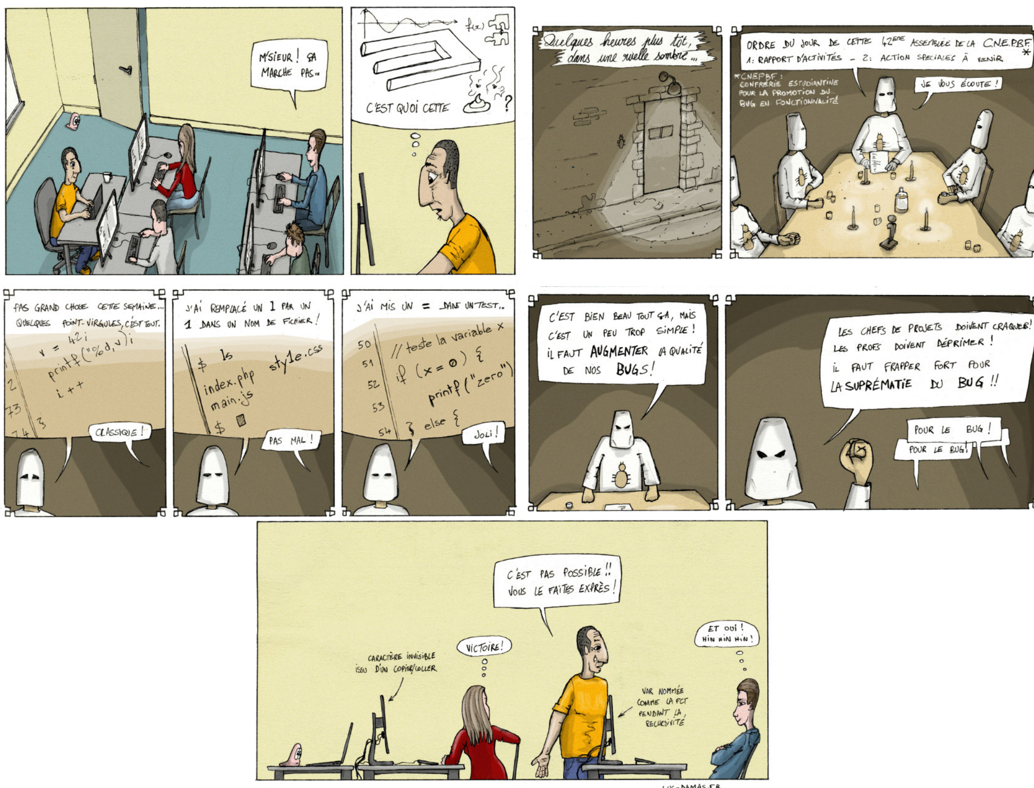
Informatique commune

MP



Lycée Sainte Geneviève

2023-2024

Table des matières

1 Algorithmes de tri **3**
1 Quelques définitions 4
2 Petit point sur PYTHON 5
3 Algorithmes de tri 6

ITC 1

Algorithmes de tri

Sommaire

1 **Quelques définitions** 4

2 **Petit point sur Python** 5

2.1 Fonctions natives de tri 5

2.2 Conventions syntaxiques 5

3 **Algorithmes de tri** 6

3.1 Tri par sélection 6

3.2 Tri par insertion 7

3.3 Tri fusion (merge-sort) 8

3.4 Tri rapide (quicksort) 10

L'objectif de ce chapitre est de réviser les tris introduits dans le cours de première année, tout en réinvestissant la notion de complexité.
Nous allons voir dans un premier temps que Python propose des fonctions qui effectuent un tri directement, des sortes de « boîtes noires » dont on ne sait pas grand-chose. Ensuite, nous aborderons les principales techniques de tri que vous avez à connaître et nous discuterons de leur complexité en temps.
Évidemment, un algorithme de tri est efficace si, lorsqu'il retourne la liste triée, il a effectué ce tri en un temps le plus court possible.
Le caractère aléatoire de l'organisation initiale de la liste à trier va nous conduire à discuter des complexités dans le meilleur et dans le pire des cas. Il est évident que nous comparerons alors les pires des cas, mais il ne faudra jamais oublier qu'il est possible, selon l'organisation initiale de la liste à trier, que la performance d'un algorithme censé être plus lent qu'un autre pourra s'inverser...

1 Quelques définitions

Voici quelques définitions à connaître :

- Clefs : c'est ce qui est utilisé pour trier des éléments. Par exemple :
  - On peut trier des mots à l'aide de leur première lettre. La clé est ainsi la première lettre
  - On peut trier des couples $(4,5)(1,2)(1,3)(2,3)(3,1)$ selon leur premier terme, ce sera donc la clé, ou selon leur deuxième terme ...
- Tri comparatif : Tri fondé sur la comparaison entre les « clefs » des éléments pour les trier
- Tri itératif (ex : tri insertion) : Tti basé sur un ou plusieurs parcours itératifs de la liste à trier
- Tri récursif (ex : tri rapide, tri fusion) : tri basé sur une méthode récursive
- Tri en place : tri qui n'utilise une quantité constante d'éléments en mémoire
- Tri avec listes auxiliaires : tri qui crée des listes auxiliaires pour réaliser le tri (quantité de mémoire utilisée non constante)
- Tri stable (ex : tri insertion et tri fusion si bien codés) : tri qui conserve l'ordre relatif des éléments de même clef.

Application

ITC1.1 : Caractérisation de tris

.

On se donne la liste suivante : `l=[(4, 5),(1, 2),(1, 3),(2, 3),(3, 1)]` Caractériser un algorithme de tri qui retourne :

Q.1. `[(1, 2),(1, 3),(2, 3),(3, 1),(4, 5)]`

Q.2. `[(1, 3),(1, 2),(2, 3),(3, 1),(4, 5)]`

2 Petit point sur Python

2.1 Fonctions natives de tri

Introduisons ici deux fonctions que bscPython propose nativement. et pour cela, regardons le résultat d'exécution des fonctions suivantes.

```
## 1 Tris natifs en Python
n = 20
L = [rd.randint(1,n) for i in range(n)]
print(L)
L.sort()
print(L)

LL = sorted(L)
print(LL)
```

Il est fort probable que ces deux fonctions soient interdites d'utilisation le jour du concours ...

2.2 Conventions syntaxiques

Les PEP (*Python Enhancement Proposal*) sont des documents qui décrivent de nouvelles fonctionnalités proposées pour Python et en documentent les aspects.

Les conventions syntaxiques sont regroupées dans la PEP 8. Elles précisent par exemple les normes d'indentation (4 espaces), les règles de nommage des variables composées (utilisation du _), les règles de saut de ligne (pas à l'intérieur d'une fonction), ...

Les PEP 484 et PEP 3107 ont introduit et précisé le mécanisme d'annotation de fonctions qui permettent de préciser le type des fonctions et de leurs valeurs de retour. Cet alourdissement de syntaxe possède un intérêt pour l'utilisation dans des projets lourds utilisant les potentialités de contrôle des I.D.E., pour la vérification automatique par des fonctions tierces, pour des utilisations inter-langages, ...
Voici quelques exemples élémentaires utiles :

Code Python

Mode éditeur

```
>>> def greeting(name: str) -> str:
...     return 'Hello ' + name
...
>>> # Appel de fonction
>>> print(greeting('MP'))
Hello MP

>>> # Accès aux annotations utilisateur
>>> print(greeting.__annotations__)
{'name': <class 'str'>, 'return': <class 'str'>}
```

3 Algorithmes de tri

Nous allons aborder quelques manières de trier que vous avez à connaître. Nous verrons ici les principes, et en TP vous les coderez directement. Appelons L la liste à trier et L la liste triée.

3.1 Tri par sélection

Le tri par sélection est sûrement le plus simple des tris. L'idée est de placer récursivement en tête de liste le minimum de sous-listes successivement triées. Le tri s'effectue ici très facilement en place.

Code .

Caractéristiques Ce tri est :

Complexité .

3.2 Tri par insertion

Principe Le tri insertion consiste à prendre chaque valeur d'une liste à trier, de la comparer à la valeur précédente, et d'intervertir les deux si la valeur actuelle est plus faible que la valeur précédente. On procède alors ainsi de suite jusqu'à ce que la valeur déplacée initialement soit supérieure à la précédente. On commence à partir de la seconde valeur dans la liste. Il n'est pas nécessaire de créer une seconde liste, on trie directement la liste initiale.

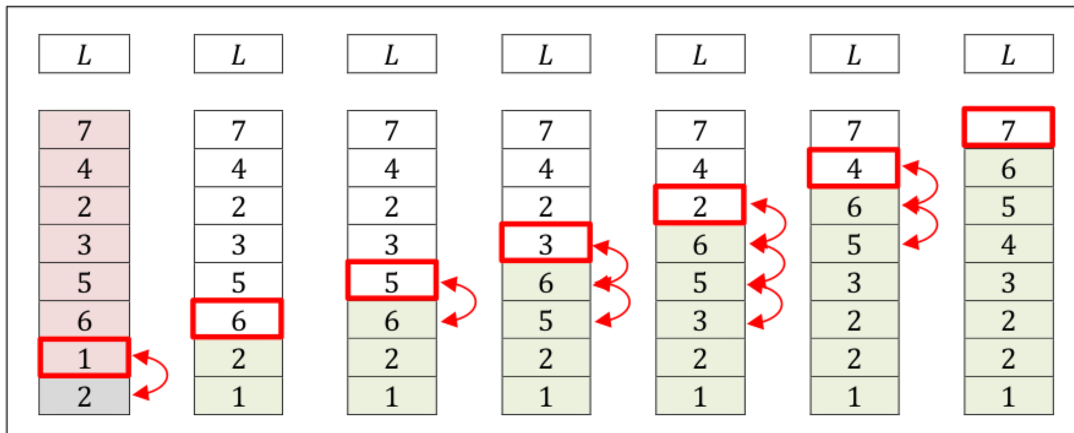


FIGURE 1.1 – Illustration d'un tri insertion

Code .

Caractéristiques Ce tri est :

Complexité La première boucle réalisé n itérations dans tous les cas.
Dans le pire des cas, la seconde boucle réalise j < n itérations, soit une complexité quadratique.
Dans le meilleur des cas(liste triée) la deuxième boucle est de coût constant, donc la complexité est linéaire.

3.3 Tri fusion (merge-sort)

Cet algorithme emploie une stratégie dite de *diviser pour régner*. Il consiste à diviser récursivement une liste en deux puis à fusionner les sous listes obtenues en les triant.
Les étapes successives sont les suivantes. Soit une liste L à trier :

- traiter le cas de base : si L ne contient qu'un terme, elle est triée ;
- partager L en 2 listes L_1 et L_2 de tailles identiques (à 1 près) ;
- appliquer récursivement la procédure aux listes L_1 et L_2 pour les trier ;
- en supposant que L_1 et L_2 ont été triées à l'étape précédente, les fusionner de manière ordonnée.

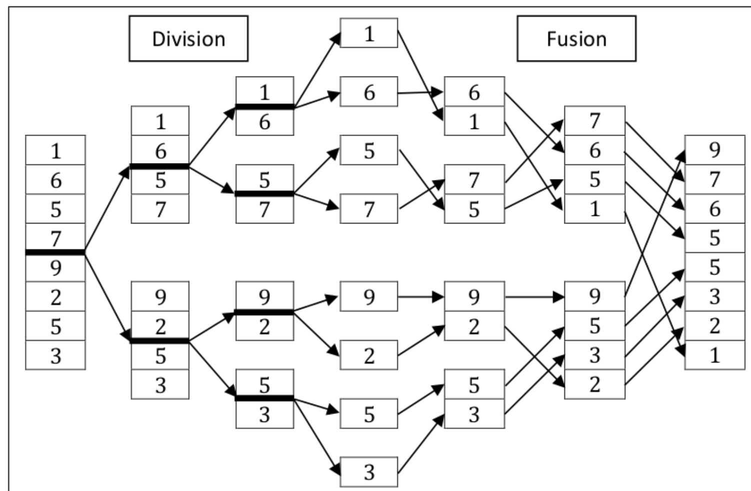


Figure 1.2 – Illustration d'un tri fusion

Code .

Remarque Le tri en place n'est pas présenté ici ; le décalage de tous les termes de la fusion n'étant pas du tout évident à mettre en place sans liste auxiliaire. On donne l'illustration ci-dessous d'une façon dont procéderait dans un tel tri.

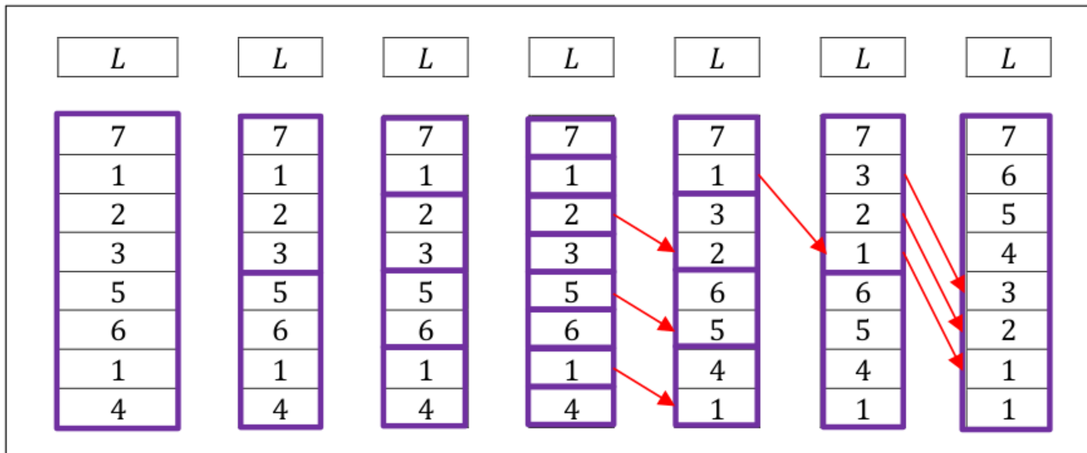


Figure 1.3 – Principe de tri fusion en place

Complexité Commençons par montrer que la complexité à l'étape n est en $\mathcal{O}(n)$ puis trouvons la complexité globale.

Il n'y a pas de distinction à faire entre meilleur et pire des cas puisque les déplacements sont tous effectués.

3.4 Tri rapide (quicksort)

Ce tri est basé sur le choix d'un pivot (une des valeurs de la liste à trier) et le partage des autres valeurs en fonction de celui-ci. Naïvement, on considère que le pivot est la première valeur de la liste. On peut effectuer un autre choix, nous en parlerons lors de l'étude de la complexité de ce tri.

3.4.1 Tri rapide avec listes auxiliaires

Le tri rapide a été inventé par Hoare vers 1960. Le principe est de partager une liste en 2 listes telles que dans la première, toutes les valeurs prises soient inférieures à celles de la seconde. On retrouve ici une stratégie dite *diviser pour régner* où un problème est décomposé en deux problèmes plus simples. À la fin, on regroupe les résultats de chaque sous-problème pour arriver au résultat.
Il est alors possible d'appliquer récursivement cette démarche afin d'obtenir, à la fin, une liste triée :

- traiter le cas de base : si la liste est vide, la renvoyer.
- choisir un élément P de L appelé « pivot » (naïvement, première valeur).
- créer les listes L_1 et L_2 telles que $\begin{cases} \forall i \neq 0, L_1[i] \leqslant P \\ \forall i \neq 0, L_2[i] > P \end{cases}$
- appliquer récursivement le tri aux listes L_1 et L_2 pour obtenir deux listes L_1' et L_2' triées
- renvoyer les listes combinées dans l'ordre : L_1', P, L_2'.

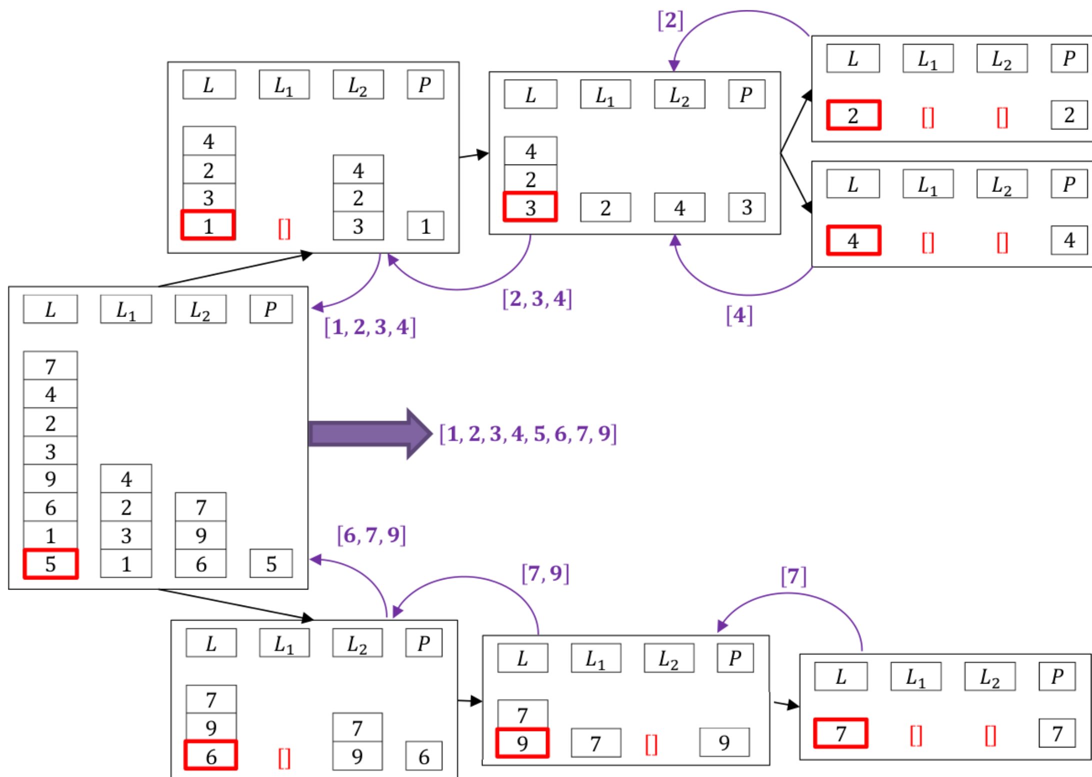


Figure 1.4 – Illustration d'un tri rapide avec listes auxiliaires

Code .

Commentaire Ce tri n'est ni stable, ni en place. La taille de la pile de récursivité peut poser problème dans le cas de listes de grande taille. Une version en place est donc à préférer.

3.4.2 Tri en place

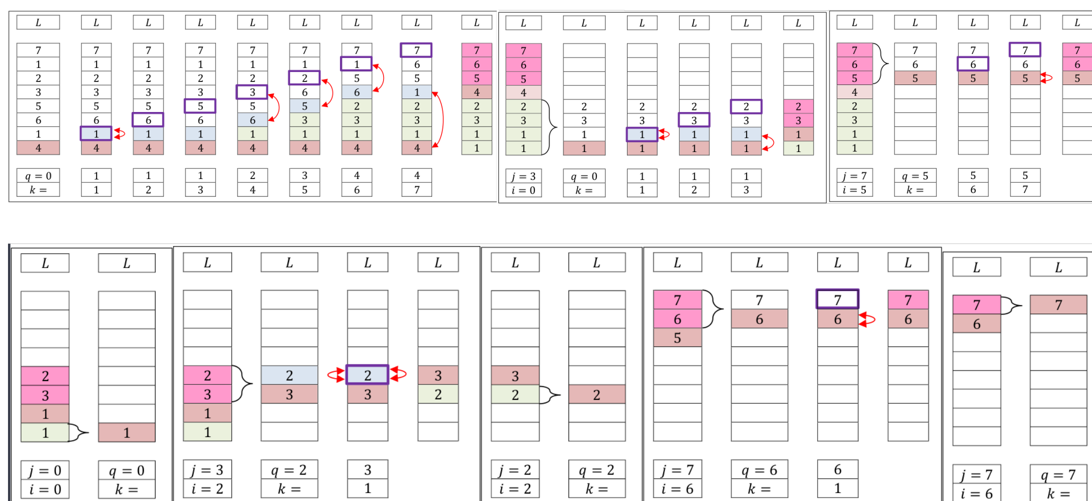
On peut réaliser un tri rapide en place, c'est-à-dire sans utiliser de mémoire additionnelle pour stocker des listes auxiliaires.
Le principe de réalisation de ce tri consiste à appliquer la démarche suivante par récurrence :

- considérer une portion de liste entre les indices `i` et `j` inclus. À la première itération, c'est la liste entière.
  Si cette sous liste contient un seul terme, ne rien exécuter, elle est déjà triée ;
- sinon :
  - choisir le pivot `P` : naïvement, le premier. Sinon, en choisir un et l'échanger avec la première valeur de la portion de liste traitée
  - appeler `p` l'indice du pivot : `p=i`
  - définir un indice `q` qui au départ vaut `p`
  - étudier chaque valeur de la sous-liste considérée pour un indice `k` variant entre les indices `i+1` et `j` inclus et procéder ainsi :
    - si `L[k]>P`, ne rien faire
    - sinon (`L[k]  = P`) :
      - `q=q+1`
      - `L[q],L[k] = L[k],L[q]`
  - à la fin, échanger le pivot avec le terme à la position `q: L[p],L[q] = L[q],L[p]`
  - appeler récursivement cette procédure sur les « sous » listes de part et d'autre du pivot de la portion de liste étudiée

Après cette étape, on obtient un pivot définitivement placé, et deux sous listes avant et après le pivot telles que :

- toutes les valeurs situées avant le pivot lui sont inférieures ;
- toutes les valeurs situées après le pivot lui sont strictement supérieures ;

Il suffit alors de procéder de même sur les deux sous-listes et ainsi de suite par récursivité. Il n'y a alors aucun renvoi, chaque sous liste étant directement triée.



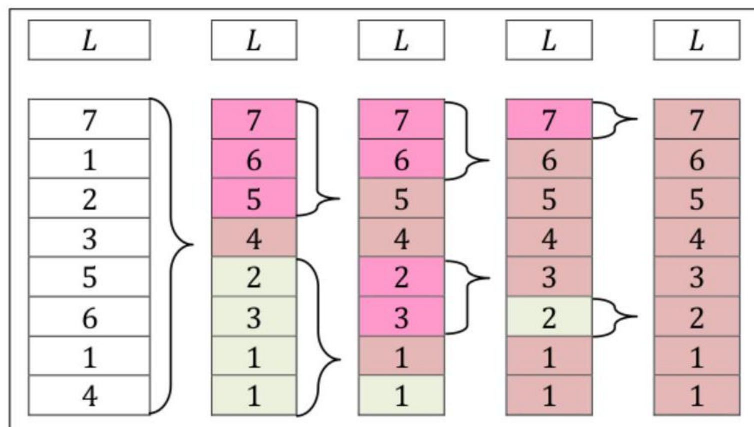


Figure 1.5 – Illustration résumée d'un tri rapide

Code .

Commentaire De base, ce tri n'est pas stable. Prenons l'exemple d'un pivot en première position égal à l'un des autres termes de la liste ; il sera replacé après les termes avec lesquels il est égal.
Une adaptation mineure de ce code permet de trouver la médiane d'une liste : on ne traite à chaque étape que la sous-liste contenant l'indice milieu de la liste jusqu'à ce que le pivot se retrouve à cette position qui est la médiane de la liste.

Complexité Dans le meilleur des cas, à chaque division, on a autant de termes dans chacune des sous listes. Ainsi, à chaque étape, il s'appelle 2 fois à l'ordre $n/2$. Pour chaque exécution à l'ordre n, il parcourt la liste de ses n éléments. La complexité à l'ordre n est donc $\mathcal{O}(n)$.
La complexité globale est donc comme précédemment quasi-linéaire : $C(n) = n \ln n$.

Dans le pire des cas, à chaque division, une sous-liste possède 0 termes, l'autre les $n-1$ restants.
Les n appels de la liste mènent à une complexité quadratique : $C(n) = \mathcal{O}(n^2)$
Le pire des cas est atteint lorsque la liste est déjà presque triée.
Une pré-étude de la liste avant de la trier, permet de choisir un pivot en milieu de liste si l'on voit qu'elle déjà organisée.

Comparaison finale On peut comparer ces deux derniers algorithmes ainsi :

- le tri rapide effectue des travaux de tri autour du pivot puis appelle récursivement ce travail de tri : on parle de **récursivité sur les résultats** ;
- le tri fusion appelle récursivement une fonction qui divise le problème et trie à la fin, puis recombine les résultats : on parle de **récursivité sur les données**.