TP 6 INFO Algorithmes de tri

Partie I Généralités

Dans ce TP, nous allons étudier différents algorithmes de tri et leurs propriétés. Un algorithme de tri permet de trier les objets d'un ensemble muni d'un ordre total (comme des entiers avec l'ordre classique ou des mots avec l'ordre alphabétique).

Les algorithmes de tri sont fondamentaux dans de nombreuses applications de notre quotidien. Par exemple, disposer de données triées permet d'effectuer rapidement des recherches en utilisant l'algorithme de recherche dichotomique.

Un algorithme de tri, appliqué à un tableau `tab` $= [t_0, \dots, t_{n-1}]$ contenant des éléments d'un ensemble muni d'un ordre total, est dit :

- **comparatif** s'il n'utilise que des comparaisons et des permutations entre deux éléments du tableau `tab` ;
- **stable** s'il conserve la position relative dans le tableau `tab` des quantités qui sont égales ;
- **en place** s'il peut s'effectuer directement dans la structure de données initiale `tab` et ne nécessite pas l'utilisation d'une nouvelle structure (i.e. sans coût en mémoire supplémentaire).

Partie II Tri par sélection

II.A - Description de l'algorithme

On considère un tableau $[t_0, \dots, t_n]$. Le principe du tri par sélection est le suivant :

- on recherche le plus petit élément dans le tableau $[t_0, \dots, t_n]$ et on l'échange avec l'élément d'indice 0 ;
- on recherche le plus petit élément dans le tableau $[t_1, \dots, t_n]$ et on l'échange avec l'élément d'indice 1 ;
- etc.
- on recherche le plus petit élément dans le tableau $[t_{n-2}, t_{n-1}]$ et on l'échange avec l'élément d'indice $n-2$.

Question 1 : Écrire une fonction `minimum(tab:list, deb:int) -> int` prenant en entrée une liste d'entiers `tab` et un entier `deb` et qui renvoie le plus petit indice `k` $\geqslant$ `deb` tel que `tab[k]` est le minimum de `tab[deb:]`.

Question 2 : Écrire une fonction `tri_selection(tab:list) -> None` prenant en entrée une liste d'entiers `tab` et qui trie le tableau `tab` en utilisant l'algorithme de tri par sélection.

II.B - Propriétés de cet algorithme de tri

Question 3 : L'algorithme de tri par sélection est-il comparatif ? stable ? en place ?

Question 4 : Montrer la correction totale de la fonction `tri_selection`.

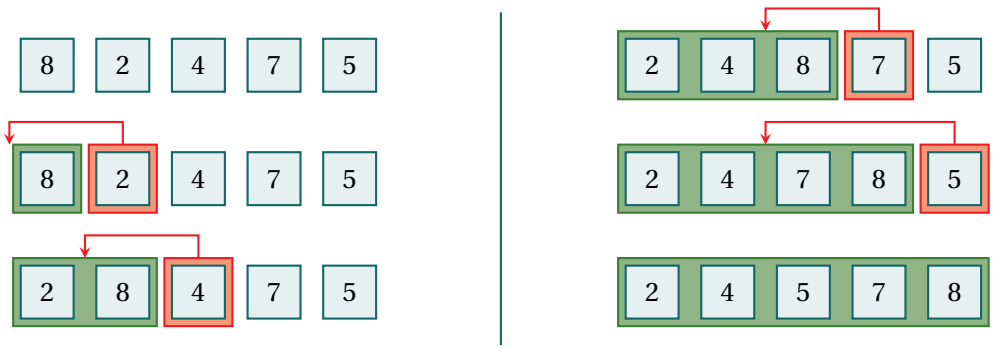
Question 5 : Déterminer la complexité temporelle dans le pire des cas de votre fonction `tri_selection`.

Partie III Tri par insertion

III.A - Description de l'algorithme

Le tri par insertion est celui utilisé naturellement par la plupart des personnes pour trier des cartes à jouer. L'algorithme consiste à parcourir le tableau en insérant à chaque étape l'élément d'indice j dans la partie du tableau déjà triée à sa gauche.

L'application de l'algorithme au tableau `tab = [8, 2, 4, 7, 5]` peut se représenter de la manière suivante.



En pratique, lorsqu'on souhaite insérer une nouvelle valeur dans la partie triée, on décale vers la droite les éléments supérieurs à celui que l'on souhaite insérer. Il suffit ensuite de placer notre élément à l'espace libéré.

Question 6 : Écrire une fonction `tri_insertion(tab:list) -> None` prenant en entrée une liste d'entiers `tab` et qui trie le tableau `tab` en utilisant l'algorithme de tri par insertion.

III.B - Propriétés de cet algorithme de tri

Question 7 : L'algorithme de tri par insertion est-il comparatif? stable? en place?

Question 8 : Montrer la correction totale de la fonction `tri_insertion`.

Question 9 : Déterminer la complexité temporelle dans le pire des cas de votre fonction `tri_insertion`.

Partie IV Tri fusion

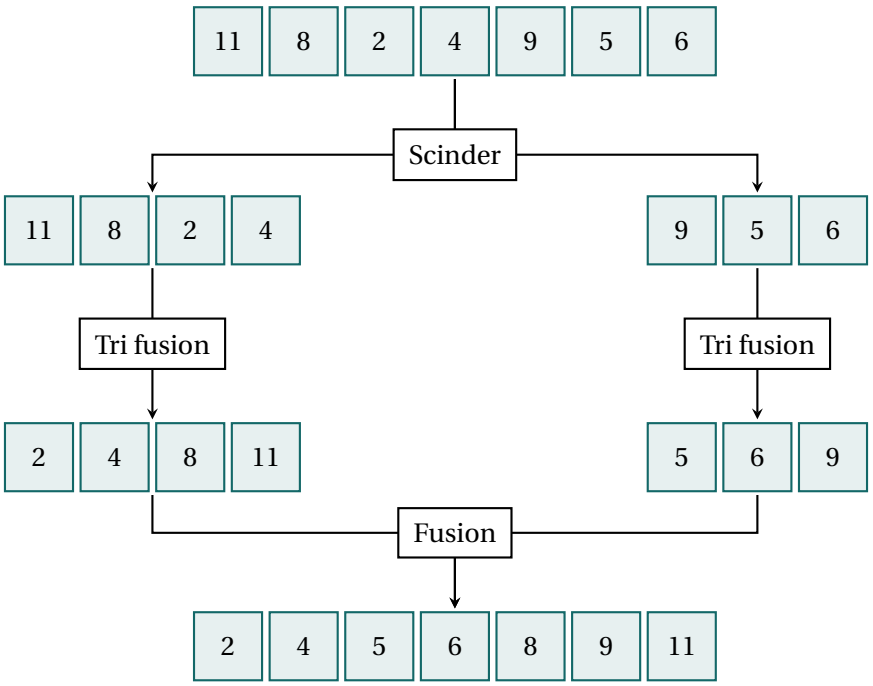
IV.A - Description de l'algorithme

Dans cette partie, on étudie l'algorithme de tri fusion qui s'appuie sur la méthode diviser pour régner.

Le tri fusion est un algorithme récursif que l'on peut décrire de la manière suivante.

1) Si le tableau est vide ou n'a qu'un élément : il est déjà trié, donc on ne fait rien ; sinon on scinde le tableau en deux parties approximativement de même taille.
2) On trie récursivement chacune des deux parties avec l'algorithme de tri fusion.
3) On fusionne les deux tableaux triés en un seul tableau trié.

L'application de l'algorithme au tableau `tab = [11, 8, 2, 4, 9, 5, 6]` peut se représenter de la manière suivante.



Question 10 : Écrire une fonction `fusionner(tab1:list, tab2:list) -> list` qui prend en argument deux listes triées d'entiers `tab1` et `tab2` et renvoie une liste triée contenant les mêmes éléments que `tab1` et `tab2`. La complexité de votre fonction doit être linéaire en `len(tab1)+len(tab2)`.

Question 11 : Écrire une fonction `tri_fusion(tab:list) -> None` qui trie la liste `tab` en utilisant l'algorithme de tri fusion.

IV.B - Propriétés de cet algorithme de tri

Question 12 : L'algorithme de tri fusion est-il comparatif ? stable ? en place ?

Pour étudier la complexité de la fonction `tri_fusion`, on se limite dans un souci de simplification au cas où la taille du tableau est de la forme $n = 2^p$ avec $p \in \mathbb{N}$. On note $C(p)$ la complexité temporelle dans le pire des cas de la fonction `tri_fusion` pour un tableau de taille $n = 2^p$.

Question 13 : Montrer pour tout $p \in \mathbb{N}^*$ que $C(p) = 2C(p-1) + O\big(2^p\big)$.

Question 14 : En déduire une expression explicite de $C(p)$ pour tout $p \in \mathbb{N}$.

Question 15 : En déduire que $C(p) = O\big(n\log(n)\big)$ où $n = 2^p$.

Remarques 1 :

a) On peut démontrer que la complexité temporelle dans le pire des cas du tri fusion est $O\big(n\log(n)\big)$ où n est la taille du tableau en entrée.

b) On peut également démontrer que la complexité temporelle dans le pire des cas d'un tri comparatif ne peut pas être meilleure que $O\big(n\log(n)\big)$.

c) Un tri comparatif de complexité temporelle dans le pire des cas en $O\big(n\log(n)\big)$ est qualifié d'optimal.

IV.C - Une application des tris optimaux

On souhaite déterminer l'écart minimal entre deux éléments distincts d'un tableau d'entiers. Par exemple, si on considère le tableau `tab = [15, 2, 28, 18, 22]`, alors l'écart minimal est $3 = |15 - 18|$.

On peut par exemple utiliser la fonction définie ci-dessous.

```
def ecart_minimal(tab:list) -> int:
    long = len(tab)
    ecart_min = abs(tab[1]-tab[0])

    for i in range(long):
        for j in range(i+1, long):
            ecart = abs(tab[i]-tab[j])
            if ecart < ecart_min:
                ecart_min = ecart

    return(ecart_min)
```

Question 16 : Déterminer la complexité temporelle de la fonction `ecart_minimal`.

Question 17 : En utilisant la fonction `tri_fusion`, écrire une fonction `ecart_minimal_bis(tab:list) -> int` plus efficace en termes de complexité temporelle et effectuant la même tache que la fonction `ecart_minimal`.

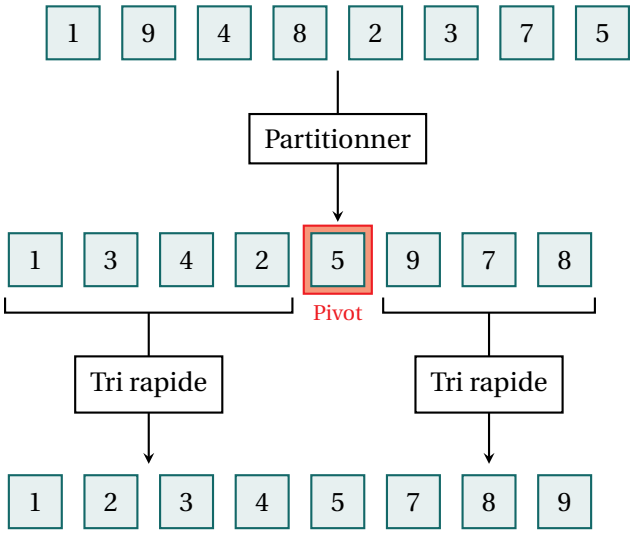
Partie V Tri rapide

V.A - Description de l'algorithme

Dans cette partie, on étudie l'algorithme de tri rapide qui s'appuie sur la méthode diviser pour régner.

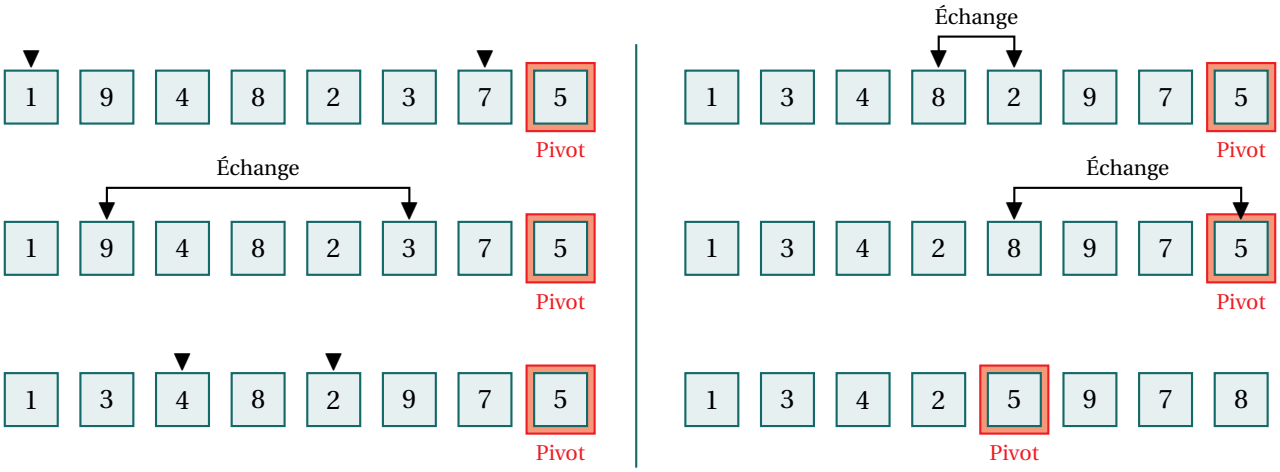
Le tri rapide est un algorithme récursif que l'on peut décrire de la manière suivante.

1) Si le tableau est vide ou n'a qu'un élément : il est déjà trié, donc on ne fait rien ; sinon on choisit le dernier élément du tableau en pivot.
2) Tous les éléments inférieurs ou égaux au pivot sont placés au début du tableau, puis le pivot est placé à la fin de ces éléments.
3) On trie récursivement avec l'algorithme de tri rapide la partie se trouvant avant le pivot et celle se trouvant après le pivot.



Plusieurs méthodes sont possibles pour réaliser l'étape « partitionner ». Par exemple, on peut utiliser deux curseurs : un placé au début du tableau qui avance et un autre placé à la fin (avant le pivot) qui recule. Lorsque le premier pointe un élément plus grand que le pivot et le second un élément plus petit que le pivot, on échange les deux éléments. On s'arrête lorsque les deux curseurs se croisent.

Par exemple, l'application de l'algorithme au tableau `tab = [1, 9, 4, 8, 2, 3, 7, 5]` peut se représenter de la manière suivante.



Pour pouvoir implémenter l'algorithme de tri rapide en place, nous allons rajouter des paramètres `deb` et `fin` : l'étape « partitionner » ne se fera dans le tableau `tab` que sur les éléments d'indice `i` vérifiant `deb` $\leqslant$ `i` < `fin`.

Question 18 : Écrire une fonction `partitionner(tab:list, deb:int, fin:int) -> int` qui modifie la liste `tab` en utilisant l'algorithme décrit ci-dessus et renvoie l'indice de la position finale du pivot dans la liste.

Question 19 : Écrire une fonction `tri_rapide(tab:list) -> None` qui trie la liste `tab` en utilisant l'algorithme de tri rapide.

V.B - Propriétés de cet algorithme de tri

Question 20 : L'algorithme de tri par rapide est-il comparatif? stable? en place?

Question 21 : Déterminer la complexité temporelle dans le pire des cas de votre fonction `tri_rapide`.

Remarque 2 : On peut démontrer que la complexité temporelle en moyenne du tri rapide est $O\big(n\log(n)\big)$.

Partie VI Tri par comptage

VI.A - Description de l'algorithme

Le tri par comptage (ou tri casier) est un algorithme de tri par dénombrement qui s'applique à un tableau `tab` d'entiers.

On peut décrire l'algorithme de tri par comptage de la manière suivante.

1) On détermine le minimum `mini` et le maximum `maxi` du tableau `tab`.
2) On construit un tableau auxiliaire `aux` de taille `maxi-mini+1` de sorte que :
   - l'élément `aux[0]` compte le nombre d'occurrences de `mini` dans le tableau `tab`;
   - l'élément `aux[1]` compte le nombre d'occurrences de `mini+1` dans le tableau `tab`;
   - etc.
   - l'élément `aux[maxi-mini]` compte le nombre d'occurrences de `maxi` dans le tableau `tab`.
3) Le tableau `aux` permet de construire une version triée du tableau `tab`.

Question 22 : Écrire une fonction `tri_comptage(tab:list) -> None` qui trie la liste `tab` en utilisant l'algorithme de tri par comptage.

VI.B - Propriétés de cet algorithme de tri

Question 23 : L'algorithme de tri par comptage est-il comparatif?

Question 24 : Déterminer la complexité temporelle dans le pire des cas de votre fonction `tri_comptage`.

Question 25 : Donner une situation où cet algorithme est peu efficace.