Recherche dans une liste

Résumé

Un contact dans un répertoire de mille noms, un mot dans un dictionnaire de cinquante mille entrées, un fichier parmi des millions : le problème de la **recherche** est omniprésent en informatique. Comment retrouver efficacement un élément dans une collection ? La méthode la plus simple (parcourir un par un) fonctionne toujours, mais elle peut être très lente. La **recherche dichotomique**, qui divise l'espace de recherche par deux à chaque étape, est spectaculairement plus rapide, à condition que les données soient triées. Comprendre ces deux stratégies, c'est comprendre pourquoi certains programmes sont rapides et d'autres lents.

Prérequis : Les listes, Recherche d'extremum.

① Mémo

Recherche séquentielle

Principe Parcourir la liste du début à la fin, comparer chaque élément à la valeur cherchée.
Résultat Renvoie l'indice de la première occurrence, ou -1 si l'élément est absent.
Complexité $O(n)$ dans le pire cas.
Avantage Fonctionne sur n'importe quelle liste, triée ou non.

Variantes du parcours séquentiel

Compter les occurrences Parcourir toute la liste en incrémentant un compteur à chaque correspondance. Complexité $O(n)$, même si l'élément est absent.
Toutes les positions Même parcours, en ajoutant chaque indice trouvé à une liste de résultats au lieu de s'arrêter à la première occurrence.

Recherche dichotomique

Précondition La liste doit être **triée**.
Principe Comparer la valeur cherchée à l'élément du milieu, puis réduire l'intervalle de moitié.
Complexité $O(\log_2 n)$: pour 10^6 éléments, au plus 20 comparaisons.
Invariant Si la valeur existe, elle se trouve dans l'intervalle `[gauche, droite]`.

⚠ Pièges fréquents

- Placer le `return -1` **dans** la boucle (dans un `else`) : la fonction s'arrête dès le premier élément qui ne correspond pas.
- Appliquer la dichotomie sur une liste non triée : le résultat est faux sans message d'erreur.
- Confondre `milieu = (gauche + droite) // 2` (correct) avec `milieu = (gauche + droite) / 2` (donne un flottant).

⚠ Erreurs classiques

Code erroné	Code correct	Explication
`if L[i] != x: return -1` (dans la boucle)	Placer `return -1` **après** la boucle	Le `return -1` exécuté dès que `L[i] != x` interrompt la boucle à la première case différente de `x` : la fonction conclut à l'absence sans examiner le reste de la liste.
`milieu = (g + d) / 2`	`milieu = (g + d) // 2`	La division `/` renvoie un flottant : `L[2.5]` provoque un `TypeError`. Il faut la division entière `//`.
`while g < d:`	`while g <= d:`	Avec `<` strict, le cas `g == d` (un seul élément restant) n'est pas examiné : on peut manquer la valeur cherchée.
Dichotomie sur une liste non triée	Trier d'abord, ou utiliser la recherche séquentielle	La dichotomie suppose la liste triée. Sur une liste non triée, elle peut conclure à tort qu'un élément est absent (aucune erreur affichée).

② Exemples

Programme 1 · Recherche séquentielle

```
def recherche(L, x):
    for i in range(len(L)):
        if L[i] == x:
            return i     # trouvé : indice de la première occurrence
    return -1            # absent

print(recherche([4, 8, 2, 6, 3, 8], 2))   # 2
print(recherche([4, 8, 2, 6, 3, 8], 5))   # -1
```

Programme 2 · Recherche dichotomique

```
def dichotomie(L, x):
    gauche = 0
    droite = len(L) - 1
    while gauche <= droite:
        milieu = (gauche + droite) // 2
        if L[milieu] == x:
            return milieu
        elif L[milieu] < x:
            gauche = milieu + 1
        else:
            droite = milieu - 1
    return -1
```

Trace : recherche de **4** dans L = [1, 3, 5, 7, 9, 11, 13].

Étape	gauche	droite	milieu	Comparaison	Conséquence
1	0	6	3	L[3] = 7 > 4	on cherche à gauche
2	0	2	1	L[1] = 3 < 4	on cherche à droite
3	2	2	2	L[2] = 5 > 4	on cherche à gauche
4	2	1		gauche > droite	**4 est absent**

À chaque étape, l'intervalle est divisé par deux : quatre comparaisons suffisent pour conclure, contre sept pour une recherche séquentielle. À l'inverse, chercher **7** dans la même liste se termine dès l'étape 1, puisque L[3] = 7.

Programme 3 · Nombre d'occurrences

```
def nb_occurrences(L, x):
    compteur = 0
    for elem in L:
        if elem == x:
            compteur += 1
    return compteur
print(nb_occurrences([3, 1, 4, 1, 5, 1], 1))   # 3
```

③ Exercices

Exercice 1 *Toutes les occurrences* Écrire une fonction `toutes_occurrences(L, x)` qui renvoie la liste des indices de toutes les occurrences de `x` dans `L`. Quelle est sa complexité ?

Exercice 2 *Nombre d'occurrences* Écrire une fonction `nb_occurrences(L, x)` qui renvoie le nombre d'apparitions de `x` dans `L`, sans utiliser `.count()`.

Exercice 3 *Trace de la dichotomie* On considère `L = [2, 5, 8, 12, 16, 23, 38, 42, 55, 67]`.

Donner la trace complète (`gauche`, `droite`, `milieu` à chaque étape) pour :

1. la recherche de `23` ;
2. la recherche de `10` (absent).

Exercice 4 *Comparaison des méthodes*

1. Écrire une fonction `recherche_compteur(L, x)` qui effectue une recherche séquentielle et renvoie le tuple `(indice, nb_comparaisons)`.
2. Écrire une fonction `dichotomie_compteur(L, x)` qui effectue une recherche dichotomique et renvoie le tuple `(indice, nb_comparaisons)`.
3. Pour $L = [0, 1, 2, \ldots, 999]$ et $x = -1$ (absent), combien de comparaisons effectue chaque méthode ? Vérifier avec le code.

Solutions des exercices

Corrigé de l'exercice 1.

```
def toutes_occurrences(L, x):
    indices = []
    for i in range(len(L)):
        if L[i] == x:
            indices.append(i)
    return indices
```

Vérification : `toutes_occurrences([3, 1, 4, 1, 5, 1], 1)` = `[1, 3, 5]`.

La complexité est $O(n)$: on parcourt toujours toute la liste car on cherche *toutes* les occurrences.

Corrigé de l'exercice 2.

```
def nb_occurrences(L, x):
    compteur = 0
    for elem in L:
        if elem == x:
            compteur += 1
    return compteur
```

Vérification : `nb_occurrences([3, 1, 4, 1, 5, 1], 1)` renvoie 3.

Corrigé de l'exercice 3. Recherche de 23 :

1. `gauche=0`, `droite=9`, `milieu=4`, `L[4]=16 < 23`, donc `gauche=5`.
2. `gauche=5`, `droite=9`, `milieu=7`, `L[7]=42 > 23`, donc `droite=6`.
3. `gauche=5`, `droite=6`, `milieu=5`, `L[5]=23`, trouvé à l'indice 5.

Trois comparaisons suffisent.

Recherche de 10 :

1. `gauche=0`, `droite=9`, `milieu=4`, `L[4]=16 > 10`, donc `droite=3`.
2. `gauche=0`, `droite=3`, `milieu=1`, `L[1]=5 < 10`, donc `gauche=2`.
3. `gauche=2`, `droite=3`, `milieu=2`, `L[2]=8 < 10`, donc `gauche=3`.
4. `gauche=3`, `droite=3`, `milieu=3`, `L[3]=12 > 10`, donc `droite=2`.
5. `gauche=3 > droite=2` : 10 est absent.

Corrigé de l'exercice 4.

```python
def recherche_compteur(L, x):
    for i in range(len(L)):
        if L[i] == x:
            return i, i + 1
    return -1, len(L)

def dichotomie_compteur(L, x):
    gauche, droite, nb = 0, len(L) - 1, 0
    while gauche <= droite:
        nb += 1
        milieu = (gauche + droite) // 2
        if L[milieu] == x:
            return milieu, nb
        elif L[milieu] < x:
            gauche = milieu + 1
        else:
            droite = milieu - 1
    return -1, nb
```

Résultat pour $n = 1000$**,** $x = -1$ **:** séquentielle $= 1000$ comparaisons, dichotomie $= 9$ comparaisons pour $x = -1$ (le pire cas est $\lceil \log_2(1000) \rceil = 10$, atteint pour d'autres valeurs).