Complexité algorithmique

Badraddine AGHOUTANE
b.aghoutane@umi.ac.ma

Complexité algorithmique

Pourquoi Analyser la Complexité des Algorithmes?

Le but est de :

1. Calculer le **temps d'exécution** théorique d'un algorithme et **l'espace mémoire** nécessaire à ses données;
2. **Comparer des algorithmes** en termes de complexité et de choisir le meilleur;
3. Tenter **d'optimiser l'algorithme** conçu en réduisant au maximum sa complexité.

Critère d'évaluation de complexité

L'évaluation de la **complexité d'un algorithme** se fait par l'analyse relative des deux **ressources de l'ordinateur** :

- **Q1: temps de calcul,**
- **Q2: espace mémoire.**

Ces **deux ressources** sont utilisées par un **programme**, pour **transformer** les **données** d'un problème en un ensemble de **résultats**.

- L'**analyse de la complexité** consiste à mesurer ces deux grandeurs pour choisir l'**algorithme optimal** pour résoudre un problème :

  ➔**Algorithme le plus rapide et le moins gourmant en consommation mémoire**

N.B : On s'intéresse à l'analyse de la **complexité temporelle (Q1)** d'un algorithme en déterminant un **ordre de grandeur** (théorique), indépendamment de l'ordinateur utilisé.

Définition de la complexité algorithmique

La **complexité temporelle** est une mesure du **temps requis par un algorithme** pour accomplir sa tâche. Le calcul de la complexité se base sur :

1. **Taille de donnée :** la complexité sera notée en fonction de cette taille. Par exemple, un algorithme qui manipule un **tableau** (la taille, c'est le nombre d'éléments). Ainsi, traiter un tableau avec N=100 éléments est bien différent que de traiter un tableau avec N=1 000 000 éléments.
2. **Unité d'évaluation :** le temps d'exécution d'un algorithme est proportionnel au nombre d'opérations jugées fondamentales. Par exemple : pour rechercher un élément X dans un tableau, l'unité pourra être la **durée d'exécution d'une instruction élémentaire** (on suppose que les instructions élémentaires ont toutes le même temps d'exécution). On peut également prendre comme unité le coût d'une **comparaison de l'élément recherché X à un élément du tableau**.

Calcul de la complexité

Calculer la complexité d'un **algorithme** revient à trouver un **ordre de grandeur** : **Coût(n)=O(...)** représentant le **temps d'exécution théorique** de l'algorithme en fonction de la **taille n** des données d'entrée et les **opérations fondamentales.**

- **Instruction simple :**

La complexité d'une instruction d'affectation (*sans appel de fonction*), de lecture ou d'écriture peut en général se mesurer par **O(1)** (indépendant de n).

- **Séquence :**

La complexité d'une **séquence de n instructions** "i" est égale à la somme des complexités de chaque instruction "i" qui la compose.

$$\text{Coût}_{\text{Séquence}}(d)=\sum_{i=1...n} \text{coût}_{\text{Instruction i}}(d)$$

- **Sous-programme (fonction ou procédure) :**

Pour les appels des procédures/fonctions (sans récursivité), la complexité est égale à la somme des complexités de chacune de ses instructions.

Calcul de la complexité

- **Le choix :**

La complexité d'une instruction conditionnelle :

Si <condition> alors séquence1 sinon séquence2 FinSi

La **complexité d'une structure conditionnelle (choix)** est égale à la somme de la **complexité de la condition d'évaluation** et la **plus grande complexité** entre la séquence d'instructions exécutée si la condition est vraie et celle exécutée si la condition est fausse.

$$\text{Coût}_{\text{Choix}}(d)= \text{coût}_{\text{évaluation-condition}}(d) + \text{Max}\ (\text{coût}_{\text{séquence1}}(d), \text{coût}_{\text{séquence2}}(d))$$

- **Répétition :**

La complexité d'une boucle est égale à la somme de la complexité de la condition d'évaluation de sortie et celle de la séquence d'instructions qui constitue le corps de la boucle.

$$\text{Coût}_{\text{Boucle}}(d)= \text{coût}_{\text{évaluation-condition-sortie}}(d) + \text{coût}_{\text{séquence}}(d)$$

Calculer la complexité d'un algorithme

- Pour calculer la **complexité d'un algorithme** donné, il convient tout d'abord de compter le nombre d'opérations impliquées par son exécution.
- Notation O **(Ordre de ...)** :
  - La notation la plus utilisée pour noter la complexité d'un algorithme est la **notation O (Ordre de ...)**, qui dénote un ordre de grandeur.
  - Par exemple, on dira d'un algorithme qu'il est **O(15)**, s'il nécessite au **maximum 15 opérations** (*dans le pire cas*) pour se compléter.

Catégorie de complexité d'un algorithme

Il existe plusieurs **catégories de complexité** à savoir la:

- **complexité constante** : indépendante de la taille des données (**O(15), O(1)**)
- **complexité logarithmique :** est en (***$log_2(n)$***) *(resp en* ***$O(log_2(n))$***)
- **complexité linéaire :** est en (**n**) (*resp en* ***O(n)***)
- **complexité quadratique :** est en ($\mathbf{n^2}$) (*resp en* ***$O(n^2)$***)
- **complexité polynomiale :** est en ($\mathbf{n^p}$) (*pour un certain p*)
- **complexité exponentielle,**
- **complexité factorielle,**
- ...

Algorithmes de complexité constante

Soit une fonction qui prend en paramètre le rayon d'une sphère, calcul le volume de cette sphère, et retourne cette valeur au programme appelant. On a le code suivant :

1. On peut compter l'instruction notée **(0)** comme étant une opération
2. On peut compter l'instruction notée **(1)** comme **cinq opérations ou bien comme une seule opération**
3. L'instruction notée **(2)**, du retour de la valeur résultante de l'exécution de la fonction, est aussi une opération.

```
Fonction volume_sphere(rayon : Réel) : Réel
Constante PI = 3,14      // (0)
Variable volume : Réel
Début
        volume ← 4 / 3 * PI * rayon ^ 3; // (1)
        retourne(volume);   // (2)
FinFonction
```

➔ **L'algorithme sera donc O(3) ou O(7), en fonction de la manière de compter les différentes opérations.**

Algorithmes de complexité constante

- Le plus important ici n'est pas la valeur exacte suivant le Grand O (*O(3) ou O(7), …*), mais le fait que cette valeur soit **constante (***indépendante de la taille des entrées***).**
- Lorsqu'un algorithme est O(c) où « c » est une constante, on dit qu'il s'agit alors d'un **algorithme en temps constant.**

➔ Une **complexité constante** est la complexité algorithmique idéale, puisque **peu importe la taille de l'échantillon à traiter**, l'algorithme prendra toujours un nombre fixé à l'avance d'opérations pour réaliser sa tâche.

- Tous les **algorithmes en temps constant** font partie d'une **classe nommée O(1)**. En général, qu'un algorithme soit **O(3)**, **O(17)** ou **O(100000)**, on dira qu'il est en fait **O(1)** puisque la différence de performance entre deux algorithmes en temps constant peut être satisfaite par un simple remplacement matériel (*utiliser un processeur plus rapide, par exemple*).

Exemple en Python

Exemple 1 : la fonction append

- Le code ci-dessous consiste à programmer la fonction append

```
def AjoutFin(L,a) :
    L=L+[a]
    return L
```

- Exemple

```
>>> L = [1, 2]
>>> AjoutFin(L, 3)
[1, 2, 3]
```

- Le nombre d'opérations est : 3 (concaténation, affectation et return) ;
- Quelque soit la taille de liste, le nombre d'opérations est constant ;
- Temps de calcul est constant
- Complexité : $O(1)$

Règles de calcul de la complexité

- **La complexité asymptotique**, est une approximation de la fonction du temps, appelée la notation grand O.
  - **Exemple :** Prenons un algorithme dont la fonction du temps est **$C(n) = 4n^3$.** La complexité asymptotique **ne s'occupe pas du facteur constant 4**, mais garde uniquement la *forme générale* (approximation) de la fonction : **$O(n^3)$**
  - Cette **approximation (grand O)** permet de ne pas tenir compte de la différence de vitesse d'exécution entre les différents ordinateurs (*plus ou moins rapide*).

De ce fait, d'autres règles de calcul de la complexité asymptotique d'un algorithme :

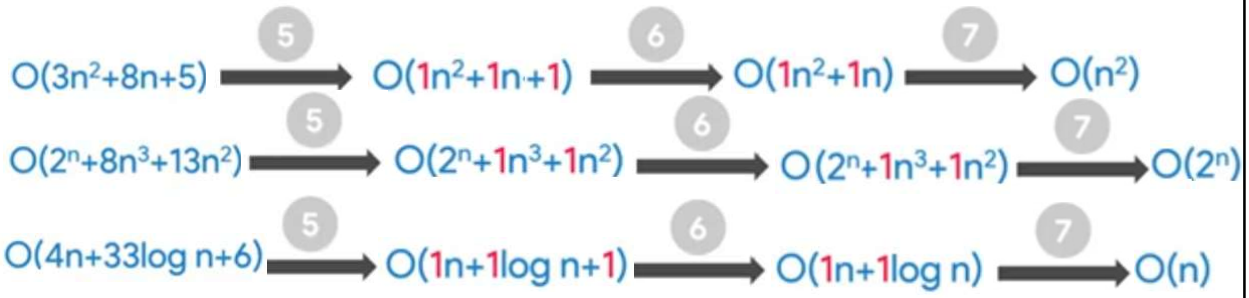
5. Les constantes multiplicatives sont **remplacées par 1** **($4n^4+3 \rightarrow 1n^4+3 \rightarrow n^4+3$)**
6. Les constantes additives sont **annulées** **($n^4+3 \rightarrow n^4$)**
7. Le terme le plus élevé est **conservé** **($n^4+n^2+n \rightarrow n^4$)**

Règles de calcul de la complexité

De ce fait, d'autres règles pour calculer la complexité asymptotique:

5. Les constantes multiplicatives sont **remplacées par 1**
6. Les constantes additives sont **annulées**
7. Le terme le plus élevé est **conservé.**

Exercice : Donner la complexité asymptotique?



Algorithmes de complexité linéaire

Exemple 1 :

Calculez la complexité (nombre d'actions exécutées par le **segment S)**, du code suivant:

```
Pour i←2   jusqu'à n pas 1 faire
        somme← somme + i
fin pour
```

- **Complexité de la boucle: En détail, nous avons:**

$$\text{Coût}_{\text{Boucle}}(d) = \text{coût}_{\text{condition}}(d) * (\text{Nbre-itération} + 1) + \text{coût}_{\text{séquence}}(d) * \text{Nbre-itération}$$

Nbre-itération d'une boucle (Pour… faire ou Tant que… faire) avec pas =1 est:

Nbre-itération=V.F–V.I+1

(valeur finale de l'indice de parcours, moins sa valeur initiale, plus 1

Dans notre cas, le code du **segment S** :

$$\text{Nbre-itération} = V.F - V.I + 1 = n - 2 + 1 = n - 1$$

Algorithmes de complexité linéaire

- La condition de sortie dans le **code S** est le test **(i>n)** qui est de **O(1)**, donc :

$$\text{coût}_{\text{évaluation-condition-sortie}}(d) * (\text{Nbre-iteration} + 1) = (n-1+1)\ \theta(1) = n\ \theta(1)$$

- Le corps de la boucle du **code S** est une **affectation**, donc de **O(1)**

$$\text{coût}_{\text{séquence}}(d) * \text{Nbre-iteration} = (n-1)\ \theta(1)$$

Ainsi :

$$\text{Coût}_{\text{Boucle}}(d) = \text{coût}_{\text{évaluation-condition-sortie}}(d) * (\text{Nbre-itération} + 1) + \text{coût}_{\text{séquence}}(d) * \text{Nbre-itération}$$

$$\text{Coût}_{\text{Boucle}}(d) = (n-1)\ \theta(1) + n\ \theta(1) = (2n-1)\ \theta(1). \quad \text{Ainsi: } \text{Coût}_S(n) \simeq \theta(n)$$

Algorithmes de complexité linéaire

Exemple 2 :

Algorithme **moyenne-de-n-nombre**

Variables n, somme, i, nombre : entier; moyenne : réel;

Début

1. Lire(n)
2. somme ← 0
3. i← 1
4. **Tant que** i <= n **faire**
5. lire(nombre)
6. somme← somme + nombre
7. i ← i+1

Fin tanque

8. moyenne← somme/n

fin

Exemple 2: Algorithme **moyenne_de_n_nombre**

Chacune des actions **1, 2, et 3** sera exécutée **une seule** fois.

Chacune des actions **5, 6 et 7** sera exécutée **n** fois (**Nbre-itération=V.F–V.I+1=n -1 +1**)

L'action **4** qui contrôle la boucle sera exécutée **n + 1** fois (**Nbre-itération + 1= n+1**)

Et l'action **8** sera exécutée une **seule fois**:

Actions	Nombre de fois	Temps d'exécution (ordre de grandeur)	Justification
1	1	θ (1)	Instruction Simple
2	1	θ (1)	Instruction Simple
3	1	θ (1)	Instruction Simple
4	n + 1	(n+1)θ (1)	Évaluation Cond boucle avec pas =1 (Nbre-itération + 1) *1
5	n	nθ (1)	Corps Boucle: Nbre-itération *1
6	n	nθ (1)	Corps Boucle: Nbre-itération *1
7	n	nθ (1)	Corps Boucle: Nbre-itération *1
8	1	θ (1)	Instruction Simple

$Coût_{moyenne\text{-}de\text{-}n\text{-}nombre}(n) = [1+1+1+(n+1)+n+n+n+1]\ \theta(1) = (4n+5)\ \theta(1) \simeq \theta(n)$

$Coût_{moyenne\text{-}de\text{-}n\text{-}nombre}(n) = (4n+5)\ \theta(1) \simeq \theta(n)$

Exemple en Python

Exemple 3 : boucle simple

```
for i in range(n) :
  print("Bonjour")#une instruction s
```

- Dans la boucle for,on a une seule instruction : print
- Temps de calcul de print : T_s est constant $T_s = C$
- Nombre de fois d'exécution de cette instruction est n
- Le nombre total d'opérations est $n * 1 = n$
- Temps de calcul total : $T(n) = n * T_s$
- Complexité : $O(n)$

Algorithmes de complexité linéaire

- Par complexité linéaire, ou O(n), on dénotera des algorithmes pour lesquels le nombre d'étapes à effectuer changera en proportion directe de la taille de l'échantillon à traiter : ***Si l'échantillon croît par un facteur de 100, la complexité sera étendue et augmentera aussi par un facteur de 100.***
- Calculer la complexité de l'algorithme qui calcul la somme des éléments d'un tableau :

```
Fonction somme_elemts(tableau tab[N] : Entier) : Entier
Variable somme, i : Entier
Début
                    somme ← 0;
                    Pour i←0 à N-1 [par 1] faire
                        somme ← somme + tab[i];
                    FinPour
                    retourne somme;
FinFonction
```

Exemple en Python

Exemple 4 : remplir un tableau

```
def RemplirTab(T) :
    for i in range(len(T)) :
        print("T[",i,"]=",end="")
        T[i]=int(input())
```

- Le paramètre de complexité est la taille du tableau d'entrée T.
- On fixant *i* on exécute 3 opérations : print, input et affectation
- Nombre de fois d'exécution de ces 3 opérations est : $len(n)$
- Le nombre total d'opérations est $3 * len(n)$
- Complexité : $O(len(T))$

Algorithmes de complexité linéaire

L'algorithme qui permet de chercher une valeur dans un tableau et de retourner sa position.

On en compte ici, en plus de la boucle, deux opérations :

1. **« avant la boucle »** l'initialisation de la variable i←1,
2. **« après la boucle »** la valeur de retour de la fonction (**retourner -1**)

En plus, pour chaque itération de la **boucle pour**, on a trois opérations (*comparaison (i<=N), un test si la case du tableau contient la valeur recherchée et la valeur de retour dans ce cas (l'indice i)).*

```
Fonction  balayage (tableau tab[N] : Entier, valeur_cherchee : Entier) : Entier
Variable i, N : Entier
Début
    Pour i allant de 1 à N faire
                Si (tab[i]=valeur_cherchee) alors
                        Retourner i
    FinPour
    Retourner -1
Fin
Ecrire(balayage([5,7,12,14,23,27,35,40,41,45], 35)
```

```
#Recherche par balayage
def balayage(tableau, valeur_cherchee):
    N=len(tableau)
    for i in range (N):
        if (tableau[i]==valeur_cherchee):
            return i
    return -1

print(balayage([5,7,12,14,23,27,35,40,41,45], 35))
```

Algorithmes de complexité linéaire

Soit un autre algorithme qui permet de chercher une valeur dans un tableau et de retourner sa position.

On en compte ici, en plus de la boucle, quatre (4) opérations :

1. **« avant la boucle »** – l'initialisation de la variable compteur,
2. **« après la boucle »** la comparaison de compteur avec N,
3. **l'affectation** d'une valeur à la variable indice,
4. la valeur de retour de la fonction (**retourne**)

En plus, pour chaque itération de la **boucle Tantque**, on a trois opérations (*deux comparaisons et une incrémentation du compteur).*

```
Variable N : Entier
Fonction trouver_indice(tableau tab[N] : Entier, val : Entier) : Entier
Variable : indice  , compteur : Entier
Début
            compteur = 0;   (1)
    TantQue (compteur < N && tab[compteur] != val) faire
            compteur ← compteur +1;
    FinTantQue
            si (compteur < N) Alors                    (2)
              indice = compteur;                       (3)
            Finsi
            retourne indice;                           (4)
FinFonction
```

Algorithmes de complexité logarithmique

- **Dans le meilleur cas** (si val se trouve à l'élément 0 de tab[]), il nous faut seulement (2+4) opérations pour produire la solution, où le « 2 » constitue les deux conditions à évaluer au début de la boucle, et le « 4 » est le nombre d'opérations incontournables à faire.

```
Variable N : Entier
Fonction trouver_indice(tableau tab[N] : Entier, val : Entier) : Entier
Variable i, compteur : Entier
Début
                compteur = 0;          (1)
    TantQue (compteur < N && tab[compteur] != val) faire
                    compteur ← compteur +1;
     FinTantQue
                si (compteur < N) Alors                    (2)
                   indice = compteur;                      (3)
                Finsi
                retourne indice;                           (4)
FinFonction
```

Algorithmes de linéaire

- **Dans le cas moyen**, (*si val se trouve au milieu du tableau)*, il faudra en moyenne 3*(N/2)+4 opérations pour produire la solution.
- Cela signifie qu'en **moyenne**, cet algorithme nécessitera :
  - **19**, donc 3×(10/2)+4, opérations pour un tableau de **10 éléments**
  - **154**, donc 3×(100/2)+4, opérations pour un tableau de **100 éléments**
  - **1504**, donc 3×(1000/2)+4, opérations pour un tableau de **1000 éléments**

C(n)=(3*(N/2)+4)*O(1)≈ O(N)

```
Variable N : Entier
Fonction trouver_indice(tableau tab[N] : Entier, val : Entier) : Entier
Variable : indice  , compteur : Entier
Début
                compteur = 0;          (1)
    TantQue (compteur < N && tab[compteur] != val) faire
                    compteur ← compteur +1;
     FinTantQue
                si (compteur < N) Alors                    (2)
                   indice = compteur;                      (3)
                Finsi
                retourne indice;                           (4)
FinFonction
```

Algorithmes de complexité linéaire

- **Dans le pire cas** (*si val se trouve au dernier élément du tableau, ou si cette valeur n'est pas du tout présente dans le tableau*), alors, il nous faut **« 3*N+4 »** opérations pour produire la solution *(pour N itérations : 3*N, auquel s'ajoutent les 4 opérations de base*).
- Notez toutefois que dans **le pire cas**, cet algorithme nécessitera :
  - **34**, donc 3×10+4, opérations pour un tableau de **10 éléments**
  - **304**, donc 3×100+4, opérations pour un tableau de **100 éléments**
  - **3004**, donc 3×1000+4, opérations pour un tableau de **1000 éléments**

C(n)=(3*N+4)*O(1)≈ O(N)

La **complexité d'un algorithme** est un calcul de ses performances **asymptotiques** dans le **pire des cas**

```
Variable N : Entier
Fonction trouver_indice(tableau tab[N] : Entier, val : Entier) : Entier
Variable : indice , compteur : Entier
Début
                compteur = 0;   (1)
    TantQue (compteur < N && tab[compteur] != val) faire
                compteur ← compteur +1;
    FinTantQue
                si (compteur < N) Alors                    (2)
                   indice = compteur;                      (3)
                Finsi
                retourne indice;                           (4)
FinFonction
```

Complexité logarithmique : Recherche dichotomique

La recherche dichotomique est un algorithme de recherche qui permet de déterminer la position d'un élément dans un tableau trié.

1. Cet algorithme compare la valeur recherchée à la valeur du milieu du tableau.
2. Si c'est la valeur recherchée, on s'arrête et on retourne sa position.
3. Si la valeur recherchée est plus petite, alors la valeur recherchée est située dans la partie gauche du tableau, sinon elle est dans la partie droite.

On répète le procédé de comparaison jusqu'à ce que l'on obtienne la valeur recherchée, ou jusqu'à ce que l'on ait réduit l'intervalle de recherche à un intervalle vide : ce qui signifie que la valeur recherchée n'est pas présente dans le tableau.

À chaque étape, la zone de recherche de la valeur est divisée par deux.

➔ Pour chaque itération (étape), l'algorithme va réduire (*de moitié*) la partie des données à traiter : **complexité logarithmique**

Complexité logarithmique : Recherche dichotomique

L'algorithme de **recherche dichotomique** retourne la position de l'élément recherché si celui-ci se trouve dans le tableau, et -1 sinon.

```
#Recherche dichotomique : Version itérative
Fonction dichotomie(tableau tab[] : Entier, valeur_cherchee : Entier) : Entier
        Variables debut_idx, fin_idx, milieu : Entier
        Début
                debut_idx ← 0
                fin_idx ← len(tableau)-1
                TantQue (fin_idx >= debut_idx) faire
                        milieu ← (debut_idx + fin_idx) / 2
                        si (valeur_cherchee = tableau[milieu]) alors
                                retourner milieu
                        si (valeur_cherchee > tableau[milieu]) alors
                                debut_idx ← milieu+1
                        sinon
                                fin_idx ← milieu-1
                 FinTantQue
                        retourner -1
        FinFonction
```

Complexité logarithmique : Recherche dichotomique

Le pire cas : consiste à éliminer successivement la moitié des éléments restant à explorer jusqu'à ce qu'il n'en reste plus qu'un seul, qui sera le bon élément ou bien on indiquera que l'élément cherché est absent du tableau (retourner -1).

Analysons un peu plus ce pire cas :

- Si on explore un tableau de **10 éléments**, on aura besoin au pire des cas de **4 itérations**: l'espace à explorer sera initialement de 10 éléments, puis de 5 éléments, puis 2, puis d'un seul.
- Si on explore un tableau de **100 éléments**, on aura besoin au pire des cas de **7 itérations**: l'espace à explorer sera initialement de 100 éléments, puis de 50 éléments, puis 25, puis 12, puis 6, puis à 3, puis d'un seul…

La **complexité de cet algorithme** est : **$O(b+(i\times log_2(N)))\approx O(log2(N))$**

Où N est la taille du tableau, b est le nombre incontournable d'opérations (ici, b=3) et i est la complexité d'une seule itération (ici, i=5).

Complexité logarithmique : Recherche dichotomique

Programme en Python

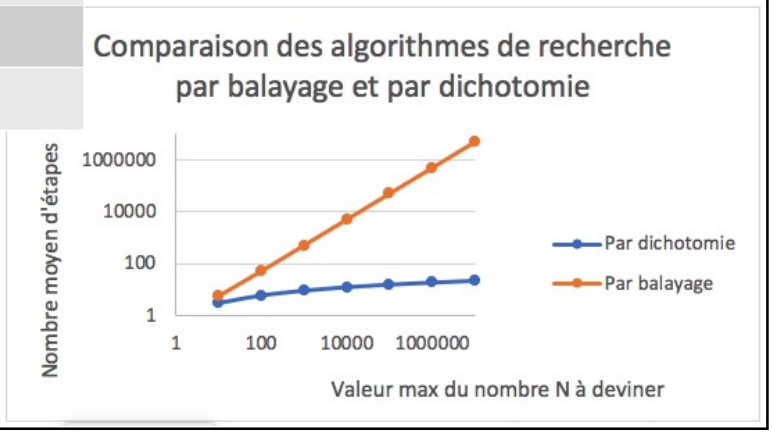
On va écrire un programme Python qui retourne la position de l'élément x si celui-ci se trouve dans le tableau, et None si l'élément ne s'y trouve pas.

```
#Recherche dichotomique : Version itérative
def dichotomie(tableau, valeur_cherchee):
    debut = 0
    fin = len(tableau)-1
    while fin >= debut:
        milieu = (debut + fin)//2
        if valeur_cherchee == tableau[milieu]:
            return "L'élément se trouve dans l'indice : "+str(milieu)
        if valeur_cherchee > tableau[milieu]:
            debut = milieu+1
        else:
            fin = milieu-1
    return "Valeur Absente"

print(dichotomie([5, 7, 12, 14, 23, 27, 35, 40, 41, 45], 45))
```

Complexité logarithmique : Recherche dichotomique

N	O(3*N + 4)	O(3*(N/2) + 4)	O(5+5*$log_2 N$)
10	34	19	**25**
100	304	154	**40**
1000	3004	1504	**55**
10000	30004	15004	**75**



Les classes de complexité