# ACT ALGORITHME DE DICHOTOMIE

Voir l’animation :

<https://pixees.fr/tranche-de-formation-toi-meme-chapitre-3-la-dichotomie-et-les-prenoms/>

## I. Rappels de tris d’une liste

Il existe deux façons de trier une liste en Python.

1. La fonction **sorted()**.

Cette fonction trie la liste et la renvoie. La liste de départ n’est pas modifiée et il faut stocker le résultat dans une autre variable si on doit l’utiliser.

2. La méthode **.sort()**.

Cette méthode trie la liste « en place ». Celle-ci prend la valeur de la liste triée, et la liste de départ est perdue.

***Exemples :***

|  |  |
| --- | --- |
| >>L = [1, 2, 6, 5, 8, 3, 10]  >>L\_triee = sorted(L)  >>L  >>L\_triee | >>L = [1, 2, 6, 5, 8, 3, 10]  >>L.sort()  >>L |

## II. Principe de l’algorithme de dichotomie

### 1. Ecriture de l’algorithme

L’algorithme de dichotomie s’applique à un **tableau trié**.

A chaque étape,

* Diviser le tableau en deux parties égales.
* Prendre la valeur du milieu et comparer avec la valeur cherchée.
* Supprimer la partie du tableau dans laquelle ne se trouve pas la valeur cherchée.

***Travail demandé :***

1. Exécuter cet algorithme pas à pas, *« à la main »*, sur la liste [1 , 5 , 12 , 15 , 17 , 18] dans laquelle on cherche la valeur 18. Recommencer avec la valeur 7.
2. Ecrire en langage naturel, ou pseudo-code, un algorithme de dichotomie permettant de déterminer si un nombre choisi est dans une liste de nombres entiers aléatoires de longueur n (penser à trier la liste).
3. Coder cet algorithme en Python et le tester pour différentes valeurs.

Entrées : tab (tableau dans lequel on cherche) et v (valeur à trouver)

a = 0 (indice de gauche)

b = longueur de la liste -1 (indice de droite)

Tant que la liste n’est pas vide

Prendre la valeur « du milieu », on note m son indice (m = (a+b)//2)

Si v = valeur du milieu

Renvoyer True

Sinon si valeur du milieu < x

a prend la valeur m+1

Sinon (cad valeur du milieu >x)

b prend la valeur m-1

Renvoyer False

Pour savoir si la liste n’est pas vide, on peut vérifier que l’indice de droite est bien inférieur à l’indice de gauche. On continue donc tant que a <= b.

En pseudo code

Entrées : tab (tableau dans lequel on cherche) et v (valeur à trouver)

a 🡨 0 (indice de gauche)

b 🡨 longueur de la liste -1 (indice de droite)

Tant que a <= b :

m 🡨 (a+b)//2

Si v == liste[m] :

Renvoyer True

Sinon si liste[m] < x :

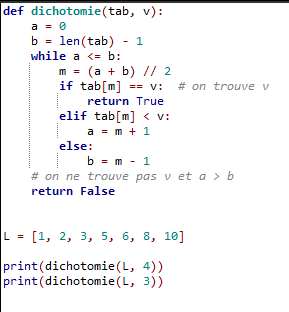
a 🡨 m+1

Sinon :

b 🡨 m-1

Renvoyer False

1. Coder cet algorithme en Python et le tester pour différentes valeurs.



### 2. Correction de l’algorithme

1. Montrer que la propriété suivante est bien un invariant de boucle.

*Si v est présent dans tab, alors il existe a et b tels que tab[a] ≤ v < tab[b]*

Au démarrage, on peut dire que si v appartient à tab, alors v est bien compris en les bornes de tab, donc la propriété est vraie.

Si v est présent dans tab, supposons alors qu’il existe a et b tels que tab[a] ≤ v < tab[b]. On a bien a ≤ b ce qui permet de rentrer dans la boucle while.

On divise le tableau en définissant m = (a + b) // 2.

Puisque tab[a] ≤ v < tab[b], il y a 3 cas possibles :

1er cas : tab[m] == v Alors tab[a] ≤ v < tab[b]

on retourne True

2nd cas : tab[m] < v Alors tab[m + 1] ≤ v < tab[b]

on incrémente alors a : a = m + 1

3ème cas : tab[m] > v Alors tab[a] ≤ v < tab[m – 1]

on décrémente alors b : b = m – 1

Si v appartient à tab, il existe donc toujours a et b tels que tab[a] ≤ v < tab[b].

La propriété est vérifiée : c’est donc l’invariant de boucle.

1. Prouver que l’algorithme se termine après un nombre fini d’étapes. On peut utiliser la variant de boucle d - g où d et g sont les indices des bornes de la liste.

Au départ : d – g ≤ len(tab)

Tant que la condition a ≤ b est vraie, on parcourt la boucle while : à chaque itération, on divise l’intervalle en 2 partie.

1ère itération : L’intervalle se réduit au pire de 1 et on trouve a et b

tels que a - b < d – g

soit a - b ≤ d – g – 1

2nde itération : De même, on arrive à

a – b ≤ d – g – 2

…

nième itération : On arrive à

a – b ≤ d – g - n

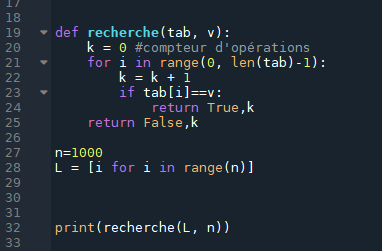
Au bout de n itérations, si on trouve v, alors on sort de la boucle et on retourne True. Sinon, on arrivera à un intervalle vide et on retourne False.

L’algorithme se termine donc au bout d’un nombre fini d’étapes.

### 3. Coût de l’algorithme

On reprend l’exemple précédent.

1. Ecrire un autre programme qui effectue la même tâche mais en parcourant cette fois la liste du premier au dernier élément. Quel est son coût ?



1. Comparer leur temps d’exécution pour différentes valeurs de n et regrouper les résultats dans un tableau :

|  |  |  |
| --- | --- | --- |
| n | Coût Dichotomie  Algo question 3) | Coût standard  Algo question 6) |
| 10 | 8 | 9 |
| 100 | 14 | 99 |
| 1000 | 20 | 999 |
| 10000 | 28 | 9999 |
| 100000 | 34 | 99999 |
| ... |  |  |

1. Le coût est-il linéaire ?

On constate que le coût n’est pas linéaire.

1. A partir des fonctions proposées (calculatrice mode statistique, dans calc, tester les différents modèles de régression, plus r est proche de 1, mieux c’est …), déterminer la complexité de l’algorithme de dichotomie proposé.

En testant les différentes fonctions de la calculatrice, on peut conclure que le coût est logarithmique.