Dans le chapitre précédent on a vu que l'algorithme de recherche d'un élément dans un tableau a en général une complexité linéaire. On ne peut a priori pas faire mieux puisque la recherche nécessite toujours dans le pire cas de vérifier chaque élément du tableau.

Ce raisonnement ne vaut cependant que si on ne dispose d'aucune information sur le tableau. Dans ce chapitre, nous allons voir un algorithme de recherche plus efficace à condition de l'exécuter sur un tableau trié : la recherche dichotomique.

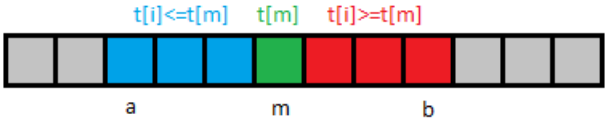
1 Implémentation

Considérons `t` un tableau trié de taille `n` dont on veut savoir s'il contient ou non un certain élément `v`.

Considérons également un élément se trouvant au milieu du tableau par exemple pour `m = n//2`. Puisque le tableau est trié, on sait que :

— tous les éléments à gauche de `t[m]`, c'est à dire d'indice compris entre `0` et `m-1` sont plus petits que `t[m]`.

— tous les éléments à droite de `t[m]`, c'est à dire d'indice compris entre `m+1` et `n-1` sont plus grands que `t[m]`.



Pour notre recherche, il a alors trois possibilités :

1. Si `v == t[m]` alors `v` est présent dans `t` : on peut arrêter la recherche.
2. Si `v<t[m]` alors `v` ne peut être qu'à gauche de `y` : on va recommencer la recherche sur la partie gauche de `t`.
3. Si `v>t[m]` alors `v` ne peut être qu'à droite de `y` : on va recommencer la recherche sur la partie droite de `t`.

Notons que si on recommence la recherche sur la partie gauche ou droite, la partie du tableau dans laquelle on recherche est alors toujours strictement plus petite, de sorte qu'au bout d'un moment, il ne restera plus aucun élément à rechercher. Dans ce cas, si `v` n'a pas été trouvé, on pourra en déduire que `v` n'est pas dans `t`.

```
def recherche_dichotomique(t, v):
    """renvoie une position de v dans le tableau t,
    suppose trie, et None si v ne s y trouve pas"""
    g = 0
    d = len(t) - 1
    while g <= d:
        # variant decroissant : d-g, arret si d-g<0
        # invariant : 0 <= g et d < len(t)
        # invariant : v ne peut se trouver que dans t[g..d]
        m = (g + d) // 2
        if t[m] < v:
            g = m + 1
        elif t[m] > v:
            d = m - 1
        else:
            return m
    # la valeur ne se trouve pas dans le tableau
    return None
```

2 Terminaison et correction

Terminaison Démontrons dans un premier temps que la boucle `while` termine toujours. Nous allons montrer que `d-g` en est un variant strictement décroissant.

Petit résultat utile : lors de l'exécution de la boucle, définit `m = (g+d)//2` la partie entière de la moyenne de `g` et `d`. Tant que la boucle n'est pas arrêtée on a donc toujours :

```
g <= m <= d
```

Étant donné la structure du code, il y a trois possibilité lors de la boucle :

— Ou bien `v` est trouvé et la boucle s'arrête.

— Ou bien `g` est remplacé par `m+1`. Dans ce cas, `d-g` devient

```
d-m-1 <= d-g-1 < d-g
```

Donc `d-g` décroît strictement.

— Ou bien `d` est remplacé par `m-1`. Dans ce cas, `d-g` devient

```
m-1-g <= d-g-1 < d-g
```

Donc `d-g` décroît strictement.

Nous avons montré que `d-g` est un variant strictement décroissant la boucle et de plus, la condition `g<=d` de la boucle est équivalent la la condition `d-g>=0`. Par la propriété des variants, cela assure que la boucle termine.

Correction Voyons dans un premier temps les sorties de fonction :

— Les `return m` n'ont lieu que si `t[m]==v`, auquel cas, `v` est bien présent dans le tableau et la fonction est correcte.

— Le `return None` n'a lieu qu'en sortie de la boucle, on doit donc s'assurer que, si la boucle termine de manière non-prématurée, `v` n'est pas présent dans `t`.

L'invariant de boucle sera le suivant :

P : « `v` ne peut se trouver qu'entre les indices `g` et `d`. »

Démontrons que cette propriété est héréditaire :

— Supposons qu'au début d'un tour de boucle, P soit vrai, autrement dit, `v` ne peut se trouver qu'entre les indices `g` et `d`.

— Exécutons la boucle :

1. Si `t[m]==v`, on sort de la boucle et on n'a besoin de rien démontrer.
2. Si `t[m]<v` et puisque `t` est trié, `v` ne peut se trouver à un indice strictement supérieur à `m` soit supérieur à `m+1` valeur par laquelle on remplace `g`. Au prochain tour, P sera donc toujours vraie.
3. Si `t[m]>v` et puisque `t` est trié, `v` ne peut se trouver à un indice strictement inférieur à `m` soit inférieur à `m-1` valeur par laquelle on remplace `d`. Au prochain tour, P sera donc toujours vraie.

— P est bien héréditaire.

Démontrons que la propriété est vraie avant la boucle : c'est la cas, `v` ne peut être présent qu'entre les indices `g=0` et `d=len(t)-1`.

P est un invariant, vraie avant la boucle et donc vraie à sa sortie.

Si on sort de la boucle de manière non-prématurée, c'est que `g>d` mais P étant vraie, `v` ne peut se trouver qu'entre les indice `g` et `d` ce qui est ici impossible : `v` n'est pas dans `t`.

3 Complexité

Une analyse du déroulement de l'algorithme montre qu'à chaque étape de la boucle, la zone dans laquelle peut encore se trouver la valeur cherchée est coupée en deux. La boucle s'arrêtant lorsque la taille de cette zone atteint 0, son nombre de tours est donc, au pire, le nombre de division par deux nécessaires pour passer de la taille n du tableau à zéro.

Ainsi, le nombre d'opérations effectuées par l'algorithme sera, dans le pire cas et à une constante près, proportionnel à ce nombre de divisions. Notons que ce nombre est aussi égal au nombre de multiplications par deux nécessaires afin de passer de 1 à un nombre supérieur à n. Il s'agît donc du plus petit entier k tel que $2^k > n$.

n	k
10	4
100	7
1000	10
10000	14

Un tel nombre k peut s'exprimer en fonction de n par :

$$k = \lfloor \log_2(n) \rfloor \leq \log_2(n)$$

Où la fonction $\log_2$ désigne le logarithme de base 2 (plus de détail en Term spé maths).

L'algorithme de recherche dichotomique est de **complexité logarithmique** $\mathcal{O}(\log(n))$. Il s'agît d'une bien meilleure complexité qu'une complexité linéaire (cf tableau n, k).