Algorithmie

Recherche par dichotomie dans une tableau *trié*

I Exposé du problème

En informatique, il est très souvent nécessaire de savoir si un élément appartient ou non à une liste.

Dans le cas où un tableau est *trié*, la recherche par dichotomique est beaucoup plus performante que la recherche séquentielle. Trier un tableau est couteux en temps, mais si l'on a souvent besoin de recherche dans un tableau, il devient intéressant de le trier afin d'utiliser cet algorithme.

Dichotomie : Ce mot vient du grec ancien *dikhotomia* qui signifie « couper en deux »

II Principe de la recherche dichotomique

Le principe est de comparer l'élément recherché au milieu de la liste.

- Si l'élément recherché est l'élément central, on a une solution. La recherche s'arrête.
- Si l'élément recherché est plus grand que l'élément central. Il est impossible qu'il se trouve avant l'élément central, on recherche dans la moitié "droite" de ce tableau.
- Sinon c'est que l'élément recherché n'est pas après l'élément central. On recherche dans la moitié "gauche" de ce tableau.

La recherche dichotomique est un exemple d'algorithme du type **diviser pour régner**.

La méthode **diviser pour régner** consiste, pour résoudre un problème à:

- Diviser: partager le problème en sous-problème
- Régner: résoudre chacun des sous-problème
- Combiner: fusionner les solutions pour obtenir la solution.

D'autres algorithmes du type diviser pour régner sont le tri fusion, le tri rapide, la transformée de Fourrier rapide, la rotation d'une image carrée d'un quart de tour.

Exemple 1

- Recherche de l'élément 12 dans le tableau trié [2,5,9,12,13,24,34,35,46]. Ce tableau est de longueur 9, ses indices vont de 0 à 8.

Étape 1

indice début 0, indice fin 8, indice milieu `(0+8)//2) = 4` , largeur $= 9 \leq 2^4 - 1$

2	5	9	12	**13**	24	34	35	46

$$13 > 12$$

12 peut se trouver dans la première moitié du tableau. On cherche entre les indices 0 et 3. **Étape 2**

indice début 0, indice fin 3, indice milieu `(0+3)//2 = 1` , largeur $= 4 \leq 2^3 - 1$

2	**5**	9	12	13	24	34	35	46

$$5 < 12$$

12 peut se trouver de la seconde moitié du tableau restant. On cherche entre les indices 2 et 3.

Étape 3

indice début 2, indice fin 3, indice milieu `(2+3)//2 = 2` , largeur $= 2 \leq 2^2 - 1$

2	5	**9**	12	13	24	34	35	46

$$9 < 12$$

9 est plus petit que 12. On cherche entre les indices 12 et 12.

Étape 4

indice début 3, indice fin 3, indice milieu `(3+3)//2` , largeur $= 1 \leq 2^1 - 1$

2	5	9	**12**	13	24	34	35	46

12 == 12. L'élément 12 a été trouvé.

Exemple 2

- Recherche de l'élément 30 dans la liste triée [2,5,9,12,13,24,34,35,46]

Étape 1

indice début 0, indice fin 8, indice milieu `(0+8)//2 = 4` , largeur $= 9 \leq 2^4 - 1$

2	5	9	12	**13**	24	34	35	46

$$13 < 30$$

On cherche donc dans la seconde moitié.

Étape 2

indice début 5 indice fin 8, indice milieu `(5+8)//2 =6` , largeur $= 4 \leq 2^3 - 1$

2	5	9	12	13	24	**34**	35	46

$$34 > 30$$

On cherche dans la première moitié restante.

Étape 3

indice début 5, indice fin 5, indice milieu `(5+5)//2 = 5` , largeur $= 1 \leq 2^2 - 1$

2	5	9	12	13	**24**	34	35	46

24 > 30 Non trouvé

Il suffit de 3 étapes pour conclure qu'un élément n'est pas dans une liste de taille 9. Ce qui est plus performant qu'avec une recherche séquentielle, qui aurait fallu 9 étapes.

III Exemple d'implémentation en Python

Recherche dichotomique dans une liste triée:

```python
def rechercheDicho(tab: list,element: int) -> bool:
    """
    Prend en argument un tableau ordonnée de nombres et un nombre.
    Retourne True si ce nombre est dans la liste et False sinon
    """
    indice_debut = 0
    indice_fin = len(tab) - 1
    while  indice_debut <=  indice_fin:
        indice_central = (indice_debut + indice_fin)//2
        valeur_centrale = tab[indice_central]
        if valeur_centrale == element:
            return True
        elif valeur_centrale < element:
            indice_debut = indice_central + 1
        else:
            indice_fin = indice_central - 1
    return False
```

IV Terminaison, correction et complexité

A Terminaison En algorithmie, démontrer la *terminaison* d'un algorithme sert à montrer que quelque soit les données il va s'arrêter et fournir une réponse. Autrement dit, qu'il n'y ai pas de boucles infinies. Ici, il faut s'assurer qu'il sort de la boucle "tant que".

Pour se faire, nous allons utiliser un **variant de boucle**.

Définition

Un *variant de boucle* est une quantité *entière* qui:

- doit être *positive ou nulle* dans la boucle;
- doit *décroître strictement* à chaque itération.

Si l'on trouve une telle quantité, il est clair que cette valeur entière positive strictement décroissante va devenir négative au bout d'un certain nombre d'itérations. Ce qui assure que l'on va sortir de la boucle à un moment donné et que donc le programme se termine.

Montrons que pour la dichotomie le variant de boucle est:

`largeur = indice_fin - indice_debut`

- `largeur` est un entier car c'est la différence entre deux entiers.
- `indice_fin - indice_debut >= 0` tant que l'on est dans la boucle car `indice_debut <= indice_fin`.
- À chaque étape:
  - Si `valeur_centrale == element`, on sort de la boucle à l'aide d'un `return True`
  - Sinon `indice_debut <= indice_central <= indice_fin`
    * Soit `indice_debut = indice_central + 1` donc `indice_debut`augmente strictement et donc `indice_fin-indice_debut` décroit strictement.
    * Soit `indice_fin = incide_central-1` donc `indice_fin` décroit strictement et donc `indice_fin-indice_debut` décroit strictement.

On a montré que si l'on ne sort pas de la boucle par `return True`, l'entier positif `indice_fin-indice_debut` décroit strictement.

C'est un variant de boucle. La boucle se termine nécessairement.

B Correction La définition d'un algorithme comporte une partie spécification.

Un algorithme prend des arguments en entrée et une sortie. Ce que calcule l'algorithme est le lien entre les deux.

Etablir la correction (partielle) d'un algorithme c'est démontrer que si l'algorithme se termine alors il fait les calculs attendus par les spécification de l'algorithme.

La correction est dite totale si on a montré la terminaison et la correction partielle.

Autrement dit, il retourne bien ce qui est attendu quelque soit les données en entrée.

Pour montrer la correction d'un algorithme, il faut utiliser un **invariant de boucle**.

On appelle invariant de boucle une propriété qui est vraie avant et après chaque itération.

On se place d'abord dans le cas ou l'élément recherché est présent dans le tableau.

On veut montrer que:

"Tout indice tel que `tab[indice] == element` vérifie `indice_debut <= indice <= indice_fin`"

est un invariant de boucle.

- Avant la boucle: `indice_debut = 0` et `indice_fin = len(tab) - 1`, comme on suppose que `element` est dans le tableau la propriété est vérifée.
- On suppose que l'on est arrivé à une étape ou l'on a la propriété: "Tout indice tel que `tab[indice] == element` vérifie `indice_debut <= indice <= indice_fin`". On veut montrer qu'a l"étape suivante cette propriété est toujours vraie.
  - Si `tab[indice_central] == element` on sort de la boucle et dans ce cas les valeurs de `indice_debut` et `indice_fin` n'ont pas changées la propriété est vérifiée.
  - Sinon: - Soit `tab[indice_central] < element` dans ce cas on a `indice_debut = indice_central + 1` et la propriété est vérifiée. - Soit `tab[indice_central] > element` dans ce cas on a `indice_fin = indice_central -1` et la propriété est toujours vérifiée.

Dans tous les cas la propriété est vérifiée après le passage de boucle. Il s'agit bien d'un invariant de boucle.

- Si `element` est dans le tableau `tab` l'invariant de boucle nous assure qu'à chaque passage dans la boucle l'indice de l'élément recherché est compris entre `indice_debut` et `indice_fin` . On sort donc nécessairement par la ligne `if valeur_centrale == element:`et la fonction retourne `True`

- Si `element` n'est pas dans le tableau `tab` alors la condition `valeur_centrale == element` n'est jamais vérifiée donc on sort de la boucle lorsque `indice_debut > indice_fin` et la fonction renvoie bien `False` .

On a montré la correction (partielle) et la terminaison, on a donc démontré la correction totale de l'algorithme.

C Complexité - Coût temporel Le but est de déterminer un ordre de grandeur du nombre d'opérations à effectuer pour un tableau de taille n.

Le corps de la boucle:

```
indice_central = (indice_debut+ indice_fin)//2
        valeur_centrale = tab[indice_central]
        if valeur_centrale == element:
            return True
        elif valeur_centrale < element:
            indice_debut = indice_central + 1
        else:
            indice_fin = indice_central - 1
```

Prend un temps à peu près constant entre chaque passage.

La différence entre deux exécutions de l'algorithme sur un tableau sera principalement due au nombre de passage dans la boucle.

Nous avons vu que pour une même taille de tableau, le nombre de passage dans la boucle peut varier.

On va s'intéresser au nombre de passage "au pire" dans le corps de la boucle. C'est-à-dire le nombre de passages dans la boucle lorsque le tableau ne contient pas l'élément recherché.

On note $p(n)$ le nombre de passages dans la boucle dans le "pire des cas" pour un tableau de taille n.

Pour un tableau de taille $1\ 001$, il y $p(1001)$ passage dans la boucle.

Après le premier passage dans la boucle l'élément central n'est pas le bon, il est cherché dans la moitié gauche ou dans la moitié droite du tableau. Comme $1001 = 2t \times 500 + 1$ chacun des ces tableaux est de taille 500 donc nécessiterons $p(500)$ tour de boucle.

On a donc $p(1001) = p(500) + 1$

De façon plus générale, on retient que $\boxed{p(2k) = p(k) + 1}$.

Autrement dit, en multipliant par deux la taille du tableau on ajoute dans le "pire des cas" un passage supplémentaire dans la boucle.

De façon approximative, on a:

$p(4000) = p(2000) + 1$

$p(2000) = p(1000) + 1$

$p(500) = p(1000) + 1$.

On en déduit que : $p(4000) = p(500) + 3$ soit $p(500 \times 2^3) = p(500) + 3$

Plus généralement, on retient que $\boxed{p(n \times 2^k) = p(n) + k}$.

Dans une recherche dichotomique sur un tableau trié de taille n, dans le pire des cas le nombre d'itérations à effectuer est le plus petit entier k tel que $n \leq 2^k - 1$.

On peut aussi dire que le nombre d'itérations a effectuer est égale au nombre de bits nécessaire pour écrire n la taille du tableau en binaire à une unité près.

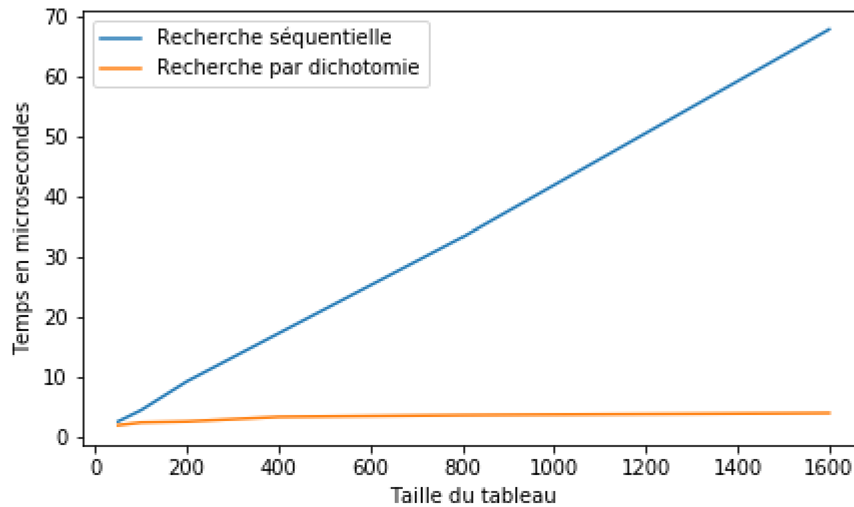
La fonction mathématique qui permet de connaître le nombre de bits nécessaire pour écrire un nombre en binaire est la fonction logarithme de base 2 notée $\log_2$

On dit que la recherche dichotomique à une complexité temporelle **logarithmique**. On note $\mathrm{O}(\log_2(n))$ ce qui se lit "grand O de logarithme en base 2 de n".

Ce tableau compare le nombre d'itérations à effectuer avec la recherche dichotomique et la recherche séquentielle dans le pire des cas.

Taille du tableau	10	100	1 000	10^6	10^9	10^N
Recherche séquentielle	10	100	1 000	10^6	10^9	10^N
Recherche dichotomique	4	7	10	20	30	$\log_2(N)$

Comparaison des temps entre recherche séquentielle et par dichotomie pour des tableaux de taille 50 à 1600:



On constate que la recherche par dichotomie, qui est de complexité temporelle logarithmique, est bien plus rapide que la recherche séquentielle, qui est de complexité temporelle *linéaire*.

Vous devez retenir aussi que lorsque la complexité temporelle est logarithmique, pour doubler le temps d'exécution, il faut mettre au carré la taille du tableau.

Pour aller plus loin:

- https://cache.media.eduscol.education.fr/file/NSI/76/3/RA_Lycee_G_NSI_algo-dichoto_1170763.pdf
- https://professeurb.github.io/articles/dichoto/
- https://iamjmm.ovh/NSI/dichotomie/site/index.html