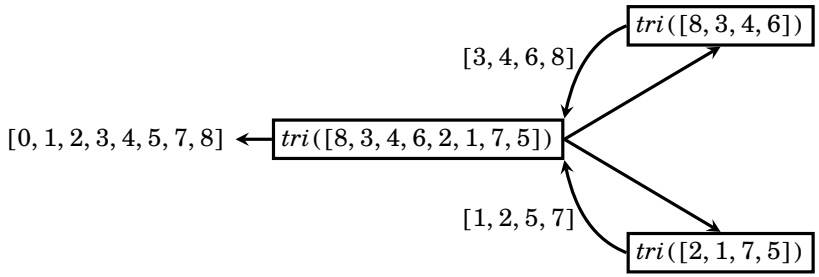
ITC – cours n°11

Algorithmes de tris

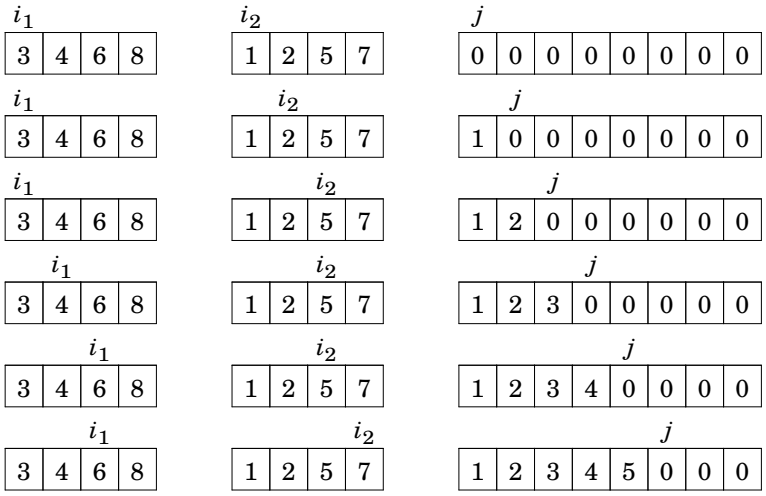
I. Généralités

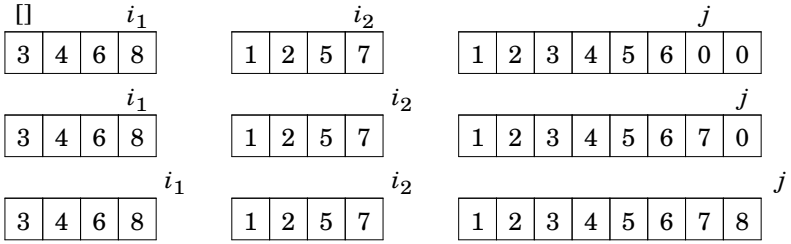
1.1. Retour sur le tri par partition-fusion

Le tri fusion applique le paradigme diviser-pour-régner[1] : l'idée est qu'avec deux tableaux triés, on peut facilement les fusionner pour créer un nouveau tableau lui-même trié :



Le cœur de la procédure est l'opération de fusion des deux sous-tableaux (combiner) : on la réalise en parcourant les deux tableaux en parallèle à l'aide de deux compteurs i_1 et i_2, et en choisissant à chaque étape le bon élément à copier dans un tableau de résultat :





On propose l'implémentation suivante, dans laquelle on renvoie une copie triée du tableau :

```python
def fusionner(t1, t2):
    """
    Prend en paramètre deux tableaux triés et renvoie un tableau
    contenant les éléments de l'ensemble, trié.
    """
    n1, n2 = len(t1), len(t2)
    i1, i2 = 0, 0    # compteurs pour savoir quel est l'élément courant
    tr = [0] * (n1+n2)  # tableau à renvoyer
    for j in range(n1+n2):
        if i2 == n2 or (i1 < n1 and t1[i1] <= t2[i2]):
            tr[j], i1 = t1[i1], i1+1
        else:
            tr[j], i2 = t2[i2], i2+1
    return tr

def tri_fusion(tab):
    """
    Renvoie une copie triée de tab
    """
    n = len(tab)
    if n == 1:  # condition d'arrêt
        return tab
    t1 = tri_fusion(tab[:n//2])  # diviser + régner sur une moitié
    t2 = tri_fusion(tab[n//2:])  # diviser + régner sur l'autre moitié
    return fusionner(t1, t2)     # combiner
```

La complexité est en $\mathcal{O}(n \log_2 n)$: on rappelle la démonstration pour un tableau de taille $n = 2^k$. La fonction fusion effectue une boucle d'opérations élémentaires, on peut

[1] Paradigme qui, comme son nom ne l'indique pas, comprend trois étapes : diviser le problème en sous-problèmes plus simples, les résoudre (régner), puis les **combiner** pour obtenir la solution du problème complet

donc l'écrire $D_n = an$ et un appel à tri-fusion sur un tableau de taille n coûte donc

$$C_n = 2C_{n//2} + D_n + b \simeq 2C_{n//2} + an \quad \text{soit} \quad C_{2^k} = 2C_{2^{k-1}} + a2^k$$

$$\Rightarrow \quad \frac{C_{2^k}}{2^k} = \frac{C_{2^{k-1}}}{2^{k-1}} + a \quad \Rightarrow \quad w_k = w_{k-1} + a \quad \text{en posant} \quad w_k = \frac{C_{2^k}}{2^k} = \frac{C_n}{n}$$

Il s'agit d'une suite arithmétique, donc

$$w_k = w_0 + ka \quad \Rightarrow \quad \frac{C_n}{n} = b + a\log_2 n \quad \Rightarrow \quad \boxed{C_n = \mathcal{O}(n\log_2 n)}$$

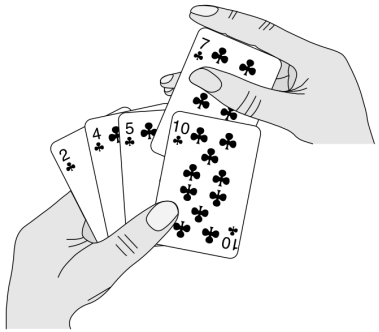
Le résultat se maintient par encadrement si $n \neq 2^k$:

$$2^k \leqslant n < 2^{k+1} \quad \Rightarrow \quad k2^k \leqslant C_n < (k+1)\,2^{k+1} \quad \Rightarrow \quad C_n = \mathcal{O}(n\log_2 n)$$

1.2. Retour sur le tri insertion

Le tri par insertion est généralement décrit comme le tri classique des joueurs de cartes : partant des cases $[0 : i]$[2] triées, on considère ensuite la valeur initialement en case i et on remonte les cases précédentes pour trouver la position $j \in [0 : i+1]$ à laquelle l'insérer, en décalant les valeurs précédentes vers la droite si besoin est. On a alors les cases $[0 : i+1]$ triées, et on recommence. On peut donc avoir, par exemple, l'exécution suivante :

1er passage	3	**2**	1	5	4	2
2e passage	2	3	**1**	5	4	2
3e passage	1	2	3	**5**	4	2
4e passage	1	2	3	5	**4**	2
5e passage	1	2	3	4	5	**2**
6e passage	1	2	2	3	4	5



Nous pouvons proposer l'implémentation suivante en Python :

```python
def tri_insertion(tab):
    """ Trie en place un tableau par insertion """
    n = len(tab)
    for i in range(n):                  # Inv : tab[:i] trié
        j, x = i-1, tab[i]
        while j >= 0 and tab[j] > x:    # décalage de j
            tab[j+1] = tab[j]
            j -= 1
        tab[j+1] = x                    # insertion
```

[2] Dans tout le cours, on utilisera une notation « Python » : $[i_1 : i_2]$ représente donc l'ensemble des entiers de i_1 inclus à i_2 exlus.

Nous avions montré que sa complexité dans le pire cas est $\mathcal{O}(n^2)$. Pour tenir également compte du meilleur cas, elle s'évalue ainsi : la boucle while a une complexité

- $D_{j,n} = aj$ dans le pire cas (il faut remonter tout le tableau pour insérer l'élément j) ;
- $D_{j,n} = a$ dans le meilleur cas (l'élément j est déjà à la bonne place)

donc le tri a pour complexité

$$C_n = a' + \sum_{i=0}^{n-1} D_{i,n} = \begin{cases} \mathcal{O}(n^2) & \text{dans le pire cas / cas moyen} \\ \mathcal{O}(n) & \text{dans le meilleur cas} \end{cases}$$

> ℹ Quand on ne précise pas, on s'intéresse par défaut à la complexité dans le pire cas.

On comprend que le meilleur cas est celui d'un tableau déjà trié (ou presque trié, au sens où chaque élément est à une distance bornée de sa position finale) et le pire celui d'un tableau trié en ordre décroissant. Le cas moyen reste en $\mathcal{O}(n^2)$.

En pratique, le tri par insertion reste donc très performant pour les petits tableaux, ou les tableaux presque triés. Il est ainsi souvent utilisé pour compléter/optimiser les algorithmes de type diviser-pour-régner. Par exemple, si le tri par insertion est plus rapide que le tri par partition-fusion sur des tableaux de taille S ou moins (S dépend des détails d'implémentation et de la machine, mais est souvent de l'ordre de 10 ou 15), on prendra comme cas de base du tri fusion $n < S$, et on fera dans ce cas un tri par insertion.

1.3. Généralisation : le problème du tri

D'un point de vue théorique, le problème du tri se présente ainsi :

- soit en entrée une séquence de n nombres (ou autres objets pouvant être comparés) $(a_0, a_1, \ldots a_{n-1})$;
- la sortie est une permutation $(a'_0, a'_1, \ldots a'_{n-1})$ de la séquence d'entrée telle que $a'_0 \leqslant a'_1 \leqslant \ldots \leqslant a'_{n-1}$

Dans le cadre du cours, nous travaillerons essentiellement sur des séquences de nombres entiers, mais en pratique les données sont souvent des agrégats complexes contenant plusieurs informations, et on les trie selon une de ces informations, appelée clé. Par exemple, pour des entrées de type

(nom, prénom, note maths, note physique, note informatique, groupe de colle)

on peut choisir n'importe lequel de ces champs comme clé.

> ℹ Pour les chaînes de caractères en Python, les opérateurs de comparaison <, > etc ...trient dans l'ordre lexicographique, c'est-à-dire l'ordre du dictionnaire.

Prenons l'exemple précédent, et adaptons le tri insertion pour des tuples, si on veut trier par la clé correspondant à l'index *ind* du tuple :

```python
def tri_insertion(tab, ind):
    """
    Trie en place un tableau par insertion, selon la clé en indice ind
    """
    n = len(tab)
    for i in range(n):                          # Propriété : tab[0:i] trié
        j, x = i-1, tab[i]
        while j >= 0 and tab[j][ind] > x[ind]:  # décalage de j
            tab[j+1] = tab[j]
            j -= 1
        tab[j+1] = x                            # insertion
```

1.4. Caractéristiques générales des tris

On voit apparaître à la lumière de ces deux exemples deux caractéristiques essentielles :

- la **complexité** : il est bien évident que l'on souhaite avoir un aussi bon tri que possible en temps de calcul ;
- le caractère en place ou non : en effet, le tri insertion modifie les données fournies en entrée afin de les réarranger, tandis que le tri par partition-fusion ne touche pas à la liste d'entrée et fabrique une copie des données, qui elle sera triée. On dit que **le tri insertion se fait en place, et le tri par partition-fusion par copie**. Les tris en place ont généralement l'avantage d'utiliser moins de mémoire et d'être plus rapides, puisqu'ils nécessitent moins de copie ; cependant il est parfois préférable de ne pas modifier les données de départ, ce qui impose une copie. Il faut donc choisir selon l'application souhaitée un tri en place ou non.

> On peut toujours transformer un tri en place en un tri qui ne l'est pas, et vice-versa ; par exemple, on peut recopier le tableau obtenu par tri fusion dans le tableau de départ, afin que l'ensemble se comporte comme un tri en place. Cependant, ce genre de « conversion » rajoute typiquement n copies, donc un temps de calcul supplémentaire non négligeable. Il est donc presque toujours plus adapté d'identifier le besoin de tri en place ou non, puis de choisir l'algorithme en conséquence.

Une troisième caractéristique est importante pour des données plus complexes ; reprenons l'exemple des entrées d'étudiant·e·s avec

(nom, prénom, note maths, note physique, note informatique, groupe de colle)

et trions la liste d'abord par nom, puis par groupe de colle selon les tris précédents. On obtient le résultat suivant :

Nom	Prénom	M	P	I	Gpe
Amersu	Xiaan	13	9	11	2
Carsen	Kira	8	11	16	3
Jinn	Qui-Gon	11	11	11	1
Kenobi	Obi-Wan	10	18	17	2
Palpatine	Rey	19	19	19	3
Rwoh	Vernestra	7	15	11	3
Skywalker	Luke	8	4	3	1
Skywalker	Anakin	20	20	20	2
Windu	Mace	15	13	12	1

Nom	Prénom	M	P	I	Gpe
Jinn	Qui-Gon	11	11	11	1
Skywalker	Luke	8	4	3	1
Windu	Mace	15	13	12	1
Amersu	Xiaan	13	9	11	2
Kenobi	Obi-Wan	10	18	17	2
Skywalker	Anakin	20	20	20	2
Carsen	Kira	8	11	16	3
Palpatine	Rey	19	19	19	3
Rwoh	Vernestra	7	15	11	3

On constate que les ordres relatifs de nom n'ont pas bougé au cours de la seconde opération de tri ; autrement dit, à groupe de colle égal, l'ordre lexicographique des noms est maintenu. On dit des tris insertion et fusion qu'ils sont **stables**.

> Pour le tri insertion, on peut même dire plus : il est **online**, c'est-à-dire qu'il fonctionne sans perte de performances si le tableau n'est fourni qu'au fur et à mesure, et non en une fois au début du tri.

1.5. Une subtilité liée à Python lui-même : les tris récursifs en place

Essayons de proposer une version récursive du tri insertion : on appelle récursivement la fonction sur le tableau restreint aux éléments $[0 : n-1]$, sauf si $n = 0$, puis on insère l'élément $n-1$ pour obtenir le tableau trié. L'implémentation pourrait être la suivante :

```
def tri_insertion_naïf(tab):
    n = len(tab)
    if n != 0:                          # cas de base inversé
        tri_insertion_naïf(tab[0:n-1])  # après appel, tab[0:n-1] trié
        j, x = n-2, tab[n-1]
        while j >= 0 and x > tab[j]:    # recherche de la position
            tab[j+1] = tab[j]
            j -= 1
        tab[j+1] = x                    # insertion
```

Cependant, en testant cette proposition, on constate...qu'elle ne fonctionne pas. Plus précisément, seul le dernier élément a effectivement été ramené vers la gauche, jusqu'à être placé à droite d'un plus petit.

Le problème est le suivant : **les tranches créent une copie indépendante de la liste**. Ainsi, la liste elle-même n'est pas modifiée dès le deuxième appel récursif. Cela n'était pas problématique pour le tri par partition-fusion, puisque ce tri n'est pas en place : on fait de toutes façons des copies du tableau. Mais pour les tris en place, il faut trouver une autre approche pour fournir les sous-tableaux lors des appels récursifs.

La solution usuelle consiste à changer les fonctions pour transmettre tout le tableau, accompagné de l'indice qui définit le sous-tableau à considérer pour un appel récursif. On change ainsi la fonction :

```
def tri_insertion_borné(tab, fin):
    """
    Effectue un tri insertion sur le sous-tableau tab[0:fin]
    """
    if fin != 0:                          # cas de base inversé
        tri_insertion_borné(tab, fin-1)   # après appel, tab[0:fin-1] trié
        j, x = fin-2, tab[fin-1]
        while j >= 0 and x < tab[j]:      # recherche de la position
            tab[j+1] = tab[j]
            j -= -1
        tab[j+1] = x                      # insertion
```

La fonction récursive de tri elle-même doit aussi transmettre ce paramètre pour les appels récursifs, mais ce n'est pas utile pour l'utilisateur lors du premier appel. Les deux solutions classiques consistent

- soit à utiliser un paramètre par défaut, non utilisé par l'utilisateur ;

```
def tri_insertion(tab, fin=-1):
    if fin == -1:       # appel initial de l'utilisateur
        fin = len(tab)  # prêt pour les appels récursifs
    if fin != 0:                          # cas de base inversé
      tri_insertion(tab, fin-1)  # appel récursif -> tab[0:fin-1] trié
      j, x = fin-2, tab[fin-1]
      while j >= 0 and x < tab[j]:      # recherche de la position
          tab[j+1] = tab[j]
          j -= -1
      tab[j+1] = x                      # insertion
```

- soit à créer deux fonctions, une « porte d'entrée » naturelle pour l'utilisateur qui initialise les appels correctement, et une « interne » qui fait réellement les appels récursifs (ici `tri_insertion_borné`).

```
def tri_insertion(tab):
    fin = len(tab)
    tri_insertion_borné(tab, fin)
```

Cette deuxième solution évitant le test `fin == -1` à chaque appel récursif, elle est plus performante.

> Les tranches de tableaux NumPy, elles, ne provoquent pas de recopie mais envoient directement une vue modifiable sur le sous-tableau : la version naïve fonctionne donc avec de tels tableaux. Cela nous rappelle la nécessité de s'interroger sur le type de données manipulées...

II. Tri par sélection

2.1. Présentation

Le tri par sélection consiste à **sélectionner** le plus petit élément dans le tableau de taille n (indices $[0 : n-1]$) et le permuter avec l'élément en position 0, puis recommencer avec le sous-tableau de taille $n-1$ (indices $[1 : n-1]$) et ainsi de suite. On pourra donc avoir, par exemple, l'exécution suivante :

1er passage	3	2	**1**	5	4	2
2e passage	1	**2**	3	5	4	2
3e passage	1	2	3	5	4	**2**
4e passage	1	2	2	5	4	**3**
5e passage	1	2	2	3	**4**	5

2.2. Implémentation possible

L'approche simple pour ce tri est impérative :

```
def sélectionner(tab, début, fin):
    """
    Renvoie l'indice du plus petit élément dans le sous-tableau [dé-
but:fin]
    """
    mini, ind = tab[début], début
    for i in range(début, fin):
        if tab[i] < mini:
            mini = tab[i]
            ind = i
    return ind
```

```
def tri_sélection(tab):
    """
    Effectue un tri par sélection
    """
    n = len(tab)
    for i in range(n):
        j = sélectionner(tab, i, n)
        tab[i], tab[j] = tab[j], tab[i]
```

2.3. Caractéristiques

De manière évidente, le tri par sélection est en place.

On constate par ailleurs qu'il n'est pas stable : de deux clés égales, celle qui arrive en premier peut être rejetée très loin au cours d'une permutation avec une clé plus petite, et ne plus être sélectionnée en premier.

Sa complexité est facile à calculer : la fonction `sélectionner`, appelée entre les indices i et $n-1$ a un temps de calcul de la forme $D_{i,n} = a + b(n-i)$, donc

$$C_n = a' + \sum_{i=0}^{n-1} (b' + D_{i,n}) = a' + \sum_{i=0}^{n} (a + b(n-i)) = \mathcal{O}(n^2)$$

III. Tri à bulles

3.1. Présentation

Le tri à bulle s'opère en parcourant le tableau et faisant « remonter » le plus grand élément rencontré : on compare chaque élément à son suivant, et on permute si besoin. Après un passage, le plus grand élément rencontré est nécessairement en dernière position ; on recommence alors les passages jusqu'à ne plus avoir à effectuer de permutation.

Par exemple, pour un tableau contenant 3-2-1-5-4-2, on aura l'éxécution suivante du tableau :

1er passage						2e passage						3e passage						4e passage					
3	**2**	1	5	4	2	**2**	**1**	3	4	2	5	**1**	**2**	3	2	4	5	**1**	**2**	2	3	4	5
2	**3**	**1**	5	4	2	1	**2**	**3**	4	2	5	1	**2**	**3**	2	4	5	1	**2**	**2**	3	4	5
2	1	**3**	**5**	4	2	1	2	**3**	**4**	2	5	1	2	**3**	**2**	4	5	1	2	**2**	**3**	4	5
2	1	3	**5**	**4**	2	1	2	3	**4**	**2**	5	1	2	2	**3**	**4**	5	1	2	2	**3**	**4**	5
2	1	3	4	**5**	**2**	1	2	3	2	**4**	**5**	1	2	2	3	**4**	**5**	1	2	2	3	**4**	**5**
2	1	3	4	2	5	1	2	3	2	4	5	1	2	2	3	4	5	1	2	2	3	4	5

où les cases entourées en gras sont celles qui sont comparées.

3.2. Implémentation possible

Nous pouvons proposer l'implémentation suivante en Python, impérative :

```
def remontée(tab):
    """
    Effectue un passage de "remontée de bulle" sur le tableau fourni.
    Renvoie la position de la dernière permutation effectuée.
    """
    n, permutation = len(tab), 0
    for i in range(n-1):
        if tab[i] > tab[i+1]:  # l'inégalité stricte assure le carac-
tère en place
            tab[i], tab[i+1], permutation = tab[i+1], tab[i], i
    return permutation
```

```
def tri_bulles(tab):
    """
    Procédure qui trie en place un tableau de nombres fourni.
    """
    j = len(tab)
    while j > 0:
        j = remontée(tab)
```

3.3. Caractéristiques

La complexité du tri à bulles se calcule simplement : la fonction remontée effectue n tours de boucle, donc son temps de calcul s'écrit $D_n = an$; elle renvoie la position j à laquelle a eu lieu la dernière permutation, ainsi la boucle while fera :

- un seul tour si aucune permutation n'a lieu dès le début (le tableau est déjà trié)
- n tours si chaque élément doit remonter tout le tableau pour être positionné (le tableau est trié en ordre inverse)

d'où une complexité $\mathcal{O}(n)$ dans le meilleur cas (tableau trié) et $\mathcal{O}(n^2)$ dans le pire cas (tableau trié en ordre inverse). En moyenne, la complexité reste en $\mathcal{O}(n^2)$, malgré la possibilité d'un arrêt plus rapide en cas d'ordre favorable.

Par ailleurs, on constate que le tri à bulles est naturellement en place (il s'appuie sur des permutations au sein du tableau, comme le tri sélection et insertion) et le choix d'une comparaison stricte dans la procédure de remontée permet d'assurer le caractère stable.

3.4. Variante : tri cocktail

Le tri à bulles va rapidement amener en place les éléments majorants du tableau qui se trouvent initialement dans les premières positions (on les appelle les « lièvres ») ; en revanche, les minorants initialement en fin de tableau ne reculeront que très lentement, en étant progressivement repoussés par les éléments plus grands (on les appelle les « tortues »). Une façon d'exploiter cette caractéristique est de faire les permutations vers la remontée, puis à la descente, de façon alternée ; ainsi on s'approche plus rapidement de l'état trié.

En pratique, on devine que cette amélioration fait gagner, en ordre de grandeur, un facteur 2 sur la complexité (le calcul exact est plus complexe) ; mais cela laisse tout de même des performances inférieures à celles du tri insertion.

IV. Tri rapide (ou tri pivot)

4.1. Présentation

Le tri rapide est un des algorithmes les plus utilisés : il s'appuie sur la méthode du diviser-pour-régner. La méthode est la suivante :

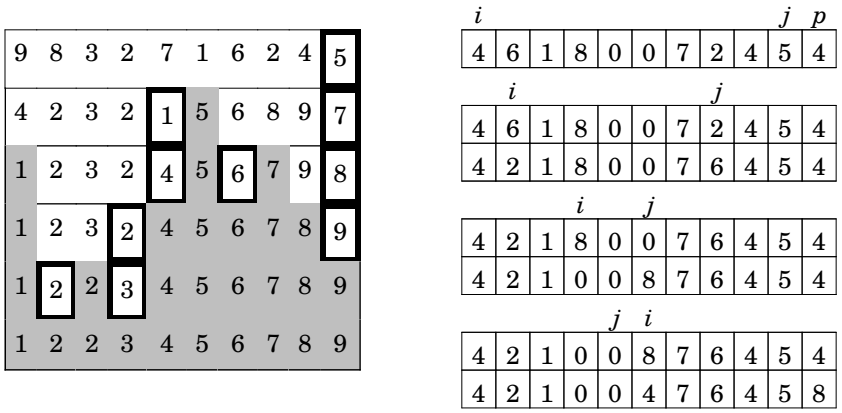
- étant donné un tableau de taille n, choisir un pivot *piv* (en général la valeur de la dernière case) ;
- **partitionner le tableau**, c'est-à-dire le réarranger de manière à avoir toutes les cases de valeur plus petite que *piv* à gauche de toutes celles plus grandes que *piv* ;
- on place *piv* entre les deux ;
- on trie récursivement les sous-tableaux ainsi délimités.

L'exécution des appels récursifs conduit une exécution comme celle de la figure 1(a), où le pivot est encadré et les cases fixées (anciens pivots) sont grisées :

Toute la difficulté réside donc dans la procédure de partition. Une méthode classique est la suivante :

- on parcourt le tableau suivant les indices i, et on s'arrête au premier élément plus grand que le pivot ;
- on parcourt le tableau à l'envers suivant les indices j, et on s'arrête au premier élément plus petit que le pivot ;
- on les permute ;
- on continue jusqu'à ce que $i \geqslant j$; dans ce cas, on permute le pivot avec la case i, et on renvoie i.

L'exécution de cet algorithme de partition est illustrée sur l'exemple de la figure 1(b).



(a) Exemple d'appels récursifs (b) Exemple de partition

Figure 1 : Tri rapide : exemple

4.2. Implémentation possible

```
def partitionner(tab, début, fin):
    """
    Partitionne le sous-tableau fourni entre les indices [début:fin]
    """
    piv = tab[fin-1]
    i, j = début, fin-2  # deux pointeurs : gauche et droite
    while True:
        while i < fin and tab[i] <= piv:
            i += 1
        while j > début-1 and tab[j] >= piv:
            j -= 1
        if i < j:
            tab[i], tab[j] = tab[j], tab[i]
        else:
            tab[i], tab[fin-1] = tab[fin-1], tab[i]
            return i

def tri_rapide(tab):
    tri_rapide_interne(tab, 0, len(tab))

def tri_rapide_interne(tab, i0, i1):
    if i1 - i0 > 1:  # cas de base (inversé)
        j = partitionner(tab, i0, i1)
        tri_rapide_interne(tab, i0, j)
        tri_rapide_interne(tab, j+1, i1)
```

4.3. Caractéristiques

De façon évidente, le tri rapide est en place, et c'est même son principal avantage sur le tri fusion : il économise de nombreuses copies. En revanche, il n'est clairement pas stable : la procédure de partition, telle qu'elle est présentée ici, peut permuter des éléments qui étaient initialement ordonnés.

Sa complexité est nettement différente entre le meilleur et le pire cas :

- dans le meilleur cas, le pivot conduit à une séparation en deux sous-tableaux de taille équivalente, ce qui permet en effet une bonne accélération du travail. Dans ce cas, le calcul pour le tri par partition-fusion s'applique et on retrouve une complexité $\mathcal{O}(n\log_2(n))$, mais avec des constantes multiplicatives plus petites grâce au caractère en place ;
- dans le pire cas, on coupe toujours le tableau en des sous-tableaux très inégaux : le pivot est seul d'un côté et tous les autres restent de l'autre. On voit donc que l'algorithme tel que nous l'avons donné est particulièrement inefficace...lorsque le tableau est déjà trié ! On a alors $n-1$, puis $n-2$, puis $n-3$...comparaisons, donc au total $\sum_{i=n}^{1}(i-1) = \mathcal{O}(n^2)$ comparaisons et une complexité en $\mathcal{O}(n^2)$.

On comprend donc que la complexité est dépendante du choix du pivot, et qu'on ne peut pas choisir **a priori** le pivot optimal. Cependant, la complexité dans le pire cas est rarement atteinte : le cas moyen garde une complexité très proche de celle du meilleur cas ; c'est pour cela que cet algorithme est un des plus utilisés en pratique.

4.4. Variantes

D'après ce qui précède, on comprend que le choix du pivot peut avoir une influence sur le temps de calcul. Il y a donc plusieurs façons de choisir le pivot :

- soit, comme nous l'avons fait, on choisit arbitrairement le pivot toujours au même endroit (première position, dernière position, milieu du tableau...). On comprend qu'à moins d'un ordre particulièrement agaçant dans le tableau d'entrée, on aura en moyenne un comportement correct
- soit on choisit un élément aléatoirement à chaque fois ; dans ce cas ce n'est pas l'ordre de départ mais la suite aléatoire de choix qui peut conduire au pire cas, cependant on intuite que celui-ci sera rare puisqu'il faut alors tirer aléatoirement soit le plus petit soit le plus grand élément à chaque itération
- soit on estime à l'aide d'un algortihme auxilliaire la médiane du tableau pour assurer de choisir de façon à peu près optimale le pivot ; cependant cette estimation a elle-même un coût. Il existe un algorithme adapté à cette problématique, dit de la « médiane des médianes », qui présente une complexité en $\mathcal{O}(n)$. Bien que cette approche soit en théorie meilleure (car elle évite à coup sûr le pire cas), elle est généralement dépassée dans la pratique par le choix aléatoire du pivot.

Un autre variante consiste à combiner le tri rapide avec des tris plus efficaces dans des cas particuliers. Par exemple, on peut effectuer le tri rapide en arrêtant la récursivité pour des tableaux de taille modeste environ 10 éléments) ; le tableau est alors « trié par blocs » ; on utilise le tri par insertion pour finaliser le travail dans ce grand tableau ainsi presque trié.

V. Tri par comptage

5.1. Présentation

Le tri par comptage suit une méthode assez simple, mais est limité au tri de nombres entiers entre des bornes connues, ce qui est une précondition très forte. Le principe consiste à compter les occurrences de chaque valeur possible (comme pour construire un histogramme des valeurs) et à réécrire ensuite le tableau trié en écrivant chaque valeur le bon nombre de fois.

5.2. Implémentation possible

On propose l'implémentation suivante pour des tableaux ne contenant que des entiers compris entre 0 (inclus) et *maxi* (exclus).

```
def créer_histogramme(tab, maxi):
    hist = [0]*maxi
    for elt in tab:
        hist[elt] += 1
    return hist

def tri_comptage(tab, maxi):
    hist = créer_histogramme(tab, maxi)
    tab_trié = [0] * len(tab)
    i = 0
    for val in range(maxi):
        i_val = 0
        while i_val < hist[val]:
            tab_trié[i] = val
            i_val += 1
            i += 1
    return tab_trié
```

5.3. Caractéristiques

Le tri par comptage n'est pas en place, et il n'est pas stable, même si cette propriété a peu d'intérêt pour ce tri : la stabilité est importante pour des tableaux de données agrégées complexes, pas pour des entrées qui doivent être des entiers purs.

La complexité est intéressante à étudier : pour un tableau de taille n ne pouvant contenir que k valeurs distinctes, on s'aperçoit qu'elle est en $\mathcal{O}(n+k)$, soit – semble-t-il – meilleur que le tri rapide ! On voit immédiatement la contrainte associée au tri par comptage (uniquement des entiers, de valeurs possibles connues à l'avance), ainsi que la demande supplémentaire d'espace mémoire (un tableau de taille k si on fait le tri en place).

Il se cache un résultat plus profond derrière cette remarque. Il est prouvé que les tris utilisant des comparaisons ne peuvent pas avoir une complexité meilleure que $\mathcal{O}(n \log_2 n)$: c'est le cas de tous les tris que nous avons étudiés, sauf le tri par comptage, mais c'est finalement cohérent : le tri par comptage ne compare jamais les éléments entre eux. C'est cette spécificité qui le rend à la fois plus contraignant, adapté uniquement à quelques problèmes restreints, mais plus performant dans le cadre de ces problèmes.

On peut adapter le tri par comptage à d'autres types de clés que les entiers purs : on stocke les occurrences dans un dictionnaire, mais il faut alors trier les clés avec un algorithme par comparaisons. La complexité totale est alors $\mathcal{O}(n + k \log_2 k)$. Ce genre de situation se rencontre très rarement dans la pratique...