Rappels : Recherche par Dichotomie

Chaddaï Fouché

2023

L'objectif est de déterminer à quel indice d'un tableau on trouve une certaine valeur. Il est possible d'étendre ceci à la recherche d'éléments ayant une certaine caractéristique (tant que cette caractéristique peut être ordonnée).

Un algorithme pratiquement identique permet également de vérifier simplement si un élément est présent ou non dans un tableau.

I] Recherche linéaire

La solution naïve et la seule possible sans pré-traitement du tableau est la **recherche linéaire** (on examine les éléments les uns après les autres sur une ligne) aussi appelée *recherche séquentielle* ou *recherche par balayage* :

```python
def recherche_linéaire(T : list, x) -> int:
    """Renvoie l'indice de la première occurrence de x dans le tableau T
    ou None si x n'est pas dans T"""
    for i in range(len(T)):
        if x == T[i]:
            return i
    return None # automatique si la fonction s'achève sans return
```

Notez qu'il est assez standard de renvoyer (-1) lorsqu'on ne trouve pas l'élément dans d'autres langages mais en Python (-1) est un indice valide pour tout tableau non-vide, on évite donc généralement cette solution : nous renverrons `None` dans nos fonction, `list.index` la méthode Python qui effectue une recherche linéaire soulève l'exception `ValueError` si elle ne trouve pas l'élément dans le tableau.

Malheureusement cet algorithme est à la fois inévitable si nous ne savons rien de précis sur un tableau et assez lent : pour un tableau de longueur n, nous avons besoin d'au pire n tours de boucle pour réaliser que l'élément n'est pas dans le tableau, ainsi la complexité temporelle de la recherche linéaire est en $O(n)$, c'est un **algorithme de complexité linéaire** (prend un temps proportionnel à la taille des données).

⚠**Attention** : Le test booléen d'appartenance d'un élément à un tableau `x in T` (à ne pas confondre avec le parcours dans une boucle `for`) utilise la recherche linéaire (car Python ne sait rien sur T) et est donc une instruction très lente sur les grands tableaux, à éviter dans les portions sensibles d'un programme (celles qui sont souvent répétées).

II] Recherche dichotomique

Si nous avons plus d'informations sur le tableau ou nous pouvons pré-traiter les données avant d'effectuer plusieurs recherches, nous pouvons faire beaucoup mieux avec la recherche dichotomique.

La dichotomie (racines grecques : díkha « en deux » et tomós « section, coupure ») est une méthode résultant du paradigme « diviser pour régner » qui cherche à diviser l'espace de recherche (les indices *possibles* pour la valeur) en deux à chaque étape. Pour réaliser ceci, on doit procéder à partir d'un **tableau trié** et comparer l'élément recherché à l'élément au milieu de l'espace de recherche pour décider dans laquelle de ses moitiés il *peut* se trouver.

Voyons comment une recherche dichotomique de 4 dans le tableau `T = [1,3,4,5,7,8,10,13,14]` va se dérouler. On peut voir ceci comme une enquête où l'on cherche quel indice contient 4. On va procéder **par élimination**.

L'espace de recherche (les indices qui *pourraient* contenir 4) sera coloré en rouge et l'élément au milieu surligné en bleu. En grisé sont les valeurs que l'algorithme n'a pas encore consulté.

Nous commençons avec tous les indices suspects, du premier `0` au dernier `len(T) - 1 == 8` et l'élément au milieu est 7 d'indice `(0+8) // 2 == 4` (moyenne des indices de début et de fin de l'espace de recherche) :

indices	0	1	2	3	4	5	6	7	8
valeurs	1	3	4	5	7	8	10	13	14

On compare 4 à 7, 4 étant plus petit que 7, il ne peut se trouver à un indice supérieur ou égal à celui de 7 (le tableau étant trié) donc on peut éliminer tous ces indices et garder uniquement les indices de 0 à 3 dans l'espace de recherche. L'élément au milieu est donc d'indice `(0+3)//2 == 1`.

indices	~~0~~	1	2	3	~~4~~	~~5~~	~~6~~	~~7~~	~~8~~
valeurs	1	3	4	5	7	8	10	13	14

On compare 4 à 3, 4 est plus grand, il ne peut donc se trouver à un indice inférieur ou égal à 1, l'indice de 3, donc on garde uniquement les indices de 2 à 3. L'élément au milieu est d'indice `(2+3)//2 == 2`.

indices	~~0~~	~~1~~	2	3	~~4~~	~~5~~	~~6~~	~~7~~	~~8~~
valeurs	1	3	4	5	7	8	10	13	14

On compare 4 et 4… On a trouvé l'élément cherché, on renvoie donc son indice : 2.

On remarque que l'immense majorité des valeurs n'ont même pas été consultées par l'algorithme, la recherche dichotomique permet de trouver ou de conclure à l'absence d'une valeur dans un tableau en n'en consultant que quelques valeurs !

Si l'on recherche un élément qui n'est pas dans le tableau, l'algorithme procède de même jusqu'à ce que l'espace de recherche soit vide (ce qui arrive forcément puisqu'on élimine au moins un indice par étape, celui de l'élément milieu).

En résumé, pour cet algorithme :

- On doit garder trace des indices de début et de fin (inclus) de l'espace de recherche, souvent nommés `gauche` et `droite`.
- À chaque étape on calcule la moyenne (tronquée à l'entier) de ces indices pour obtenir l'indice de l'élément au milieu de l'espace de recherche.
- On compare la valeur recherchée à l'élément au milieu et on a trois cas :
  - ils sont égaux : on a trouvé notre valeur, on renvoie l'indice milieu ;
  - l'élément cherché est inférieur à l'élément au milieu : tous les indices à partir de l'indice milieu sont éliminés, la fin de l'espace de recherche (`droite`) devient l'indice précédent l'indice milieu ;
  - l'élément cherché est supérieur à l'élément au milieu : tous les indices jusqu'à l'indice milieu sont éliminés, le début de l'espace de recherche (`gauche`) devient l'indice suivant l'indice milieu.

- On s'arrête lorsque la fin de l'espace de recherche devient inférieure au début (il n'y a plus d'indices pouvant contenir la valeur cherchée) sinon on recommence notre recherche dans notre nouvel espace de recherche.

Voici donc deux implémentations, une itérative et une récursive :

```
def recherche_dichotomique_iter(T : list, x) -> int:
    """Renvoie l'indice d'une occurrence de x dans T
    ou None si x n'est pas dans T"""
    gauche, droite = 0, len(T)-1
    while gauche <= droite:
        milieu = (gauche + droite) // 2
        if x == T[milieu]:
            return milieu
        elif x < T[milieu]:
            droite = milieu - 1
        else: # T[milieu] < x
            gauche = milieu + 1
    return None

def recherche_dichotomique_rec(T : list, x) -> int:
    """Renvoie l'indice d'une occurrence de x dans T
    ou None si x n'est pas dans T"""
    # on définit la recherche récursive sur les indices gauche..droite
    def récursif(gauche, droite) -> int:
        if gauche > droite:
            return None
        milieu = (gauche + droite) // 2
        if x == T[milieu]:
            return milieu
        elif x < T[milieu]:
            return récursif(gauche, milieu - 1)
        else: # T[milieu] < x
            return récursif(milieu + 1, droite)

    # et on l'utilise sur tout le tableau
    return récursif(0, len(T)-1)
```

Attention, ces fonctions ne renvoient pas forcément l'indice de la **première** occurrence de `x` dans `T`, il est assez facile d'y remédier néanmoins en décrémentant l'indice tant que l'élément précédent vaut `x` :

```
def recherche(T : list, x) -> int:
    i = recherche_dichotomique(T,x)
    if i is None:
        return None
    while i > 0 and T[i-1] == x:
        i -= 1
    return i
```

Théoriquement cela pourrait être lent mais en pratique, cela ne pose pas problème sauf si vos tableaux contiennent de très nombreux exemplaires d'une même valeur.

La complexité de la recherche dichotomique peut être calculée en constatant qu'à chaque étape, le

nombre d'indices suspects est au moins divisé par 2, ainsi si on part d'un tableau de taille $n = 2^p$, il suffira de p étapes pour avoir au plus un indice suspect (car chaque division par 2 réduit la puissance de 1 : $\frac{2^p}{2} = \frac{2^p}{2^1} = 2^{p-1} \ldots$) donc d'au plus $p + 1$ étapes pour trouver un indice ou constater l'absence de l'élément dans le tableau. Cela signifie que le nombre d'étapes est proportionnel à p qui est le logarithme en base 2 de n (la puissance à laquelle il faut élever 2 pour obtenir n) aussi noté $\log_2(n)$. Tous les logarithmes (binaire, décimal, naturel) étant proportionnels, on omet souvent la base et on dit que la recherche dichotomique est en $O(\log n)$, c'est un **algorithme de complexité logarithmique** (prend un temps proportionnel au logarithme du nombre de valeurs).

Pour constater quelle différence cela fait, voici quelques valeurs de n, $\log_2(n)$ et les temps approximativement pris par les recherches linéaires et dichotomiques d'éléments absents de tableaux de ces tailles sur un ordinateur raisonnablement puissant (en 2015…) :

n	10	100	1000	1 000 000	10 000 000
$\lceil \log_2(n) \rceil$	3	7	10	20	24
linéaire	0,5 µs	4 µs	43 µs	45 ms	0,5 s
dichotomique	0,5 µs	1 µs	1,5 µs	3,5 µs	4 µs

Comme on le voit, ces deux algorithmes sont relativement similaires sur les petits tableaux mais dès qu'on augmente la taille, la recherche dichotomique devient immensément plus rapide : pour 10 000 000 d'éléments, la recherche dichotomique met de l'ordre de 100 000 fois moins de temps que la recherche linéaire (qui atteint des temps perceptibles par un humain, pour chaque recherche).

Il peut donc être très intéressant de trier un tableau avant d'y effectuer des recherches. Si le tableau est déjà trié ou doit l'être pour une autre raison, il n'y a aucune question à se poser, mais sinon, il faut réfléchir au nombre de recherche qu'on va effectuer sur ce tableau : en effet le tri (en $O(n \log(n))$ pour les plus rapides) est plus lent qu'une simple recherche linéaire (en $O(n)$), il n'y aura donc gain de temps que si le nombre de recherche à effectuer par la suite dépasse sensiblement $\log_2(n)$.