Complexité et preuves d'algorithmes

11 mai 2020

Quelles qualités peut-on demander à un algorithme ou à un programme ?

- la première est bien sûr qu'il soit juste, c'est-à-dire qu'il réalise effectivement la tâche qu'on lui a demandé
- le code doit être bien écrit, compréhensible par une tierce personne en vue d'une maintenance ou d'une amélioration
- on peut ensuite lui demander d'être performant, pour cela on distingue deux critères essentiels :
  - le temps de calcul nécessaire, directement lié au nombre d'opérations qu'effectue l'algorithme
  - la quantité de mémoire nécessaire.

  On parle respectivement de complexité temporelle et de complexité spatiale.

Les deux scripts suivants qui échangent les valeurs des variables `x` et `y` valeurs illustrent ces deux problématiques :

```
temp = x
x = y
y = temp
```

```
x = x + y
y = x - y
x = x -y
```

La première procédure nécessite l'utilisation de 3 variables et coûte 3 affectations. La seconde est plus économique en mémoire, seulement deux variables, mais nécessite 3 affectations et en plus 3 additions. La première procédure a donc une complexité temporelle meilleure mais une complexité spatiale moins bonne.

Nous allons essentiellement nous intéresser à la complexité temporelle. On peut mesurer le temps de calcul de façon expérimentale en chronométrant mais il est intéressant d'avoir une estimation théorique, en déterminant le nombre d'opérations significatives que fait l'algorithme.

Dans la suite le terme complexité désignera la complexité temporelle.

1 Notion de complexité

1.1 Un exemple introductif : entiers sommes de deux carrés

Voici quatre scripts qui déterminent la liste des décompositions possibles d'un entier $n \in \mathbb{N}$ comme somme de deux carrés.

```
def algo1(n):
    t = []
    for i in range(n+1):
        for j in range(n+1):
            if i**2 + j**2 == n:
                t.append((i,j)) # on ajoute le tuple (i,j)
    return t
```

Il y a exactement $(n+1)^2$ itérations, donc $(n+1)^2$ additions et $2(n+1)^2$ multiplications.

```
def algo2(n):
    t = []
    for i in range(n+1):
        for j in range(0,i+1): # pour j de 0 à i. On évite les doublons (i,j) et (j,i)
            if i**2 + j**2 == n:
                t.append((i,j))
    return t
```

Le nombre d'itérations est $\sum_{i=1}^{n}\sum_{j=0}^{i} 1 = \sum_{i=0}^{n}(i+1) = 1+2+\ldots+(n+1) = \dfrac{(n+1)(n+2)}{2}$. Il y a donc $\frac{(n+1)(n+2)}{2}$ additions et $(n+1)(n+2)$ multiplications.

```
def algo3(n):
    t = []
    N = math.floor(math.sqrt(n))
    for i in range(N+1): # si i^2+j^2 = n, alors i^2 <= n donc i <= sqrt(n)
        for j in range(0,i+1): # pour j de 0 à i. On évite les doublons (i,j) et (j,i)
            if i**2 + j**2 == n:
                t.append((i,j))
    return t
```

Le nombre d'itérations est $p = \frac{(\lfloor\sqrt{n}\rfloor+1)(\lfloor\sqrt{n}\rfloor+2)}{2}$. Il y a p additions, $2p$ multiplications et un calcul de racine carrée.

```
def algo4(n):
    t = []
    N = math.floor(math.sqrt(n))
    for i in range(N+1):
        j = math.sqrt(n -i**2) # candidat pour être solution
        if j == math.floor(j): # si j est entier
                t.append((i,int(j)))
    return t
```

Le nombre d'itérations est $p = (\lfloor\sqrt{n}\rfloor + 1)$. Il y a p additions, p multiplications et $p+1$ calculs de racine carrée.

Voici les temps de calcul en secondes observés pour $n = 10^k$:

k	1	2	3	4	5	6	7
algo 1	0.001	0.125	9.74	1000			
algo 2	0.002	0.064	4.807	481			
algo 3	0	0.001	0.005	0.05s	0.485	4.85	48 s
algo 4	0	0	0.001	0.004	0.01	0.02	0.04

Commentaires des observations :

- Effectivement algo2 semble être deux fois plus rapide que algo1 (on fait la moitié des calculs).
- On observe pour algo1 que si n est multiplié par 10, le temps de calcul est environ multiplié par 100. Pour $n = 10^5$, on peut donc prévoir un temps de calcul de 100000 secondes, ce qui représente plus de 24h de calculs. De même, pour $n = 10^7$, on peut prévoir un temps de 10^9 secondes, soit plus de 30 années ! Inutile donc de tenter le calcul !
- Pour algo3, si n est multiplié par 10, le temps de calcul semble aussi environ multiplié par 10.

1.2 Formalisation

Nous allons formaliser tout ceci :

Pour définir la complexité d'un algorithme dépendant d'un paramètre n, on peut estimer le nombre d'opérations «significatives» qu'il effectue. On dira qu'il a une complexité en $O(f(n))$ s'il existe une constante K telle que à partir d'un certain rang son coût est inférieur à $Kf(n)$.

Par exemple, si on reprend l'exemple précédent des décompositions en sommes de carrés, on a

$$(n+1)(n+2) \underset{n\to+\infty}{\sim} n^2 = O(n^2) \quad \text{et} \quad \frac{(n+1)(n+2)}{2} \underset{n\to+\infty}{\sim} \frac{n^2}{2} = O(n^2).$$

Les fonctions algo1 et algo2 ont donc une même complexité en $O(n^2)$. On parle de complexité quadratique.

En revanche pour algo3, on a

$$\frac{(\lfloor\sqrt{n}\rfloor + 1)(\lfloor\sqrt{n}\rfloor + 2)}{2} \underset{n\to+\infty}{\sim} \frac{\sqrt{n}\times\sqrt{n}}{2} = \frac{n}{2} = O(n)$$

On dit que algo3 a une complexité en $O(n)$, on parle de complexité linéaire.

	$\log_2 n$	n	$n.\log_2 n$	n^2	n^3	2^n
10^2	0.66 μ s	10 μ s	66 μ s	1 ms	0.1s	4×10^{15} ans
10^3	1 μ s	100 μ s	1 ms	0.1s	100 s	
10^4	1.3 μ s	1 ms	13 ms	10 s	1 jour	
10^5	1.6 μ s	10 ms	0.1 ms	16 min	3 ans	
10^6	2 μ s	100 ms	2 s	1 jour	3100 ans	
10^7	2.3 μ s	1 s	23 s	115 jours	3×10^6 ans	

En particulier, on réalisera ce qui suit : on note $T(n)$ le temps d'exécution en fonction de n la taille de l'entrée

- si $T(n) = n$, (complexité linéaire), si l'on multiplie par 10 la taille de l'entrée, alors le temps est multiplié par 10 :
$$T(10n) = 10T(n).$$
- si $T(n) = n^2$ (complexité quadratique), si l'on multiplie par 10 la taille de l'entrée, alors le temps est multiplié par 10^2 :
$$T(10n) = (10n)^2 = 10^2 T(n).$$
- si $T(n) = \log(n)$, (complexité logarithmique) en , si l'on multiplie par 10 la taille de l'entrée, on ajoute 1 au temps de calcul :
$$T(10n) = \log 10 + \log n = 1 + T(n).$$

2 Problèmes de terminaison, quelques premiers exemples

Dans tous les scripts, on note n_k la valeur prise par la variable `n` à la fin de la k-ième itération et on pose $n_0 = n$.

1. Script 1

```
n = 256
while n>0:
    n = n-2
```

   À chaque itération, on enlève 2 à `n`, donc au bout de k itérations, $n_k = n - 2k$. L'algo s'arrête dès que $n_k \leqslant 0$, c'est-à-dire pour $2k \geqslant n$, donc pour $k \geqslant \frac{n}{2}$. Le nombre d'itérations est donc $p = \lceil \frac{n}{2} \rceil$.

2. Script 2

```
n = 256
while n>0:
    n = n//2 # quotient de n par 2
```

   À chaque itération, `n` est au moins divisé par 2, donc au bout de k itérations, $n_k \leqslant \frac{n}{2^k}$. L'algo s'arrête lorsque $n_k < 1$ car dans ce cas $n_k = 0$ puisque c'est un entier naturel. Cette condition est réalisée dès que $\frac{n}{2^k} < 1$, c'est-à-dire pour $2^k > n$ et donc $k > \log_2(n)$. Le nombre p d'itérations est donc inférieur à $\lfloor \log_2(n) \rfloor + 1$.

   Calcul exact : de plus, pour $k \in [\![0, p-1]\!]$ on a $n_k = 2n_{k+1} + r_k$ avec $0 \leqslant r_k \leqslant 1$, donc $n_{k+1} = \frac{n_k - r_k}{2} \geqslant \frac{n-1}{2}$, d'où
$$n_{k+1} + 1 \geqslant \frac{n_k + 1}{2}.$$

Ainsi par récurrence immédiate, $n_p + 1 \geqslant \frac{n_0+1}{2^p}$. Comme $n_p \leqslant 0$, on en déduit que $1 \geqslant n_p + 1 \geqslant \frac{n+1}{2^p}$, donc $2^p \geqslant n + 1$ et ainsi $p \geqslant \log_2(n+1) > \log_2 n$. On a donc prouvé que

$$\log_2 n < p \leqslant \log_2 n + 1.$$

Ainsi $p = \lfloor \log_2(n) \rfloor + 1$.

Remarque : il faut être vigilant sur la condition d'arrêt : si on avait remplacé la condition `while n > 0` par `while n >= 0`, l'algorithme aurait bouclé indéfiniment, car à partir d'un certain rang $n_k = 0$ et donc $n_{k+1} = \lfloor \frac{n_k}{2} \rfloor = 0$.

3. Script 3 : Math Vs Info :

Considérons le script suivant :

```
n = 1
nbre_iterations = 0
while n>0:
    n = n/2
    nbre_iterations += 1
```

D'un point de vue théorique, cet algorithme ne se termine pas, car la variable n est divisé par 2 à chaque itération donc vaudra $\frac{1}{2^k}$ à la fin de la k-ième itération, en particulier la condition d'entrée dans la boucle $n_k > 0$ est toujours vérifiée. Pourtant, lorsqu'on l'exécute, il se termine en effectuant exactement 1075 itérations. Ceci provient du fait que la variable `n` est de type nombre flottant, codé sur 64 bits. En particulier, il existe un plus petit nombre flottant e strictement positif[1] . Comme la suite $1/2^k$ tend vers 0, à partir d'un certain rang, elle sera plus petite que ce nombre e et donc n_k vaudra 0, ainsi l'algorithme s'arrête.

3 Pourquoi et comment prouver un algorithme ? Notion d'invariant de boucle

Comment être sûr qu'un algorithme est juste ? On doit bien sûr le tester sur des valeurs quelconques mais aussi extrêmes. S'il y a une erreur, il est faux, mais si nos tests sont bons, il est «probablement juste». On peut arriver à des certitudes, et prouver au «sens mathématique» qu'un algorithme est juste. On utilise pour cela la notion d'**invariant de boucle** : c'est une propriété qui si elle est vraie en début de boucle, reste vraie en fin de boucle.

1. Le plus petit nombre flottant strictement positif (codé sur 64 bits) est 2^{-1022}. Mais il y a aussi des nombres flottants dits dénormalisés,codés avec un exposant nul qui permettent d'atteindre des nombres encore plus petits. Leur valeur est $0.mantisse * 2^{-decalage+1}$ au lieu de $1.mantisse * 2^{-decalage}$ pour un flottant normalisé. Pour avoir le plus petit flottant strictement positif dénormalisé, on prend donc une mantisse minimale non nulle $0000000\ldots01$ (52 bits) et comme l'exposant est nul, ce flottant a pour valeur $2^{-52} \times 2^{-1022} = 2^{-1074}$. Et effectivement, 2^{-1075} est déclaré nul par Python.

3.1 L'exemple modèle de l'exponentiation rapide

On se propose de découvrir cette notion à travers l'exercice suivant sur l'exponentiation rapide :

```
def expo_rapide(x,n):
    """Données: x un entier et n un entier naturel
       Résultat: l'entier x puissance n
    """
    p = 1 # p comme produit
    a = x # nombre que l'on exponentie
    e = n # e comme exposant
    while e != 0:
        if e % 2 == 1:
            p = p*a
            e = e - 1
        e = e // 2
        a = a * a
    return(p)
```

Traçons à la main l'exécution de `expo_rapide(2,10)` (k représente le numéro de l'itération) :

k	0	1	2	3	4
p	1	1	4	4	1024
a	2	4	16	256	256^2
e	10	5	2	1	0
$e \neq 0$	V	V	V	V	F
$p \times a^e$	1024	1024	1024	1024	1024

.

La valeur renvoyée est donc 1024 qui vaut 2^{10}. Après plusieurs tests, il semblerait que `expo_rapide(x,n)` renvoie x^n. Prouvons le !

La dernière ligne de ce tableau est inutile pour le calcul de `expo_rapide(2,10)` mais montre que la quantité $p \times a^e$ est invariante au cours des itérations...

Nous allons montrer que la propriété (IB) : « $x^n = p \times a^e$» est un **invariant de boucle**, c'est-à-dire :

- (IB) est vrai avant la première itération (initialisation)
- (IB) reste vrai après chaque passage dans la boucle (hérédité)

Dans ce cas, lorsqu'on sortira de la boucle, (IB) sera encore vrai, donc $pa^e = x^n$. De plus, en sortie de boucle, on a $e = 0$, ce qui donne $p = x^n$. Puisqu'on renvoie p, on renvoie bien x^n, ce qui achèvera la preuve.

Montrons donc que (IB) est un invariant de boucle :

- initialisation : avant la première itération $p \times a^e = 1 \times x^n = x^n$.

- supposons que (IB) est vrai avant une itération. On a donc $p \times a^e = x^n$. Notons p', a', e' les valeurs des variables p, a, e à la fin de l'itération.

  Premier cas : si e est impair, alors $p' = p \times a$, $e' = \frac{e-1}{2}$ et $a' = a^2$. Ainsi

  $$p' \times a'^{e'} = p \times a \times (a^2)^{\frac{e-1}{2}} = paa^{e-1} = pa^e = x^n.$$

  Deuxième cas, e est pair, alors $p' = p$, $a' = a^2$, et $e' = \frac{e}{2}$. Alors

  $$p' \times a'^{e'} = p \times (a^2)^{\frac{e}{2}} = pa^e = x^n.$$

  Dans les deux cas, (IB) est vérifié en fin d'itération.

3.2 Compléments

1. Et la méthode naturelle ?

   On peut bien sûr calculer x^n de la façon naturelle suivante : $x^n = x \times x \times \cdots \times x$. La fonction `expo_naif` suivante implémente cette méthode.

   ```
   def expo_naif(x,n):
       """Données: x un entier et n un entier naturel
          Résultat: l'entier x puissance n
       """
       p = 1
       for i in range(n):
           p = p * x
       return p
   ```

   Pour calculer `expo_naif(x,n)`, on exécute n itérations et donc n multiplications. C'est une complexité linéaire.

   Nous allons maintenant voir que l'on fait beaucoup mieux avec l'exponentiation rapide.

2. Terminaison et complexité de `expo_rapide`. :

   On note $e_0 = n$ la valeur de la variable `e` avant d'entrer dans la boucle «while». Pour $k \in \mathbb{N}^*$, on note e_k la valeur de la variable `e` à la fin de la k-ième itération de la boucle « while».

   Pour tout k, si e_k est impair, $e_{k+1} = \frac{e_k-1}{2}$ et si e_k est pair, $e_{k+1} = \frac{e_k}{2}$. Dans les deux cas, $e_{k+1} \leqslant \frac{e_k}{2}$.

   Autrement dit, à chaque itération, l'exposant est au moins divisé par 2, donc au bout de k itérations, on a $e_k \leqslant \frac{n}{2^k}$.

   L'algorithme se termine dès que l'entier e_k est nul, c'est-à-dire vérifie $e_k < 1$, ce qui est vrai lorsque $\frac{n}{2^k} < 1$. Cette condition équivaut à $2^k > n$ et donc à $k > \log_2(n)$.

   L'algorithme se termine donc et le nombre d'itérations est inférieur à $\log_2(n) + 1$.

   Comme il y a au pire deux multiplications par itérations, le nombre de multiplications est donc majoré par $2(\log_2(n) + 1)$. C'est donc une complexité logarithmique, c'est beaucoup mieux que la méthode naïve !

3. Pour compléter montrons que $p = \lfloor \log_2(n) \rfloor + 1$. Pour tout $k \in [\![0, p]\!]$, e_{k+1} est le quotient de la division euclidienne de a_k par 2. Il existe donc un entier $0 \leqslant r_k < 1$ (le reste) tel que $e_k = 2e_{k+1} + r_k$, ainsi

$$e_{k+1} = \frac{e_k - r_k}{2} \geqslant \frac{e_k - 1}{2}$$
$$\frac{e_{k+1} + 1}{2} \geqslant \frac{e_k + 1}{2}$$

Par récurrence immédiate,

$$\frac{e_p + 1}{2} \geqslant \frac{e_0 + 1}{2^p} = \frac{n+1}{2^p}$$

On en déduit que $2^p \geqslant n + 1$ et donc $p \geqslant \log_2(n+1) > \log_2(n)$. On a donc

$$\log_2(n) < p \leqslant \log_2(n) + 1$$

Ceci montre que

$$p = \lfloor \log_2(n) \rfloor + 1.$$

Remarques :

- cet entier p est le nombre de chiffres en base deux de l'entier n.
- Le «pire des cas» se produit lorsque à chaque itération $e\%2$ est égal à 1 (en effet dans ce cas à chaque itération, il y a 2 multiplications), c'est-à-dire lorsque les p bits de n sont tous égaux à 1. Dans ce cas, , $n = (11\ldots1)_2 = 2^p - 1$. On a donc montré que le pire des cas se produit lorsque n est un nombre de Mersenne $(2^p - 1)$, on effectue alors $2p = 2(\log_2(n) + 1)$ multiplications.
- A l'inverse le meilleur des cas se produit si tous les bits de n sauf le dernier sont nuls (dans ce cas une seule multiplication par itération), alors $n = (10\ldots0)_2 = 2^{p-1}$. Autrement dit, si $n = 2^{p-1}$ est une puissance de deux, on effectue $p - 1 + 2 = p + 1$ multiplications (lors de la dernière itération e vaut 1, et on fait donc deux multiplications).

4 Un nouvel exemple : recherche dans une liste triée

On considère une liste nommée `t` **triée** de nombres (ou de chaînes de caractères) dans l'ordre croissant. On voudrait écrire un algorithme qui détermine si une valeur donnée nommée `v` est un élément de cette liste.

4.1 Un premier algorithme naïf

On regarde si le premier élément de la liste est égal à `v`, sinon, on passe au deuxième... Écrivons le pseudo-code de cet algorithme que nous appelons `recherche_naive()`.

```
Algo: recherche_naive
Données: t une liste de nombres et v un nombre
Résultat: vrai si mot est dans liste et faux sinon
n = longueur de t
Pour i de 0 à n-1,
   si l[i] = v
      renvoie vrai
   fin si
Fin pour
Renvoie faux
```

Si on a beaucoup de chances, `v` est le premier élément de la liste et une seule comparaison sera nécessaire. À l'opposé, si `v` n'est pas dans la liste, on devra réaliser toutes les boucles, c'est-à-dire n comparaisons si n est la longueur de la liste. On dit alors que dans **le pire des cas**, la complexité temporelle est en $O(n)$, donc linéaire.

4.2 Un deuxième algorithme avec dichotomie

Il y a une situation concrète, dans laquelle nous sommes emmenés à chercher un mot dans une liste triée : la recherche d'un mot dans un dictionnaire. Il est clair que nous ne cherchons pas le mot comme précédemment. Nous pouvons utiliser une stratégie dite de dichotomie, «l'art de diviser en deux». Nous appelons `recherche_dichotomie` notre algorithme.

```
Algo: recherche_dichotomie
Données: t une liste triée et v une valeur
Résultat: vrai si v est un élément de t et faux sinon
n = longueur de t
a = 0  # indice de la borne de gauche
b = n-1 # indice de la borne de droite
Tant que b-a >= 0  # tant qu'il y a un élément dans [a,b]
    m = quotient de a+b par 2 # partie entière du milieu de a et b
    si t[m] = v
       renvoie vrai
    sinon si v < t[m]
       b = m - 1
    sinon
       a = m + 1
# arrivé ici, ici b-a < 0 donc il n'y a plus de mots d'indice dans [a,b], le mot est dor
Renvoie faux
```

Prenons par exemple $t = [2, 4, 7, 12, 15, 20, 25, 28, 31, 36]$ et traitons les exemples avec $v = 25$ puis $v = 24$.

a	0	5	5	6
b	9	9	6	6
$b - a \geqslant 0$	V	V	V	V
m	4	7	5	6
$t[m]$	15	28	20	25
$t[m] = 25$	F	F	F	V

a	0	5	5	6	6
b	9	9	6	6	5
$b - a \geqslant 0$	V	V	V	V	F
m	4	7	5	6	
$t[m]$	15	28	20	25	
$t[m] = 24$	F	F	F	F	

Prouvons tout d'abord que l'algorithme se termine bien. Il faut pour cela montrer qu'au bout d'une certain nombre d'itérations, la condition $b-a \geqslant 0$ devient fausse. L'idée est qu'après chaque itération la quantité $b-a$ est au moins divisée par 2 et donc on travaille avec une liste qui a deux fois moins d'éléments.

On note a_k et b_k les valeurs respectives des variables a et b à la sortie de la k-ième itération pour $k \in \mathbb{N}^*$. On pose aussi $a_0 = 0$ et $b_0 = n-1$, qui correspondent aux valeurs des variables a et b avant la première itération.

Comme l'entier m est la partie entière de $\frac{a+b}{2}$ le milieu de $[a,b]$, il y a deux cas :

- si m est le milieu de $[a,b]$, alors $b_{k+1} - a_{k+1} = \frac{b_k - a_k}{2} - 1$.
- si m n'est pas le milieu de $[a,b]$. Alors $m = \frac{a+b}{2} - \frac{1}{2}$. Il y a alors deux sous-cas :
  Si $a_{k+1} = m+1$ et $b_{k+1} = b_k$, alors $b_{k+1} - a_{k+1} = \frac{b_k - a_k}{2} - \frac{1}{2}$.
  Si $a_{k+1} = a_k$ et $b_{k+1} = m-1$ alors $b_{k+1} - a_{k+1} = \frac{b_k - a_k}{2} - \frac{3}{2}$.

Dans tous les cas, on a $b_{k+1} - a_{k+1} \leqslant \frac{b_k - a_k}{2} - \frac{1}{2}$, ce qui se réécrit

$$b_{k+1} - a_{k+1} + 1 \leqslant \frac{b_k - a_k + 1}{2}.$$

Cette relation est fondamentale. Si l'algorithme boucle indéfiniment, alors la condition $b - a \geqslant 0$ est toujours vraie, donc on a pour tout entier k, $b_k - a_k + 1 \geqslant 1 > 0$ et donc

$$\forall k \in \mathbb{N}^*,\ b_{k+1} - a_{k+1} + 1 \leqslant \frac{b_k - a_k + 1}{2} < b_k - a_k + 1.$$

En particulier la suite d'entiers $(b_k - a_k)_k$ est strictement décroissante et positive, ce qui est impossible. Ainsi la condition $b - a \geqslant 0$ deviendra fausse, ce qui assure que l'algorithme s'arrête.

On peut en plus établir la complexité. Dans le pire des cas, `v` est absent de la liste.

On obtient par récurrence immédiate,

$$\forall k \in \mathbb{N},\ b_k - a_k + 1 \leqslant \frac{b_0 - a_0 + 1}{2^k} = \frac{n}{2^k}.$$

Si N est le nombre d'itérations, on a $b_{N-1} - a_{N-1} \geqslant 0$, puis $b_{N-1} - a_{N-1} + 1 \geqslant 1$. On a donc

$$\frac{n}{2^{N-1}} \geqslant 1 \iff 2^{N-1} \leqslant n \iff N - 1 \leqslant \log_2(n) \iff N \leqslant \log_2(n) + 1.$$

Nous avons donc prouvé que dans le pire des cas, le nombre d'itérations est majoré par $\lceil \log_2 n \rceil + 1$. C'est donc une **complexité logarithmique**. C'est incroyablement mieux qu'une complexité linaire.

Remarques :

- la dichotomie n'est pas une méthode gadget, nous la retrouverons par exemple pour la résolution d'équations algébriques. C'est une stratégie de type «diviser pour régner».

- bien que l'algorithme ne soit pas très difficile à écrire, il faut être très vigilant à la condition d'arrêt. Quelques pièges se présentent : par exemple si l'on avait écrit «Tant que $b-a > 0$» au lieu de «$b-a \geqslant 0$», en reprenant la trace de l'exemple précédent avec $v = 25$, on aurait eu

a	0	5	5	6
b	9	9	6	6
$b-a>0$	V	V	V	F
m	4	7	5	
$t[m]$	15	28	20	
$t[m]=25$	F	F	F	

  et l'algorithme aurait renvoyé Faux alors que 25 est bien présent dans le tableau.

- Dans cet algorithme, on maintient **l'invariant** (de boucle) suivant : les valeurs du tableau d'indice strictement inférieur à a sont strictement inférieures à v et les valeurs du tableau d'indice strictement supérieur à b sont strictement supérieur à v, ce qui peut se schématiser par le tableau suivant :

indice		a		b	
valeur	$<$		v		$>$

Voici une possibilité de code Python.

```
def recherche_dicho(liste,mot):
    """Données: t est une liste triée de nombres ou de chaînes de caractères
                v est un nombre ou une chaîne de caractère
       Résutlat: si v est un élément de t, renvoie l'indice de v,
       sinon renvoie  "absent"
       Traitement: on utilise une méthode de dichotomie
    """
    a=0
    b=len(t)-1
    while b-a >= 0:  # tant qu'il y a un élement d'indice dans [a,b]
        m = (a+b)//2
        if v == t[m]:
            return(m)
        elif v < t[m]:
            b = m-1
        else:
            a = m + 1
    # arrivé ici b-a < 0 donc il n'y a plus de mots d'indice dans [a,b], le mot est donc
    return("absent")
```