Algorithmes de base

① MÉMO

Recherche du maximum et du minimum

Principe Parcourir toute la liste en conservant le plus grand (ou le plus petit) élément rencontré.
Initialisation Toujours avec le premier élément de la liste, jamais avec 0 (qui serait faux pour une liste de négatifs).
Complexité $\mathrm{O}(n)$ (un seul parcours de la liste).
Variante Pour le minimum, il suffit de remplacer la comparaison $>$ par $<$.

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 trouvée, ou -1 si l'élément est absent.
Complexité $\mathrm{O}(n)$ dans le pire cas (élément absent ou en dernière position).
Avantage Fonctionne sur n'importe quelle liste, triée ou non.

Recherche dichotomique

Prérequis La liste doit être triée.
Principe Comparer la valeur cherchée à l'élément du milieu, puis réduire l'intervalle de recherche de moitié.
Complexité $\mathrm{O}(\log_2 n)$. Pour une liste de 10^6 éléments, au plus 20 comparaisons au lieu de 10^6.
Invariant La valeur cherchée, si elle existe, se trouve toujours dans l'intervalle `[gauche, droite]`.

Algorithmes de tri

Tri par sélection À chaque étape, on sélectionne le minimum de la partie non triée et on le place au début. Complexité : $\mathrm{O}(n^2)$ dans tous les cas.
Tri par insertion On insère chaque élément à sa place dans la partie déjà triée. Complexité : $\mathrm{O}(n^2)$ dans le pire cas, $\mathrm{O}(n)$ si la liste est presque triée.

Comptage et occurrences

Principe Parcourir la liste en incrémentant un compteur à chaque correspondance.
Complexité $\mathrm{O}(n)$ (parcours complet, même si l'élément est absent).

② EXEMPLES

Programme 1 · Recherche du maximum

```
def maximum(L):
    maxi = L[0]
    for x in L:
        if x > maxi:
            maxi = x
```

```
    return maxi

# maximum([3, -1, 7, 2]) renvoie 7
```

Programme 2 · Maximum avec indice

```
def indice_maximum(L):
    """Renvoie l'indice du maximum dans L."""
    i_max = 0
    for i in range(1, len(L)):
        if L[i] > L[i_max]:
            i_max = i
    return i_max

# indice_maximum([3, 7, 2, 9, 1]) renvoie 3
```

Programme 3 · Recherche séquentielle

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

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

Programme 4 · 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 : on cherche la valeur **4** dans $\mathrm{L} = [1, 3, 5, 7, 9, 11, 13]$.

Étape	gauche	droite	milieu	Comparaison	Conséquence
1	0	6	3	$\mathrm{L}[3] = 7 > 4$	on réduit à la moitié gauche
2	0	2	1	$\mathrm{L}[1] = 3 < 4$	on réduit à la moitié droite
3	2	2	2	$\mathrm{L}[2] = 5 > 4$	on réduit à la moitié gauche
4	2	1		gauche > droite	**4 est absent de la liste**

À chaque étape, l'intervalle de recherche est divisé par deux. Ici, quatre comparaisons suffisent pour conclure que 4 n'est pas dans la liste (contre sept pour une recherche séquentielle).

Contre-exemple : chercher **7** dans la même liste. Dès l'étape 1, $L[3] = 7$, donc 7 est trouvé en une seule comparaison.

Programme 5 · Tri par sélection

```
def tri_selection(L):
    """Trie L en place par sélection."""
    n = len(L)
    for i in range(n - 1):
        i_min = i
        for j in range(i + 1, n):
            if L[j] < L[i_min]:
                i_min = j
        L[i], L[i_min] = L[i_min], L[i]  # échange
```

Programme 6 · Tri par insertion

```
def tri_insertion(L):
    """Trie L en place par insertion."""
    for i in range(1, len(L)):
        cle = L[i]
        j = i - 1
        while j >= 0 and L[j] > cle:
            L[j + 1] = L[j]
            j -= 1
        L[j + 1] = cle
```

Programme 7 · Nombre d'occurrences

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

# nb_occurrences([3, 1, 4, 1, 5, 1], 1) renvoie 3
```

3 EXERCICES

Exercice 1 *Recherche séquentielle*

1. Écrire une fonction `toutes_occurrences(L, x)` qui renvoie la liste des indices de toutes les occurrences de `x` dans `L`.
2. Combien de comparaisons effectue cette fonction dans le pire cas pour une liste de taille n ?

Exercice 2 *Dichotomie : trace* On considère la liste triée `L = [2, 5, 8, 12, 16, 23, 38, 42, 55, 67]`.

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

1. la recherche de `23` ;

2. la recherche de `10`.

Exercice 3 ***Tri par sélection : trace*** Donner l'état de la liste `[5, 3, 8, 1, 4]` après chaque passage du tri par sélection.

Exercice 4 ***Deux valeurs extrêmes*** Écrire une fonction `deux_plus_grands(L)` qui renvoie les deux plus grands éléments d'une liste (de taille $\geqslant 2$) en un seul parcours, sans trier la liste.

SOLUTIONS DES EXERCICES

Corrigé de l'exercice 1.

```python
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]`.

Dans le pire cas, on effectue n comparaisons (on parcourt toute la liste quelle que soit la situation, car on cherche *toutes* les occurrences). La complexité est $O(n)$.

Corrigé de l'exercice 2. **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 pour une liste de taille 10. On vérifie : $\lceil \log_2(10) \rceil = 4$, c'est bien inférieur.

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`, l'élément est absent, renvoie -1.

Corrigé de l'exercice 3.

1. Recherche du minimum dans `[5, 3, 8, 1, 4]` : c'est `1` (indice 3). Échange avec l'indice 0 : `[1, 3, 8, 5, 4]`.
2. Recherche du minimum dans `[3, 8, 5, 4]` : c'est `3` (indice 1), déjà en place : `[1, 3, 8, 5, 4]`.
3. Recherche du minimum dans `[8, 5, 4]` : c'est `4` (indice 4). Échange avec l'indice 2 : `[1, 3, 4, 5, 8]`.
4. Recherche du minimum dans `[5, 8]` : c'est `5` (indice 3), déjà en place : `[1, 3, 4, 5, 8]`.

La liste est triée en quatre passages. Nombre total de comparaisons : $4 + 3 + 2 + 1 = 10 = \dfrac{n(n-1)}{2}$ avec $n = 5$.

Corrigé de l'exercice 4.

```python
def deux_plus_grands(L):
    if L[0] >= L[1]:
        max1, max2 = L[0], L[1]
    else:
        max1, max2 = L[1], L[0]
    for i in range(2, len(L)):
        if L[i] >= max1:
            max2 = max1
            max1 = L[i]
        elif L[i] > max2:
            max2 = L[i]
    return max1, max2
```

Principe : on maintient deux variables : `max1` (le plus grand) et `max2` (le deuxième). Quand un élément dépasse `max1`, l'ancien `max1` devient `max2`.

Vérification : `deux_plus_grands([3, 7, 2, 9, 1])` = `(9, 7)`. Complexité : $\mathrm{O}(n)$.