TP Chapitre 1 : Rappel sur les algorithmes

Objectifs du TP

Objectifs du TP. À l'issue de cette séance, tu dois être capable de :

— retrouver et écrire les algorithmes du minimum et du maximum sur une liste de prix ;

— adapter ces algorithmes à des objets de la classe `Article` et retrouver l'indice du minimum ou du maximum ;

— écrire les tris par insertion et par sélection, en versions croissante et décroissante ;

— écrire la recherche naïve et la recherche par dichotomie, et justifier la condition d'arrêt ;

— comparer les coûts des algorithmes (linéaire, quadratique, logarithmique) en comptant les comparaisons.

Consignes générales. Ce TP reprend les algorithmes du chapitre (minimum, maximum, tris, recherche) en les appliquant à un catalogue d'articles d'une boutique : chaque article possède un numéro et un prix. La classe `Article` est une adaptation de la classe `Cadeau` du cours. Tous les exercices se traitent en Python (console ou fichier). Enregistre régulièrement ton travail. Quand un programme ne fonctionne pas, lis le message d'erreur : il indique souvent la ligne et le type d'erreur. La partie B se fait *en autonomie* : tu écris des programmes entiers à partir d'une spécification, aucune ligne de code n'est fournie.

Partie A — Application guidée

Les exercices suivants se programment en Python. Ils sont classés du plus simple au plus difficile. Chaque exercice se termine par un *point de validation* (encadré vert) : le résultat exact que doit produire ton programme pour être juste. Complète les squelettes de code en respectant l'indentation.

Exercice A1 — Le prix minimum et maximum d'un catalogue

Le gérant d'une boutique veut connaître les prix extrêmes de son catalogue. Il dispose d'une liste de prix (en euros) et souhaite deux fonctions : l'une qui renvoie le plus petit prix, l'autre le plus grand.

```
def minimum(liste):
    m = liste[0]
    for v in liste:
        # A COMPLETER : si v est plus petit que m, alors m = v
    return m

def maximum(liste):
    M = liste[0]
    for v in liste:
        # A COMPLETER : si v est plus grand que M, alors M = v
    return M

liste_prix = [8, 3, 10, 5]
print("Minimum :", minimum(liste_prix))
print("Maximum :", maximum(liste_prix))
```

Point de validation

Vérifie ton programme : il doit afficher `Minimum : 3` puis `Maximum : 10`.

Exercice A2 — L'article le moins cher (adaptation aux objets)

Le catalogue est une liste d'objets de la classe `Article` : chaque article possède un numéro et un prix aléatoire entre 1 et 1000 €. Pour trouver l'article le moins cher, on ne compare plus des nombres mais l'*attribut* `prix` des objets. La fonction demandée mémorise le plus petit prix rencontré et son *indice* (la fonction `enumerate` donne l'indice et l'élément à chaque tour).

Pour tester, on remplace les prix aléatoires par des prix connus : c'est la seule façon d'obtenir un résultat reproductible.

```
import random as rd

class Article:
    def __init__(self, numero):
        self.numero = numero
        self.prix = rd.randint(1, 1000)

    def __str__(self):
        return f"l'article numero {self.numero} coute {self.prix} euros"

def indice_du_moins_cher(articles):
    m = articles[0].prix
    indice = 0
    for i, article in enumerate(articles):
        # A COMPLETER : si article.prix < m, mettre a jour m et indice
    return indice

# On remplace les prix aleatoires par des prix connus pour tester
catalogue = [Article(i) for i in range(5)]
prix_connus = [45, 12, 78, 30, 56]
for i in range(5):
    catalogue[i].prix = prix_connus[i]

print("Indice de l'article le moins cher :", indice_du_moins_cher(catalogue)
    )
print(catalogue[indice_du_moins_cher(catalogue)])
```

Point de validation

Vérifie ton programme : il doit afficher `Indice de l'article le moins cher : 1` puis `l'article numero 1 coute 12 euros`.

Exercice A3 — Le tri par insertion du catalogue

Le gérant veut afficher son catalogue par prix croissant. On utilise le **tri par insertion** vu dans le cours : on construit progressivement une partie gauche triée en glissant chaque article à sa place, en décalant vers la droite les articles plus chers que lui.

```
def tri_insertion_articles(articles):
    for i in range(1, len(articles)):
        article = articles[i]
        j = i - 1
        while j >= 0 and articles[j].prix > article.prix:
            # A COMPLETER : decaler articles[j] vers la droite, puis j = j -
                1
        # A COMPLETER : placer article dans le trou laisse par le decalage
    return articles

liste_triee = tri_insertion_articles(catalogue[:])
for article in liste_triee:
```

```
    print(article)
```

Point de validation

Avec le catalogue de l'exercice A2 (prix 45, 12, 78, 30, 56), les prix affichés doivent être dans l'ordre croissant : `12`, `30`, `45`, `56`, `78`.

Exercice A4 — La recherche naïve d'un prix

Un client demande : « y a-t-il un article à ce prix dans le catalogue ? ». On écrit une **recherche naïve** : on parcourt la liste du début à la fin et on s'arrête dès qu'un article a exactement le prix cherché.

```
def recherche_prix(articles, prix):
    for article in articles:
        # A COMPLETER : si article.prix == prix, renvoyer True
    return False

print(recherche_prix(catalogue, 30))
print(recherche_prix(catalogue, 99))
```

Point de validation

Vérifie ton programme : il doit afficher `True` (il existe un article à 30 €) puis `False` (aucun article à 99 €).

Partie B — En autonomie

Tu écris maintenant des programmes *entiers*, sans squelette. Pour chaque exercice : lis la spécification (entrées, traitement, sorties, contraintes), réponds d'abord aux questions de conception de l'encadré orange (sur papier ou à voix haute), puis écris ton programme dans un fichier et teste-le avec les valeurs de l'encadré vert. L'objectif est de concevoir toi-même la solution : ne demande pas de lignes de code toutes faites.

Exercice B1 — Le catalogue du plus cher au moins cher (tri par sélection)

Le gérant veut aussi un affichage par prix *décroissant* : du plus cher au moins cher. On utilise le **tri par sélection**, en cherchant cette fois le *maximum* à chaque étape.

Énoncé. Écrire une fonction `tri_selection_decroissant(articles)` qui trie une liste d'articles par prix décroissant, à l'aide de l'algorithme de tri par sélection adapté au maximum.

Spécification.

— *Entrées* : une liste d'articles (classe `Article`, attribut `prix`).

— *Traitement* : pour chaque position i de la liste, chercher l'indice du prix *maximal* dans la partie non triée (de i à la fin), puis échanger l'article de cette position avec l'article de l'indice trouvé.

— *Sorties* : la liste triée par prix décroissant, modifiée *en place* et renvoyée.

— *Contraintes* : deux boucles imbriquées ; on mémorise un *indice*, jamais un article ; la liste d'origine est modifiée (penser à la copier avec `[:]` avant d'appeler la fonction si on veut la conserver).

Pour t'aider — questions de conception (sans donner le code)

- Pourquoi faut-il mémoriser l'*indice* du maximum et pas le maximum lui-même ?
- Que garantit-on sur la partie gauche de la liste après l'étape i ? Pourquoi peut-on la laisser de côté ensuite ?
- Que se passe-t-il si le maximum est déjà à la bonne place ? (l'échange est-il gênant ?)
- Pour une liste de n articles, combien de comparaisons le tri par sélection effectue-t-il, quel que soit l'ordre des prix ?

Point de validation

Avec le catalogue de l'exercice A2 (prix 45, 12, 78, 30, 56), les prix affichés doivent être `78`, `56`, `45`, `30`, `12`.

Exercice B2 — La recherche par dichotomie

Sur un grand catalogue, la recherche naïve est lente. Si la liste est *triée* par prix croissant, la **recherche par dichotomie** divise l'intervalle de recherche par deux à chaque comparaison.

Énoncé. Écrire une fonction `recherche_dichotomique(articles, prix)` qui renvoie `True` si un article a exactement ce prix, `False` sinon.

Spécification.

- *Entrées* : une liste d'articles *déjà triée* par prix croissant, et un prix entier.
- *Traitement* : tant que l'intervalle de recherche n'est pas vide, regarder l'article au milieu : s'il a le bon prix, c'est gagné ; sinon, continuer dans la moitié gauche ou dans la moitié droite selon la comparaison.
- *Sorties* : `True` si le prix est présent, `False` sinon.
- *Contraintes* : la liste passée en entrée doit être triée (précondition) ; boucle `while` ; la condition d'arrêt est `debut <= fin`.

Pour t'aider — questions de conception (sans donner le code)

- Pourquoi la liste doit-elle être triée ? Que se passerait-il sur une liste non triée ?
- Écris en français le déroulement d'une étape : on compare le prix cherché au prix du milieu. Quels sont les trois cas ?
- Pourquoi la condition d'arrêt est-elle `debut <= fin` ? Que signifie `debut > fin` ?
- Pour une liste de 16 articles, combien de comparaisons au maximum ? Et pour 1000 ?

Point de validation

Sur la liste triée de l'exercice A3 (prix 12, 30, 45, 56, 78) : `recherche_dichotomique(liste_triee, 45)` doit renvoyer `True` et `recherche_dichotomique(liste_triee, 99)` doit renvoyer `False`.

Exercice B3 — La meilleure offre sous budget

Un client dispose d'un budget : il veut l'article le *moins cher* parmi ceux qu'il peut s'offrir (prix inférieur ou égal au budget). C'est une variante de l'algorithme du minimum : on ne compare que les articles respectant le budget.

Énoncé. Écrire une fonction `meilleure_offre(articles, budget)` qui renvoie l'article le moins cher dont le prix ne dépasse pas le budget, ou `None` si aucun article ne convient. Le programme principal affiche l'offre trouvée, ou `Aucun article dans le budget.`

Spécification.

— *Entrées* : une liste d'articles et un budget (entier positif, en euros).

— *Traitement* : parcourir la liste une seule fois en mémorisant l'article le moins cher dont le prix est inférieur ou égal au budget.

— *Sorties* : l'article choisi (via `print`), ou le message `Aucun article dans le budget.`

— *Contraintes* : un seul parcours de la liste (coût linéaire) ; la variable qui mémorise la meilleure offre est initialisée à `None`.

Pour t'aider — questions de conception (sans donner le code)

— Quelle est la différence avec l'algorithme du minimum du cours ? (deux différences)

— Pourquoi initialiser la meilleure offre à `None` plutôt qu'au premier article de la liste ?

— Dans quel cas la fonction doit-elle renvoyer `None` ? Que doit alors afficher le programme principal ?

— Le coût de cette fonction reste-t-il linéaire ?

Point de validation

Avec le catalogue de l'exercice A2 et un budget de 50, le programme doit afficher `l'article numero 1 coute 12 euros` ; avec un budget de 5, il doit afficher `Aucun article dans le budget.`

Exercice B4 — Comparaison des coûts des tris

Le gérant veut savoir lequel de ses deux tris est le plus efficace sur les données de vente. On va *compter les comparaisons de prix* effectuées par chacun : plus il y en a, plus le tri est long.

Énoncé. Écrire deux fonctions `tri_insertion_compteur(articles)` et `tri_selection_compteur(articles)` qui trient une liste d'articles (comme aux exercices A3 et B1) et renvoient en plus le nombre de comparaisons de prix effectuées. Le programme principal teste les deux fonctions sur une petite liste aux prix connus, puis sur des listes aléatoires de tailles 50, 100 et 200, et affiche les nombres de comparaisons.

Spécification.

— *Entrées* : aucune saisie (les listes de test sont construites dans le programme ; la classe `Article` est disponible).

— *Traitement* : chaque fonction trie sa liste et compte les comparaisons ; le programme teste sur les prix connus `[4, 1, 3, 2]` puis `[1, 2, 3, 4]`, puis génère des listes aléatoires de tailles 50, 100 et 200 avec une graine fixée.

— *Sorties* : pour chaque petit test, le nombre de comparaisons des deux tris ; pour chaque grande taille, une ligne `n = ... : insertion ... comparaisons ; selection ... comparaisons`.

— *Contraintes* : chaque fonction renvoie un couple `(liste_triee, nb_comparaisons)` ; on ne compte que les comparaisons *entre deux prix* (les tests de borne ne comptent pas) ; une graine fixée (`rd.seed(1)`) garantit des listes reproductibles ; les deux tris doivent travailler sur des *copies* de la même liste.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi le tri par sélection effectue-t-il *toujours* le même nombre de comparaisons, quelle que soit la liste ? (rappelle-toi la question B1)

— Pourquoi le tri par insertion en fait-il beaucoup moins sur une liste déjà triée ?

— Comment faire pour que les deux tris comparent exactement les mêmes listes ? (deux moyens : copies et graine)

— Sans l'exécuter, prévois l'ordre de grandeur du nombre de comparaisons du tri par sélection pour $n = 200$: environ combien ?

Point de validation

Sur la liste aux prix connus `[4, 1, 3, 2]`, les deux tris doivent afficher `6` comparaisons chacun ; sur la liste déjà triée `[1, 2, 3, 4]`, l'insertion doit afficher `3` comparaisons et la sélection `6`.

Partie C — Exercice de réflexion (à finir à la maison)

À finir à la maison

Le catalogue complet de la boutique.

Contexte. Le gérant veut un programme unique qui gère tout son catalogue d'articles : générer les articles, afficher les prix extrêmes avec leur position, trier par prix croissant et décroissant, rechercher un prix, proposer la meilleure offre sous budget — et vérifier les coûts des algorithmes.

Objectif. Écrire un programme qui enchaîne ces traitements en *réutilisant* les fonctions écrites dans les parties A et B. Le programme doit être bien organisé : les fonctions d'un côté, le programme principal (saisies et affichages) de l'autre.

Questions à se poser avant de coder.

— Quelles fonctions des parties A et B peut-on réutiliser telles quelles ? Y a-t-il des fonctions à écrire en plus (par exemple l'indice du plus cher) ?

— Comment rendre le catalogue reproductible d'un essai à l'autre ? (graine du générateur aléatoire)

— Quel ordre pour les traitements ? Pourquoi faut-il trier *avant* la recherche dichotomique ?

— Comment vérifier chaque fonction séparément avant de tout assembler ?

— Comment afficher le nombre de comparaisons de chaque tri pour un grand catalogue ?

Étapes suggérées.

1. Version 0 : générer le catalogue (nombre d'articles saisi) avec une graine fixée, puis l'afficher avec une boucle `for`.
2. Version 1 : afficher l'article le moins cher et l'article le plus cher, avec leur numéro et leur prix (et leur position dans la liste).
3. Version 2 : trier une *copie* du catalogue par prix croissant (insertion) et une autre par prix décroissant (sélection), puis afficher les prix triés.
4. Version 3 : rechercher un prix par dichotomie dans la liste triée, puis proposer la meilleure offre sous budget.
5. Bonus : compter les comparaisons des deux tris et comparer sur plusieurs tailles de catalogue (50, 100, 200, 500). Que remarques-tu sur l'ordre de grandeur ?

Contraintes. Graine fixée au début du programme ; prix entiers entre 1 et 1000 € ; chaque fonction testée séparément avant l'assemblage ; programme commenté ; les messages peuvent être écrits sans accents (ex. `coute`, `numero`) si ton éditeur pose problème.

Fin du TP — la partie C est à finir à la maison.