Chapitre 11
Algorithmes de tri et dichotomie

Trier et chercher sont deux opérations fondamentales en informatique. Trier, c'est mettre des données dans un ordre donné (du plus petit au plus grand, par ordre alphabétique, etc.) ; chercher, c'est retrouver une valeur précise dans une collection de données. Ce chapitre présente deux algorithmes de tri classiques — le **tri par insertion** et le **tri par sélection** — et deux algorithmes de recherche — la **recherche naïve** (déjà rencontrée au chapitre 9) et la **recherche dichotomique**, beaucoup plus rapide, mais qui exige un tableau trié.

I. Les premiers algorithmes : rappels

Avant d'étudier les tris, on rappelle deux algorithmes vus au chapitre 9 : la recherche du minimum et la recherche d'une valeur dans un tableau.

Définition 11.1.
L'algorithme du **minimum** parcourt le tableau en mémorisant la plus petite valeur rencontrée. On initialise le minimum avec la première case, puis on compare chaque case au minimum courant : si elle est plus petite, elle devient le nouveau minimum.

■ **Exemple 11.1.**
On cherche le minimum de `[8, 3, 10, 5]`. On part de `m = 8`. On compare `3 < 8` : `m = 3`. Puis `10 < 3` est faux, `5 < 3` est faux. Le minimum vaut `3`. ■

Définition 11.2.
La **recherche naïve** (ou recherche séquentielle) parcourt le tableau du début à la fin et compare chaque case à la valeur cherchée. Elle s'arrête dès que la valeur est trouvée, ou à la fin du tableau si elle n'y est pas.

■ **Exemple 11.2.**
On cherche `7` dans `[3, 7, 12]`. On compare `3 != 7`, puis `7 == 7` : la valeur est trouvée en 2 comparaisons. Si l'on cherchait `9`, il faudrait parcourir tout le tableau pour conclure que la valeur est absente. ■

R Le coût de la recherche naïve est **linéaire** : pour un tableau de n éléments, il faut au plus n comparaisons. C'est le coût le plus simple possible, mais on peut faire beaucoup mieux quand le tableau est trié, comme on le verra avec la dichotomie.

II. Les algorithmes de tri

Trier un tableau, c'est réordonner ses éléments dans un ordre donné. On se limite ici aux tableaux de nombres triés en **ordre croissant**. Deux algorithmes élémentaires sont au programme : le tri par insertion et le tri par sélection. Tous les deux travaillent **en place** : ils modifient le tableau d'origine sans en construire un nouveau, en échangeant et en déplaçant les éléments.

1. Le tri par insertion

Définition 11.3.

Le **tri par insertion** consiste à construire progressivement une partie gauche triée du tableau : à chaque étape, on prend l'élément courant et on le **glisse** à sa place dans la partie déjà triée, en décalant vers la droite les éléments plus grands que lui. C'est exactement le geste du joueur de cartes qui range sa main : chaque nouvelle carte est insérée au bon endroit parmi celles déjà triées.

R Le principe du tri par insertion est illustré par l'exemple animé de la page Wikipedia « Tri par insertion » : `https://fr.wikipedia.org/wiki/Tri_par_insertion`. On peut aussi le réaliser avec un jeu de cartes : c'est l'exercice proposé plus bas.

Voici le pseudo-code du tri par insertion.

```
procedure tri_insertion(tableau T)
    pour i de 1 a taille(T) - 1
        # memoriser T[i] dans x
        x <- T[i]
        # decaler les elements T[0]..T[i-1] plus grands que x, en partant
           de T[i-1]
        j <- i
        tant que j > 0 et T[j - 1] > x
            T[j] <- T[j - 1]
            j <- j - 1
        # placer x dans le "trou" laisse par le decalage
        T[j] <- x
fin procedure
```

■ **Exemple 11.3.**
On trie `[7, 3, 9, 1]` par insertion. La partie triée est au départ `[7]`. À chaque étape, on insère l'élément courant à sa place dans la partie triée. ■

Étape	**Tableau après l'étape**
$i = 1$: on insère `3`	`[3, 7, 9, 1]`
$i = 2$: on insère `9`	`[3, 7, 9, 1]`
$i = 3$: on insère `1`	`[1, 3, 7, 9]`

■ **Exemple 11.4.**
À l'étape $i = 3$, l'élément `1` est plus petit que tous ceux de la partie triée `[3, 7, 9]` : on décale `9`, puis `7`, puis `3` vers la droite, et l'on place `1` en première position. Le décalage s'arrête quand on rencontre un élément plus petit que `x`, ou quand on a atteint le début du tableau. ■

Voici la fonction Python correspondante.

```
def tri_insertion(t):
    for i in range(1, len(t)):
        valeur = t[i]
        j = i - 1
        while j >= 0 and t[j] > valeur:
            t[j + 1] = t[j]
            j = j - 1
        t[j + 1] = valeur
    return t
```

R La boucle interne commence à $j = i - 1$ et remonte vers la gauche tant que les éléments sont plus grands que `valeur`. La condition `j >= 0` est indispensable : sans elle, on tenterait d'accéder à `t[-1]` et le tri serait incorrect. À la fin, `t[j + 1]` est le « trou » dans lequel on dépose `valeur`.

2. Le tri par sélection

Définition 11.4.

Le **tri par sélection** consiste à chercher, à chaque étape, le **plus petit élément** de la partie non triée du tableau, puis à l'**échanger** avec la première case de cette partie non triée. La partie gauche du tableau est alors définitivement triée.

R Le principe du tri par sélection est illustré par l'animation de la page Wikipedia « Tri par sélection » : https://fr.wikipedia.org/wiki/Tri_par_s%C3%A9lection.

Voici le pseudo-code du tri par sélection.

```
procedure tri_selection(tableau t)
    n <- longueur(t)
    pour i de 0 a n - 2
        min <- i
        pour j de i + 1 a n - 1
            si t[j] < t[min] alors min <- j
        fin pour
        si min != i alors echanger t[i] et t[min]
    fin pour
fin procedure
```

Exemple 11.5.
On trie `[7, 3, 9, 1]` par sélection. La partie triée est au départ vide ; à chaque étape, on cherche le minimum de ce qui reste. ■

Étape	Tableau après l'étape
$i = 0$: minimum `1`, échangé avec `7`	`[1, 3, 9, 7]`
$i = 1$: minimum `3`, déjà en place	`[1, 3, 9, 7]`
$i = 2$: minimum `7`, échangé avec `9`	`[1, 3, 7, 9]`

Exemple 11.6.
À l'étape $i = 0$, on parcourt tout le tableau pour trouver le minimum : `1` se trouve à l'indice 3. On l'échange avec `t[0]`. La première case est alors définitivement à sa place : on ne la touche plus. À l'étape $i = 1$, le minimum de `[3, 9, 7]` est `3`, qui est déjà en position : aucun échange n'est nécessaire. ■

Voici la fonction Python correspondante.

```
def tri_selection(t):
    n = len(t)
    for i in range(n - 1):
        indice_min = i
        for j in range(i + 1, n):
            if t[j] < t[indice_min]:
                indice_min = j
        t[i], t[indice_min] = t[indice_min], t[i]
    return t
```

R On mémorise l'**indice** du minimum, pas sa valeur : c'est l'indice qui permet de faire l'échange à la fin de la boucle interne. L'échange `t[i], t[indice_min] = t[indice_min], t[i]` place le minimum à la position i. Si `indice_min == i`, l'échange ne change rien, ce qui est correct.

3. Terminaison, correction et coût

Propriété 11.1.
Les tris par insertion et par sélection se **terminent** toujours : leurs boucles parcourent un nombre fini d'indices (i va de 1 à $n-1$ pour l'insertion, de 0 à $n-2$ pour la sélection), et chaque boucle interne fait décroître un indice ou avance dans le tableau.

Propriété 11.2.
Le tri par insertion est **correct** : on peut prouver par un **invariant de boucle** que, après chaque itération, la partie gauche du tableau (les indices de 0 à i) est triée. Au départ, la partie `[t[0]]` est triviale. Après l'insertion de l'élément i, la partie gauche reste triée. À la fin, tout le tableau est trié.

■ **Exemple 11.7.**
De même, pour le tri par sélection, l'invariant est : après chaque itération, les i premiers éléments sont triés **et** sont les i plus petits éléments du tableau. C'est pourquoi ils n'ont plus jamais besoin d'être déplacés. ■

Propriété 11.3.
Dans le **pire cas** (tableau trié à l'envers), les deux tris effectuent environ $n^2/2$ comparaisons : leur coût est **quadratique**, de l'ordre de n^2. Le tri par insertion a un meilleur cas intéressant : sur un tableau déjà trié, il ne fait que $n-1$ comparaisons (coût linéaire), car aucun décalage n'est nécessaire.

■ **Exemple 11.8.**
Pour un tableau de $n = 10$ éléments, le tri par sélection effectue toujours $9 + 8 +$

$\cdots + 1 = 45$ comparaisons. Le tri par insertion en effectue aussi 45 dans le pire cas, mais seulement 9 si le tableau est déjà trié. ■

III. La recherche dans un tableau

1. La recherche naïve

Définition 11.5.
La **recherche naïve** (ou recherche séquentielle) parcourt le tableau case par case jusqu'à trouver la valeur cherchée, ou jusqu'à la fin du tableau. Elle fonctionne sur n'importe quel tableau, trié ou non.

■ **Exemple 11.9.**
Voici une fonction de recherche naïve qui renvoie `True` si la valeur est dans le tableau, `False` sinon. ■

```
def recherche_naive(t, cible):
    for v in t:
        if v == cible:
            return True
    return False
```

R La fonction s'arrête dès qu'elle trouve la valeur (`return True`). Si la valeur n'est pas dans le tableau, elle parcourt tout le tableau et renvoie `False`. Dans le pire cas, il y a n comparaisons : coût linéaire.

2. La recherche dichotomique

Définition 11.6.
La **dichotomie** (du grec « couper en deux ») est une méthode de recherche qui divise l'intervalle de recherche en deux à chaque étape. Elle fonctionne **uniquement sur un tableau trié** dans l'ordre croissant.

Le principe est le suivant : on regarde l'élément au **milieu** de l'intervalle de recherche. S'il est égal à la valeur cherchée, c'est gagné. Sinon, on sait que la valeur ne peut se trouver que dans la moitié gauche (si elle est plus petite que le milieu) ou dans la moitié droite (si elle est plus grande). On recommence sur la moitié concernée, jusqu'à ce que l'intervalle soit vide ou que la valeur soit trouvée.

R Une illustration classique de la dichotomie se trouve sur Wikipedia (image « Binary search into array ») : la valeur cherchée est repérée en éliminant à chaque étape la moitié du tableau où elle ne peut pas être. On peut aussi regarder la vidéo proposée dans le notebook du chapitre : https://www.youtube.com/watch?v=14c5N8CfnUo.

Voici le pseudo-code de la recherche dichotomique.

```
debut <- 0
fin <- longueur(t) - 1
trouve <- Faux
tant que trouve != Vrai et debut <= fin
    mil <- (debut + fin) // 2
    si t[mil] == val alors
        trouve <- Vrai
    sinon si val > t[mil] alors
        debut <- mil + 1
    sinon
        fin <- mil - 1
    fin si
fin tant que
```

■ **Exemple 11.10.**
On cherche `12` dans le tableau trié `[2, 5, 8, 12, 15, 19]`. On note à chaque étape l'intervalle de recherche et le milieu. ■

début	fin	mil	t[mil]	Action
0	5	2	8	$8 < 12$: on cherche à droite, début $= 3$
3	5	4	15	$15 > 12$: on cherche à gauche, fin $= 3$
3	3	3	12	$12 = 12$: trouvé

■ **Exemple 11.11.**
À la première étape, l'intervalle est `[0, 5]`, le milieu est $(0 + 5)//2 = 2$ et `t[2] = 8`. Comme `8 < 12`, la valeur cherchée ne peut être que dans la moitié droite : on fixe `début = 3`. L'intervalle de recherche passe de 6 cases à 3 cases en une seule comparaison. ■

Voici la fonction Python correspondante.

```
def recherche_dichotomique(t, val):
    debut = 0
    fin = len(t) - 1
    while debut <= fin:
        milieu = (debut + fin) // 2
        if t[milieu] == val:
            return True
        elif val < t[milieu]:
            fin = milieu - 1
        else:
            debut = milieu + 1
    return False
```

■ **Exemple 11.12.**
Le nombre d'étapes de la dichotomie est très petit : chaque comparaison divise l'intervalle par deux. Pour un tableau de 16 éléments, il faut au plus 5 comparaisons ; pour 1000 éléments, au plus 10 ; pour un million d'éléments, au plus 20. C'est le coût **logarithmique** : environ $\log_2(n)$ comparaisons. ■

R La condition `debut <= fin` est la condition d'arrêt : quand l'intervalle devient vide (`debut > fin`), la valeur n'est pas dans le tableau et la fonction renvoie `False`. La terminaison est garantie car l'intervalle rétrécit à chaque étape.

IV. Trier et chercher des objets

Jusqu'ici, on a trié des nombres. Mais tous les algorithmes du chapitre fonctionnent aussi sur des objets, à condition de savoir **comparer** deux objets. On reprend l'exemple du notebook : des cadeaux, chacun muni d'un numéro et d'une valeur aléatoire.

```
import random as rd

class Cadeau:
    def __init__(self, numero):
        self.numero = numero
        self.valeur = rd.randint(1, 1000)

    def __str__(self):
        return f"le cadeau numero {self.numero} a pour valeur {self.valeur}
            "
```

■ **Exemple 11.13.**
On crée une liste de 500 cadeaux avec une compréhension de liste, puis on l'affiche.
■

```
liste_cadeau = [Cadeau(i) for i in range(500)]

for cadeau in liste_cadeau:
    print(cadeau)
```

■ **Exemple 11.14.**
Pour trier les cadeaux par valeur croissante, on adapte les algorithmes du chapitre : au lieu de comparer `t[j] > valeur`, on compare `cadeaux[j].valeur > valeur.valeur`. On peut aussi utiliser la fonction `sorted` avec une clé. ■

```
tries = sorted(liste_cadeau, key=lambda cadeau: cadeau.valeur)
print(tries[0]) # le cadeau de plus petite valeur
```

R La recherche dichotomique s'applique de la même façon sur une liste d'objets triée : on compare `cadeaux[milieu].valeur` avec la valeur cherchée. Le principe ne change pas, seule la comparaison est adaptée.

V. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre : tri par insertion, tri par sélection, recherche naïve et recherche dichotomique. Pensez à bien lire les énoncés, à tester vos programmes, et à interpréter les messages d'erreur éventuels.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : lecture de code court, prédiction d'affichage, étape suivante d'un tri, calcul d'un milieu de dichotomie. Chaque question doit pouvoir être traitée en moins de trente secondes.

■ Exercice 11.1.

Rappels : types, calculs et booléens. Répondre aux questions suivantes sans exécuter les programmes.

1. Quel est le type de la valeur renvoyée par l'expression suivante ?

```
3 ** 0
```

2. Que vaut l'expression suivante ?

```
23 % 6
```

3. Que vaut l'expression suivante ?

```
(10 + 4) // 2
```

4. Que vaut l'expression suivante ?

```
not (4 >= 9)
```

■

■ Exercice 11.2.

Rappels : algorithmes du chapitre 9. Répondre aux questions suivantes sans exécuter les programmes.

1. Qu'affiche le programme suivant, qui cherche le minimum ?

```
t = [6, 2, 9, 4]
m = t[0]
for v in t:
    if v < m:
        m = v
print(m)
```

2. Qu'affiche le programme suivant, qui cherche une valeur ?

```
t = [3, 8, 1, 6]
```

```
cible = 8
trouve = False
for v in t:
    if v == cible:
        trouve = True
print(trouve)
```

3. Qu'affiche le programme suivant, qui calcule une somme ?

```
t = [3, 5, 7]
somme = 0
for v in t:
    somme = somme + v
print(somme)
```

4. Quelle expression complète ce programme, qui cherche le maximum ?

```
t = [4, 1, 6, 2]
M = t[0]
for v in t:
    if v > M:
        M = ____
print(M)
```

■

■ **Exercice 11.3.**

Tri par insertion : étape suivante et principe.

1. Voici l'état d'un tri par insertion en cours : `[3, 7, 2, 9]`, partie triée `[3, 7]`. Que devient le tableau après l'insertion du `2` ?
2. Voici l'état d'un tri par insertion en cours : `[1, 5, 8, 4, 6]`, partie triée `[1, 5, 8]`. Que devient le tableau après l'insertion du `4` ?
3. Quelle expression complète ce tri par insertion ?

```
def tri_insertion(t):
    for i in range(1, len(t)):
        valeur = t[i]
        j = i - 1
        while j >= 0 and t[j] > valeur:
            t[j + 1] = t[j]
            j = ____
        t[j + 1] = valeur
```

4. Quel est le principe du tri par insertion ? (On décrit la méthode en une phrase.)
5. Que vaut le tableau `[6, 4, 9]` après la première itération ($i = 1$) du tri par insertion ?

■

■ **Exercice 11.4.**
Tri par sélection : étape suivante et principe.

1. Voici l'état d'un tri par sélection en cours : `[2, 6, 9, 3]`, partie triée `[2]`. Que devient le tableau après l'étape suivante ?
2. Quelle expression complète ce tri par sélection ?

```
def tri_selection(t):
    n = len(t)
    for i in range(n - 1):
        indice_min = i
        for j in range(i + 1, n):
            if t[j] < t[indice_min]:
                indice_min = ____
        t[i], t[indice_min] = t[indice_min], t[i]
```

3. Quelle expression complète l'échange de ce tri par sélection ?

```
____ = t[indice_min], t[i]
```

4. Quel est le principe du tri par sélection ? (On décrit la méthode en une phrase.)
5. Que vaut le tableau `[5, 8, 2]` après la première itération ($i = 0$) du tri par sélection ?

■

■ **Exercice 11.5.**
Dichotomie : indice du milieu et comparaison.

1. Dans une recherche dichotomique, quel est l'indice du milieu si `début = 0` et `fin = 9` ?
2. Dans une recherche dichotomique, quel est l'indice du milieu si `début = 4` et `fin = 9` ?
3. Dans une recherche dichotomique de `15`, on compare `t[mil] = 11`. Où continue-t-on la recherche ?
4. Dans une recherche dichotomique de `4`, on compare `t[mil] = 7`. Où continue-t-on la recherche ?
5. Quelle condition le tableau doit-il vérifier pour que la recherche dichotomique fonctionne ?

■

■ **Exercice 11.6.**
Dichotomie : nombre d'étapes et condition d'arrêt.

1. Au plus combien de comparaisons la recherche dichotomique effectue-t-elle

dans un tableau trié de 8 éléments ?
2. Au plus combien de comparaisons la recherche dichotomique effectue-t-elle dans un tableau trié de 32 éléments ?
3. Quelle condition complète la boucle de la recherche dichotomique ?

```
while ____:
    milieu = (debut + fin) // 2
    ...
```

4. La recherche dichotomique de `6` dans `[1, 3, 5, 7]` : combien de comparaisons effectue-t-elle avant de conclure que `6` n'est pas présent ?

■

Exercices d'application

Les exercices d'application reprennent ceux du notebook du chapitre, complétés par des exercices supplémentaires. Ils vont du plus simple au plus exigeant.

■ **Exercice 11.7.**
Rappels : minimum et recherche naïve. Refaire l'algorithme du minimum, puis l'algorithme de la recherche naïve, avec une liste de nombres.

1. Écrire un programme qui affiche le minimum de la liste `[12, 5, 18, 9, 3]`.
2. Écrire une fonction `recherche_naive(t, cible)` qui renvoie `True` si `cible` est dans la liste `t` et `False` sinon. Tester avec `t = [4, 9, 2, 7]` et `cible = 9`, puis `cible = 5`.

```
t = [12, 5, 18, 9, 3]
# Ecrire ici l'algorithme du minimum
```

■

■ **Exercice 11.8.**
Tri par insertion avec des cartes. Réaliser l'exemple de Wikipedia (page « Tri par insertion ») avec un jeu de cartes, puis refaire le tri avec une configuration de cartes différente de l'exemple. Pour chaque configuration, décrire les étapes : quelle carte est insérée, où, et quelles cartes sont décalées. ■

■ **Exercice 11.9.**
Tri par insertion en Python.

1. Faire une liste par compréhension de 30 nombres compris entre 1 et 20.
2. Faire un script qui trie par insertion cette liste de nombres.

3. Faire une fonction qui prend en entrée une liste et qui renvoie cette liste triée.
4. Appliquer cette fonction sur la liste de la question 1 et afficher le résultat.

```
from random import randint
t = [randint(1, 20) for i in range(30)]
# Ecrire ici le script de tri par insertion
```

■

■ **Exercice 11.10.**
Tri par sélection avec des cartes. Réaliser l'animation de Wikipedia (page « Tri par sélection ») avec un jeu de cartes, puis refaire le tri avec une configuration de cartes différente. Pour chaque configuration, décrire les étapes : quel est le minimum de la partie non triée, avec quelle carte est-il échangé. ■

■ **Exercice 11.11.**
Tri par sélection en Python.

1. Faire une liste par compréhension de 30 nombres compris entre 1 et 20.
2. Faire un script qui trie par sélection cette liste de nombres.
3. Faire une fonction qui prend en entrée une liste et qui renvoie cette liste triée.
4. Appliquer cette fonction sur la liste de la question 1 et afficher le résultat.

```
from random import randint
t = [randint(1, 20) for i in range(30)]
# Ecrire ici le script de tri par selection
```

■

■ **Exercice 11.12.**
Recherche naïve en fonction. Écrire une fonction `recherche_naive(t, cible)` qui prend en entrée une liste de nombres et une valeur, et qui renvoie `True` si la valeur est dans la liste et `False` sinon. Tester sur `[2, 7, 4, 9]` avec `cible = 4` puis `cible = 1`. ■

■ **Exercice 11.13.**

Recherche dichotomique avec des cartes. Réaliser l'exemple de l'image du notebook (recherche binaire dans un tableau trié) avec des cartes, puis refaire la recherche avec une configuration de cartes différente. Pour chaque configuration, décrire les étapes : quel est le milieu, que compare-t-on, quelle moitié est conservée.
■

■ **Exercice 11.14.**
Recherche dichotomique en Python.

1. Faire une liste par compréhension de 15 nombres compris entre 1 et 30, puis la trier.
2. Faire un script qui recherche par dichotomie la valeur `7` dans cette liste.
3. Faire une fonction `recherche_dichotomique(t, val)` qui prend en entrée une liste triée et une valeur, et qui renvoie `True` si la valeur est dans la liste et `False` sinon.
4. Appliquer cette fonction sur la liste de la question 1 avec la valeur `7`.

```
from random import randint
t = sorted([randint(1, 30) for i in range(15)])
# Ecrire ici le script de recherche dichotomique de 7
```

■

■ **Exercice 11.15.**
Trier à la main par insertion. Écrire toutes les étapes du tri par insertion du tableau `[7, 3, 9, 1, 5]`. Pour chaque étape, indiquer l'élément inséré, les décalages effectués et le tableau obtenu. ■

■ **Exercice 11.16.**
Trier à la main par sélection. Écrire toutes les étapes du tri par sélection du tableau `[7, 3, 9, 1, 5]`. Pour chaque étape, indiquer le minimum trouvé, l'échange effectué et le tableau obtenu. ■

■ **Exercice 11.17.**
Compléter un tri par insertion. Compléter les pointillés pour que la fonction trie correctement la liste.

```
def tri_insertion(t):
```

```
    for i in range(1, len(t)):
        valeur = t[i]
        j = i - 1
        while j >= 0 and ____:
            t[j + 1] = t[j]
            j = ____
        ____ = valeur
```

■

■ **Exercice 11.18.**
Compléter un tri par sélection. Compléter les pointillés pour que la fonction trie correctement la liste.

```
def tri_selection(t):
    n = len(t)
    for i in range(n - 1):
        indice_min = ____
        for j in range(i + 1, n):
            if t[j] < t[indice_min]:
                indice_min = ____
        t[i], t[indice_min] = ____
```

■

■ **Exercice 11.19.**
Recherche dichotomique pas à pas. Dérouler la recherche dichotomique de la valeur 7 dans `[1, 3, 5, 7, 9, 11]` en notant, à chaque étape, les valeurs de `début`, `fin`, `milieu`, la valeur de `t[milieu]` et l'action effectuée. Combien de comparaisons au total ?
■

■ **Exercice 11.20.**
Nombre d'étapes de la dichotomie.

1. Au plus combien de comparaisons la recherche dichotomique effectue-t-elle dans un tableau trié de 64 éléments ? Justifier.
2. Même question pour un tableau de 100 éléments. Comparer avec la recherche naïve.

■

■ **Exercice 11.21.**
Vérifier qu'un tableau est trié. Écrire une fonction `est_triee(t)` qui renvoie `True` si la liste `t` est triée en ordre croissant et `False` sinon. Tester sur `[1, 3, 3, 7]`, `[5, 2, 9]` et `[]`.

```
def est_triee(t):
    # Ecrire ici la verification
```

■

■ **Exercice 11.22.**
Tous les algorithmes sur une liste de cadeaux. Reprendre la classe `Cadeau` du cours et la liste `liste_cadeau` de 500 cadeaux, puis :

1. Chercher le cadeau de plus petite valeur (algorithme du minimum).
2. Chercher si un cadeau d'une valeur donnée existe (recherche naïve).
3. Trier la liste des cadeaux par valeur croissante (tri par insertion ou tri par sélection adaptés).
4. Rechercher une valeur par dichotomie dans la liste triée (en comparant les attributs `.valeur`).

```
liste_cadeau = [Cadeau(i) for i in range(500)]
# Ecrire ici les algorithmes adaptes aux cadeaux
```

■

Exercices de réflexion

Les exercices de réflexion demandent plus d'analyse : implémentation complète, comparaison d'algorithmes, repérage d'erreurs, preuve de correction. Ils sont classés du plus abordable au plus exigeant.

■ **Exercice 11.23.**
Implémenter le tri par insertion. Écrire une fonction `tri_insertion(t)` qui trie une liste `t` en place et la renvoie. Ajouter des `print` pour observer le tableau après chaque itération, et tester sur `[5, 2, 8, 1, 9]` et sur `[1, 2, 3, 4]`.

```
def tri_insertion(t):
    # Ecrire ici l'implementation
```

■

■ **Exercice 11.24.**

Implémenter le tri par sélection. Écrire une fonction `tri_selection(t)` qui trie une liste `t` en place et la renvoie. Ajouter des `print` pour observer le tableau après chaque itération, et tester sur `[5, 2, 8, 1, 9]` et sur `[9, 7, 5, 3, 1]`.

```
def tri_selection(t):
    # Ecrire ici l'implementation
```

■

■ **Exercice 11.25.**
Recherche naïve ou dichotomique ? Une liste triée de 1000 nombres contient la valeur cherchée.

1. Au plus combien de comparaisons la recherche naïve effectue-t-elle ? Et la recherche dichotomique ?
2. Sur quelle liste la dichotomie est-elle impossible ? Pourquoi ?
3. Que conclure sur l'intérêt de trier les données avant de les chercher ?

■

■ **Exercice 11.26.**
Erreur à corriger (tri par insertion). Le programme suivant devrait trier `[4, 2, 7, 1]` mais il produit `[4, 1, 2, 7]`. Expliquer l'erreur et corriger le programme.

```
def tri_insertion(t):
    for i in range(1, len(t)):
        valeur = t[i]
        j = i - 1
        while j > 0 and t[j] > valeur:
            t[j + 1] = t[j]
            j = j - 1
        t[j + 1] = valeur
```

1. Pourquoi le programme ne trie-t-il pas correctement ?
2. Corriger le programme et vérifier sur `[4, 2, 7, 1]`.

■

■ **Exercice 11.27.**
Erreur à corriger (tri par sélection). Le programme suivant devrait trier `[4, 2, 7, 1]` mais il ne modifie pas la liste. Expliquer l'erreur et corriger le programme.

```
def tri_selection(t):
    n = len(t)
```

```
    for i in range(n - 1):
        indice_min = i
        for j in range(i + 1, n):
            if t[j] < t[indice_min]:
                indice_min = j
    return t
```

1. Pourquoi le programme ne modifie-t-il pas la liste ?
2. Corriger le programme et vérifier sur `[4, 2, 7, 1]`.

■

■ **Exercice 11.28.**
Dichotomie sur un cas concret. Un dictionnaire papier contient 100 000 mots rangés par ordre alphabétique.

1. En cherchant un mot par dichotomie (on ouvre le dictionnaire au milieu, puis dans la moitié concernée, etc.), au plus combien de mots doit-on examiner ? Justifier.
2. Pourquoi la dichotomie est-elle impossible dans un dictionnaire dont les mots ne sont pas triés ?
3. Le tri par insertion de ces 100 000 mots nécessiterait environ 5×10^9 comparaisons dans le pire cas. Expliquer pourquoi il vaut mieux trier une fois, puis chercher par dichotomie, plutôt que de chercher naïvement dans le dictionnaire non trié.

■

■ **Exercice 11.29.**
Prouver la correction du tri par insertion. On considère l'invariant suivant : « après chaque itération de la boucle `for i in range(1, len(t))`, la partie gauche du tableau (indices 0 à i) est triée ».

1. Pourquoi l'invariant est-il vrai au départ, avant la première itération ?
2. Expliquer pourquoi une itération préserve l'invariant : après l'insertion de `t[i]` à sa place, la partie gauche reste triée.
3. En déduire que le tableau entier est trié à la fin du programme.

■