## Chapitre 1
# Recherche dichotomique

*« Plus petit ou plus grand ? »*

## 1. Présentation de l'algorithme

Lorsqu'on cherche si un élément appartient ou non à une liste, il suffit de la parcourir en comparant chacun de ses éléments à celui que l'on cherche.
Cette démarche peut être améliorée si la liste possède des propriétés particulières, notamment si c'est une liste d'entiers triée.
On veut écrire une fonction `recherche_dichotomique` qui :
En entrée prend

- une liste `liste_triee` de $n$ entiers *triée dans l'ordre croissant* ;
- un entier `val`.

Renvoie

- l'indice de `val` dans `liste_triee` si `val` appartient à `liste_triee` ;
- -1 si a n'appartient pas à `liste_triee`.

### Exemple

- `recherche_dichotomique([11, 20, 32, 33, 54], 32)`
  renvoie 2 car 32 est l'élément d'indice 2 de la liste.
- `recherche_dichotomique([20, 32, 33, 54], 40)`
  renvoie -1 car 40 ne figure pas dans la liste.

### Méthode

On compare `val` avec l'élément `m` qui se situe « à peu près au milieu de `liste_triee` ».

- si `val` est égal à `m` c'est gagné, on renvoie l'indice de `m` dans liste_triee;

- sinon si `val > m` on recommence avec la liste des éléments situés après `m`.
- sinon c'est que `val < m` et on recommence avec la liste des éléments situés avant `m`.

On itère ce procédé tant que la liste des valeurs à examiner n'est pas vide. Si on arrive à une liste vide c'est que `val` , n'est pas dans `liste_triee`.

Voici l'algorithme traduit en PYTHON :

**Python**

```python
def recherche_dichotomique(liste_triee, val):
    # début de la plage de valeurs à regarder
    gauche = 0
    # fin de la plage
    droite = len(liste_triee) - 1
    # tant que la plage est non vide
    while gauche <= droite:
        # on prend grosso modo le milieu
        milieu = (gauche + droite) // 2
        # si on trouve val au milieu c'est gagné
        if liste_triee[milieu] == val:
            return milieu
        # si on a dépassé val
        elif liste_triee[milieu] > val:
            # alors on regarde avant
            droite = milieu - 1
        # sinon on regarde après
        else:
            gauche = milieu + 1
    # si on est sorti de la boucle
    # c'est qu'on n'a pas trouvé val
    return -1
```

## 2. Comprendre l'algorithme

Commençons par le faire « tourner à la main » sur un exemple, avec `liste_triee` valant `[1, 3, 4, 8, 9, 13, 20, 21]` .

On cherche la valeur 4.

Pour chaque itération on a noté dans un tableau les valeurs des variables (celles de `gauche` et `droite` avant qu'elles ne soient modifiées) et si la fonction renvoie quelque chose ou non.

| n° d'itér | gauche | droite | milieu | liste_triee[milieu] | valeur renvoyée |
|---|---|---|---|---|---|
| 1 | 0 | 7 | 3 | 8 | NON |
| 2 | 0 | 2 | 1 | 3 | NON |
| 3 | 2 | 2 | 2 | 4 | OUI : 2 |

Ainsi la fonction a renvoyé 2, indice de la valeur 4 dans la liste, au bout de 3 itérations.

On cherche la valeur 15 :

| n°itér | gauche | droite | milieu | liste_triee[milieu] | return ? |
|---|---|---|---|---|---|
| 1 | 0 | 7 | 3 | 8 | NON |
| 2 | 4 | 7 | 5 | 13 | NON |
| 3 | 6 | 7 | 2 | 20 | NON |
| - | 7 | 6 | - | - | OUI : -1 |

La dernière ligne du tableau signifie qu'au bout de la 3$^{e}$ itération, les conditions de boucles ne sont plus vérifiées (car `gauche > droite`) et que -1 est renvoyé.

**Exercice 1**

On cherche la valeur 6 dans la liste précédente.
Complète le tableau (des lignes resteront peut-être vides).

| n°itér | gauche | droite | milieu | liste_triee[milieu] | return ? |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

On cherche la valeur 21, complète le tableau (des lignes resteront peut-être vides).

| n°itér | gauche | droite | milieu | liste_triee[milieu] | return ? |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

## 3. Analyse de l'algorithme

Quatre questions se posent :

1. Pourquoi, lorsque la fonction renvoie un entier positif, est-ce bien la position de `val` dans `liste_triee` ? C'est un problème de *correction*.
2. Quand la fonction renvoie -1, est-ce que cela veut bien dire que `val` n'est pas dans `liste_triee` ? C'est un problème de *complétude*.
3. Pourquoi la boucle *tant que* s'arrête-t-elle toujours ? On dit que c'est un problème de *terminaison*.
4. Enfin, pourquoi cette fonction est-elle plus rapide qu'un parcours des éléments un par un ? C'est un problème de *complexité*.

## 4. Correction de l'algorithme

Quand la fonction renvoie un entier positif, c'est à la ligne 12 , ce qui signifie qu'on a effectivement trouvé `val` dans `liste_triee`, à la position renvoyée.

## 5. Complétude de l'algorithme

Pour prouver que cette fonction est complète, on doit utiliser un *invariant de boucle*.

**Définition**

> Un *invariant de boucle* est une propriété $\mathcal{P}$ dépendant éventuellement des variables du programme.
>
> – $\mathcal{P}$ doit être vraie avant l'entrée dans la boucle ;
>
> – $\mathcal{P}$ doit rester vraie à chaque itération de boucle ;
>
> – à la fin de la boucle, $\mathcal{P}$ doit nous permettre de conclure que la fonction « fait bien ce qu'elle doit faire ».

Dans notre cas voici l'invariant de boucle :

$\mathcal{P}$ : « si `val` est dans `liste_triee` son indice est entre `gauche` et `droite` »

– avant l'entrée dans la boucle `while`, on a `gauche == 0` et `droite == len(liste_triee) - 1` donc P est trivialement vérifiée ;

- dans la boucle, si `liste_triee[milieu] == val` alors on renvoie `val` et la fonction s'arrête et donne bien le résultat attendu ;

- sinon si `liste_triee[milieu] > val` alors puisque la liste est triée, la position de `val` ne peut être qu'entre `gauche` et `milieu-1`, or `droite` est actualisée avec cette valeur, et P reste vraie ;

- de même si `liste_triee[milieu] < val` ;

- En sortie de boucle $\mathcal{P}$ est toujours vérifiée et puisque `gauche > droite` cela signifie que `val` n'est pas dans `liste_triee`.

On a donc prouvé la complétude de notre fonction.

## 6. Terminaison

Pour prouver qu'une boucle *tant que* se termine, *en théorie* on détermine un *variant* de boucle.

**Définition**

> Un variant de boucle est un *entier positif qui décroît strictement à chaque itération de boucle*. On le choisit de sorte à ce que lorsqu'il atteint zéro (ou un, en tout cas une petite valeur) la boucle se termine.

Dans notre cas, le variant de boucle est l'entier `v` défini par `v = droite - gauche` : la condition du `while` est liée à `v` puisque `gauche <= droite` équivaut à `v >= 0`.

Pour montrer que `v` décroît strictement il suffit de montrer que ou bien gauche augmente strictement ou bien droite décroît strictement.
Or lors d'une itération, `m` est toujours entre `gauche` et `droite` (au sens large) et

- soit on trouve que `liste_triee[m]` vaut `val` et la boucle s'arrête ;

- sinon ou bien `gauche` devient `m + 1` donc augmente strictement, ou bien `droite` devient `m - 1` donc décroît strictement.

Ainsi les valeurs de v décroissent strictement, donc finissent (si on ne trouve pas `val`) par atteindre zéro et la boucle se termine.
On dit qu'on a prouvé la *terminaison* de la fonction.

## 7. Complexité

On va ici évaluer le nombre d'étapes nécessaires au déroulement de la fonction.
On va raisonner dans le pire des cas : `val` n'appartient pas à la liste.
À chaque itération de boucle, le nombre de valeurs qui restent à examiner) est au moins divisé par 2 et lorsque cette valeur vaut 1, c'est qu'on est à la dernière itération de boucle et on est sûr ou bien de trouver `val` à cet endroit, ou bien on sort de la boucle et on renvoie `-1` .

Ainsi, pour une liste triée de taille $n$, le nombre d'itérations de la boucle dans le pire des cas, c'est le plus petit entier $k$ tel que $2^k$ dépasse $n$.

Pour une liste de longueur 2 on est sûrs d'arriver au résultat en 2 itérations, pour une liste de longueur 4, en 3 itérations et en généralisant, si la liste est de longueur $2^n$, en $n+1$ itérations.

Pour un tableau de longueur 1000, puisque $2^9 < 1000 < 2^{10}$, on est sûr d'arriver au résultat au plus en 10 itérations.

**Définition**

> Soit $n$ un entier naturel non nul, on appelle *logarithme en base 2* de $n$ l'unique réel $x$ solution de
> $$2^x = n$$
> Ce nombre $x$ est noté $\log_2(n)$.

Ce que l'on vient de prouver, c'est que pour une liste de taille $n$, la fonction `recherche_dichotomique` nécessitera au plus $E(\log_2(n)) + 1$ itérations pour déterminer si oui ou non une valeur appartient à cette liste ($E$ représente la fonction *partie entière*).

**Propriété**

> Soit une liste triée de longueur $n \in \mathbb{N}^*$.
> Soit $p$ le nombre de bits nécessaires pour écrire $n$ en base 2.
>
> La recherche dichotomique d'une valeur dans la liste nécessite **au plus** $p$ accès à cette liste.

Pour cette raison la complexité de l'algorithme de recherche dichotomique est dite *logarithmique*. C'est bien mieux que celle de la recherche simple.

### Exercice 2 : efficacité de l'algorithme

1. Dans une liste triée de taille 10 000, en combien d'étapes l'algorithme de recherche dichotomique s'arrête-t-il *dans le pire des cas* ?
2. Même question pour une liste de taille 100 000 et pour une liste de taille 1 000 000.