

Exercice 1 : Révisions

1. Soit **tab** un tableau de taille n , trié jusqu'au rang $j - 1$ ($j < n$). Écrire la fonction récursive **insérer**(**tab**: list, **j**: int) → None qui place l'élément de rang j dans la partie triée du tableau.
2. Écrire la fonction **tri_insertion**(**tab**: list) → None qui trie **tab** en utilisant la fonction **insérer**.
3. Construire par compréhension un tableau de 20 entiers compris entre 0 et 100.
4. Tester la fonction de tri sur le tableau.

Exercice 2 : Tri stable

Un tri est stable quand les éléments de même valeur garde leur place relative.

```
1 t = [(1, 5), (3, 4), (1, 1), (2, 9), (1, 2)]
2 # On décide de trier t par rapport au premier entier de chaque tuple
3 t = [(1, 5), (1, 1), (1, 2), (2, 9), (3, 4)]
4 # les deux premiers tuples gardent leur place relative.
```

Code 1 – Tri stable

1. Dérouler le tri par insertion de l'exercice précédent sur le tableau du code 1. Le tri par insertion semble-t-il stable ?
2. Même question pour le tri fusion.
3. Écrire la fonction **tri_selection**(**tab**: list) → None.
4. Montrer grâce à un exemple que le tri par sélection n'est pas stable.

Exercice 3 : Dichotomie

1. Écrire la fonction impérative **dichotomie_imp**(**tab**: int, **x**: int) → int de la recherche dichotomique, qui renvoie la position de **x** dans **tab** ou -1 s'il n'est pas présent.
2. Écrire la version récursive **dichotomie_rec**(**tab**: list, **x**: int, **debut**: int, **fin**: int) → int
3. Tester les deux fonctions sur un tableau de 50 entiers aléatoires.
4. Déterminer la complexité de l'algorithme de recherche dichotomique.

Exercice 4 : La fonction **mystere** implémente un algorithme de type *diviser pour régner*.

```
1 def mystere(tab: list, debut: int, fin: int) -> int:
2     if debut == (fin-1):
3         return tab[debut]
4     else:
5         milieu = (debut + fin)//2
6         gauche = mystere(tab, debut, milieu)
7         droite = mystere(tab, milieu, fin)
8         if (gauche > droite):
9             return gauche
10        else:
11            return droite
```

Soit la liste :

```
1 tab = [5, 71, 23, 45, 28, 89, 63, 39]
```

1. Dessiner l'arbre des séparations engendré par la fonction sur la liste *tab*.
2. Dessiner l'arbre des recombinaisons. Quelle valeur renvoie l'appel `mystere(tab)` ?
3. Que fait cette fonction ?
4. Discuter de la complexité de la fonction.

Exercice 5 : Le tri rapide est un autre exemple d'algorithme utilisant la méthode *diviser pour régner*. C'est un algorithme naturellement récursif qui peut se décrire ainsi :

- Choisir un élément pivot.
 - Sélectionner tous les éléments inférieurs au pivot.
 - Sélectionner tous les éléments supérieurs ou égaux au pivot.
 - Placer récursivement à gauche du pivot les éléments inférieurs à ce-dernier et à droite les éléments supérieurs.
1. Écrire la fonction récursive `tri_rapide(tab : list) → list` qui implémente l'algorithme du tri rapide et renvoie un tableau trié. Le premier élément du tableau sera choisi comme pivot. Les éléments inférieurs au pivot seront stockés dans un tableau **petit** et ceux supérieurs dans **grand**.
 2. Construire en compréhension une liste `t` de vingt éléments compris entre 0 et 100 et tester la fonction de tri.

La fonction précédente crée deux nouveaux tableaux à chaque appel récursif. Le coût en mémoire peut s'avérer important. Un tri en place serait plus efficace.

3. Écrire la fonction `partitionner(tab: list, deb: int, fin: int) → None` qui étudie les éléments de `tab` de l'indice `deb` (inclus) à celui d'indice `fin` exclus. La fonction positionne tous les éléments inférieurs à `tab[deb]` avant ce-dernier.
4. Écrire la fonction récursive `tri_rapide(tab: list, deb: int, fin: int) → None` qui trie le tableau en place. La fonction utilisera `partitionner`.

Remarque

Le tri rapide a une complexité en $O(n \times \log_2(n))$.