# RECHERCHE DICHOTOMIQUE

- ▷ Histoire de l'informatique
- ▷ Représentation des données
- ▷ Traitement des données
- ▷ Interactions entre l'homme et la machine sur le Web
- ▷ Architectures matérielles et systèmes d'exploitation
- ▷ Langages et programmation
- ▷ Algorithmique

## 1. Généralités

Des volumes importants de données sont susceptibles d'être traitées par les ordinateurs. Des algorithmes efficaces sont alors nécessaires pour réaliser ces opérations comme, par exemple, la sélection et la récupération des données. Les *algorithmes de recherche* entrent dans cette catégorie. Leur rôle est de déterminer si une donnée est présente et, le cas échéant, d'en indiquer sa position, pour effectuer des traitements annexes. La recherche d'une information dans un annuaire illustre cette idée. On cherche si telle personne est présente dans l'annuaire afin d'en déterminer l'adresse. Plus généralement, c'est l'un des mécanismes principaux des bases de données : à l'aide d'un identifiant, on souhaite retrouver les informations correspondantes.

Dans cette famille d'algorithmes, la *recherche dichotomique* permet de traiter efficacement des données représentées dans un tableau de façon ordonnée. Le livre [1] [1] consacre une partie importante du chapitre 9 à cet algorithme et à son étude.

## 2. Présentation de l'algorithme

### 2.1. Approche naïve

Une première façon de rechercher une valeur dans un tableau est d'effectuer une recherche naïve à l'aide d'un parcours de tableau, que l'on peut programmer ainsi :

```
def recherche_naive(tab, val):
    for i in range(len(tab)):
        if tab[i] == val:
            return i
    return -1
```

Ici, on renvoie un entier positif ou nul en cas de succès, qui correspond à une position de la valeur recherchée dans la tableau, et −1 en cas d'échec.
Comme tout algorithme ayant cette forme, la complexité est linéaire : le temps de recherche double lorsque la longueur de la liste double. Mais avec cette méthode, on n'exploite pas le caractère ordonné du tableau, ce qui fait savoir que telle valeur du tableau n'est pas la valeur recherchée n'apprend absolument rien sur les autres valeurs du tableau.

1. Il est disponible en ligne à l'adresse https ://www.lri.fr/~mcg/PDF/FroidevauxGaudelSoria.pdf.

### 2.2. Approche par dichotomie

L'idée centrale de cette approche repose sur l'idée de réduire de moitié l'espace de recherche à chaque étape : on regarde la valeur du milieu et si ce n'est pas celle recherchée, on sait qu'il faut continuer de chercher dans la première moitié ou dans la seconde. Plus précisément, en tenant compte du caractère trié du tableau, il est possible d'améliorer l'efficacité d'une telle recherche de façon conséquente en procédant ainsi :

1. on déterminer l'élément `m` au milieu du tableau;
2. si c'est la valeur recherchée, on s'arrête avec un succés;
3. sinon, deux cas sont possibles :
   (a) si `m` est plus grand que la valeur recherchée, comme la tableau est trié, cela signifie qu'il suffit de continuer à chercher dans la première moitié du tableau;
   (b) sinon, il suffit de chercher dans la moitié droite.
4. on répète cela jusque avoir trouvé la valeur recherché, ou bien avoir réduit l'intervalle de recherche à un intervalle vide, ce qui signifie que la valeur recherchée n'est pas présente.

À chaque étape, on coupe l'intervalle de recherche en deux, et on en choisit une moitié. On dit que l'on procède par *dichotomie*, du grec *dikha* (en deux) et *tomos* (couper). On peut trouver un exemple animé de l'exécution de cet algorithme à l'adresse

https ://professeurb.github.io/articles/dichoto/.

Une implémentation en Python est la suivante :

```python
def recherche_dichotomique(tab, val):
    gauche = 0
    droite = len(tab) - 1
    while gauche <= droite:
        milieu = (gauche + droite) // 2
        if tab[milieu] == val:
            # on a trouvé val dans le tableau,
            # à la position milieu
            return milieu
        elif tab[milieu] > val:
            # on cherche entre gauche et milieu - 1
            droite = milieu - 1
        else: # on a tab[milieu] < val
            # on cherche entre milieu + 1 et droite
            gauche = milieu + 1
    # on est sorti de la boucle sans trouver val
    return -1
```

Un exemple d'exécution est représenté figure 1.

# 3. Analyse de l'algorithme

Pour s'assurer que le programme ci-dessus fonctionne correctement, il faut se poser deux questions importantes :

1. Le programme renvoie-t-il bien un résultat ? Comportant une boucle non bornée, est-on sûr d'en sortir à un moment donné ?
2. La réponse renvoyée par le programme est-elle correcte ? Ici, il y a deux sortes de réponses : soit on a obtenu $-1$, auquel cas la valeur recherchée *ne doit pas* est présente dans le tableau; soit on a obtenue un entier positif ou nul qui correspond à la position de la valeur recherchée dans le tableau (ou, en cas de répétition, l'une des positions).

Nous allons traiter ces deux questions et en donner des réponses rigoureuses dans la suite, prouvant ainsi que le comportement de notre programme est bien celui espéré, ainsi qu'étudier sa complexité.

On recherche la valeur `10` dans le tableau `[3, 3, 5, 6, 8, 11, 13, 14, 14, 17, 19, 21, 23]`.

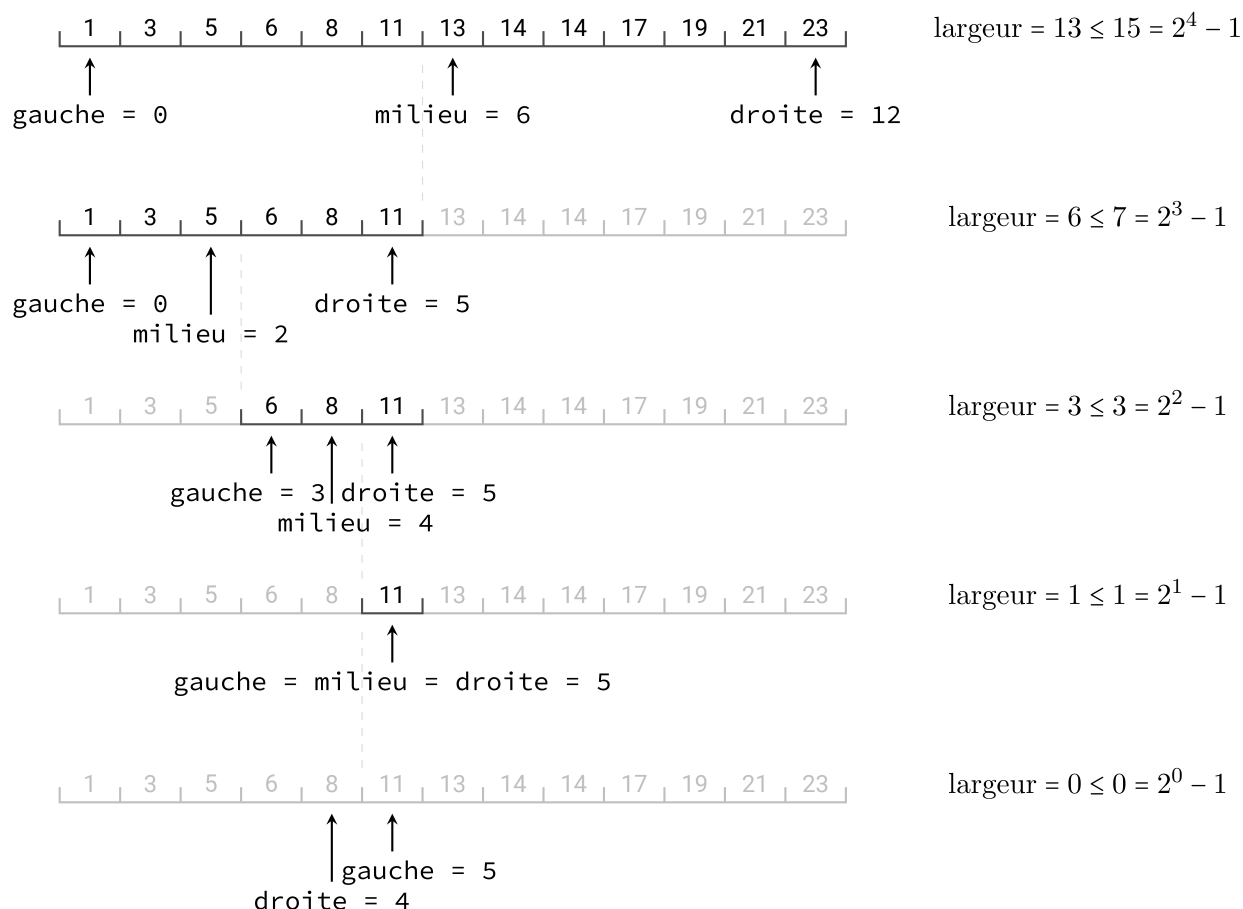


FIGURE 1 – Exemple de recherche dichotomique

## 3.1. Terminaison du programme

La fonction `recherche_dichotomique` contient une boucle non bornée, une boucle *while*, et pour être sûr de toujours obtenir un résultat, il faut s'assurer que le programme termine, que l'on ne reste pas bloqué infiniment dans la boucle.
Pour prouver que c'est bien le cas, nous allons utiliser un *variant de boucle*.

### Variant de boucle

Il s'agit d'une quantité entière qui :

- ▷ doit être positive ou nulle pour rester dans la boucle ;
- ▷ doit décroîtrement à chaque itération.

Si l'on arrive trouver une telle quantité, il est évident que l'on a nécessairement sortir de la boucle au boût d'un nombre fini d'itérations, puisque un entier positif ne peut décroître infiniment.

### Preuve de la terminaison

Pour le cas qui nous occupe, un variant est très facile à trouver : il s'agit de la largeur de la quantité `droite - gauche`. La condition de boucle étant `gauche <= droite`, cela correspond exactement à ce que notre variant soit positif ou nul.
Montrons maintenant que le variant décroit strictement lors de l'exécution du corps de la boucle. On commence par définir `milieu` ↪ `= (gauche + droite) // 2`. En particulier, on a alors

```
gauche <= milieu <= droite
```

Ensuite, trois cas sont possibles.

- si `tab[milieu] == val`, on sort directement de la boucle à l'aide d'un **return**. La terminaison est assurée.
- si `tab[milieu] > val`, on modifie la valeur de `gauche`. En appelant `gauche'` cette nouvelle valeur, on a :

```
droite - gauche' < droite - milieu <= droite - gauche
```

  Ainsi, le variant a strictement décru.
- sinon, on modifie `droite` et on a de même :

```
droite' - gauche < milieu - gauche <= droite - gauche
```

  De même, le variant a strictement décru.

Ayant réussi à exhiber un variant pour notre boucle, nous avons prouvé qu'elle termine bien.

Bien sûr, l'utilisation très basique de ce variant ne permet pas, telle qu'elle, de justifier la complexité de la recherche dichotomique. On a prouvé qu'il diminue de $1$ (au moins) par étape, et l'on peut donc seulement affirmer que le nombre d'itérations est majoré par la taille initiale du tableau. Cela correspond à la complexité linéaire de l'algorithme naïf. On peut améliorer cette borne en exploitant l'idée de la dichotomie.

## 3.2. Complexité

Supposons que l'on doive effectuer une recherche dans un tableau trié contenant 174 valeurs (un annuaire, par exemple). En procédant par dichotomie, on regarde la valeur au milieu et suivant la cas, on s'arrête ou l'on continue la recherche dans l'une des deux moitiés restantes. En négligeant la valeur supprimée, les moitiés contiennent chacune 174/2 = 87 valeurs. La fois suivante, si l'on n'a pas trouvé la valeur recherchée, on continue de sélectionner l'une des moitiés de ce qui reste, qui contiennent 43 ou 44 valeurs. Le processus se poursuit jusqu'à n'avoir au plus qu'une valeur :

$$174 \xrightarrow{\text{itération 1}} 87 \xrightarrow{\text{itération 2}} 44 \xrightarrow{\text{itération 3}} 22 \xrightarrow{\text{itération 4}} 11 \xrightarrow{\text{itération 5}} 6 \xrightarrow{\text{itération 6}} 3 \xrightarrow{\text{itération 7}} 2 \xrightarrow{\text{itération 8}} 1$$

Pour pouvoir majorer le nombre maximum d'itérations, si le tableau contient $l$ valeurs, et si on a un entier $n$ tel que $l \leq 2^n$, alors puisque qu'à chaque itération, on sélectionne une moitié de ce qui reste, au bout d'une itération, une moitié de tableau aura au plus $2^n/2 = 2^{n-1}$ éléments, un quart aura au plus $2^{n-2}$ et au bout de $k$ itérations, la taille de ce qui reste à étudier est de taille au plus $2^{n-k}$.
En particulier, si l'on fait $n$ itérations, il reste au plus $2^{n-n} = 1$ valeur du tableau à examiner. On est sûr de s'arrêter cette fois-ci. On a donc montré que si l'entier $n$ vérifie $l \leq 2^n$, alors l'algorithme va effectuer au plus $n$ itérations. La plus petite valeur est obtenue pour

$$n = \lceil \log_2 l \rceil.$$

Ainsi, la complexité de la fonction est de l'ordre du *logarithme* de la longueur de la liste.

Pour être un peu plus précis dans notre raisonnement, en prenant compte de la valeur que l'on teste (et que l'on n'a donc pas besoin de concerver), il est plus adapté de majorer $l$ par un entier de la forme $2^n - 1$. L'exemple d'exécution présenté figure 1 indique cette majoration.

## 3.3. Correction du programme

Commençons par examiner la question de la *correction* du programme. Comme évoqué plus tôt, deux cas sont à considérer, suivant que la valeur recherchée se trouve ou non dans le tableau ou bien, de façon équivalente, suivant que la valeur retournée est égale à `-1` ou est un entier positif ou nul.

Dans le code du programme, les lignes contenant un **return** sont celles numérotées $8$ et $16$ et, clairement, le renvoi d'un résultat positif ou nul correspond au *return* de la ligne $8$. L'exécution de ce *return* étant subordonné au test `tab[milieu] == val`, le résultat renvoyé correspond bien à la présence dans le tableau de la valeur recherchée à la place indiquée.

Penchons-nous maintenant sur le cas où le programme renvoie `-1`, indiquant ainsi que la valeur recherchée n'est pas présente dans le tableau. C'est le cas le plus intéressant, puisque le programme « affirme » cela sans avoir examiné toutes les valeurs stockées dans le tableau. Cela est bien possible du fait que le tableau est supposé trié.

### Invariant de boucle

Pour prouver cela, nous allons utiliser un *invariant de boucle*, une propriété $P$ (dépendant éventuellement de variables du programme) qui a le comportement suivant :

Si $P$ est vérifiée au début de l'exécution du corps de la boucle,
alors $P$ est encore vérifiée à la fin de l'exécution du corps de la boucle.

Rappelons que l'exécution d'une boucle *while* s'effectue de la façon suivante : on commence par effectuer le test, et si le résultat du test est `true`, on effectue le corps de la boucle et on recommence (en retournant au test). Ainsi, si $P$ est un invariant de la boucle, l'exécution du corps de la boucle va préserver la véracité de $P$.

En particulier, si $P$ est vérifiée en entrant dans la boucle lors du premier test, alors tant que le test donne une réponse positive, la préservation de $P$ lors de l'exécution du corps de la boucle fait que la propriété sera toujours vérifiée à chaque nouveau test. En particulier, si la condition du test n'est plus vérifiée et que l'on sort de la boucle, la propriété $P$ sera encore vérifiée à ce moment-là.

En résumé, si une propriété $P$ est vérifiée en entrant dans une boucle, et si c'est un invariant de la boucle, alors $P$ est enocre vérifiée en sortant de la boucle.

### Preuve de la correction lorsque la valeur n'est pas présente

Pour prouver que si le programme renvoie `-1`, alors c'est bien que la valeur recherchée n'est pas présente dans le tableau, nous allons utiliser l'invariant de boucle suivant :

**Inv** : Si `val` est présente dans `tab`, c'est nécessairement à un indice compris entre `gauche` et `droite` (inclus).

Prouvons qu'il s'agit bien d'un invariant de la boucle de la fonction `recherche_dichotomique`. Pour cela, il faut s'intéresser à une exécution qui va jusqu'à la fin du corps de la boucle : on ne tient pas en compte de cas où l'on sort prématurément à l'aide d'un `return`.
On suppose donc qu'en entrée de boucle, **Inv** est vérifié. Après avoir défini `milieu`, trois cas sont examinés :

1. si `tab[milieu] == val`, on sort de la boucle prématurément à l'aide de l'instruction `return milieu`, donc ce cas ne nous intéresse pas dans le cadre de la preuve d'invariant de boucle.
2. si `tab[milieu] > val`, et si `val` est présente dans le tableau, alors comme celui-ci est ordonné de façon croissante, cela implique que `val` ne peut être présent à l'indice `milieu`, ni avant. On en déduit d'après **Inv** que si `val` se trouve dans `tab` c'est nécessairement à un indice compris entre `milieu + 1` et `droite`. Ainsi, après l'affectation `gauche = ↪ milieu + 1`, **Inv** est encore vérifié.
3. sinon, on a `tab[milieu] < val` et, de même, cela implique que l'on ne peut trouver `val` dans le tableau à un indice inférieur ou égal à `milieu`. Ainsi, en supposant **Inv** vrai au départ et en effectuant l'affectation `droite = milieu - 1`, l'invariant reste vérifié.

Ainsi, dans tous les cas d'une exécution menant à la fin du corps de la boucle, si **Inv** est vérifié au début du corps, il l'est encore à la fin. C'est donc un invariant de la boucle de la fonction `recherche_dichotomique`.

Voyons maintenant comme cela nous permet de prouver la correction de cette fonction lorsque le résultat est `-1`. Avant d'entrer dans la boucle, on a `gauche = 0` et `droite = milieu - 1`, et donc si `val` est présente dans `tab`, c'est nécessairement à un indice compris entre `gauche` et `droite`. Ainsi, **Inv** est vérifié en entrant dans la boucle.
Puisqu'il s'agit d'un invariant de cette boucle, il est encore vérifié en sortant de la boucle. Mais à ce moment, on sait que :

- **Invariant** : si `val` est présente dans `tab` à une position `pos`, alors on a `gauche <= pos <= droite`.
- **Sortie de boucle** : la condition de boucle `gauche <= droite` n'est plus vérifiée, on a donc `droite < gauche`.

En combinant les deux, si `val` est présente dans `tab` à une position `pos`, alors `gauche <= pos <= droite`, ce qui implique que `gauche <= droite`, qui est incompatible avec `droite < gauche`.
Ainsi, en sortie de boucle, `val` ne peut être présente dans le tableau (car aucun indice n'est possible) et le résultat renvoyé `-1` est donc correct.

En conclusion, la fonction `recherche_dichotomique` a deux comportements possibles : si le résultat renvoyé est un entier positif, on a montré qu'il correspond à un indice où se trouve la valeur recherchée dans le tableau ; sinon, le résultat renvoyé est `-1` et on a montré que dans ce cas, la valeur recherchée n'apparaît pas dans le tableau.

# Références

[1] Froidevaux, Gaudel, and Soria. *Types de données et algorithmes*. McGraw-Hill, 1990.