Algorithmes de tri

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Tris par comparaison

Le coût d'un algorithme de tri dépend de la structure de donnée utilisée : on choisit de trier des tableaux :

- coût d'accès constant à un élément ;
- structure de données mutable : on souhaite un tri en place.

Tris par comparaison

Le coût d'un algorithme de tri dépend de la structure de donnée utilisée : on choisit de trier des tableaux :

- coût d'accès constant à un élément ;
- structure de données mutable : on souhaite un tri en place.

Le coût d'un algorithme de tri dépend du choix des opérations élémentaires qu'on s'autorise :

- comparaisons entre éléments du tableau ;
- permutation de deux éléments : `t[i], t[j] = t[j], t[i]`.

Tris par comparaison

Le coût d'un algorithme de tri dépend de la structure de donnée utilisée : on choisit de trier des tableaux :

- coût d'accès constant à un élément ;
- structure de données mutable : on souhaite un tri en place.

Le coût d'un algorithme de tri dépend du choix des opérations élémentaires qu'on s'autorise :

- comparaisons entre éléments du tableau ;
- permutation de deux éléments : `t[i], t[j] = t[j], t[i]`.

$\preccurlyeq$ est un pré-ordre total lorsque :

- $\forall (x,y) \in E^2, x \preccurlyeq y$ ou $y \preccurlyeq x$;
- $\forall x \in E, x \preccurlyeq x$;
- $\forall (x,y,z) \in E^3, x \preccurlyeq y$ et $y \preccurlyeq z \Rightarrow x \preccurlyeq z$;

Mais deux éléments x et y peuvent vérifier $x \preccurlyeq y$ et $y \preccurlyeq x$ sans avoir $x = y$ (ils sont dits *équivalents*).

Tris par comparaison

Le coût d'un algorithme de tri dépend de la structure de donnée utilisée : on choisit de trier des tableaux :

- coût d'accès constant à un élément ;
- structure de données mutable : on souhaite un tri en place.

Le coût d'un algorithme de tri dépend du choix des opérations élémentaires qu'on s'autorise :

- comparaisons entre éléments du tableau ;
- permutation de deux éléments : `t[i], t[j] = t[j], t[i]`.

$\preccurlyeq$ est un pré-ordre total lorsque :

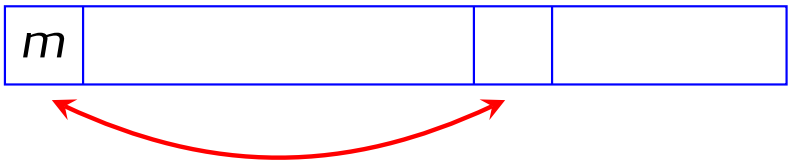
- $\forall (x,y) \in E^2, x \preccurlyeq y$ ou $y \preccurlyeq x$;
- $\forall x \in E, x \preccurlyeq x$;
- $\forall (x,y,z) \in E^3, x \preccurlyeq y$ et $y \preccurlyeq z \Rightarrow x \preccurlyeq z$;

Mais deux éléments x et y peuvent vérifier $x \preccurlyeq y$ et $y \preccurlyeq x$ sans avoir $x = y$ (ils sont dits *équivalents*).
Un algorithme de tri est stable lorsqu'il préserve l'ordre des éléments équivalents : si a_i et a_j sont équivalents et $i < j$, dans le tableau trié a_i sera toujours placé avant a_j.

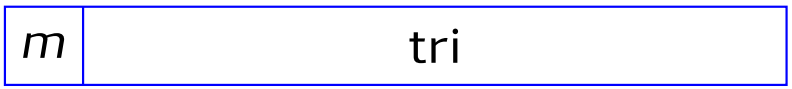
Tri par sélection

– recherche d'un élément minimal :

– permutation avec le premier :



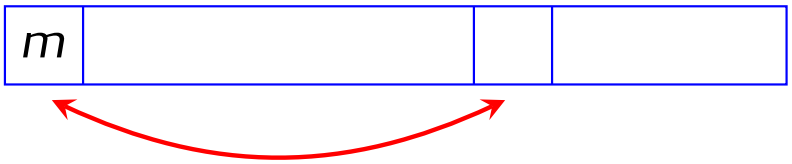
– tri du tableau restant :



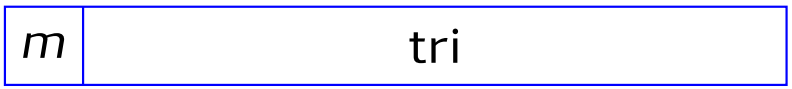
Tri par sélection

– recherche d'un élément minimal :

– permutation avec le premier :



– tri du tableau restant :



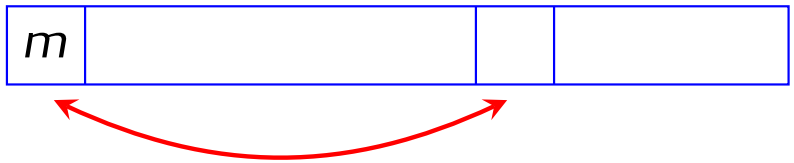

```
def minimum(t, j):
    i, m = j, t[j]
    for k in range(j+1, len(t)):
        if t[k] < m:
            i, m = k, t[k]
    return i

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

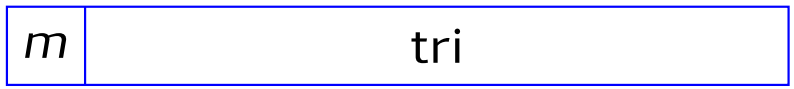
Tri par sélection

– recherche d'un élément minimal :

– permutation avec le premier :



– tri du tableau restant :



Nombre de comparaisons : $\frac{n(n-1)}{2} \sim \frac{n^2}{2}$.

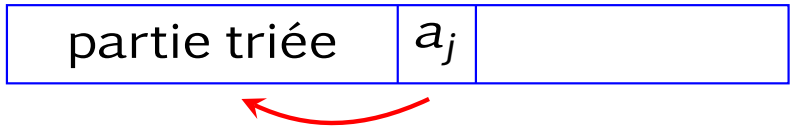
Nombre d'échanges : $\leqslant n-1$.

Il peut présenter un intérêt dans le cas où le coût d'une comparaison est notablement inférieur au coût d'un échange.

Il s'agit d'un algorithme stable à condition, quand il y a plusieurs minimum équivalents, de sélectionner le premier rencontré.

Tri par insertion

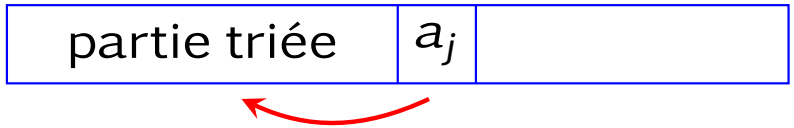
– insertion dans un tableau trié :



– tri du tableau restant :

Tri par insertion

– insertion dans un tableau trié :



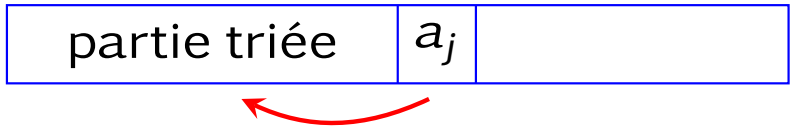
– tri du tableau restant :


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

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

Tri par insertion

– insertion dans un tableau trié :



– tri du tableau restant :

Deuxième version :

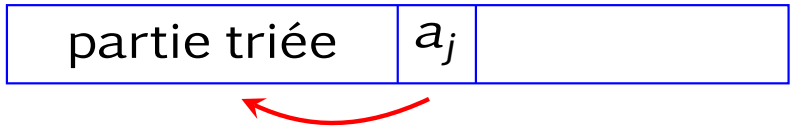
```
def insere(t, j):
    k, a = j, t[j]
    while k > 0 and t[k] < t[k-1]:
        t[k] = t[k-1]
        k -= 1
    t[k] = a
```

(deux fois moins d'affectations utilisées.)

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

Tri par insertion

– insertion dans un tableau trié :



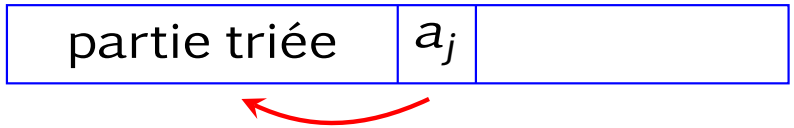
– tri du tableau restant :

Coût dans le pire des cas (liste triée par ordre décroissant).

Nombre de comparaisons et d'échanges : $\frac{n(n-1)}{2} \sim \frac{n^2}{2}$.

Tri par insertion

– insertion dans un tableau trié :



– tri du tableau restant :

Coût dans le pire des cas (liste triée par ordre décroissant).

Nombre de comparaisons et d'échanges : $\frac{n(n-1)}{2} \sim \frac{n^2}{2}$.

Coût en moyenne.

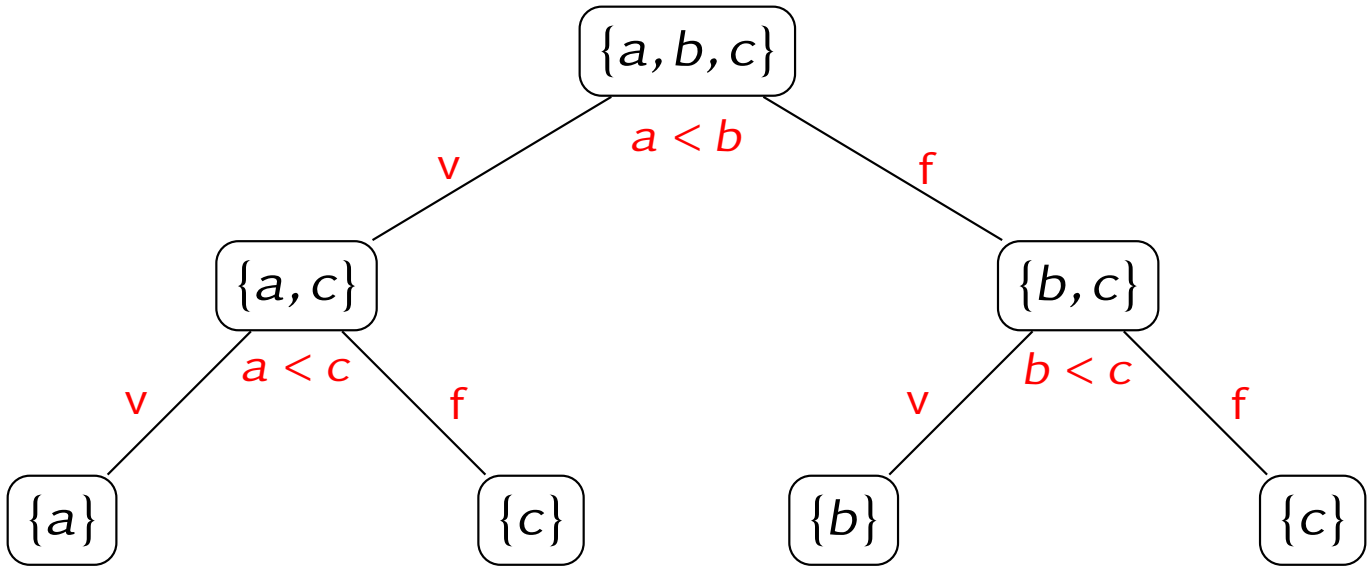
Nombre de comparaisons et d'échanges $\sim \frac{n^2}{4}$.

C'est le nombre moyens d'inversions d'une permutation de $\mathfrak{S}_n$.

Il s'agit d'un algorithme stable à condition de me pas permuter deux éléments voisins et équivalents.

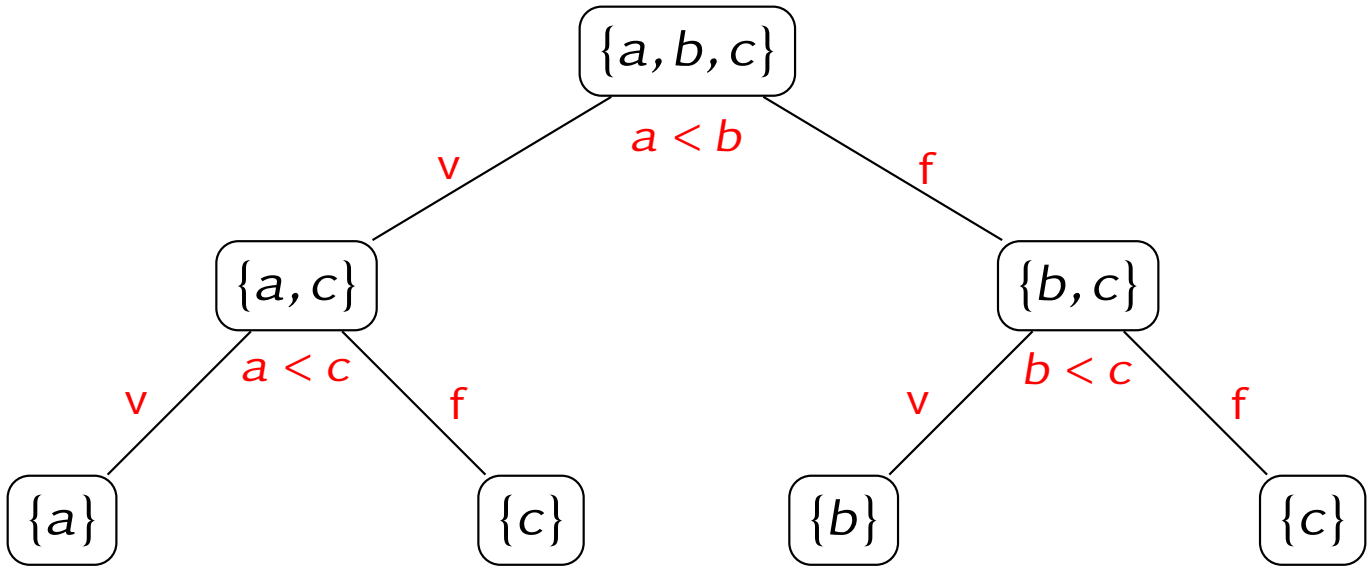
Arbre de décision

Un exemple d'arbre de décision pour calculer $\min(a, b, c)$:



Arbre de décision

Un exemple d'arbre de décision pour calculer $\min(a, b, c)$:



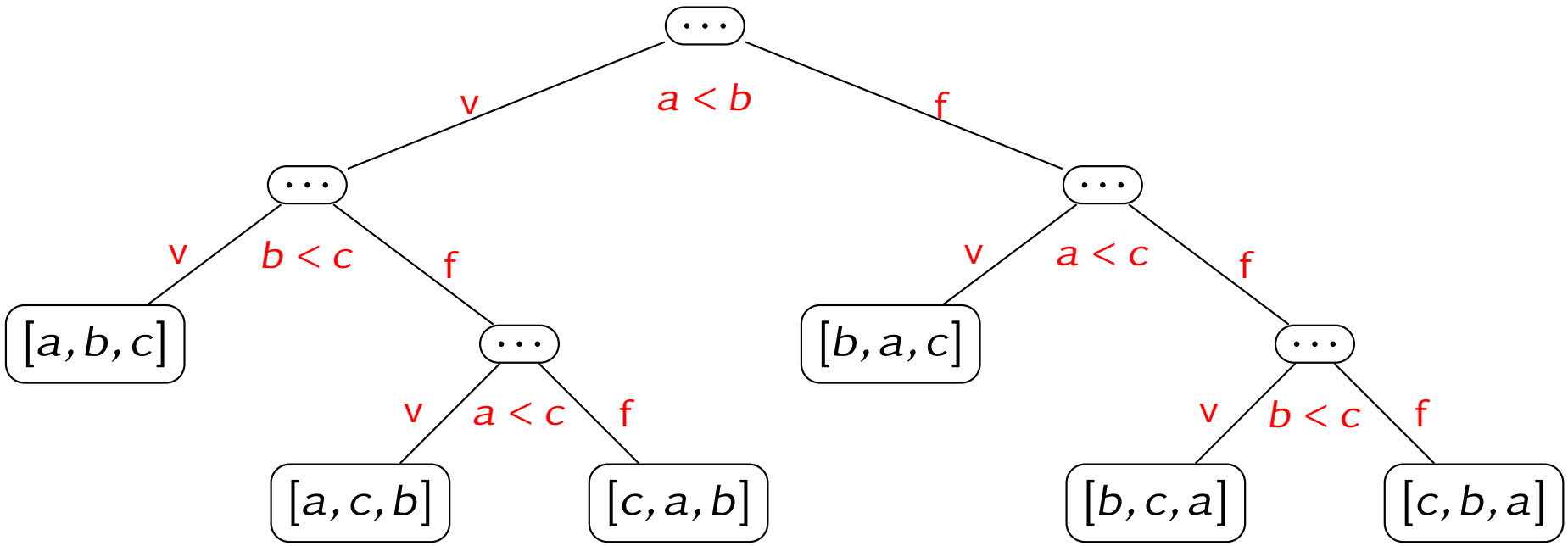
- Un arbre de décision pour un choix dans un ensemble de cardinal n doit posséder au moins n feuilles.
- Un arbre binaire de hauteur h possède au plus 2^h feuilles.

La hauteur h d'un arbre de décision doit donc vérifier :

$$2^h \geqslant n, \text{ soit } h \geqslant \lceil \log n \rceil.$$

Arbre de décision

du tri par insertion de trois éléments



- Meilleur des cas : $[a, b, c]$ (2 comparaisons, pas d'échange) ;
- Pire des cas : $[c, b, a]$ (3 comparaisons, 3 échanges) ;
- Nombre moyen de comparaisons : 8/3 ;
- Nombre moyen d'échanges : 3/2.

Arbre de décision

D'un algorithme de tri par comparaison

Tout algorithme de tri par comparaison utilise dans le pire des cas comme en moyenne au moins $\lceil \log n! \rceil$ comparaisons.

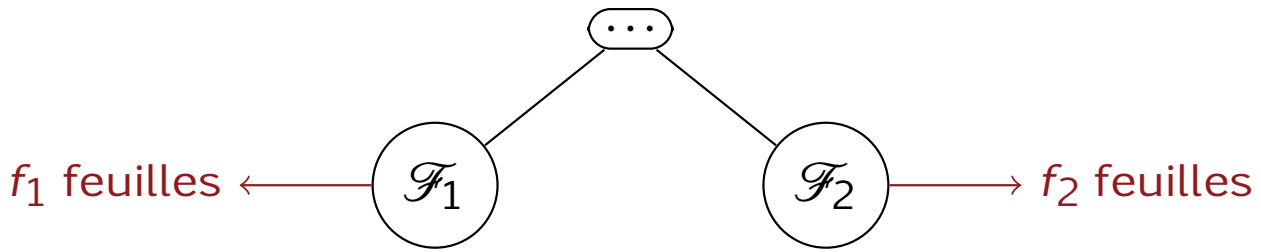
La hauteur h de l'arbre de décision vérifie : $n! \leqslant 2^h$, soit $h \geqslant \lceil \log n! \rceil$, donc il y a au moins un cas pour lequel $\lceil \log n! \rceil$ comparaisons au moins sont nécessaires.

Arbre de décision

D'un algorithme de tri par comparaison

Tout algorithme de tri par comparaison utilise dans le pire des cas comme en moyenne au moins $\lceil \log n! \rceil$ comparaisons.

Soit h_m la profondeur moyenne des feuilles d'un arbre binaire strict $\mathscr{F}$ à f feuilles :



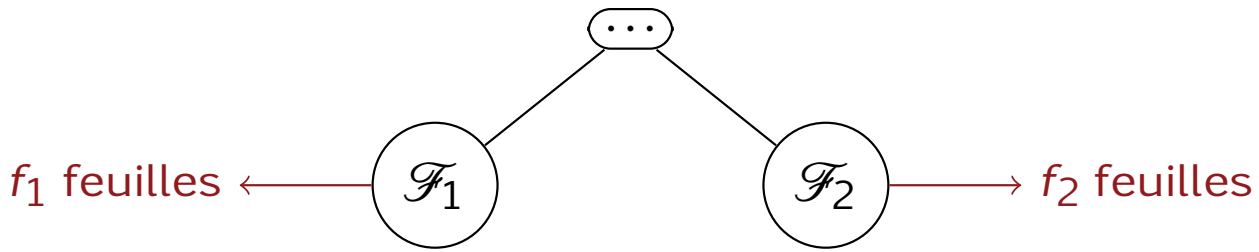
On prouve par récurrence forte que $h_m \geqslant \log f$.

Arbre de décision

D'un algorithme de tri par comparaison

Tout algorithme de tri par comparaison utilise dans le pire des cas comme en moyenne au moins $\lceil \log n! \rceil$ comparaisons.

Soit h_m la profondeur moyenne des feuilles d'un arbre binaire strict $\mathscr{F}$ à f feuilles :



On prouve par récurrence forte que $h_m \geqslant \log f$.

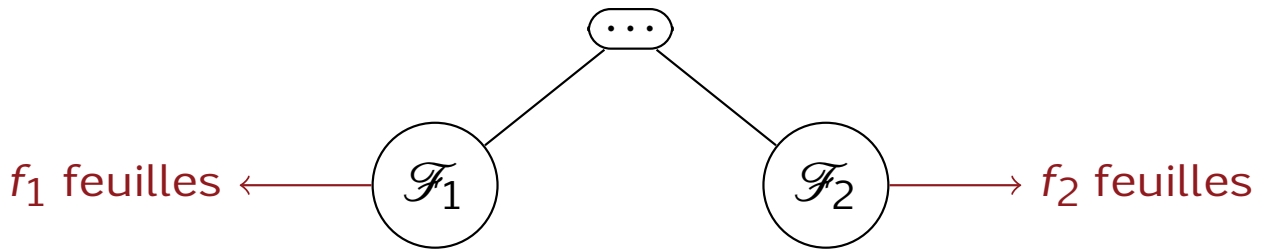
$$h_m = \frac{1}{f}\sum_{x\in\mathscr{F}} h(x) = \frac{1}{f}\sum_{x\in\mathscr{F}_1}\Big(1+h_1(x)\Big)+\frac{1}{f}\sum_{x\in\mathscr{F}_2}\Big(1+h_2(x)\Big) = 1+\frac{1}{f}\sum_{x\in\mathscr{F}_1} h_1(x)+\frac{1}{f}\sum_{x\in\mathscr{F}_2} h_2(x).$$

Arbre de décision

D'un algorithme de tri par comparaison

Tout algorithme de tri par comparaison utilise dans le pire des cas comme en moyenne au moins $\lceil \log n! \rceil$ comparaisons.

Soit h_m la profondeur moyenne des feuilles d'un arbre binaire strict $\mathscr{F}$ à f feuilles :



On prouve par récurrence forte que $h_m \geqslant \log f$.

$$h_m = \frac{1}{f}\sum_{x\in\mathscr{F}} h(x) = \frac{1}{f}\sum_{x\in\mathscr{F}_1}\Big(1+h_1(x)\Big)+\frac{1}{f}\sum_{x\in\mathscr{F}_2}\Big(1+h_2(x)\Big) = 1+\frac{1}{f}\sum_{x\in\mathscr{F}_1} h_1(x)+\frac{1}{f}\sum_{x\in\mathscr{F}_2} h_2(x).$$

Par hypothèse de récurrence,

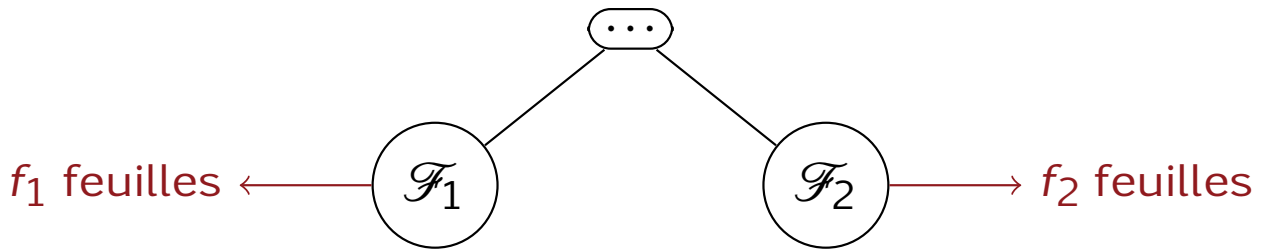
$$h_m \geqslant 1+\frac{1}{f}\Big(f_1\log f_1 + f_2 \log f_2\Big) \geqslant 1+\frac{1}{f}(f_1+f_2)\log\Big(\frac{f_1+f_2}{2}\Big) = \log f.$$

Arbre de décision

D'un algorithme de tri par comparaison

Tout algorithme de tri par comparaison utilise dans le pire des cas comme en moyenne au moins $\lceil \log n! \rceil$ comparaisons.

Soit h_m la profondeur moyenne des feuilles d'un arbre binaire strict $\mathscr{F}$ à f feuilles :



On prouve par récurrence forte que $h_m \geqslant \log f$.

$$h_m = \frac{1}{f}\sum_{x\in\mathscr{F}} h(x) = \frac{1}{f}\sum_{x\in\mathscr{F}_1}\Big(1+h_1(x)\Big)+\frac{1}{f}\sum_{x\in\mathscr{F}_2}\Big(1+h_2(x)\Big) = 1+\frac{1}{f}\sum_{x\in\mathscr{F}_1} h_1(x)+\frac{1}{f}\sum_{x\in\mathscr{F}_2} h_2(x).$$

Par hypothèse de récurrence,

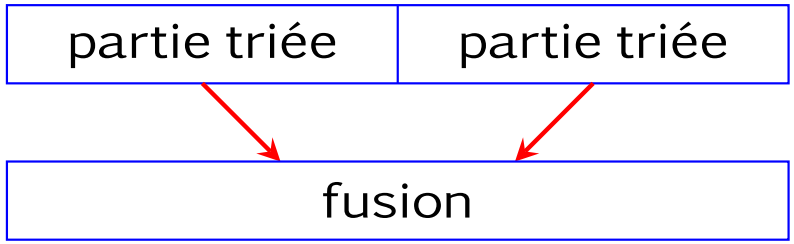
$$h_m \geqslant 1+\frac{1}{f}\Big(f_1\log f_1 + f_2 \log f_2\Big) \geqslant 1+\frac{1}{f}(f_1+f_2)\log\Big(\frac{f_1+f_2}{2}\Big) = \log f.$$

En particulier, la profondeur moyenne des feuilles d'un arbre de décision pour un algorithme de tri par comparaison est supérieure ou égale à $\log(n!) \sim n\log n$.

Tri fusion

– tri des deux moitiés du tableau :

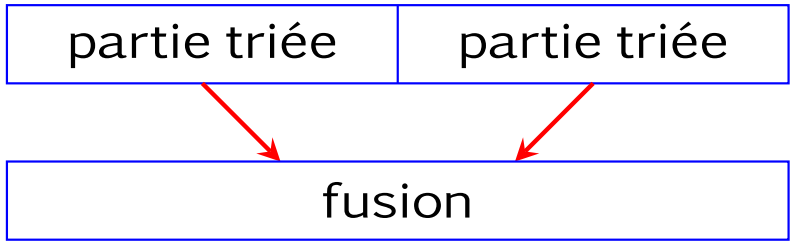
– fusion des parties triées :



Tri fusion

– tri des deux moitiés du tableau :

– fusion des parties triées :

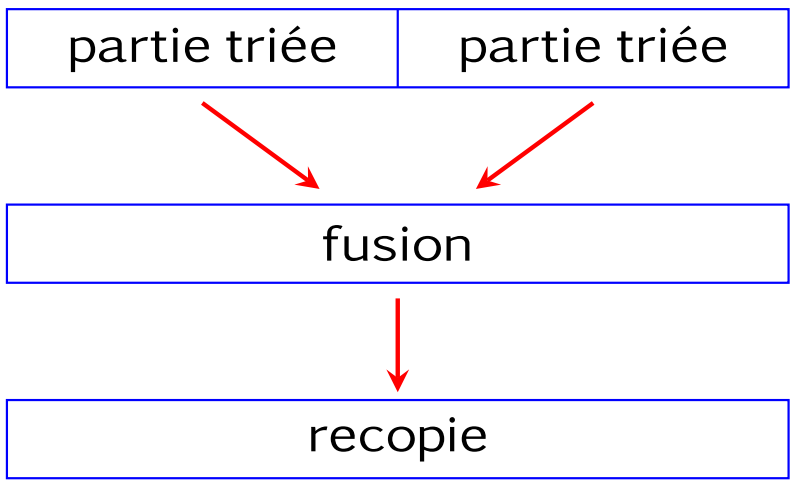


Une difficulté : comment fusionner deux demi-tableaux en place, c'est à dire en s'autorisant uniquement la permutation de deux éléments dans le tableau ?
⟶ On s'autorise l'utilisation d'un tableau provisoire pour y stocker le résultat de la fusion :

– tri des deux moitiés du tableau :

– fusion dans un tableau auxiliaire :

– recopie dans le tableau initial :



Tri fusion

On suppose les sous-tableaux $t[i : j]$ et $t[j : k]$ triés par ordre croissant et on les fusionne dans $aux[i : k]$.

```
def fusion(t, i, j, k, aux):
    a, b = i, j
    for s in range(i, k):
        if a == j or (b < k and t[b] < t[a]):
            aux[s] = t[b]
            b += 1
        else:
            aux[s] = t[a]
            a += 1

def merge_sort(t):
    aux = [None] * len(t)
    def merge_rec(i, k):
        if k > i + 1:
            j = (i + k) // 2
            merge_rec(i, j)
            merge_rec(j, k)
            fusion(t, i, j, k, aux)
            t[i:k] = aux[i:k]
    merge_rec(0, len(t))
```

Tri fusion

Étude de la complexité

Coût spatial : création du tableau *aux* (coût linéaire) et pile d'exécution de la fonction récursive `merge_rec`. Notons $C(n)$ ce dernier coût :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(1).$$

Tri fusion

Étude de la complexité

Coût spatial : création du tableau *aux* (coût linéaire) et pile d'exécution de la fonction récursive `merge_rec`. Notons $C(n)$ ce dernier coût :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(1).$$

Lorsque $n = 2^p$ $u_p = C(2^p)$ vérifie $u_p = 2u_{p-1} + \Theta(1)$ donc $u_p = \Theta(2^p)$ et $C(n) = \Theta(n)$. Le coût spatial est linéaire.

Tri fusion

Étude de la complexité

Coût spatial : création du tableau *aux* (coût linéaire) et pile d'exécution de la fonction récursive `merge_rec`. Notons $C(n)$ ce dernier coût :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(1).$$

Lorsque $n = 2^p$ $u_p = C(2^p)$ vérifie $u_p = 2u_{p-1} + \Theta(1)$ donc $u_p = \Theta(2^p)$ et $C(n) = \Theta(n)$. Le coût spatial est linéaire.

Coût temporel : la fusion de $t[i : j]$ et $t[j : k]$ dans $aux[i : k]$ puis la copie de $aux[i : k]$ vers $t[i : k]$ sont des opérations de coût linéaire donc :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

Tri fusion

Étude de la complexité

Coût spatial : création du tableau *aux* (coût linéaire) et pile d'exécution de la fonction récursive `merge_rec`. Notons $C(n)$ ce dernier coût :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(1).$$

Lorsque $n = 2^p$ $u_p = C(2^p)$ vérifie $u_p = 2u_{p-1} + \Theta(1)$ donc $u_p = \Theta(2^p)$ et $C(n) = \Theta(n)$. Le coût spatial est linéaire.

Coût temporel : la fusion de $t[i : j]$ et $t[j : k]$ dans $aux[i : k]$ puis la copie de $aux[i : k]$ vers $t[i : k]$ sont des opérations de coût linéaire donc :

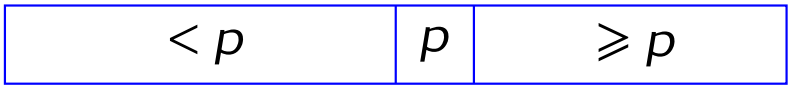
$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

Lorsque $n = 2^p$ $u_p = C(2^p)$ vérifie $u_p = 2u_{p-1} + \Theta(2^p)$ soit : $\frac{u_p}{2^p} = \frac{u_{p-1}}{2^{p-1}} + \Theta(1)$. Par télescopage $\frac{u_p}{2^p} = \Theta(p)$ et $u_p = \Theta(p2^p)$, soit $C(n) = \Theta(n \log n)$.
Le coût temporel est semi-logarithmique.

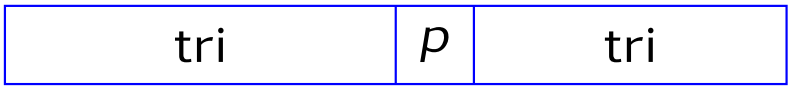
Tri rapide

– choix d'un pivot :

– segmentation :



– appels récursifs :

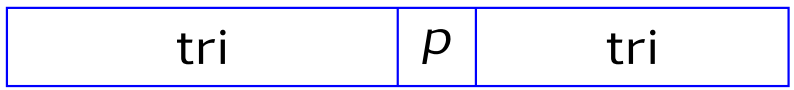


Tri rapide

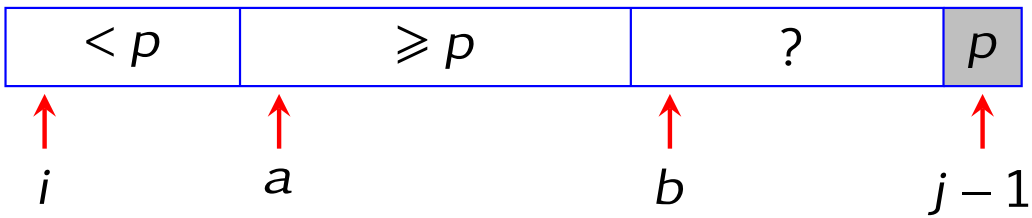
– choix d'un pivot :

– segmentation :

– appels récursifs :



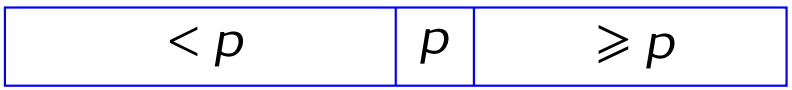
On segmente le tableau $t[i : j]$ en choisissant pour pivot $p = t[j-1]$ et en maintenant l'invariant suivant :



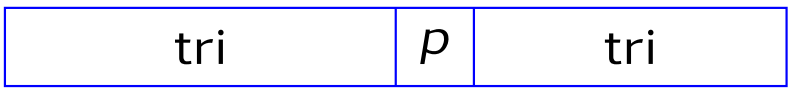
Tri rapide

– choix d'un pivot :

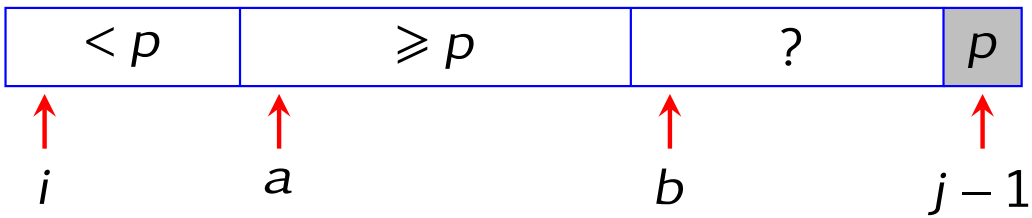
– segmentation :



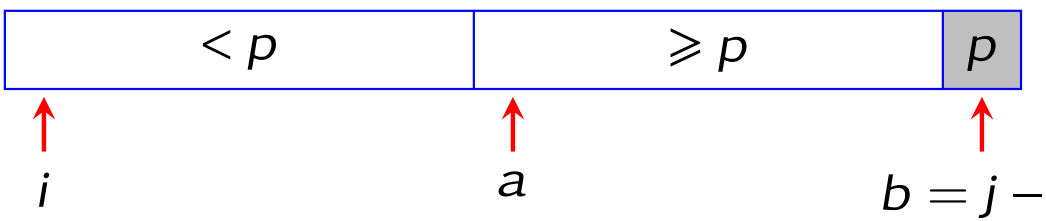
– appels récursifs :



On segmente le tableau $t[i : j]$ en choisissant pour pivot $p = t[j-1]$ et en maintenant l'invariant suivant :

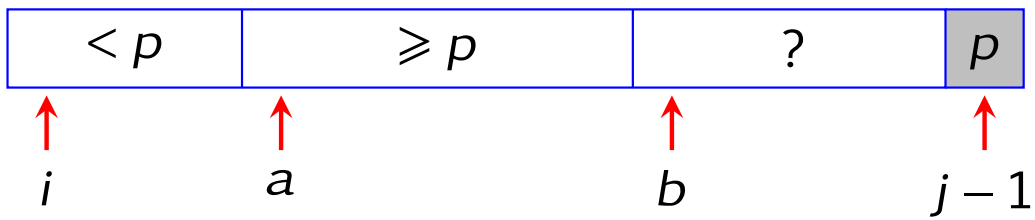


Le processus se termine lorsque $b = j - 1$; on permute alors les cases a et b.

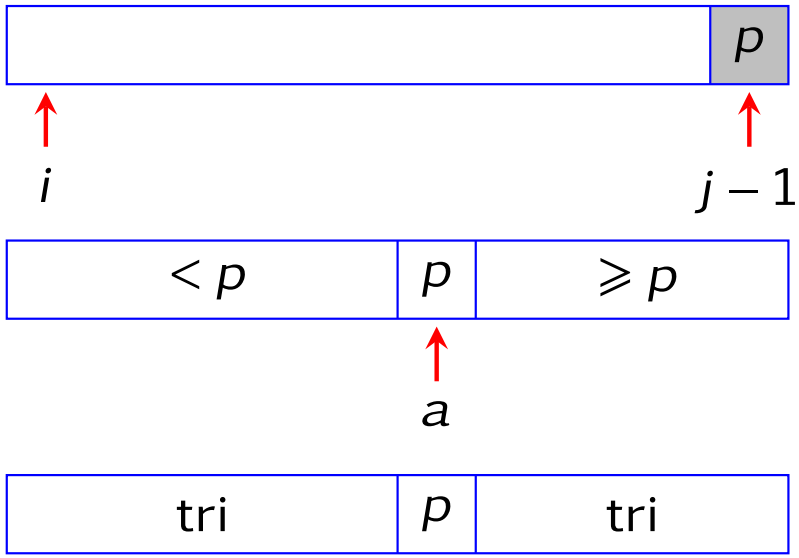


Tri rapide

```
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
```



Tri rapide




```
def quick_sort(t, *args):
    if len(args) == 0:
        i, j = 0, len(t)
    else:
        i, j = args
    if i + 1 < j:
        a = segmente(t, i, j)
        quick_sort(t, i, a)
        quick_sort(t, a + 1, j)
```

Étude de la complexité

Si $C(n)$ désigne le coût (en nombre de comparaisons et de permutations), alors : $C(n) = C(n_1) + C(n_2) + \Theta(n)$ avec $n_1 + n_2 = n - 1$.

Étude de la complexité

Si $C(n)$ désigne le coût (en nombre de comparaisons et de permutations), alors : $C(n) = C(n_1) + C(n_2) + \Theta(n)$ avec $n_1 + n_2 = n - 1$.

- Si $n_1 = 0$ et $n_2 = n - 1$, $C(n) = c(n-1) + \Theta(n)$ et $C(n) = \Theta(n^2)$.

Étude de la complexité

Si $C(n)$ désigne le coût (en nombre de comparaisons et de permutations), alors : $C(n) = C(n_1) + C(n_2) + \Theta(n)$ avec $n_1 + n_2 = n - 1$.

- Si $n_1 = 0$ et $n_2 = n-1$, $C(n) = c(n-1) + \Theta(n)$ et $C(n) = \Theta(n^2)$.
- Si $n_1 = n_2$, $C(n) = 2C(\lfloor n/2 \rfloor) + \Theta(n)$.
  Lorsque $n = 2^p - 1$ on pose $u_p = C(2^p - 1)$; alors $u_p = 2u_{p-1} + \Theta(2^p)$ donc $u_p = \Theta(p2^p)$ et $C(n) = \Theta(n \log n)$.

Étude de la complexité

Si $C(n)$ désigne le coût (en nombre de comparaisons et de permutations), alors : $C(n) = C(n_1) + C(n_2) + \Theta(n)$ avec $n_1 + n_2 = n - 1$.

- Si $n_1 = 0$ et $n_2 = n-1$, $C(n) = c(n-1) + \Theta(n)$ et $C(n) = \Theta(n^2)$.
- Si $n_1 = n_2$, $C(n) = 2C(\lfloor n/2 \rfloor) + \Theta(n)$.
  Lorsque $n = 2^p - 1$ on pose $u_p = C(2^p - 1)$; alors $u_p = 2u_{p-1} + \Theta(2^p)$ donc $u_p = \Theta(p2^p)$ et $C(n) = \Theta(n \log n)$.

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

Étude de la complexité

Si $C(n)$ désigne le coût (en nombre de comparaisons et de permutations), alors : $C(n) = C(n_1) + C(n_2) + \Theta(n)$ avec $n_1 + n_2 = n - 1$.

- Si $n_1 = 0$ et $n_2 = n-1$, $C(n) = c(n-1) + \Theta(n)$ et $C(n) = \Theta(n^2)$.
- Si $n_1 = n_2$, $C(n) = 2C(\lfloor n/2 \rfloor) + \Theta(n)$.
  Lorsque $n = 2^p - 1$ on pose $u_p = C(2^p - 1)$; alors $u_p = 2u_{p-1} + \Theta(2^p)$ donc $u_p = \Theta(p2^p)$ et $C(n) = \Theta(n \log n)$.

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

On suppose l'existence d'une constante M telle que $C(k) \leqslant Mk^2$ pour tout $k < n$.

$$\begin{aligned} C(n) &\leqslant M(n_1^2 + n_2^2) + \Theta(n) = M(n_1^2 + (n-1-n_1)^2) + \Theta(n) \\ &\leqslant M(n-1)^2 + \Theta(n) = Mn^2 - M(2n-1) + \Theta(n) \end{aligned}$$

Il suffit donc d'avoir choisi une constante M suffisamment grande pour que le terme $M(2n-1)$ domine le terme en $\Theta(n)$ pour obtenir $C(n) \leqslant Mn^2$.

Étude de la complexité

Imaginons que le partitionnement produise systématiquement un découpage dans une proportion de 9 pour 1 :

$$C(n) = C(n/10) + C(9n/10) + \Theta(n).$$

Étude de la complexité

Imaginons que le partitionnement produise systématiquement un découpage dans une proportion de 9 pour 1 :

$$C(n) = C(n/10) + C(9n/10) + \Theta(n).$$

Supposons l'existence de M telle que $C(k) \leqslant Mk \log k$ pour $k < n$. Alors

$$\begin{aligned} C(n) &\leqslant M\frac{n}{10}(\log n - \log 10) + M\frac{9n}{10}(\log n + \log 9 - \log 10) + \Theta(n) \\ &\leqslant Mn\log n - \frac{Mn}{10}(10\log 10 - 9\log 9) + \Theta(n) \end{aligned}$$

Il suffit de choisir M assez grand pour en déduire $C(n) \leqslant Mn\log n$.

Étude de la complexité

Imaginons que le partitionnement produise systématiquement un découpage dans une proportion de 9 pour 1 :

$$C(n) = C(n/10) + C(9n/10) + \Theta(n).$$

Supposons l'existence de M telle que $C(k) \leqslant Mk \log k$ pour $k < n$. Alors

$$\begin{aligned} C(n) &\leqslant M\frac{n}{10}(\log n - \log 10) + M\frac{9n}{10}(\log n + \log 9 - \log 10) + \Theta(n) \\ &\leqslant Mn\log n - \frac{Mn}{10}(10\log 10 - 9\log 9) + \Theta(n) \end{aligned}$$

Il suffit de choisir M assez grand pour en déduire $C(n) \leqslant Mn\log n$.

Raisonnement identique pour une proportion de 99 pour 100, etc.

Étude de la complexité

En moyenne, $C(n) = \Theta(n \log n)$.

Étude de la complexité

En moyenne, $C(n) = \Theta(n \log n)$.

On suppose qu'à l'issue du processus de segmentation le pivot a la même probabilité de se trouver dans chacune des cases du tableau :

$$c_n = n - 1 + \frac{1}{n} \sum_{k=0}^{n-1} (c_k + c_{n-k-1}) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} c_k.$$

Étude de la complexité

En moyenne, $C(n) = \Theta(n \log n)$.

On suppose qu'à l'issue du processus de segmentation le pivot a la même probabilité de se trouver dans chacune des cases du tableau :

$$c_n = n - 1 + \frac{1}{n} \sum_{k=0}^{n-1} (c_k + c_{n-k-1}) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} c_k.$$

On a $nc_n = n(n-1) + 2 \sum_{k=0}^{n-1} c_k$ et $(n-1)c_{n-1} = (n-1)(n-2) + 2 \sum_{k=0}^{n-2} c_k$ donc :

$$nc_n - (n-1)c_{n-1} = 2(n-1) + 2c_{n-1} \iff \frac{c_n}{n+1} - \frac{c_{n-1}}{n} = \frac{4}{n+1} - \frac{2}{n}.$$

Étude de la complexité

En moyenne, $C(n) = \Theta(n \log n)$.

On suppose qu'à l'issue du processus de segmentation le pivot a la même probabilité de se trouver dans chacune des cases du tableau :

$$c_n = n - 1 + \frac{1}{n} \sum_{k=0}^{n-1} (c_k + c_{n-k-1}) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} c_k.$$

On a $nc_n = n(n-1) + 2 \sum_{k=0}^{n-1} c_k$ et $(n-1)c_{n-1} = (n-1)(n-2) + 2 \sum_{k=0}^{n-2} c_k$ donc :

$$nc_n - (n-1)c_{n-1} = 2(n-1) + 2c_{n-1} \iff \frac{c_n}{n+1} - \frac{c_{n-1}}{n} = \frac{4}{n+1} - \frac{2}{n}.$$

$$\frac{c_n}{n+1} - c_0 = 4 \sum_{k=1}^{n} \frac{1}{k+1} - 2 \sum_{k=1}^{n} \frac{1}{k} = 2 \sum_{k=1}^{n} \frac{1}{k} + \frac{4}{n+1} - 4 = 2h_n - \frac{4n}{n+1}.$$

Étude de la complexité

En moyenne, $C(n) = \Theta(n \log n)$.

On suppose qu'à l'issue du processus de segmentation le pivot a la même probabilité de se trouver dans chacune des cases du tableau :

$$c_n = n - 1 + \frac{1}{n} \sum_{k=0}^{n-1} (c_k + c_{n-k-1}) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} c_k.$$

On a $nc_n = n(n-1) + 2 \sum_{k=0}^{n-1} c_k$ et $(n-1)c_{n-1} = (n-1)(n-2) + 2 \sum_{k=0}^{n-2} c_k$ donc :

$$nc_n - (n-1)c_{n-1} = 2(n-1) + 2c_{n-1} \iff \frac{c_n}{n+1} - \frac{c_{n-1}}{n} = \frac{4}{n+1} - \frac{2}{n}.$$

$$\frac{c_n}{n+1} - c_0 = 4 \sum_{k=1}^{n} \frac{1}{k+1} - 2 \sum_{k=1}^{n} \frac{1}{k} = 2 \sum_{k=1}^{n} \frac{1}{k} + \frac{4}{n+1} - 4 = 2h_n - \frac{4n}{n+1}.$$

$h_n \sim \ln n$ donc $c_n \sim (2 \ln 2) n \log n$ avec $2 \ln 2 \approx 1,4$.

Tri par dénombrement

On suppose que les éléments à trier ne peuvent prendre qu'un nombre fini de valeurs dans $[\![0, k-1]\!]$.

```
def counting_sort(t, k):
    occ = [0] * k
    for i in t:
        occ[i] += 1
    s = 0
    for i in range(k):
        for j in range(occ[i]):
            t[s] = i
            s += 1
```

Tri par dénombrement

On suppose que les éléments à trier ne peuvent prendre qu'un nombre fini de valeurs dans $[\![0, k-1]\!]$.

```
def counting_sort(t, k):
    occ = [0] * k
    for i in t:
        occ[i] += 1
    s = 0
    for i in range(k):
        for j in range(occ[i]):
            t[s] = i
            s += 1
```

À l'issue de la première boucle le tableau *occ* contient dans sa case d'indice i le nombre d'occurrences de $i \in [\![0, k-1]\!]$ dans t.
$\longrightarrow$ coût temporel en $\Theta(n)$.

Tri par dénombrement

On suppose que les éléments à trier ne peuvent prendre qu'un nombre fini de valeurs dans $[\![0, k-1]\!]$.

```
def counting_sort(t, k):
    occ = [0] * k
    for i in t:
        occ[i] += 1
    s = 0
    for i in range(k):
        for j in range(occ[i]):
            t[s] = i
            s += 1
```

À l'issue de la première boucle le tableau *occ* contient dans sa case d'indice i le nombre d'occurrences de $i \in [\![0, k-1]\!]$ dans t.

$\longrightarrow$ coût temporel en $\Theta(n)$.

La seconde boucle écrit pour chacune des valeurs de i autant de fois i dans le tableau t que nécessaire.

$\longrightarrow$ coût temporel en $\Theta(k+n)$.

Tri par dénombrement

On suppose que les éléments à trier ne peuvent prendre qu'un nombre fini de valeurs dans $[\![0, k-1]\!]$.

```
def counting_sort(t, k):
    occ = [0] * k
    for i in t:
        occ[i] += 1
    s = 0
    for i in range(k):
        for j in range(occ[i]):
            t[s] = i
            s += 1
```

À l'issue de la première boucle le tableau *occ* contient dans sa case d'indice i le nombre d'occurrences de $i \in [\![0, k-1]\!]$ dans t.

$\longrightarrow$ coût temporel en $\Theta(n)$.

La seconde boucle écrit pour chacune des valeurs de i autant de fois i dans le tableau t que nécessaire.

$\longrightarrow$ coût temporel en $\Theta(k+n)$.

Total : coût temporel en $\Theta(n+k)$ et coût spatial en $\Theta(k)$.

Lorsque $k = O(n)$ le coût est linéaire.

Tri par dénombrement

Cas d'un pré-ordre

On suppose donnée une fonction $c : E \to [\![0, k-1]\!]$ qui servira de clé du tri : $\forall (x,y) \in E^2, x \leqslant y \iff c(x) \leqslant c(y)$. Ainsi défini, $\leqslant$ est un pré-ordre.

Tri par dénombrement

Cas d'un pré-ordre

On suppose donnée une fonction $c : E \to [\![0, k-1]\!]$ qui servira de clé du tri : $\forall (x,y) \in E^2, x \leqslant y \iff c(x) \leqslant c(y)$. Ainsi défini, $\leqslant$ est un pré-ordre.

```
def counting_sort(t, c, k):
    occ = [0] * k
    s = [None] * len(t)
    for x in t:
        occ[c(x)] += 1
    for i in range(1, k):
        occ[i] += occ[i-1]
    for x in reversed(t):
        occ[c(x)] -= 1
        s[occ[c(x)]] = x
    t[:] = s
```

La première boucle calcule le nombre d'occurrence de chacune des clés.
La seconde boucle calcule l'emplacement final du dernier élément de chacune des clés.
La dernière boucle range les données ordonnées par la fonction c.

Tri par dénombrement

Cas d'un pré-ordre

On suppose donnée une fonction $c : E \to [\![0, k-1]\!]$ qui servira de clé du tri : $\forall (x,y) \in E^2, x \leqslant y \iff c(x) \leqslant c(y)$. Ainsi défini, $\leqslant$ est un pré-ordre.

```
def counting_sort(t, c, k):
    occ = [0] * k
    s = [None] * len(t)
    for x in t:
        occ[c(x)] += 1
    for i in range(1, k):
        occ[i] += occ[i-1]
    for x in reversed(t):
        occ[c(x)] -= 1
        s[occ[c(x)]] = x
    t[:] = s
```

La première boucle calcule le nombre d'occurrence de chacune des clés.
La seconde boucle calcule l'emplacement final du dernier élément de chacune des clés.
La dernière boucle range les données ordonnées par la fonction c.
On obtient un algorithme de tri stable dont la complexité tant temporelle que spatiale est un $\Theta(n+k)$.

Tri par dénombrement

Exemple du tri d'un tableau d'entiers suivant le dernier chiffre

123	451	678	942	102	134	156	741	684	235	910	236

Tri par dénombrement

Exemple du tri d'un tableau d'entiers suivant le dernier chiffre

123	451	678	942	102	134	156	741	684	235	910	236

On calcule les occurrences de chaque dernier chiffre :

0	1	2	3	4	5	6	7	8	9
1	2	2	1	2	1	2	0	1	0

Tri par dénombrement

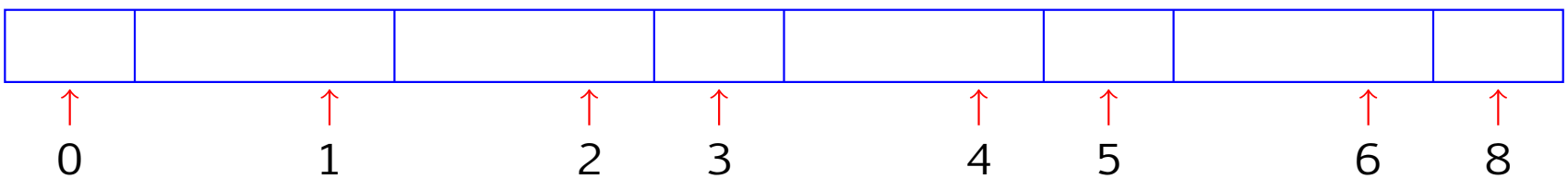
Exemple du tri d'un tableau d'entiers suivant le dernier chiffre

123	451	678	942	102	134	156	741	684	235	910	236

On calcule les occurrences de chaque dernier chiffre :

0	1	2	3	4	5	6	7	8	9
1	2	2	1	2	1	2	0	1	0

On détermine les zones attribuées à chacun des derniers chiffres :



Tri par dénombrement

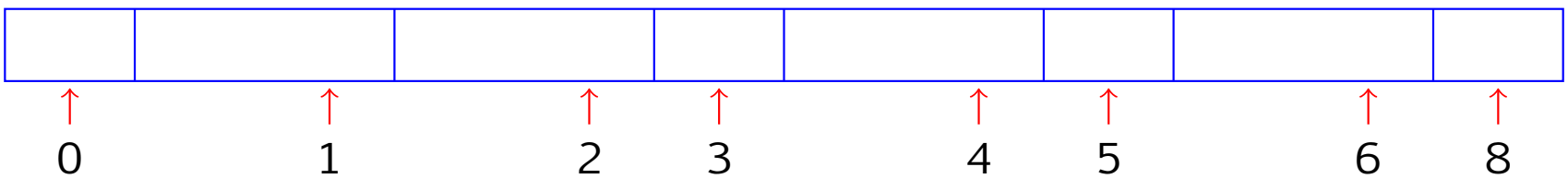
Exemple du tri d'un tableau d'entiers suivant le dernier chiffre

123	451	678	942	102	134	156	741	684	235	910	236

On calcule les occurrences de chaque dernier chiffre :

0	1	2	3	4	5	6	7	8	9
1	2	2	1	2	1	2	0	1	0

On détermine les zones attribuées à chacun des derniers chiffres :



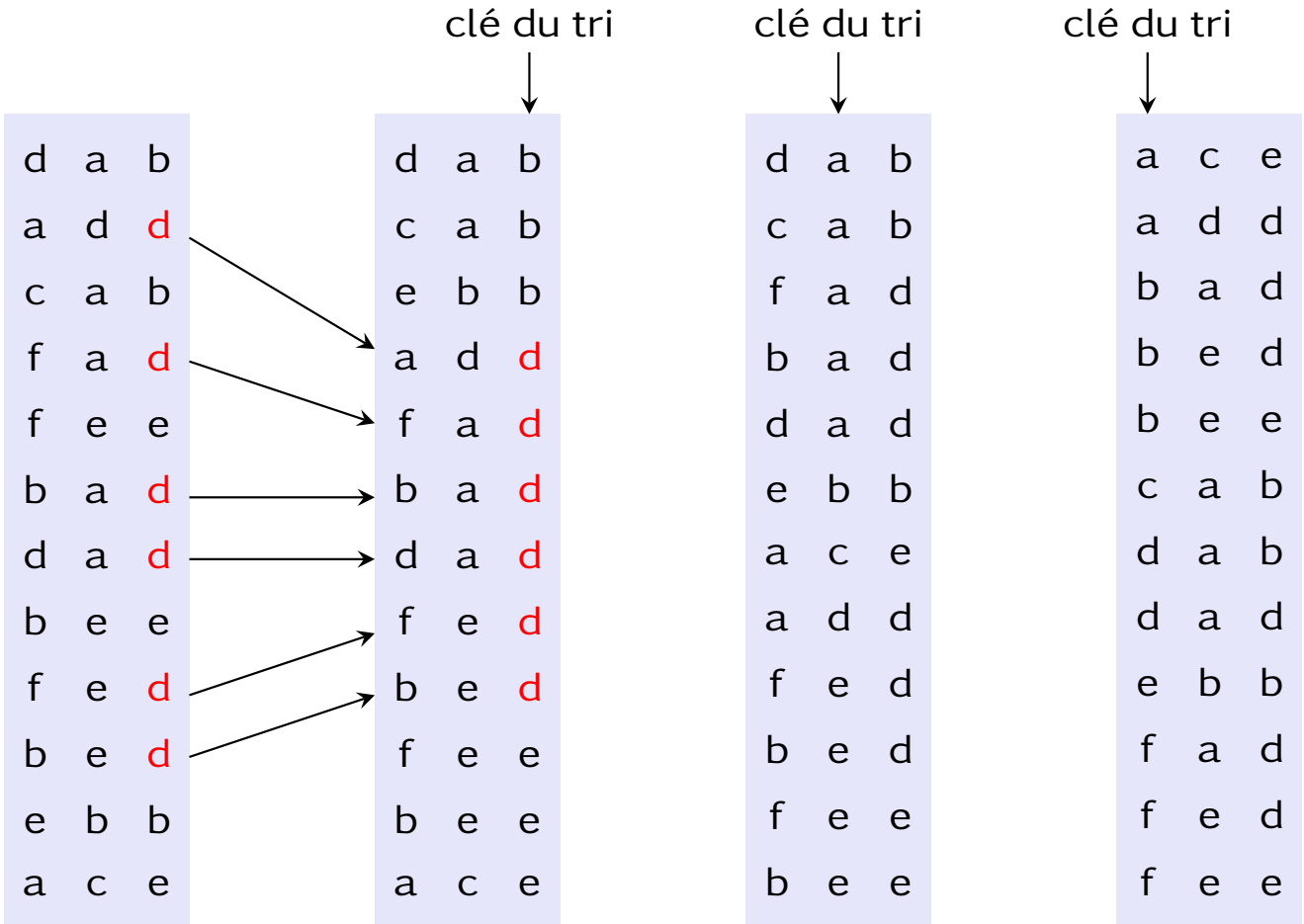
On remplit chacune de ces zones de droite à gauche :

910	451	741	942	102	123	134	684	235	156	236	678

Tri par base

LSD radix sort

On trie les éléments à partir de leur chiffre le moins significatif. Cette démarche suppose que tous ces éléments sont de même taille et que les tris partiels soient stables.



Tri par base

LSD radix sort

On trie les éléments à partir de leur chiffre le moins significatif. Cette démarche suppose que tous ces éléments sont de même taille et que les tris partiels soient stables.

Exemple.

```
In [1]: t = [123, 451, 678, 942, 102, 134, 156, 741, 684, 235, 910, 236]

In [2]: counting_sort(t, lambda x: x % 10, 10)

In [3]: t
Out[3]: [910, 451, 741, 942, 102, 123, 134, 684, 235, 156, 236, 678]

In [4]: counting_sort(t, lambda x: (x // 10) % 10, 10)

In [5]: t
Out[5]: [102, 910, 123, 134, 235, 236, 741, 942, 451, 156, 678, 684]

In [6]: counting_sort(t, lambda x: (x // 100) % 10, 10)

In [7]: t
Out[7]: [102, 123, 134, 156, 235, 236, 451, 678, 684, 741, 910, 942]
```

Tri par base

LSD radix sort

On trie les éléments à partir de leur chiffre le moins significatif. Cette démarche suppose que tous ces éléments sont de même taille et que les tris partiels soient stables.

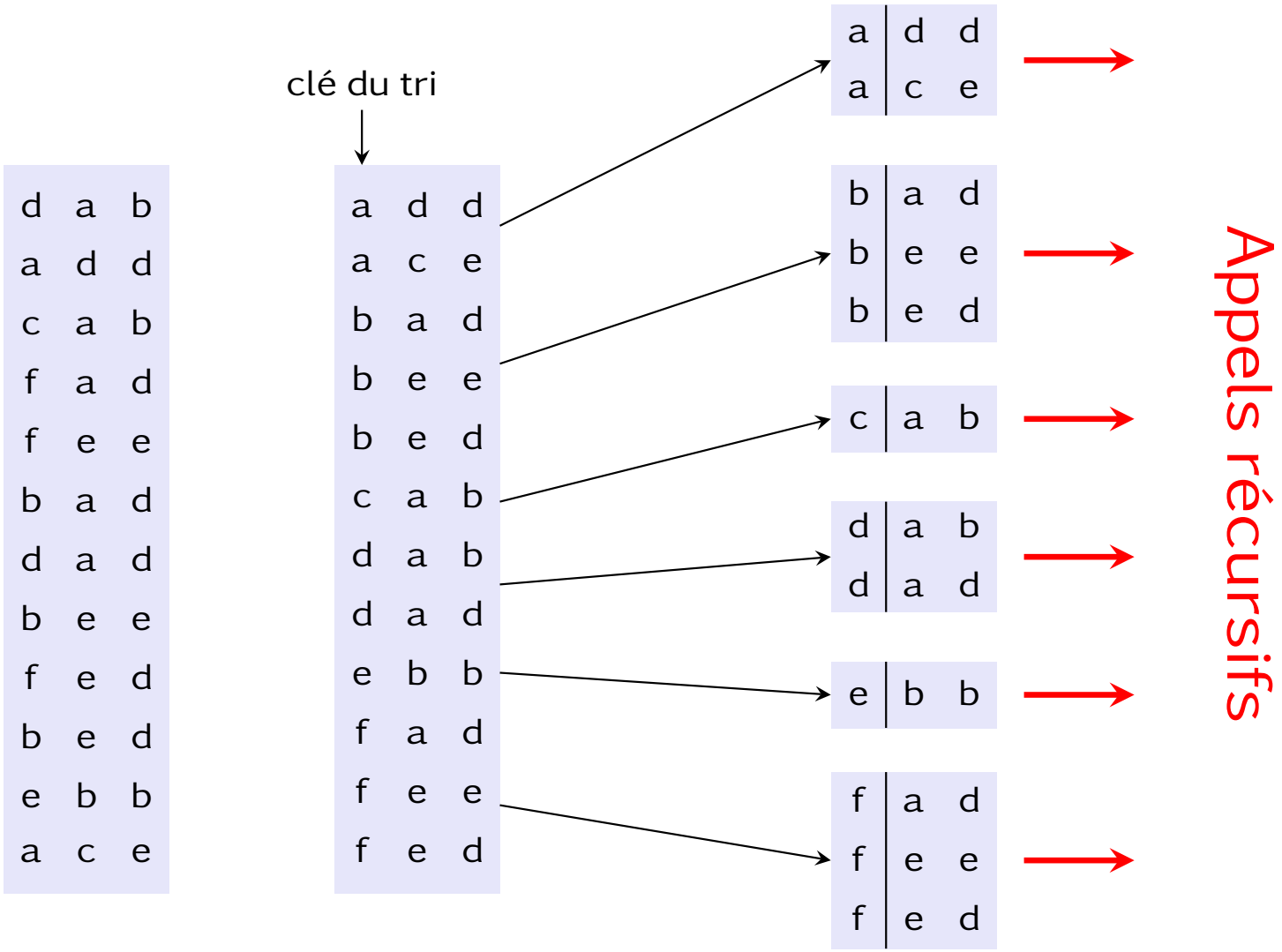
Complexité.
Si on procède comptage pour un coût en $\Theta(n+k)$ (où k désigne la base choisie) le coût total de LSD radix sort est un $\Theta(d(n+k))$, où d est le nombre de caractères de chaque mot.

Cet algorithme est plus intéressant qu'un algorithme de tri par comparaison en $\Theta(n \log n)$ lorsque $d = \mathrm{O}(\log n)$.

Tri par base

MSD radix sort

On trie cette fois à partir du chiffre le plus significatif.



Tri par base

MSD radix sort

On trie cette fois à partir du chiffre le plus significatif.

Inconvénients :

- multiples appels récursifs (donc inadapté en PYTHON) ;

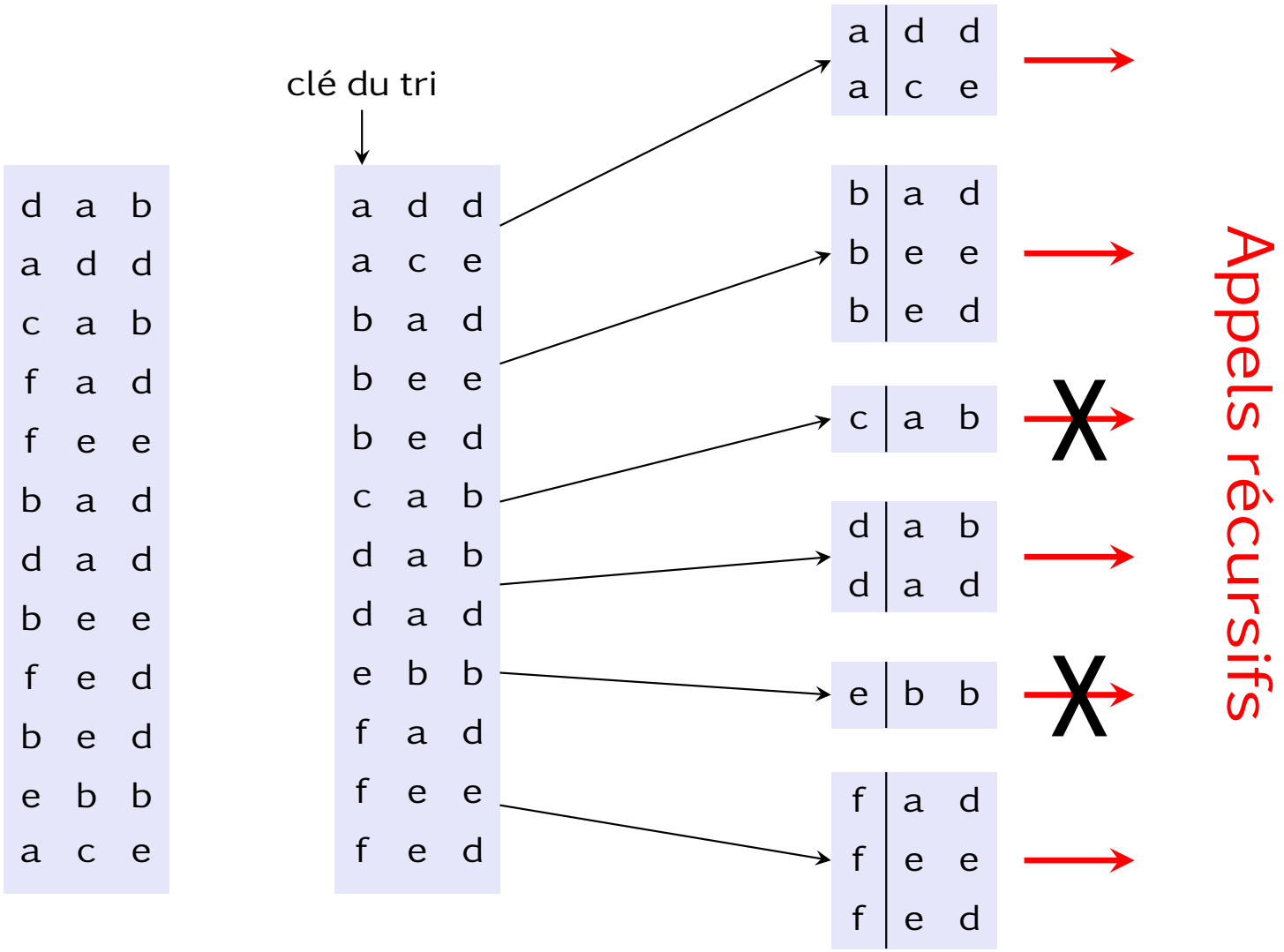
Avantages :

- adaptée au tri lexicographique des chaînes de caractères ;
- il n'est pas toujours nécessaire d'examiner toutes les clés.

Tri par base

MSD radix sort

On trie cette fois à partir du chiffre le plus significatif.



Tri par base

MSD radix sort

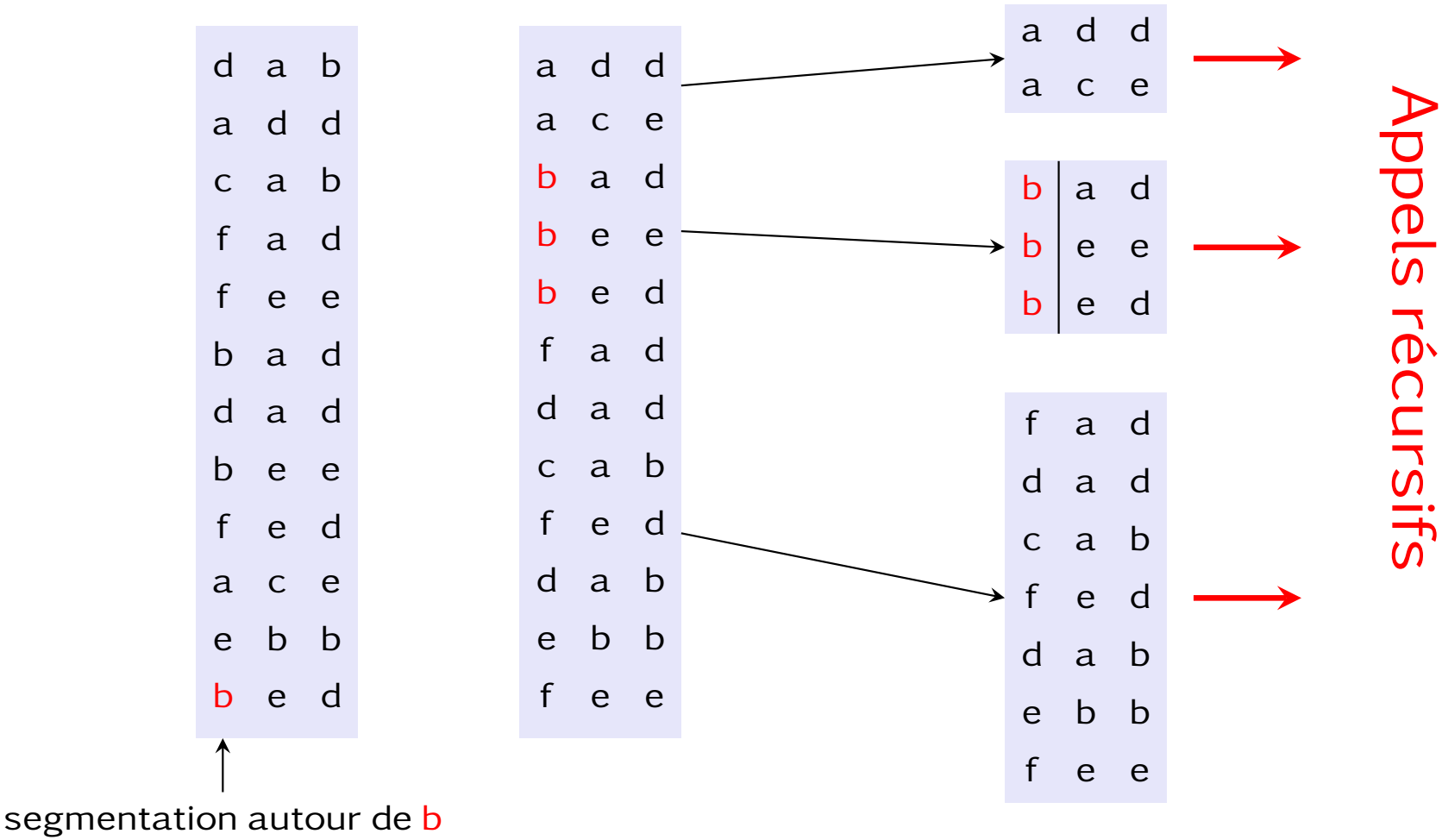
On trie cette fois à partir du chiffre le plus significatif.

a	c	e
a	d	d
b	a	d
b	e	d
b	e	e
c	a	b
d	a	b
d	a	d
e	b	b
f	a	d
f	e	d
f	e	e

Seuls 78% des caractères ont étés examinés (cette proportion peut se révéler beaucoup plus basse pour de plus grandes chaînes de caractères).

MSD radix sort et segmentation

On peut combiner les avantages du tri rapide et du tri par base pour trier efficacement des chaînes de caractères en procédant à une segmentation du tableau.



MSD radix sort et segmentation

On peut combiner les avantages du tri rapide et du tri par base pour trier efficacement des chaînes de caractères en procédant à une segmentation du tableau. Le plus efficace pour trier des chaînes de caractères !