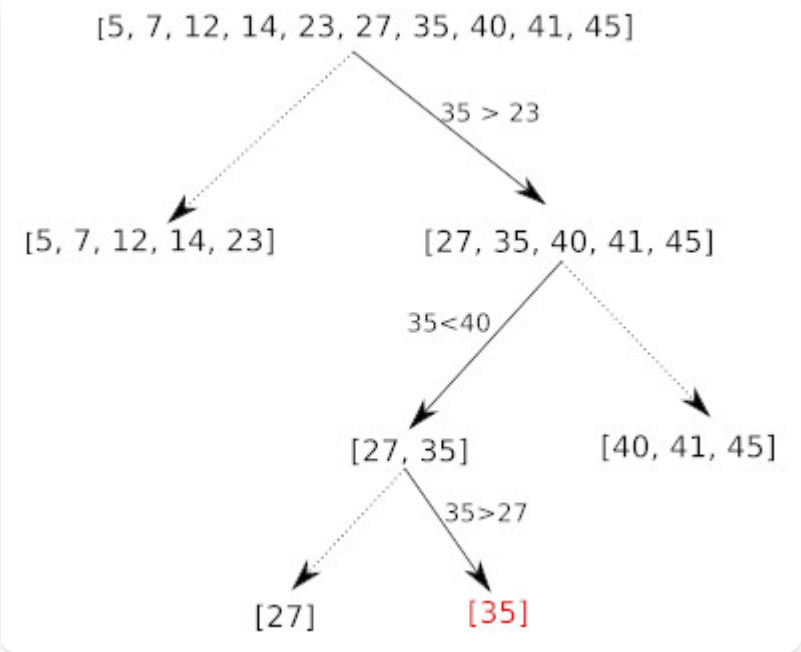
1) introduction

Nous avons déjà eu l'occasion d'étudier un algorithme de recherche d'un entier dans un tableau. Dans le pire des cas (l'entier recherché n'est pas dans le tableau), l'algorithme parcourt l'ensemble du tableau, nous avions donc une complexité O(n). Est-on obligé de parcourir l'ensemble du tableau pour vérifier qu'un entier x ne se trouve pas dans un tableau t ? A priori oui, sauf si le tableau t est trié !

2) Principe

Il est aussi possible de représenter le principe de l'algorithme de recherche dichotomique avec le schéma suivant (on recherche la valeur x=35 dans le tableau t = [5, 7, 12, 14, 23, 27, 35, 40, 41, 45] :



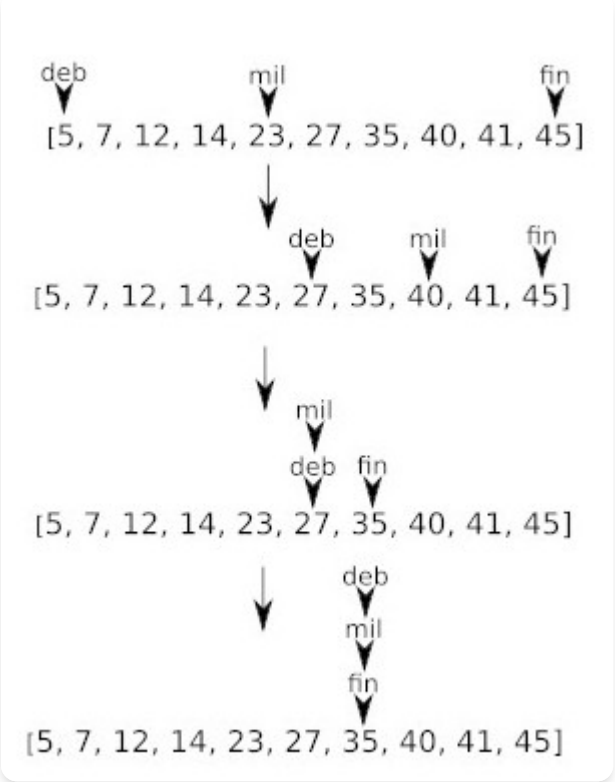
Dans le schéma ci-dessus, à chaque étape, on garde uniquement le tableau désigné par la flèche en trait plein, on abandonne le tableau désigné par la flèche en pointillé. Dans la première étape on part du tableau [5, 7, 12, 14, 23, 27, 35, 40, 41, 45], ce tableau est divisé en 2 tableaux : [5, 7, 12, 14, 23] et [27, 35, 40, 41, 45]. La valeur recherchée (35) ne peut pas être dans le premier tableau ([5, 7, 12, 14, 23]) puisque 35 > 23 et que les autres valeurs du premier tableau sont forcément plus petite que 23 (le tableau est trié). On garde donc uniquement le second tableau ([27, 35, 40, 41, 45]) et on recommence le processus (on divise ce tableau en deux…) jusqu'au moment où l'on "tombe" sur la valeur recherchée ou que l'on se retrouve avec un tableau contenant un seul élément : si l'élément unique du tableau n'est pas l'élément recherché, cela signifie que l'élément recherché n'est pas dans le tableau.

3) algorithme

Voici l'algorithme qui permet d'effectuer une recherche dichotomique :

```
VARIABLE
t : tableau d'entiers trié
mil : nombre entier
fin : nombre entier
deb : nombre entier
x : nombre entier // x : l'entier recherché
tr : booléen
DEBUT
tr ← FAUX
deb ← 1
fin ← longueur(t)
tant que tr == FAUX et que deb ⩽ fin :
 mil ← partie_entière((deb+fin)/2)
 si t[mil] == x :
  tr = VRAI
 sinon :
  si x > t[mil] :
   deb ← mil+1
  sinon :
   fin ← mil-1
  fin si
 fin si
fin tant que
renvoyer la valeur de tr
FIN
```

On peut résumer le principe de fonctionnement de l'algorithme de recherche dichotomique par le schéma suivant :



Les variables *deb*, *mil* et *fin* du schéma correspondent aux variables *deb*, *mil* et *fin* de l'algorithme.

4) complexité de l'algorithme

Pour étudier la complexité, nous allons nous intéresser à la boucle : au niveau de la boucle, combien doit-on effectuer d'itérations pour un tableau de taille n dans le cas le plus défavorable (l'entier x n'est pas dans le tableau t) ?

Sachant qu'à chaque itération de la boucle on divise le tableau en 2, cela revient donc à se demander combien de fois faut-il diviser le tableau en 2 pour obtenir, à la fin, un tableau comportant un seul entier ? Autrement dit, combien de fois faut-il diviser n par 2 pour obtenir 1 ?

Mathématiquement cela se traduit par l'équation $\frac{n}{2^a} = 1$ avec a le nombre de fois qu'il faut diviser n par 2 pour obtenir 1. Il faut donc trouver a !

A ce stade il est nécessaire d'introduire une nouvelle notion mathématique : le "logarithme base 2" noté log_2. Par définition $log_2(2^x) = x$

Nous avons donc :

$\frac{n}{2^a} = 1$ => $n = 2^a$ => $log_2(n) = log_2(2^a) = a$, nous avons donc $a = log_2(n)$

Nous pouvons donc dire que la complexité en temps dans le pire des cas de l'algorithme de recherche dichotomique est $O(log_2(n))$

Afin de pouvoir comparer l'efficacité des différents algorithmes, voici une représentation graphique des fonctions $y = x$ (en rouge), $y = x^2$ (en bleu) et $y = log_2(x)$ (en vert)

Comme vous pouvez le constater l'algorithme de recherche dichotomique est plus efficace que l'algorithme de recherche qui consiste à parcourir l'ensemble du tableau, car $x > log_2(x)$ quelque soit x.

Cependant, il ne faut pas perdre de vu que dans le cas de la recherche dichotomique, il est nécessaire d'avoir un tableau trié, si au départ le tableau n'est pas trié, il faut rajouter la durée du tri.

5) terminaison de l'algorithme

Pour terminer, nous allons démontrer que cet algorithme se termine dans tous les cas (on ne peut pas "tomber dans une boucle infinie") :

Nous avons les variables *fin* et *deb*. Définissons fin_i et deb_i avec i qui correspond à la ième itération de la boucle : deb_0 correspond à la valeur de la variable *deb* avant la première itération de la boucle (nous avons donc $deb_0 = 0$). Même chose pour fin_i (nous avons donc $fin_0 = n$). À la fin de la première itération de la boucle, nous aurons fin_1 et deb_1...

On définit aussi $m_i = (deb_i + fin_i) / 2$.

Partons du principe que nous sommes à la k ième itération (i=k), nous avons plusieurs cas à considérer :

- si $deb_k > fin_k$ ou si $t[m_k] = x$, l'algorithme se termine, car on n'entre pas dans la boucle.
- si $deb_k \leqslant fin_k$ et si $x < t[m_k]$, on entre dans la boucle : $deb_{k+1} = deb_k$ et $fin_{k+1} = m_k - 1$. On a alors $fin_{k+1} - deb_{k+1} < fin_k - deb_k$
- si $deb_k \leqslant fin_k$ et si $x > t[m_k]$, on entre dans la boucle : $deb_{k+1} = m_k + 1$ et $fin_{k+1} = fin_k$. On a alors $fin_{k+1} - deb_{k+1} < fin_k - deb_k$

Quel que soit le cas, nous avons $fin_{k+1} - deb_{k+1} < fin_k - deb_k$, nous pouvons donc dire que $fin_i - deb_i$ est strictement décroissante.

Il existe donc un entier p tel que :

- soit $deb_p > fin_p$, dans ce cas l'algorithme va se terminer, car on "n'entre pas" dans la boucle et l'algorithme renvoie FAUX.
- soit $x = t[m_p]$ avec $m_p = (deb_p + fin_p) / 2$, dans ce cas l'algorithme va se terminer, car on n'entre pas dans la boucle et l'algorithme renvoie VRAI.

Nous venons démontrer que l'algorithme se termine à un moment ou à un autre. Pour effectuer cette démonstration nous avons utilisé le fait que $fin_i - deb_i$ est strictement décroissante. $fin_i - deb_i$ est appelé un variant de boucle.

Un variant de boucle est une grandeur qui, comme son nom l'indique, varie à chaque itération, cette variation fait qu'à un moment ou à un autre, l'algorithme finira par s'arrêter.

activité 30.1

Faites un schéma permettant d'expliquer le principe de la recherche dichotomique en vous basant sur l'exemple suivant : t = [5, 7, 12, 14, 23, 27, 35, 40, 41, 45] et x = 40 (valeur recherchée)

activité 30.2

Faites un schéma permettant d'expliquer le principe de la recherche dichotomique en vous basant sur l'exemple suivant : t = [5, 7, 12, 14, 23, 27, 35, 40, 41, 45] et x = 9 (valeur recherchée)

activité 30.3

Soit l'algorithme de recherche dichotomique :

```
VARIABLE
t : tableau d'entiers trié
mil : nombre entier
fin : nombre entier
deb : nombre entier
x : nombre entier // x : l'entier recherché
tr : booléen
DEBUT
tr ← FAUX
deb ← 1
fin ← longueur(t)
tant que tr == FAUX et que deb ⩽ fin :
 mil ← partie_entière((deb+fin)/2)
 si t[mil] == x :
  tr = VRAI
 sinon :
  si x > t[mil] :
   deb ← mil+1
  sinon :
   fin ← mil-1
  fin si
 fin si
fin tant que
renvoyer la valeur de tr
FIN
```

1. Appliquez cet algorithme au tableau t = [5, 7, 12, 14, 23, 27, 35, 40, 41, 45] et x = 40. Vérifiez que l'algorithme renvoie bien VRAI

2. Appliquez cet algorithme au tableau t = [5, 7, 12, 14, 23, 27, 35, 40, 41, 45] et x = 9. Vérifiez que l'algorithme renvoie bien FAUX

activité 30.4

Implémentez l'algorithme de recherche dichotomique en Python.

Ce qu'il faut savoir

- l'algorithme de recherche dichotomique permet de rechercher un élément dans un tableau **trié**, en divisant ce tableau en 2 à chaque étape de la recherche
- la complexité en temps dans le pire des cas de cet algorithme est en $O(\log_2(n))$ (donc meilleur que l'algorithme qui cherche l'élément en parcourant entièrement le tableau qui est en $O(n)$).

Ce qu'il faut savoir faire

Vous devez être capable d'appliquer cet algorithme sur un exemple donné.