Chapitre 3
Recherche dichotomique

Lorsqu'un tableau est trié, il devient plus facile d'effectuer certaines opérations. En particulier, il devient très facile (efficace) de rechercher un élément. Pour cela, il suffit de comparer la valeur cherchée à la valeur située au milieu du tableau. Si elle est plus petite, alors on peut restreindre la recherche à la partie gauche du tableau, sinon à la partie droite. En répétant ce procédé, la taille de la zone de recherche est divisée par deux à chaque fois. On va donc très rapidement trouver la valeur recherchée ou constater qu'elle n'est pas dans le tableau. On appelle cela la *recherche dichotomique*.

Ce principe est connu en informatique sous le nom « *diviser pour régner* » et il est appliqué dans de nombreux algorithmes, la recherche dichotomique en est un exemple simple.

I. Algorithme de recherche dichotomique

On cherche à écrire une fonction `recherche_dichotomique(v, T)` qui recherche une valeur `v` dans un tableau trié `T` dont voici la spécification.

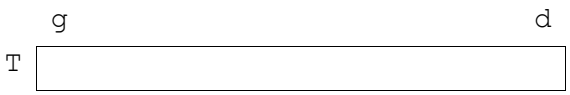
Spécification de l'algorithme

Entrée : tableau `T` (de taille n constitué d'entiers), un entier `v`
Sortie : un entier `m` ou `None`
Rôle : renvoyer une position `m` de `v` dans `T` ou `None` si $v \notin T$
Précondition : `T` est **trié** par ordre croissant
Postcondition : Si None est renvoyé alors $v \notin T$, sinon `T[m] = v`

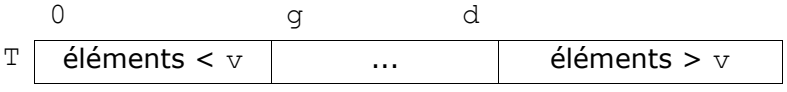
Mise en place de l'algorithme

On va noter `g` et `d` les indices définissant la zone de recherche de `v` dans le tableau `T`.

Au départ, on a donc `g = 0` et `d = len(T) - 1` :



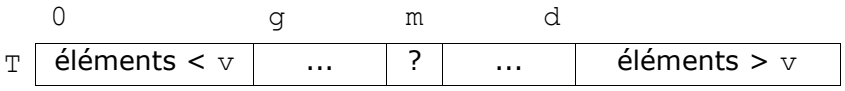
Après un certain nombre d'étapes, on se trouve dans la situation suivante :



Le programme doit répéter ce principe de dichotomie tant que la zone de recherche n'est pas vide, donc tant que $g \leq d$. L'utilisation d'une répétitive `tant que` est toute indiquée !

```
while g <= d:
```

Il faut, à chaque étape, comparer l'élément `v` avec l'élément central de la zone de recherche. On va noter `m` l'indice de l'élément central. On a la situation suivante :



Pour calculer la position médiane m, il suffit de faire la moyenne de g et d :

```
m = (g + d) // 2
```

Il faut faire attention, il s'agit d'une division entière pour être certain de ne pas obtenir de valeurs décimales (`m` est nécessairement un entier).

Il reste ensuite à comparer `v` avec `T[m]`. Trois cas de figure se présentent selon que v soit plus petite, plus grande ou égale à `T[m]`. Si `v` est plus petite que `T[m]` alors on se restreint à la moitié gauche en modifiant la valeur de `d` en conséquence :

```
if v < t[m]:
    d = m - 1
```

Si en revanche elle est plus grande, on se restreint à la moitié droite et on modifie la valeur de `g` en conséquence :

```
if v > t[m]:
    d = m + 1
```

Enfin, si les deux valeurs sont égales, c'est qu'on a trouvé la valeur `v` en position `m` et il donc il suffit de renvoyer `m`.

```
else:
    return m
```

Il se peut également qu'on finisse par sortir de la boucle car la condition `g <= d` devient fausse. Cela signifie que la valeur `v` ne se trouve pas dans `T` et dans ce cas on renvoie `None`.

Algorithme de recherche dichotomique

```
def recherche_dichotomique(v, T):
    """renvoie une position de v dans le tableau T, supposé trié,
    et None si v ne s'y trouve pas"""
    g = 0
    d = len(T) - 1
    while g <= d:
        m = (g + d) // 2
        if v < T[m]:
            d = m - 1
        elif v > T[m]:
            g = m + 1
        else:
            return m
    return None
```

Un exemple

Vous allez appliquer l'algorithme pour chercher la valeur 11 dans le tableau `T` suivant.

`T = [2, 7, 11, 12, 12, 18, 20, 31, 55, 57, 60, 70, 75, 77, 100].`

1. Complétez le tableau suivant avec l'état des variables à l'issue de chaque itération.

	`g <= d` ?	`m`	`T[m]`	Moitié à conserver (droite ou gauche ou fin ?)	`g`	`d`
Avant l'itération 1					0	14
Après l'itération 1	True	7	31	Gauche (car 11 < 31)	0	6
Après l'itération 2	True	3	12	Gauche (car 11 < 12)	0	2
Après l'itération 3	True	1	7	Droite (car 11 > 7)	2	2
Après l'itération 4	True	2	11	Fin (on a trouvé 11)		

2. Quelle est la valeur renvoyée par l'algorithme ?

II. Correction de l'algorithme

Correction partielle

La correction partielle de cet algorithme est admise. A titre d'information, un invariant de boucle peut être : `0 <= g` et `d < len(T)` et `v` ne peut se trouver que dans `T[g..d]`

Terminaison (à savoir faire)

Montrons que la répétitive `while g <=d:` termine. Un invariant possible est : `d - g`.

En effet :

- D – g est un entier de manière évidente
- Avant la première itération : d – g = len(T) – 1. Si T est vide alors d – g = -1 et on n'entre pas dans la répétitive donc l'algorithme termine. Si T n'est pas vide, alors d – g ≥ 0 et on entre dans la répétitive.
- A chaque itération :
  - soit d diminue d'une unité et alors d – g également ;
  - soit g augmente d'une unité et alors d – g diminue d'une unité ;
  - soit la valeur v est trouvée et le return termine l'algorithme.

La quantité `d - g` est donc bien un variant puisque c'est une quantité entière qui diminue strictement donc on finira par avoir `d < g` et une sortie de boucle, si on n'a pas trouvé la valeur v avant bien sûr.

III. Efficacité de l'algorithme

Pour calculer le coût de cet algorithme on peut compter le nombre d'itérations de la boucle `while`, c'est-à-dire le nombre de valeurs du tableau `T` qui ont été examinées. On se place dans le pire des cas, donc dans le cas où la valeur `v` ne se trouve pas dans `T` puisque c'est ce qui va occasionner le plus grand nombre d'itérations.

Prenons un exemple pour comprendre. Si on cherche de cette façon un mot dans un dictionnaire de 60 000 mots. Après 1 comparaison, on le cherche dans un ensemble d'au plus 30 000 mots ; après 2 comparaisons, on le cherche dans un ensemble d'au plus 15 000 mots ; etc. On arrive très vite au mot cherché ou on constate qu'il ne se trouve pas dans le dictionnaire.
En combien d'étapes au pire ? réponse : 16 car en partant de 60 000, il faut diviser 16

fois successivement par 2 pour obtenir un nombre inférieur ou égal à 1. ($\frac{60000}{2^{15}} > 1$ et $\frac{60000}{2^{16}} \leq 1$).

n	k
10	4
100	7
1000	10
1 000 000	20
1 000 000 000	30

Figure 1 - Nombre maximal d'itérations k pour une taille de tableau n

On voit qu'il s'agit d'un algorithme extrêmement efficace puisque même pour rechercher une valeur dans un tableau de taille 1 milliard, il suffit d'examiner 30 valeurs dans le pire des cas : la recherche dichotomique sera donc instantanée. A titre de comparaison, avec l'algorithme de recherche séquentielle il faudrait examiner 1 milliard de valeurs dans le pire des cas !

Coût de l'algorithme (hors programme)

Si on considère un tableau T de taille n, alors le nombre d'itérations correspond au nombre de fois qu'il faut diviser n par 2 pour obtenir un nombre inférieur ou égal à 1.

Autrement, on cherche le plus petit entier k tel que :

$$\begin{aligned} &\frac{n}{2^k} \leq 1 \\ \Leftrightarrow\ &n \leq 2^k \\ \Leftrightarrow\ &\log(n) \leq \log(2^k) \\ \Leftrightarrow\ &\log(n) \leq k\log(2) \\ \Leftrightarrow\ &\frac{\log(n)}{\log(2)} \leq k \end{aligned}$$

On note $\log_2(n)$ le quotient $\frac{\log(n)}{\log(2)}$ et ce nombre s'appelle le logarithme en base 2 de n.

Le nombre d'itérations est donc égale au plus petit entier supérieur ou égal à $\log_2(n)$.
Le coût de la recherche dichotomique est dit logarithmique, noté $O(\log_2(n))$. Il s'agit d'un coût nettement inférieur à celui de la recherche séquentielle qui est linéaire (O(n)).

A retenir

La **recherche dichotomique** permet de rechercher une valeur dans un tableau trié. Le principe consiste à **couper en deux** à chaque fois la portion du tableau dans laquelle s'effectue la recherche. Cet algorithme est **très efficace**. Il ne faut que quelques dizaines de comparaisons pour chercher une valeur dans un tableau en contenant des milliards.

Ressources :

- Documents ressources du DIU EIL, Université de Nantes.
- *Numérique et Sciences Informatiques*, T. BALABONSKI, S. CONCHON, J.-C. FILLIATRE, K. NGUYEN, Ellipses.