Introduction :

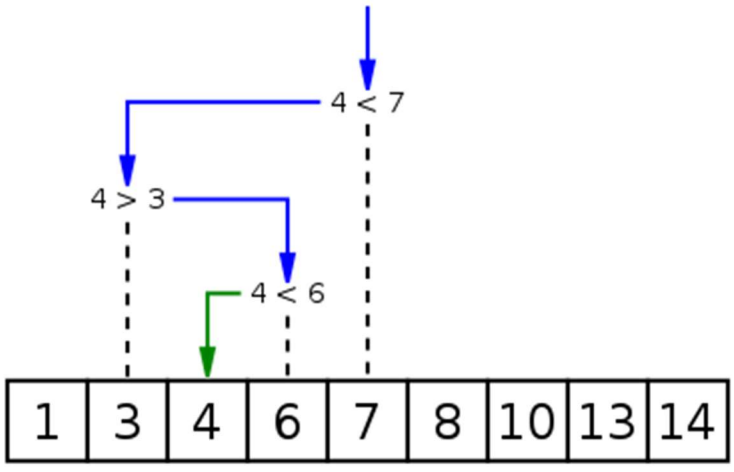
La méthode dichotomique, est une méthode qui consiste simplement à couper en deux parties une base de données, on ne garde que la partie susceptible de contenir l'information recherchée. Ainsi, en réitérant plusieurs fois cette action, on est certain que la base finale contiendra l'information recherchée !

Cette méthode est bien plus efficace que de simplement inspecter les éléments un par un, en effet cela réduit considérablement le nombre d'étapes dans notre algorithme.

Pour rendre cela un peu plus évident, voici un **exemple simple** :

- Je prends un dictionnaire français classique
- Je cherche le mot « Numérique » par exemple, la méthode la plus efficace n'est évidemment pas de tourner pages après pages.
- Je déchire donc mon dictionnaire en deux au niveau de la 13ème lettre (m) étant donné que l'alphabet français est composé de 26 lettres (et 26/2 = 13…)
- Cela signifie que j'aurai un « demi-dictionnaire » contenant les mots de a à m, ainsi qu'un autre « demi-dictionnaire » contenant les mots de n à z !
- « Numérique » se situe dans la catégorie des mots commençant par n, je peux donc jeter le « demi-dictionnaire » contenant les mots de a à m, il ne m'est plus d'aucune utilité.
- Je dispose maintenant d'une base de données contenant des éléments allant de n à z, soit 13 catégories de lettres, je peux donc redécouper en deux parties égales la liste. 13 / 2 = 6.5 dans mon algorithme je choisi de prendre la valeur inférieure soit 6, j'aurais donc un « quart de dictionnaire » contenant les mots de n à s, et un autre allant de t à z.
- Comme vous l'avez compris je garde le « quart de dictionnaire » contenant les mots de n à s.
- Et ainsi de suite jusqu'à obtenir l'élément recherché, petit à petit la machine traite de moins en moins d'éléments !

Représentation :



Programmation :

Algorithme :

```python
import time

tabl_of_number = []
victory=0

for num in range(0, 100001):
    tabl_of_number.append(num)  # Je génère une liste d'entiers allant de 0 à 100 000.

def trouver_bornes(tabl):
    '''
    Cette fontion permet de déterminer la borne inférieur et supérieur d'une liste d'entier.
    '''
    indice_depart = tabl[0]
    indice_fin = tabl[len(tabl)-1]
    return (indice_depart, indice_fin)

def trouver_indice(tabl, valeur, i, j):
    '''
    Fonction qui réalise (à proprement parlé) l'algo. de Dichotomie
    '''
    global victory

    print("\nDepart =", i)  # Affichage des nouvelles bornes dans la console.
    print("Fin =", j)


    rang = int(i + (j-i)/2) # Formule


    print("Médiane =", rang)
    print("Tableau =", tabl)

    if len(tabl) == 2 :
        print("Valeur trouvée à la position :", rang+1)
        victory = 1
    elif rang < valeur :                    # Choix entre les deux parties.
        tabl = tabl[tabl.index(rang):]
    elif rang > valeur :
        tabl = tabl[:tabl.index(rang)]
    elif rang == valeur :
        print("Valeur trouvée à la position :", rang)
        victory = 1
    new_indices = trouver_bornes(tabl)
    return (tabl, new_indices)

bornes = trouver_bornes(tabl_of_number)     # Définition des bornes initiales.

valeur_search = int(input("Quelle valeur voulez-vous rechercher dans la liste : \n"))

# Boucle Principale :
while victory != 1 :
    result = trouver_indice(tabl_of_number, valeur_search, bornes[0], bornes[1])
    tabl_of_number = result[0]
    bornes = (result[1])
    time.sleep(2)       # Je mets un temps d'attente pour rendre l'algo. plus compréhensible.
```

Complexité :

Tous comme l'algorithme de Dichotomie, il existe d'autres méthodes pour **retrouver un élément dans une série ordonnée**, en voici quelques-unes :

Recherche Séquentielle, Hachage, Arbre Binaire, Recherche par interpolation, Matrices Judy, Filtres de Bloom, Arbres de van Emde Boas, …

Mais étant donné que nous travaillons ici avec le langage Python, nous allons comparer le temps d'exécution de l'algorithme de Dichotomie, par rapport à la méthode native « ***list.index(element)*** ».

Pour les besoins de l'expérience, je vais bien évidemment **désactiver l'affichage utilisateur et la temporisation**, afin d'éviter tout ralentissements superflus !

J'ai donc demandé à ma machine (**via l'algo. de Dichotomie**) de **déterminer l'index du nombre « 84 327 » dans ma liste de 100 000 valeurs.**

Puis j'ai réitéré la même problématique à ma machine, mais en lui demandant d'**utiliser la fonction native** cette fois ci !

Et voici le résultat :

	Temps d'exécution total
Algorithme de Dichotomie	1.824 millisecondes
Méthode Native « .index() »	0.684 millisecondes

La méthode native reste donc **la méthode la plus rapide**.