

Objectif

☞ Complexité en moyenne.

Problème

[Complexité en moyenne du tri rapide]

1. Réaliser un suivi à la trace de la procédure `partitionBis` appliquée aux instances suivantes :

$$I_1 = [2, 6, 0, 4, 3, 1, 5], I_2 = [7, 6, 5, 4, 3, 2, 1] \text{ et } I_3 = [1, 2, 3, 4, 5, 6, 7].$$

Instruction	Description/Remarque	T	indpiv	pospiv	x	j
<code>partitionBis(I₁, 0, 6)</code>	T <- I ₁ , deb <- 0, fin <- 6	[2, 6, 0, 4, 3, 1, 5]				
<code>indpiv <- deb</code>	indpiv <- 0	[2, 6, 0, 4, 3, 1, 5]	0			
<code>pospiv <- deb</code>	pospiv <- 0	[2, 6, 0, 4, 3, 1, 5]	0	0		
<code>x <- T(deb)</code>	x <- T(0)	[2, 6, 0, 4, 3, 1, 5]	0	0	2	
[Pour] j <- deb + 1	j <- 0 + 1	[2, 6, 0, 4, 3, 1, 5]	0	0	2	1
[Pour] j <= fin	1 <= 6 est vrai, la boucle continue	[2, 6, 0, 4, 3, 1, 5]	0	0	2	1
[Si] T(j) <= x	6 <= 2 est faux, on retourne a la boucle	[2, 6, 0, 4, 3, 1, 5]	0	0	2	1
[Pour] j <- j + 1	j <- 1 + 1	[2, 6, 0, 4, 3, 1, 5]	0	0	2	2
[Pour] j <= fin	2 <= 6 est vrai, la boucle continue	[2, 6, 0, 4, 3, 1, 5]	0	0	2	2
[Si] T(j) <= x	0 <= 2 est vrai	[2, 6, 0, 4, 3, 1, 5]	0	0	2	2
<code>pospiv <- pospiv + 1</code>	pospiv <- 0 + 1	[2, 6, 0, 4, 3, 1, 5]	0	1	2	2
[Si] j > pospiv	2 > 1 est vrai	[2, 6, 0, 4, 3, 1, 5]	0	1	2	2
<code>echanger(T, pospiv, j)</code>	T(1) <- 0, T(2) <- 6	[2, 0, 6, 4, 3, 1, 5]	0	1	2	2
[Pour] j <- j + 1	j <- 2 + 1	[2, 0, 6, 4, 3, 1, 5]	0	1	2	3
[Pour] j <= fin	3 <= 6 est vrai, la boucle continue	[2, 0, 6, 4, 3, 1, 5]	0	1	2	3
[Si] T(j) <= x	4 <= 2 est faux	[2, 0, 6, 4, 3, 1, 5]	0	1	2	3
[Pour] j <- j + 1	j <- 3 + 1	[2, 0, 6, 4, 3, 1, 5]	0	1	2	4
[Pour] j <= fin	4 <= 6 est vrai, la boucle continue	[2, 0, 6, 4, 3, 1, 5]	0	1	2	4
[Si] T(j) <= x	3 <= 2 est faux	[2, 0, 6, 4, 3, 1, 5]	0	1	2	4
[Pour] j <- j + 1	j <- 4 + 1	[2, 0, 6, 4, 3, 1, 5]	0	1	2	5
[Pour] j <= fin	5 <= 6 est vrai, la boucle continue	[2, 0, 6, 4, 3, 1, 5]	0	1	2	5
[Si] T(j) <= x	1 <= 2 est vrai	[2, 0, 6, 4, 3, 1, 5]	0	1	2	5
<code>pospiv <- pospiv + 1</code>	pospiv <- 1 + 1	[2, 0, 6, 4, 3, 1, 5]	0	2	2	5
[Si] j > pospiv	5 > 2 est vrai	[2, 0, 6, 4, 3, 1, 5]	0	2	2	5
<code>echanger(T, pospiv, j)</code>	T(2) <- 1, T(5) <- 6	[2, 0, 1, 4, 3, 6, 5]	0	2	2	5
[Pour] j <- j + 1	j <- 5 + 1	[2, 0, 1, 4, 3, 6, 5]	0	2	2	6
[Pour] j <= fin	6 <= 6 est vrai, la boucle continue	[2, 0, 1, 4, 3, 6, 5]	0	2	2	6
[Si] T(j) <= x	5 <= 2 est faux	[2, 0, 1, 4, 3, 6, 5]	0	2	2	6
[Pour] j <- j + 1	j <- 6 + 1	[2, 0, 1, 4, 3, 6, 5]	0	2	2	7
[Pour] j <= fin	7 <= 6 est faux, la boucle termine	[2, 0, 1, 4, 3, 6, 5]	0	2	2	7
[Si] indpiv < pospiv	0 < 2 est vrai	[2, 0, 1, 4, 3, 6, 5]	0	2	2	7
<code>echanger(T, indpiv, pospiv)</code>	T(0) <- 1, T(2) <- 2	[1, 0, 2, 4, 3, 6, 5]	0	2	2	7

Nombre d'échanges : 3, Nombre de comparaison : 6

Notre tableau en sorti [1, 0, 2, 4, 3, 6, 5] a été partitionné en deux parties autour du pivot 2. A gauche, l'on a $T_g = [1, 0]$ dont tous les éléments $g \in T_g$ satisfont $g < 2$ et à droite l'on a $T_d = [4, 3, 6, 5] \mid \forall d \in T_d, d > 2$. La schéma de partition de l'algorithme `partitionBis` parcourt le tableau pour compter le nombre des valeurs qui sont inférieures au pivot. En le comptant, s'il y a des valeurs plus grand que le pivot dans le sous-tableau à gauche, `partitionBis` va effectuer un échange. Donc, on verra que ce schéma doit opérer $n-1$ comparaisons des éléments du tableau à trier.

Instruction	Description/Remarque	T	indpiv	pospiv	x	j
partitionBis(I ₂ , 0, 6)	T <- I ₂ , deb <- 0, fin <- 6	[7, 6, 5, 4, 3, 2, 1]				
indpiv <- deb	indpiv <- 0	[7, 6, 5, 4, 3, 2, 1]	0			
pospiv <- deb	pospiv <- 0	[7, 6, 5, 4, 3, 2, 1]	0	0		
x <- T(deb)	x <- 7	[7, 6, 5, 4, 3, 2, 1]	0	0	7	
[Pour] j <- deb + 1	j <- 0 + 1	[7, 6, 5, 4, 3, 2, 1]	0	0	7	1
[Pour] j <= fin	1 <= 6 est vrai, la boucle continue	[7, 6, 5, 4, 3, 2, 1]	0	0	7	1
[Si] T(j) <= x	6 <= 7 est vrai	[7, 6, 5, 4, 3, 2, 1]	0	0	7	1
pospiv <- pospiv + 1	pospiv <- 0 + 1	[7, 6, 5, 4, 3, 2, 1]	0	1	7	1
[Si] j > pospiv	1 > 1 est faux	[7, 6, 5, 4, 3, 2, 1]	0	1	7	1
[Pour] j <- j + 1	j <- 1 + 1	[7, 6, 5, 4, 3, 2, 1]	0	1	7	2
[Pour] j <= fin	2 <= 6 est vrai, la boucle continue	[7, 6, 5, 4, 3, 2, 1]	0	1	7	2
[Si] T(j) <= x	5 <= 7 est vrai	[7, 6, 5, 4, 3, 2, 1]	0	1	7	2
pospiv <- pospiv + 1	pospiv <- 1 + 1	[7, 6, 5, 4, 3, 2, 1]	0	2	7	2
[Si] j > pospiv	2 > 2 est faux	[7, 6, 5, 4, 3, 2, 1]	0	2	7	2
[Pour] j <- j + 1	j <- 2 + 1	[7, 6, 5, 4, 3, 2, 1]	0	2	7	3
[Pour] j <= fin	3 <= 6 est vrai, la boucle continue	[7, 6, 5, 4, 3, 2, 1]	0	2	7	3
[Si] T(j) <= x	4 <= 7 est vrai	[7, 6, 5, 4, 3, 2, 1]	0	2	7	3
pospiv <- pospiv + 1	pospiv <- 2 + 1	[7, 6, 5, 4, 3, 2, 1]	0	3	7	3
[Si] j > pospiv	3 > 3 est faux	[7, 6, 5, 4, 3, 2, 1]	0	3	7	3
[Pour] j <- j + 1	j <- 3 + 1	[7, 6, 5, 4, 3, 2, 1]	0	3	7	4
[Pour] j <= fin	4 <= 6 est vrai, la boucle continue	[7, 6, 5, 4, 3, 2, 1]	0	3	7	4
[Si] T(j) <= x	3 <= 7 est vrai	[7, 6, 5, 4, 3, 2, 1]	0	3	7	4
pospiv <- pospiv + 1	pospiv <- 3 + 1	[7, 6, 5, 4, 3, 2, 1]	0	4	7	4
[Si] j > pospiv	4 > 4 est faux	[7, 6, 5, 4, 3, 2, 1]	0	4	7	4
[Pour] j <- j + 1	j <- 4 + 1	[7, 6, 5, 4, 3, 2, 1]	0	4	7	5
[Pour] j <= fin	5 <= 6 est vrai, la boucle continue	[7, 6, 5, 4, 3, 2, 1]	0	4	7	5
[Si] T(j) <= x	2 <= 7 est vrai	[7, 6, 5, 4, 3, 2, 1]	0	4	7	5
pospiv <- pospiv + 1	pospiv <- 4 + 1	[7, 6, 5, 4, 3, 2, 1]	0	5	7	5
[Si] j > pospiv	5 > 5 est faux	[7, 6, 5, 4, 3, 2, 1]	0	5	7	5
[Pour] j <- j + 1	j <- 5 + 1	[7, 6, 5, 4, 3, 2, 1]	0	5	7	6
[Pour] j <= fin	6 <= 6 est vrai, la boucle continue	[7, 6, 5, 4, 3, 2, 1]	0	5	7	6
[Si] T(j) <= x	1 <= 7 est vrai	[7, 6, 5, 4, 3, 2, 1]	0	5	7	6
pospiv <- pospiv + 1	pospiv <- 5 + 1	[7, 6, 5, 4, 3, 2, 1]	0	6	7	6
[Si] j > pospiv	5 > 5 est faux	[7, 6, 5, 4, 3, 2, 1]	0	6	7	6
[Pour] j <- j + 1	j <- 6 + 1	[7, 6, 5, 4, 3, 2, 1]	0	6	7	7
[Pour] j <= fin	7 <= 6 est faux, la boucle termine	[7, 6, 5, 4, 3, 2, 1]	0	6	7	7
[Si] indpiv < pospiv	0 < 6 est vrai	[7, 6, 5, 4, 3, 2, 1]	0	6	7	7
echanger(T, indpiv, pospiv)	T(0) <- 1, T(6) <- 7	[1, 6, 5, 4, 3, 2, 7]	0	6	7	7

Nombre d'échanges : 1, Nombre de comparaison : 6

Chaque élément du tableau qu'on a comparé avec le pivot en était inférieur. Cependant, on a réussi à partitionner le tableau avec un seul échange dans le dernier étape. Avec le pivot 7, $T_g = [1, 6, 5, 4, 3, 2]$ et il vérifie $g < 7 \forall g \in T_g$.

Instruction	Description/Remarque	T	indpiv	pospiv	x	j
partitionBis(I ₃ , 0, 6)	T <- I ₃ , deb <- 0, fin <- 6	[1, 2, 3, 4, 5, 6, 7]				
indpiv <- deb	indpiv <- 0	[1, 2, 3, 4, 5, 6, 7]	0			
pospiv <- deb	pospiv <- 0	[1, 2, 3, 4, 5, 6, 7]	0	0		
x <- T(deb)	x <- 1	[1, 2, 3, 4, 5, 6, 7]	0	0	1	
[Pour] j <- deb + 1	j <- 0 + 1	[1, 2, 3, 4, 5, 6, 7]	0	0	1	1
[Pour] j <= fin	1 <= 6 est vrai, la boucle continue	[1, 2, 3, 4, 5, 6, 7]	0	0	1	1
[Si] T(j) <= x	2 <= 1 est faux	[1, 2, 3, 4, 5, 6, 7]	0	0	1	1
[Pour] j <- j + 1	j <- 1 + 1	[1, 2, 3, 4, 5, 6, 7]	0	0	1	2
[Pour] j <= fin	2 <= 6 est vrai, la boucle continue	[1, 2, 3, 4, 5, 6, 7]	0	0	1	2
[Si] T(j) <= x	3 <= 1 est faux	[1, 2, 3, 4, 5, 6, 7]	0	0	1	2
[Pour] j <- j + 1	j <- 2 + 1	[1, 2, 3, 4, 5, 6, 7]	0	0	1	3
[Pour] j <= fin	3 <= 6 est vrai, la boucle continue	[1, 2, 3, 4, 5, 6, 7]	0	0	1	3
[Si] T(j) <= x	4 <= 1 est faux	[1, 2, 3, 4, 5, 6, 7]	0	0	1	3
[Pour] j <- j + 1	j <- 3 + 1	[1, 2, 3, 4, 5, 6, 7]	0	0	1	4
[Pour] j <= fin	4 <= 6 est vrai, la boucle continue	[1, 2, 3, 4, 5, 6, 7]	0	0	1	4
[Si] T(j) <= x	5 <= 1 est faux	[1, 2, 3, 4, 5, 6, 7]	0	0	1	4
[Pour] j <- j + 1	j <- 4 + 1	[1, 2, 3, 4, 5, 6, 7]	0	0	1	5
[Pour] j <= fin	5 <= 6 est vrai, la boucle continue	[1, 2, 3, 4, 5, 6, 7]	0	0	1	5
[Si] T(j) <= x	6 <= 1 est faux	[1, 2, 3, 4, 5, 6, 7]	0	0	1	5
[Pour] j <- j + 1	j <- 5 + 1	[1, 2, 3, 4, 5, 6, 7]	0	0	1	6
[Pour] j <= fin	6 <= 6 est vrai, la boucle continue	[1, 2, 3, 4, 5, 6, 7]	0	0	1	6
[Si] T(j) <= x	7 <= 1 est faux	[1, 2, 3, 4, 5, 6, 7]	0	0	1	6
[Pour] j <- j + 1	j <- 6 + 1	[1, 2, 3, 4, 5, 6, 7]	0	0	1	7
[Pour] j <= fin	7 <= 6 est faux, la boucle termine	[1, 2, 3, 4, 5, 6, 7]	0	0	1	7
[Si] indpiv < pospiv	0 < 0 est faux	[1, 2, 3, 4, 5, 6, 7]	0	0	1	7

Nombre d'échanges : 0, Nombre de comparaison : 6

Tous les éléments de $I_3 = [1, 2, 3, 4, 5, 6, 7]$ sont déjà triés donc on effectue aucun échange. On constate qu'il y a encore $n - 1 = 6$ comparaisons parce qu'on a traversé le tableau à partir du premier élément jusqu'à l'avant-dernier élément. Le tableau est, bien entendu, bien partitionné autour $p = 1$, tel que $T_d = [2, 3, 4, 5, 6, 7]$.

2. Démontrer que la procédure partitionBis est correcte et analyser sa complexité.

Pour démontrer que la procédure partitionBis est correcte on doit prouver que donné n'importe quel instance, l'algorithme partitionBis renvoie un tableau qui est partitionné. Pour procéder, on définit donc un tableau partitionné est celui dont tous les éléments à gauche d'un élément dit pivot est inférieure au pivot, et tout élément à droite du pivot est supérieure au pivot. Afin d'analyser si le programme est correcte ou pas, on va s'orienter autour de deux questions principales : Est-ce que l'élément qu'on a choisit d'être le pivot finit d'être dans la bonne place? Si oui, est-ce que tous les éléments à gauche sont inférieurs au pivot et respectivement supérieurs pour les éléments à droite. Commençons avec la question de la position du pivot.

Si on observe notre suivie à la trace pour les trois instances, on voit que l'algorithme parcourt le tableau à partir du deuxième élément (le premier est choisi pour être le pivot), jusqu'à la fin. Si la valeur qu'on visite est inférieure au pivot, on incrémente pospiv. En effet, on parcourt le tableau et on compte le nombre d'éléments qui sont inférieurs au pivot. En d'autre termes, on compte le nombre de valeurs qui doivent être à gauche du pivot dans la configuration finale. Pour prouver que le pivot se trouve *toujours* dans la bonne place, on considère l'effet de la dernière condition i.f. La condition est : Si indpiv < pospiv, échange le pivot avec l'élément à T(pospiv). Il n'est pas évident mais avec un peu de réflexion on raisonne que indpiv reste 0 pendant l'exécution de l'algorithme, donc le seule fois où cette condition n'est pas satisfaite c'est si pospiv n'est jamais incrémenté. Quand est-ce que cela se passe? On a vu que pour l'instance I_3 , il n'y a aucun élément entre $T(1)$ et $T(fin)$ qui est moins que le pivot, 1. Donc, la condition $T(j)$ n'est jamais satisfaite et du coup on incrémente jamais la variable pospiv. Comme $0 < 0$ est faux, on ne fait pas bouger le pivot quand il est déjà à la bonne position. Comme on choisit le premier élément pour chaque appel à partitionBis, on effectue un échange du pivot si et seulement si il n'est pas dans la bonne place. Et donc, où est-ce qu'on fait bouger le pivot si cette condition est vérifiée? On le met à T(pospiv), la position dans le tableau avec pospiv éléments à gauche, qu'on va démontrer qu'ils sont tous inférieurs au pivot. Par conséquent, la procédure partitionBis assure que le pivot finit toujours dans la bonne position.

Procédons pour démontrer que tous les éléments à gauche du pivot sont inférieurs au pivot. Pour ce faire, on considère le fonctionnement des phrases i.f à l'intérieur de la boucle principale. Si l'élément qu'on visite est inférieur au pivot, on incrémente pospiv et on teste une autre condition. Si $j > pospiv$, on effectue un échange. Digérons qu'est-ce que ces branches de logiques font. Quand est-ce que cette condition n'est pas vérifiée? Comme j incrémente à chaque itération, cette condition n'est pas vérifiée si pospiv est égale à j. Cela se passe si on incrémente pospiv à chaque fois, ce qui se passe seulement quand la valeur qu'on visite est plus petite que le pivot. Donc, à chaque fois qu'on teste cette condition, il est vérifié s'il y a au moins un élément qu'on a visité qui est inférieur au pivot.

Pensons comment ça s'applique aux instances au-dessus. pour I_2 , chaque élément que l'on visite est inférieur au pivot, du coup pospiv est toujours égale à j et la condition nécessaire pour échanger deux éléments est fausse. On

effectue donc aucun échange jusqu'à la dernière condition où on teste si il ya des éléments inférieurs au pivot. Comme il y'en a 6, on fait un seul échange pour mettre le pivot à la bonne place, ce qu'on a discuté en haut. Ca c'est le cas où tous les éléments sont inférieurs au pivot, donc on incrémente pos_{piv} à chaque fois et donc on fait aucun échange dans la boucle principale. Si on réfléchit à ce qui se passe avec I_1 , des qu'il y a un élément qui est plus petit que le pivot et qu'on a déjà visité un élément plus grand, on effectue un échange pour mettre celui qui est plus petit à gauche de la position finale. Comme cela, lorsqu'on parcourt le tableau, on met les valeurs inférieures au pivot à gauche.

L'algorithme donc satisfait les deux conditions pour tous les instances - que le pivot soit dans la bonne place et que tous les éléments à gauche sont inférieurs au pivot; tous les éléments à droite du pivot sont supérieurs, et par définition il est correcte.

3. Quels changements, s'ils existent, à apporter au pseudo-code du tri rapide?

La valeur de sortie de la procédure `partitionBis` est exactement la même chose que pour `partition` parce que toutes les deux renvoient l'indice de la position finale de l'élément choisit comme le pivot. Si on donne en entrée le même tableau et on choisit le même élément comme pivot, on renvoie la même position finale. Donc, le seul changement que l'on doit apporter au pseudo-code du tri rapide est de changer la ligne 10 pour affecter le résultat de `partitionBis(T, deb, fin)` à π , au lieu de lui affecter le résultat de `partition`. Pour être clair, à la place d'appeler la procédure `partition` afin de partitionner notre tableau, on choisit le schématique de `partitionBis`.

4. Conduire une analyse de complexité en moyenne du tri rapide utilisant la procédure `partitionBis` à la place de la procédure `partition`.

Commençons l'analyse de complexité de la procédure `partitionBis` en étudiant la moyenne du nombre de comparaisons pour trier m tableau de taille n dont les éléments sont $\{0, 1, \dots, n-1\}$ mélangés selon le battage de Fisher-Yates. Fixons $m = 100$ et $n = \{10, 50, 110, \dots, 4950\}$ Il est facile à comptabiliser le nombre de comparaisons μ_{cmp} pour exécuter le tri rapide avec le schéma de partition `partitionBis`. Pour ce faire, on observe qu'il y a un comparaison chaque passage du boucle. Donc, on écrit le code suivant en C

```
int partition_bis_count(int *__arr, int __lo, int __hi, int *__ncmp, int *__nech) {
    int i = __lo;
    int ipivot = __lo;
    int pivot = __arr[__lo];

    for (int j = __lo + 1; j <= __hi; j++) {
        if (__arr[j] <= pivot) {
            ipivot++;
            if (j > ipivot) exchange_count(__arr, ipivot, j, __nech);
        }
        inc(__ncmp, 1); // increment once per check
    }

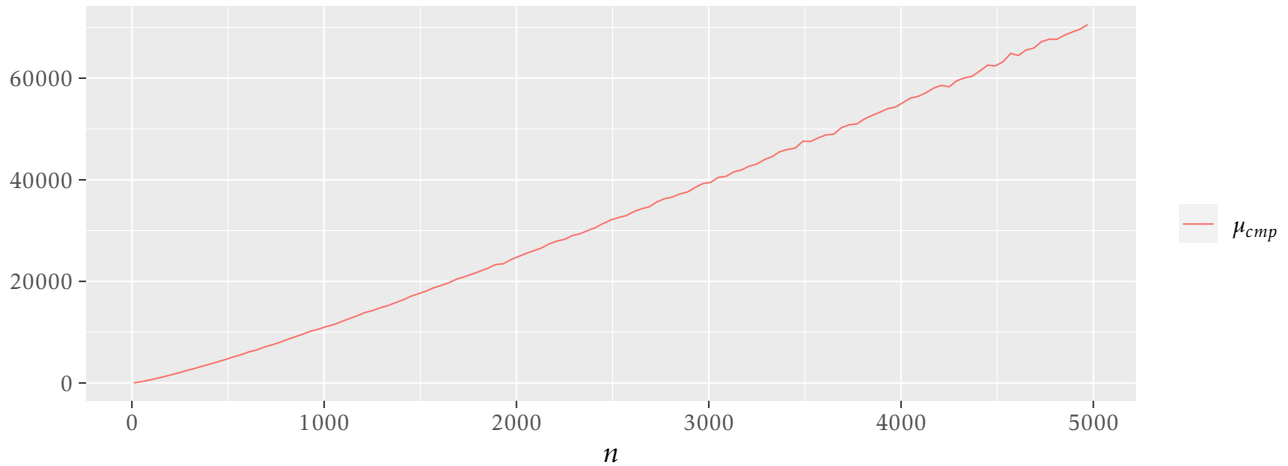
    if (i < ipivot) exchange_count(__arr, i, ipivot, __nech);
    return ipivot;
}
```

FIGURE 1 – Définition de la procédure `partition_bis_count` qui incrémente le nombre de comparaisons une fois par boucle. La comparaison des éléments du tableau est souligné en violet.

Pour $n_{min} = 10, n_{max} = 5000, \Delta n = 40$, on trie 100 tableau générés automatiquement et on comptabilise le nombre d'échanges n_{cmp} effectué après m tries. Avec $m = 100$, on calcule $\mu_{cmp} = \frac{n_{cmp}}{100}$

Moyenne des comparaisons

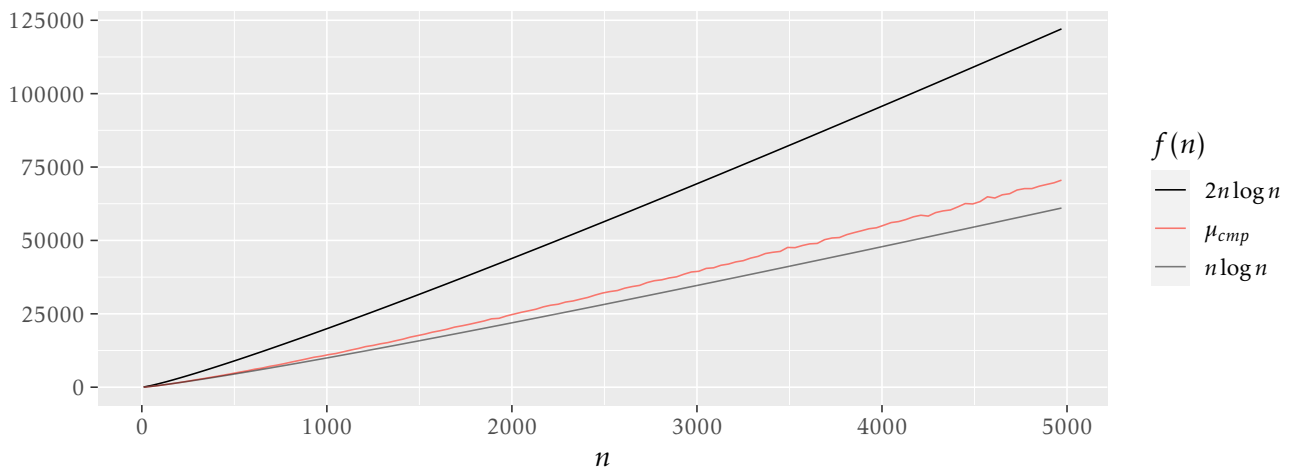
Schéma partitionBis appliqué aux tableaux de taille n



Il nous semble que le graphe n'explose pas lorsque n augmente. On peut tracer les courbes de la forme $\alpha n \log n$ pour établir des fonctions qui dominent, et qui sont dominé par, μ_{cmp} .

Moyenne des comparaisons

$\text{nombreComparaisons}(n) \in \Theta(n \log n)$



On estime que la moyenne nombre de comparaisons est une fonction de n qui est dominé par $2n \log n$ et qui domine $n \log n$. Alors, par définition, $\text{nombreComparaison}(n) \in \Theta(n \log n)$. On peut préciser la valeur de α dans l'équation $\text{nombreComparaison}(n) = \alpha n \log n$ si on exécute une regression linéaire de μ_{cmp} par rapport à $n \log n$.

Avec un coefficient de corrélation $r = 0.9999282$, on trouve la regression linéaire par rapport à $x = n \log n$ est : $L(n \log n) = L(x) = 1.16x - 549.56$.

Moyenne des comparaisons

$$\text{nombreComparaisons}(n) \approx 1.16(n \log n) - 549.56$$

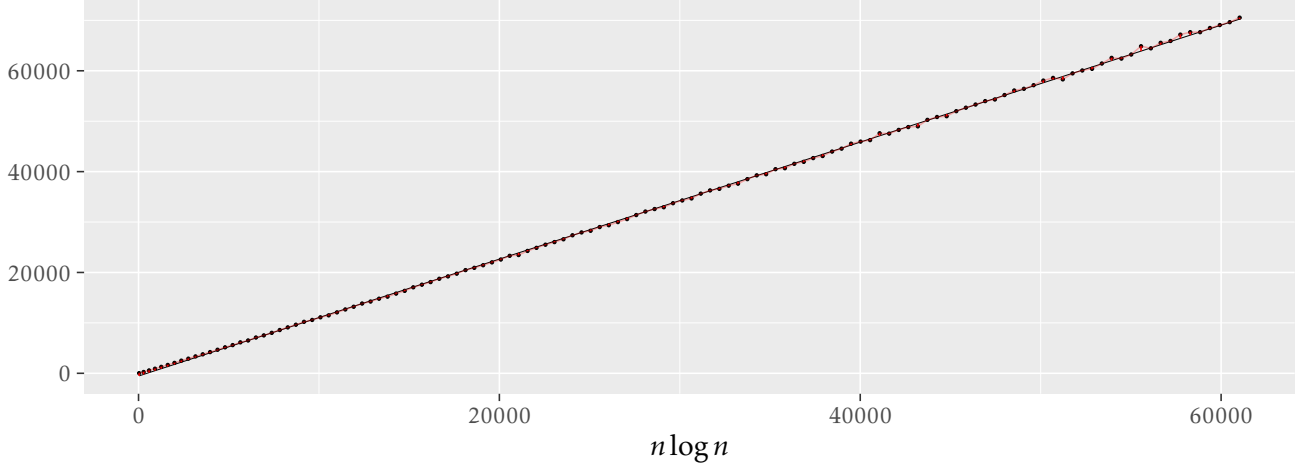


FIGURE 2 – Regression linéaire du variable μ_{cmp} par rapport à $n \log n$. Cette regression linéaire a un MSE = 63167 et un RMS = 251.3314

Maintenant, on va analyser comment la moyenne nombre d'échanges varie avec les mêmes paramètres n_{min} , n_{max} , Δn , et m définis au-dessus. Pour ce faire, on incrémente une variable n_{ech} à chaque fois qu'on échange deux éléments du tableau. Examinons la définition de la procédure `exchange_count` comme il est appelé dans Figure 1.

```
void exchange_count(int *__arr, int __i, int __j, int *__nech) {
    int temp = __arr[__i];
    __arr[__i] = __arr[__j];
    __arr[__j] = temp;
    inc(__nech, 1); // Once again, we only care about array assignments.
                  // increment the number of exchanged by 1 per call
}
```

FIGURE 3 – Implémentation d'une procédure qui échange deux éléments d'un tableau et qui incrémente une variable (`*__nech`) qui stocke le nombre d'échanges effectués jusqu'au présent.

Moyenne des échanges

$$\text{nombreEchanges}(n) \in \Theta(n \log n)$$

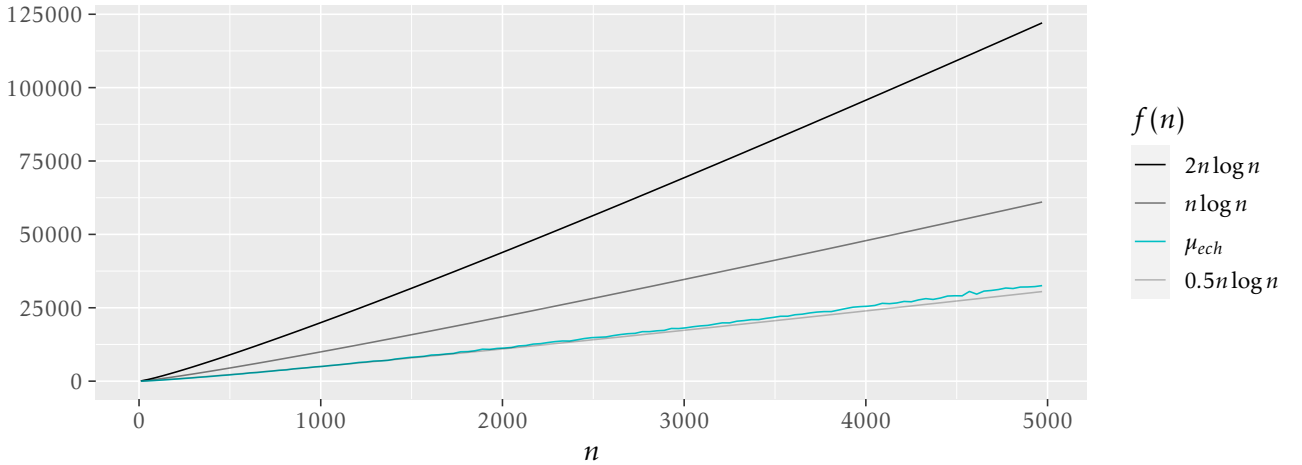


FIGURE 4 – Le moyenne nombre d'échanges se situe entre $n \log n$ et $\frac{1}{2}n \log n$. En réalisant une regression linéaire on trouve que $\text{nombreEchanges}(n) \approx 0.5399(n \log n) - 370.9022$ avec un coefficient de corrélation $r = 0.999788$, erreur moyenne carré (MSE) = 40429.08, RMS = 201.0698.

En accord avec nos suivis à la trace de la procédure `partitionBis`, on remarque qu'il y a, en moyenne, moins d'échanges que de comparaisons au cours du tri. Cela n'est pas étonnant. On a constaté que la procédure vérifie $(n-1)$ comparaisons pour **chaque** appel à `partitionBis`. Cependant, il se peut qu'on effectue aucun échange pourvu que le tableau donné en entrée est déjà trié. Ce déséquilibre explique la différence de valeurs de α dans la regression linéaire. Pour procéder, on visualise la complexité des échanges et comparaisons ensemble.

Moyenne des comparaisons et échanges

$\text{nombreCmpPlusEch}(n) \in \Theta(n \log n)$

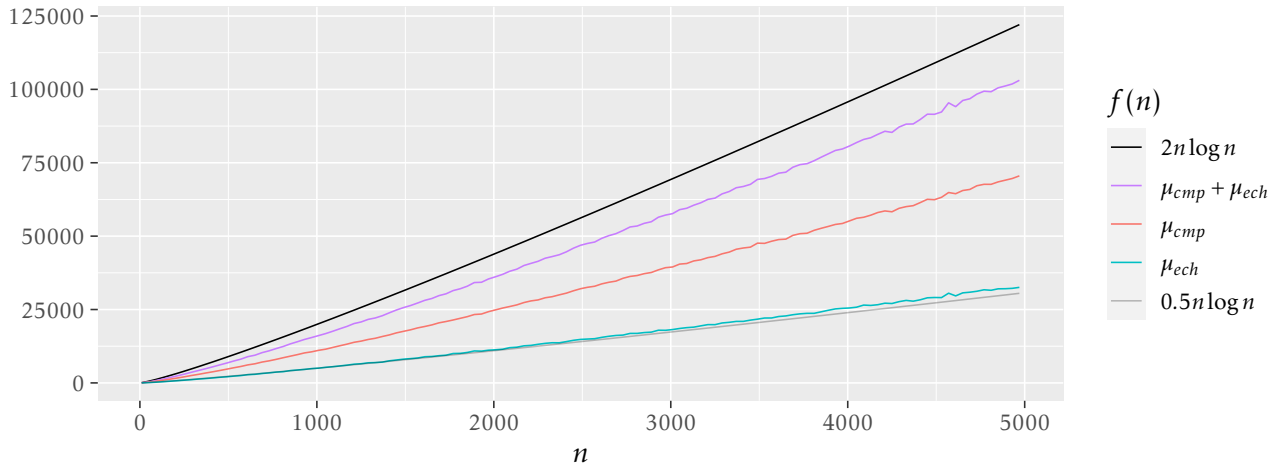


FIGURE 5 – Une regression linéaire révèle que $\text{nombreCmpPlusEch}(n)$ a une valeur de $\alpha_{sum} = 1.7$, $\text{nombreComparaisons}(n)$ a $\alpha_{cmp} = 1.16$, et $\text{nombreEchanges}(n)$ a $\alpha_{ech} = 0.54$.

Toujours bornés par $2n \log n$, on trouve que la valeur de alpha pour la moyenne du somme des comparaisons et échanges est en effet le somme des pentes de moyenne comparaisons et de moyenne échanges. Autrement dit, $1.7 = 1.16 + 0.54$.

5. D'après votre expérimentation, laquelle des deux méthodes partition et partitionBis est la plus efficace?

On peut conclure que la méthode partition est la plus efficace, parce que elle fait moins d'échanges que la méthode partitionBis. Considérez le graphe suivant.

Moyenne des comparaisons et échanges

$\text{nombreCmpPlusEch}(n) \in \Theta(n \log n)$

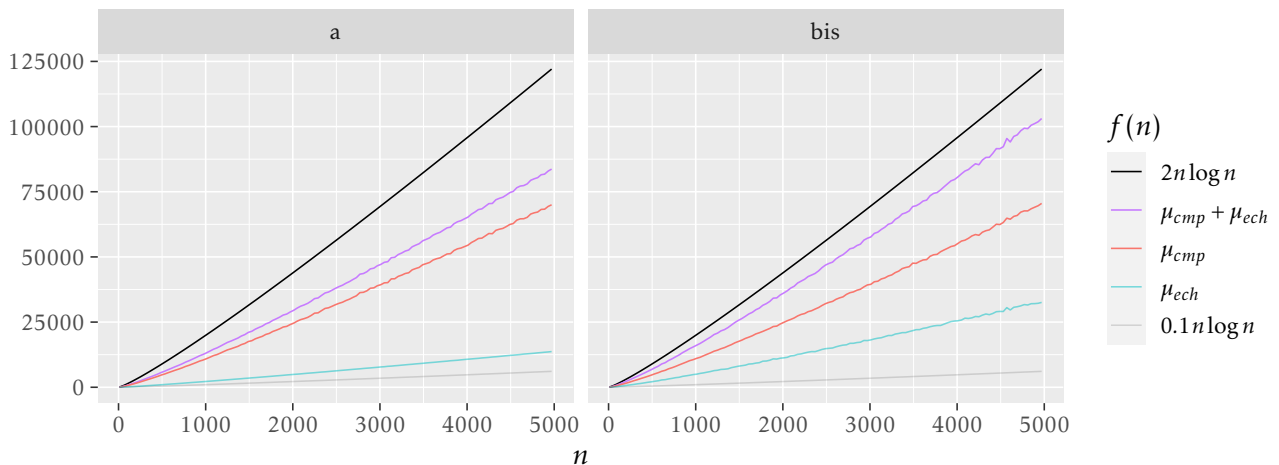


FIGURE 6 – Comparaisons des schématique de partition implémenté en partie A et partie B.

On remarque que le nombre de comparaisons est à peu près la même courbe, et cela s'explique par le fait que dans les deux méthodes on parcourt le tableau entier. En parti à, on le parcourt de deux directions différentes, jusqu'à ce que les pointeurs se croisent. En partie B, on parcourt le tableau entier depuis le début (+ 1) jusqu'à la fin. Pourtant, on fait moins d'échanges en partie A parce que avec cette première méthode on effectue des échanges qui sont plus "efficaces", c'est-à-dire qu'en sachant qu'on parcourt le tableau de deux directions différentes, on met deux éléments (un petit, un grand) dans le bon côté dans une seul opération.

Il ne s'arrête pas là. J'ai implémenté des autres schématiques de partition pour comparer avec partitionBis et partition.

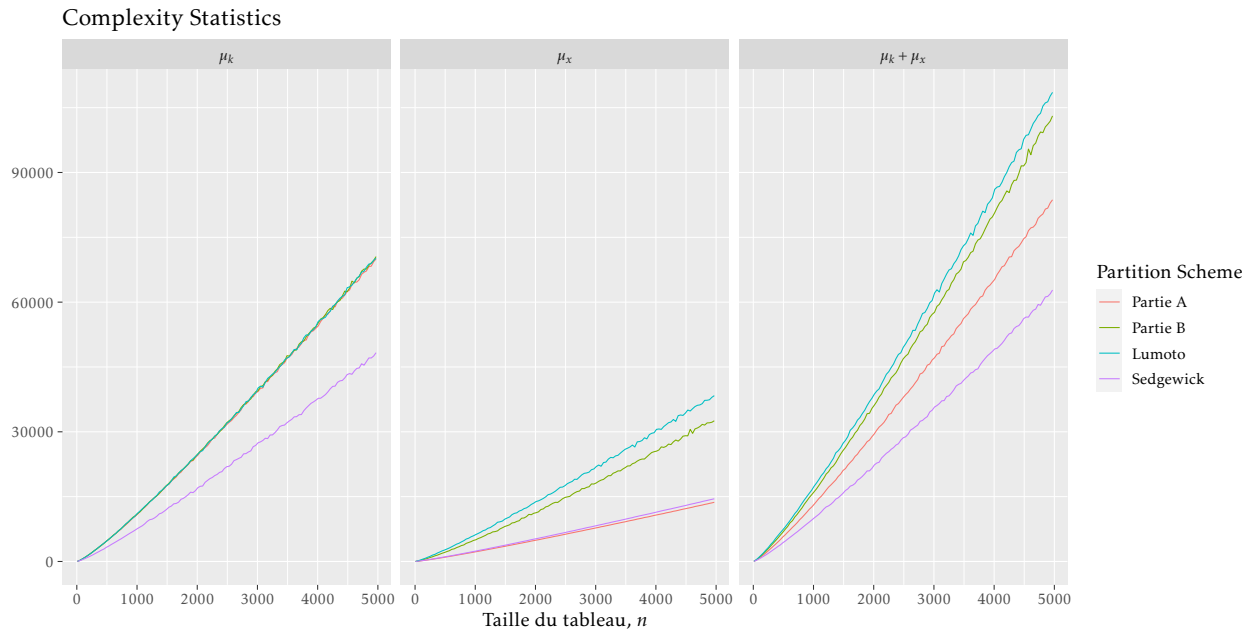


FIGURE 7 – Comparaisons des schématique de partition. La schématique Sedgewick est une version modifié de celui de partie A, et Lumoto diffère du partie B avec le choix du pivot.

La schématique proposé dans le livre *Algorithms* par Robert Sedgewick est, en effet, la plus efficace. Pour être honnête, je ne comprends pas exactement pourquoi, donc si vous avez des idées vous pouvez jeter un coup d'oeil dans le code, où la procédure `partition_tp` correspond à partie A et la procédure `partition_a` correspond à la partition de Sedgewick.

Dernières remarques

Le code que j'ai écrit pour ce TP dépend d'un projet externe à moi (qui fournit la fonctionnalité de générer des valeurs aléatoirement, stocker les données dans une objet Vector, etc). Ce projet ce trouve ici. Il faudrait compiler et installer la bibliothèque et les fichiers entête. Sinon, il y a un binaire compilé pour l'architecture x86-64 qui va effectuer 10 tries des tableau de taille 1000 avec les quatres schématiques de partition. Les graphes ont été dessiné avec `ggplot2` et c'était un plaisir d'apprendre un nouveau outil et surtout pour l'appliquer à un tp. Les fichiers `.csv` sont générés par le programme décrit en `qmain.c` si les fonctions `analyzeComplexity` sont appelées.