TP Chapitre 11 : Algorithmes de tri et dichotomie

Objectifs du TP

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

- écrire l'algorithme du minimum et la recherche naïve dans une liste ;
- trier une liste en place par insertion et par sélection ;
- justifier la terminaison des tris et décrire leur invariant de boucle ;
- écrire la recherche dichotomique dans un tableau trié et justifier sa terminaison ;
- comparer les coûts des algorithmes : linéaire, quadratique, logarithmique ;
- concevoir et écrire un programme complet à partir d'une spécification (entrées, traitement, sorties).

Consignes générales. 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. Les algorithmes de ce TP travaillent sur des listes et les modifient *en place* : après un tri, la liste d'origine est réordonnée, on n'en construit pas une nouvelle. 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 — Retrouver une dépense dans le relevé (recherche naïve)

Chaque mois, tu enregistres les montants de tes dépenses dans une liste. Tu veux savoir si une dépense d'un montant précis figure dans le relevé : c'est une *recherche*. La recherche naïve parcourt la liste du début à la fin et compare chaque montant à la valeur cherchée.

Compléter la fonction `recherche_naive` qui renvoie `True` si `cible` figure dans la liste `t` et `False` sinon.

```
def recherche_naive(t, cible):
    for montant in t:
        # A COMPLETER : si le montant est egal a la cible, renvoyer True
    # A COMPLETER : la valeur n'a pas ete trouvee, renvoyer False

releve = [12.5, 45.0, 89.9, 23.0, 150.0]
print(recherche_naive(releve, 89.9))
print(recherche_naive(releve, 30.0))
```

Point de validation

Vérifie ton programme : il doit afficher `True` puis `False`.

Exercice A2 — Trier les dépenses du mois (tri par insertion)

Pour préparer ton budget, tu veux ranger tes dépenses de la plus petite à la plus grande. Le **tri par insertion** glisse chaque élément à sa place dans la partie gauche déjà triée, en décalant vers la droite les éléments plus grands que lui — exactement comme on range des cartes dans sa main.

Compléter la fonction `tri_insertion` qui trie la liste `t` en place et la renvoie.

```
def tri_insertion(t):
    for i in range(1, len(t)):
        valeur = t[i]
        j = i - 1
        # A COMPLETER : decaler vers la droite les elements plus grands que
            valeur
        # A COMPLETER : placer valeur dans le trou laisse par le decalage
    return t

depenses = [45.0, 12.5, 89.9, 23.0]
print(tri_insertion(depenses))
```

Point de validation

Vérifie ton programme : il doit afficher `[12.5, 23.0, 45.0, 89.9]`.

Exercice A3 — Trier les factures du trimestre (tri par sélection)

Chaque trimestre, tu reçois des factures (électricité, internet, téléphone, transport) pour des montants différents. Le **tri par sélection** cherche à chaque étape le plus petit montant de la partie non triée et l'échange avec la première case de cette partie : la partie gauche est alors définitivement triée.

Compléter la fonction `tri_selection` qui trie la liste `t` en place et la renvoie.

```
def tri_selection(t):
    n = len(t)
    for i in range(n - 1):
        indice_min = i
        # A COMPLETER : chercher l'indice du plus petit element de t[i+1] a
            t[n-1]
        # A COMPLETER : echanger t[i] et t[indice_min]
    return t

factures = [78.5, 29.99, 120.0, 45.0]
print(tri_selection(factures))
```

Point de validation

Vérifie ton programme : il doit afficher `[29.99, 45.0, 78.5, 120.0]`.

Exercice A4 — Chercher un prix dans le catalogue soldé (recherche dichotomique)

Pendant les soldes, un magasin affiche son catalogue trié par prix croissant. Pour savoir si un prix précis figure dans le catalogue, on peut utiliser la **recherche dichotomique** : on compare la valeur cherchée avec l'élément du milieu de l'intervalle, et on ne garde que la moitié où elle peut se trouver. Cette méthode exige une liste *triée*.

Compléter la fonction `recherche_dichotomique` qui renvoie `True` si `val` figure dans la liste triée `t` et `False` sinon.

```
def recherche_dichotomique(t, val):
    debut = 0
    fin = len(t) - 1
    while debut <= fin:
        milieu = (debut + fin) // 2
        # A COMPLETER : si t[milieu] == val, renvoyer True
        # A COMPLETER : si val < t[milieu], chercher dans la moitie gauche
```

```
        # A COMPLETER : sinon, chercher dans la moitie droite
    return False

catalogue = [9.99, 14.5, 19.99, 25.0, 49.99, 89.0]
print(recherche_dichotomique(catalogue, 25.0))
print(recherche_dichotomique(catalogue, 30.0))
```

Point de validation

Vérifie ton programme : il doit afficher `True` puis `False`.

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 doit être trié (fonction `est_triee`)

Avant de lancer une recherche dichotomique dans un catalogue de prix, il faut vérifier que la liste est bien triée : la dichotomie ne fonctionne que sur une liste triée.

Énoncé. Écrire une fonction `est_triee(t)` qui renvoie `True` si la liste `t` est triée en ordre croissant et `False` sinon. Tester sur `[9.99, 14.5, 19.99]`, sur `[14.5, 9.99, 19.99]` et sur la liste vide `[]`.

Spécification.

— *Entrées* : une liste de nombres `t`.

— *Traitement* : parcourir les cases deux à deux ; dès qu'une case est plus grande que la suivante, la liste n'est pas triée.

— *Sorties* : `True` si `t` est triée en ordre croissant, `False` sinon.

— *Contraintes* : fonction sans `input` ni `print` (elle renvoie une valeur) ; la liste vide et la liste à un élément sont considérées comme triées.

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

— Combien de comparaisons adjacentes suffisent pour vérifier qu'une liste de n éléments est triée ? (on ne trie pas la liste)

— Pourquoi peut-on s'arrêter dès la première « chute » (une case plus grande que la suivante) ?

— Pourquoi la liste vide est-elle considérée comme triée ? Et la liste à un seul élément ?

— Pourquoi cette vérification est-elle indispensable avant une recherche dichotomique ?

Point de validation

Vérifie : `est_triee([9.99, 14.5, 19.99])` doit renvoyer `True` ; `est_triee([14.5, 9.99, 19.99])` doit renvoyer `False` ; `est_triee([])` doit renvoyer `True`.

Exercice B2 — Le budget mensuel trié (tri par insertion complet)

Tu saisis chaque mois tes dépenses (courses, transport, sorties…). Le programme doit les ranger de la plus petite à la plus grande, puis afficher le budget trié et le total.

Énoncé. Écrire un programme qui demande le nombre de dépenses, puis le montant de chaque dépense. Il mémorise les montants dans une liste, les trie par insertion en place grâce à une fonction `tri_insertion(t)`, puis affiche la liste triée et le total des dépenses.

Spécification.

— *Entrées* : un entier N (nombre de dépenses, $N \geq 1$), puis N montants (flottants, en euros).

— *Traitement* : mémoriser les montants dans une liste ; trier cette liste par insertion ; additionner les montants pour obtenir le total.

— *Sorties* : la liste triée du plus petit au plus grand, puis le total (ex. `Total : ... euros`).

— *Contraintes* : fonction `tri_insertion(t)` qui modifie la liste en place et la renvoie ; boucle `for` pour la saisie ; le tri ne construit pas de nouvelle liste.

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

— Pourquoi, contrairement à la caisse du chapitre 1, faut-il ici mémoriser les montants dans une liste avant de calculer le total ?

— Quel est le rôle de la variable `j` dans le tri par insertion ? Pourquoi la condition `j >= 0` est-elle indispensable ?

— Pourquoi mémoriser l'élément courant dans une variable `valeur` avant le décalage ?

— Comment vérifier que ton tri est correct ? (pense à tester une liste déjà triée, puis une liste à l'envers)

Point de validation

Vérifie : avec 4 dépenses de 45.0 euros, 12.5 euros, 89.9 euros et 23.0 euros, le programme doit afficher la liste triée `[12.5, 23.0, 45.0, 89.9]` et `Total : 170.4 euros`.

Exercice B3 — Le tri par sélection des factures, pas à pas (tri par sélection complet)

Les factures du trimestre (électricité, internet, téléphone, assurance) arrivent dans le désordre. Le programme doit les trier par sélection et afficher la liste après chaque étape, pour observer l'algorithme.

Énoncé. Écrire un programme qui demande le nombre de factures, puis le montant de chaque facture. Il les trie par sélection grâce à une fonction `tri_selection(t)`, en affichant l'état de la liste après chaque étape, puis affiche la liste triée finale.

Spécification.

— *Entrées* : un entier N (nombre de factures, $N \geq 1$), puis N montants (flottants, en euros).

— *Traitement* : mémoriser les montants dans une liste ; trier par sélection ; afficher la liste après chaque étape de la boucle externe.

— *Sorties* : l'état de la liste après chaque étape, puis la liste triée finale.

— *Contraintes* : fonction `tri_selection(t)` ; échange par affectation simultanée `t[i], t[indice_min] = t[indice_min], t[i]` ; l'affichage des étapes peut se faire dans la fonction ou dans le programme principal.

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

— Pourquoi mémorise-t-on l'*indice* du minimum et pas sa valeur ?

— Pourquoi l'échange se fait-il après la boucle interne, et pas pendant ?

— Que se passe-t-il si `indice_min == i` ? L'échange est-il un problème ?

— Combien de comparaisons le tri par sélection effectue-t-il pour 5 factures ? (et pour N fac-

tures ?)

Point de validation

Vérifie : avec 4 factures de 78.5 euros, 29.99 euros, 120.0 euros et 45.0 euros, la liste triée finale doit être `[29.99, 45.0, 78.5, 120.0]`.

Exercice B4 — Le relevé annuel trié : recherche dichotomique avec compteur (dichotomie complète)

Ta banque te fournit un relevé annuel trié par montant croissant. Tu veux savoir si une dépense d'un montant précis y figure, et *combien de comparaisons* la recherche a effectuées : c'est le coût de l'algorithme.

Énoncé. Écrire une fonction `recherche_dichotomique(t, val)` qui renvoie un p-uplet `(trouve, nb_comparaisons)` : `trouve` vaut `True` si `val` est dans la liste triée `t`, `False` sinon, et `nb_comparaisons` est le nombre de comparaisons effectuées. Puis écrire un programme qui demande le montant cherché, appelle la fonction sur le relevé `[12.5, 23.0, 45.0, 89.9, 150.0, 220.0]` et affiche le résultat et le nombre de comparaisons.

Spécification.

— *Entrées* : le montant cherché (flottant) ; la liste `releve` est fournie, déjà triée.

— *Traitement* : recherche dichotomique dans la liste triée ; compter chaque comparaison effectuée.

— *Sorties* : `True` ou `False`, puis le nombre de comparaisons (ex. `Nombre de comparaisons : 3`).

— *Contraintes* : boucle `while` ; compteur de comparaisons ; la fonction renvoie un p-uplet ; la liste est triée (précondition).

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

— Pourquoi la liste doit-elle être triée ? Que risquerait-on avec une liste non triée ?

— Quand la boucle s'arrête-t-elle ? Quelle condition garantit que la valeur n'est pas dans la liste ?

— Comment passer de l'intervalle `[debut, fin]` à la moitié concernée ? (deux cas à écrire en français)

— Pourquoi le nombre de comparaisons est-il si petit ? (6 éléments : au plus combien de comparaisons ?)

Point de validation

Vérifie avec le relevé `[12.5, 23.0, 45.0, 89.9, 150.0, 220.0]` : en cherchant 89.9, le programme doit afficher `True` puis `Nombre de comparaisons : 3` ; en cherchant 100.0, il doit afficher `False` puis `Nombre de comparaisons : 3`.

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

À finir à la maison

Le grand inventaire des dépenses : trier une fois, chercher souvent.

Contexte. Pendant une année, tu as relevé chaque jour le montant de tes dépenses (courses, transport, sorties, abonnements…). Tu disposes d'une liste de 1 000 dépenses, toutes comprises entre 1 euro et 200 euros. Tu veux pouvoir répondre rapidement à la question : « une dépense de tel montant exact figure-t-elle dans mon relevé ? »

Objectif. Écrire un programme qui génère la liste des 1 000 dépenses, vérifie qu'elle n'est pas triée, la trie par insertion ou par sélection (au choix), puis répond aux questions de recherche : la dépense figure-t-elle dans le relevé ? Combien de comparaisons chaque recherche coûte-t-elle ? Vaut-il mieux chercher naïvement ou trier d'abord ?

Questions à se poser avant de coder.

— Comment générer 1 000 montants réalistes entre 1 et 200 euros ? (penser à `randint` et à une compréhension de liste)

— Comment vérifier qu'une liste est triée ? (la fonction de l'exercice B1 est réutilisable telle quelle)

— Quel tri choisir : insertion ou sélection ? Quels sont leurs invariants et leurs coûts ? (quadratique dans le pire cas)

— Comment compter les comparaisons d'une recherche ? (reprendre l'idée de l'exercice B4)

— Pourquoi la dichotomie exige-t-elle une liste triée ? Que coûte le tri par rapport à la recherche ?

Étapes suggérées.

1. Version 0 : générer la liste des 1 000 dépenses et afficher les 10 premières ; vérifier avec `est_triee` que la liste n'est pas triée.
2. Version 1 : trier la liste par insertion ou par sélection (en place) ; vérifier ensuite avec `est_triee` qu'elle est triée.
3. Version 2 : recherche naïve d'un montant donné, avec compteur de comparaisons. Tester avec un montant présent et un montant absent.
4. Version 3 : recherche dichotomique sur la liste triée, avec compteur de comparaisons. Comparer avec la version 2.
5. Bonus : lancer 100 recherches. Estimer le nombre total de comparaisons avec la recherche naïve (100 × 1 000 au pire) et avec la dichotomie (100 × environ 10). Que conclure sur l'intérêt de trier une fois, puis de chercher par dichotomie ?

Contraintes. Données crédibles (dépenses entre 1 et 200 euros) ; programme testé et commenté ; on peut limiter l'affichage aux 10 premières dépenses et aux bilans ; les messages peuvent être écrits sans accents (ex. `depenses`, `comparaisons`) si ton éditeur pose problème.

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