# Recherche dichotomique

Christophe Viroulaud

Première - NSI

**Algo 03**

Rechercher un élément dans un tableau est une opération courante. Cette tâche a un coût qui dépend de la taille du tableau.

| 3 | 8 | 7 | 1 | 9 | 5 | 18 | 2 | 10 | 4 |
|---|---|---|---|---|---|---|---|---|---|

- recherche de 3,
- recherche de 9,
- recherche de 6.

Cependant, si le tableau est déjà trié est-il possible d'accélérer la recherche ?

| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 | 10 | 18 |
|---|---|---|---|---|---|---|---|---|---|

Comment implémenter une recherche efficace dans un tableau trié ?

# Sommaire

# Génération des données

Imaginons un supermarché qui référence chaque article par un entier. Les références, au nombre de cent mille, sont contenues dans un tableau.

12

6780

376

134900

**Activité 1 :** Construire par compréhension un tableau de cent mille entiers compris entre 0 et 1000000.

# Correction

```
1 from random import randint
2
3 entiers = [randint(0, 1000000) for _ in range(100000)]
```

Jeu de données

# Sommaire

# Recherche dans les données





Pour vérifier la présence d'une valeur dans les données, il faut parcourir le tableau élément par élément.

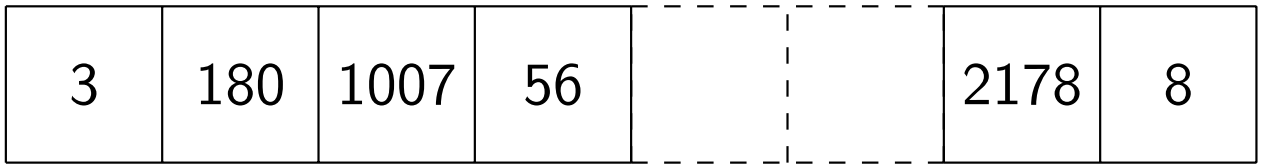



FIGURE 1 – Parcours séquentiel

# Dans le pire des cas

| nombre d'éléments | nombre de comparaisons |
|---|---|
| 100 | 100 |
| 10000 | 10000 |
| 1000000 | 1000000 |

## À retenir

Dans le pire des cas le nombre d'opérations de la recherche dépend du nombre d'éléments.
La complexité est **linéaire**.

**Activité 2 :**

1. Écrire la fonction `recherche_classique(tab: list, cherche: int)` $\rightarrow$ `bool` qui renvoie `True` si l'entier `cherche` est présent dans le tableau.
2. Tester la fonction : vérifier si le nombre 575000 a été choisi par une personne.

# Correction



```
1 def recherche_classique(tab: list, cherche: int) -> bool:
2     """
3     Renvoie True si 'cherche' est dans 'tab'
4     """
5     for element in tab:
6         if element == cherche:
7             return True
8     # à la fin de la boucle on n'a pas trouvé 'cherche'
9     return False
```

Code 1 – Création de la fonction

```
1 entiers = [randint(0, 1000000) for _ in range(100000)]
2 recherche_classique(entiers, 57500)
```

Code 2 – Utilisation de la fonction

**Activité 2 :** Dans le programme principal, créer une variable `COMPTEUR` initialisée à 0. Cette variable de test sera utilisée *dans la fonction* pour compter le nombre d'itérations de la boucle de recherche. On parle alors de **variable globale** car elle n'est pas propre à la fonction. Il faudra ajouter le code 3 au début de la fonction.

```
1 global COMPTEUR
```

Code 3 – Déclaration d'une variable globale

# Correction





```
1  COMPTEUR = 0
2
3  def recherche_classique(tab: list, cherche: int) -> bool:
4      """
5      Renvoie True si 'cherche' est dans 'tab'
6      """
7      global COMPTEUR
8      for element in tab:
9          COMPTEUR += 1
10         if element == cherche:
11             return True
12     # à la fin de la boucle on n'a pas trouvé 'cherche'
13     return False
```

```
1  print(recherche_classique(entiers, 57500))
2  print(COMPTEUR)
```

## À retenir

La variable `COMPTEUR` est utilisée ici uniquement pour effectuer des tests.

**D'une manière générale, modifier une variable globale dans une fonction est une mauvaise pratique.**

# Sommaire

# Des données ordonnées

Considérons maintenant que les références sont triées par ordre croissant au fur et à mesure de leur ajout dans le tableau de données.

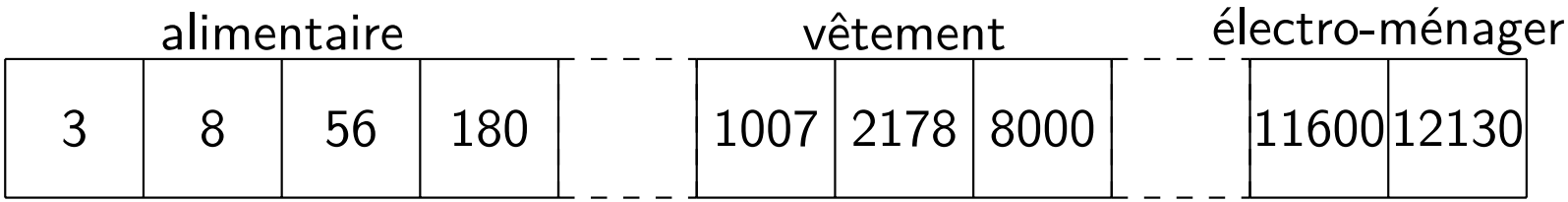



FIGURE 2 – Références triées

**Activité 3 :** Pour simplifier nous allons utiliser la méthode `sort` pour trier les données.

1. Construire par compréhension un tableau de cent mille entiers compris entre 0 et 1000000.
2. Trier le tableau.

# Correction





```
1 entiers = [randint(0, 1000000) for _ in range(100000)]
2 entiers.sort()
```

Jeu de données

# Sommaire

# Recherche dichotomique

Les données étant triées, le principe de la dichotomie, pour chercher la présence d'un élément, consiste à :

- couper le tableau en deux parties égales,
- ne garder que la partie contenant l'élément,
- répéter l'opération jusqu'à trouver l'élément ou bien avoir une partie vide.

Cherchons 302 dans le tableau suivant :

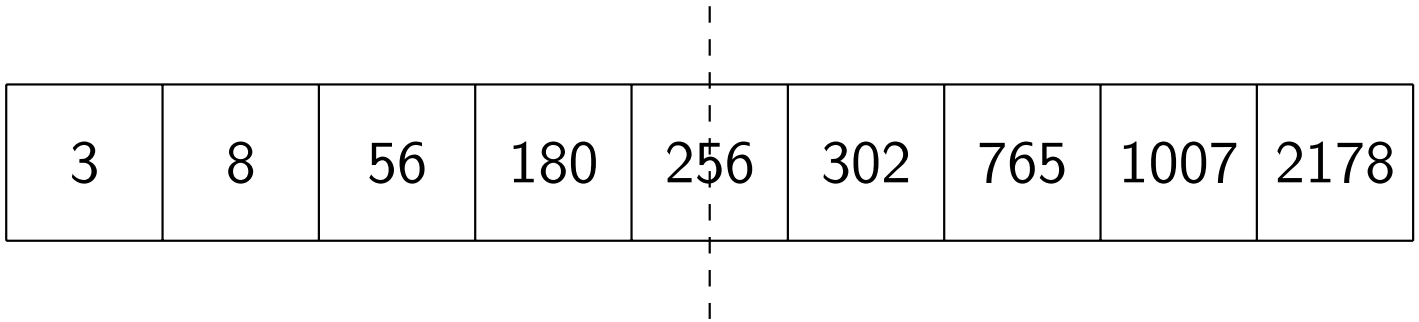



FIGURE 3 – Séparons les données en deux parties

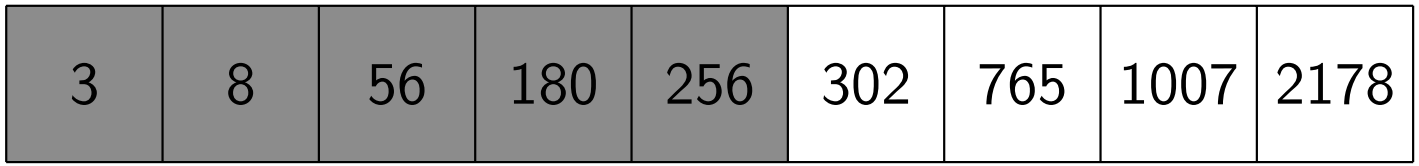



FIGURE 4 – 256 n'est pas le nombre recherché et il est inférieur à 302

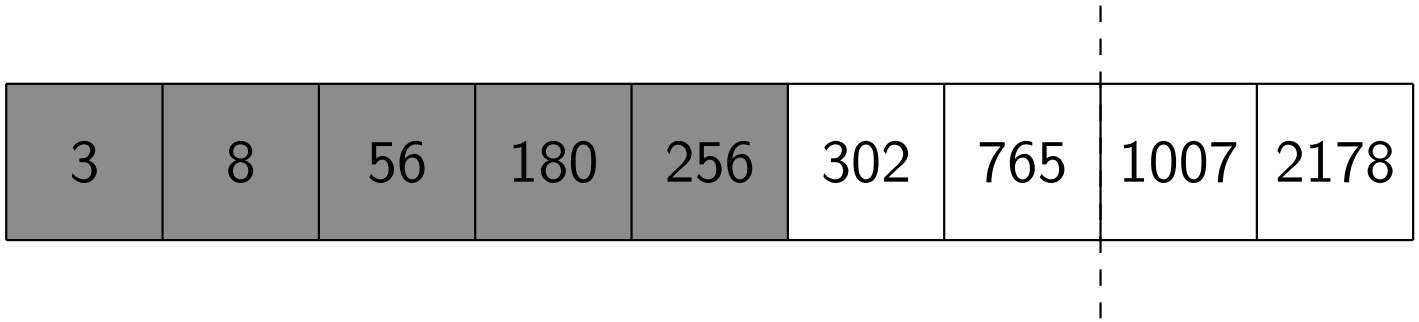



FIGURE 5 – Séparons les données restantes en deux parties

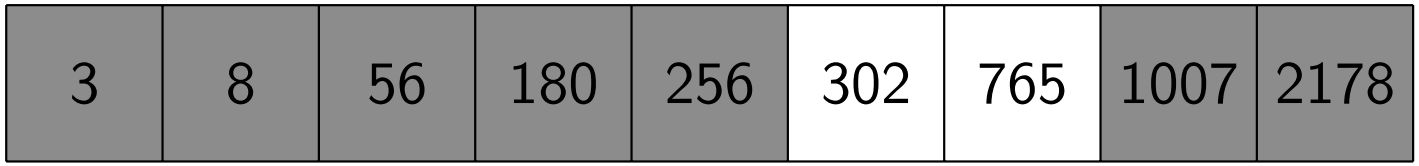



FIGURE 6 – Nous pouvons éliminer la partie supérieure.

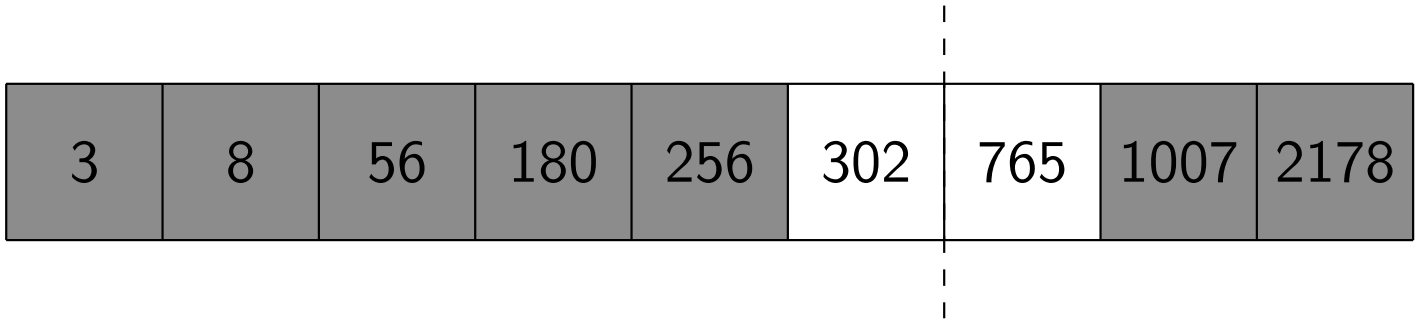



FIGURE 7 – Dernière séparation

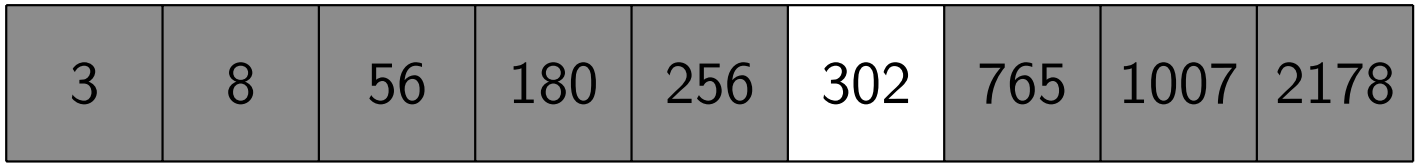



FIGURE 8 – 302 a été trouvée en trois itérations

## Remarque

En pratique, on utilise les indices pour trouver le milieu.

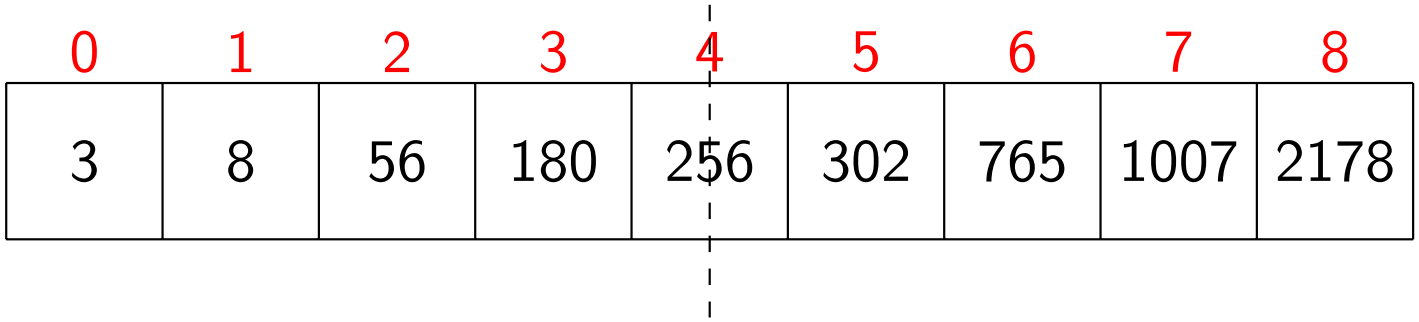



FIGURE 9 – $\frac{8+0}{2} = 4$ l'indice médian est 4

```
1 i_debut = 0
2 i_fin = 8
```

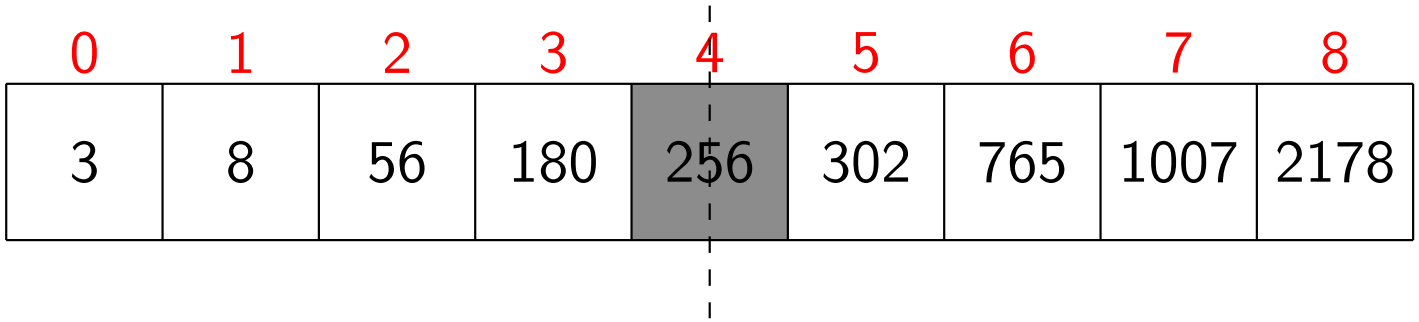



FIGURE 10 – 256 n'est pas le nombre recherché

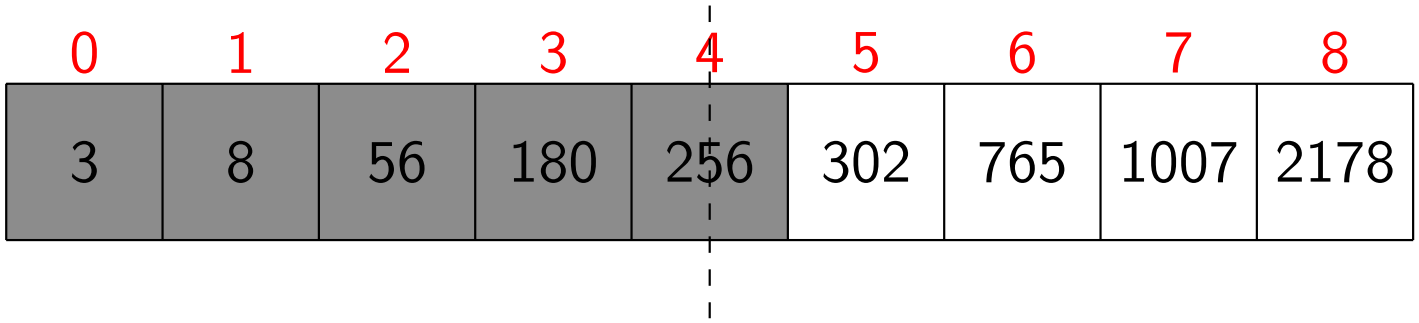



FIGURE 11 – 256 est inférieur au nombre recherché.

```
1 i_debut = 5
2 i_fin = 8
```

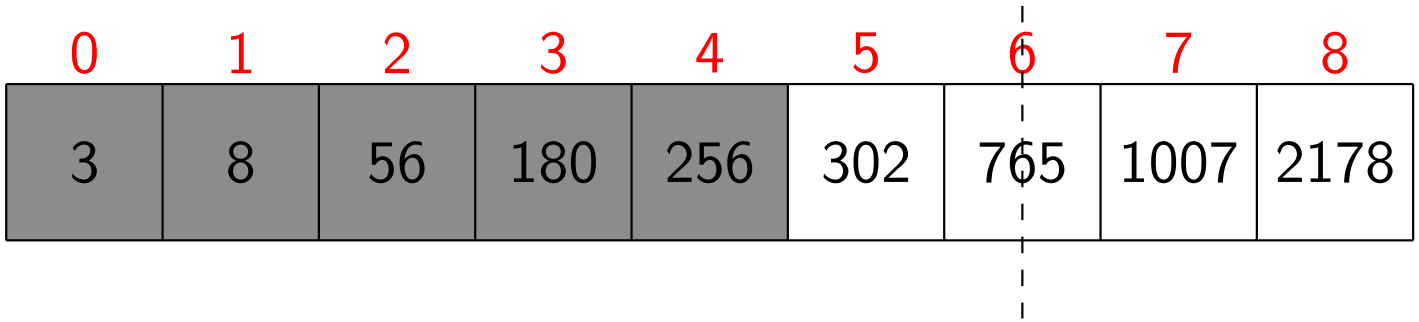



FIGURE 12 – $\frac{8+5}{2} = 6$ l'indice médian est 6

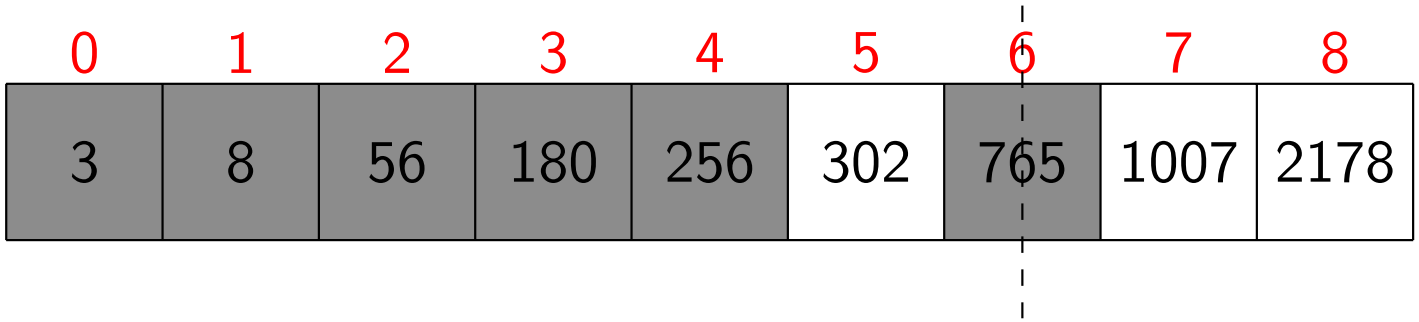



FIGURE 13 – 765 n'est pas le nombre recherché.

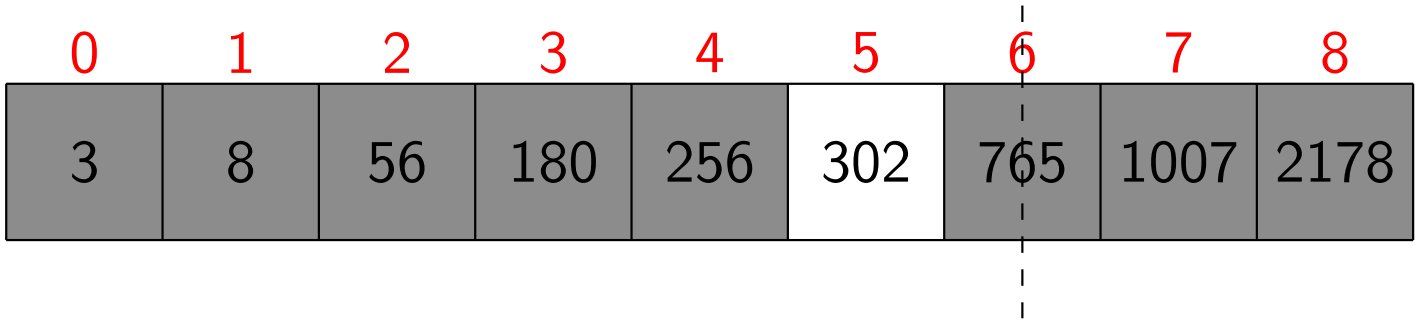



FIGURE 14 – 765 est supérieur au nombre recherché.

```
1 i_debut = 5
2 i_fin = 5
```

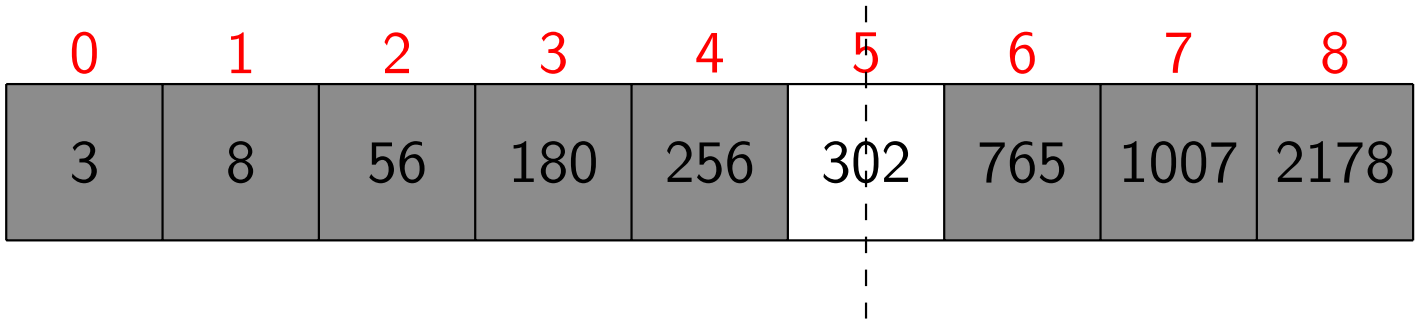



FIGURE 15 – $\frac{5+5}{2} = 5$ l'indice médian est 5.

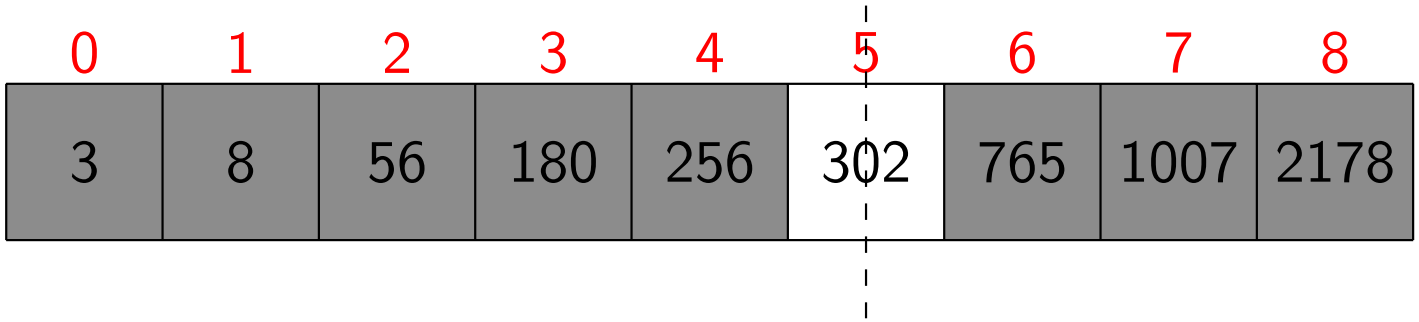



FIGURE 16 – On a trouvé l'élément.

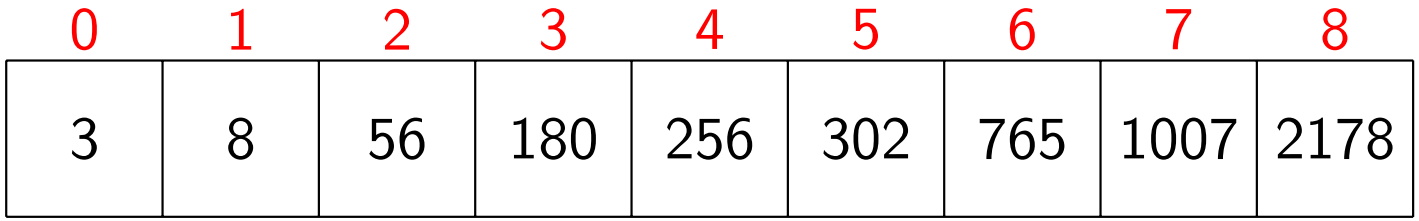
**Activité 4 :** Écrire la fonction `recherche_dicho(tab: list, cherche: int)` $\rightarrow$ `bool` qui applique le principe de la dichotomie :

- Définir les indices `i_debut` et `i_fin`.
- Tant que `i_fin` $\geq$ `i_debut`
  - Calculer `i_milieu`
  - Vérifier si l'élément d'indice `i_milieu` est celui cherché
  - Sinon redéfinir `i_debut` et `i_fin` pour ne garder que la partie contenant l'élément cherché.

# Correction

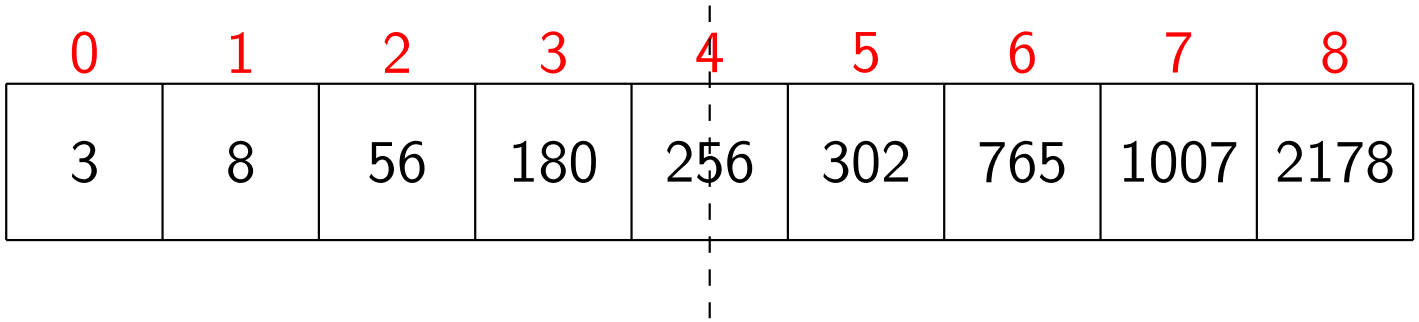






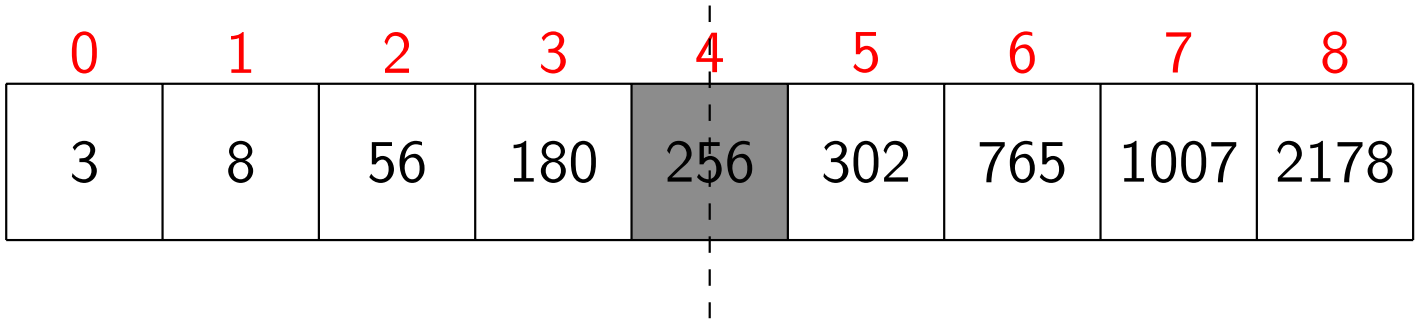

```
1 def recherche_dicho(tab: list, cherche: int) -> bool:
2     i_debut = 0
3     i_fin = len(tab)-1
```

# Correction



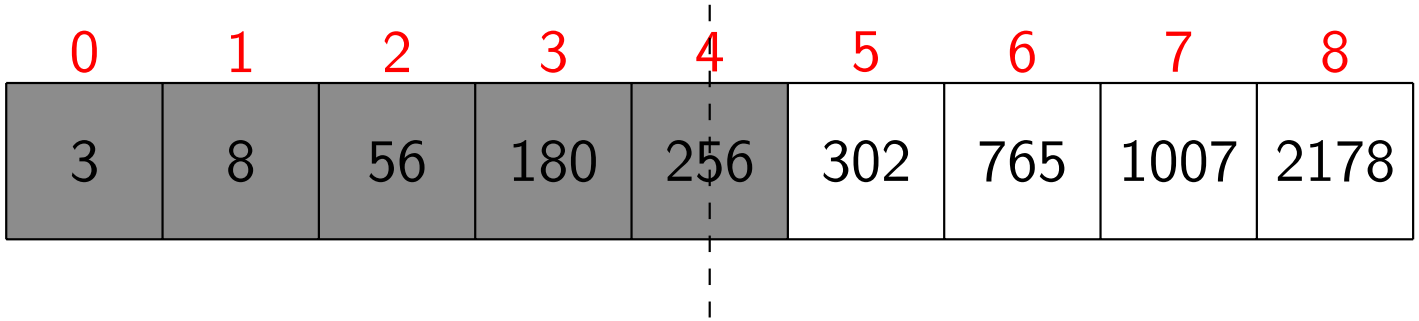

```
1        while i_fin >= i_debut:
2            i_milieu = (i_debut+i_fin) // 2
```

# Correction




```
1          if cherche == tab[i_milieu]:
2              return True
```

# Correction




```
1            elif cherche < tab[i_milieu]:
2                i_fin = i_milieu-1
3            else:  # cherche > tab[i_milieu]
4                i_debut = i_milieu+1
```

# Correction





```
1  def recherche_dicho(tab: list, cherche: int) -> bool:
2      i_debut = 0
3      i_fin = len(tab)-1
4      while i_fin >= i_debut:
5          i_milieu = (i_debut+i_fin) // 2
6          if cherche == tab[i_milieu]:
7              return True
8          elif cherche < tab[i_milieu]:
9              i_fin = i_milieu-1
10         else:  # cherche > tab[i_milieu]
11             i_debut = i_milieu+1
12     # à la fin de la boucle on n'a pas trouvé 'cherche'
13     return False
```

# Sommaire

# Efficacité

**Activité 5 :**

1. En utilisant une variable `COMPTEUR`, compter le nombre d'itérations de la boucle de recherche dichotomique.
2. Tester pour différentes tailles de tableau.

# Correction





```
1  COMPTEUR = 0
2
3  def recherche_dicho(tab: list, cherche: int) -> bool:
4      global COMPTEUR
5      i_debut = 0
6      i_fin = len(tab)-1
7      while i_fin >= i_debut:
8          COMPTEUR += 1
9          i_milieu = (i_debut+i_fin) // 2
10         if cherche == tab[i_milieu]:
11             return True
12         elif cherche < tab[i_milieu]:
13             i_fin = i_milieu-1
14         else:  # cherche > tab[i_milieu]
15             i_debut = i_milieu+1
16     # à la fin de la boucle on n'a pas trouvé 'cherche'
17     return False
```

```
1 print(recherche_dicho(entiers, 57200))
2 print(COMPTEUR)
```

Code 5 – Utilisation de la fonction

À chaque itération la quantité de données (notée **n**) à étudier est divisée par deux. Dans le pire des cas, on divise jusqu'à ce que la taille de la partie restante soit inférieure ou égale à 1.

$$\frac{n}{2^x} = 1$$

$$\Leftrightarrow n = 2^x$$

$$n = 2^x$$

**Activité 6 :**

1. Encadrer la valeur de $x$ par deux entiers, si le tableau contient $n = 10000$ éléments.
2. Effectuer le même encadrement pour cent mille, un million d'éléments.

# Correction

$$2^{13} = 8192 < x < 2^{14} = 16384$$

# Dans le pire des cas

| nombre d'éléments | nombre de comparaisons |
|---|---|
| 10 | 3-4 |
| 100 | 6-7 |
| 1000 | 9-10 |
| 10000 | 13-14 |
| 100000 | 16-17 |
| 1000000 | 19-20 |

## À retenir

La complexité temporelle de la recherche dichotomique est **logarithmique** :

$$\log_2 n = x$$

# Code complet

Le code complet se trouve ici.