Leçon d'informatique : tris rapides

S. Benlhajlahsen - PCSI$_1$

Sommaire

I Retour sur les tris classiques **1**

II Tri-fusion **1**

III Tri pivot ou tri rapide (Quicksort) **6**

IV Vérification expérimentale de la complexité **10**

Extrait du programme :

Thèmes	Exemples d'activité, au choix du professeur et non exigibles des étudiants. Commentaires
Tris	Algorithmes quadratiques : tri par insertion, par sélection. Tri par partition-fusion. Tri rapide. Tri par comptage. *On fait observer différentes caractéristiques (par exemple, stable ou non, en place ou non, comparatif ou non, etc).*

I Retour sur les tris classiques

Les tris par sélection et par insertion ont été écrits de manière itérative mais il peuvent aussi s'écrire de manière récursive. On peut les résumer par :

1. On sépare un élément du reste :
   - le dernier élément dans le cas du tri par insertion,
   - le plus petit élément dans le cas du tri par sélection.
2. On trie les éléments restants de manière récursive.
3. On ajoute l'élément séparé :
   - on l'insère dans le cas du tri par insertion,
   - c'est automatique dans le cas du tri par sélection.

On va généraliser ce schéma en n'imposant plus de couper la liste de taille n en un élément et une liste de taille $n-1$ mais en séparant en deux parties de tailles quelconques.

1. On sépare la liste en deux sous-listes `L1` et `L2`.
2. On trie `L1` et `L2` de manière récursive
3. On assemble `L1` et `L2` pour obtenir une liste triée (voir figure 1).

Dans le cas du tri par insertion la séparation est facile, on isole un terme mais l'assemblage est difficile car on doit insérer ce terme à sa place. Par contre, dans le cas du tri par sélection, la séparation est difficile, on doit chercher la plus petit élément mais l'assemblage est facile car les éléments sont à leur place.
Il serait idéal de trouver un tri pour lequel séparation et assemblage sont simples mais cela semble impossible. Nous allons proposer deux tris : pour l'un la séparation est immédiate mais l'assemblage est difficile, c'est le **tri-fusion**, pour l'autre la séparation sera la partie coûteuse, c'est **le tri rapide** (voir figure 2)..
On supposera que les éléments des listes à trier sont comparés à l'aide d'une fonction, nommée `plusGrand`, qui renvoie `True` si le premier argument est strictement supérieur au second. Dans le cas d'éléments simples, la fonction `plusGrand` pour être éventuellement remplacé par l'opérateur `>=` (voir figure 3).

Rappel : Le « strictement supérieur » permet la stabilité du tri.

II Tri-fusion

II.A Principe

Le tri-fusion est une application du principe **diviser pour régner** : on sépare les données en deux parts presque égales, on traite chaque partie, on rassemble (voir figure 4). La dernière étape de ce schéma est l'assemblage des deux listes triées. On en montre les détails en figure 5.

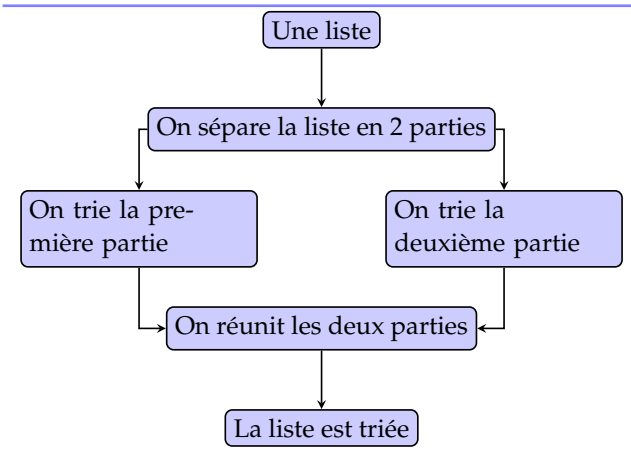


FIGURE 1

	Séparation facile, assemblage difficile	Séparation difficile, assemblage facile
Un singleton et le reste	Tri par insertion	Tri par sélection
Deux parties tailles quelconques	Tri fusion	Tri pivot

FIGURE 2

```
1 def plusGrand(x1, x2):
2     """Entree : deux nombres
3        Sortie : True ou False selon que x1 > x2 ou non"""
4     return x1 > x2
```

FIGURE 3

II.B Écriture en python

Lors de la séparation de la liste en deux, la taille des listes diminue sauf dans le cas d'une liste de longueur 0 ou 1. C'est un **cas terminal**, dans ce cas on renvoie une copie de la liste. On voit que les éléments sont placés sans qu'il semble possible de le faire avec des échanges simples ; on les place dans une nouvelle liste, ce qui impose de définir des listes supplémentaires.

Copie profonde (`deecopy`) : Il faut rappeler que le symbole d'affectation `a=b` entre deux entiers `a` et `b` correspond à une vraie copie : `a` et `b` sont alors deux variables indépendants. Par contre, l'instruction `L2 = L1` correspond à une copie superficielle, c'est-à-dire qu'elle copie uniquement l'adresse des valeurs mais ne crée par deux listes indépendantes.

Il reste à écrire la fusion de deux liste ordonnées. L'idée, illustrée en figure 6, est de placer, dans l'ordre, les éléments des deux listes en choisissant le plus petit des éléments non encore placés. Dans l'exemple les 5 premiers éléments sont ajoutés en comparant effectivement les premiers éléments non encore utilisés dans chaque liste. Par contre les deux derniers éléments sont placés sans comparaison car la première liste a complètement été placée. On donne une implémentation en figure 7.

Explication

- Ligne 8 : on initialise la liste à renvoyer.
- Ligne 9-10 : dans les listes à fusionner on doit savoir où sont les éléments à comparer. Ces éléments sont indiqués par deux indices de position `pos1` et `pos2`. Ils indiquent le premier élément de chaque liste non encore placé dans la liste finale. Comme les deux listes sont triées le plus petit élément restant est à l'une de ces deux positions.
- Ligne 11 : on répète autant de fois qu'il y des éléments restant dans les deux listes.
- Lignes 12-14 : si le plus petit élément restant est dans la première liste, on l'ajoute au résultat et on incrémente la position du premier élément à comparer.
- Lignes 15-17 : si le plus petit élément restant est dans la seconde liste on l'ajoute au résultat.
- Ligne8 18-21 : en sortie de la boucle une des deux liste a été complètement utilisée et on ajoute ce qui reste de l'autre au résultat.

Remarque : On a employé une boucle `while`, on peut préférer la sécurité d'une boucle `for` (voir figure 8).

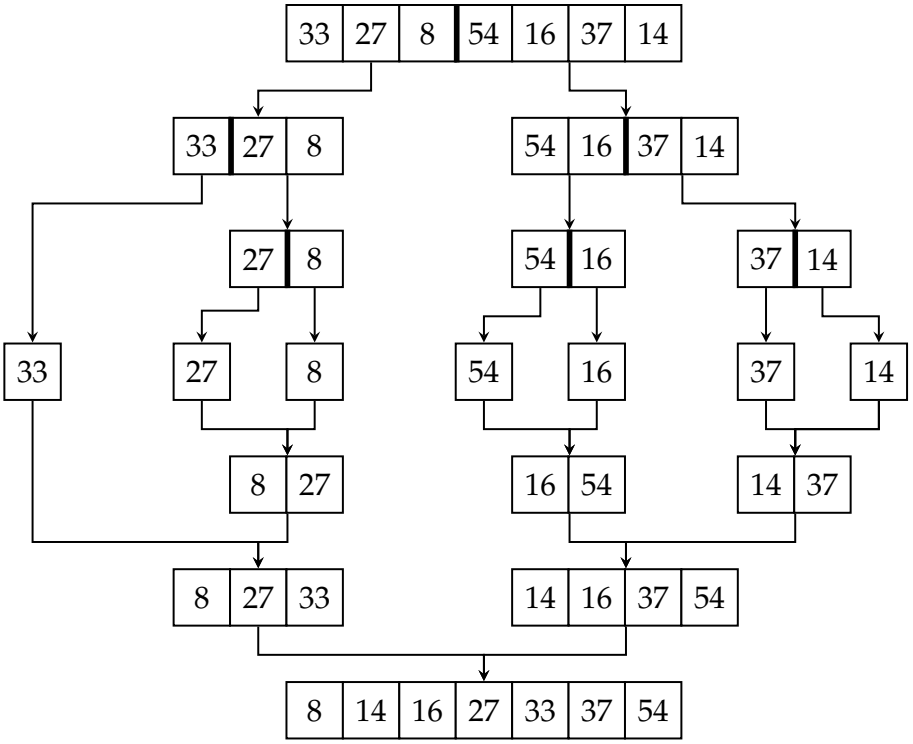


FIGURE 4

II.C Analyse

II.C.1 Terminaison

- `fusion` termine car `i1+i2` augmente de 1 au moins à chaque étape, on parvient donc à `pos1 >= n1` ou `pos2 >= n2` après un nombre fini de passages. Dans le cas d'une boucle `for` elle termine naturellement.
- La terminaison de la fonction récursive `triFusion` se fait par récurrence.
  - Elle termine directement si la liste a moins d'un élément.
  - Si elle termine pour les listes de moins de $n-1$ éléments avec $n \geq 2$, l'appel de la fonction pour une liste de taille n fait appel à la fonction pour des listes de taille $\left\lfloor \frac{n}{2} \right\rfloor$ et $n - \left\lfloor \frac{n}{2} \right\rfloor$.
    - Pour n pair ($n = 2p$), on a $\left\lfloor \frac{n}{2} \right\rfloor = n - \left\lfloor \frac{n}{2} \right\rfloor = p < 2p = n$ car p est non nul pour $n \geq 2$.
    - Pour n impair ($n = 2p+1$), on a $\left\lfloor \frac{n}{2} \right\rfloor = p < n - \left\lfloor \frac{n}{2} \right\rfloor = p + 1 < 2p + 1 = n$.

    Dans tous les cas les appels récursifs terminent d'après l'hypothèse de récurrence.
    Comme `fusion` termine on en déduit que `triFusion` termine pour les liste de taille n.
  - La fonction `triFusion` termine pour toutes les listes.

II.C.2 Preuve/correction

- On commence par la preuve de la fusion On suppose que les listes `L1` et `L2` sont triées. Une propriété vérifiée à chaque passage dans la boucle `while` est que la liste `resultat` est triée et tous ses éléments sont majorés par les éléments de `l1[pos1:n1]` et de `l2[pos2:n2]`.
- On peut alors prouver le tri. Une liste de taille 0 ou 1 est recopiée et est déjà triée, l'algorithme est correct dans ce cas. On suppose que la fonction renvoie une liste triée avec les mêmes élément pour toute liste de taille majorée par $n-1$. On considère une liste de taille n. Les listes extraites sont de taille strictement inférieure à n donc les listes `L1` et `L2` sont triées et contiennent les éléments des deux listes extraites. La preuve de la fusion montre alors que la liste renvoyée est triée et contient tous les éléments de la liste initiale.

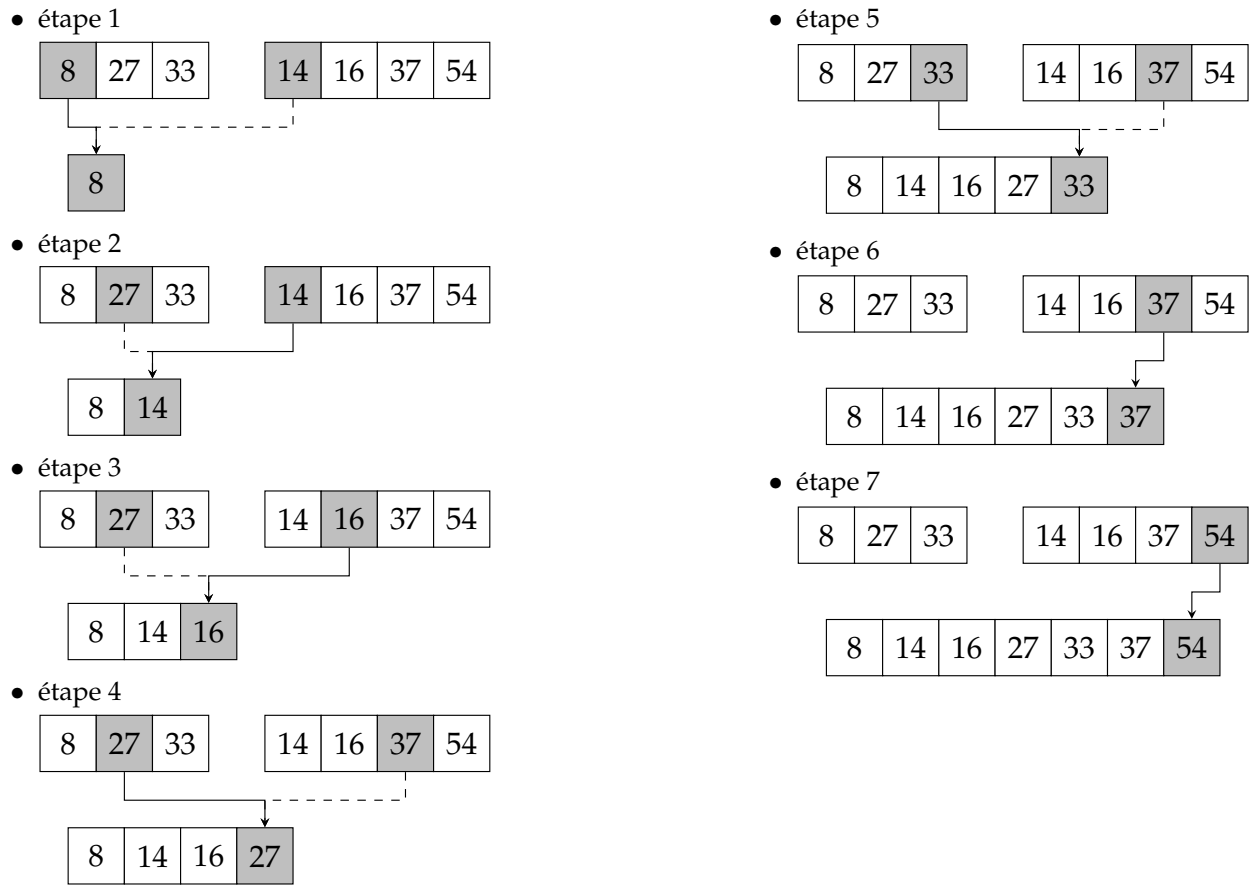


FIGURE 5

II.C.3 Complexité

La complexité est le nombre de comparaisons d'éléments de la liste. La complexité de la fusion de deux listes de taille respectives n_1 et n_2 est au plus $n_1 + n_2$ car on fait au plus une comparaison pour chaque i dans la boucle (en fait au plus $n_1 + n_2 - 1$).
Lors du tri fusion, on sépare en deux listes de tailles respectives $n_1 = \left\lfloor \frac{n}{2} \right\rfloor$ et $n_2 = n - \left\lfloor \frac{n}{2} \right\rfloor$ que l'on trie puis on fusionne les listes triées.

- $n_1 = n_2 = p$ pour n pair, $n = 2p$,
- $n_1 = p$, et $n_2 = p + 1$ pour n impair, $n = 2p + 1$

La complexité du tri, $C(n)$, vérifie donc $C(n) \leq C(n_1) + C(n_2) + n$.

Complexité du tri-fusion La complexité du tri fusion d'une liste de taille n en nombre de comparaisons d'éléments vérifie $C(n) = \mathcal{O}(n \log_2(n))$.

Démonstration :

On va montrer par récurrence sur p que $C(n) \leq p \cdot 2^p$ pour $n \leq 2^p$.

- Pour $p = 0$ cela découle de $C(0) = C(1) = 0$.
- Si la propriété est vraie pour p avec $n \leq 2^p \leq 2^{2p+1}$.
  - Pour n pair, $n_1 = n_2 = \frac{n}{2} \leq 2^p$,
  - pour n impair, on a $n \leq 2^{p+1} - 1$ et $n_1 < n_2 = \frac{n+1}{2} \leq 2^p$.

  On en déduit, d'après l'hypothèse de récurrence, $C(n_1) \leq p2^p$ et $C(n_2) \leq p2^p$ d'où

$$C(n) \leq C(n_1) + C(n_2) + n \leq p2^p + p2^p + 2^{p+1} = (p+1)2^{p+1} \text{ : la propriété est vraie pour } p+1$$

```python
from copy import deepcopy
def triFusion(liste):
    """Entree : une liste
       Sortie : une liste triée avec les mêmes éléments"""
    n = len(liste)
    if n <= 1:
        return deepcopy(liste)
    else:
        L1 = triFusion(liste[0:n//2])
        L2 = triFusion(liste[n//2:n])
        return fusion(L1,L2)
```

FIGURE 6

```python
def fusion(L1, L2):
    """Entree : deux listes
       Requis : les listes sont triées
       Sortie : une liste triée contenant tous
                les éléments des deux listes initiales"""
    n1 = len(L1)
    n2 = len(L2)
    resultat = []
    pos1 = 0
    pos2 = 0
    while pos1 < n1 and pos2 < n2:
        if plusGrand(L2[pos2],L1[pos1]):
            resultat.append(L1[pos1])
            pos1 = pos1 + 1
        else:
            resultat.append(L2[pos2])
            pos2 = pos2 + 1
    if pos1 == n1:
        return resultat + L2[pos2: n2]
    else:
        return resultat + L1[pos1: n1]
```

FIGURE 7

- Si p est choisi tel que $2^{p-1} < n \leq 2^p$ on a $2^p \leq 2n$ et $p \leq 1 + \log_2(n)$ d'où

$$C(n) \leq \big(1 + \log_2(n)\big)2n = \mathcal{O}\big(n \log_2(n)\big)$$

```python
def fusion(L1, L2):
    """Entree : deux listes
       Requis : les listes sont triées
       Sortie : une liste triée contenant tous
                les éléments des deux listes initiales"""
    n1 = len(L1)
    n2 = len(L2)
    resultat = []
    pos1 = 0
    pos2 = 0
    while pos1 < n1 and pos2 < n2:
        if plusGrand(L2[pos2],L1[pos1]):
            resultat.append(L1[pos1])
            pos1 = pos1 + 1
        else:
            resultat.append(L2[pos2])
            pos2 = pos2 + 1
    if pos1 == n1:
        return resultat + L2[pos2: n2]
    else:
        return resultat + L1[pos1: n1]
```

FIGURE 8

III Tri pivot ou tri rapide (Quicksort)

Le tri pivot inverse la difficulté : plutôt que fusionner deux sous-listes triées qui viennent d'un découpage arbitraire il découpe la liste selon un **pivot** qui sert de borne pour séparer les éléments selon le pivot : d'un coté les élément plus petits que le pivot, de l'autre les éléments plus grands.
La séparation en deux listes demande donc un travail, par contre l'assemblage des listes triées est immédiat car les éléments de la première sous-liste sont inférieurs aux éléments de la seconde.

III.A Écriture du découpage

Dans le tri fusion, on a renvoyé une nouvelle liste (triée) car la fusion en place est ardue.
Pour le tri rapide, il est possible de réaliser le tri en place car le découpage peut se faire ainsi.
On doit choisir l'élément qui sert de pivot : nous allons utiliser le dernier élément de la liste à trier mais on pourrait utiliser le premier élément de la liste ou un élément choisi au hasard. Pour ce faire, il suffirait d'échanger l'élément choisi et le dernier avant d'appliquer les fonctions ci-dessous.
Comme les sous-listes sont incluses dans la liste de taille n, on devra écrire des fonctions intermédiaires qui travaillent sur une portion de la liste, que l'on nommera **segment**. Il faudra inclure deux paramètres qui correspondent au début et à la fin du segment, on choisira les paramètre a et b tels que les indices utilisés sont ceux allant de a à b, **bornes comprises**.

Découpage : On commence par le découpage en deux sous-segments. On veut transformer un segment de la forme

33	27	8	54	16	31	14	29

en un segment, qui pourrait être

27	8	16	14	29	31	54	33

On peut remarquer que le pivot est à sa place. En effet les termes placés avant lui sont inférieurs et les termes placés ensuite sont supérieurs.

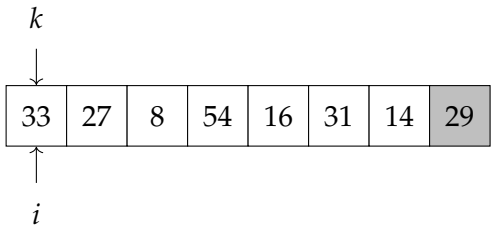
Parcours : On parcourt la liste entre a et $b-1$ puisque le pivot est en b. On maintient 2 variables d'indice :

- la première, i, est l'indice de la boucle `for` qui est la position de l'élément étudié,

- la seconde, k, est la première position après les termes plus petits que le pivot.

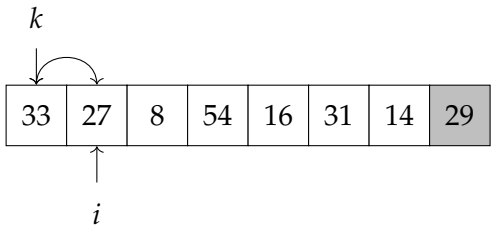
On compare chaque terme au pivot. S'il est plus grand, on le laisse en place.

- étape 1

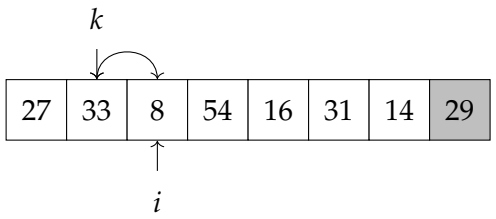


S'il est plus petit, on le permute pour le mettre à la position k puis on incrémente k.

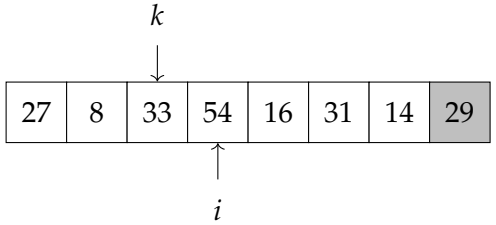
- étape 2



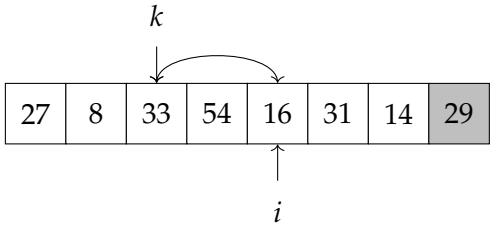
- étape 3



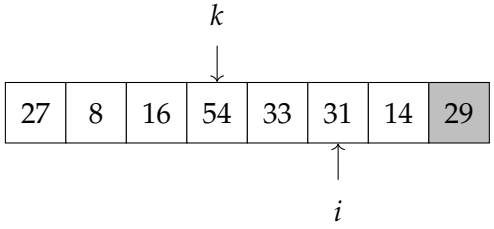
- étape 4



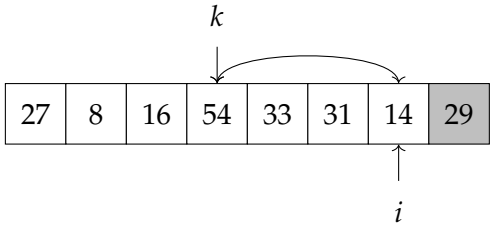
- étape 5



- étape 6

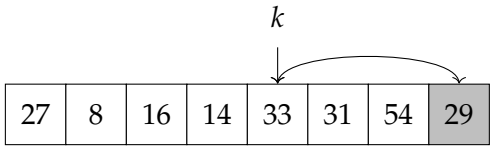


- étape 7



À la fin on permute le pivot à la position k.

- étape 8



On renverra la position finale du pivot car elle servira à définir les deux segments suivants.

Pour permuter deux éléments dans une liste on utilise la fonction classique (voir 9). Ensuite, on donne en figure 10 une fonction qui réalise ce « pivotage ».

III.B Écriture du tri

Pour écrire le tri on utilise une fonction auxiliaire, récursive qui trie entre 2 bornes (voir figure 11).

III.C Analyse

III.C.1 Terminaison

- `pivotage` termine car elle ne fait appel qu'à une boucle `for`.

```python
def echange(liste, i, j):
    temp = liste[i]
    liste[i] = liste[j]
    liste[j] = temp

L = [0, 1, 2, 3, 4, 5]
echange(L, 0, 1)
print(L)     #affiche [1, 0, 2, 3, 4, 5]
```

FIGURE 9

```python
def pivotage(liste,a,b):
    """Entree : une liste et deux entiers
       Requis : 0 <= a <= b < len(liste)
       Sortie : un entier p (a <= p <= b) avec
       le dernier élément placé en p qui sépare
       les élément plus petits et plus grands que lui """
    k = a
    pivot = liste[b]
    for i in range(a, b):
        if plusGrand(pivot, liste[i]):
            echanger(liste, i, k)
            k = k + 1
    echanger(liste, k, b)
    return k
```

FIGURE 10

- On prouve que la fonction récursive `tri_entre` termine par récurrence sur la longueur de la portion à trier car, lors de chaque appel récursif, celle-ci diminue de 1 au moins.
- On en déduit immédiatement que `triRapide` termine.

III.C.2 Preuve

La fonction de séparation doit produire, à chaque étape, quatre parties :

1. les éléments plus petits que le pivot, on a choisit une inégalité large,
2. suivis des éléments strictement supérieurs au pivot,
3. les éléments non traités suivent,
4. le pivot est à la dernière position durant la boucle.

On peut alors prouver que les propriétés suivantes forment un invariant.

$$\mathcal{P}(i)\begin{cases}(1) & \texttt{liste[a:k]} \text{ est majorée par le pivot}\\ (2) & \texttt{liste[k:i]} \text{ est strictement minorée par le pivot}\end{cases}$$

La preuve du tri s'en déduit.

III.C.3 Complexité

- `pivotage(L, a, b)` fait une comparaison pour chaque i entre a et $b-1$, sa complexité, en nombre de comparaisons, est donc $b-a$.
- On note $C(L,a,b)$ le nombre de comparaisons lors de l'appel de `tri_entre(L, a, b)`.
- On a donc $C(L,a,b) = b-a+C(L,a,p-1)+C(L,p+1,b)$ si la fonction `pivotage` a renvoyé p.

```python
def tri_entre(liste, a, b):
    """Entree : une liste et deux entiers
       Requis : 0 <= a <= b < len(liste)
       Sortie : la liste est triee entre a et b"""
    if b > a:
        p = pivotage(liste, a, b)
        tri_entre(liste, a, p-1)
        tri_entre(liste, p+1, b)
```

```python
def triRapide(liste):
    """Entree : une liste
       Sortie : la liste est triee"""
    n = len(liste)
    tri_entre(liste, 0, n-1)
```

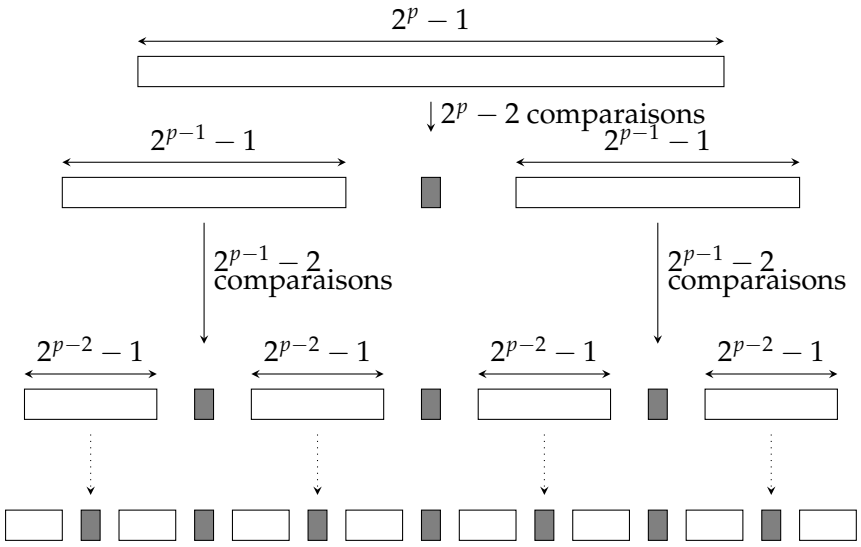
FIGURE 11

Exemple 1 : Dans le cas où `pivotage(L, a, b)` renvoie b à chaque appel, ce qui est le cas lorsque la liste initiale est déjà triée, la relation ci-dessus devient :

$$C(L,a,b) = b - a + C(L,a,b-1) + C(L,b+1,b) = b - a + C(L,a,b-1)$$

car la complexité pour une liste vide est nulle. On en déduit qu'alors $C(L,a,b) = b - a + (b - a - 1) + \cdots + 1$. En particulier $C(L,0,n-1) = \frac{n(n-1)}{2}$: la complexité du tri d'une liste de taille n est un $\mathcal{O}(n^2)$, le tri **n'est pas efficace dans ce cas**.

Exemple 2 : On suppose que $n = 2^p - 1$ et qu'à chaque étape `pivotage(L, a, b)` renvoie un indice situé au milieu, en particulier, s'il y a un nombre impair d'éléments, `pivotage(L, a, b)` renvoie $\frac{a+b}{2}$.



À chaque étape on double le nombre de segments et, entre deux segments, il y a un élément qui a été un pivot. Si on note m_k le nombre de segments de taille $2^k - 1$, il sont séparés par $m_k - 1$ pivots donc on a $2^p - 1 = m_k \cdot (2^k - 1) + m_k - 1 = m_k \cdot 2^k - 1$: on en déduit $m_k = 2^{p-k}$.

Ces m_k segments occasionnent chacun $2^k - 2$ comparaisons. Le nombre de comparaison est donc

$$\begin{aligned} C(n) &= \sum_{k=2}^{p} m_k.(2^k - 2) = \sum_{k=2}^{p} 2^{p-k}.(2^k - 2) \\ &= \sum_{k=2}^{p} (2^p - 2^{p-k+1}) = \sum_{k=2}^{p} 2^p - \sum_{k=2}^{p} 2^{p-k+1} \\ &= (p-1)2^p - \sum_{i=1}^{p-1} 2^i = (p-1)2^p - (2^p - 2) = (p-2)2^p - 2 \end{aligned}$$

On a $2^p = n+1$ donc $p = \log_2(n+1)$ donc $C(n) = \big(\log_2(n+1) - 2\big)(n+1) + 2$, la complexité est un $\mathcal{O}\big(n \log_2(n)\big)$, le **tri est efficace dans ce cas**.

Entre ces deux extrêmes on a le cas moyen.

Complexité moyenne du tri rapide La complexité moyenne du tri rapide d'une liste de taille n en nombre de comparaisons d'éléments est un $\mathcal{O}\big(n \log_2(n)\big)$. Ce résultat doit être connu mais pas sa démonstration.

IV Vérification expérimentale de la complexité

La méthode `time()` du module `time` renvoie la date en seconde au moment de son exécution. La syntaxe ci-dessous permet de calculer la durée d'une suite du tri d'une liste.

```
1 from time import time
2 t1 = time()
3 triFusion(L)
4 t2 = time()
5 duree  = t2-t1
```

On peut alors représenter le temps d'exécution du tri d'une liste de taille n construite aléatoirement (voir figures 12 et 13). En échelle logarithmique, on vérifie la complexité quadratique.

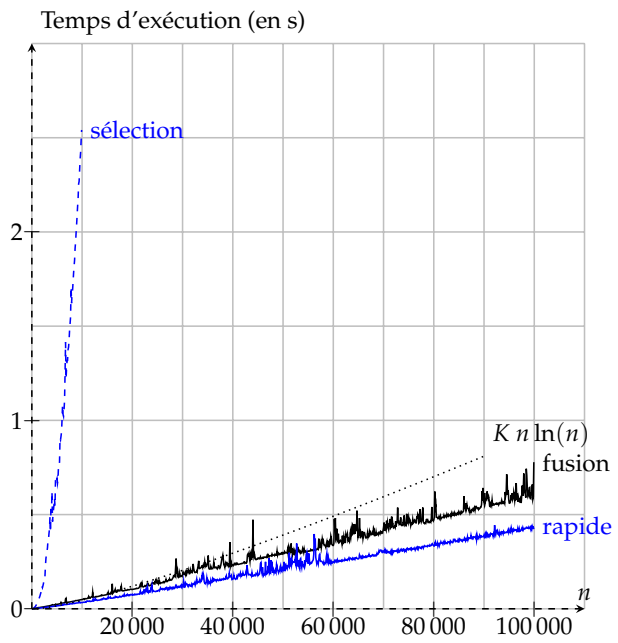


FIGURE 12 – Temps d'exécution en fonction de n en échelle linéaire. On a mis en trait noir le cas du tri fusion et en trait bleu le cas du tri rapide. Enfin, on a rajouté une courbe représentative de $n \mapsto K\,n\,\ln(n)$ en pointillés.

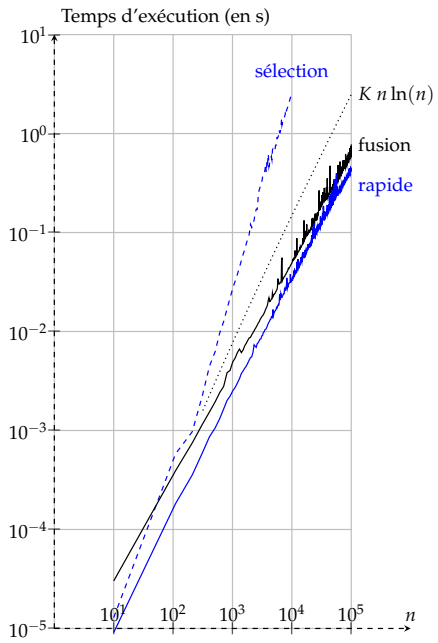


FIGURE 13 – Temps d'exécution en fonction de n en échelle logarithmique. On a mis en trait noir le cas du tri fusion et en trait bleu le cas du tri rapide. Enfin, on a rajouté une courbe représentative de $n \mapsto K\,n\,\ln(n)$ en pointillés.