

Pour chaque exercice des exemples de code, structures de données et tests sont fournis en Annexe.

Exercice 1 : Listes (40min)

Exercice 1.1 (10min) – Calculez la moyenne d’une liste de données. Cette liste vous est fournie sous forme textuelle¹ et la moyenne portera sur un attribut fourni en paramètre (dans le cas ci-dessous l’attribut sera "note" et le résultat est 16.).

```
nom;prenom;filiere;note;absences
Dupond;Pierre;MP;19;7
Dupond;Jeanne;MP;19;5
Clavier;Christian;PSI;14;1
Gilles;Eric;PC;16;3
Arthaud;Nathalie;MP;15;0
```

Exercice 1.2 (10min) – Triez la liste de données par ordre croissant avec un simple *tri par sélection* donné en Annexe (qu’il faut adapter), et comparez votre tri avec la méthode `sorted`² de Python.

Exercice 1.3 (20min) – Détectez si il existe une sous-liste *continue* d’élèves dont la somme des "absence" est égale à k (ex : étant donnée la liste d’étudiant ci-dessus, pour $k = 6$ une sous-liste d’élèves a été trouvée entre les indices 1 et 2 (compris)). Proposez dans un premier temps une méthode naïve en $O(n)$ et testez avec la liste triée. Si vous avez le temps une méthode optimisée.

Exercice 2 : Piles, Files de priorité et Tas (1h20min)

Rappel. Les Piles permettent de manipuler des valeurs (ou objets) de manière séquentielle. Elles ont deux principales opérations : ajout (*push*) et enlève (*pop*) qui renvoie le dernier éléments ajouté.

Exercice 2.1 (10min) – Créez une File dont le comportement est inverse à une Pile lors du renvoi de valeur : on renvoi le premier élément ajouté et non le dernier ajouté (il faut donc modifier le *pop*).

Exercice 2.2 (10min) – Transformez votre File en File de Priorité, avec une méthode `renvoie` qui retourne le plus grand élément de la File et le supprime.

Exercice 2.3 (50min) – Proposez une optimisation du renvoi en utilisant un Tas (décrit en Annexe) et donnez sa complexité.

Exercice 2.4 (10min) – Utilisez le Tas afin de réaliser une méthode de tri par Tas et réalisez une comparaison empirique des performances entre ce tri et la méthode Python `sorted`.

¹Voici le fichier correspondant <https://gitlab.ec-lyon.fr/rvuille/inf-tc1/-/blob/master/TD02/code/etudiants.txt>

²<https://docs.python.org/3/howto/sorting.html>

Annexe : Exercice 1

Chargement d'un fichier

Voici le code³ permettant de charger une liste fournie sous format textuel où chaque ligne du fichier représente un fichier de données. Ici il s'agit d'un étudiant par ligne, avec ses notes, sauf la première ligne du fichier qui contient les informations sur l'organisation de chaque ligne (nom/prénom, filière, etc.).

```
data = []

with open("etudiants.txt") as f:
    keys = None
    for line in f:
        l = [w.strip() for w in line.split(';')]
        if keys is None:
            keys = l
        else:
            data.append({k:v for k, v in zip(keys, l)})

print(data)
```

Le format du fichier ci-dessus est dit CSV (Comma-Separated Values), où chaque information est séparée par un point-virgule.

Tri par sélection

Voici le code permettant un tris par selection en Python :

```
def selectionSort(l: list = []) -> list:
    """Tri par selection en ligne"""
    for i in range(0, len(l)):
        min = i
        for j in range(i+1, len(l)):
            if(l[j] < l[min]):
                min = j
        tmp = l[i]
        l[i] = l[min]
        l[min] = tmp

    return l
```

Voici le fonctionnement de la méthode `sorted` qui peut être utilisée avec des tris secondaires (en utilisant le paramètre `key`).

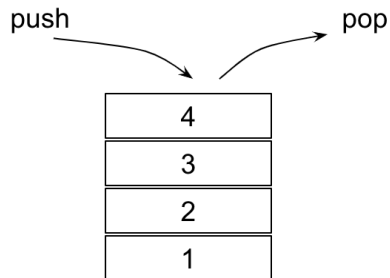
```
sorted(etudiants, reverse=True, key=lambda x: (x[1], x[0]))
```

Tests

Vous pouvez utiliser le fichier `etudiants-large100.txt` qui contient un grand nombre d'étudiant qu'il doit être possible de trier. Vous pourrez générer un grand jeu de données en utilisant le site <https://generatedata.com/>.

³Disponible ici <https://gitlab.ec-lyon.fr/rvuillem/inf-tc1/-/tree/master/TD02/code/load.py>

Annexe : Exercice 2 – Piles



```
class Pile():
    def __init__(self, values = []):
        self.__values = []
        for v in values:
            self.ajoute(v)

    def ajoute(self, v):
        self.__values.append(v)
        return v

    def supprime(self):
        v = self.__values.pop()
        return v

    def affiche(self):
        for v in self.__values:
            print(v)

    def taille(self):
        return len(self.__values)
```

Pour tester votre **Pile**, ajoutez les étudiants dans leur ordre d'apparition dans le fichier fourni dans l'exercice 1, et dépiler dans le bon ordre, comme par exemple :

```
>>> p = Pile()
>>> for d in data:
>>>     p.ajoute(d)
>>>
>>> e = p.supprime()
>>> print(e['nom'] + " " + e['prenom'])
```

Arthaud Nathalie

Annexe : Exercice 2 – Tas

Une structure de donnée de **Tas** permet de réduire la complexité de manipulation d'une file de priorité. En particulier une implémentation d'un **Tas en arbre binaire complet**. Cet arbre (dans sa configuration min-heap, où les données sont renvoyées par ordre croissant) satisfait la propriété suivante : la valeur de tout noeud est inférieure à celle de ses enfant.

Arbre binaire sous forme de tableau

Cet arbre binaire sera de surcroît implémenté en utilisant un tableau (car il s'agit d'un arbre *complet* où tous les niveaux sont remplis, sauf éventuellement le dernier). L'arbre binaire possède des noeuds ayant un index i , avec un fils gauche et un fils droit. Le tableau et l'arbre sont reliés de la façon suivante :

- La racine a la position $i = 0$ (cette valeur sera renvoyée par la fonction `get_racine`)
- Le parent a la position $\lfloor (i - 1) / 2 \rfloor$ (fonction `get_parent`)
- Le fils gauche a la position $2 \times i + 1$ (fonction `get_fils_gauche`)

- Le fils droit a la position $2 \times i + 2$ (fonction `get_fils_droit`)

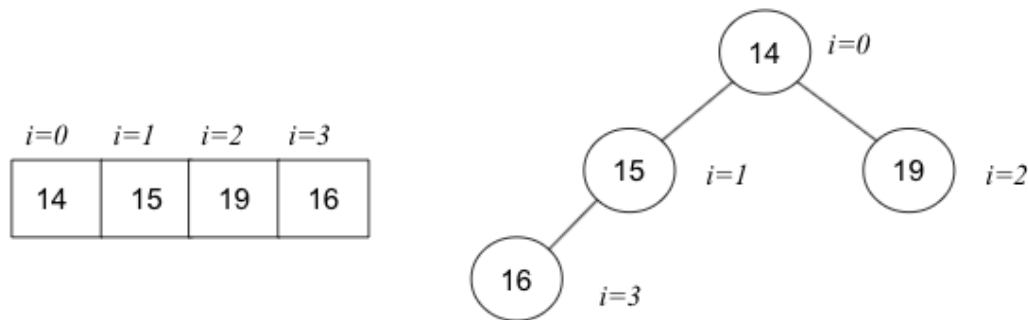


Figure 1: Exemple d'arbre binaire (droite) et son lien avec les index d'un tableau (gauche). L'arbre possède une propriété de Tas (la racine est inférieure à ses enfants).

L'ajout de données dans un Tas sera de complexité $O(\log(n))$ car il s'agira de mettre à jour l'arbre sur un chemin depuis la racine vers une feuille (autrement en parcourant la hauteur de l'arbre). De même pour la suppression. A noter qu'un autre avantage de l'utilisation des Tas avec l'implémentation ci-dessus est que la complexité spatiale est de $O(1)$ (ils ne nécessitent pas de place supplémentaire en mémoire, le tri est fait sur place).

Etapas afin de construire un Tas (min-heap)

1. Créez une structure de données de **Tas** similaire à une **File de Priorité** (avec les méthodes `ajoute` et `supprime`)
2. Créez des méthodes `get_racine`, `get_parent`, `get_fils_gauche`, `get_fils_droit` comme indiquées ci-dessus correspondant aux accès du tableau (qui stocke les valeurs de l'arbre) sans suppression.
3. Créez une méthode `insérer` (que l'on utilisera à la place d'`ajoute`) dont le principe est le suivant :
 - (a) Chaque nouveau noeud est rajouté comme dernier élément du tableau (à la fin donc)
 - (b) Comparez ce noeud à son parent et si il est plus grand que ce parent inversez-le
 - (c) Répétez tant que la condition ci-dessus est vraie et que la racine n'est pas atteinte
4. Créez une méthode `enlever` dont le principe est le suivant :
 - (a) Enlever l'élément racine de l'arbre (premier élément du tableau)
 - (b) Déplacer le dernier noeud de l'arbre (dernier élément du tableau) à la place de la racine de l'arbre
 - (c) Vérifier que la racine conserve la propriété de Tas (qu'elle est inférieure à ses enfants); si ce n'est pas le cas alors implémenter une méthode `descendre` qui :
 - i. prend le plus petit des enfants
 - ii. échange sa place avec lui si il est plus petit
 - iii. répète cela tant qu'il existe des enfants

Attention : pensez à tester si il existe un fils droit et un fils gauche lors des opération de descente lors de l'insertion.

Tests

Comparez votre implémentation des piles et files avec l'implémentation Python dans le module `queue`. Voici un exemple d'utilisation de ce module <https://docs.python.org/3/library/queue.html> :

```
import queue
pile = queue.LifoQueue()

for i in range(5): pile.put(i)

while not pile.empty():
    print(pile.get(), end=" ")
```

4 3 2 1 0

Comparez votre implémentation des Tas avec l'implémentation Python dans le module `heapq`. Voici un exemple d'utilisation de ce module <https://docs.python.org/fr/3.7/library/heapq.html> :

```
import heapq
tas = []

for i in range(5): heapq.heappush(tas, i)

while not len(tas) == 0:
    print(heapq.heappop(tas), end=" ")
```

0 1 2 3 4

Analyse empirique des tris

```
import time
import random
import matplotlib.pyplot as plt

nvalues = [100, 500, 1500, 2000, 2500, 3000]

timesSorted = []
timesSort = []

for i in nvalues:

    random.seed()
    p = 12**2 # Ordre de grandeur des valeurs
    liste = []

    for x in range(i): liste.append(random.randint(0, p))

    c = liste.copy()
    a=time.perf_counter()
```

```
triSorted = sorted(c)
b=time.perf_counter()
timesSorted.append(b-a)

c = liste.copy()
a=time.perf_counter()
triSort = c
triSort.sort()
b=time.perf_counter()
timesSort.append(b-a)

#plt.plot(nvalues, timesTriSelection, "r-", label="Python heap")
plt.plot(nvalues, timesSorted, "g-", label="Tri 1")
plt.plot(nvalues, timesSort, "b-", label="Tri 2")
plt.xlabel("Taille du jeu de données")
plt.ylabel("Temps")
plt.legend(loc="upper left")
plt.title("Comparaison des performances des algorithmes de tri")
plt.show()
```

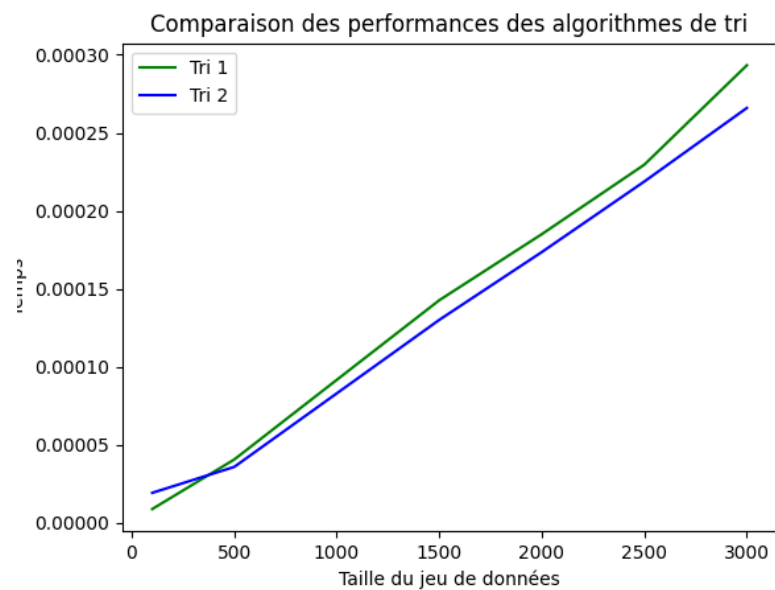


Figure 2: Exemple de résultat d'analyse des tris.