons de la liste
  4. Recherche l'entier 100 000 dans la liste avec la fonction `recherche_brute` (pire des cas)
  5. Recherche l'entier 100 000 dans la liste avec la fonction `recherche_dicho`
  6. Affiche si l'entier à été trouvé et en quelle durée pour chacune des deux recherches

Note : Vous pouvez multiplier les durées par 1000 pour avoir un résultat en millisecondes qui sera plus lisible

In [12]:

```python
# à compléter
from timeit import default_timer as timer

elt=2000000

# Génération de la liste
maliste=liste_entiers_au_hasard(elt)
maliste.sort()
maliste=list(set(maliste))

# Démarrage du chrono
debut_chrono_b = timer()
# Recherche
trouve_b=recherche_brute(elt,maliste)
# Fin du chrono et calcul de la durée
fin_chrono_b = timer()
duree_b = fin_chrono_b - debut_chrono_b

# Démarrage du chrono
debut_chrono_d = timer()
# Recherche
trouve_d=recherche_dicho(elt,maliste)
# Fin du chrono et calcul de la durée
fin_chrono_d = timer()
duree_d = fin_chrono_d - debut_chrono_d


# Affichage du résultat
if trouve_d:
    print("L'entier " + str(elt) + " a été trouvé dans la liste")
else:
    print("L'entier " + str(elt) + " n'a pas été trouvé dans la liste")

print("\tBRUTE : "+ str(duree_b*1000) + " ms")
print("\tDICHO : "+ str(duree_d*1000) + " ms")
```

```
L'entier 2000000 a été trouvé dans la liste
        BRUTE : 181.4000010000001 ms
        DICHO : 0.0999989999996842 ms
```

- Modifiez maintenant votre code pour augmenter le nombre d'éléments et affiner votre comparaison.

Complexité des deux méthodes de recherche

Recherche brute

Pour effectuer une recherche brute, nous allons successivement comparer l'élément recherché avec les éléments de la liste.
Si nous sommes dans le pire des cas : l'élément est le dernier de la liste, nous allons donc effectuer autant de comparaisons que d'éléments dans la liste.
Si `n` est le nombre d'éléments dans la liste nous effectuerons donc **n** comparaisons.

La complexité de la recherche brute est donc **linéaire**

Recherche dichotomique

Pour effectuer une recherche dichotomique, nous allons comparer l'élément recherché l'élément central de la liste puis couper la liste en deux.
Dans le pire des cas : l'élément est le dernier de la liste (ou le premier), nous diviser successivement le nombre d'éléments de la liste par 2.
Si `n` est le nombre d'éléments dans la liste nous effectuerons donc **$\log_2(n)$** comparaisons.

La complexité de la recherche dichotomique est donc **logarithmique**

Exemples :

Prenons par exemple la liste triée suivante :

`[0, 3, 4, 5, 6, 9, 10, 11, 12, 14, 15, 16, 21, 23, 24, 26, 27, 30, 31, 34, 36, 37, 38, 39, 40, 43, 44, 47, 48, 50, 51, 52]`

Elle contient 32 éléments, recherchons le dernier élément `52` :

Recherche brute :

- Nous allons comparer 52 à tous les éléments un par un de 0 à 52 pour finalement le trouver.
  => Le nombre de comparaisons est de **32** soit le nombre d'éléments dans la liste

Recherche dichotomique :

- Nous comparons 52 à l'élément central : 26 -> 1 comparaison
- Nous divisons la liste en deux et le nombre est plus grand
- Nous comparons 52 à l'élément central de la nouvelle liste : 39 -> 2 comparaisons
- Nous divisons la liste en deux et le nombre est plus grand
- Nous comparons 52 à l'élément central de la nouvelle liste : 47 -> 3 comparaisons
- Nous divisons la liste en deux et le nombre est plus grand
- Nous comparons 52 à l'élément central de la nouvelle liste : 51 -> 4 comparaisons
- Nous divisons la liste en deux et le nombre est plus grand
- Nous comparons 52 à l'élément central de la nouvelle liste : 52 -> 5 comparaisons
  => Le nombre de comparaisons est donc : **5** soit **$\log_2(32)$** (car **2^5=32**)

En Python :

In [17]:

```python
import math

# Fonction de recherche brute précédente qui renvoie le nombre de comparaisons effectuées pour évaluer sa compl
def brute1(elt,liste):
    # initialisation à 0
    nbcomp=0
    for e in liste:
        # On ajoute 1 à chaque fois qu'on va comparer
        nbcomp+=1
        if e==elt :
            return nbcomp
    return nbcomp

# Fonction de recherche dichotomique précédente qui renvoie le nombre de comparaisons effectuées pour évaluer s
def dicho1(elt,liste):
    # initialisation à 0
    nbcomp=0
    debut=0
    fin=len(liste)-1
    while debut<=fin:
        milieu=(debut+fin)//2
        # On ajoute 1 à chaque fois qu'on va comparer
        nbcomp+=1
        if liste[milieu] == elt:
            return nbcomp
        else:
            if liste[milieu] < elt :
                debut=milieu+1
            else:
                fin=milieu-1
    return nbcomp

# Nombre d'éléments de la liste de départ
elt=2000000
# Génération de la liste
maliste=liste_entiers_au_hasard(elt)
# Tri de la liste
maliste.sort()
# Suppression des doublons
maliste=list(set(maliste))

# Recherche Brute du dernier élément possible de la liste (elt)
nbrut=brute1(elt,maliste)
# Recherche dichotomique
ndich=dicho1(elt,maliste)

# Affichage des résultats
print("Eléments dans la liste : \t",len(maliste))
print("Opérations recherche brute : \t",nbrut)
print("Opérations recherche dicho : \t",ndich)
# Calcul de la complexité théorique
print("log(",len(maliste),")= \t\t",math.log2(len(maliste)))
```

```
Eléments dans la liste :         1263418
Opérations recherche brute :     1263418
Opérations recherche dicho :     21
log( 1263418 )=                  20.26890060097344
```