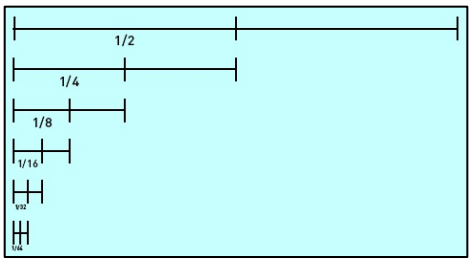


“La recherche dichotomique ne fonctionne que sur un tableau trié. Il faut donc être sûr que le tableau est trié sinon il faut au préalable utiliser un algorithme de tri.”

PARCOURS SEQUENTIEL D’UN TABLEAU ET DICHOTOMIE

Numérique et Sciences Informatiques
1ère

Support de cours :
Jean-Christophe BONNEFOY

Objectifs :

- Savoir écrire des algorithmes de recherche d’une occurrence sur des valeurs de type quelconque.
- Savoir rechercher un extremum, calculer une moyenne sur un tableau de valeurs numériques.
- Montrer la terminaison de la recherche dichotomique par un variant de boucle.

1. Rappels et compléments sur le parcours des tableaux :

1.1 Définition

Un tableau est une structure de données représentant une séquence finie d'éléments auxquels on peut accéder par leur position (appelé aussi indice ou index) dans la séquence. On parle aussi de tableau indexé.

Le plus souvent, en Python, les tableaux sont implémenté par des listes.

Exemple :

liste_ville = ["Clermont-Ferrand","Le Puy-en-Velay","Saint-Etienne","Lyon"]
liste_num = [15, 20, 36, 14]

1.2 Parcours séquentiel d'un tableau

Le parcours séquentiel d'un tableau consiste à faire évoluer un indice, et ainsi à déplacer un pointeur sur chaque case du tableau. La première valeur de l'indice est 0 (le début du tableau). La dernière valeur de l'indice est (longueur - 1).

Dans le langage Python, le parcours d'un tableau (d'une liste) s'implémente par :

```
for i in range(0, len(liste), 1):
    .....
```

ou bien par :

```
for element in liste:
    .....
```

2. Moyenne des éléments (entiers / réels) d'un tableau :

Pour calculer la moyenne des valeurs d'une liste, il faut calculer la somme de ces valeurs et diviser par le nombre de valeurs contenues dans la liste.
Pour calculer la somme on initialise une variable *somme* à zéro, et on itère sur l'ensemble des éléments de la liste. A chaque itération, on ajoute à *somme* la valeur de *element*.

Exemple : Quelle est la moyenne du tableau ci-dessous :

45	21	56	12	1	8	30	22	6	33

L'algorithme en pseudo-code donne :

```
Fonction Moyenne(tableau) :
    somme ← 0
    pour i allant de 0 à longueur (exclue) de tableau :
        somme ← somme + tableau[i]
    moyenne ← somme / longueur
    retourner moyenne
```

Quelle est la complexité en temps de cet algorithme, si l'on considère un tableau à *n* éléments ?

Il y a n passage dans la boucle « pour » et dans chaque passage, il y a 1 affectation et à l'extérieur de cette boucle, il y a 2 affectations : C(n) = n +2, la complexité est donc **linéaire.**

3. Recherche de maximum :

Il s'agit de déterminer l'élément de plus grande valeur présent dans un tableau.

Pour connaître la valeur maximum des valeurs contenues dans un tableau, il faut évaluer tour à tour tous les éléments du tableau. La valeur maximum est mémorisée dans une variable *maximum*, initialisée avec le premier élément du tableau. A chaque étape, on compare l'élément du tableau au contenu de la variable *maximum*. Si l'élément est supérieur à *maximum*, *maximum* prend alors la valeur de cet élément.

L'algorithme en pseudo-code donne :

```
Fonction Maximum(tableau) :
    maximum ← tableau[0]
    pour i allant de 1 à longueur (exclue) de tableau :
        Si tableau[i] > maximum :
            maximum ← tableau[i]
    retourner maximum
```

La complexité en temps est : linéaire.

4. Recherhe dichotomique dans un tableau trié :

4.1 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 :

```
Fonction : Recherche_naive (tab, val) :
    pour i allant de 0 à longueur de tab (exclue) :
        Si tab[i] == val :
            Retourner i
    retourner -1
```

Remarque : Ici, on renvoie un entier positif ou nul en cas de succès, qui correspond à une position de la valeur recherchée dans le tableau, et-1en cas d'échec.

La complexité est linéaire, on n'exploite pas le caractère ordonné du tableau !

4.2 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 :

- on déterminer l'élément m au milieu du tableau
- si c'est la valeur recherchée, on s'arrête avec un succés
- sinon, deux cas sont possibles :
  - 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
  - sinon, il suffit de chercher dans la moitié droite.
- 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.

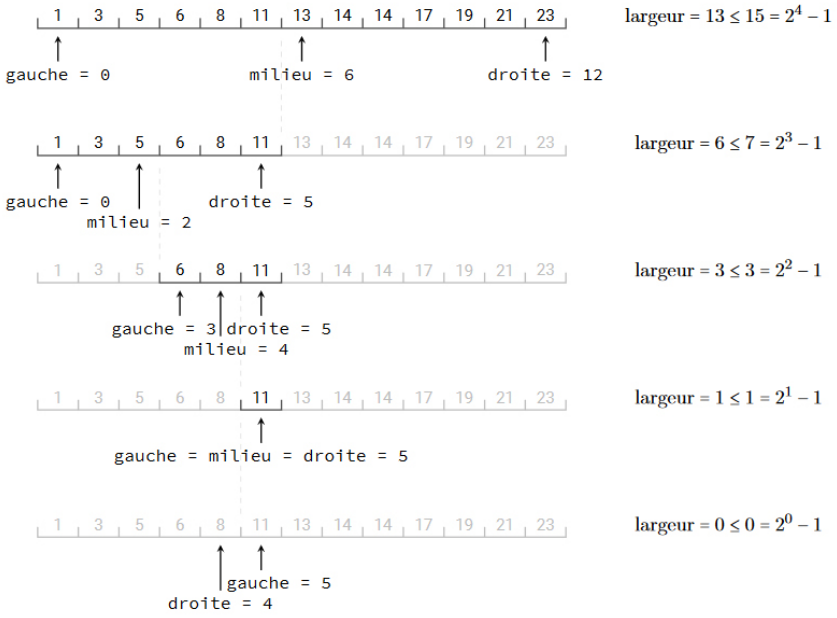
À chaque étape, on coupe l'intervalle de recherche en deux, et on en choisit une moitié. On dit que l'on procède par dichotomie, du grec *dikha*(en deux) et *tomos*(couper)

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

L'algorithme de recherche est le suivant :

```
Fonction : Recherche_dichotomique (tab, val) :
    Gauche ← 0
    Droite ← longueur de tab -1
    Tant que Gauche <= Droite :
            Milieu ← (Gauche + Droite) // 2
            Si tab[Milieu] == val :
                    Retourner milieu
            Sinon :
                    Si tab[Milieu] > val :
                            Droite ← Milieu -1
                    Sinon :
                            Gauche ← Milieu +1
    retourner -1
```

Exemple : On recherche la valeur 10 dans le tableau [1,3,5,6,8,11,13,14,14,17,19,21,23]



4.3 Terminaison de l'algorithme

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 se 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**.

Un variant de boucle est une quantité entière qui :
- ▷doit être positive ou nulle pour rester dans la boucle ;
- ▷doit décroître à chaque itération.

Si l'on arrive à trouver une telle quantité, il est évident que l'on va nécessairement sortir de la boucle au bout d'un nombre fini d'itérations, **puisqu'un entier positif ne peut décroître infiniment** !

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. 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 diminué.

- sinon, on modifie *Droite* et on a de même :
  Droite – Gauche < Milieu – Gauche <= Droite – Gauche
  De même, le variant a strictement diminué.

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

4.4 Complexité

Pour pouvoir majorer le nombre maximum d'itérations, si le tableau contient *m* valeurs, et si on a un entier *n* tel que $m \leq 2^n$, 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,
- au bout de 2 itérations, un quart de tableau aura au plus 2^{n-2} éléments,
- 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 $m \leq 2^n$, alors l'algorithme va effectuer au plus *n* itérations.

La plus petite valeur est obtenue pour $n = \lceil log_2\, m \rceil$. Ainsi, la complexité est **logarithmique et est noté O(log n)**. C'est donc bien plus efficace qu'en O(n) !