Numérique et sciences informatiques
Classe de première

D Algorithmique
3 Recherche dichotomique dans un tableau trié

I L'algorithme de recherche dichotomique dans un tableau trié

I.1 Le problème à résoudre

• On dispose d'un tableau `tab` (donc de type **`list`**) homogène de valeurs triées dans l'ordre croissant. On supposera qu'il s'agit d'un tableau d'entiers (objet de type **`int`**) mais il pourrait aussi s'agir d'un tableau de « nombres à virgule » (**`float`**), d'un tableau de chaînes de caractères **`str`** ou de valeurs d'un autre type que l'on peut comparer avec les opérateurs `<` et `>`.

• On souhaite écrire une fonction qui permet de déterminer si une valeur donnée du type concerné (**`int`** dans nos exemples) se trouve dans le tableau `tab`.

* Si elle s'y trouve, la fonction doit renvoyer un indice correspondant à un emplacement du tableau où elle se trouve.
* Si elle ne s'y trouve pas, la fonction doit renvoyer `None`.

• On va ici exploiter le fait que le tableau `tab` est trié pour construire une algorithme plus efficace (c'est-à-dire plus rapide) que l'algorithme de recherche séquentielle que nous avons déjà vu pour un tableau quelconque (non trié).

I.2 La méthode suivie

• Nous allons décrire la méthode de **recherche dichotomique** (ou **par dichotomie**). Le terme de « dichotomie » provient du grec ancien *dikhotomia* qui signifie « division en deux parties ». Cette méthode consiste en effet à « couper en deux parties » la zone de recherche à chaque étape. Si on n'a pas trouvé la valeur recherchée à cette étape, on dispose alors d'un critère pour décider dans laquelle de ces deux parties on va continuer à chercher.

• L'idée centrale de la recherche est donc comparer la valeur recherchée `val` à la valeur qui se trouve « au milieu » de la zone de recherche dans le tableau (au début de la recherche cette zone est formée de la totalité du tableau).

• Notons

* `tab` le tableau dans lequel on recherche ;
* `lg` la longueur du tableau, c'est-à-dire son nombre d'éléments ;
* `g` (pour "gauche") le plus petit indice de la zone de recherche. Au début de l'algorithme, on l'initialise à `0` ;
* `d` (pour "droite") le plus grand indice de la zone de recherche. Au début de l'algorithme, on l'initialise à `lg-1` ;
* `m` l'indice correspondant au milieu de la zone de recherche. La valeur située à cet indice sera donc notée `tab[m]`.

• On a alors trois situations possibles :

* Si `val` est strictement inférieur à `tab[m]` alors, puisque le tableau `tab` est trié, la valeur `val` ne peut se trouver que *dans la « première partie »*, c'est-à-dire celle située entre le début de la zone de recherche et l'indice `m-1` inclus (si elle se trouve dans le tableau). En effet, toutes les valeurs du tableau situées *après* l'indice `m` seront supérieures ou égales à

`tab[m]` donc strictement supérieures à `val`. On poursuit donc la recherche dans cette partie du tableau exclusivement.

* Si `val` est strictement supérieur à `tab[m]` alors la valeur ne peut se trouver que *dans la « deuxième partie »*, c'est-à-dire celle située entre l'indice `m+1` inclus et la fin de la zone de recherche. En effet, toutes les valeurs du tableau situées *avant* l'indice `m` seront inférieures ou égales à `tab[m]` donc strictement inférieures à `val`.

* Si aucune de ces deux conditions n'est vérifiée, alors `tab[m]` est égal à `val`. On a alors trouvé un indice `m` tel que `tab[m]` est égal à `val` et on renvoie donc `m`.

• Dans les deux premiers cas, on n'a pas encore trouvé la valeur `val` mais on a réduit la recherche à une zone (au moins) deux fois plus petite. Il suffit donc de renouveler cette procédure pour que, à chaque étape, la recherche soit effectuée dans une zone dont la longueur est (au moins) divisée par deux. On aura donc une recherche bien plus rapide que si on recherche la valeur en la comparant séquentiellement à toutes les valeurs du tableau.

• Dans le pire des cas, notamment dans le cas où la valeur `val` ne se trouve pas dans le tableau `tab`, on arrivera à un moment où on recherche la valeur dans une zone vide et on pourra alors renvoyer `None`.

I.3 Implémentation de la méthode en Python

• On obtient le code suivant :

```
def recherche_dicho(tab, val) :
"""Recherche dans un tableau trié par dichotomie.
* Tab est de type list
* renvoie l'indice d'un élément de valeur 'val' ou
'None'."""
    g = 0
    d = len(tab)-1
    while g <= d : # la zone de recherche n'est pas vide
        m = (d + g)//2 # indice du milieu de zone
        if val < tab[m] : # on recherche à gauche
            d = m-1
        elif val > tab[m] : #on recherche à droite
            g = m+1
        else : # tab[m]==val : on a trouvé
            return m
    return None # 'val' ne se trouve pas dans le tableau
```

• *Commentaires :*

* Dans ce code, les variables `g` et `d` désignent respectivement l'indice de gauche et l'indice de droite de la zone de recherche.

* `m` est l'indice de milieu de la zone de recherche. On le calcule à partir des variables `g` et `d` par la formule suivante :

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

Il est indispensable ici d'utiliser l'opérateur `//` qui donne le quotient de la division euclidienne plutôt que l'opérateur `/` de la division classique. En effet, puisque `m` désigne un indice du tableau, il doit toujours être de type **`int`**.

* Si la valeur `val` ne se trouve pas dans le tableau, on arrivera à un moment où la variable `d` sera strictement inférieure à `g`. Dans ce cas, la zone de recherche est vide et on renvoie `None`.

II Propriétés de l'algorithme de recherche dichotomique

II.1 Correction de l'algorithme

a) On reste dans les limites du tableau

Notons `lg` la longueur du tableau c'est-à-dire la valeur renvoyée par `len(tab)`.

• A chaque étape, on effectue l'affectation `m = (d + g)//2` après avoir vérifié que l'on a bien `g <= d`. Dans ces conditions on a alors nécessairement les inégalités suivantes :

$$g \leq m \leq d$$

• De plus, au début de l'algorithme, on a $g = 0$ et $d = lg - 1$.

* La valeur de la variable `g` *ne peut qu'augmenter* puisqu'elle ne change de valeur que par l'instruction `g = m+1`. Donc à tout moment on a

$$g \geq 0$$

* La valeur de la variable `d` *ne peut que diminuer* puisqu'elle ne change de valeur que par l'instruction `d = m-1`. Donc à tout moment on a

$$d \leq \lg - 1$$

• Finalement les inégalités suivantes sont toujours vérifiées :

$$0 \leq g \leq m \leq d \leq \lg - 1$$

La variable `m` exprime donc toujours un indice du tableau et l'expression `tab[m]` a donc toujours un sens.

• *Remarque :* On dit que les inégalités $0 \leq g \leq m \leq d \leq \lg - 1$ sont des **invariants de boucle**, c'est-à-dire qu'elles restent vraies à chaque passage dans la boucle.

b) La valeur renvoyée vérifie bien les conditions attendues

• Si on renvoie une valeur d'indice (une valeur de type **`int`**) alors cette valeur est bien un indice `m` tel que `tab[m] == val`. En effet, on n'exécutera l'instruction `return m` qu'après avoir évalué successivement les conditions `tab[m] < val` puis `tab[m] > val` et avoir obtenu `False` à chaque fois.

• On renvoie la valeur `None` si et seulement si la valeur `val` ne se trouve pas dans le tableau `tab`. En effet, on n'exécutera l'instruction `return None` qu'après avoir obtenu `False` à la condition `g <= d` et donc seulement dans le cas où, du fait de l'évolution des valeurs de `g` et de `d`, la zone de recherche est devenue vide. Or le programme est organisé de telle façon que la condition suivante est un invariant de boucle :

Si `val` se trouve dans le tableau `tab` alors `val` est situé entre l'indice `g` et l'indice `d`

c) Le programme se termine toujours

• Bien que l'on utilise une boucle `while` dans le programme, il n'y a pas de risque de « boucle infinie ». En effet, à chaque étape et dans tous les cas, la différence $d - g$ *diminue strictement*. En effet,

* si on effectue l'affectation $d = m - 1$ alors, puisque l'on a $m \leq d$ avant l'affectation, la nouvelle valeur de d est *strictement inférieure* à son ancienne valeur donc $d - g$ diminue strictement ;

* si on effectue l'affectation $g = m + 1$ alors, puisque l'on a $g \leq m$ avant l'affectation, la nouvelle valeur de g est *strictement supérieure* à son ancienne valeur donc $d - g$ diminue strictement.

Il est donc certain qu'après un nombre fini d'étapes la valeur $d - g$ deviendra strictement négatif et on sortira de la boucle `while`. On dit que la quantité $d - g$ est un **variant de boucle**.

II.2 Complexité de l'algorithme

• Notons n la longueur du tableau trié `tab` dans lequel nous recherchons une valeur `val` par la méthode de recherche dichotomique. On suppose $n \geq 1$.

• Déterminons le nombre d'instructions exécutées par le programme en fonction de n *dans le pire des cas*, c'est-à-dire dans le cas où la valeur `val` ne se trouve pas dans le tableau `tab`.

• A chaque étape de l'algorithme, c'est-à-dire à chaque passage de boucle, la longueur de la zone de recherche est au moins divisée par 2. Donc si k est le plus petit nombre entier positif tel que $n < 2^k$, c'est-à-dire l'unique entier positif k tel que

$$2^{k-1} \leq n < 2^k$$

alors l'algorithme passera k fois dans la boucle. Le nombre d'instructions exécutées par le programme sera donc (à très peu de choses près) proportionnel à k.

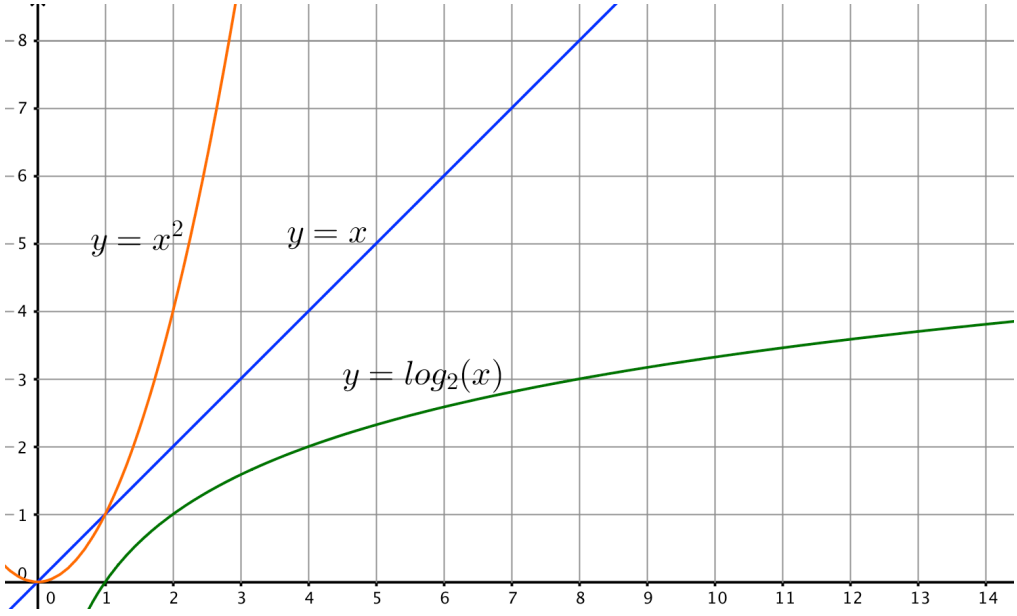
• Pour exprimer le nombre k, on peut utiliser $\log_2$, c'est-à-dire le logarithme de base 2. Rappelons l'équivalence suivante : $x = \log_2(n) \Leftrightarrow n = 2^x$. On a alors l'équivalence suivante :

$$2^{k-1} \leq 2^x < 2^k \Leftrightarrow k - 1 \leq x < k$$

D'où $\boldsymbol{k = E\left(\log_2(n)\right) + 1}$ où $E(y)$ est la partie entière de y, c'est-à-dire le plus grand des entiers inférieurs ou égaux à y.

• Pour résumer, le nombre d'instructions exécutées par le programme de recherche dichotomique est (quasiment) **proportionnel à $\boldsymbol{\log_2(n)}$.** On dit qu'il s'agit d'une **complexité logarithmique** ou en $O\left(\log_2(n)\right)$.

• *Une complexité logarithmique est une excellente complexité* car la fonction logarithme de base 2 croit beaucoup moins vite que la fonction $f(x) = x$. Comme on peut le voir sur la graphique ci-dessous :



• On peut aussi se rendre compte de l'excellence d'une complexité logarithmique avec le tableau suivant donnant la valeur du nombre k pour des valeurs de n correspondant à différentes puissances de 10 :

n	10	10^2	10^3	10^4	10^5	10^6	10^7
$k = E\left(\log_2(n)\right) + 1$	4	7	10	14	17	20	24

Multiplier la taille du tableau par 10 revient donc à *augmenter* le nombre de passages dans la boucle de seulement 3 ou 4 !