

PARTIE 2 : Algorithme de dichotomie

La recherche d'un élément dans une liste fait partie des problèmes récurrents. Lorsque la liste est triée, la **recherche dichotomique** est beaucoup plus rapide que la recherche séquentielle. **Une condition essentielle sera que la liste est triée.**

1. Rappel des fonctions natives en Python pour le tri

1.1. Fonction native sorted

La fonction native **sorted** renvoie une liste équivalente à la première mais triée. Cela veut dire que la liste originale n'est pas modifiée : elle est toujours dans le désordre.

```
listeOriginale=[13, 55, 54, 57, 12, 80, 16, 72, 30, 99]
listeTrie=sorted(listeOriginale)
print(listeTrie)
```

Avantage : on garde en mémoire la configuration initiale des données.

Inconvenient : ça prend deux fois plus de place mémoire. Avec 96 milliards de données de 4 octets, ça peut être important...

1.2. Fonction native sort

La méthode **sort** modifie sur place la liste sur laquelle on fait agir cette méthode.

```
listeOriginale2=[13, 55, 54, 57, 12, 80, 16, 72, 30, 99]
listeOriginale2.sort()
print(listeOriginale2)
```

Avantage : moins de place mémoire nécessaire

Inconvenient : si on veut retrouver l'état initial, il faut espérer que les données initiales sont stockées ailleurs que dans cette liste.

Avec ces deux fonctions on peut faire aussi di tri avec une liste de mots par ordre alphabétique. Comment fait-il ? Il regarde simplement la valeur ASCII (ou UNICODE pour les valeurs supérieures à 127) et trie par rapport à cette valeur.

A c'est 65, B c'est 66, a c'est 97.

Ainsi un mot commençant par A passera avant un mot commençant par a.

```
phrase_sans_ponctuation = "Voici la phrase qui contient deux fois voici pour montrer que UNICODE ne traite pas de la même façon le V et le v"
listeMots=phrase_sans_ponctuation.split()
listeMots2=phrase_sans_ponctuation.split()
print(listeMots)
listeMotsTries=sorted(listeMots)
print(listeMotsTries)
listeMots2.sort()
print(listeMots2)
```

🔥 Exercice D1 : nous souhaitons créer un script qui va nous générer une liste d'entiers compris entre 0 et 100. Créer deux fonctions qui prennent en paramètre le nombre d'éléments et qui renvoient une liste d'entiers :

- La fonction `créer` crée une liste renvoyée par compréhension,
- La fonction `ceer2` crée une liste renvoyée par extension (avec la méthode `append`).

```
Def creer (nbr) :
```

```
Return liste
```

```
Def ceer2 (nbr) :
```

```
Return liste2
```

🔥 Exercice D2 : reprendre les deux fonctions précédentes pour qu'elles renvoient des listes triées.

Dans toute la suite de cette partie, nous supposons que nous avons à notre disposition une liste triée.

2. Principe de la recherche par dichotomie

La recherche dichotomique est découverte en 1946 dans un article de John Mauchly, un physicien américain qui a coconçu l'ENIAC, un des premiers ordinateurs.

2.1. Principe du diviser pour mieux régner

La méthode du diviser pour mieux régner (divide and conquer en anglais) consiste en trois étapes :

- Diviser : découper un problème en sous-problèmes.
- Régner : résoudre les sous problèmes
- Combiner : calculer une solution au problème initial à partir des solutions des sous-problèmes.

Principe de la recherche dichotomique

La recherche dichotomique permet de rechercher un entier (ou une chaîne de caractère) dans une liste triée ainsi que sa position. Le principe de recherche d'une valeur v est le suivant :

- Si la liste est vide, la réponse est négative, la recherche est terminée.
- Sinon, trouver la valeur la plus centrale de la liste et comparer cette valeur à l'élément recherché :
 - Si la valeur est celle recherchée : répondre positivement, la recherche est terminée.
 - Si la valeur est strictement plus petite que l'élément recherché, reprendre la procédure avec la seconde moitié de la liste
 - Sinon reprendre la procédure avec la première moitié de la liste.

✓ EXEMPLE 1 : chercher 40 est dans la liste

La condition de base pour pouvoir réaliser cette recherche est d'avoir une liste triée.

Par exemple avec une liste triée de 20 éléments et donc d'indices compris entre 0 et 19.

[06, 08, 09, 13, 18, 37, 37, 59, 60, 60, 61, 68, 70, 80, 80, 83, 83, 89, 91, 96]

	→										←									
Indice	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
Elément	06	08	09	13	18	37	37	59	60	60	61	68	70	80	80	83	83	89	91	96

Cherchons si 40 est présent dans la liste.

En recherche séquentielle, on regarderait l'indice 0 (06), l'indice 1 (08)... Ici, il faut donc 20 boucles et 20 comparaisons.

En **recherche dichotomique**, nous allons systématiquement supprimer la moitié des données restantes en comparant la valeur associée à l'index central. Si la valeur n'est pas la bonne, on ne gardera que la partie restante à droite ou à gauche.

Etape 1 - Intervalle de départ [0 ; 19]

- On considère un point gauche d'indice 0 : $g = 0$.
- On considère un point droit d'indice 19 : $d = 19$.
- On calcule l'indice central à l'aide d'une division entière : $(0+19)//2$ donne 9 On aura donc $c = 9$.

	→								←											
Indice	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
Elément	06	08	09	13	18	37	37	59	60	60	61	68	70	80	80	83	83	89	91	96

Comme liste [9] = 60 et qu'on cherche 40 dans une liste triée, on ne garde que la partie à gauche du point central d'indice 9.

On va alors changer le point droit avec $d = 8$

On recommence ces opérations sur le nouvel intervalle [0;8].

Etape 2 : Intervalle [0;8]

- On considère un point gauche d'indice 0 : $g = 0$.
- On considère un point droit d'indice 8 : $d = 8$.
- On calcule l'indice central à l'aide d'une division entière : $(0+8)//2$ donne 4 On aura donc $c = 4$.

	→								←											
Indice	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
Elément	06	08	09	13	18	37	37	59	60	60	61	68	70	80	80	83	83	89	91	96

Comme l'indice 4 fait référence à 18 et qu'on cherche 40 dans une liste triée, on ne garde que la partie à droite de l'indice 4.

On va alors changer le point gauche avec $g = 5$

On recommence ces opérations sur le nouvel intervalle [5;8].

Etape 3 : Intervalle [5;8]

- On considère un point gauche d'indice 5 : $g = 5$.
- On considère un point droit d'indice 8 : $d = 8$.
- On calcule l'indice central à l'aide d'une division entière : $(5+8)//2$ donne 6 On aura donc $c = 6$.

						→				←										
Indice	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
Elément	06	08	09	13	18	37	37	59	60	60	61	68	70	80	80	83	83	89	91	96

Comme l'indice 6 fait référence à 37 et qu'on cherche 40 dans une liste triée, on ne garde que la partie à droite de l'indice 6.

On va alors changer le point gauche avec $g = 7$

On recommence ces opérations sur l'intervalle [7;8].

Etape 4 : Intervalle [7;8]

- On considère un point gauche d'indice 0 : $g = 7$.
- On considère un point droit d'indice 8 : $d = 8$.
- On calcule l'indice central à l'aide d'une division entière : $(7+8)//2$ donne 7 On aura donc $c = 7$.

[illegible]



Comme l'index 7 fait référence à 59 et qu'on cherche 40 dans un tableau trié, on ne garde que la partie à gauche de l'index 7.

Du coup, on obtient un ensemble vide.

Conclusion : 40 n'appartient pas à notre tableau.

Réponse obtenue avec 4 comparaisons. C'est mieux que 20 avec une recherche séquentielle.

 Exercice D3 : Réaliser une recherche dichotomique de 50 dans la liste ci-dessous.

																	
Indice	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
Élément	18	23	25	35	48	50	58	68	70	73	75	79	80	82	85	89	

Questions

Que vaut l'indice central de l'étape 1 ?

Que vaut l'intervalle au début de l'étape 2 ?

Que vaut l'indice central de l'étape 2 ?

Que vaut l'intervalle au début de l'étape 3 ?

Que vaut l'indice central de l'étape 3 ?

3. Exemple de mise en œuvre

```
# Algorithme de recherche dichotomique dans une liste triée
# Renvoie True si l'élément est dans la liste
def rechercheDicho(liste ,element):
    # initialisation des variables de depart
    indiceDebut = 0
    indiceFin = len(liste) -1
    while indiceDebut <= indiceFin:
        # Calcul de l'indice et de la valeur centrale
        indiceCentre = ( indiceDebut + indiceFin) // 2
        valeurCentre = liste[indiceCentre]
        # Comparaison de la valeur centrale avec l element
        if valeurCentre == element:
            return True
        elif valeurCentre < element:
            # On garde la sous liste droite
            indiceDebut = indiceCentre + 1
        else:
            # On garde la sous liste gauche
            indiceFin = indiceCentre - 1
    return False

assert rechercheDicho([1, 2, 5, 9, 10, 14, 17, 24, 41] ,5) == True
assert rechercheDicho([1, 2, 5, 9, 10, 14, 17, 24, 41] ,7) == False
```

./scripts/part2/dichotomie1.py

4. Complexité

Nombre d'étapes maximales

Puisqu'on divise à chaque fois l'intervalle restant par deux, on peut prévoir le nombre maximal d'étapes dans le pire des cas lorsqu'on effectue une recherche par dichotomie.

Exemple avec un tableau de 16 valeurs (16 pouvant s'écrire 2^4) :

Etape 1 : une comparaison sur un ensemble de 16 valeurs : 8 d'un côté, le pilier et 7 de l'autre.

Etape 2 : une comparaison sur au pire 8 valeurs : 4 d'un côté, le pilier et 3 de l'autre.

Etape 3 : une comparaison sur au pire 4 valeurs : 2 d'un côté, le pilier et 1 de l'autre.

Etape 4 : une comparaison sur au pire 2 valeurs : le pilier et l'autre valeur.

Etape finale avec la comparaison sur la dernière valeur possible.

Exemple avec un tableau de 32 valeurs (32 pouvant s'écrire 2^5) :

Etape 1 : une comparaison sur un ensemble de 32 valeurs.

Etape 2 : une comparaison sur au pire 16 valeurs.

Etape 3 : une comparaison sur au pire 8 valeurs.

Etape 4 : une comparaison sur au pire 4 valeurs.

Etape 5 : une comparaison sur au pire 2 valeurs.

Etape finale avec la comparaison sur la dernière valeur possible.

On voit donc que si le nombre n de données est dans l'intervalle $2^{X-1} ; 2^X]$, on aura besoin au pire de $X+1$ comparaisons pour trouver ou non si un élément appartient à ce tableau.

On parle d'un coût logarithmique et on peut écrire que cet algorithme est sur le pire des cas **$O(\log_2 n)$** .

Complexité : $O(\log_2 n)$

C'est bien entendu bien mieux que dans le cas de la recherche séquentielle.

A titre d'exemple, sur un tableau d'un million de valeurs,

On a besoin de 1 000 000 de comparaisons dans le pire des cas avec une recherche séquentielle

On a besoin de 20 comparaisons avec une recherche dichotomique

Quelques notions de mathématiques : Il s'agit de la fonction logarithme (ici en base 2).

$$\log_2(16) = 4 \quad (2^4=16)$$

Nb : quand on voit la notation \log c'est \log_{10}

Comment faire avec la calculatrice qui ne dispose que des \log_{10} ? $\log_2(x) = \frac{\log_{10}(x)}{\log(2)}$

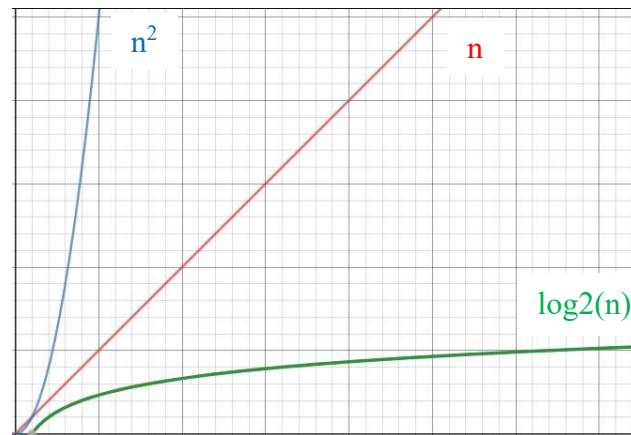
$$\log_2(1.10^6) = \frac{\log_{10}(1.10^6)}{\log(2)} = 19,93$$

Plus généralement on a n tours de boucles pour 2^n valeurs.

Il existe une fonction mathématique, appelée logarithme en base 2.

La fonction : $n \rightarrow \log_2(n)$ est telle que $\log_2(2^n) = n$.

Par conséquent si la taille d'un tableau est 2^n alors la complexité de recherche dichotomique est en $O(\log_2(n))$



5. Terminaison de la boucle While

On a **indiceFin - indiceDebut** ≥ 1 . Celle-ci représente l'amplitude de l'encadrement de l'**élément**.

L'amplitude de l'intervalle décroît à chaque tour de boucle et est à peu près divisé par 2 à chaque tour jusqu'à ce que

indiceFin - indiceDebut = 1


Or la condition d'arrêt de la boucle est **indiceDebut + 1 \geq indiceFin** c'est à dire **indiceFin - indiceDebut ≥ 1**

Donc la boucle se termine au bout d'un nombre fini de tours.

Exercices

🔧 Exercice 1 : modifier la fonction rechercheDico() de manière à ce qu'elle renvoie l'indice de la valeur recherchée. Elle renverra False si elle ne la trouve pas.

🔧 Exercice 2 : Expliquer ce qu'il se passe si la liste est vide.

 Exercice 3 : Effectuer la trace de l'algorithme dans le cas suivant : rechercheDicho([1,2,5,9],5)

#Ligne	indiceDebut	indiceFin	Condition while	IndiceCentre	ValeurCentre	Condition if
5	0					
6	0	3				
7	0	3				