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.

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 A chaque tour de boucle on divise par 2 la longueur de la partie du tableau à examiner. Cette longueur, qui est une valeur entière, est un variant de la boucle "tant que" (elle est positive et strictement décroissante). L'algorithme se termine bien.

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

On note $p(n)$ le nombre de passages dans la boucle dans le "pire des cas", lorsque l'élément recherché n'est pas dans le tableau, pour un tableau de taille n.

Pour un tableau de taille $1\ 001$, il y $p(1001)$ passages 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 $1001 = 2 \times 500 + 1$ chacun des ces tableaux est de taille 500 donc nécessiterons $p(500)$ tours 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.

On dit que la recherche dichotomique à une complexité temporelle **logarithmique**. On note $\mathrm{O}(\log_2(n))$.

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