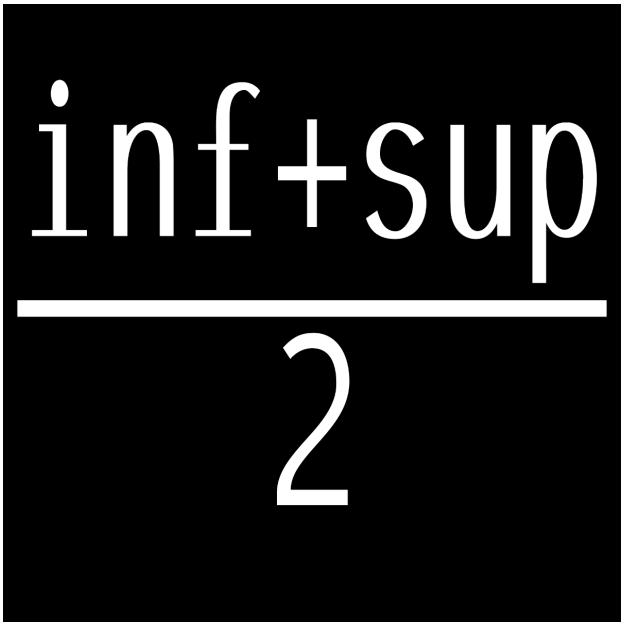
Pascal ORTIZ



La dichotomie

Version du 14 novembre 2024
Licence CC-BY

Table des matières

Présentation du tutoriel 2
Qu'est-ce qu'une recherche dichotomique ? 2
Implémentations d'une recherche dichotomique 3
Implémentation récursive de la recherche dichotomique 3
Implémentation itérative de la recherche dichotomique 5
Les deux fonctions Python de recherche dichotomique 6
Le nombre `bisect`$_right(L, x)vucommeunindice$ 6
Le nombre `bisect`$_left(L, x)vucommeunindice$ 8
`bisect(L, x)` vu comme un certain nombre d'éléments de `L` 8
Précisions sur les fonctions `bisect` 9
Test de présence d'un élément dans une liste croissante 10
Recherche dichotomique d'un objet non numérique 12
Efficacité de la recherche dichotomique en Python 12
Paramètres des fonctions de recherche dichotomique 13
Ordre défini par une clé 14
Fonctions d'insertion du module bisect 16

Présentation du tutoriel

Ce document présente en détail le fonctionnement du module standard `bisect` de recherche dichotomique dans une liste triée. Il commence par présenter le principe de la recherche dichotomique.

Si vous connaissez déjà le principe de la recherche dichotomique, vous pouvez directement aller à la section qui décrit les deux fonctions Python de recherche dichotomique.

Ce document ne traite pas des nombreuses situations où on est amené à utiliser une rechercher dichotomique, ce qui pourrait faire l'objet d'un document à part.

Qu'est-ce qu'une recherche dichotomique ?

Etant donné une liste `L` d'objets triés par ordre croissant et un objet `x`, l'algorithme de *recherche dichotomique* (*binary search* en anglais) est un algorithme « efficace » de recherche de la présence de `x` dans la liste `L`.

Etude d'un exemple

L'exemple suivant va illustrer le principe de la recherche dichotomique.

Soit `L` la liste croissante

```
12, 31, 46, 53, 81, 81, 82
```

et soit $x = 42$.

La dichotomie consiste à découper la liste `L` à peu près « au milieu » en deux listes de tailles à peu près égales à la moitié de la taille de `L`. D'où la liste `L1` :

```
12, 31, 46
```

et la liste `L2` :

```
53, 81, 81, 82
```

En comparant avec 46 (dernier terme de la première liste), on voit que `x` ne peut être que dans `L1` (car les listes sont croissantes).

On découpe à nouveau cette liste en deux sous-listes de longueur aussi proche que possible :

`[12]` et `[31 46]`

En comparant avec 12 (dernier terme de la première liste), on voit que `x` ne peut être que dans la deuxième liste `31, 46`

On découpe à nouveau cette liste en deux :

`[31]` et `[46]`

En comparant avec 31 (dernier terme de la première liste), on voit que `x` ne peut être que dans la deuxième liste [46]. Comme cette liste est de taille 1, il suffit de regarder si l'élément $x = 42$ est cet élément : ce n'est pas le cas, donc la liste **initiale** ne contient pas l'élément 42.

Principe de la recherche dichotomique

Comme illustré ci-dessus, le principe est le suivant : on découpe la liste en deux sous-listes de taille « environ » la moitié de la liste initiale et on identifie, à l'aide d'une seule comparaison, la sous-liste `M` susceptible de contenir `x`. Et on recommence la même recherche dans cette nouvelle sous-liste.

Plus précisément, si `n` est la taille de `L` le découpage en deux se fera suivant les longueurs définies par les sommes ci-dessous :

— si $\texttt{n} = 2\texttt{k}$ est pair alors on décomposera en $\texttt{n} = \texttt{k} + \texttt{k}$
— si $\texttt{n} = 2\texttt{k} + 1$ est pair alors on décomposera en $\texttt{n} = \texttt{k} + (\texttt{k} + 1)$

Noter que si les indices de `L` commencent à 0 alors, avec les notations $\texttt{n} = 2\texttt{k}$ ou $\texttt{n} = 2\texttt{k} + 1$ ci-dessus, la longueur de la première sous-liste `M` du découpage est `k` et donc l'indice `i` du dernier élément de `M` est $\texttt{k} - 1$. Or, que `n` soit pair ou pas, `k` est le quotient de la division entière de `n` par 2 et donc, $\texttt{i} = \texttt{n}//2 - 1$ où $//$ est la division entière.

Complexité de la recherche dichotomique

On peut démontrer que si la liste initiale est de taille n alors le nombre de recherches dans des sous-listes effectuées par la méthode de la dichotomie est au plus de l'ordre de $\log_2(n)$, le logarithme en base 2 de n. Autrement dit, si n est proche de 2^p alors environ p recherches sont effectuées. Par exemple, une liste de 1 million d'entiers (environ 2^{20}) nécessite une vingtaine d'appels. Une recherche dichotomique est donc très efficace.

Implémentations d'une recherche dichotomique

Le besoin d'effectuer une recherche dichotomique étant fréquent, de nombreux langages de programmation donnent la possibilité de faire directement une telle recherche sans la recoder soi-même. L'implémentation correcte d'une recherche dichotomique est considérée comme délicate (Donald Knuth parle de *tricky details*). On pourra consulter cette discussion pour prendre connaissances des principales difficultés. De nombreuses implémentations ont conservé pendant des années un bug d'overflow, voir l'exemple de Java.

Par ailleurs, pour être utile, la recherche dichotomique ne se contente pas d'examiner la présence ou l'absence d'un élément mais plutôt l'indice qu'aurait l'élément s'il était dans la liste. Et si l'élément est présent, il peut être répété ce qui pose la question du choix de l'indice.

Il existe deux types d'implémentation : l'**itérative** et la **récursive**. L'implémentation de la STL (C++), de Java et de Python sont itératives.

En Python, le module standard `bisect` (avec un seul *s*) donne accès à la recherche dichotomique. Ce module est écrit en C mais une implémentation est disponible en Python. Le terme de *bisect* fait allusion à la *bissection*.

Implémentation récursive de la recherche dichotomique

On se donne, d'une part, une liste croissante et non vide `L` formée de nombres et, d'autre part, un nombre `x`. La recherche dichotomique de `x` dans la liste L consiste à

— diviser au « milieu » la liste en deux sous-listes

— chercher dans la sous-liste appropriée.

L'implémentation de cette recherche conduit donc assez naturellement à une fonction récursive. En voici une implémentation en Python :

```
1  def dichotomie(L, debut, n, x):
2      if n==1:
3          return L[debut] == x
4      k= n//2
5      if x <= L[debut + k-1]:
6          return dichotomie(L, debut, k, x)
7      else:
8          return dichotomie(L, debut+k, n-k, x)
9
10 L=[1, 3, 4, 6, 6, 6, 9, 10]
11
12 print(L, end='\n----------\n')
13 for i in range(0,12):
14     print(f"{i:>2}", dichotomie(L, 0, len(L), i))
```

```
[1, 3, 4, 6, 6, 6, 9, 10]
----------
 0 False
 1 True
 2 False
 3 True
 4 True
 5 False
 6 True
 7 False
 8 False
 9 True
10 True
11 False
```

— Ligne 1 : `dichotomie` renvoie `True` si `x` est présent dans la sous-liste de la liste `L` commençant à l'indice `debut` et ayant `n` éléments. `L` est la liste initiale et `x` est l'élément cherché ; sinon, `dichotomie` renvoie `False`

— Lignes 6 et 8 : La fonction `dichotomie` est récursive : elle s'appelle sur l'une des sous-listes.

— Lignes 2-3 : si la liste est de taille 1 (le cas de base), il n'y a pas de sous-listes et, par simple examen (ligne 3) il est possible de répondre.

— Ligne 4 : il faut introduire cette variable `k` pour éviter le re-calcul aux lignes suivantes.

— Lignes 5-6 : si `x` n'est pas dans la sous-liste droite, la fonction `dichotomie` est rappelée pour chercher `x` dans la sous-liste gauche

— Lignes 7-8 : l'alternative.

Comme la taille de la liste diminue strictement à chaque appel, elle finit par valoir 1.

Voici les tests réalisés sur cette fonction pour « sécuriser » sa justesse :

```
def dichotomie(L, debut, n, x):
    if n==1:
        return L[debut] == x
    k= n//2
    if x <= L[debut + k-1]:
        return dichotomie(L, debut, k, x)
    else:
        return dichotomie(L, debut+k, n-k, x)

from random import randrange
ntest=50_000
for _ in range(ntest):
    N=randrange(1, 25)
    L=sorted([randrange(10, 21) for _ in range(N)])

    for z in range(0, 30):
        if dichotomie(L, 0, len(L), z)!=(z in L):
            print(L, z)
            raise ValueError("Erreur d'appartenance")

print("Tests passés")
```

```
Tests passés
```

Implémentation itérative de la recherche dichotomique

Voici une implémentation itérative (avec une boucle `while`) écrite en Python de la recherche dichotomique :

```
1  def dichotomie(L, x):
2      inf=0
3      sup=len(L)
4
5      while inf < sup:
6          mil = (inf + sup) // 2
7          if L[mil] < x:
8              inf = mil + 1
9          else:
10             sup = mil
11     return inf<len(L) and x==L[inf]
12
13 from random import randrange
14
15 ntest=50_000
16 for i in range(1, ntest+1):
17     if i%2000==0:
18         print(f"Test n°{i} ...")
19     N=randrange(1, 25)
```

```
20      L=sorted([randrange(10, 21) for _ in range(N)])
21      for z in range(0, 30):
22          inside = dichotomie(L, z)
23          if dichotomie(L, z)!=(z in L):
24              print(L, z)
25              raise ValueError("Erreur d'appartenance")
26
27  print("Tests passés")
```

— Lignes 2, 3 et 6 : `inf`, `sup` et `mil` désignent des indices.
— Lignes 2-3 : la liste courante est d'indices `inf` et `sup`.
— Ligne 6 : on calcule l'indice médian (noter la division entière).
— Lignes 7-8 : `x` est strictement du côté droit donc on monte le plancher
— Lignes 9-10 : `x` est du côté gauche donc on abaisse le plafond.
— Ligne 11 : à cette ligne, c'est que `inf` et `sup` sont égaux mais cet indice peut être invalide, par exemple, si `L = [42, 81]` et `x = 82`.

Le code ci-dessus est l'adaptation de l'implémentation proposée par le langage lui-même, cf. la fonction `bisect_left`.

Les deux fonctions Python de recherche dichotomique

Le module standard bisect dispose de deux fonctions de recherche dichotomique :

— `bisect_left`
— `bisect_right`

Voici les liens vers la documentation officielle du module bisect : en français, en anglais

On va appliquer ces fonctions à une liste croissante `L` et à un nombre `x` dont on cherche à connaître la présence dans `L`. Les deux fonctions en fait ne renvoient pas un booléen mais plutôt un nombre entier positif dont la définition n'est pas forcément immédiate à comprendre.

Ci-dessous, on décrit en détail ce que vaut `bisect_right(L, x)`. Cette valeur peut se définir de deux façons :

— en termes de nombre d'occurrences d'éléments dans `L`
— en termes d'indice d'une liste auxilaire `M`

Les deux façons vont être présentées ci-dessous.

Le calcul de `bisect_left(L, x)` est analogue et est présenté plus rapidement ensuite.

Le nombre `bisect_right(L, x)` vu comme un indice

Un appel `bisect_right(L, x)` renvoie l'indice le plus grand que `x` **aurait** dans la liste croissante obtenue en insérant `x` dans `L` (même si `x` y est déjà présent). Cette liste étendue et croissante sera appelée `M` ci-dessous. L'indice *le plus grand* se réfère au terme *le plus à droite*, d'où le nom de la fonction.

Par exemple, si `L` est la liste croissante suivante :

```
L = [10, 25, 25, 50, 81]
```

et si `x = 25`, la liste `M` obtenue en insérant `x` dans la liste `L` et tout en conservant l'ordre croissant est :

```
M = [10, 25, 25, 25, 50, 81]
```

On observe alors que l'indice de la valeur 25 la plus à droite de `M` est 3. C'est donc que `bisect_right(L, x)` vaut 3. Voyons-le avec du code :

```
from bisect import bisect_right

L =  [10, 25, 25, 50, 81]
x=25
print(L)
print(f"x = {x}")
print("Right :", f"i = {bisect_right(L, x)}")
```

```
[10, 25, 25, 50, 81]
x = 25
Right : i = 3
```

Voyons un autre exemple en gardant la même liste `L` mais avec cette fois, `x = 42`. La liste `M` obtenue en insérant `42` dans la liste `L` et tout en conservant l'ordre croissant est :

```
M = [10, 25, 25, 42, 50, 81]
```

On examine alors l'indice **le plus à droite** de `M` où la valeur de `M` soit 42 : on observe que 42 n'est présent qu'à l'indice 3 de `M` qui est donc la valeur de `bisect_right(L, 42)`. Voyons-le avec le code :

```
from bisect import bisect_right

L =  [10, 25, 25, 50, 81]
x=42
print(L)
print(f"x = {x}")
print("Right :", f"i = {bisect_right(L, x)}")
```

```
[10, 25, 25, 50, 81]
x = 42
Right : i = 3
```

On voit que l'intérêt de cette définition est de permettre de bien saisir où l'élément `x` se positionne dans la liste `L`. Toutefois l'indice obtenu **n'est pas forcément un indice valide** de `L` ce qui se produit lorsque la fonction renvoie la longueur `n` de la liste `L` autrement dit si $x \geq L[n-1]$. Voici un exemple de cette situation :

```
from bisect import bisect_right

L =  [10, 25, 25, 50, 81]
x=99
print(L)
```

```
print(f"x = {x}")
print("Right :", f"i = {bisect_right(L, x)}")
print("len(L) =", len(L))
```

```
[10, 25, 25, 50, 81]
x = 99
Right : i = 5
len(L) = 5
```

On voit que l'indice renvoyé vaut la longueur de la liste L et est donc un indice invalide de la liste.

Le nombre `bisect_left(L, x)` vu comme un indice

La fonction `bisect_left` est analogue à `bisect_right`. Si `M` est la liste obtenue en insérant `x` dans `L` et en respectant l'ordre croissant, alors l'appel `bisect_left(L, x)` renvoie le plus petit indice (donc le plus à gauche d'où le nom de la fonction) que `x` **aurait** dans la liste `M`.

Par exemple, si `L` est la liste croissante suivante :

```
L = [10, 25, 25, 50, 81]
```

alors si `x = 25`, la liste `M` obtenue en insérant `x` dans la liste `L` et tout en conservant l'ordre croissant est :

```
M = [10, 25, 25, 25, 50, 81]
```

On observe alors que l'indice de la valeur la plus à **gauche** de `M` qui vaille 25, est 1. C'est donc que `bisect_left(L, x)` vaut 1. Voyons-le avec du code :

```
from bisect import bisect_right

L =  [10, 25, 25, 50, 81]
x=25
print(L)
print(f"x = {x}")
print("Right :", f"i = {bisect_right(L, x)}")
```

```
[10, 25, 25, 50, 81]
x = 25
Left : i = 1
```

`bisect(L, x)` vu comme un certain nombre d'éléments de `L`

On peut aussi définir `bisect_left(L, x)` comme le nombre d'occurrences d'éléments `y` de `L` tels que $y < x$. De même, on peut définir `bisect_right(L, x)` comme le nombre d'occurrences d'éléments `y` de `L` tels que $y \leq x$.

Par exemple, avec la liste `L`

```
L = [10, 25, 25, 50, 81]
```

le tableau suivant

x	$y < x$	$y \leq x$	bisect_left	bisect_right
8	aucun	aucun	0	0
25	10	10, 25, 25	1	3
42	10, 25, 25	10, 25, 25	3	3

explique comment calculer les valeurs pour x parmi 8, 25 ou 42.

Cette définition (qui n'est pas mentionnée dans la documentation) a l'avantage de la simplicité et de l'homogénéité mais elle ne montre pas le placement potentiel de x dans la liste.

Alias

Le module bisect permet d'utiliser la fonction `bisect_right` sous le nom plus simple de `bisect` :

```
from bisect import bisect_right, bisect

L = [10, 25, 25, 50, 81]
x=42

print(bisect_right(L, x), bisect(L, x))
```

```
3 3
```

Reste à savoir si la fonction `bisect_right` était plus approriée que `bisect_left` à recevoir un nom simplifié.

Précisions sur les fonctions `bisect`

Confusions possibles

— Pour la première définition de `bisect_left(L, x)` et `bisect_right(L, x)`, on a introduit la liste `M` obtenue en insérant `x` dans `L` tout en maintenant la liste croissante. Attention ! l'introduction de cette liste `M` ne signifie pas que les fonctions de recherche dichotomique créent une **nouvelle** liste `M`, ce qui ralentirait le code. En réalité, on **imagine** seulement que cette liste `M` est construite.

— La liste `L` est complètement préservée par l'opération de recherche dichotomique de `x`, en particulier, la valeur `x` n'est pas insérée dans `L`.

`bisect_left` et `bisect_right` : analogies et différences

Ci-dessous, on illustrera avec la liste `L` :

```
L = [10, 25, 25, 50, 81]
```

déjà utilisée.

— Les deux fonctions peuvent renvoyer l'indice `len(L)`, lequel est invalide pour `L`. C'est le cas, pour la liste `L` ci-dessus et par exemple `x = 100`.

— Si la valeur cherchée **n'est pas** dans la liste alors les deux fonctions renvoient la même valeur. C'est le cas avec la liste ci-dessus et `x = 42`

— Si la valeur cherchée est présente **dans** la liste, les deux fonctions renvoient des valeurs forcément différentes. Avec l'exemple ci-dessus, on a `bisect_left(L, 25) = 1` et `bisect_right(L, 25) = 3`

— Si un élément recherché `x` **est présent** dans la liste `L` alors `bisect_left` renvoie un indice où l'élément est présent dans la liste (ce n'est pas le cas de `bisect_right`).

- Posons `i = bisect_left(L, x)`. Alors, si `n` $=$ `len(L)` on a
  - si `i` $=0$ on a `x` $\leq$ `L[0]`
  - si `i` $=$ `n` on a `L[n-1]` $<$ `x`
  - sinon, on a `L[i-1]` $<$ `x` $\leq$ `L[i]`
- Posons `j = bisect_right(L, x)`. Alors, si `n` $=$ `len(L)` on a
  - si `j` $=0$ on a `x` $<$ `L[0]`
  - si `j` $=$ `n` on a `L[n-1]` $\leq$ `x`
  - sinon, on a `L[j-1]` $\leq$ `x` $<$ `L[j]`
- Les fonctions de recherche dichotomique fonctionnent pour une liste vide (et renvoient alors 0).

Test de présence d'un élément dans une liste croissante

Le module bisect ne dispose pas d'une fonction booléenne qui indiquerait la présence d'un élément `x` dans une liste croissante `L`. Toutefois, la fonction `bisect_left` donne aisément la possibilité de tester la présence de `x` dans `L` :

```
1  from bisect import bisect_left
2
3  L = [10, 25, 25, 50, 81]
4  print(L)
5
6  x=42
7  i = bisect_left(L, x)
8  print(f"x = {x} : {L[i]==x}" )
9
10 x=50
11 i = bisect_left(L, x)
12 print(f"x = {x} : {L[i]==x}" )
```

```
[10, 25, 25, 50, 81]
x = 42 : False
x = 50 : True
```

Il suffit de calculer l'indice `i = bisect_left(L,x)` et de vérifier qu'à cet indice l'élément de `L` vaut bien `x`, ie que `L[i] == x`, cf. ligne 7 et 11. Cela ne fonctionne pas si on remplace la fonction `bisect_left` par la fonction `bisect_right` :

```
from bisect import bisect_right

L = [10, 25, 25, 50, 81]
print(L)

x=50
i = bisect_right(L, x)
print(f"x = {x} : {L[i]==x}" )
```

```
[10, 25, 25, 50, 81]
x = 50 : False
```

Plus petit ou plus grand indice de présence

La fonction `leftmost(L, x)` ci-dessous renvoie `None` si `x` n'est pas dans la liste croissante `L` et sinon, renvoie l'indice le plus à gauche (c'est-à-dire le plus petit) où il est présent. C'est analogue pour `rightmost(L, x)`. Voici les fonctions et un exemple :

```
from bisect import bisect_left, bisect_right

def leftmost(L, x):
    i=bisect_left(L, x)
    if i!=len(L) and L[i]==x:
        return i
    return None

def rightmost(L, x):
    i=bisect_right(L, x)
    if i>0 and L[i-1]==x:
        return i-1
    return None

L = [10, 10, 10, 25, 25, 50, 81, 81]

print(L)

for x in [9, 10, 14, 24, 25, 50, 80, 81, 82]:
    print(x, leftmost(L, x))
print("---------------")

for x in [9, 10, 14, 24, 25, 50, 80, 81, 82]:
    print(x, rightmost(L, x))
```

```
[10, 10, 10, 25, 25, 50, 81, 81]
9 None
10 0
14 None
24 None
25 3
50 5
80 None
81 6
82 None
---------------
9 None
10 2
14 None
24 None
25 4
```

```
50 5
80 None
81 7
82 None
```

Plutôt que de renvoyer `None`, le code des fonctions pourrait être plus pythonnique en levant une exception de type `ValueError` (comme la méthode `list.index` par exemple).

Recherche dichotomique d'un objet non numérique

Il est possible d'effectuer des recherches avec le module bisect pour toute séquence ordonnée croissante, pas seulement une liste de nombres.

Par exemple, on peut trouver positionner une lettre dans une suite croissante de lettres :

```
from bisect import bisect

print(bisect("aeiouy","p"))
```

qui affiche

```
4
```

autrement dit la lettre p se place juste après les quatre premières lettres.

On peut aussi chercher l'indice de placement d'une chaîne de caractères dans une liste triée de chaînes :

```
from bisect import bisect

L = ["rosa", "rosae", "rosam", "rosarum", "rosas", "rosis"]
print(bisect(L, "roses"))
```

```
5
```

Efficacité de la recherche dichotomique en Python

Les fonctions de recherche dichotomique du module `bisect` sont très efficaces. Elles permettent d'encadrer un entier dans une liste croissante de 10 millions d'éléments en quelques millisecondes :

```
1 from random import randrange
2 from time import perf_counter
3 from bisect import bisect_left
4
5 N=10**7
6 M=10**9+1
7 L=sorted([randrange(M) for _ in range(N)])
8
9 x=randrange(M)
10 print(x)
11
```

```
12 begin_perf = perf_counter()
13
14 i=bisect_left(L, x)
15 print(L[i]==x)
16
17 delta = perf_counter() - begin_perf
18
19 print(f"Temps d'exécution : {delta:.2f}s")
```

```
20 228945091
21 False
22 Temps d'exécution : 0.00s
```

— Ligne 7 : on génère une liste triée L de 10 millions d'entiers entre 0 et un milliard
— Ligne 9 : on recherche un entier aléatoire x dans la liste L
— Ligne 22 : le temps de recherche est négligeable.

La fonction `bisect` de CPython est implémentée dans le module `_bisectmodule` écrit en langage C.

Paramètres des fonctions de recherche dichotomique

La fonction `bisect_left` dispose de 4 paramètres dont les deux derniers sont optionnels :

— le paramètre `a` (pour *array*) qui est la séquence, supposée triée dans l'ordre croissant
— le paramètre `x` qui est l'élément à rechercher dans la séquence `a`
— le paramètre optionnel `lo` (pour *lower*) qui par défaut vaut l'indice 0
— le paramètre optionnel `hi` (pour *higher*) qui par défaut vaut `len(a)`.

La sémantique de l'usage de `lo` et `hi` est définie de manière assez vague dans la documentation officielle. Il semblerait que $0 \leq$ `lo` $\leq$ `hi` $\leq$ `len(a)` et que l'appel `bisect(L, x, lo, hi)` soit équivalent à `lo` + `bisect(L[lo : hi], x)`.
Voici un exemple d'utilisation de ces paramètres optionnels :

```
from bisect import bisect

L = [12, 21, 21, 26, 31, 33, 35, 37]
lo=3
hi=5
M = L[lo:hi]
print(M)

for x in [10, 22, 27, 32, 34, 39]:
    print(x, bisect(L, x, lo=lo, hi=hi))
    print(x, lo+bisect(M, x))
    print("----------------")
```

```
[26, 31]
10 3
10 3
```

```
----------------
22 3
22 3
----------------
27 4
27 4
----------------
32 5
32 5
----------------
34 5
34 5
----------------
39 5
39 5
----------------
```

La situation est analogue pour la fonction `bisect_right`.

Ordre défini par une clé

Pour illustrer l'usage d'une clé avec une fonction `bisect`, soit une liste `L` telle que chaque élément est une paire constituée ainsi :

— le nom d'une personne
— l'âge de la personne

par exemple cette liste de 6 personnes :

```
L = [
 ('jeanne', 15),
 ('jean', 21),
 ('marc', 24),
 ('paul', 38),
 ('philippe', 42),
 ('mireille', 57)]
```

On suppose la liste triée suivant les âges (observer que c'est bien le cas ci-dessus) et on veut faire une recherche dichotomique dans `L` en fournissant un âge. Depuis la version 3.10 de Python, voici comment cette opération est réalisable directement :

```
from bisect import bisect_left

L = [
 ('jeanne', 15),
 ('jean', 21),
 ('marc', 24),
 ('paul', 38),
 ('philippe', 42),
 ('mireille', 57)]
```

```
def age(p):
    return p[1]

elt = 32

print(bisect_left(L, elt, key=age))
```

```
3
```

Le retour signifie qu'un individu de 32 ans serait placé en 3e position de la liste.

Le module `bisect` permet de réaliser directement cette recherche avec les fonctions `bisect_left` ou `bisect_right` en utilisant une clé de comparaison (dans l'exemple c'est `age`), à l'instar de ce qui se fait par exemple avec la fonction sorted. Cela signifie que pour trouver l'indice, la fonction `bisect_left` va comparer la valeur donnée (dans l'exemple, c'est 32) au retour de la fonction-clé sur des éléments de `L`.

L'appel doit obligatoirement mentionner le mot-clé `key` car la signature de la fonction est

```
bisect_left(a, x, lo=0, hi=len(a), *, key=None)
```

Noter qu'assez curieusement, l'élément à rechercher n'est pas un élément ayant la forme d'un élément de la liste mais plutôt un élément qui est de la forme du retour de la fonction-clé. Autrement, dit, dans l'exemple ci-dessus, un appel tel que `bisect_left(L, ("zoé", 18), key=age)` léverait un exception de type `TypeError`.

Illustration

Le tableau ci-dessous définit les catégories de participants en athlétisme en fonction de l'âge, donné en années (d'après Wikipédia) :

Catégorie	**Début (années)**	**Fin (années)**
Eveil		9
Poussin	10	11
Benjamin	12	13
Minime	14	15
Cadet	16	17
Junior	18	19
Espoir	20	22
Sénior	23	39
Vétéran	40	

Écrivons un programme utilisant `bisect` et une clé qui à partir d'un âge détermine la catégorie sportive correspondante.

Pour cela, chaque catégorie est définie par un tuple dont la première valeur est l'âge de début de la catégorie et la deuxième est le nom de la catégorie :

```
cat =[(0, "éveil"),
      (10,"poussin"),
      (12,"benjamin"),
      (14,"minime"),
      (16,"cadet"),
```

```
     (18, "junior"),
     (20, "espoir"),
     (23, "sénior"),
     (40, "vétéran")
     ]
```

Etant donné un âge, si on fait une recherche dichotomique de cet âge dans la liste ci-dessus, on trouvera la catégorie :

```
from bisect import bisect

cat =[(0, "éveil"),
     (10,"poussin"),
     (12,"benjamin"),
     (14,"minime"),
     (16,"cadet"),
     (18, "junior"),
     (20, "espoir"),
     (23, "sénior"),
     (40, "vétéran")
     ]

def get_age(t):
    return t[0]

age = 19

i = bisect(cat, age, key=get_age)
print(age, cat[i-1][1])
```

```
19 junior
```

Fonctions d'insertion du module bisect

Le module bisect propose deux fonctions d'insertion d'un élément `x` dans une liste `L` croissante. Regardons le cas de la fonction `insort_left`, le cas de `insort_right` étant similaire.

Un appel `insort_left(L, x)` insère l'élément `x` dans la liste `L` et en conservant l'ordre croissant ; cette liste n'est autre que la liste `M` introduite dans un paragraphe précédent pour définir `bisect_left(L, x)` sauf qu'ici, `L` est modifiée et non recrée.

Voici un exemple d'utilisation de cette fonction :

```
from bisect import insort_left

L =  [10, 25, 25, 50, 81]
print(L)
print("--------------------")
for x in [25, 42]:
    M=L[:]
    print(x)
```

```
    insort_left(M, x)
    print(M)
    print("-------------------")
```

```
[10, 25, 25, 50, 81]
-------------------
25
[10, 25, 25, 25, 50, 81]
-------------------
42
[10, 25, 25, 42, 50, 81]
-------------------
```

Ces fonctions ont un intérêt très restreint. En effet, l'avantage d'une recherche dichotomique est sa faible complexité en `log(n)`. Comme la fonction `insort_left` insère un élément dans la liste, sa complexité devient linéaire. En outre, `insort_left(L, x)` est équivalent à calculer `i = bisect_left(L, x)` et à insérer `x` à l'indice `i` avec la méthode `insert`. Illustration :

```
from bisect import bisect_left, insort_left

L =  [10, 25, 25, 50, 81]
M=L[:]

x=25
print(f"L = {L}")
print(f"x = {x}")

print("-----------------")

insort_left(L, x)
print(L)


i = bisect_left(M, x)
M.insert(i, x)
print(M)
```

```
L = [10, 25, 25, 50, 81]
x = 25
-----------------
[10, 25, 25, 25, 50, 81]
[10, 25, 25, 25, 50, 81]
```