ALGORITHMES DE TRI

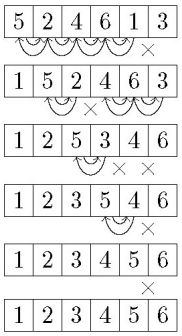
Nicolas CHIREUX

Nous nous proposons dans ce TD d'étudier divers algorithme de tri en partant des plus basiques pour aller vers des tris plus sophistiqués. Les bibliothèques à utiliser seront : math, random et time.

On définira une fonction $tab\_alea(n, mini = 0, maxi = 10 * *6)$ pour générer un tableau de n entiers aléatoires compris par défaut entre 0 et 10^6. Cette fonction utilisera $random.randint(mini, maxi)$.

1 Le tri bulle

1.1 Principe



Le tri bulle est une stratégie de tri locale, qui consiste à trier une liste d'objet en regardant les éléments de la liste deux par deux et en les mettant dans l'ordre. On part du début de la liste et on regarde les éléments consécutivement. A la fin d'un premier passage sur la liste, l'élément le plus grand se retrouve en fin de liste et les élément plus petits remontent vers le début de la liste. C'est de là que provient le nom de l'algorithme qui imite la remontée de bulles.
Si la liste est de taille n, après n passages, les n bulles seront remontées et classées dans l'ordre.

Rem : sur l'exemple la méthode est appliquée en partant de la fin de la liste. C'est une possibilité tout aussi correcte.

FIGURE 1 – Tri bulle

```
Fonction tri_bulle(t) :
    n ← longueur(t)
    Pour i de 1 à n − 1 faire
        Pour j de 1 à n − 1 faire
            Si (t_{j+1} < t_j) Alors
                échanger(t_{j+1},t_j)
            Fin Si
        Fin Pour
    Fin Pour
    Retourner t
Fin
```

Algorithme 1: Tri bulle

Une version plus subtile consiste à cesser les parcours de la liste dès qu'il n'y a plus de permutation lors d'un passage.

```
Fonction tri_bulle(t) :
    perm ← True
    n ← longueur(t)
    Tant que (perm) faire
        perm ← False
        Pour j de 1 à n − 1 faire
            Si (t_j > t_{j+1}) Alors
                échanger(t_j,t_{j+1})
                perm ← True
            Fin Si
        Fin Pour
    Fait
    Retourner t
Fin
```

1.2 Complexité

Lors de chaque boucle interne, une comparaison est faite. La boucle interne comprend, pour chaque i, $n-i-2$ itérations avec i variant entre 0 et $n-2$. Chaque itération i de la boucle externe contient donc $n-i-2$ comparaisons, et le nombre total de comparaisons s'écrit

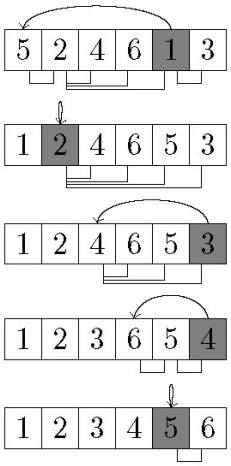
$$\sum_{i=0}^{n-2} n-2-i = \sum_{i=0}^{n} i = \frac{n(n-1)}{2} \tag{1}$$

Dans le pire des cas, il y a permutation à chaque comparaison soit 3 opérations par échange soit au total $3.\frac{n(n-1)}{2}$ opérations. Ceci donne une complexité en $\Theta(n^2)$, c'est à dire quadratique.

Concernant la complexité en mémoire, le tri bulle est un tri en place, qui ne demande pas la création d'un tableau supplémentaire. Sa complexité en mémoire est donc en $\Theta(1)$.

2 Tri par sélection

2.1 Principe



Le principe consiste à chercher le plus petit élément d'une liste à trier et à le mettre en première position, l'élément qui était en première position allant à la place libérée par le plus petit. Puis dans la liste restant à trier, nous renouvelons l'opération et ainsi de suite.

- 1 est le minimum, il est échangé avec 5 en place 1
- à partir de la position 2, 2 est le minimum, déjà en place
- à partir de la position 3, 3 est le minimum, échangé avec 4 en place 3
- à partir de la position 4, 4 est le minimum, échangé avec 6 en place 4
- à partir de la position 5, 5 est le minimum, déjà en place
- 6 est en place

;

FIGURE 2 – Tri par sélection

On se propose de définir tout d'abord une fonction *indice_mini*(m,i,j) qui renvoie le plus petit indice $k \in [i,j[$ de la liste m tel que m_k soit le minimum des éléments de la liste allant de t_i à t_{j-1}

```
Fonction indice_mini(m,i,j) :
    i_m ← i
    Pour k de i+1 à j−1 faire
        Si (m_k < m_{i_m}) Alors
            i_m ← k
        Fin Si
    Fin Pour
    Retourner i_m
Fin
```

Algorithme 2: Fonction *indice_mini*

Définir ensuite une fonction *tri_selection*(t) utilisant la fonction précédente.

```
Fonction tri_selection(t) :
    n ← longueur(t)
    Pour i de 0 à n−1 faire
        i_m ← indice_mini(t,i,n)
        échanger(t_i,t_{i_m})
    Fin Pour
    Retourner t
Fin
```

Algorithme 3: Fonction *tri_selection*

Pour ceux qui préfèrent faire le tri par sélection en une seule fonction, le pseudo-code est le suivant :

```
Fonction tri_selection(t) :
    t : liste non triée
    n ← longueur(t)
    Pour i de 0 à n − 1 faire
        i_m ← i
        Pour k de i + 1 à n − 1 faire
            Si (t_k < t_{i_m}) Alors
                i_m ← k
            Fin Si
        Fin Pour
        échanger(t_i, t_{i_m})
    Fin Pour
    Retourner t
Fin
```

Algorithme 4: Fonction *tri_selection*

2.2 Complexité

Il y a une comparaison par itération de la boucle interne. Le nombre de comparaisons est ainsi de

$$\sum_{i=0}^{n-2} \sum_{k=i+1}^{n-1} 1 = \frac{n(n+1)}{2} \tag{2}$$

Il est constant, que ce soit le pire, le meilleur, ou le cas moyen. Par contre le nombre d'échanges peut varier de $n-1$ dans le pire des cas - liste triée en sens inverse au départ - à 0 dans le meilleur des cas -liste déjà triée-. Le cas moyen semble difficile à conjecturer.

Ceci donne une complexité dans le pire des cas en $\Theta(n^2)$ comme dans le cas du tri bulle. Par contre, si nous avons à traiter une liste quasi triée, la complexité va d'autant plus se rapprocher de $\Theta(n)$ que nous serons proche du meilleur des cas.

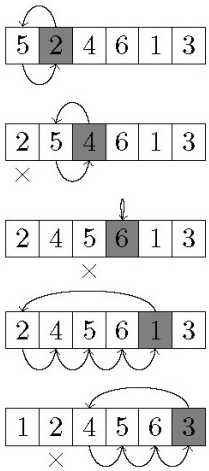
Concernant la complexité en mémoire, le tri bulle est un tri en place, qui ne demande pas la création d'un tableau supplémentaire. Sa complexité en mémoire est donc en $\Theta(1)$.

3 Tri par insertion

3.1 Principe

Le tri insertion est un algorithme efficace de tri d'un petit nombre d'éléments. Son principe de fonctionnement est celui qui est souvent utilisé pour trier un jeu de cartes. On commence avec la main gauche vide et le paquet de carte retourné sur la table.

On retourne ensuite la première carte du paquet et on l'amène à la bonne position dans la main gauche. Pour trouver cette bonne position, on compare la carte piochée aux cartes de la main gauche une par une, de droite à gauche. A chaque instant, les cartes dans la main gauche sont triées.



— Le pivot est le deuxième élément du tableau. On le compare au premier élément : il est plus petit et donc on échange donc les deux éléments. Par conséquent, les deux premiers éléments du tableau sont maintenant triés.
— Le pivot sélectionné est le troisième élément. On le compare aux élément à sa gauche, du plus proche au plus éloigné. Il est plus petit que 5 donc on continue les comparaisons. Il est par contre plus grand que 2, dans on s'arrête. On décale les éléments qui sont plus grands que 4 d'un cran vers la droite et on insère 4 à la place laissée vide
— Le pivot est le quatrième élément : on recommence la manipulation précédente et ainsi de suite

FIGURE 3 – Tri par insertion

On se propose de définir tout d'abord une fonction $place(x, m)$ qui renvoie l'indice où insérer x dans m pour que m reste triée.

```
Fonction place(x, m) :
    x,m : élément dont on cherche la place, liste en cours de tri
    ind ← 0
    n ← longueur(m)
    Tant que (ind < n  ET x > m_ind) faire
        ind ← ind + 1
    Fait
    Retourner ind
Fin
```

Algorithme 5: Fonction *place*

Définir ensuite une fonction $tri\_insertion(t)$ utilisant la fonction précédente. On utilisera une nouvelle liste *triee* pour stocker les éléments au fur et à mesure - c'est l'analogue de la main gauche aux cartes -. Ce n'est donc pas une version du tri en place.

```
Fonction tri_insertion(t) :
    t : liste non triée
    n ← longueur(t)
    triee ← []
    Pour i de 0 à n − 1 faire
        k ← place(t_i, triee)
        insérer(t_i) à la place k
    Fin Pour
    Retourner triee
Fin
```

Algorithme 6: Fonction *tri_insertion*

Pour ceux qui préfèrent faire le tri par insertion en une seule fonction, le pseudo-code est le suivant :

```
Fonction tri_insertion(t) :
    t : liste non triée
    n ← longueur(t)
    triee ← [t[0]]
    Pour i de 1 à n − 1 faire
        k ← 0
        Tant que (k < len(triee)  ET t_i > triee_k) faire
            k ← k + 1
        Fait
        insérer(t_i) à la place k
    Fin Pour
    Retourner triee
Fin
```

Algorithme 7: Fonction *tri_insertion*

Pour les rapides, tentez une version du tri par insertion en place en utilisant des swaps.

3.2 Complexité

Dans le meilleur des cas -liste déjà triée -, on ne fait qu'une comparaison et aucune insertion. Nous nous contentons d'ajouter le nouvel élément à la fin de la liste *triee*. Nous avons une complexité en $\Theta(n)$.

Dans le pire des cas - liste triée en sens inverse -, on doit comparer avec toute la liste avant de trouver sa place soit un nombre de comparaisons :

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \tag{3}$$

Pour insérer un élément à sa place, il faut faire subir des échanges à tous ceux qui plus grands que lui soit à toute la liste ici puisque la liste est classée à l'envers - chaque nouvel élément est plus petit que tous ceux déjà triés ! - soit un nombre total d'échanges

$$\sum_{i=0}^{n-1}(i-1) = \frac{n(n-1)}{2} - n \tag{4}$$

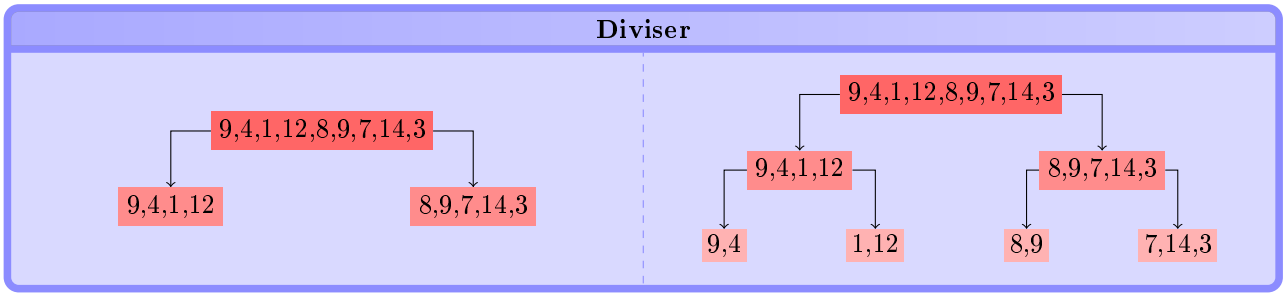
Dans le pire des cas la complexité est en $\Theta(n^2)$. Le cas moyen est difficile à prévoir.

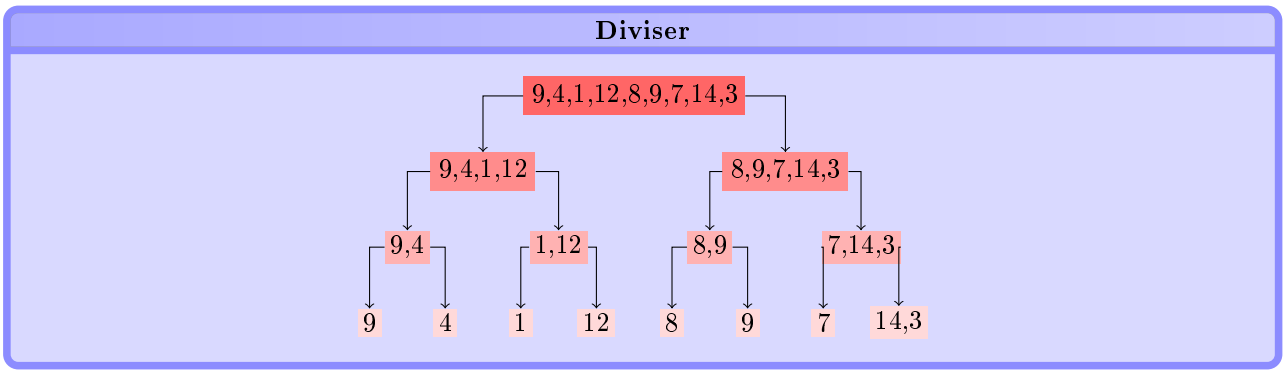
Tous les algorithmes que nous venons de voir sont en $\Theta(n^2)$. Nous allons étudier deux algorithmes de tri qui font mieux : le tri fusion -merge sort- et le tri rapide -quicksort-. Ces algorithmes fonctionnent récursivement.

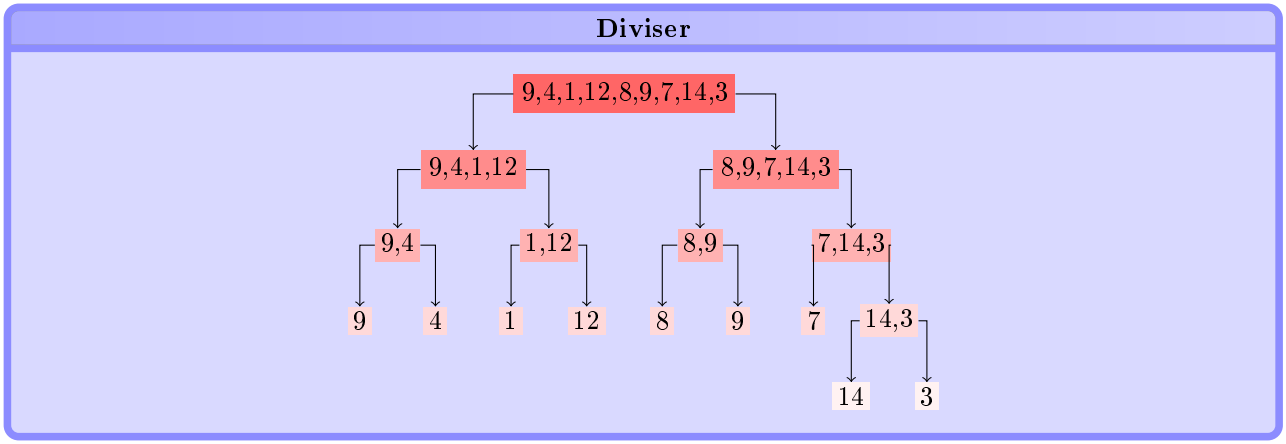
4 Le tri fusion ou Merge Sort

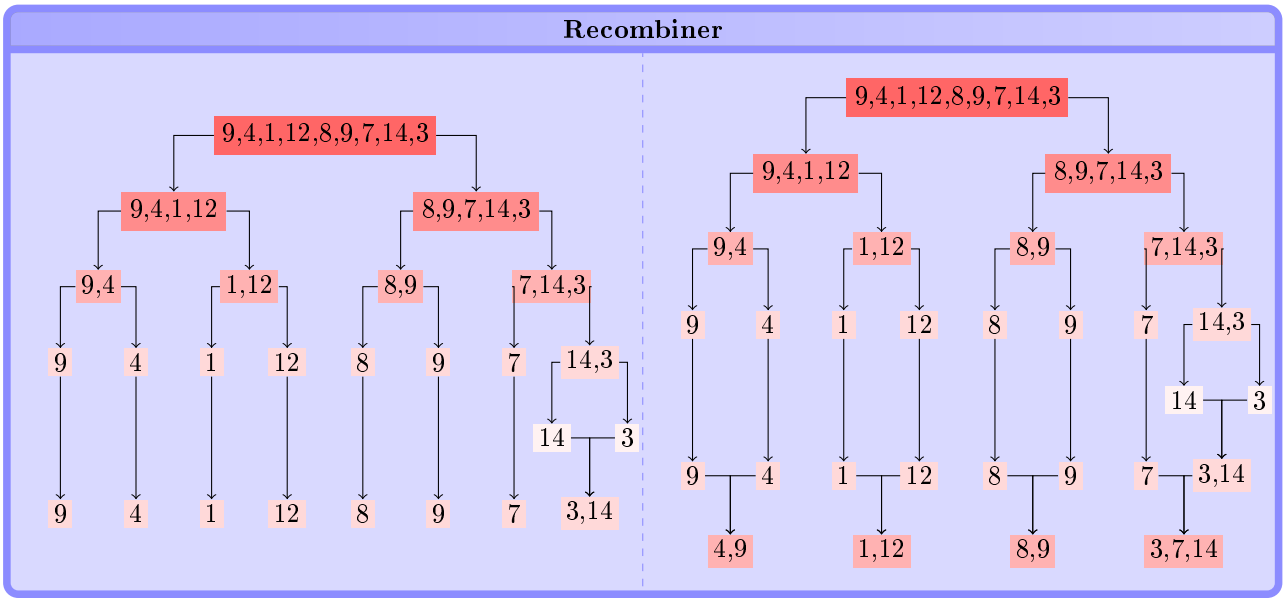
4.1 Principe

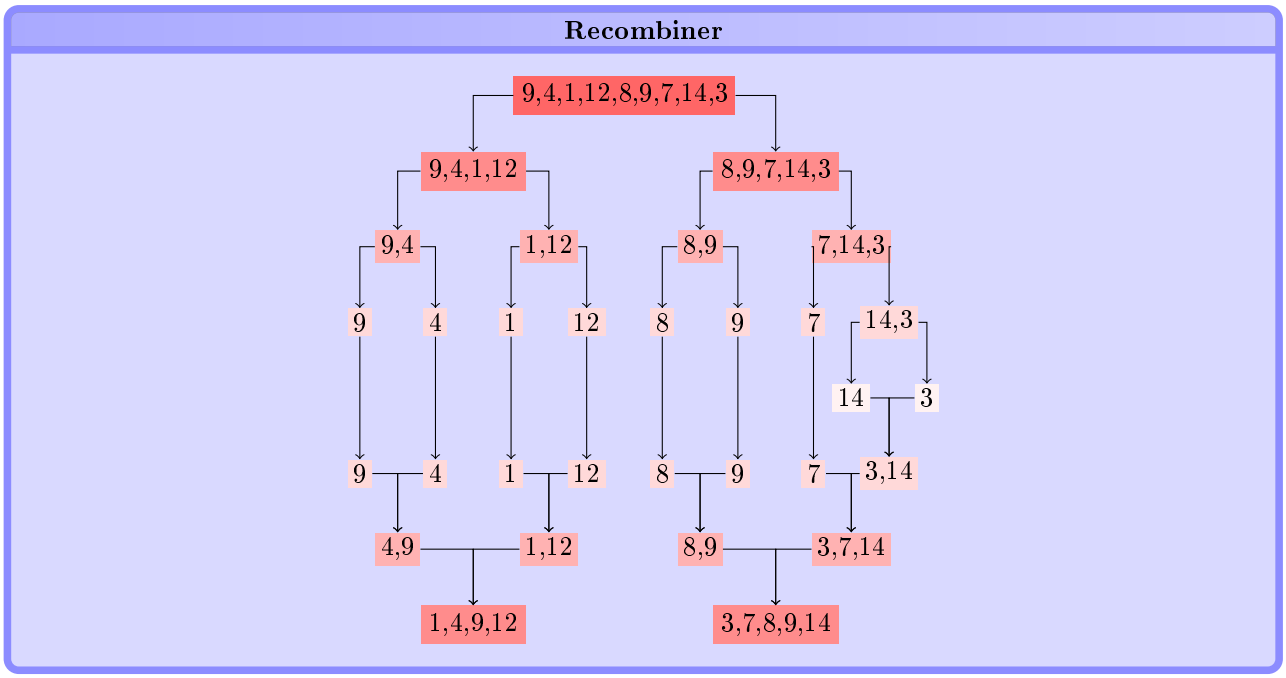
L'algorithme de tri fusion est un exemple classique de la technique algorithmique de "diviser pour régner". L'opération principale consiste à fusionner de deux sous-listes déjà triées. Pour effectuer cette fusion, on définit une fonction auxiliaire $fusionne(t_1, t_2)$ qui fusionne t_1 et t_2 pour obtenir un unique tableau trié. On utilisera une liste temporaire w. Ce n'est donc pas un tri en place. On utilisera un indice sur chaque sous-liste, puis on comparera les éléments de chaque sous-liste correspondant à ces indices et on choisira de placer dans w le plus petit des deux. Chaque fois qu'on insère dans w un élément d'une sous-liste, on incrémente l'indice associé à la sous-liste. (voir l'exemple en fin de paragraphe)

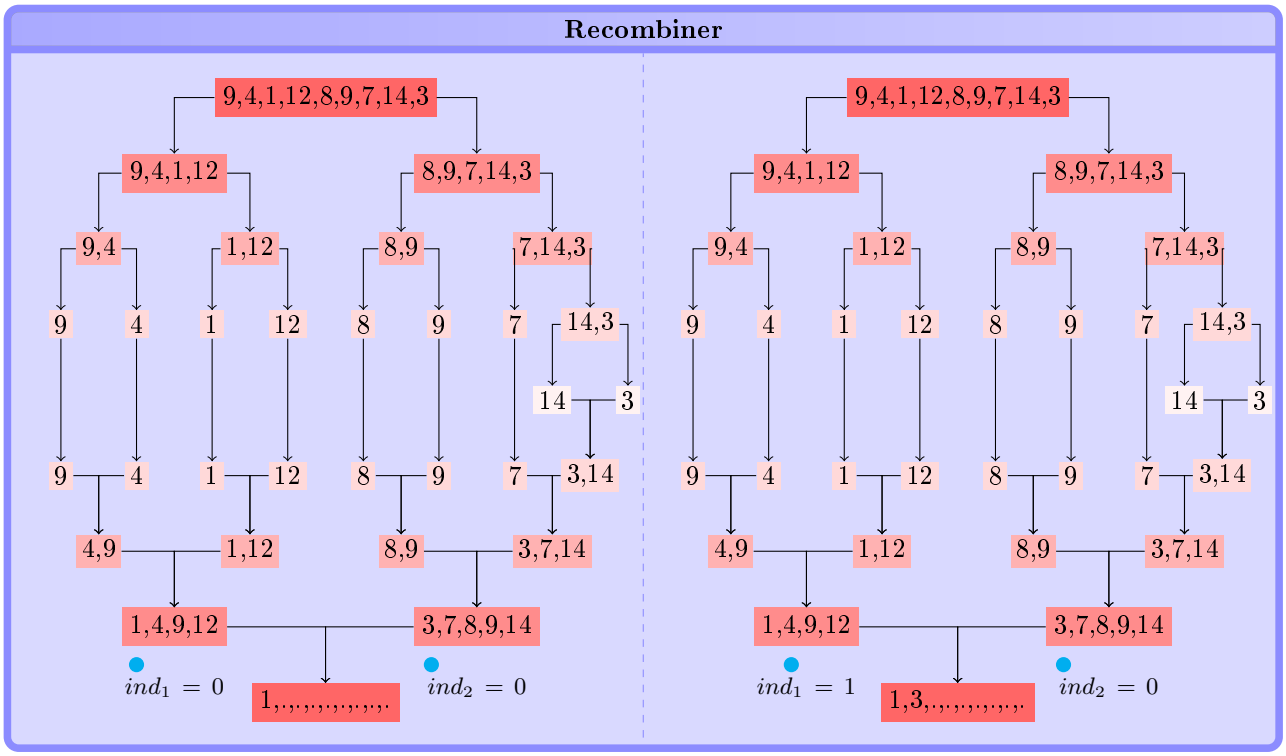


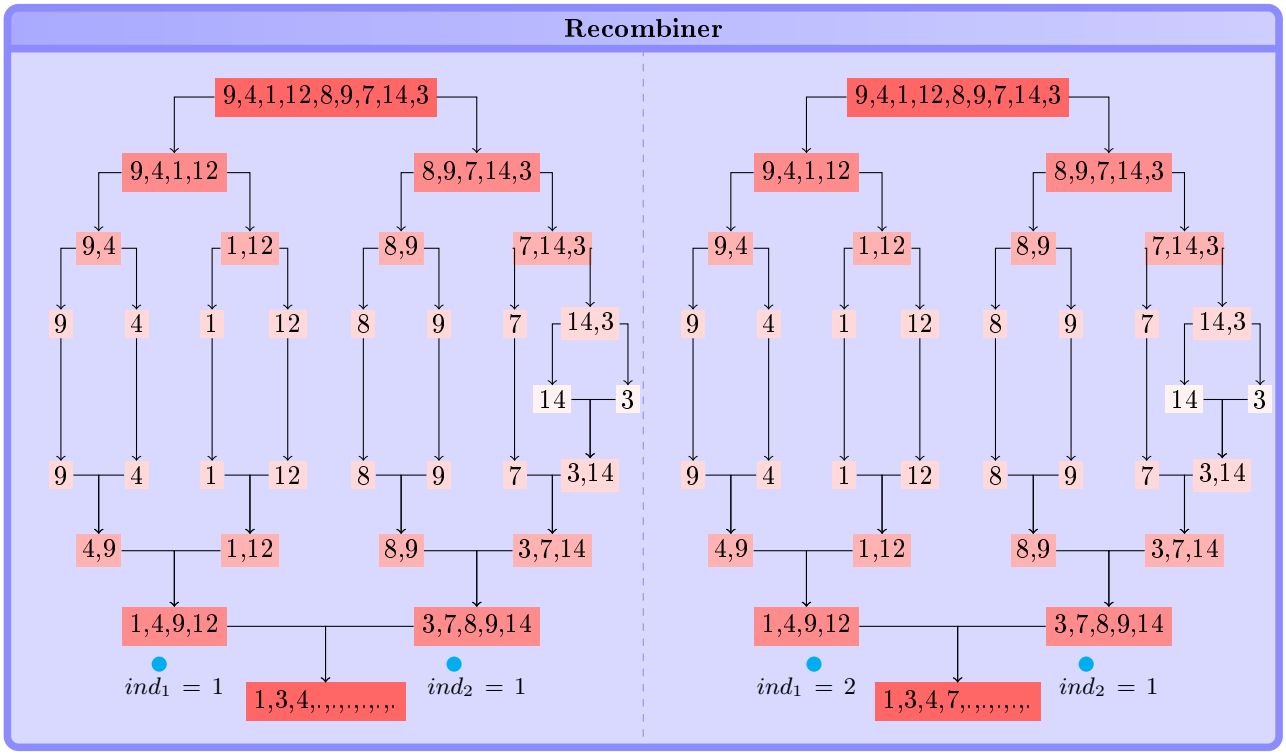


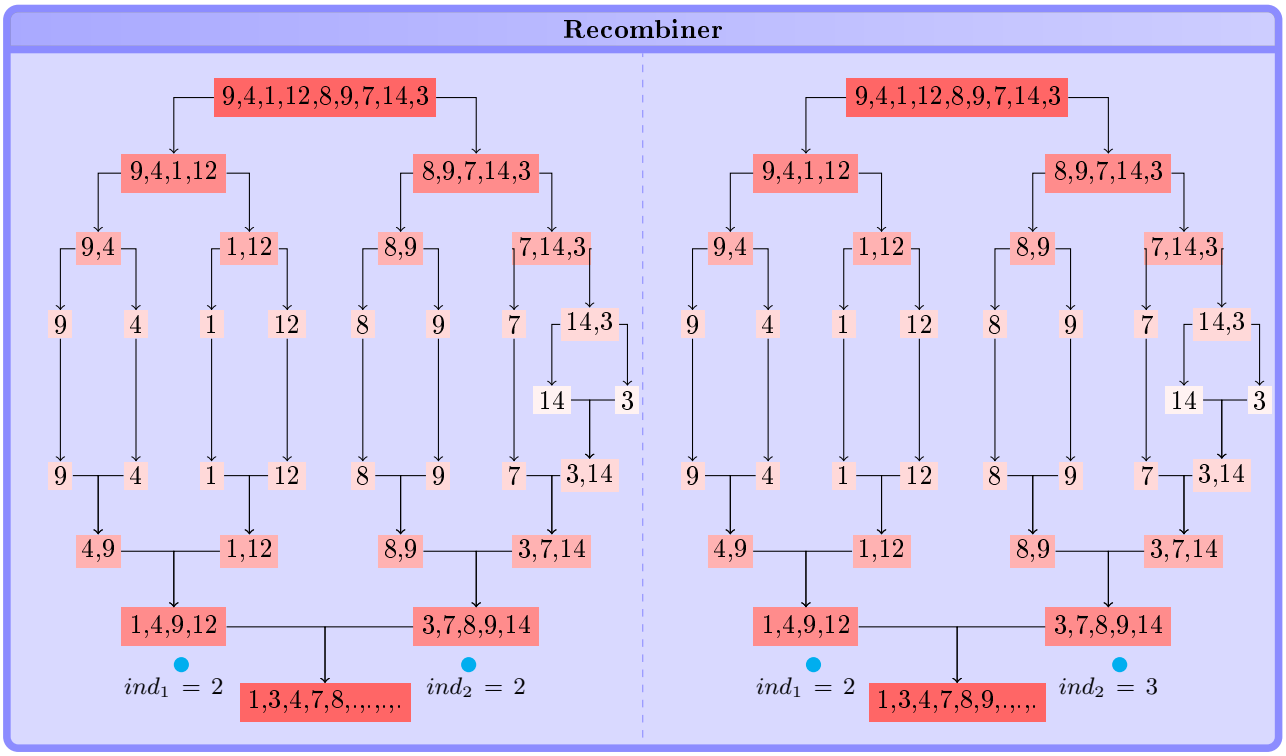


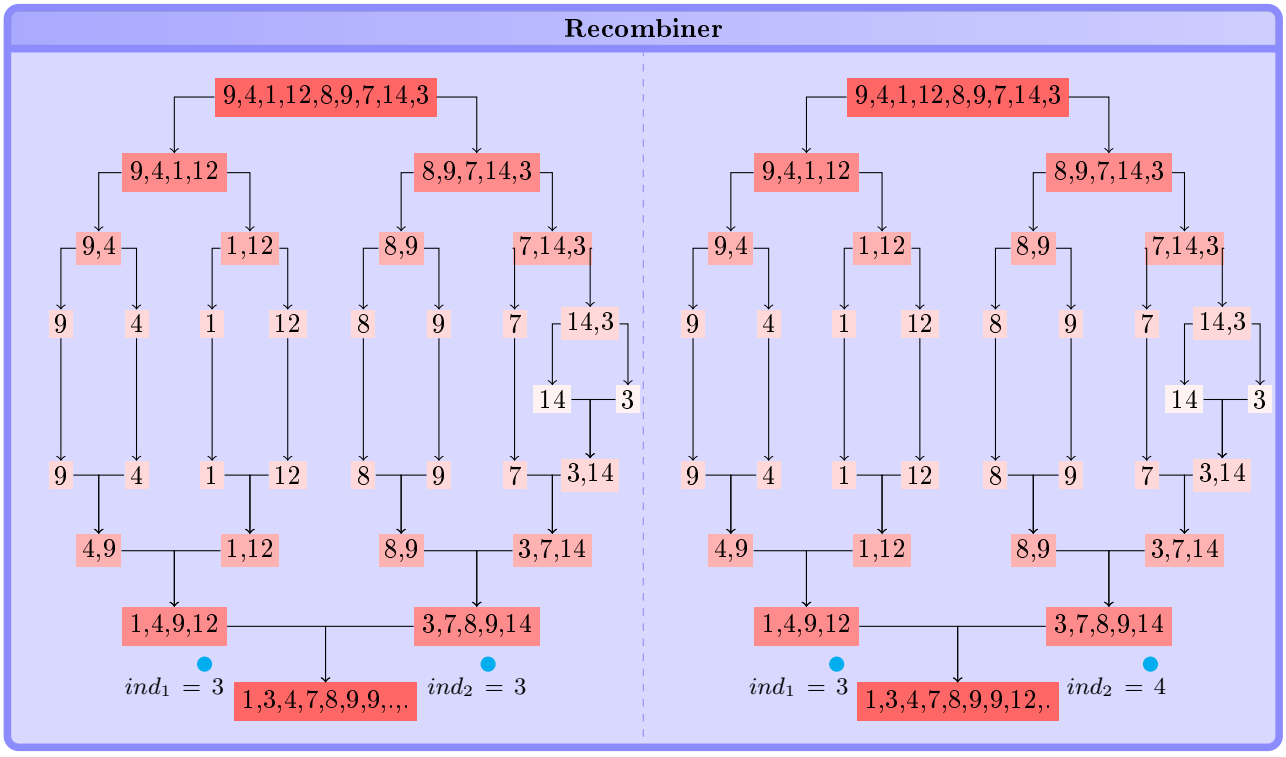

Recombiner
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
9
4
1
12
8
9
7
3,14
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
9
4
1
12
8
9
7
3,14
4,9
1,12
8,9
3,7,14

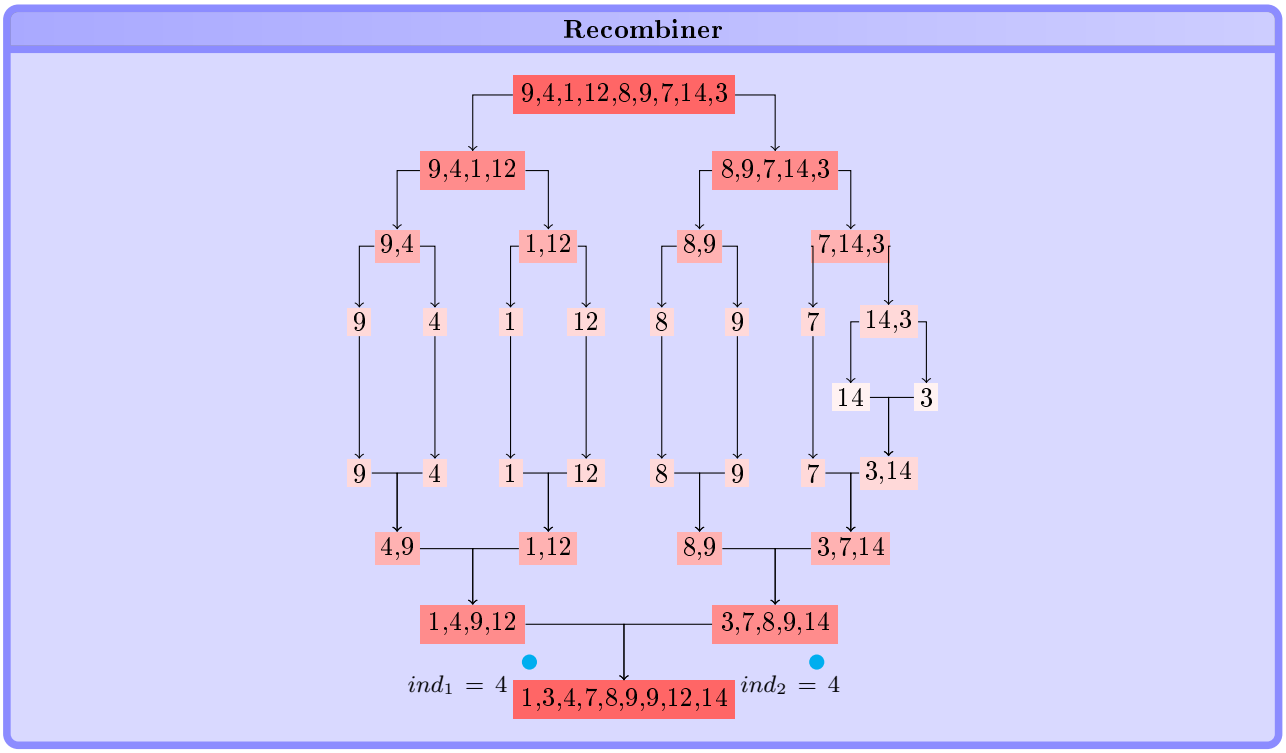

Recombiner
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
9
4
1
12
8
9
7
3,14
4,9
1,12
8,9
3,7,14
1,4,9,12
3,7,8,9,14


Recombiner
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
3,14
4,9
1,12
8,9
3,7,14
1,4,9,12
3,7,8,9,14
$ind_1 = 0$
1,.,.,.,.,.,.,.,.
$ind_2 = 0$
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
3,14
4,9
1,12
8,9
3,7,14
1,4,9,12
3,7,8,9,14
$ind_1 = 1$
1,3,.,.,.,.,.,.,.
$ind_2 = 0$


Recombiner
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
3,14
4,9
1,12
8,9
3,7,14
1,4,9,12
3,7,8,9,14
$ind_1 = 1$
1,3,4,.,.,.,.,.,.
$ind_2 = 1$
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
3,14
4,9
1,12
8,9
3,7,14
1,4,9,12
3,7,8,9,14
$ind_1 = 2$
1,3,4,7,.,.,.,.,.
$ind_2 = 1$


Recombiner
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
3,14
4,9
1,12
8,9
3,7,14
1,4,9,12
3,7,8,9,14
$ind_1 = 2$
1,3,4,7,8,.,.,.,.
$ind_2 = 2$
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
3,14
4,9
1,12
8,9
3,7,14
1,4,9,12
3,7,8,9,14
$ind_1 = 2$
1,3,4,7,8,9,.,.,.
$ind_2 = 3$


Recombiner
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
3,14
4,9
1,12
8,9
3,7,14
1,4,9,12
3,7,8,9,14
$ind_1 = 3$
1,3,4,7,8,9,9,.,.
$ind_2 = 3$
9,4,1,12,8,9,7,14,3
9,4,1,12
8,9,7,14,3
9,4
1,12
8,9
7,14,3
9
4
1
12
8
9
7
14,3
14
3
3,14
4,9
1,12
8,9
3,7,14
1,4,9,12
3,7,8,9,14
$ind_1 = 3$
1,3,4,7,8,9,9,12,.
$ind_2 = 4$



Le pseudo-code pour la fonction $fusionne(t_1, t_2)$ est donc :

Fonction fusionne(t_1,t_2) :
    $w \leftarrow [\,]$
    $n_1 \leftarrow longueur(t_1)$
    $n_2 \leftarrow longueur(t_2)$
    $ind_1 \leftarrow 0$
    $ind_2 \leftarrow 0$
    Tant que ($ind_1 < n_1$ **ET** $ind_2 < n_2$) **faire**
        Si ($t_{1_{ind_1}} < t_{2_{ind_2}}$) **Alors**
            ajouter $t_{1_{ind_1}}$ à w
            $ind_1 \leftarrow ind_1 + 1$
        Sinon
            ajouter $t_{2_{ind_2}}$ à w
            $ind_2 \leftarrow ind_2 + 1$
        Fin Si
    Fait
    ajouter à w les éléments restant dans $t1$ ou $t2$
    Retourner w
Fin

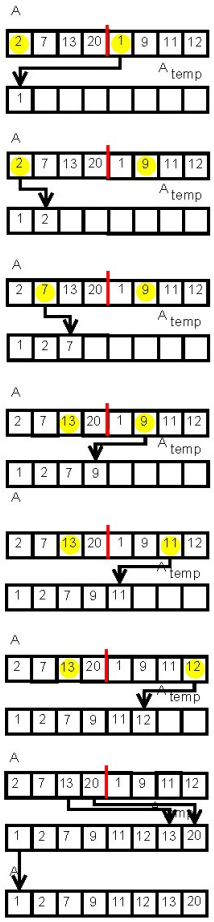
Algorithme 8: Fonction $fusionne$

Définir ensuite une fonction $tri\_fusion(t)$ utilisant la fonction précédente et fonctionnant de manière récursive :on coupe la liste initiale en 2 et on appelle la fonction tri_fusion avec chaque moitié de liste jusqu'à ce que nous ayons des sous-listes à un seul élément donc triées. Il suffit ensuite de fusionner tous les morceaux.

Fonction tri_fusion(t) :
    $n \leftarrow longueur(t)$
    Si ($n <= 1$) **Alors**
        Retourner t
    Sinon
        Retourner $fusionne(tri\_fusion(partie\_gauche(t)), tri\_fusion(partie\_droite(t)))$
    Fin Si
Fin

Algorithme 9: Fonction tri_fusion

Voici ci-dessous le principe de la fonction *fusionne* sur un exemple :



Dans l'exemple ci-contre le tableau A est divisé en de sous-tableaux par la ligne verticale rouge. On utilise un tableau temporaire pour les listes fusionnées (noté Atemp). Les éléments qu'on compare sont signalés par un cercle jaune. La flèche représente le déplacement d'un élément dans le tableau temporaire.

- On compare le premier élément du tableau de gauche (2) avec le premier élément du tableau de droite (1) ; 1 est inférieur, donc on l'envoie dans le tableau temporaire, puis on avance dans le tableau de droite : l'élément suivant est 9.
- On compare les deux éléments courants (2 et 9). Comme 2 est le plus petit, on l'ajoute au tableau temporaire et on avance dans le tableau de gauche. L'élément suivant est 7.
- On compare les deux éléments courants (7 et 9). Comme 7 est le plus petit, on l'ajoute au tableau temporaire et on avance dans le tableau de gauche. L'élément suivant est 13.
- On compare les deux éléments courants (13 et 9). Comme 9 est le plus petit, on l'ajoute au tableau temporaire et on avance dans le tableau de droite. L'élément suivant est 11.
- On compare les deux éléments courants (13 et 11). Comme 11 est le plus petit, on l'ajoute au tableau temporaire et on avance dans le tableau de droite. L'élément suivant est 12.
- On compare les deux éléments courants (13 et 12). Comme 12 est le plus petit, on l'ajoute au tableau temporaire et on avance dans le tableau de droite : il n'y a plus d'élément.
- Comme il n'y a plus d'élément dans le sous-tableau de droite, on copie les éléments restants du sous-tableau de gauche dans le tableau temporaire (il reste 13 et 20).
- On recopie le tableau temporaire dans A.

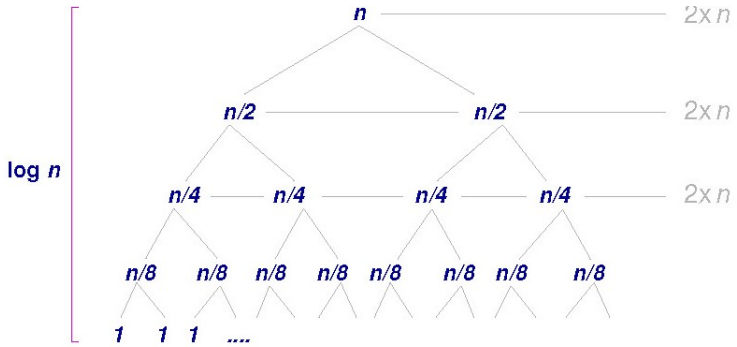
FIGURE 4 – Tri fusion

4.2 Complexité

on peut évaluer à $\Theta(1)$ le coût de la scission de la liste de départ en deux sous-listes. De même le coût de la fusion de deux sous-listes est en $\Theta(n)$. L'équation récursive du tri par fusion est donc à l'étape n si T est le coût

$$T(n) = 1 + \Theta(n) + 2.T(n/2) \tag{5}$$

On en déduit que le tri par fusion est en $\Theta(n \log n)$. On le vérifie en cumulant les nombres de comparaisons effectuées à chaque niveau de l'arbre qui représente l'exécution de la fonction : chaque nœud correspond à un appel de la fonction, ses fils correspondent aux deux appels récursifs, et son étiquette indique la longueur de la suite.



La hauteur de l'arbre est donc $\log_2 n$ et à chaque niveau le cumul des traitements locaux(scission et fusion) est $\Theta(n)$, d'où on déduit un coût total de $\Theta(n).\log_2 n = \Theta(n \log n)$.

FIGURE 5 – Arbre des appels récursifs du tri fusion

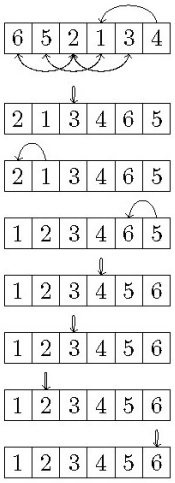
5 Le tri rapide ou Quicksort

5.1 Principe

Le principe consiste toujours à fusionner de deux sous-listes déjà triées. Par contre, la manière dont nous allons constituer ces sous-listes va différer du tri fusion. Au lieu de couper basiquement la liste en deux, nous allons choisir un pivot, puis mettre tous les éléments plus petits que le pivot dans la sous-liste gauche, et tous les éléments plus grands que le pivot dans la sous-liste droite. On trie ensuite les sous-listes gauche et droite en faisant un appel récursif à la procédure.Une fois

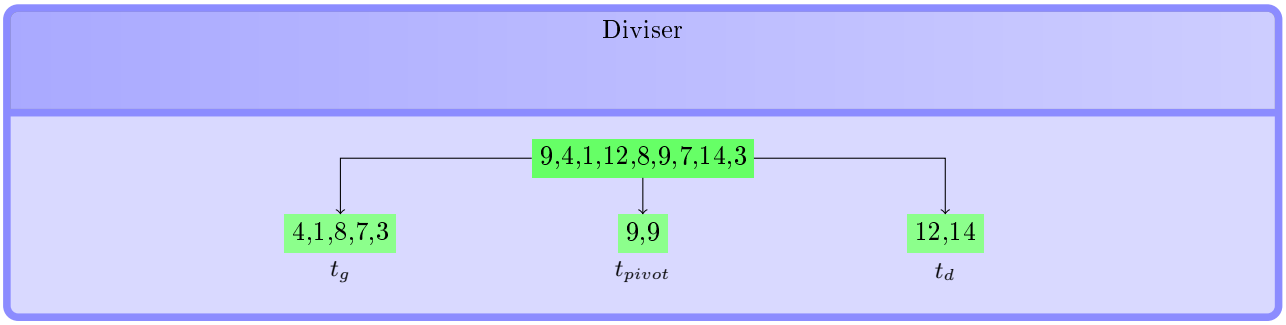
les sous-listes de taille unité, il suffit de fusionner les morceaux. Le tri rapide a été inventé par le scientifique C.A.R. Hoare en 1961.

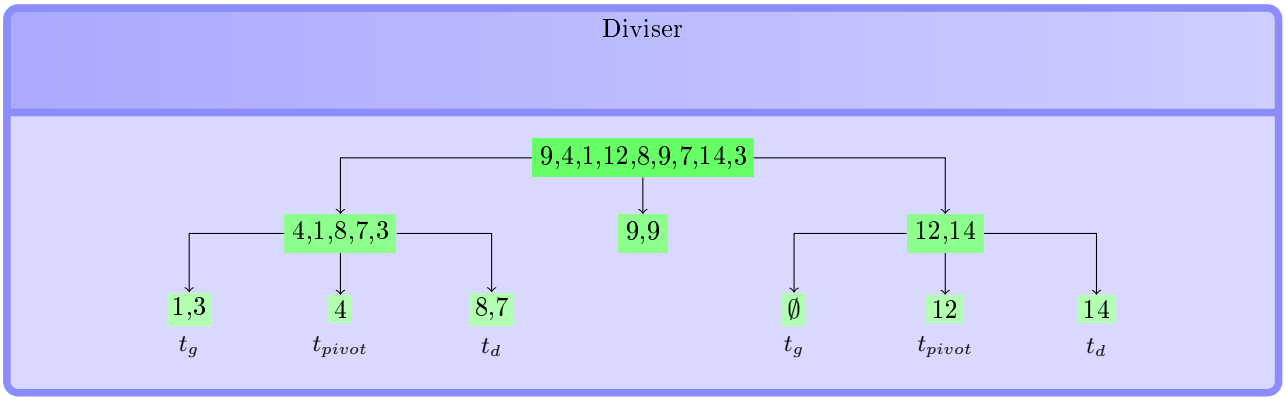
Prenons un exemple : on choisit ici le pivot à droite donc lors de la première découpe, le pivot est 4.

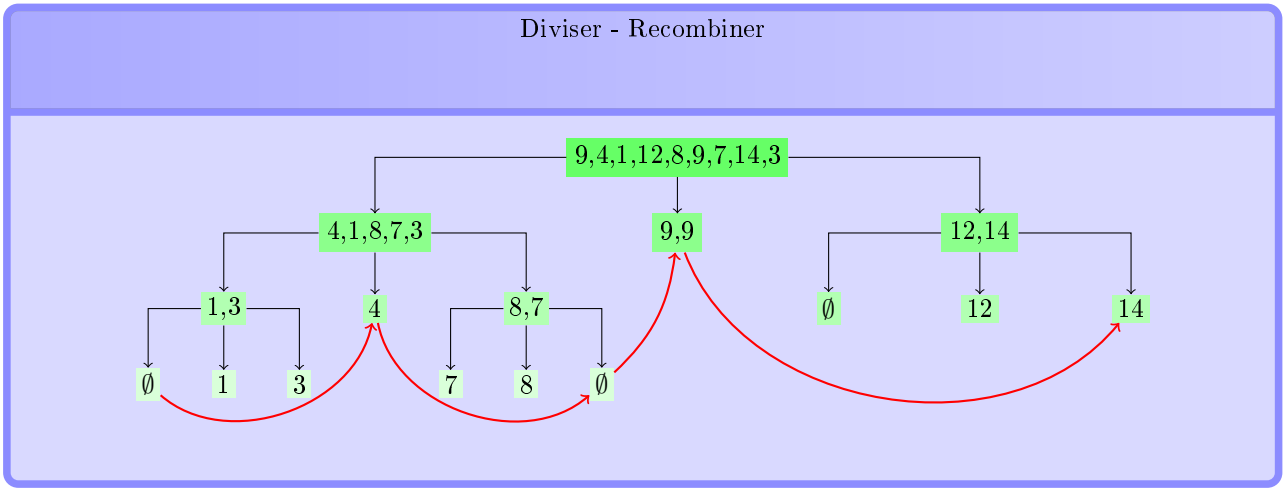


— on compare 6 et 4, 6 est plus grand, c'est le premier plus grand, en position 0 ;
— on compare 5 et 4, pas d'échange ;
— on compare 2 et 4, 2 est plus petit, donc on le met à la place du premier plus grand (5) en position 0. Le premier plus grand est maintenant 6 en position 1 ;
— on compare 1 et 4, 1 est plus petit, on l'échange avec le premier plus grand (6) en position 1. Le premier plus grand est 5 en position 2 ;
— on compare 3 et 4, 3 est plus petit, on l'échange avec le premier plus grand (5) en position 2. Le premier plus grand est 6 en position 3 ;
— terminé, on pose 4 à sa position, à la place du premier plus grand en position 3. Tous les éléments avant sont plus petits, les éléments après sont plus grands ;
— on trie les sous tableaux [2, 1, 3] et [6, 5] récursivement.
— on obtient deux tableaux [1, 2, 3] et [5, 6], que l'on fusionne avec le pivot pour obtenir [1, 2, 3, 4, 5, 6].

FIGURE 6 – Tri rapide







Le pseudo-code de l'algorithme est le suivant :

```
Fonction quicksort(t) :
    n ← longueur(t)
    Si (n <= 1) Alors
        Retourner t
    Sinon
        pivot ← t_0
        tg, t_pivots, td ← [], [], []
        Pour i de 0 à n − 1 faire
            Si (t_i < pivot) Alors
                ajouter t_i à tg
            Sinon
                Si (t_i == pivot) Alors
                    ajouter t_i à t_pivots
                Sinon
                    ajouter t_i à td
                Fin Si
            Fin Si
        Fin Pour
        Retourner quicksort(tg) + t_pivots + quicksort(td)
    Fin Si
Fin
```

Algorithme 10: Fonction *quicksort*

On peut aussi faire une version en place du tri rapide. c'est un peu plus complexe. Il faut créer une première fonction $partitionne(m, debut, fin, ind\_pivot)$ qui partitionne la partie de la liste m située entre $debut$ et fin autour de t_{ind_pivot} et renvoie la nouvelle position du pivot

```
Fonction partitionne(m,debut,fin,ind_pivot) :
    pivot ← m_{ind_pivot}
    i, j ← debut + 1, fin − 1
    permuter(m_{ind_pivot}, m_{debut}) on place le pivot au début
    Tant que (i <= j) faire
        Si (m_i <= pivot) Alors
            i ← i + 1
        Sinon
            permuter (m_i,m_j)
            j ← j − 1
        Fin Si
    Fait
    permuter(m_{i−1}, m_{debut}) on remet le pivot à sa place
    Retourner i − 1
Fin
```

Il faut ensuite créer la fonction récursive qui utilise la fonction précédente :

```
Fonction quicksort_en_place(t, d, f) :
    Si (f − d > 1) Alors
        ind_pivot ← partitionne(t, d, f, d)
        quicksort_en_place(t, d, ind_pivot)
        quicksort_en_place(t, ind_pivot + 1, f)
    Fin Si
    Retourner t
Fin
```

Algorithme 11: Fonction *quicksort_en_place*

5.2 Complexité

On remarque que la complexité de la fonction *partitionne* est bien linéaire en la longueur de la liste considérée.

Le pire des cas pour cet algorithme est celui où le pivot vient toujours à une extrémité du sous-tableau allant de l'indice *debut* à l'indice fin pour chaque appel à la fonction partition. Ce sera par exemple le cas si on trie un tableau déjà trié. Il sera alors fait appel à la fonction partition pour des listes de longueur n, puis $n-1$, puis $n-2$,..., puis 2. La somme des complexités de ces fonctions donne un algorithme en $\Theta(n^2)$.

Le meilleur des cas est celui où la liste est toujours partagée en deux sous-listes de même longueur. On aura en tout :
— un appel à la fonction partition sur une liste de longueur n
— deux appels à la fonction partition sur des listes de longueurs environ n/2
— quatre appels à la fonction partition sur des listes de longueurs environ n/4
— ...
On peut évaluer à $\Theta(n)$ le coût de la partition de la liste de départ en deux sous-listes. Par contre le coût de la fusion de deux sous-listes est en $\Theta(1)$. L'équation récursive du tri rapide est donc à l'étape n si T est le coût

$$T(n) = 1 + \Theta(n) + 2.T(n/2) \tag{6}$$

D'où une complexité en $\Theta(n)$ pour la liste de longueur n, une complexité totale en $\Theta(n)$ pour les deux sous-listes de longueur $n/2$, une complexité totale en $\Theta(n)$ pour les quatre sous-listes de longueur $n/4$, ... Comme la longueur est divisée par deux quand on passe de n à $n/2$, puis de $n/2$ à $n/4$, on a $\log_2 n$ appels récursifs en en tout une complexité de l'ordre de $n \log_2 n$.

Dans le cas moyen, c'est plus difficile à évaluer. En effet, les sous-listes gauche et droite du pivot n'ont aucune raison d'être équilibrées si le pivot est choisi au hasard. il faut donc faire la démonstration en supposant que le pivot à chaque étape peut être en n'importe quelle position de manière équiprobable. Il faut donc une moyenne statistique de la complexité de chaque cas. Au bilan, après une démonstration un peu longue, on montre que la complexité est en $n \log n$.

Remarquons que pour éviter d'avoir une complexité en n^2 lorsqu'on trie une liste déjà triée, on peut ajouter, au tout début de la fonction *partitionne*, une ligne prenant une donnée au hasard dans la liste traitée et l'échangeant avec la donnée qui se trouve en première position avant d'attaquer l'algorithme classique.

6 Recherche de médiane

Si l'on veut déterminer une médiane d'un tableau, le plus simple est de trier le tableau puis de renvoyer l'élément d'indice $[\frac{n}{2}]$ (si les indices vont de 0 à $n-1$). Vu qu'on sait trier en $n \log n$, ça nous coûte $n \log n$. Cependant, on voit assez bien (intuitivement) qu'on fait trop de travail, et il est effectivement assez facile de faire mieux en réutilisant la fonction de partition du tri rapide :
— On prend un pivot (disons le premier élément) et on partitionne. On récupère l'indice *ind_pivot* du pivot.
— Si $ind\_pivot = [\frac{n}{2}]$, on a fini
— Si $ind\_pivot > [\frac{n}{2}]$, la médiane se trouve dans la sous-liste de gauche. On relance donc un appel récursif dans cette partie en recherchant toujours l'élément de rang $[\frac{n}{2}]$
— Si $ind\_pivot < [\frac{n}{2}]$, la médiane se trouve dans la sous-liste de droite. On relance donc un appel récursif dans cette partie en recherchant l'élément de rang $[\frac{n}{2}] - ind\_pivot$. C'est là que ça peut coincer s'il y a plusieurs pivots identiques.
Au bilan on ne gagne pas grand chose par rapport au tri rapide.