L14 : Parcours séquentiel d'un tableau et dichotomie

Lorsque l'on cherche une donnée dans un tableau (ou une liste), il y a toujours plusieurs stratégies pour s'en approcher.

1. Parcours séquentiel

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:
      ...
```

a. Moyenne des éléments 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, il faut donc :

- initialiser une variable `somme` à zéro
- itérer sur l'ensemble des éléments de la liste
- à chaque itération, ajouter à `somme` la valeur de l'élément.

L'algorithme en pseudo-code est donc le suivant :

```
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 : **T(n) = n +2**, la complexité est donc **linéaire**.

b. Recherche d'un 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` qui a été initialisée avec le premier élément du tableau
- ❏ à 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
```

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

……

……

……

……

La complexité en temps est : **linéaire.**

2. Recherche par dichotomie

L'idée centrale de cette approche repose sur l'idée de réduire de moitié l'espace de recherche à chaque étape. On teste 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.

Le nom **dichotomie** provient du grec ancien *tomia*, couper et *dikha*, en deux.

Attention, 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 (comme vu au chapitre précédent).

a. La méthode de recherche par dichotomie

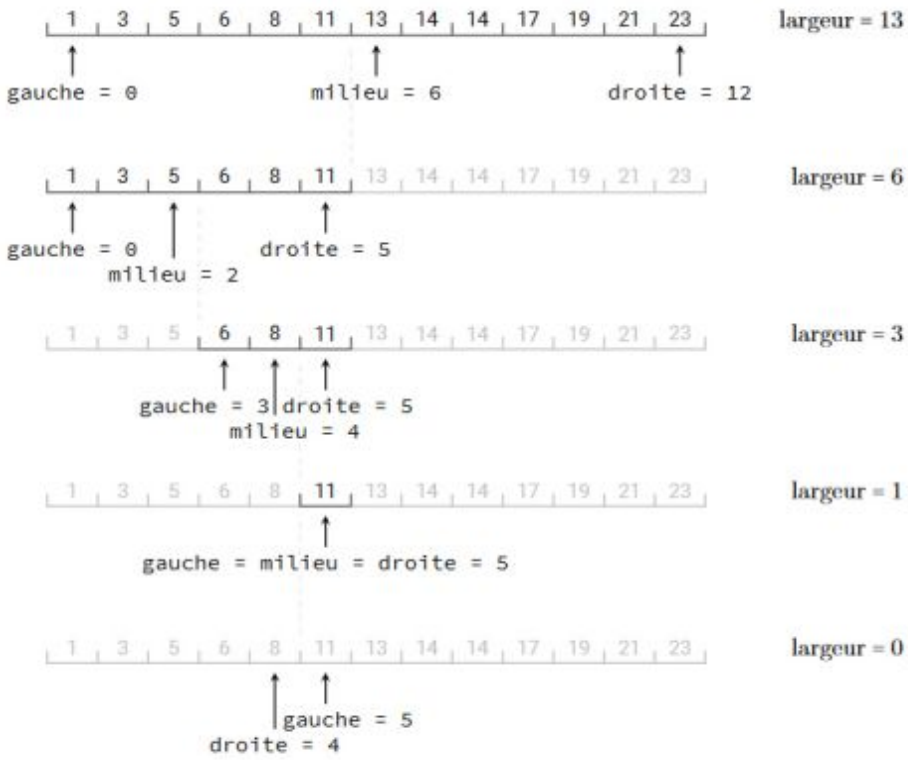
Elle consiste à :

- déterminer l'élément `m` au milieu du tableau
- 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 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 jusqu'à avoir trouvé la valeur recherchée ou bien jusqu'à avoir réduit l'intervalle de recherche à un intervalle vide, ce qui signifie que la valeur recherchée n'est pas présente.

L'algorithme en pseudo-code donne :

```
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]`



b. 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 :

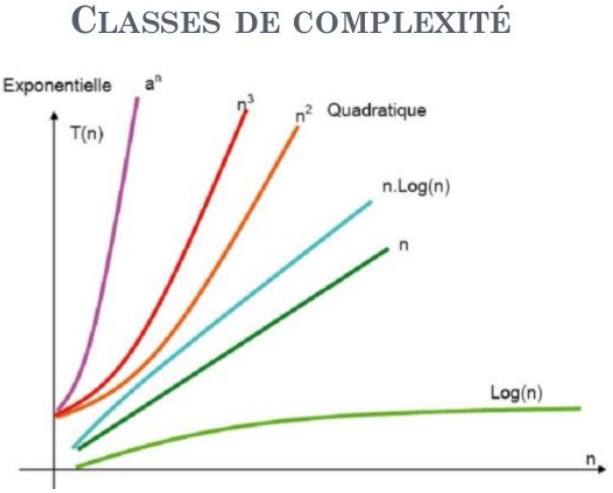
- au bout d'une itération, une moitié de tableau aura au plus $2^n/2 = 2^{n-1}$ éléments,
- au bout de 2 itérations, un quart de tableau aura au plus 2^{n-2} éléments,
- et au bout de k itérations, la taille de ce qui reste à étudier est de taille au plus 2^{n-k}.

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 notée **O(log n)**.

C'est donc bien plus efficace qu'en O(n) !



3. Exercices

Exercice 1 : Le juste prix

Dans le jeu du *Juste prix*, l'animateur présente un objet et donne une fourchette de prix. Le joueur doit deviner le prix de l'objet en un nombre maximum de propositions.

A chaque proposition du joueur, l'animateur précise si le prix proposé est supérieur ou inférieur au prix réel. Dans notre exemple, l'animateur est l'ordinateur.

1. Ecrire une fonction `juste_prix(prixSup)` qui respecte la spécification suivante :

- ❏ elle doit simuler une partie de *Juste prix* ;
- ❏ elle doit prendre comme paramètre un prix maximum ;
- ❏ elle ne doit pas afficher le résultat mais retourner un couple constitué du juste prix et du nombre de propositions soumises, qui peut ensuite être utilisé pour un affichage avec un `print`;
- ❏ l'ordinateur choisit d'abord le prix entier qu'il faut deviner entre `0` et `prixSup` euros, à l'aide de la fonction `randint` de la bibliothèque `random` ;
- ❏ dans cette question le nombre de propositions n'est pas limité.

2. Si on considère le cas d'une borne supérieure de prix de 1000 €, peut-on être sûr que la fonction déterminera le juste prix en un nombre fini d'essais ? Si oui, peut-on déterminer le nombre maximum de propositions ?

3. Modifier la fonction précédente en `juste_prix2(prixSup, nmax)`, qui prend comme deuxième paramètre un nombre maximum de propositions `nmax`.

   Par exemple, la fonction pourrait afficher des résultats comme ci-dessous :

```
Bravo ! Vous avez deviné le juste prix 357 € en 5 propositions.
```

   ou encore :

```
Vous avez atteint le nombre maximum 10 de propositions sans deviner le
juste prix de 743 € .
```

4. Écrire une fonction `seuil_nbpropositions(prixSup)` qui retourne le nombre maximum de propositions nécessaires pour déterminer un juste prix dans l'intervalle `[0; prixSup]` avec l'algorithme des fonctions `juste_prix` et `juste_prix2`.