Chapitre 13

Algorithme de dichotomie

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 une liste de façon ordonnée.

Objectifs : Optimiser la recherche dans un tableau (une liste) ordonnée.

I. Première approche

Une première façon de rechercher une valeur dans une liste est d'effectuer une recherche naïve à l'aide d'un parcours de liste.

1. Proposer une fonction `recherche_naive(tab,val)` prenant pour argument une liste et une valeur recherchée dans cette liste. Cette fonction doit retourner la position de cette valeur dans la liste si elle y est présente, la valeur -1 sinon.
2. Combien d'opérations de comparaison doivent être effectuées au pire pour une liste de taille n ?

   .....

3. Et pour une liste de taille $2n$? .....

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 de la liste, ce qui fait savoir que telle valeur de la liste n'est pas la valeur recherchée n'apprend absolument rien sur les autres valeurs de la liste.

II. Deuxième approche

1. Découverte de la dichotomie

Nous allons d'abord programmer le jeu du plus ou moins dont voici le principe :

- L'ordinateur tire au sort un nombre entre 1 et 100.
- Il vous demande de deviner le nombre. Vous entrez donc un nombre entre 1 et 100.
- L'ordinateur compare le nombre que vous avez entré avec le nombre « mystère » qu'il a tiré au sort. Il vous dit si le nombre mystère est supérieur ou inférieur à celui que vous avez entré.
- Puis l'ordinateur vous redemande le nombre.
- … et il vous indique si le nombre mystère est supérieur ou inférieur.
- Et ainsi de suite, jusqu'à ce que vous trouviez le nombre mystère.

1. Coder ce jeu :
2. Avec un peu de méthode, en combien de coups est-on sûr de déterminer la valeur recherchée ? .....

   .....

3. Sans méthode en essayant 1, puis 2 et en testant toutes les valeurs, combien de coups faudrait-il au maximum ?

   .....

   Cette méthode inadaptée est celle adoptée dans la recherche de liste dans la première approche.

2. Le principe de la dichotomie

L'idée centrale de cette 2^{nd} 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 de la liste, il est possible d'améliorer l'efficacité d'une telle recherche de façon conséquente en procédant ainsi :

- on déterminer l'élément m au milieu de la liste ;
- si c'est la valeur recherchée, on s'arrête avec un succès ;
- sinon, deux cas sont possibles :
  - si m est plus grand que la valeur recherchée, comme la liste est triée, cela signifie qu'il suffit de continuer à chercher dans la première moitié de la liste ;
  - sinon, il suffit de chercher dans l'autre moitié.
- on répète cela jusque avoir trouvé la valeur recherchée, 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).

4. Coder cet algorithme, on créera une fonction `recherche_dichotomique(tab, val))` prenant pour argument une liste et une valeur recherchée dans cette liste. Cette fonction doit retourner la position de cette valeur dans la liste si elle y est présente, la valeur -1 sinon.

Remarque :

L'algorithme est donné page suivante, cherchez le avant de le recopier. Pour la suite de ce cours, nous utiliserons celui donné page suivante.

```
def recherche_dichotomique(tab,val):
        gauche=0
        droite=len(tab)-1
        while gauche<=droite:
                milieu=(gauche+droite)//2
                if tabl[milieu]==val:
                        ..........................................................................
                        ..........................................................................
                        return milieu
                elif tabl[milieu]>val:
                        ..........................................................................
                        ..........................................................................
                        droite=milieu-1
                else:
                        ..........................................................................
                        ..........................................................................
                        gauche=milieu+1
        ..................................................................................
        ..................................................................................
        return-1
```

Exemple 1:

3. Terminaison de ce 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.

Définition 1: *Variant de boucle*

Il s'agit d'une quantité entière qui :

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

Si l'on arrive trouver une telle quantité, il est évident que l'on a nécessairement sortir de la boucle au bout d'un nombre fini d'itérations, puisqu'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`
ainsi : `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 droite. En appelant droite' cette nouvelle valeur, on a : droite - gauche < milieu – gauche <= droite – gauche. Ainsi, le variant a strictement décru.
- sinon, on modifie gauche et on a de même : droite – gauche' < droite – milieu <= 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, de cette manière, de justifier la complexité de la recherche dichotomique. On a prouvé qu'il diminue de 1 (au moins) par étape, et 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 ...

III. Complexité de l'algorithme

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 le 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 $\frac{174}{2} = 87$ valeurs. La fois suivante, si 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 :

$$174 \longrightarrow 87 \longrightarrow 44 \longrightarrow 22 \longrightarrow 11 \longrightarrow 6 \longrightarrow 3 \longrightarrow 2 \longrightarrow 1$$

Pour pouvoir majorer le nombre maximum d'itérations, si le tableau contient ℓ valeurs, et si on a un entier n tel que $\ell \leqslant 2^n$, 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 $\frac{2^n}{2} = 2^{n-1}$ 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 $\ell \leqslant 2^n$, alors l'algorithme va effectuer au plus n itérations. La plus petite valeur est obtenue pour $n = log_2(\ell)$.

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 conserver), il est plus adapté de majorer ℓ par un entier de la forme $2n-1$. L'exécution présenté dans l'exemple 1 indique cette majoration.