Introduction

Objectifs :

- *Revoir la recherche séquentielle d'un élément dans un tableau.*
- *Découvrir le principe de dichotomie pour rechercher efficacement un élément dans une séquence triée.*
- *Programmer la recherche dichotomique d'un entier dans un tableau trié.*
- *Comparer le coût d'exécution d'une recherche dichotomique et d'une recherche linéaire : notion de coût logarithmique.*

1 Recherche séquentielle

Point de cours 1 ***Recherche séquentielle***

L'algorithme de **recherche séquentielle** d'un élément `e` dans un tableau `tab` consiste à parcourir tous les éléments de `tab` en comparant chaque élément lu avec l'élément recherché.
On peut sortir prématurément de la boucle de balayage dès que l'élément `e` a été trouvé.

Exercice 1

1. Implémenter la recherche séquentielle dans la fonction `recherche_seq` qui prend en paramètre un tableau d'entier et renvoie `True` si `e` appartient à `tab` ou `False`.

```
def recherche_seq(tab, e):
    """
    Détermine si e dans tab

    Paramètres:
        tab : tableau d'entiers
        e : un entier

    Retour:
        booléen
    """
    trouve = False
    for element in tab:
        if ............ :
            .............
    return trouve
```

2. Modifier la fonction précédente pour ne plus utiliser la variable `trouve` et retourner `True` dès que l'élément `e` a été trouvé ou `False` sinon.

```
def recherche_seq2(tab, e):
    ....................
    ....................
    ....................
    ....................
    ....................
    ....................
```

3. Soit `tab` un tableau de taille 10^6. Dans quel cas le nombre de comparaisons effectué par `recherche_seq2(tab, e)` est-il minimal (*meilleur des cas*) ? maximal (*pire des cas*) ?

....

....

....

4.
```
import time

for m in range(1, 9):
    taille = 10 ** m
    tab = list(range(taille))
    e = taille + 1
    chrono = time.perf_counter()
    recherche_seq2(tab, e)
    print(f"taille = {taille}, temps(s) = {time.perf_counter() - chrono}")
```

Exécuter le code ci-dessus. Obtenez-vous les mêmes résultats que ci-dessous ? Quel élément pertinent peut nous renseigne sur la performance de l'algorithme de recherche séquentielle ? Quelle conjecture peut-on formuler sur la relation entre la taille n du tableau et le temps d'exécution de la recherche séquentielle dans ce tableau dans le *pire des cas* ?

....

....

....

....

....

```
taille = 10, temps(s) = 1.0888001270359382e-05
taille = 100, temps(s) = 1.3110999134369195e-05
taille = 1000, temps(s) = 3.356500019435771e-05
taille = 10000, temps(s) = 0.0003139219988952391
taille = 100000, temps(s) = 0.003978888998972252
taille = 1000000, temps(s) = 0.03422377499737195
taille = 10000000, temps(s) = 0.34727345899955253
taille = 100000000, temps(s) = 3.385230176998448
```

2 Principe de dichotomie

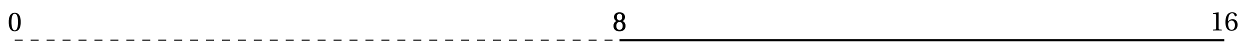
Point de cours 2 ***La dichotomie***

Considérons le jeu suivant : « Le maître du jeu choisit un entier secret entre 0 inclus et 2^n exclu que le joueur doit deviner en un minimum d'étapes, sachant qu'à chaque étape le joueur demande si le nombre secret est supérieur ou égal à sa proposition. ».

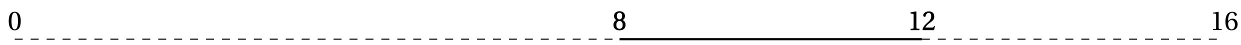
Une stratégie assez naturelle repose sur le **principe de dichotomie** : le joueur propose à chaque étape le milieu de la *zone de recherche* et peut ainsi diviser par 2 celle-ci à grâce à l'information sur la position du secret par rapport à sa proposition.

Étudions un exemple avec $n = 4$ et le secret 9 choisi entre 0 et $2^4 - 1 = 15$.

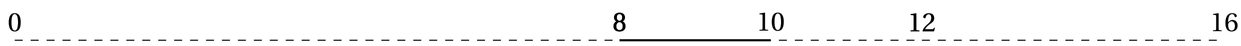
- **Itération 1 :** La zone de recherche 1 est $[0; 2^4 = 16[$ (intervalle semi-ouvert droite), le joueur propose $\left\lfloor \frac{0+16}{2} \right\rfloor = 8$ et le maître du jeu répond *supérieur ou égal* ce qui élimine la moitié inférieure $[0; 7]$ de la zone de recherche.



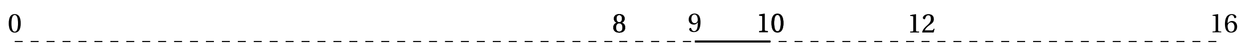
- **Itération 2 :** La zone de recherche 2 est $[8; 16[$, le joueur propose $\left\lfloor \frac{8+16}{2} \right\rfloor = 12$ et le maître du jeu répond *inférieur* ce qui élimine la moitié supérieure $[12; 16[$ de la zone de recherche.



- **Itération 3 :** La zone de recherche 3 est $[8; 12[$, le joueur propose $\left\lfloor \frac{8+12}{2} \right\rfloor = 10$ et le maître du jeu répond *inférieur* ce qui élimine la moitié supérieure $[10; 12[$ de la zone de recherche.



- **Itération 4 :** La zone de recherche 4 est $[8; 10[$, le joueur propose $\left\lfloor \frac{8+10}{2} \right\rfloor = 9$ et le maître du jeu répond *supérieur ou égal* ce qui élimine la moitié inférieure $[8; 9[$ de la zone de recherche. La nouvelle zone de recherche $[9; 10[$ contient un seul entier 9 qui est forcément l'entier secret.



On peut se convaincre qu'en n itérations on peut déterminer n'importe quel entier secret parmi les 2^n entiers entre 0 et $2^n - 1$, alors qu'avec une recherche linéaire testant tous les entiers successivement il faudrait 2^n itérations dans le pire des cas.

Le *principe de dichotomie* accélère la recherche en éliminant à chaque itération la moitié de la zone de recherche.

Exercice 2 ***Devinette***

On veut implémenter le jeu décrit précédemment : vous êtes le maître du jeu et choisissez un entier secret entre 0 inclus et 2^n exclu et l'ordinateur doit le deviner par recherche dichotomique à l'aide de vos réponses aux questions *Supérieur ou égal à ma proposition ?*. Par convention, on code par 1 une réponse positive et 0 une réponse négative. Voici un exemple d'exécution pour deviner 9 dans $[0; 2^4[$.

```
Plus grand ou égal à 8 (1 pour oui et 0 pour Non) ?1
Plus grand ou égal à 12 (1 pour oui et 0 pour Non) ?0
Plus grand ou égal à 10 (1 pour oui et 0 pour Non) ?0
Plus grand ou égal à 9 (1 pour oui et 0 pour Non) ?1
9
```

1. Compléter la fonction ci-dessous pour qu'elle implémente le jeu.

```python
def devinette(gauche, droite):
    """
    Paramètres : gauche et droite des entiers
    Précondition : 0 <= gauche < droite
    Valeur renvoyée : un entier, devine par dichotomie l'entier n choisi
         par
    l'utilisateur avec gauche <= secret < droite
    """
    assert 0 <= gauche < droite
    while ........................:
        .............................
        .............................
        .............................
        milieu = (gauche + droite) // 2
        reponse = int(input('Plus grand ou égal à ' + str(milieu) + ' (1
             pour oui et 0 pour Non) ?'))
        print(reponse)
        .............................
        .............................
        .............................
        .............................
        .............................
    return gauche
```

2. Que représente la séquence d'entiers 0 ou 1 codant les réponses aux questions ? Pourquoi à votre avis ?

....

....

....

....

....

....

Exercice 3

Appliquons le principe de la recherche dichotomique pour résoudre l'exercice *Tas de Graine* du Castor Informatique 2017 :
https://concours.castor-informatique.fr/index.php?team=castor2017

3 Recherche dichotomique dans un tableau trié

Point de cours 3

On s'intéresse au problème de la recherche d'un élément dans un tableau de valeurs : nom dans un annuaire, valeur dans une série de mesures, adresse IP dans une liste noire ...
Si le **tableau** est **trié**, on dispose d'une information supplémentaire dont on peut tirer partie pour accélérer la recherche comme dans le jeu de la devinette : on élimine à chaque itération la moitié du tableau alors que la recherche séquentielle élimine un seul élément à chaque itération.
Par exemple, prenons un tableau contenant des noms de clients, trié dans l'ordre alphabétique croissant. Pour tester l'appartenance d'un nom à ce tableau, on le compare avec celui en position médiane et trois cas peuvent se présenter :

- le nom cherché est inférieur au nom médian et on continue la recherche dichotomique dans la première partie du tableau si elle est non vide, sinon on arrête car le nom cherché ne peut être dans le tableau;
- le nom cherché est supérieur au nom médian et on continue la recherche dichotomique dans la seconde partie du tableau si elle est non vide, sinon on arrête car le nom cherché ne peut être dans le tableau;
- le nom cherché est égal au nom médian et on arrête la recherche.

Décrivons l'algorithme de recherche d'une valeur `v` dans un tableau `t` **trié dans l'ordre croissant**.

☞ **Étape 1 :** On initialise les deux bornes `g` et `d` de la zone recherche :

```python
g = 0
d = len(t) - 1
```

☞ **Étape 2 :** On détermine la position médiane `m` avec la division entière `(g + d) // 2`.

```python
m = (g + d) // 2
```

☞ **Étape 3 :** On compare `v` avec `t[m]` et on distingue trois cas :

- *Cas 1 :* si `t[m] > v` alors on peut restreindre la recherche à la zone gauche entre les index `g` et `m - 1` donc on modifie `d` avec `d = m - 1`;
- *Cas 2 :* si `t[m] < v` alors on peut restreindre la recherche à la zone droite entre les index `m + 1` et `d` donc on modifie `g` avec `g = m + 1`;
- *Cas 3 :* si `t[m] == v` alors on a trouvé la valeur `v` et on peut renvoyer `True` ou l'index `m` avec une sortie prématurée selon la spécification de la fonction.

☞ **Étape 4 :** Si on n'a pas trouvé `v` et que `g <= d` on revient à l'étape 2 et on effectue une nouvelle itération de la boucle, sinon on sort de la boucle et on renvoie `False`, `None` ou -1 selon la spécification de la fonction.

	0	g m-1	m	m+1 d	len(t)-1
t	éléments < v	Cas 1	Cas 3	Cas 2	éléments > v

Deux points de vigilance :

- *La recherche dichotomique ne peut s'appliquer qu'à un tableau trié.*
- *L'implémentation de la recherche dichotomique dans un tableau trié est plus délicate que le jeu de devinette car trois cas ($<$, $=$ ou $>$) sont possibles à chaque itération au lieu de 2 ($<$ ou $\geqslant$) et la valeur cherchée peut ne pas appartenir au tableau.*

Méthode *Fonctions de tri en Python*

Pour tableau `t` d'objets comparables (entiers, caractères ...) , Python propose deux façons de le trier :

☞ `sorted(t)` *retourne une nouveau tableau* trié dans l'ordre croissant;

☞ `t.sort()` *trie en place* le tableau `t` par ordre croissant.

Dans les deux cas, on peut obtenir un ordre décroissant avec le paramètre optionnel `reverse = True`.
Les coûts d'exécution temporels des fonctions de tri en Python sont optimaux, en $O(n\log(n))$ par rapport à la taille n du tableau.

```
>>> t = ['b','c','a']
>>> m = sorted(t)
>>> m
['a', 'b', 'c']
>>> t
['b', 'c', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c']
>>> t.sort(reverse=True)
>>> t
['c', 'b', 'a']
>>> from random import randint
>>> alea = [randint(-100, 100) for _ in range(10)]
>>> alea
[-16, -40, 79, -83, 85, -89, -96, 56, 6, 72]
>>> dec = sorted(alea, reverse=True)
>>> dec
[85, 79, 72, 56, 6, -16, -40, -83, -89, -96]
```

Exercice 4 ***Recherche dichotomique dans un tableau trié***

1. Écrire une fonction `est_croissant` correspondant à la spécification ci-dessous.

```
def est_croissant(tab):
    """
    Détermine si tab est dans l'ordre croissant
    Paramètre : tab un tableau d'entiers
    Retour :   un booléen
    """
    ............................
    ............................
    ............................
    ............................
    ............................
```

2. Écrire une fonction `recherche_dicho_tab` implémentant l'algorithme de recherche dichotomique dans un tableau trié décrit précédemment et correspondant à la spécification ci-dessous. Fournir un jeu de tests unitaires.

```
def recherche_dicho_tab(valeur, tab):
    """
    Renvoie l'index de première occurence de valeur dans tab
     ou -1 si valeur pas dans tab
    Paramètres :
       valeur : un entier
       tab : un tableau d'entiers
    Retour:
       un entier
    """
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
    ............................
```

4 Comparaison des performances des recherches séquentielle et dichotomique

Exercice 5 ***Mesurer expérimentalement un temps exécution et conjecturer le coût temporel d'un algorithme***

1. Quel est le pire des cas (en nombre de comparaisons avec `valeur`) pour une recherche séquentielle ? Quel est alors le nombre de comparaisons avec `valeur` en fonction de la taille n du tableau `tab` ?

   ....

   ....

   ....

2. Quel est le pire des cas (en nombre de comparaisons avec `valeur`) pour `recherche_dicho_tab(valeur, tab)` ?

   ....

   ....

   ....

3. Modifier la fonction `recherche_dicho_tab` en `recherche_dicho_tab_compteur` pour qu'elle renvoie un `tuple` avec l'index de la valeur cherchée (ou -1 si pas d'occurrence) et le compteur d'itérations effectuées.

   ....

   ....

   ....

4. On a évalué en console l'expression suivante. Comment interpréter sa valeur ?

```
>>> [(k, recherche_dicho_tab_compteur(2**k, list(range(2**k)))[1]) for k in
      range(1, 15)]
[(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10),
     (11, 12), (12, 13), (13, 14), (14, 15)]
```

   ....

   ....

   ....

   ....

   Quelle conjecture peut-on en tirer sur l'ordre de grandeur du coût en nombre de comparaisons avec `valeur` pour la fonction de recherche dichotomique dans un tableau trié de taille 2^n ?

   ....

   ....

   ....

Démontrons cette conjecture [a]

Soit un tableau `tab` *trié dans l'ordre croissant de taille* 2^n.
Supposons qu'on recherche dans `tab` *un élément* `e` *n'appartenant pas à* `tab`.
Initialement la taille de la zone de recherche est 2^n.
A chaque itération, si la zone de recherche est l'intervalle d'index $[g\,;d[$, *alors on on compare l'élément d'index médian* $m = \left\lfloor \frac{g+d}{2} \right\rfloor$ *avec* `e` *et comme* `e` *n'est pas dans* `tab`, *on poursuite la recherche dans l'intervalle d'index* $\left[a; \left\lfloor \frac{g+d}{2} \right\rfloor\right[$ *ou* $\left[\left\lfloor \frac{g+d}{2} \right\rfloor + 1; b\right[$ *dont la taille est inférieure ou égale à la moitié de la zone de recherche précédente* $\frac{g-d}{2}$.
Ainsi au bout de n *itérations la taille de la zone de recherche est inférieure ou égale à la taille de la zone de recherche initiale divisée* n *fois de suite par* 2 *donc par* 2^n. *Après* n *itérations la taille de la zone de recherche est donc inférieure ou égale à* $\frac{2^n}{2^n} = 1$. *Il reste au plus* 1 *élément donc la* $(n+1)^e$ *itération termine l'algorithme en au plus* $\dots\dots$ *comparaisons au total.*

5. On fournit dans l'activité Capytale du chapitre plusieurs fonctions :

- `evolution_ratio_diff_pire_cas` prend en paramètre une fonction de recherche dans un tableau (séquentielle ou dichotomique) et affiche les temps moyens d'exécution de la recherche dans le pire des cas sur des échantillons de tableaux de même taille choisie dans une liste `benchmark` de tailles croissantes. Elle affiche aussi le quotient et la différence entre les temps moyens pour des tailles successives.
- `graphique_comparaison_temps_pire_cas` Prend en paramètre deux fonctions de recherche dans un tableau, les exécute sur des échantillons de tableaux de même taille dans le pire des cas et affiche un graphique avec les temps de l'une en fonction de l'autre.

Exécuter les instructions suivantes :

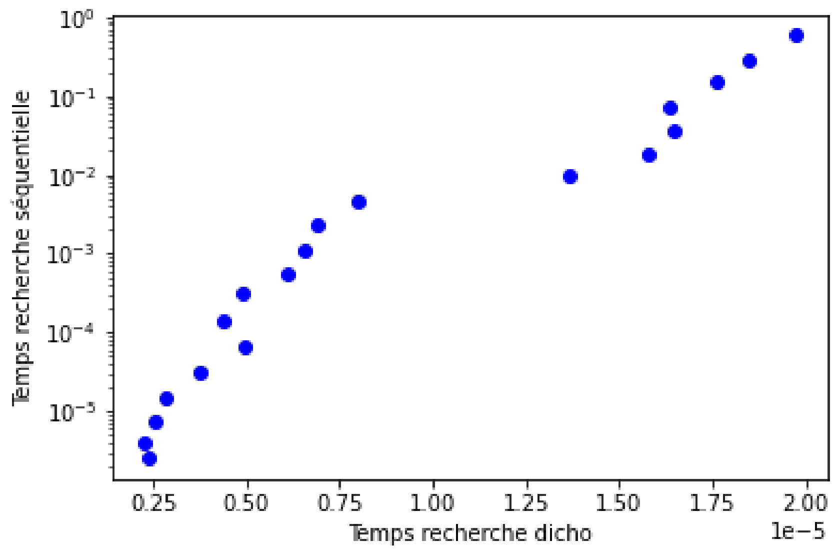
```
evolution_ratio_diff_pire_cas(recherche_sequentielle, [10 ** k for k in
     range(3, 7)])
evolution_ratio_diff_pire_cas(recherche_dicho_tab, [10 ** k for k in range
     (3, 7)])
graphique_comparaison_temps_pire_cas(recherche_sequentielle,
     recherche_dicho_tab2, [2 ** k for k in range(5, 23)],
  "comparaison_seq_dicho.png", semilogx = False, semilogy=True)
```

Voici un exemple de sortie (les temps dépendent de la machine ...)

```
recherche_sequentielle(1000,list(range(1000))) en 7.9e-05 secondes,
 tps/tps_pre = 0.0e+00 et tps - tps_pre = 0.0e+00

recherche_sequentielle(10000,list(range(10000))) en 7.7e-04 secondes,
 tps/tps_pre = 9.8e+00 et tps - tps_pre = 6.9e-04

recherche_sequentielle(100000,list(range(100000))) en 7.0e-03 secondes,
 tps/tps_pre = 9.1e+00 et tps - tps_pre = 6.3e-03

recherche_sequentielle(1000000,list(range(1000000))) en 6.8e-02 secondes,
 tps/tps_pre = 9.8e+00 et tps - tps_pre = 6.1e-02
```

```
recherche_sequentielle(10000000,list(range(10000000))) en 6.8e-01 secondes,
 tps/tps_pre = 9.9e+00 et tps - tps_pre = 6.1e-01

recherche_dicho_tab(1000,list(range(1000))) en 3.8e-06 secondes,
 tps/tps_pre = 0.0e+00 et tps - tps_pre = 0.0e+00

recherche_dicho_tab(10000,list(range(10000))) en 5.8e-06 secondes,
 tps/tps_pre = 1.5e+00 et tps - tps_pre = 2.0e-06

recherche_dicho_tab(100000,list(range(100000))) en 9.6e-06 secondes,
 tps/tps_pre = 1.7e+00 et tps - tps_pre = 3.8e-06

recherche_dicho_tab(1000000,list(range(1000000))) en 1.8e-05 secondes,
 tps/tps_pre = 1.8e+00 et tps - tps_pre = 7.9e-06

recherche_dicho_tab(10000000,list(range(10000000))) en 2.1e-05 secondes,
 tps/tps_pre = 1.2e+00 et tps - tps_pre = 3.6e-06
```

La dernière instruction doit générer le graphique à échelle semi-logarithmique (en y) ci-dessous :



Comment interpréter ce graphique ?

....

....

....

....

....

....

Quelles confirmations peut-on tirer de ces expériences sur la comparaison des coûts d'exécution temporels dans le pire des cas pour les recherches séquentielle et dichotomique dans un tableau trié ?

....

....

....

....

....

a. On note $\lfloor x \rfloor$ la partie entière de x : on a

Table des matières