Chapitre IV
Algorithmes de tri

1. Introduction

Outre l'intérêt intrinsèque que peut représenter le tri des éléments d'un ensemble, il peut être utile, en préalable à un traitement de données, de commencer par trier celles-ci. Considérons par exemple le problème de la recherche d'un élément dans un tableau. Ce problème a un coût linéaire lorsque le tableau n'est pas trié, et un coût logarithmique si ce dernier est trié. Ainsi, si on prévoit de faire de nombreuses recherches, il peut devenir intéressant de commencer par trier ces données, même si le coût du tri s'ajoute au coût des différentes recherches.
À titre d'exemple, notons n le nombre d'éléments du tableau, et p le nombre de recherches à effectuer. Si on ne trie pas le tableau la complexité totale est un $p \times O(n) = O(np)$; si on trie le tableau au préalable, la complexité totale est égale à $C(n) + p \times O(\log n) = C(n) + O(p \log n)$, où $C(n)$ est la complexité de l'algorithme de tri choisi. Nous verrons dans ce chapitre que les algorithmes de tris efficients ont une complexité $C(n) = O(n \log n)$; la complexité totale de la deuxième méthode est alors égale à un $O((n+p)\log n)$.
On a $(n+p)\log n \leqslant np \iff p \geqslant \dfrac{n \log n}{n - \log n} \sim \log n$ donc lorsque $p \geqslant \log n$, il devient préférable de trier la liste au préalable.

1.1 Présentation du problème

Avant toute chose, il importe de prendre conscience que la performance d'un tri va être directement liée à la structure de données utilisée et en particulier du temps d'accès à un élément : nous savons que l'accès à un élément d'un tableau est de coût constant, mais d'autres structures de données peuvent avoir des temps d'accès de complexité non constante. Certains algorithmes peuvent donc se révéler plus adaptés à une structure de donnée plutôt qu'à une autre. Les méthodes de tris peuvent aussi différer suivant que la structure de donnée à trier soit mutable ou pas. Dans le cas d'une structure mutable (un tableau par exemple), on distinguera deux type de tris :

- les tris *en place*, pour lesquels on procède par permutation des éléments au sein de la structure de données ;
- les autres tris, pour lesquels les éléments sont recopiés et triés, dans une nouvelle structure de données.

L'intérêt des tris en place est de posséder une complexité spatiale moindre (en général constante) que les autres tris, qui ont une complexité spatiale au moins égale à la taille de la structure de donnée à trier.

Par exemple, en Python, la fonction `sorted` ne réalise pas un tri en place : dans l'exemple qui suit le tableau `t` n'est pas modifié.

```
In [1]: t = [5, 3, 6, 9, 1, 2, 4, 7, 8]

In [2]: sorted(t)
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: t
Out[3]: [5, 3, 6, 9, 1, 2, 4, 7, 8]
```

En revanche, la méthode `sort` réalise un tri en place :

```
In [4]: t.sort()

In [5]: t
Out[5]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

L'autre élément pertinent de mesure de la complexité est la manière dont on détermine l'ordre relatif des éléments à trier. Le programme se limite aux tris qui procèdent par *comparaison entre les éléments* de la structure

à trier. Nous allons donc considérer que ces éléments appartiennent à un ensemble E muni d'une relation d'ordre $\leqslant$ et nous autoriser uniquement à comparer *entre eux* deux éléments de la structure. Nous supposerons de plus que cette comparaison est de complexité constante.

Dans la pratique, les algorithmes que nous allons étudier seront illustrés en Python par le tri d'un tableau (de type `list` ou `numpy.array`) à valeurs numériques. Ainsi, l'affectation et la comparaison de deux éléments entre eux se réaliserons à coût constant et constituerons la mesure de complexité de ces algorithmes.

Sous ces hypothèses nous admettrons le théorème suivant :

Théorème 1.1 — *Tout algorithme de tri par comparaison d'un tableau de n cases possède une complexité dans le pire des cas au moins égale à un* $O(n\log n)$.

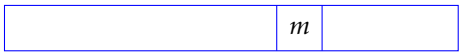
Remarque. Il existe des algorithmes qui n'utilisent pas de comparaison entre éléments mais tirent profit d'une information supplémentaire dont on dispose sur les éléments à trier. Le tri par base (*radix sort* en anglais) utilise la décomposition dans une base donnée des nombres entiers pour les trier. D'autres algorithmes tirent partie de la représentation en mémoire des objets à trier.

Nous allons maintenant étudier quelques algorithmes de tri par comparaison élémentaires, avant de nous intéresser aux algorithmes de tri les plus performants.

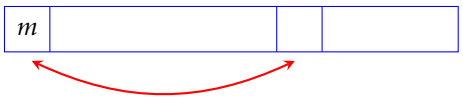
1.2 Le tri par sélection

Appelé *selection sort* en anglais, c'est l'algorithme le plus simple qui soit : on cherche d'abord le plus petit élément du tableau, que l'on échange avec le premier. On applique alors cette méthode au sous-tableau restant.

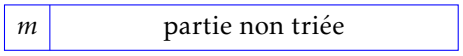
– recherche de l'élément minimal :



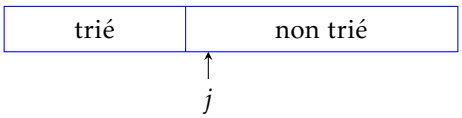
– permutation avec le premier :



– tri du tableau restant :



Considérons le tableau en cours de tri. La partie gauche $t[:j]$ du tableau est triée, les éléments de cette partie sont à leurs places définitives, et il reste à trier la partie droite $t[j:]$.



Nous allons donc rédiger une fonction `minimum(t, j)` qui prend pour arguments un tableau t et un entier j et qui renvoie l'indice du minimum de la coupe $t[j:]$, pour en déduire une fonction `selectionSort(t)` qui prend pour argument un tableau t et réalise le tri en place de ce dernier en suivant la méthode du tri par sélection.

```
def minimum(t, j):
    i, m = j, t[j]
    for k in range(j+1, len(t)):
        if t[k] < m:
            i, m = k, t[k]
    return i
```

```
def selectionSort(t):
    for j in range(len(t)-1):
        i = minimum(t, j)
        if i != j:
            t[i], t[j] = t[j], t[i]
```

Étude de la complexité

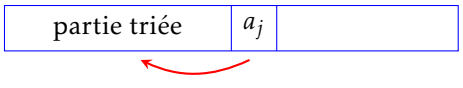
Le nombre de comparaisons effectuées par la fonction `minimum(t, j)` est égal à $n-1-j$ donc le nombre total de comparaisons est égal à $\sum_{j=0}^{n-2}(n-1-j) = \frac{n(n-1)}{2} \sim \frac{n^2}{2}$ (quelle que soit la configuration du tableau) et le nombre d'échanges inférieur ou égal à $n-1$. Il s'agit donc d'un algorithme de complexité quadratique $O(n^2)$. Une de

ses particularités est le nombre réduit (linéaire) d'échanges effectués, mais à part cela c'est le plus mauvais (en termes de comparaisons) que nous étudierons.

1.3 Le tri par insertion

Appelé *insertion sort* en anglais, il 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, un peu à la manière d'un joueur de cartes insérant dans son jeu trié les cartes au fur et à mesure qu'il les reçoit.

– insertion dans un tableau trié :



– tri du tableau restant :

Nous allons donc rédiger une fonction `insere(t, j)` qui prend pour arguments un tableau t et un entier j en supposant la coupe $t[:j]$ triée et qui réalise l'insertion de l'élément $t[j]$ dans cette partie triée, puis une fonction `insertionSort(t)` qui prend pour argument un tableau t et réalise le tri en place de ce dernier en suivant la méthode du tri par insertion.

```
def insere(t, j):
    k, a = j, t[j]
    while k > 0 and a < t[k-1]:
        t[k] = t[k-1]
        k -= 1
    t[k] = a
```

```
def insertionSort(t):
    for j in range(1, len(t)):
        insere(t, j)
```

Étude de la complexité
Insérer un élément dans un tableau trié de longueur j nécessite au plus j comparaisons, ce qui conduit à un nombre maximal de comparaisons égal à $\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} \sim \frac{n^2}{2}$. Cette situation a effectivement lieu dans le cas où la liste initiale est triée par ordre décroissant, ce qui nous permet d'énoncer le :

Théorème 1.2 — *Dans le pire des cas, le nombre de comparaisons du tri par insertion est équivalent à* $\frac{n^2}{2}$.

Il s'agit donc là encore d'un algorithme de complexité quadratique $O(n^2)$. De plus, le nombre de comparaisons dans le pire des cas est identique au nombre de comparaisons utilisé par l'algorithme de tri par sélection. Cependant, lorsque le tableau est déjà trié l'algorithme ne réalise que n comparaisons, ce qui est un des points forts du tri par insertion (il est très efficace pour trier des tableaux quasi-triés), et il s'avère aussi plus performant en moyenne puisqu'il est possible de prouver le résultat suivant, que nous admettrons :

Théorème 1.3 — *Le tri par insertion effectue en moyenne un nombre de comparaisons équivalent à* $\frac{n^2}{4}$.

2. Algorithmes de tris efficients

Nous venons d'étudier deux algorithmes de tri par comparaison, tous deux de complexité quadratique, aussi bien dans le pire des cas qu'en moyenne. Nous allons maintenant nous intéresser à deux algorithmes plus efficients : le tri fusion déjà étudié au chapitre précédent, et le tri rapide.

2.1 Le tri fusion

Appelée *merge sort* en anglais, nous l'avons déjà rencontré comme exemple d'algorithme récursif. Cette méthode adopte une approche « diviser pour régner » : on partage le tableau en deux parties de tailles respectives $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$ que l'on trie par un appel récursif, puis on fusionne les deux parties triées (illustration figure 1).

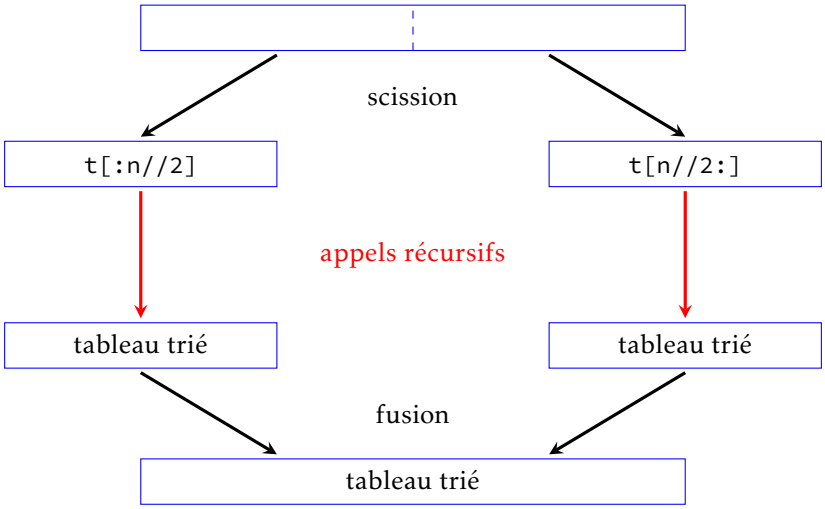


FIGURE 1 – Une illustration du tri fusion.

Cependant, cette méthode de tri possède un inconvénient : il est très difficile (pour ne pas dire impossible) de fusionner deux demi-tableaux en place, c'est à dire en s'autorisant uniquement la permutation de deux éléments dans le tableau sans espace mémoire supplémentaire. Pour cette raison, les implémentations du tri fusion ne sont pas des tris en place, autrement dit calculent un *nouveau* tableau trié.

Le code du tri fusion a déjà été donné au chapitre précédent.

Étude de la complexité temporelle

Notons $T(n)$ la complexité temporelle du tri fusion. La fusion de deux tableaux triés se réalise en temps linéaire vis-à-vis de la somme des tailles des deux tableaux à fusionner donc $T(n)$ vérifie une relation de récurrence de la forme :

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n).$$

Autrement dit, il existe une constante K telle que pour tout $n \geqslant 2$, $T(n) \leqslant T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + Kn$.
Lorsque $n = 2^p$ la suite $u_p = T(2^p)$ vérifie la relation $u_p \leqslant 2u_{p-1} + K2^p$ que l'on peut encore écrire : $\dfrac{u_p}{2^p} - \dfrac{u_{p-1}}{2^{p-1}} \leqslant K$.
Par télescopage on en déduit $\dfrac{u_p}{2^p} - u_0 \leqslant Kp$ puis $u_p \leqslant (Kp + u_0)2^p$, soit $T(n) = O(n \log n)$. Nous admettrons que cette formule reste vraie pour un entier n quelconque. Ainsi, *le coût temporel du tri fusion est semi-logarithmique*.

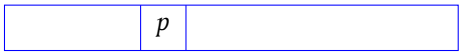
Étude de la complexité spatiale

Notons $S(n)$ la complexité spatiale du tri fusion. Telle que nous l'avons écrit au chapitre précédent, la fusion a un coût linéaire vis-à-vis de la somme des tailles de ses arguments, ce qui conduit à une relation de la forme : $S(n) = S(\lfloor n/2 \rfloor) + S(\lceil n/2 \rceil) + O(n)$, soit $S(n) = O(n \log n)$. Cependant il est possible assez facilement de ramener ce coût à un $O(n)$.

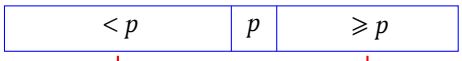
2.2 Le tri rapide

Appelé *quick sort* en anglais, ce tri adopte lui aussi une démarche de type « diviser pour régner » : on commence par segmenter le tableau autour d'un pivot choisi parmi les éléments du tableau en plaçant les éléments qui lui sont inférieurs à sa gauche, et les éléments qui lui sont supérieurs, à sa droite. À l'issue de cette étape, le pivot se trouve à sa place définitive, et les parties gauche et droite sont triées par l'intermédiaire d'un appel récursif.

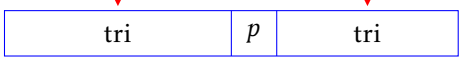
– choix d'un pivot :



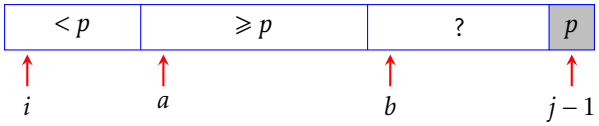
– segmentation :



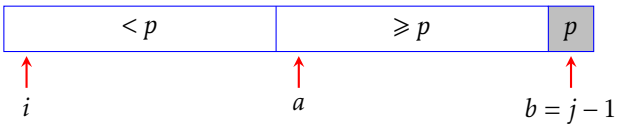
– appels récursifs :



Il existe plusieurs façons de segmenter une portion de tableau $t[i:j]$. la méthode que nous allons suivre consiste à choisir pour pivot l'élément $p = t[j-1]$ et à maintenir l'invariant suivant :



Les valeurs initiales sont $a = b = i$ et la condition d'arrêt $b = j-1$. Lorsque le processus se termine, la situation est la suivante :



et il suffit alors de permuter les cases d'indices a et $j-1$ pour achever la segmentation du tableau $t[i:j]$.

On commence donc par rédiger une fonction `segmente(t, i, j)` qui prend pour arguments un tableau t et deux indices i et j, qui segmente la coupe $t[i:j]$ en utilisant l'élément $p = t[j-1]$ comme pivot et qui renvoie l'indice a du pivot en position finale. Ensuite, il nous reste à écrire la fonction `quickSort(t, i, j)` qui prend pour argument un tableau t et deux indices i et j et réalise le tri en place de la coupe $t[i:j]$ en suivant la méthode du tri rapide.

```
def segmente(t, i, j):
    p = t[j-1]
    a = i
    for b in range(i, j-1):
        if t[b] < p:
            t[a], t[b] = t[b], t[a]
            a += 1
    t[a], t[j-1] = t[j-1], t[a]
    return a
```

```
def quickSort(t, i, j):
    if i + 1 < j:
        a = segmente(t, i, j)
        quickSort(t, i, a)
        quickSort(t, a + 1, j)
```

Pour avoir une fonction de tri en place d'un tableau par la méthode du tri rapide il reste à définir la fonction :

```
def triRapide(t):
    quickSort(t, 0, len(t))
```

2.3 Étude de la complexité du tri rapide

Il apparaît clairement que la complexité de la segmentation est linéaire. Cette dernière ayant pour objet de partager le tableau en deux sous-ensembles de tailles n_1 et n_2 avec $n_1 + n_2 = n-1$ (le pivot n'appartient à aucun de ces deux sous-ensembles), la complexité de l'algorithme de tri rapide vérifie la relation : $C(n) = C(n_1) + C(n_2) + O(n)$.

L'intuition nous pousse à penser que cette méthode est d'autant meilleure que n_1 et n_2 sont proches et qu'à l'inverse un partitionnement très déséquilibré risque d'avoir une influence négative sur le coût total. Avant de justifier rigoureusement ces assertions, intéressons-nous à ces deux situations extrêmes.

Partitionnement dans le cas le plus défavorable
Supposons qu'à chaque partitionnement l'une des deux parties du tableau segmenté soit vide : nous avons alors $n_1 = n-1$ et $n_2 = 0$ et la relation de récurrence devient : $C(n) = C(n-1) + O(n)$, ce qui conduit immédiatement à $C(n) = O(n^2)$. Cette situation a lieu par exemple lorsque le tableau est déjà trié (dans un sens comme dans l'autre) puisque le pivot choisi est à chaque étape un élément extrémal de la partie à segmenter.

Partitionnement dans le cas le plus favorable
À l'inverse, supposons qu'à chaque étape du partitionnement les deux parties du tableau segmenté soient exactement de même taille ; nous avons alors $n = 2^p - 1$ et la relation de récurrence devient $C(n) = 2C(\lfloor n/2 \rfloor) + O(n)$. Posons $u_p = C(2^p - 1)$; alors $u_p = 2u_{p-1} + O(2^p)$. Nous avons déjà effectué ce calcul pour le tri fusion et obtenu $u_p = O(p2^p)$ soit $C(n) = O(n \log n)$.

Nous allons maintenant prouver le :

Théorème 2.1 — *Lorsque le choix du pivot est arbitraire, le coût de l'algorithme de tri rapide dans le pire des cas est quadratique.*

Partitionnement équilibré
Pourquoi le qualifier de tri rapide alors que le cas défavorable est quadratique ? Parce qu'en moyenne la complexité est quasi-linéaire (c'est-à-dire en $O(n \log n)$), résultat que nous admettrons :

Théorème 2.2 — *En moyenne, la complexité du tri rapide est un* $O(n \log n)$.

Remarque. Plus précisément on peut démontrer que $C(n) \sim 2n \ln n \approx 1,4\, n \log n$. Or on peut montrer que tout algorithme de tri par comparaison réalise dans le pire des cas au moins $n \log n$ comparaisons (c'est un raffinement du théorème 1.1). La constante $1,4$ qui apparaît dans la complexité moyenne du tri rapide montre que l'on est pas loin de la valeur optimale. Ainsi, ce tri se montre dans la pratique plus rapide que la plus-part des autres algorithmes de tri de complexité $O(n \log n)$, ce qui justifie son nom.

2.4 Choix du pivot

Nous l'avons constaté, le choix du dernier élément comme pivot pour segmenter $t[i : j]$ donne en moyenne une complexité quasi-linéaire, mais pour des tableaux triés ou presque triés la complexité est quadratique.
Dans le cas de figure où l'on prévoit d'appliquer l'algorithme de tri rapide à des données peu mélangées, on peut avoir intérêt à choisir au hasard un pivot dans $t[i : j]$. La modification de la fonction de segmentation est mineure : il suffit de tirer au hasard un entier α dans l'intervalle $[\![i, j-1]\!]$ puis à échanger les éléments $t[\alpha]$ et $t[j-1]$ avant de procéder à la segmentation. Le pivot étant maintenant choisi au hasard on peut s'attendre à ce que le partitionnement du tableau d'entrée soit en moyenne relativement équilibré.
Paradoxalement, une autre solution pour trier un tableau presque trié à l'aide du tri rapide consiste … à commencer par mélanger ce dernier ! Néanmoins, cette solution s'avère en pratique moins intéressante que le fait de choisir au hasard le pivot.

3. Exercices

Tris par comparaison

Exercice 1

On désire un algorithme qui détermine si un tableau présente des doublons en son sein.

a. Rédiger un algorithme naïf qui résout le problème. Quelle est sa complexité ?

b. Rédiger maintenant un second algorithme en supposant cette fois le tableau trié. Quelle est sa complexité ? A-t-on intérêt à trier un tableau pour résoudre ce problème ?

Exercice 2

Un singe trie un paquet de cartes de la façon suivante : il prend les cartes, les jette en l'air, les ramasse puis regarde si elles sont triées. Si ce n'est pas le cas il relance les cartes.
On suppose donnée une fonction `shuffle(t)` qui prend une séquence t pour argument et en mélange son contenu en place[a]. Rédiger une fonction `monkeySort(t)` qui réalise le tri en place suivant cette méthode.

a. Cette fonction est présente dans la bibliothèque `numpy.random`.

Exercice 3

Montrer que $n-1$ comparaisons et échanges *entre éléments consécutifs* permettent de placer l'élément maximal en queue d'un tableau à trier. En déduire une nouvel algorithme de tri en place (le tri bulle, ou *bubble sort*), que vous rédigerez en Python.
Faire une analyse de sa complexité temporelle.

Exercice 4

Imaginé par Donald Shell en 1959, *shellsort* est une optimisation du tri par insertion.
Dans le tri par insertion, un élément se rapproche de sa place finale en progressant lentement, case par case. L'accélération consiste à le déplacer en commençant par faire des grands pas, puis des pas de plus en plus petits, jusqu'à, évidemment, des pas de 1 pour que le tableau soit trié.
On considère une suite d'entiers $(h_p)_{p\in\mathbb{N}^*}$ strictement croissante, débutant par $h_1 = 1$. Pour tout $n \in \mathbb{N}^*$, il existe donc un unique entier $p \in \mathbb{N}^*$ tel que $h_p \leqslant n < h_{p+1}$.
L'étape de base de l'algorithme consiste à trier à l'aide du tri par insertion chacun des sous-tableaux débutant respectivement par $a_0, \dots, a_{h_p-1}$, et dont les éléments sont séparés de h_p cases. Une fois cette étape achevée, on dit que le tableau initial est h_p-trié. On répète alors cette opération de base avec la valeur h_{p-1}, et ce jusqu'à la valeur finale 1.

a. Rédiger une fonction `insert(t, h)` qui réalise le h-tri en place du tableau t.

b. Hibbard a démontré en 1963 que la suite $h_p = 2^p - 1$ conduit à un coût en $O(n^{3/2})$. Rédiger en Python le tri de Shell pour cette suite de valeurs.

Remarque. Connaître la suite (h_p) conduisant à la meilleure complexité pour ce tri est à l'heure actuelle un problème ouvert, ainsi que la valeur de cette complexité optimale. La meilleure complexité obtenue à ce jour est en $O(n(\log n)^2)$.

Exercice 5

On considère l'algorithme suivant :

```
def stoogeSort(t, i, j):
    if t[i] > t[j-1]:
        t[i], t[j-1] = t[j-1], t[i]
    if j - i > 2:
        k = (j - i) // 3
        tri(t, i, j-k)
        tri(t, i+k, j)
        tri(t, i, j-k)
```

a) Montrer que si t est un tableau de taille n, l'exécution de `stoogeSort(t, 0, n)` trie correctement le tableau t.

b) Donner une relation de récurrence vérifiée par la complexité temporelle $C(n)$ de cet algorithme, puis, en admettant qu'il existe un réel α tel que $C(n) = O(n^\alpha)$, déterminer cette valeur α.

Remarque. Les trois Stooges (Moe, Larry et Curly) étaient une troupe de comiques américains actifs entre 1930 et 1960 ; ils sont connus en France sous le nom des trois corniauds.

Segmentation et tri rapide

Exercice 6

(Algorithme du *drapeau tricolore*)
On considère un tableau contenant des objets possédant une couleur qui ne peut être que rouge, blanc ou bleu. On souhaite trier ce tableau de sorte que les éléments rouges soient placés en tête, ensuite les éléments blancs, puis enfin les éléments bleus[a].
Pour modéliser cette situation en Python, on suppose que les objets à trier possèdent un attribut `couleur` ne pouvant prendre que les valeurs `'rouge'`, `'blanc'` ou `'bleu'`.
Rédiger en Python une fonction effectuant ce tri en procédant par segmentation ; l'algorithme ne devra effectuer qu'un seul parcours du tableau, et ne déterminer la couleur qu'une fois par élément.

a. ce problème est attribué à Edsger Dijkstra, il suit donc l'ordre des couleurs du drapeau néerlandais.

Exercice 7

Déduire du principe de segmentation un algorithme, qu'on rédigera en Python, pour déterminer le k^{e} plus petit élément d'un vecteur, puis pour déterminer l'élément médian d'un tableau. Que penser du coût de ce dernier algorithme ?