3
Algorithmes de tri

Sommaire

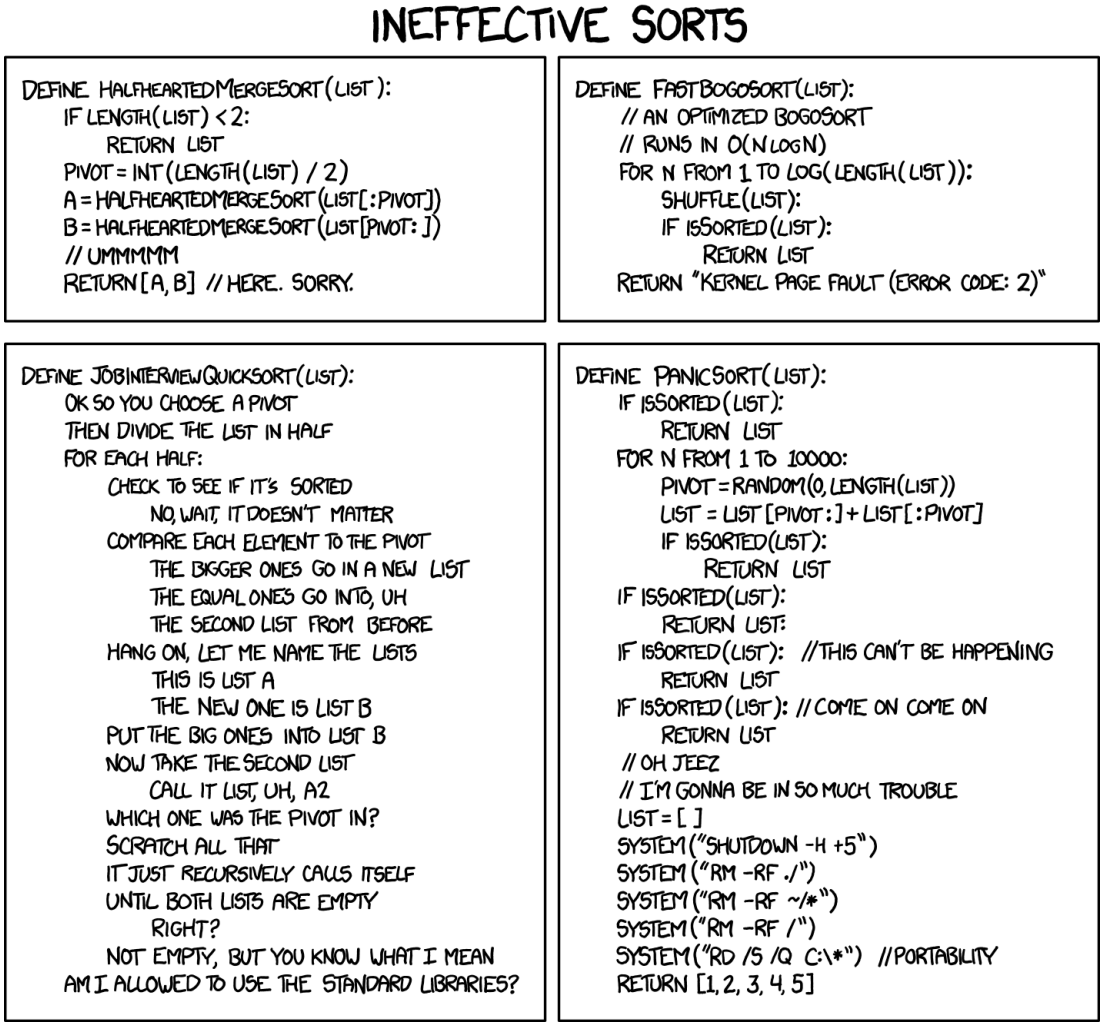


FIGURE 3.1 – From https ://xkcd.com/1185/

3.1 En guise d'introduction

Trier une collection de données c'est ranger ses éléments du plus petit au plus grand pour une relation d'ordre donnée, et les applications sont nombreuses :

(a) Recherche de fichiers sur un ordinateur : trier les fichiers par nom, taille, date de modification, etc.

(b) Organiser les contacts dans un téléphone portable : trier les contacts en fonction du nom ou du numéro de téléphone.

(c) Trier les produits dans une boutique en ligne : trier les produits en fonction de leur prix, de leur popularité, de leur date de sortie, etc.

(d) Organisation des tâches dans un gestionnaire de tâches : trier les tâches en fonction de leur date d'échéance, de leur priorité, etc.

(e) Tri des résultats de recherche sur un moteur de recherche : classer les résultats de recherche en fonction de leur pertinence, de leur date de publication, etc.

D'un pur point de vu algorithmique, trier des données peut être très utile avant de les traiter. Par exemple nous savons que rechercher un élément dans un tableau est un problème qui a un coût linéaire si le tableau est quelconque et un coût logarithmique si le tableau est trié. Donc si de nombreuses recherches sont à faire dans un tableau il peut être préférable de le trier au préalable.

Dans ce chapitre nous ramènerons le problème au tri d'ensembles de nombres, et pour se faire nous ne nous autoriserons que deux types d'opérations :

(a) comparer entre eux deux éléments de la liste.

(b) échanger la place de deux éléments de la liste.

Pour comparer deux algorithmes de tris entre eux, on ne tiendra compte que de ces deux types d'opérations, l'échange d'éléments correspondant à deux affectations.

Pour les calculs de complexité on supposera que ces deux types d'opérations s'effectuent en temps constant. Ce qui est presque le cas pour les listes en Python.

Plusieurs types de complexité semblent pertinents pour comparer deux algorithmes de tris. Nous n'en n'utiliserons que deux : celui au pire des cas et au meilleur des cas. Soit $\mathcal{A}$ un algorithme de tri.

(a) La complexité au pire des cas, qui fixe le maximum d'opérations nécessaires pour trier une liste de taille $n \in \mathbb{N}$:

$$T_{\mathcal{A},\text{pire}} = \max_{tab \text{ liste de taille } n} T(\mathcal{A}, tab),$$

où $T(\mathcal{A}, tab)$ est le nombre d'opérations effectuées pour trier la liste `tab` avec l'algorithme $\mathcal{A}$.

(b) La complexité au meilleur des cas :

$$T_{\mathcal{A},\text{meilleur}} = \min_{tab \text{ liste de taille } n} T(\mathcal{A}, tab).$$

(c) La complexité en moyenne. Son étude est beaucoup plus « complexe » en général et n'est pas au programme, elle présuppose l'existence d'une probabilité sur l'ensemble des suites de taille finie $n \in \mathbb{N}^*$ que l'on souhaite trier. Pour simplifier, supposons que nous ne souhaitons trier que des listes de la forme $[\sigma(0), \dots, \sigma(n-1)]$ où $\sigma \in \mathcal{S}_n$ est une permutation de l'ensemble $[\![0, n-1]\!]$. Alors :

$$T_{\mathcal{A},\text{moyenne}} = \frac{1}{n!} \sum_{\sigma \in \mathcal{S}_n} T(\mathcal{A}, \sigma),$$

où $T(\mathcal{A}, \sigma)$ est le nombre d'opérations effectuées pour trier la liste $[\sigma(0), \dots, \sigma(n-1)]$ avec l'algorithme $\mathcal{A}$.

Commençons par écrire une procédure d'échange, qui nous sera utile plus d'une fois.

Exercice 3.1.

Écrire une procédure `swap(tab : list, i : int, j : int)-> None` qui échange les places des éléments `tab[i]` et `tab[j]` dans la liste `tab`. Cette fonction ne retournera rien, elle modifiera la liste passée en argument. Quelle est sa complexité ?

Solution.

3.2 Premiers algorithmes de tri

3.2.1 Le tri par sélection du minimum

Algorithm 22: Selection Sort

```
1 def selection_sort(t):
      Data: t tab of numbers
      Result: t sorted
2     n ← len(t)
3     for i ← 0 to n − 2 do
4         i_min ← i
5         for j ← i + 1 to n − 1 do
6             if t[j] < t[i_min] then
7                 i_min ← j
8         t[i_min] ↔ t[i]
```

Le principe de cet algorithme est le suivant :

(a) déterminer le minimum de la liste `tab`,

(b) placer ce minimum au début de la liste,

(c) recommencer avec la liste `tab[1:]`.

Evidemment, on peut procéder par sélection du maximum, en le plaçant à la fin du tableau.

Assurons-nous que l'on sait déterminer l'indice du minimum d'un tableau `tab` et commençons par écrire une fonction `ind_min(tab : list)-> int`. Pour le trouver, on suppose que l'indice du minimum est `ind_tmp = 0`, puis on parcourt le tableau en comparant chaque élément à celui d'indice `ind_tmp` en le faisant évoluer s'il le faut.

Exercice 3.2.

Écrire une fonction `ind_min(tab : list)-> int` qui détermine l'indice du minimum de la liste `liste`. Quelle est sa complexité ?

Solution.

Terminaison : c'est une boucle `for`...

Correction : La correction de cet algorithme se démontre par récurrence en utilisant l'invariant de boucle :

« À l'entrée de la boucle d'indice i le terme d'indice `ind_min` est le minimum des $i+1$ éléments de la liste passée en argument. »

Maintenant on peut écrire la procédure `selection_sort` :

Algorithmes 16 (Tri par selection)**.**

```
def selection_sort(tab : list)-> None:
    n = len(tab)
    for i in range(n-1):
        i_min = i
        for j in range(i+1, n):
            if tab[j] < tab[i_min]:
                i_min = j
        swap(tab, i_min, i)
```

Terminaison : L'algorithme itératif se termine puisqu'il est constitué, au total, de deux boucles `for`.

Correction : Pour établir la correction de cette version itérative, on utilise l'invariant de boucle :

« À l'entrée dans la boucle d'indice i, la liste `tab[0:i]` est triée et pour tout $k \geq i$, `tab[i-1]` $\leq$ `tab[k]`. »

Observons comment notre algorithme trie une liste :

```
>>> from numpy.random import permutation
>>> tab = list(permutation(range(11)))
>>> select_sort(tab)
[1, 9, 4, 10, 8, 2, 3, 7, 5, 0, 6]
```

```
[0, 9, 4, 10, 8, 2, 3, 7, 5, 1, 6]
[0, 1, 4, 10, 8, 2, 3, 7, 5, 9, 6]
[0, 1, 2, 10, 8, 4, 3, 7, 5, 9, 6]
[0, 1, 2, 3, 8, 4, 10, 7, 5, 9, 6]
[0, 1, 2, 3, 4, 8, 10, 7, 5, 9, 6]
[0, 1, 2, 3, 4, 5, 10, 7, 8, 9, 6]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Complexité : Le graphique suivant suggère une complexité quadratique de notre algorithme.

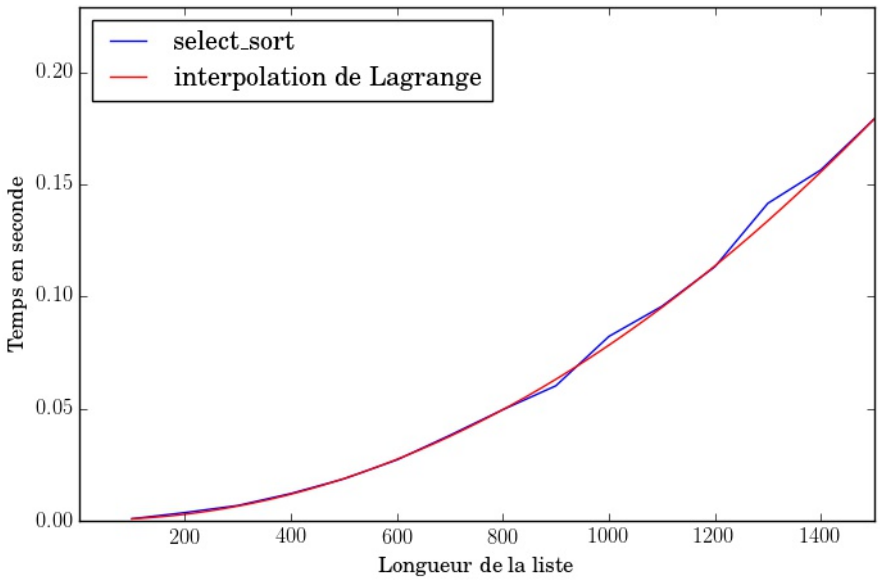


FIGURE 3.2 – Temps d'exécution de select_sort en fonction de la longueur de la liste.

Complexité : Nous allons compter séparément les comparaisons et les affectations. Commençons par remarquer que le nombre de comparaisons ne dépend pas de tableau mais seulement de sa taille. On en fait $n-i-1$ lors de la seconde boucle, donc au total :

$$\sum_{i=0}^{n-2} n-i-1 = \frac{n(n-1)}{2} \text{ comparaisons.}$$

(a) Déterminons la complexité au pire. Elle ne dépend finalement que du nombre d'échanges effectués sur la liste passée en argument. Dans le pire des cas on effectue un échange (soit deux affectations) à chaque tour de boucle `for`, soit $n-1$ échange, ou $2(n-1)$ affectations. Donc la complexité au pire est quadratique.

(b) Dans le meilleur des cas la liste est triée dans l'ordre croissant et on ne fait aucun échange. Donc la complexité au meilleur est quadratique.

Finalement la complexité est en $\Theta(n^2)$.

3.2.2 Le tri par insertion

Le tri par insertion et celui du joueur de carte. On insère chaque élément à sa place dans la partie de la liste déjà triée, comme un joueur de carte le ferait s'il avait toutes les cartes du jeu devant lui sur une table et qu'il insérait carte après carte dans sa main.

Algorithm 23: Insertion Sort(t)

```
1 def insertion_sort(t):
      Data: t tab of numbers
      Result: t sorted
2     n ← len(t)
3     for i ← 1 to n − 1 do
4         key ← t[i]
5         j ← i − 1
6         while j ≥ 0 and t[j] > key do
7             t[j + 1] ← t[j]
8             j ← j − 1
9         t[j + 1] ← key
```

On commence par écrire une procédure `insert(tab : list , i : int)`, qui insère l'élément `tab[i]` dans la partie `tab[0:i]` déjà triée de la liste `tab`.

Algorithmes 17 (Une fonction d'insertion).

```
def insert(tab : list, i : int)-> None:
    j = i - 1
    key = tab[i]
    while j >= 0 and tab[j] > key:
        tab[j+1] = tab[j]
        j -= 1
    tab[j+1] = key
```

Exercice 3.3.

Si `tab[0:i]` contient k éléments supérieurs à `tab[i]`. Combien d'affectations réalise la procédure pour insérer `tab[i]` ?

Solution.

Terminaison : j est un variant de boucle qui prouve la terminaison de la boucle `while`.

Correction : La boucle while a pour effet de déplacer d'un cran vers la droite tous les éléments d'indice strictement inférieur à `i` qui sont strictement supérieurs à `key`. On démontre alors sa correction à l'aide de l'invariant :

« Pour tout $j < k \leq i$, `tab[k] > key`. »

On peut maintenant écrire la procédure principale.

Algorithmes 18 (Tri par insertion)**.**

```
def insertion_sort(tab : list)-> None:
    range_prep = range(1,len(tab))
    for i in range_prep:
        insert(tab, i)
```

Terminaison : Le corps de la fonction est constitué d'une boucle `for`. La terminaison de la fonction `insert(tab, i)` ayant été démontrée, celle de la fonction principale est établie.

Correction : La correction de cet algorithme s'obtient en prouvant l'invariant :

« À l'entrée dans la boucle d'indice i la liste `tab[0:i]` est triée. »

Observons comment notre algorithme fonctionne :

```
>>> tab = list(permutation(range(6)))
>>> tab
[5, 0, 1, 3, 2, 4]
>>> insertion_sort(tab)
[0, 5, 1, 3, 2, 4]
[0, 1, 5, 3, 2, 4]
[0, 1, 3, 5, 2, 4]
[0, 1, 2, 3, 5, 4]
[0, 1, 2, 3, 4, 5]
```

Complexité : En ce qui concerne la complexité, elle semble quadratique comme le montre le graphique suivant : Déterminons séparément le nombre de comparaisons et d'affectations.

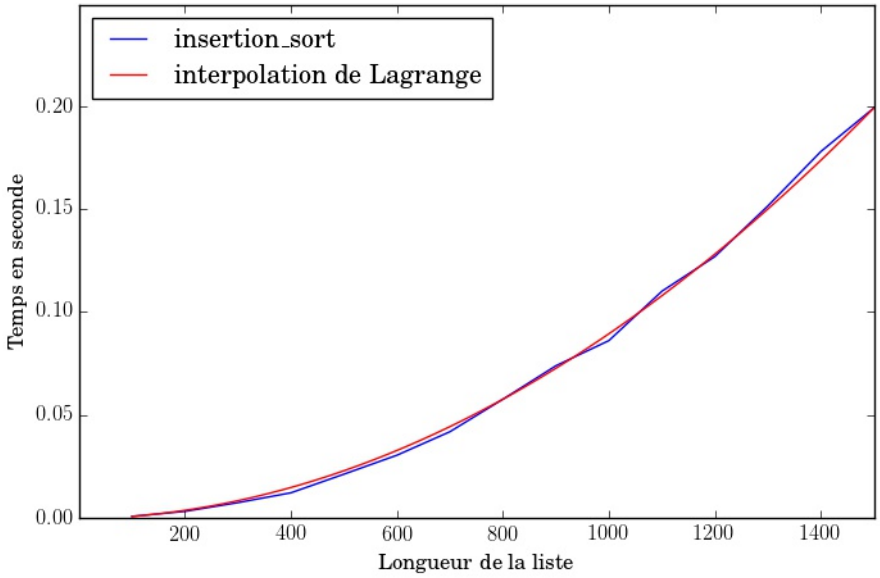


FIGURE 3.3 – Temps d'exécution de insertion_sort en fonction de la longueur de la liste.

(a) Dans le pire des cas, la liste passée en argument est triée dans l'ordre décroissant. Dans ce cas la boucle `while`, de la fonction `insert` s'arrête lorsque $j = 0$, donc elle est exécutée j fois, pour j comparaisons entre éléments de la liste et j affectations à la liste. Pour chaque tour de boucle `for` s'ajoute deux affectations. Au total :

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \text{ comparaisons et } \sum_{i=1}^{n-1}(i+2) = \frac{(n+1)(n+4)}{2} \text{ affectations.}$$

Donc la complexité au pire est bien quadratique.

(b) Dans le meilleur des cas, la liste est triée dans l'ordre croissant est la condition de la boucle `while` n'est jamais vérifiée. Donc pour chaque tour de boucle `for` on effectue une comparaison et deux affectations, donc au total $n-1$ et $2(n-1)$. La complexité au meilleur des cas est donc linéaire.

On peut remarquer qu'il y a une très grande différence entre la complexité au pire et au meilleur dans ce cas. La complexité de l'algorithme est en $O(n^2)$.

Ce tri n'est pas optimal[i] mais il est tout de même très utilisé. Il est établi de manière empirique qu'il est le plus rapide sur des listes « presque » triées, ou sur des listes de petite taille, le langage JAVA l'utilise d'ailleurs sur les listes de taille inférieure ou égale à 7.

i. Dans un sens que l'on précisera.

3.2.3 Le tri à bulles ou bubble sort

Algorithm 24: Bubble_sort

```
1 def bubble_sort(t):
      Data: t tab of numbers
      Result: t sorted
2     n ← len(t)
3     for i ← n − 1 to 1 do
4         for j ← 0 to i − 1 do
5             if t[j] > t[j + 1] then
6                 t[j] ↔ t[j + 1]
```

L'algorithme parcourt la liste et compare les éléments consécutifs. Si deux éléments consécutifs ne sont pas dans l'ordre, ils sont échangés. D'où son nom, les éléments les plus grands de la liste la « remontent », comme le font des bulles d'air dans un liquide.

Après un premier parcours complet de la liste, le plus grand élément est nécessairement à sa position définitive : en fin de liste. En effet, aussitôt que le plus grand élément est rencontré durant le parcours, il est mal trié par rapport à tous les éléments suivants, donc échangé à chaque fois jusqu'à la fin du parcours.

Après le premier parcours, le plus grand élément étant à sa position définitive, il n'a plus à être traité. Le reste du tableau est en revanche encore en désordre. Il faut donc le re-parcourir, en s'arrêtant à l'avant-dernier élément. Après ce deuxième parcours, les deux plus grands éléments sont à leur position définitive. Il faut donc répéter les parcours de la liste, jusqu'à ce que les deux plus petits éléments soient bien placés.

Algorithmes 19 (Bubble sort).

```
def bubble_sort(tab : list)-> None:
    n = len(tab)
    for i in range(n-1,0,-1):
        for j in range(i):
            if tab[j+1] < tab[j]:
                swap(tab, j, j+1)
```

On peut améliorer cet algorithme, en remarquant que si un parcours de la liste c'est fait sans qu'aucun échange n'ait été effectué c'est que la liste était déjà triée...

Algorithmes 20 (Bubble sort optimisé).

```
def bubble_sort2(tab : list)-> None:
    n = len(tab)
    is_sorted = False
    while not is_sorted:
        is_sorted = True
        for j in range(n-1):
            if tab[j+1] < tab[j]:
                swap(tab, j, j+1)
                is_sorted = False
        n -= 1
```

Terminaison : Pour la version optimisée, n est un variant de boucle pour la boucle `while`.

Correction : On considère l'invariant suivant pour la boucle `while` :

`tab[n:len(tab)]` est triée, et tous ses éléments sont plus grands que les éléments de la liste `tab[0 : n]`, et `is_sorted = False` ou `tab[0:n]` est triée.

Exercice 3.4.

Déterminer les complexités dans le pire et dans le meilleur des cas des versions optimisée et non optimisée du tri à bulles.

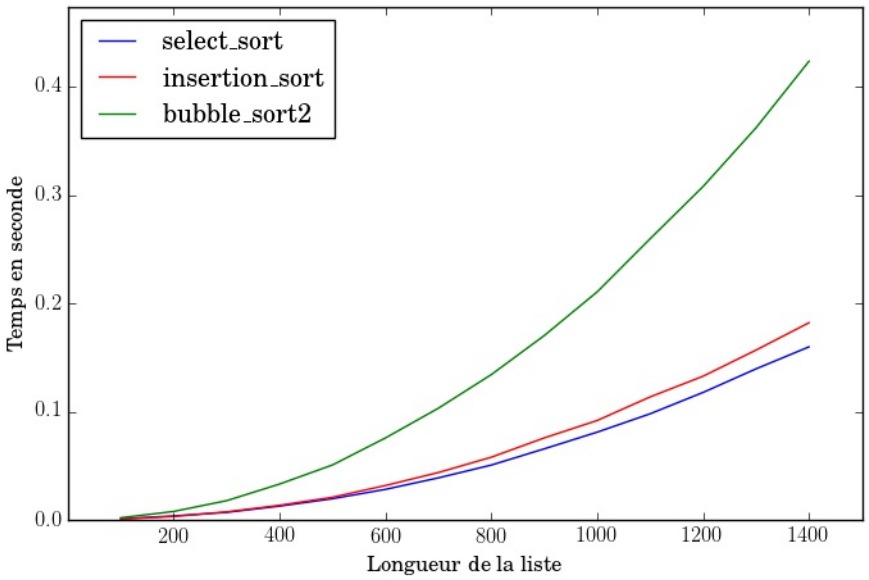
Le tri à bulles est le moins efficace des tris présentés ici. Il admet des variantes comme le tri cocktail (qui alterne le sens du parcourt de la liste), qui reste de complexité quadratique, le tri jump-down (qui ne compare pas des valeurs adjacentes mais une valeur à celle qui est à la place du plus grand), ou enfin le tri combsort qui lui est optimal.

3.2.4 Comparaisons des tris quadratiques

Nous avons établi les résultats suivants :

Tri	Opérations	Meilleur	Pire	Complexité
Sélection	Comparaisons	$n^2/2$	$n^2/2$	$\Theta(n^2)$
	Affectations	0	2n	
Insertion	Comparaisons	n	$n^2/2$	$O(n^2)$
	Affectations	2n	$n^2/2$	
Bulles	Comparaisons	n	$n^2/2$	$O(n^2)$
	Affectations	0	$n^2/2$	

Observons le graphique suivant. Il présente les temps nécessaires aux tris de mêmes listes de longueurs allant de 100 à 1400 par pas de 100 par nos trois algorithmes.



Il semblerait qu'en Python le tri par sélection soit légèrement plus efficace que le tri par insertion.

Or, on peut montrer que la complexité moyenne du tri par sélection est équivalente à $\frac{n^2}{2}$ alors que celle du tri par insertion est équivalente à $\frac{n^2}{4}$. Si en Python la tendance, pour de petites listes et de petites valeurs des données, s'inverse c'est certainement que le coût des affectations est supérieur à celui des comparaisons puisque le tri par sélection n'en fait au pire que de l'ordre de $2n$ alors que le tri par insertion en fait de l'ordre de $\frac{n^2}{2}$.

3.3 Propriétés des algorithmes de tris

3.3.1 Algorithmes de tris en place

Un algorithme de tri est dit en place si sa complexité spatiale est en $O(\ln(n))$.

Autrement, dit si la mémoire supplémentaire nécessaire à l'algorithme pour trier une liste de longueur n est au plus logarithmique en n. Certains auteurs demandent que l'algorithme ait une complexité spatiale en $O(1)$, ce qui est beaucoup trop restrictif comme nous le verrons plus loin.

Les trois tris que nous venons de voir sont des tris en place, la complexité spatiale de leur implémentation est en $O(1)$, puisque nous utilisons des listes, qui sont mutables, et que seul quelques indices sont nécessaires. Donc les trois sont des tris en place.

Cette propriété d'un tri dépend de son implémentation.

3.3.2 Tris stables

Considérons la liste de notes suivante :

Nom	Note
Sophie	15
Bob	17
Alice	15
Eve	10
Yoda	10

La figure **(3.4)** montre ce que l'on obtient si on utilise `selection_sort` et `insertion_sort` pour la trier par rapport à la colonne Note.

Tableau initial

Nom	Note
Sophie	15
Bob	17
Alice	15
Eve	10
Yoda	10

Après `selection_sort`

Nom	Note
Eve	10
Yoda	10
Alice	15
Sophie	15
Bob	17

Après `insertion_sort`

Nom	Note
Eve	10
Yoda	10
Sophie	15
Alice	15
Bob	17

FIGURE 3.4 – Exemple de tri stable et non stable

Quelle est la différence ? Et bien, dans le cas de `selection_sort` les étudiants qui ont la même note apparaissent dans la liste triée dans le même ordre qu'ils ont été rencontré dans la liste initiale, ce qui n'est pas le cas dans la liste triée par `insertion_sort`.

Un tri est dit stable s'il préserve l'ordre des éléments qui ont une valeur identique, il est dit instable sinon.

L'intérêt des tris stables apparait lorsque pour trier des données ayant plusieurs attributs on procède successivement en les triant attribut par attribut. Par exemple, si l'on souhaite que les étudiants ayant les mêmes notes apparaissent dans l'ordre alphabétique, dans notre tableau, il suffit de trier le tableau suivant la colonne **Nom** dans l'ordre lexicographique puis d'appliquer un tri stable sur la colonne **Note**.

3.4 Un exemple de tri sans comparaison : le tri par dénombrement

Le tri par dénombrement s'applique à des listes, ou tableaux, de nombres entiers majorés, strictement, par une constante entière borne ≥ 0. Son principe est simple, il suffit de parcourir la liste une fois et de dénombrer le nombre d'occurences de chaque valeur $i \in [\![0, \text{borne} - 1]\!]$, puis de reconstruire une liste triée où apparaissent ces valeurs autant de fois que nécessaire dans l'ordre croissant.

Exercice 3.5.

Écrire une première procédure `couting_sort(tab : list, borne : int)-> None` suivant cet algorithme.

Exercice 3.6.

Prouver la terminaison et la correction de cet algorithme.

Exercice 3.7.

Déterminer la complexité spatial et temporelle de cet algorithme.

Remarque : Si borne $= O(n)$, alors la complexité de cet algorithme est linéaire.

L'implémentation naïve de cette fonction n'est pas stable. Mais on peut facilement rendre `counting_sort` stable à peu de frais. L'idée, est qu'après avoir fait un premier parcourt de comptage, il faut construire le tableau des effectifs cumulés croissants qui nous donnera les positions d'insertion des nombres dans le tableau final, et de parcourir encore une fois notre tableau de la droite vers la gauche comme illustrée sur la figure **(3.5)**.

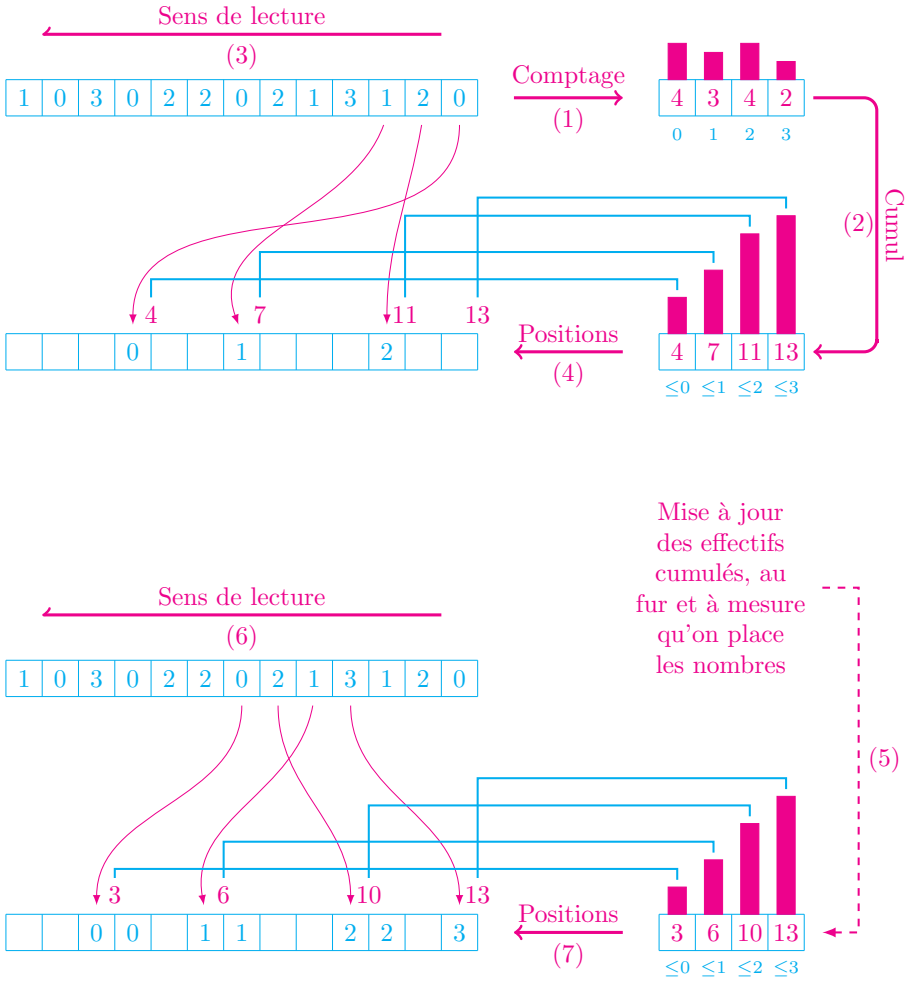


FIGURE 3.5 – Version stable du tri comptage.

Une implémentation de cette version de `counting_sort` est la suivante.

Algorithmes 21 (Tri par dénombrement version stable).

```
def counting_sort(t : list[int], borne: int)->list[int]:
      c = [0 for __ in range(borne)]
      n = len(t)
      b = [0 for __ in range(n)]
# n contient la longueur de la liste t
# c et b sont deux listes
# de longueurs borne et n
# initialisées avec des 0
      for i in range(len(t)):
            c[t[i]] += 1
# c[t[i]] contient le nombre d'occurences de l'élément t[i]
# dans le tableau t
      for i in range(1,borne):
            c[i] += c[i-1]
# c contient maintenant les effectifs cumulés croissants
# c[i] est le nombre d'éléments de t plus petits que i
# b ne contient que des 0
      for i in range(n-1,-1,-1):
 # en lisant t de la droite vers la gauche, c[t[i]] est le nombre
 # d'éléments de t plus petits que t[i]
            c[t[i]] -= 1
 # c[t[i]] est diminué de 1 car c[t[i]]-1 est le nombre de places
 # nécessaires pour stocker les éléments plus petits que t[i],
 # qui est placé dans b à la place c[t[i]]
            b[c[t[i]]] = t[i]
 # b contient les éléments de t dans l'ordre croissant
      return b
```

3.5 Exercices I

Exercice 3.8.

Exercice 3.9.

Écrire des versions récursives des algorithmes de tris par sélection du minimum et par insertion.

Exercice 3.10.

Comment peut-on rendre stable un tri instable ?

Exercice 3.11.

Bubble sort est-il stable ?

Exercice 3.12.

Exercice difficile à traiter en fin de deuxième année : Si l'on considère que toutes les permutations de $\mathcal{S}_n$ sont équiprobables, déterminer le nombre moyen d'échanges effectués par les algorithmes de tri par sélection et par insertion appliqués à une liste contenant les entiers de 1 à n.

3.6 Corrections

Corrigé 3.1 Le code :

```
def swap(tab : list, i : int, j : int)-> None:
    tab[i], tab[j] = tab[j], tab[i]
```

Sa complexité est en $\Theta(1)$.

Corrigé 3.2 Le code :

```
# Une vraie version algorithmique !
def ind_min(tab : list)-> int:
    ind_tmp = 0
    for i in range(1,len(tab)):
        if tab[i] < tab[ind_tmp]:
            ind_tmp = i
    return ind_tmp

# Une deuxième version
def ind_min2(tab : list)-> int:
    ind_tmp = 0
    range_prep = range(1,len(tab))
    for i, elt in enumerate(tab):
        if elt < tab[ind_tmp]:
            ind_tmp = i
    return ind_tmp

# En trichant
def ind_min3(tab : list)-> int:
    return liste.index(min(tab))
```

La complexité au pire de ces fonctions est en $\Theta(n)$.

Corrigé 3.3 La fonction décale simplement les k éléments et effectue deux autres affectations. Donc $k+2$ affectations en tout.

Corrigé 3.4 Pour le tri non optimisé, la complexité en temps est de $O(n^2)$, avec n la taille du tableau.

Pour le tri optimisé, le nombre d'itérations de la boucle externe est compris entre 1 et n. En effet, on peut démontrer qu'après la i-ème étape, les i derniers éléments du tableau sont à leur place. À chaque itération, il y a exactement $n-1$ comparaisons et au plus $n-1$ échanges.

(a) Le pire cas (n itérations) est atteint lorsque le plus petit élément est à la fin du tableau. La complexité est donc en $\Theta(n^2)$.

(b) Le meilleur cas (une seule itération) est atteint quand le tableau est déjà trié. Dans ce cas, la complexité est linéaire soit en $\Theta(n)$.

Corrigé 3.5 Le code :

```
def counting_sort(tab : list,borne : int)->None:
    nb_occurence = [0] * borne
    for elt in tab:
        nb_occurence[elt] += 1
    i = 0
    for j in range(borne):
        for k in range(nb_occurence[j]):
            tab[i] = j
            i += 1
```

Corrigé 3.6 Si les éléments de la liste `liste` passée en argument sont bien majorés strictement par `borne`, alors les boucles `for` ne produiront pas d'erreurs et se termineront normalement.

Dans ce cas la correction est évidente après la première boucle `for` la liste `nb_occurences` contient à la place i le nombre d'apparitions de i dans la liste `liste`. Puis on inscrit `nb_occurences[i]` fois le nombre i dans la liste `liste`.

Corrigé 3.7 La création de la liste `nb_occurences` est en $\Theta(n)$. Chaque valeur de cette liste va être inscrite autant de fois que nécessaire dans la liste `liste`, ce qui a un coût en $O(n + borne)$.
Finalement, la complexité temporelle est en $O(n + borne)$.
La complexité spatiale de l'algorithme est en $\Theta(borne)$, donc suivant les cas le tri est en place ou non.

Corrigé 3.9 **Tri par sélection :** L'usage de l'indice j peut paraitre artificiel, mais il est nécessaire pour ne pas avoir à calculer `len(tab)` plusieurs fois, ou de la passer en argument.

```
def ind_min(tab : list, i : int, j : int)-> int:
    if i == j-1:
        return i
    else:
        ind_min = i
        for k in range(i+1,j):
            if tab[k] < tab[ind_min]:
                ind_min = k
    return ind_min

def selection_sort_rec(tab : list)-> None:
    n = len(liste)
    __selection_sort_rec(liste,0, n)

def __selection_sort_rec(tab : list, i : int, j : int)-> None:
    if i == j:
        return None
    else:
        swap(liste, i, ind_min(liste, i,j))
        __select_sortion_rec(liste, i+1, j)
```

Tri par insertion :

```
def insert_rec(tab : list, j : int)-> None:
    if j > 0 and tab[j] < tab[j-1]:
        swap(tab, j, j-1)
        insert_rec(tab, j-1)

def insertion_sort_rec(tab , j=1):
    if j < len(tab):
        insert_rec(tab, j)
        insertion_sort_rec(tab, j+1)
```

Corrigé 3.10 Il suffit d'ajouter un index d'apparition en suffixe aux données puis de trier la liste et enfin de retirer cet index.

Tableau initial		Ajout de l'index		Après un tri		Suppression de l'index	
Nom	**Note**	**Nom**	**Note**	**Nom**	**Note**	**Nom**	**Note**
Sophie	15	Sophie	1500	Eve	1003	Eve	10
Bob	17	Bob	1701	Yoda	1004	Yoda	10
Alice	15	Alice	1502	Sophie	1500	Sophie	15
Eve	10	Eve	1003	Alice	1502	Alice	15
Yoda	10	Yoda	1004	Bob	1701	Bob	17

FIGURE 3.6 – Rendre stable un tri instable.

Corrigé 3.11 Oui, le bubble sort est stable.

Corrigé 3.12 **Tri par insertion :** Supposons avoir trouver les i_0 plus petits éléments de la liste et les avoir mis à leur place. La partie non triée du tableau est une permutation des éléments de $[\![i_0+1, n]\!]$, donc la probabilité, sachant que toutes les permutations sont équiprobables, que l'élément i_0+1 se trouve à sa place est :

$$\frac{\text{Card}\left(\{\sigma \in S_{n-i_0} \mid \sigma(i_0+1) = i_0+1\}\right)}{\text{Card}\left(\mathcal{S}_{n-j}\right)} = \frac{(n-i_0-1)!}{(n-i_0)!} = \frac{1}{n-i_0}$$

Si X_i désigne le nombre d'échanges à la i-ème étape du tri, alors X_i suit une loi de Bernouilli de paramètre $p = 1 - \dfrac{1}{n-i}$, Donc le nombre total d'échanges est $X = \sum_{i=2}^{n-2} X_i$ et son espérance est par linéarité :

$$E(X) = \sum_{i=0}^{n-2} E(X_i) = \sum_{i=0}^{n-2} 1 - \frac{1}{n-i} = n - 1 - \sum_{i=2}^{n} \frac{1}{i} = n - \ln(n) + \gamma + o(1).$$

Donc $T_{\text{select\_sort,moyenne}}(n) = O(n)$ pour ce qui est des échanges.

Tri par insertion : Soit $n \geq 2$, si $\sigma \in \mathcal{S}_n$ est une permutation, on appelle inversion de σ tout couple (i,j) tel que $i < j$ et $\sigma(i) > \sigma(j)$. Le nombre d'inversions d'une permutation σ est noté $I(\sigma)$.

Le nombre d'échanges nécessaires pour trier par insertion une liste correspondant à σ est donc $I(\sigma)$.

Notons $\tilde{\sigma}$ la permutation, dite « mirroir » de σ, définie par $\tilde{\sigma}(i) = \sigma(n-i+1)$. Un couple (i,j) est une inversion de σ si et seulement si ce n'est pas une inversion de $\tilde{\sigma}$, donc la somme des inversions de $I(\sigma) + I(\tilde{\sigma})$ est égal au nombre de couple (i,j), avec $i < j$, de l'ensemble $[\![1, n]\!]$, soit $\binom{n}{2}$.
Maintenant, si on partitionne $\mathcal{S}_n$, en deux sous-ensembles Σ et $\tilde{\Sigma}$ équipotents tels que $\sigma \in \Sigma \Leftrightarrow \tilde{\sigma} \in \tilde{\Sigma}$ et si on note $I(\sigma)$ le nombre d'inversions de σ on a donc :

$$\sum_{\sigma \in \mathcal{S}_n} I(\sigma) = \sum_{\sigma \in \Sigma} I(\sigma) + \sum_{\tilde{\sigma} \in \tilde{\Sigma}} I(\tilde{\sigma}) = \text{Card}\left(\Sigma\right) \times \binom{n}{2} = n! \, \frac{n(n-1)}{4}.$$

Le nombre moyen d'inversions est donc $\dfrac{n(n-1)}{4}$, et $T_{\text{insertion\_sort,moyenne}} = \dfrac{n(n-1)}{4} \sim \dfrac{n^2}{4}$, comme annoncé dans le cours.

3.7 Algorithmes efficaces de tris par comparaisons

3.7.1 Algorithme optimal de tri par comparaisons

Un arbre binaire de décision est un arbre qui permet de faire un choix parmi $n \in \mathbb{N}^*$ éléments d'un ensemble E de cardinal n.

Un tel arbre présente :

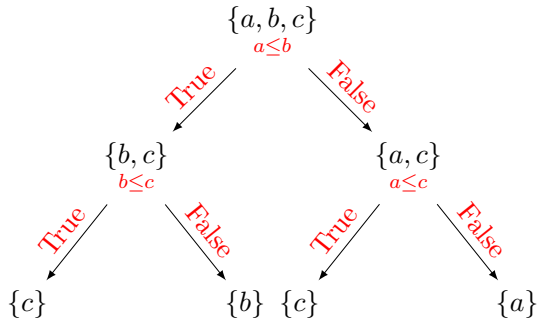
(a) l'ensemble E à sa racine.

(b) un singleton de E à chacune de ses feuilles.

(c) chaque noeud intermédiaire représente un sous-ensemble $E' \subset E$, et les deux branches qui en partent représentent des sous-ensembles stricts T et F de E' tels que $E' = T \cup F$.

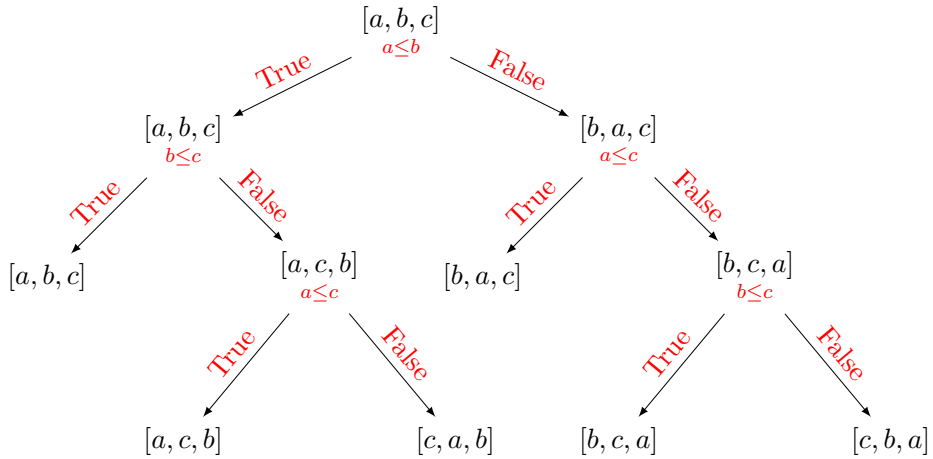
Les deux branches qui partent d'un noeud E' portent le résultat d'un test booléen appliqué à E', `True` pour T, à gauche, et `False` pour F à droite.

Par exemple, voici l'arbre de décision qui permet de choisir le maximum d'un ensemble $E = \{a, b, c\}$ de trois nombres.



Un arbre de décision permettant de faire un choix dans un ensemble à $n \in \mathbb{N}^*$ éléments doit avoir au moins n feuilles. Or, un arbre binaire de hauteur h ayant au plus 2^h feuilles, on a nécessairement $2^h \geq n$. À chaque algorithme de tri, par comparaison, on peut associer un arbre binaire de décision, sur l'ensemble des $n!$ permutations possibles des éléments du tableau (ou de la liste) à trier.

Par exemple voici l'arbre de décision correspondant au tri par insertion de la liste $[a, b, c]$.



La hauteur d'un arbre de décision associé à un algorithme de tri par comparaison a au moins $n!$ feuilles (chacune correspondant à un ordre), donc une une hauteur d'au moins $\log_2(n!)$, donc la complexité au

pire des cas d'un algorithme de tri par comparaison est au moins en $\log_2(n!)$.

On peut démontrer, avec un peu plus de travail, que la complexité en moyenne d'un algorithme de tri par comparaison est aussi supérieure ou égale à $\log(n!)$.

Exercice 3.13.

Démontrer que $\ln(n!) \underset{+\infty}{\sim} n\ln(n)$.

Solution.

Théorème 3.1 (Complexité optimale d'un algorithme de tri par comparaison)**.**

La complexité optimale d'un algorithme de tri par comparaison est dans le pire des cas (et en moyenne) au moins semi-linéaire. C'est-à-dire est en $\Omega(n\ln(n))$.

3.7.2 « Diviser pour mieux régner »

La stratégie « diviser pour mieux régner » repose sur les principes suivants :

(a) diviser le problème à résoudre en sous-problèmes de tailles inférieures, et les traiter récursivement.

(b) reconstituer la solution du problème à partir des solutions des sous-problèmes.

Les deux algorithmes de tris que nous allons présenter maintenant tirent profit de cette stratégie.

3.7.3 Le tri fusion ou merge sort

Nous avons déjà rencontré cet algorithme, dû à Von Neumann, dans les chapitres consacrés à la récursivité et la complexité. Pour rappel, son principe est simple :

- si le tableau n'a qu'un élément : il est trié.
- sinon :
  - (Diviser) diviser le tableau en deux parties à peu près égales.
  - (Régner) trier récursivement les deux parties de tableau par tri fusion.
  - (Combiner) fusionner les deux tableaux triés en un tableau trié.

Voici le pseudo-code d'un `merge_sort` en place.

Algorithm 25: Merge Sort

Data: t un tableau, i premier indice, k dernier indice
Result: t trié dans l'ordre croissant

```
1 def merge_sort(t, i, k):
2     if k − i > 1 then
3         j ← ⌊(i+k)/2⌋
4         merge_sort(t, i, j)
5         merge_sort(t, j, k)
6         merge(t, i, j, k)
```

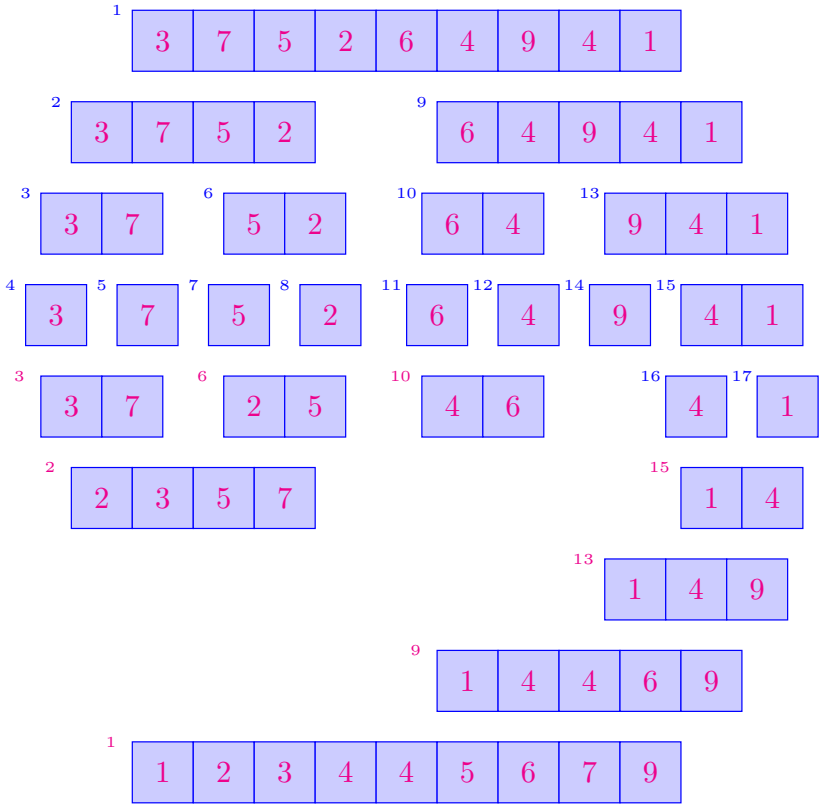


FIGURE 3.7 – Exemple de tri par tri fusion. Les numéros bleus correspondent aux appels récursifs et les rouges aux résultats de ces appels.

La procédure de fusion quant à elle peut s'écrire :

Algorithm 26: Merge

```
1  def merge(t, i, j, k):
2      ℓ ← i
3      r ← j
4      merged ← [ ]
5      for m ← i to k − 1 do
6          if r = k or (ℓ < j and t[ℓ] ≤ t[r]) then
7              merged ← merged + t[ℓ]
8              ℓ ← ℓ + 1
9          else
10             merged ← merged + t[r]
11             r ← r + 1
12     t[i..k − 1] ← merged
```

Une implémentation est donnée par les trois procédures suivantes.

Algorithmes 22 (Fusion des tableaux triés en un tableau trié).

```
def merge(tab : list, i : int, j : int, k : int )-> None:
    l = i
    r = j
    tmp = []
    for m in range(i,k):
        if  r == k or (l < j and tab[l] <= tab[r]):
            tmp.append(tab[l])
            l += 1
        else:
            tmp.append(tab[r])
            r += 1
    tab[i:k] = tmp
```

Terminaison : Si $i \leq k$ la boucle `for` se terminera sans lever d'exception.

Correction : La boucle admet comme invariant : La liste `B` est triée et est constituée des éléments des deux listes triées `tab[i:j]` et `tab[j:k]`.

Complexité : Si on pose $n = k - i$, alors la quatrième ligne s'effectue exactement n fois et les lignes 5 à 10 au plus n fois. Donc la complexité est en $\Theta(n)$.

Algorithmes 23 (Tri fusion).

```
def __merge_sort(tab : list, i : int, k : int)-> None:
    if k-i > 1:
        j = (i+k)//2
        __merge_sort(tab, i, j)
        __merge_sort(tab, j, k)
        merge(tab, i, j, k)
```

Algorithmes 24 (Tri fusion fonction appelante).

```
def merge_sort(tab)-> None:
    n = len(tab)
    __merge_sort(tab, 0, n)
```

Une autre solution plus pytonique, qui utilise `*args`.

Algorithmes 25 (Tri fusion version pytonique).

```
def merge_sort(tab : list, *args)-> None:
    if len(args) == 0:
        i, k = 0, len(tab)
    else:
        i, k = args
    if k-i > 1:
        j = (i+k)//2
        merge_sort(tab, i, j)
        merge_sort(tab, j, k)
        merge(tab, i, j, k)
```

Terminaison : La quantité $k-i$ diminue strictement à chaque appel et la fonction `merge` se termine.

Correction : On obtient la correction par récurrence forte. Pour $n = 0$ ou $n = 1$ c'est immédiat. Si c'est vrai pour les listes de tailles $m < n$ alors comme listes `tab[i:j]` et `tab[j:k]` sont de tailles strictement plus petites que n, elle sont triées après les appels `__merge_sort(tab, i, j)` et `__merge_sort(tab, j, k)` et comme la fonction `merge` est correcte la liste tab est triée.

Complexité : Nous avons déjà établi que la complexité du tri fusion est en $\Theta(n\ln(n))$
Ce tri est donc de complexité temporelle optimale. On pourrait faire un décompte plus précis des nombres d'affectations et de comparaisons dans le pire et le meilleur des cas, qui mènerait aux mêmes résultats.

3.7.4 Le tri rapide ou quick sort

Le tri rapide est une autre mise en oeuvre du principe de « diviser pour mieux régner ». Il consiste à choisir un élément `elt` dans la liste qu'on appelle pivot, puis à créer deux sous-listes constitués respectivement des éléments strictement inférieurs à `elt` et ceux supérieurs ou égaux. Enfin on tri chacune des sous-listes de manière récursive, avant de reconstitué une seule liste en intercalant le pivot.

Autrement dit, son principe est :

- si le tableau n'a qu'un élément : il est trié.
- sinon :
  - (Diviser) diviser les éléments du tableau en deux parties : ceux inférieurs et ceux supérieurs à un pivot choisi parmi les éléments du tableau et les répartir de part et d'autre de celui-ci.
  - (Régner) trier récursivement les deux parties de tableau par tri rapide.

En pseudo-code on a donc :

Algorithm 27: Quick Sort

```
1 def quick_sort(t, ℓ, r):
2     if r − ℓ > 1 then
3         p ← partition(t, ℓ, r)
4         quick_sort(t, ℓ, p)
5         quick_sort(t, p, r)
6
```

Pour partitionner la portion de la liste `tab` de ℓ à r :

i. on choisit $x = t[\ell]$ comme pivot.

ii. on initialise $i = \ell - 1$ et $j = r$

iii. on incrémente i de 1 jusqu'à ce que $t[i] \geq x$

iv. on décrémente j de 1 jusqu'à ce que $t[j] \leq x$

v. Si $j \leq i$ on retourne la position $i + (i == \ell)$. La quantité $i == \ell$ vaut 1 si $i = \ell$ et 0 sinon, elle sert à traiter le cas où $t[i]$ est le minimum de la tranche $t[i : r]$

Algorithm 28: Partition

```
1  def partition(t, ℓ, r):
2      x ← t[ℓ]
3      i ← ℓ − 1
4      j ← r
5      while True do
6          i ← i + 1
7          while t[i] < x do
8              i ← i + 1
9          j ← j − 1
10         while t[j] > x do
11             j ← j − 1
12         if j ≤ i then
13             return i + (i == ℓ)
14         t[i] ↔ t[j]
```

Terminaison : La boucle « infinie » de la ligne **5** se termine car à chaque passage dans les boucles des lignes **7** et **10** la valeur de $i - j$ diminue, c'est donc un variant de boucle.

Correction : L'algorithme peut se terminer exactement de deux manières :

— Soit $i = j$ et dans ce cas $t[i] = x$
— Soit $i = j + 1$.

Il n'est pas possible que i soit supérieur à $j + 1$, car toutes les valeurs à gauche de i sont inférieurs ou égales à x, donc la boucle de la ligne **10** qui fait diminuer j s'arrêtera dès qu'elle passera i.

On suppose que la tranche $t[l : r]$ contient au moins deux valeurs c'est-à-dire que $r - \ell \geq 2$.

Pour s'assurer que la valeur renvoyée p satisfait bien $\ell < p < r$, supposons que $p = \ell$, alors la ligne **7** n'a été exécutée qu'une seule fois, ce qui signifie que la ligne **10** s'exécutera jusqu'à $j = i = \ell$. Cependant, dans ce cas, la ligne **11** renverrait $i + 1$ donc pas ℓ.

Complexité : Posons $n = \ell - r$. Les seules comparaisons se font lignes **7** et **10**, il y en a soit $n+1$ soit $n+2$. Quant aux échanges d'éléments au mieux il n'y en a aucun, au pire n. Donc la complexité de cette fonction est en $\Theta(n)$.

Algorithmes 26 (Partitionement de la liste)**.**

```
def partition(tab, l, r):
    x = tab[l]
    i, j = l-1, r
    while True:
        i += 1
        while tab[i] < x:
            i += 1
        j -= 1
        while tab[j] > x:
            j -= 1
        if j <= i:
            return i + (i == l)
        tab[i], tab[j] = tab[j], tab[i]
```

Passons à la fonction de tri en elle-même :

Algorithmes 27 (Le tri rapide).

```
def quick_sort(tab : list, *args)-> None:
    if len(args) == 0:
        l, r = 0, len(tab)
    else:
        l, r = args
    if r-l > 1:
        p = partition(tab, l, r)
        quick_sort(tab, l, p)
```

Observons comment cette fonction trie une liste suivant ses appels à la fonction `partition`.

```
>>> tab = [3, 7, 1, 0, 9, 2, 4, 6, 8, 5]
>>> quick_sort(tab)
partition <- ([3, 7, 1, 0, 9, 2, 4, 6, 8, 5], 0, 10)
partition -> 3
partition <- ([2, 1, 0, 3, 9, 7, 4, 6, 8, 5], 0, 3)
partition -> 2
partition <- ([0, 1, 2, 3, 9, 7, 4, 6, 8, 5], 0, 2)
partition -> 0
partition <- ([0, 1, 2, 3, 9, 7, 4, 6, 8, 5], 1, 2)
partition -> 1
partition <- ([0, 1, 2, 3, 9, 7, 4, 6, 8, 5], 4, 10)
partition -> 9
partition <- ([0, 1, 2, 3, 5, 7, 4, 6, 8, 9], 4, 9)
partition -> 5
partition <- ([0, 1, 2, 3, 4, 5, 7, 6, 8, 9], 4, 5)
partition -> 4
partition <- ([0, 1, 2, 3, 4, 5, 7, 6, 8, 9], 6, 9)
partition -> 7
partition <- ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 6, 7)
partition -> 6
partition <- ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], 8, 9)
partition -> 8
>>> liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Complexité : Posons encore $n = r - \ell$. Notons $T(n)$ la complexité de `quick_sort` sur un tableau de longueur n. On a $T(1) = \Theta(1)$, et si $n > 1$ on a :

$$T(n) = \Theta(n) + T(m) + T(n-m),$$

où m est différents à chaque appel.

On peut traiter plusieurs cas :

— Si on a toujours $m = n//2$, alors dans ce cas on a :

$$T(n) = \begin{cases} \Theta(1) & \text{si} \quad n = 1 \\ \Theta(n) + 2T\left(n//2\right) & \text{sinon.} \end{cases}$$

On reconnait la complexité de `merge_sort`, donc dans ce cas $T(n) = \Theta(n\ln(n))$.
Ce cas est par exemple celui d'un tableau qui ne contient qu'une seule valeur.

— Si on a toujours $m = 1$, alors dans ce cas on a :

$$T(n) = \begin{cases} \Theta(1) & \text{si} \quad n = 1 \\ \Theta(n) + T(n-1) & \text{sinon.} \end{cases}$$

On a alors $T(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$, ce qui n'est pas de chance lorsqu'on s'appelle `quick_sort`.
Le résultat est le même si $m = L$ est une constante.
Le cas $m = 1$, est par exemple celui d'un tableau trié dans l'ordre croissant.

Les deux cas précédents sont le pire et le meilleur cas pour cet algorithme. On peut démontrer qu'en moyenne la complexité est encore en $\Theta(n\ln(n))$.

Le tri rapide peut se faire en places i on l'optimise un peu, ce qui est un avantage certain sur le tri fusion.

Pour éviter de se retrouver dans le pire des cas, on peut choisir le premier pivot au hasard parmi les éléments de la liste.

3.8 Exercices II

Exercice 3.14.

On peut optimiser le tri fusion en ne fusionnant pas la partie gauche et droite du tableau lors de l'appel récursif si tous les éléments de la partie gauche sont plus petits que ceux de la partie droite. Modifier la fonction `merge_sort` pour qu'elle tienne compte de cette amélioration.

Exercice 3.15.

Nous allons voir ici quelques optimisations que l'on peut apporter à `quick_sort`.

i. Le second appel récursif de `quick_sort` est terminal, on peut donc procéder à une **tail call elimination**, il s'agit de transformer le dernier appel récursif en une boucle. Écrire en pseudo-code ce que l'on obtient.

ii. Cette optimisation ne change rien à la complexité temporelle, mais diminue l'espace mémoire utilisé en soulageant la pile d'appels. On peut d'ailleurs encore faire mieux, en choisissant de toujours faire l'appel récursif sur le plus petit côté du tableau. Écrire en pseudo-code ce que l'on obtient.

iii. Comme dernière optimisation, on peut cesser les appels récursifs lorsque le tableau est assez petit, disons de taille inférieure à 15, et finir de le trier avec `insertion_sort`. Écrire le pseudo-code de cette version.

Exercice 3.16.

On dit qu'un tableau est k-presque trié lorsque :

i. si chacun des éléments se trouvent à au plus k places de la position qu'il devrait occuper.

ii. ou bien si au plus k éléments ne sont pas à leur place.

Démontrer que que la complexité du tri par insertion est en $O(kn)$ sur les tableaux k-presque triés. Donc en $O(n)$ à k fixé.

Exercice 3.17.

Le mélange de « Knuth » permet de mélanger un tableau, de sorte que toutes les permutations possibles soient équiprobables. Son principe est le suivant : si t est un tableau de longueur n, pour tout $i \in [\![0, n-1]\!]$ on choisit de manière uniforme un élément $j \in [\![i, n-1]\!]$ et on échange $t[i]$ avec $t[j]$. Implémenter cet algorithme.

Exercice 3.18.

Modifier la fonction `selection_sort` pour déterminer le k-ième élément le plus petit d'une liste. Déterminer les complexités au pire et au meilleur des cas de la fonction obtenue.