TP 7
Algorithmes de tri

L'objet de ce TP est de comprendre et implémenter quelques algorithmes de tri, en particulier les algorithmes de tri par insertion et tri à bulles déjà vus en première année, puis l'algorithme de tri rapide (quicksort).

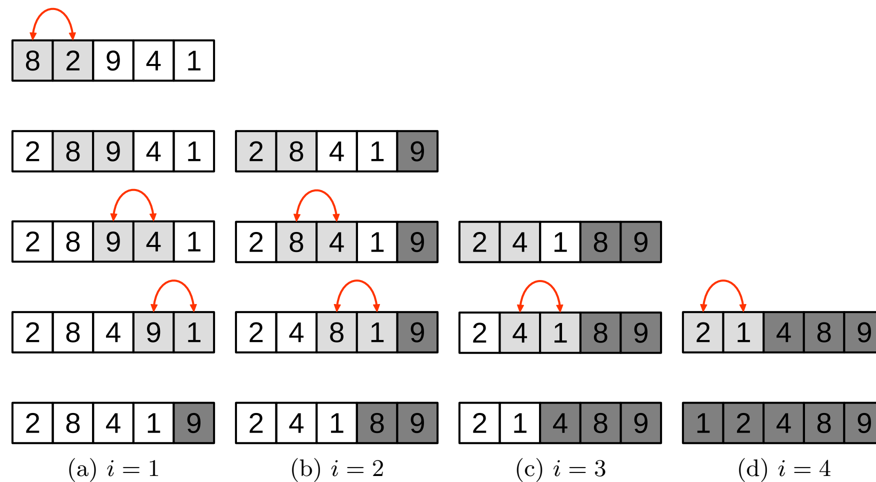
1 Tri à bulles

Le tri à bulles est un algorithme de tri classique. Son principe est simple, et il est très facile à implémenter.

On considère un tableau de nombres `T`, de taille N. L'algorithme parcourt le tableau, et dès que deux éléments consécutifs ne sont pas ordonnés, les échange. Après un premier passage, on voit que le plus grand élément se situe bien en fin de tableau. On peut donc recommencer un tel passage, en s'arrêtant à l'avant-dernier élément, et ainsi de suite. Au i-ème passage on fait remonter le i-ème plus grand élément du tableau à sa position définitive, un peu à la manière de bulles qu'on ferait remonter à la surface d'un liquide, d'où le nom d'algorithme de tri à bulles.

On donne ci-dessous l'algorithme de tri à bulles en *pseudo-code*.

```
1: procedure TRI_BULLES(T)
2:     N ← taille de T
3:     pour i de N − 1 à 1 faire
4:         pour j de 0 à i − 1 faire
5:             si T[j] > T[j + 1] alors
6:                 échanger T[j] et T[j + 1]
7:             fin si
8:         fin pour
9:     fin pour
10: fin procedure
```



(a) $i = 1$ (b) $i = 2$ (c) $i = 3$ (d) $i = 4$

FIGURE 1 – Exemple d'exécution de l'algorithme de tri à bulles. Les cases gris clair représentent les éléments comparées, les flèches rouges les échanges d'éléments, et les case gris sombre les éléments placés définitivement.

L'algorithme de tri à bulles a une complexité en temps en $O(N^2)$ en pire cas, où N est la taille du tableau. Le pire cas correspond ici au cas où le tableau est initialement trié par ordre décroissant : dans ce cas l'algorithme doit faire remonter chaque élément jusqu'à la i-ème place à chaque étape i, en effectuant à chaque fois un échange. La complexité en temps indique ici que le temps pris par l'algorithme pour trier un tableau de taille N augmente quadratiquement en fonction de N lorsque N est très grand.

On peut parler également de complexité en temps *en moyenne*, qui correspond pour simplifier au temps que prend l'algorithme pour un tableau de taille N tiré au hasard. Ici, la complexité en moyenne du tri à bulles est également en $O(N^2)$, ce qui le rend peu efficace comparé à d'autres algorithmes de tri comme le tri fusion ou le tri rapide. Le tri à bulle est en fait un des algorithmes de tri les plus lents, il est donc rarement utilisé en pratique.

Q1 Appliquer l'algorithme de tri à bulles « à la main » au tableau ci-dessous, à la manière de la figure 1.

2	1	6	9	8	4

Q2 Écrire une fonction `est_trie(T)` retournant `True` ou `False` selon que le tableau `T` est trié ou non.

Q3 Écrire une fonction `tri_bulles(T)` triant le tableau `T` par l'algorithme de tri à bulles. Puis utiliser la fonction `est_trie` pour vérifier que l'implémentation est correcte, en générant aléatoirement des tableaux de nombres.

Remarque : la fonction `tri_bulles` *doit modifier directement le tableau* `T`*, elle ne retourne aucune valeur.*

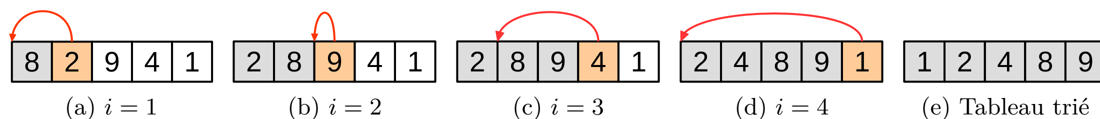
2 Tri par insertion

Le tri par insertion est également un algorithme de tri classique, simple à implémenter et intuitif, puisqu'il est celui que les joueurs de cartes utilisent naturellement pour trier leurs cartes.

On considère un tableau de nombres `T` de taille N qu'il s'agit de trier par ordre croissant. Le principe de l'algorithme est le suivant. On parcourt le tableau du début à la fin ($i = 1$ à $N - 1$), et à l'étape i, on considère que les éléments de 0 à $i - 1$ du tableau sont déjà triés. On va alors placer le i-ème élément à sa bonne place parmi les éléments précédents du tableau, en le faisant « redescendre » jusqu'à atteindre un élément qui lui est inférieur.

On donne ci-dessous l'algorithme de tri par insertion en *pseudo-code*.

```
1:  procedure TRI_INSERTION(T)
2:      N ← taille de T
3:      pour i de 1 à N − 1 faire
4:          x ← T[i]
5:          j ← i
6:          tant que j > 0 et T[j − 1] > x faire
7:              T[j − 1] ← T[j]
8:              j ← j − 1
9:          fin tant que
10:         T[j] ← x
11:     fin pour
12: fin procedure
```



(a) $i = 1$ (b) $i = 2$ (c) $i = 3$ (d) $i = 4$ (e) Tableau trié

FIGURE 2 – Exemple d'exécution de l'algorithme de tri par insertion. Les cases gris clair représentent les éléments déjà triés, et la case orange l'élément à placer à la bonne position parmi les éléments précédents.

L'algorithme de tri par insertion a une complexité en temps en $O(N^2)$ en pire cas et en moyenne, où N est la taille du tableau. Le pire cas correspond au cas où le tableau est initialement trié par ordre décroissant : dans ce cas l'algorithme doit faire redescendre jusqu'au début le i-ème élément du tableau, à chaque étape i. Le tri par insertion est donc également en moyenne moins efficace que d'autres algorithmes de tri, mais il est tout de même rapide lorsque le tableau considéré est déjà « presque » trié, ou de petite taille.

Q4 Appliquer l'algorithme de tri par insertion « à la main » au tableau ci-dessous, à la manière de la figure 2.

2	1	6	9	8	4

Q5 Écrire une fonction `tri_insertion(T)` triant le tableau `T` par l'algorithme de tri par insertion, puis vérifier que celle-ci fonctionne comme attendu.

3 Tri rapide version facile

L'algorithme de tri rapide (*quicksort*) a été inventé par Charles Antony Richard Hoare (C.A.R. Hoare) en 1961, et est basé sur le principe « diviser pour régner ». Ce principe, souvent utilisé en algorithmique, consiste à diviser le problème à résoudre en sous-problèmes, eux-mêmes divisés à leur tour, et ainsi de suite, jusqu'à arriver à des problèmes simples, dont la résolution permet d'obtenir simplement la solution du problème de départ.

On considère un tableau `T` de N nombres. On désigne le dernier élément du tableau comme *pivot*, et on va alors parcourir le tableau jusqu'à l'avant-dernier élément pour séparer les éléments strictements inférieurs au pivot et les éléments supérieurs au pivot, en créant deux nouveaux tableaux. Cette étape est appelee *partitionnement*. Une fois cette opération effectuée, on peut trier chacun des deux tableaux de manière *récursive* (la fonction de tri s'appelle elle-même, mais sur des tableaux strictement plus courts), puis reconstituer le tableau trié en concaténant le tableau trié des éléments plus petits que le pivot, le pivot, et le tableau trié des éléments plus grands que le pivot.

On voit que la fonction de tri va s'appeler elle-même sur des tableaux de plus en plus courts, jusqu'à arriver à des tableaux de 0 ou 1 élément, qui sont naturellement triés. C'est vraiment le principe « diviser pour régner ».

On donne donc ci-dessous l'algorithme de tri rapide en pseudo-code.

1: **procedure** TRI_RAPIDE(T)
2: $N \leftarrow longueur(T)$
3: **si** $N > 1$ **alors**
4: $pivot \leftarrow T[N-1]$
5: $Tinf, Tsup \leftarrow partitionner(T)$
6: $Ttrie \leftarrow$ concaténer $tri_rapide(Tinf)$, $pivot$ et $tri_rapide(Tsup)$
7: **retourner** $Ttrie$
8: **sinon**
9: **retourner** T
10: **fin si**
11: **fin procedure**

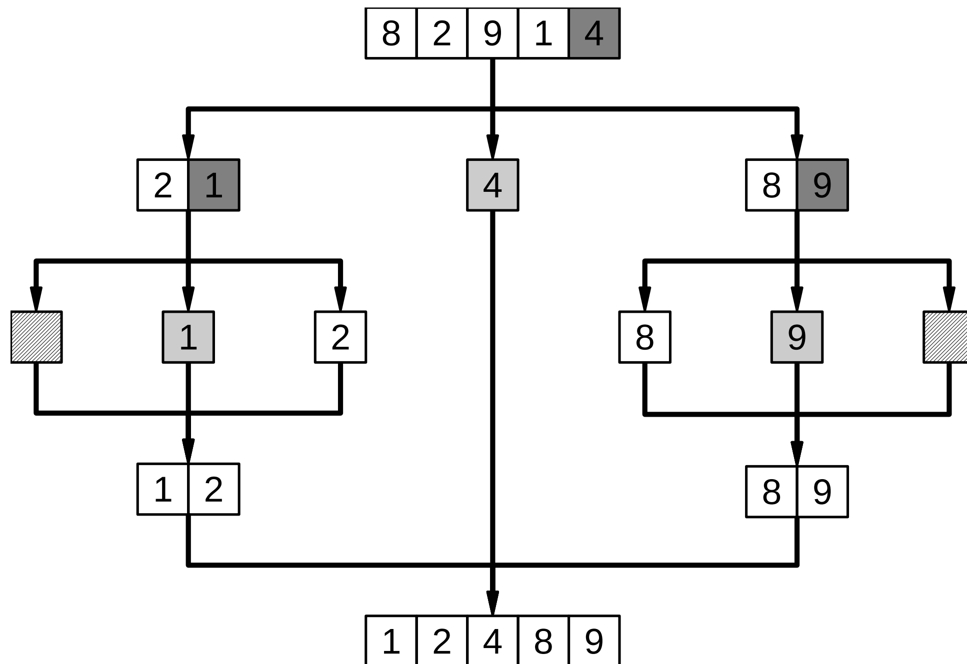
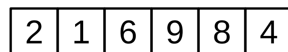


FIGURE 3 – Arbre représentant l'exécution de l'algorithme de tri rapide. En gris les pivots.

L'algorithme de tri rapide a une complexité en temps en pire cas en $O(N^2)$, lequel a lieu lorsque le tableau est trié par ordre décroissant, mais une complexité en moyenne en $O(Nlog(N))$, ce qui le rend nettement plus efficace que les deux algorithmes vus précédemment. Le tri rapide est donc largement utilisé en pratique.

Q6 Appliquer à la main un appel à la fonction `tri_rapide(T)` au tableau `T` ci-dessous, à la manière de la figure 3.



Q7 Écrire une fonction `partitionnement(T)` se servant du dernier élément de `T` comme pivot et retournant un couple de tableaux `(Tinf,Tsup)` tel que `Tinf` (respectivement `Tsup`) contienne les éléments de `T` strictement inférieurs (respectivement supérieurs ou égaux) au pivot, *en omettant le dernier élément.*

Q8 Écrire une fonction `tri_rapide(T)` implémentant l'algorithme de tri rapide et retournant le tableau `T` trié.

4 Tri rapide version « en place »

Dans la section précédente, la fonction de tri rapide créait de nouveaux tableaux, et retournait une copie triée de `T`, au lieu de modifier directement le tableau `T` sans utiliser davantage de mémoire, comme on le faisait pour les algorithmes de tri par insertion et tri à bulles. Autrement dit, l'algorithme n'opérait pas « en place ».

En pratique, on aime opérer en place. On se propose donc ici d'implémenter une version en place de l'algorithme de tri rapide. On procède de manière très similaire, à ceci près que la fonction de partitionnement ne crée pas deux nouveaux tableaux, mais modifie le tableau `T` de sorte que le pivot soit placé à sa place définitive, que tous les éléments à sa gauche lui soient inférieurs, et tous les éléments à sa droite lui soient supérieurs. Une fois cette opération effectuée, on peut la réappliquer au sous-tableau à gauche et au sous-tableau à droite du pivot, et ainsi de suite. Nécessairement, on finira par vouloir l'appliquer à un tableau de taille 0 ou 1, où il n'y a plus rien à faire. Au final le tableau est trié. Les fonctions de partitionnement et de tri prendront donc deux arguments supplémentaires, des entiers `debut` et `fin`, indiquant les indices de début et de fin du sous-tableau de `T` qu'il s'agit de modifier.

On donne ci-dessous l'algorithme de partitionnement en *pseudo-code*, qui prend en entrée un tableau `T`, un indice de début et un indice de fin, et qui partitionne le sous-tableau constitué par les éléments de `T` entre ces deux indices (le pivot est l'élément `T[fin]`), en retournant l'indice de la place définitive du pivot.

1: **procedure** PARTITIONNEMENT_EN_PLACE(T,$debut$,fin)
2: $\quad j \leftarrow debut$
3: $\quad$ **pour** i de $debut$ à $fin - 1$ **faire**
4: $\quad\quad$ **si** $T[i] \leq T[fin]$ **alors**
5: $\quad\quad\quad$ échanger $T[i]$ et $T[j]$
6: $\quad\quad\quad$ $j \leftarrow j + 1$
7: $\quad\quad$ **fin si**
8: $\quad$ **fin pour**
9: $\quad$ échanger $T[j]$ et $T[fin]$
10: $\quad$ **retourner** j
11: **fin procedure**

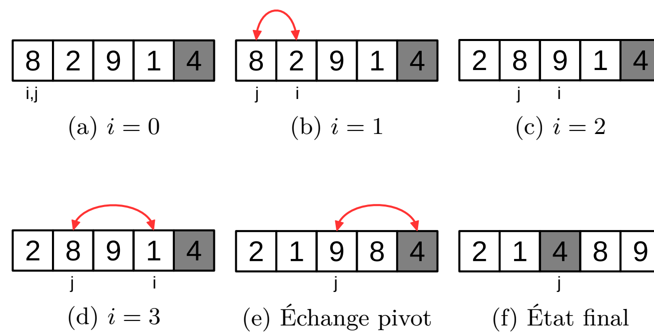


FIGURE 4 – Exemple d'exécution du partitionnement en place, pour $debut = 0$ et $fin = 4$. En-dessous du tableau on indique les valeur des indices i et j, en gris foncé le pivot, et les échanges d'éléments sont représentés par les flèches rouges.

Q9 Appliquer à la main un appel à la fonction `partitionnement_en_place(T,0,N-1)` au tableau `T` ci-dessous, où N est la taille de `T`, à la manière de la figure 4.

2	1	6	9	8	4

Q10 On donne ci-dessous en *pseudo-code* l'algorithme de tri rapide en place, qui utilise la fonction `partitionnement_en_place` définie précédemment. Comme dans sa version simple, la fonction s'appelle *elle-même*, elle est *récursive*.

```
1: procedure TRI_RAPIDE_EN_PLACE(T,debut,fin)
2:     si premier < dernier alors
3:         position_pivot ← partitionner(T, debut, fin)
4:         tri_rapide(T, debut, position_pivot − 1)
5:         tri_rapide(T, position_pivot + 1, fin)
6:     fin si
7: fin procedure
```

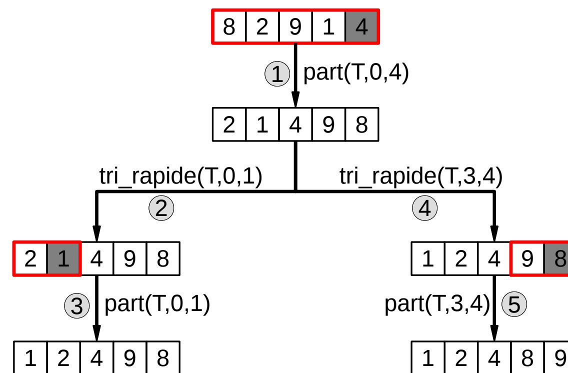


FIGURE 5 – Arbre représentant l'exécution d'un appel à la fonction de tri rapide en place. La fonction partitionnement a été abrégée en « part ».

Appliquer à la main un appel à la fonction `tri_rapide(T,0,5)` au tableau `T` de la question précédente, à la manière de la figure 5.

Q11 Implémenter la fonction `tri_rapide_en_place(T,debut,fin)` qui trie le sous-tableau de `T` donné par les indices `debut` et `fin` à l'aide de l'algorithme de tri rapide.

Remarque : Pour trier entièrement le tableau `T`*, il suffit d'appeler* `tri_rapide(T,0,len(T)-1)`.