Chapitre 1
Rappel sur les algorithmes

Ce chapitre est un chapitre de rappel : il réactive les algorithmes fondamentaux étudiés en Première, en les appliquant cette fois à des listes d'**objets**. On va faire tourner tous nos algorithmes sur des listes d'objets. Ces objets seront des cadeaux (faut qu'on pense positif !) Ces cadeaux auront des valeurs aléatoires. Nous reverrons ainsi l'algorithme du **minimum** et du **maximum**, les **tris** par insertion et par sélection, et les algorithmes de **recherche** (recherche naïve et recherche par dichotomie).

■ **Exemple 1.1.**
Pour créer nos cadeaux, on définit une classe `Cadeau`. Chaque cadeau possède un numéro (passé en paramètre) et une valeur aléatoire comprise entre 1 et 1000. ■

```
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 1.2.**
On crée alors une liste de 500 cadeaux avec une compréhension de liste, puis on l'affiche avec une boucle `for`. ■

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

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

I. Les algorithmes du minimum et du maximum

Dans ces algorithmes, on va chercher le min (ou le max) en comparant tous les objets à l'objet qui a la plus petite valeur (qu'on connaît). Si l'objet est plus petit que l'objet le plus petit, alors le nouvel objet est le minimum ; sinon l'ancien objet reste le minimum.

Définition 1.1.
L'algorithme du **minimum** parcourt la liste 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 1.3.**
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`. ■

Le pseudo-code est le suivant :

```
Entree : Liste
Sortie : m minimum

Algo
m = L[0]
Pour chaque element de Liste:
    Faire
        Si element < m Alors m = element
    Fin Faire
Renvoyer m
```

R Ce pseudo-code s'applique tel quel à des nombres. Pour des objets comme nos cadeaux, il y a une petite modification à faire : on ne compare pas les objets eux-mêmes, mais leur **attribut** `valeur`. C'est cette adaptation qui est demandée dans les exercices.

La fonction Python correspondante, sur des nombres, est la suivante :

```
def minimum(t):
    m = t[0]
    for v in t:
        if v < m:
            m = v
    return m
```

■ **Exemple 1.4.**
Adaptons cette fonction à nos cadeaux : on compare `cadeau.valeur` avec `m.valeur`.
■

```
def minimum_cadeaux(cadeaux):
    m = cadeaux[0]
    for cadeau in cadeaux:
        if cadeau.valeur < m.valeur:
            m = cadeau
    return m
```

■ **Exemple 1.5.**
On teste sur la liste des 500 cadeaux : la fonction renvoie le cadeau de plus petite valeur. ■

```
print(minimum_cadeaux(liste_cadeau))
```

On veut maintenant renvoyer en plus du minimum **l'indice** du minimum. La fonction `enumerate` permet de parcourir une liste en récupérant à la fois l'indice et l'élément : pour chaque couple `(i, cadeau)`, `i` est l'indice et `cadeau` l'élément.

■ **Exemple 1.6.**
On mémorise à la fois la plus petite valeur `m` et l'indice `indice` auquel elle se trouve.
■

```
def indice_du_minimum(cadeaux):
    m = cadeaux[0].valeur
    indice = 0
    for i, cadeau in enumerate(cadeaux):
        if cadeau.valeur < m:
            m = cadeau.valeur
            indice = i
    return indice
```

■ **Exemple 1.7.**
Pour le maximum, on procède exactement de la même manière, en inversant le sens de la comparaison : on cherche l'objet le plus grand et l'indice du maximum. ■

```
def maximum_cadeaux(cadeaux):
    M = cadeaux[0]
    for cadeau in cadeaux:
        if cadeau.valeur > M.valeur:
            M = cadeau
    return M

def indice_du_maximum(cadeaux):
    M = cadeaux[0].valeur
    indice = 0
    for i, cadeau in enumerate(cadeaux):
        if cadeau.valeur > M:
            M = cadeau.valeur
            indice = i
    return indice
```

R La structure des algorithmes du minimum et du maximum est identique : seule la comparaison change (< ou >). Ces algorithmes parcourent toute la liste : pour une liste de n éléments, ils effectuent $n-1$ comparaisons. On dit que leur coût est **linéaire**, de l'ordre de n.

II. Algorithmes de tris

Trier une liste, c'est réordonner ses éléments dans un ordre donné (croissant ou décroissant). Nous allons revoir deux algorithmes de tri classiques : le **tri par insertion** et le **tri par sélection**.

R Une petite vidéo illustre bien le fonctionnement de ces tris : https://www.youtube.com/watch?v=14c5N8CfnUo. On peut aussi les réaliser physiquement avec un jeu de cartes.

1. Le tri par insertion

Définition 1.2.

Le **tri par insertion** consiste à construire progressivement une partie gauche triée de la liste : 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.

Voici le pseudo-code du tri par insertion (merci Wikipedia) :

```
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] qui sont plus grands que x
        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 1.8.**
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	Liste 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 1.9.**
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 de la liste. ■

Voici la fonction Python correspondante, adaptée aux cadeaux (on compare les valeurs des cadeaux) :

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

■ **Exemple 1.10.**
On crée une première liste `liste_par_insertion`, copie de `liste_cadeau`, triée par valeur croissante grâce à notre fonction. ■

```
liste_par_insertion = tri_insertion_cadeaux(liste_cadeau[:])
print(liste_par_insertion[0]) # le cadeau de plus petite valeur
print(liste_par_insertion[-1]) # le cadeau de plus grande valeur
```

R La copie `liste_cadeau[:]` est importante : le tri modifie la liste **en place**. Sans copie, la liste d'origine serait elle aussi triée.

2. Le tri par sélection

Définition 1.3.
Le **tri par sélection** consiste à chercher, à chaque étape, le **plus petit élément** de la partie non triée de la liste, puis à l'**échanger** avec la première case de cette partie non triée. La partie gauche de la liste est alors définitivement triée.

Voici le pseudo-code du tri par sélection (merci Wikipedia) :

```
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 1.11.**
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	**Liste 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 1.12.**
À l'étape $i = 0$, on parcourt toute la liste 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, adaptée aux cadeaux :

```
def tri_selection_cadeaux(cadeaux):
    n = len(cadeaux)
    for i in range(n - 1):
        indice_min = i
        for j in range(i + 1, n):
            if cadeaux[j].valeur < cadeaux[indice_min].valeur:
                indice_min = j
        cadeaux[i], cadeaux[indice_min] = cadeaux[indice_min], cadeaux[i]
    return cadeaux
```

■ **Exemple 1.13.**
On veut créer une liste `liste_par_selection` de cadeaux triée par valeur **décroissante**. Pour cela, on inverse le sens de la comparaison dans la recherche du minimum : on cherche le **maximum** à chaque étape. Le code de cette adaptation est demandé dans les exercices. ■

R On mémorise l'**indice** du minimum (ou du maximum), pas sa valeur : c'est l'indice qui permet de faire l'échange à la fin de la boucle interne. L'échange `cadeaux[i], cadeaux[indice_min] = cadeaux[indice_min], cadeaux[i]` place le minimum à la position i. Si `indice_min == i`, l'échange ne change rien, ce qui est correct.

3. Coût des tris

Propriété 1.1.
Dans le **pire cas** (liste triée à l'envers), les tris par insertion et par sélection 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 une liste déjà triée, il ne fait que $n-1$ comparaisons (coût linéaire), car aucun décalage n'est nécessaire.

■ **Exemple 1.14.**
Pour une liste 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 la liste est déjà triée. ■

III. Algorithmes de recherche

1. La recherche naïve

Définition 1.4.
La **recherche naïve** (ou recherche séquentielle) parcourt la liste 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 de la liste si elle n'y est pas. Elle fonctionne sur n'importe quelle liste, triée ou non.

■ **Exemple 1.15.**
Voici une fonction de recherche naïve, sur des nombres : elle renvoie `True` si la valeur est dans la liste, `False` sinon. ■

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

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

2. La recherche par dichotomie

Définition 1.5.
La **dichotomie** (du grec « couper en deux ») est une méthode de recherche qui divise l'intervalle de recherche en deux à chaque étape. Cet algorithme fonctionne **seulement** sur des tableaux triés (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.

Voici le pseudo-code de la recherche dichotomique (merci Wikipedia) :

```
trouve = Faux
debut = 0
fin = N # N est la taille du tableau
val = valeur cherchee

Tant que trouve != Vrai et debut <= fin:
    mil <- partie_entiere((debut + fin) / 2)
    si t[mil] == val:
        trouve <- Vrai
    sinon:
        si val > t[mil]:
            debut <- mil + 1
        sinon:
            fin <- mil - 1
```

■ **Exemple 1.16.**
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 1.17.**
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, adaptée aux cadeaux :

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

■ **Exemple 1.18.**
On cherche si un cadeau de valeur 43 se trouve dans la liste triée par ordre croissant. On passe une **copie** triée de la liste à la fonction. ■

```
liste_triee = tri_insertion_cadeaux(liste_cadeau[:])
print(recherche_dichotomique(liste_triee, 43))
```

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

■ **Exemple 1.19.**
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. ■

IV. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre : minimum et maximum, tris par insertion et par sélection, recherche naïve et recherche dichotomique, le tout sur des objets de la classe `Cadeau`. Pensez à bien lire les énoncés, à écrire des programmes propres et à tester vos fonctions.

Exercices d'automatismes

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

■ Exercice 1.1.

Dire si les affirmations suivantes sont vraies ou fausses.

1. Dans l'algorithme du minimum, on initialise le minimum avec la première case de la liste.
2. La recherche par dichotomie fonctionne sur n'importe quelle liste, triée ou non.
3. Dans le tri par insertion, on décale vers la droite les éléments plus grands que l'élément à insérer.
4. La recherche naïve s'arrête dès que la valeur cherchée est trouvée.
5. Dans le tri par sélection, on échange le minimum de la partie non triée avec la première case de cette partie.

■

■ Exercice 1.2.

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

1. Que vaut l'expression suivante ?

```
23 % 6
```

2. Que vaut l'expression suivante ?

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

3. Que vaut l'expression suivante ?

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

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

```
3 ** 0
```

■

Exercice 1.3.

Prédire l'affichage des programmes suivants, sans les exécuter.

1. Qu'affiche ce programme, 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 ce programme, qui cherche une valeur ?

```
t = [3, 8, 1, 6]
cible = 8
trouve = False
for v in t:
    if v == cible:
        trouve = True
print(trouve)
```

Exercice 1.4.

Compléter les programmes suivants.

1. 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
```

2. Quelle expression complète la recherche du minimum dans 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]
```

■ **Exercice 1.5.**
Étapes suivantes de tri et de dichotomie.

1. Voici l'état d'un tri par insertion en cours : `[3, 7, 2, 9]`, partie triée `[3, 7]`. Que devient la liste après l'insertion du `2` ?
2. Voici l'état d'un tri par sélection en cours : `[2, 6, 9, 3]`, partie triée `[2]`. Que devient la liste après l'étape suivante ?
3. On cherche `7` dans `[1, 3, 7, 9, 12, 15]`. Quelles sont les valeurs de `début`, `fin` et `milieu` à la première étape de la recherche dichotomique ?

■

Exercices d'application

Les exercices d'application demandent d'écrire les fonctions du chapitre, en les adaptant aux objets de la classe `Cadeau`.

■ **Exercice 1.6.**
Écrire le code Python qui renvoie le cadeau avec la plus petite valeur, en adaptant la fonction `minimum` vue dans le cours (il y a une petite modification à faire). Tester sur la liste `liste_cadeau`.

```
def minimum_cadeaux(cadeaux):
    ...
```

■

■ **Exercice 1.7.**
Annale de Bac (presque). Proposer une fonction qui prend en entrée une valeur `v` et qui renvoie `True` s'il y a un cadeau de valeur `v` dans la liste et `False` sinon.

```
def recherche_naive_cadeaux(cadeaux, v):
    ...
```

■

■ **Exercice 1.8.**
Adapter le code de la question précédente (on peut utiliser `enumerate`) pour qu'il renvoie en plus du minimum **l'indice** du minimum.

```
def indice_du_minimum(cadeaux):
    ...
```

■

■ **Exercice 1.9.**
Réécrire le code précédent avec le **maximum** : on veut l'objet le plus grand et l'indice du maximum.

```
def indice_du_maximum(cadeaux):
    ...
```

■

■ **Exercice 1.10.**
Écrire le code du tri par insertion pour créer une première liste `liste_par_insertion` de cadeaux triée par valeur **croissante**.

```
def tri_insertion_cadeaux(cadeaux):
    ...
```

■

■ **Exercice 1.11.**
Écrire le code du tri par sélection pour créer une première liste `liste_par_selection` de cadeaux triée par valeur **décroissante**.

```
def tri_selection_decroissant(cadeaux):
    ...
```

■

■ **Exercice 1.12.**
Implémenter l'algorithme de recherche par dichotomie en Python, puis chercher s'il y a un cadeau de valeur 43 dans la liste triée par ordre croissant.

```
def recherche_dichotomique(cadeaux, v):
    ...
```

■

Exercices de réflexion

Les exercices de réflexion demandent d'aller plus loin : analyser les algorithmes, les adapter, et comparer leurs coûts.

■ **Exercice 1.13.**
Modifier la fonction `recherche_dichotomique` pour qu'elle renvoie l'**indice** de la valeur cherchée si elle est présente, et `-1` sinon.

1. Expliquer pourquoi la valeur `-1` est un bon choix pour signaler l'absence de la valeur.
2. Écrire la fonction modifiée.
3. Combien de comparaisons faut-il pour chercher la valeur 43 dans une liste de 500 cadeaux triée ?

■

■ **Exercice 1.14.**
On veut trier les cadeaux par valeur décroissante avec le tri par insertion.

1. Quelle condition de la boucle interne faut-il modifier ?
2. Écrire la fonction `tri_insertion_decroissant`.
3. On veut maintenant trier les cadeaux par **numéro** croissant. Quelle comparaison faut-il changer ? Écrire la fonction `tri_insertion_par_numero`.

■

■ **Exercice 1.15.**
Comparer la recherche naïve et la recherche dichotomique.

1. Pour une liste triée de n éléments, combien de comparaisons la recherche naïve effectue-t-elle dans le pire cas ? Et la recherche dichotomique ?
2. Compléter le tableau suivant.

Taille de la liste	**Recherche naïve (pire cas)**	**Dichotomie (au plus)**
$n = 100$		
$n = 10\,000$		
$n = 1\,000\,000$		

3. Le tri d'une liste a un coût quadratique (de l'ordre de n^2). Expliquer pourquoi, sur une liste triée une seule fois, il peut tout de même être rentable de la trier avant de faire plusieurs recherches dichotomiques.

■