

# TEA SÉANCE 3

AAP - ALGORITHMIQUE AVANCÉE ET PROGRAMMATION

---

BASTOS SANTOS VITÓRIA  
FERREIRA BRASIL REBECA  
LESSA GUERRA BERNADO  
REIS GUEVARA DAVID

*École Centrale de Lille*

2022

---

# Liste de Figures

1	Résultat Tri Fusion dans un Tableau . . . . .	6
2	Graphique Tri Fusion dans un Tableau . . . . .	7
3	Résultat Tri Rapide dans un Tableau . . . . .	10
4	Graphique Tri Rapide dans un Tableau . . . . .	11
5	Results Tri Fusion dans une Liste . . . . .	14

---

# Résumé

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Tri Fusion</b>	<b>4</b>
2.1	Développement . . . . .	4
2.2	Code <i>fusion_sort.c</i> . . . . .	4
2.3	Resultats . . . . .	6
<b>3</b>	<b>Tri Rapide</b>	<b>8</b>
3.1	Développement . . . . .	8
3.2	Code <i>quick_sort.c</i> . . . . .	8
3.3	Resultats . . . . .	9
<b>4</b>	<b>Tri Fusion de Listes</b>	<b>11</b>
4.1	Développement . . . . .	11
4.2	Code <i>main.c</i> . . . . .	11
4.3	Code <i>fusion_sort_list.c</i> . . . . .	12
4.4	Resultats . . . . .	14
<b>5</b>	<b>Conclusion</b>	<b>15</b>
<b>6</b>	<b>Références</b>	<b>16</b>

---

# 1. Introduction

Ce TEA est une suite directe de la séance 3.

Pendant la séance, il a été expliqué comment les fonctions de Tris Optimaux fonctionnent, et le but de la TEA est de compléter les algorithmes afin d'obtenir trois fonctions :

1. Tri Fusion;
2. Tri Rapide;
3. Tri Fusion de listes.

Au cours de ce rapport, le fonctionnement de chaque fonction sera expliqué et leurs performances seront comparées.

---

Le développement se déroulera en trois parties, chacune dédiée à une fonction, puisqu'elles sont indépendantes.

## 2. Tri Fusion

### 2.1. Développement

Pour la fonction Tri Fusion, les codes fournis en cours (dans le **ex6**) ont été utilisés.  
Les fonctions utilisées pour construire cette partie-ci, sont:

- *fusionSort(T\_data d, int n)*
- *fusionsort(void \*base, int d, int f, int (\*compare\_ints)(const void \* a, const void \* b))*
- *compare\_ints(const void \* a, const void \* b)*
- *void fusionner(T\_elt t [], int d, int m, int f)*

Tout d'abord, nous avons **fusionSort**, fonction qui applique le Tri fusion pour une T\_data donnée; La fonction **fusionsort** applique le tri fusion en utilisant lui-même, la fonction fusionner et compare\_ints.; **fusionner** est responsable pour ordonner les éléments et **compare\_ints** pour les comparer.

### 2.2. Code *fusion\_sort.c*

```
#include <string.h>
#include <stdio.h>
#include "test_utils.h"
#include "include/traces.h"

void fusionner(T_elt t [], int d, int m, int f);
void fusionsort(void *base, int d, int f, int (*compare_ints)(const void *,
const void *));
int compare_ints(const void * a, const void * b);

T_data fusionSort(T_data d, int n) {
    stats.nbComparisons++;
    T_elt *base = d.pElt;
    fusionsort(base, 0, n-1, compare_ints);
    return genData(0, base);
}

void fusionsort(void *base, int d, int f, int (*compare_ints)(const void *,
const void *)){
    if (d < f){
        int m = d+(f-d)/2;
        fusionsort(base, d, m, compare_ints);
        fusionsort(base, m+1, f, compare_ints);
        fusionner(base, d, m, f);
    }
}

void fusionner(T_elt t [], int d, int m, int f) {
```

---

```

T_elt aux[f - d + 1];
    int i, j, k;
    memcpy(aux, &t[d], (f - d + 1) * sizeof(T_elt));
    stats.nbOperations += (f - d + 1);

    i = 0; j = m - d + 1; k = 0;
    while (i <= m - d && j <= f - d) {
        stats.nbComparisons+=2;
        stats.nbOperations++;
        if (aux[i] <= aux[j]) {
            t[d + k++] = aux[i++];
        }
        else {
            t[d + k++] = aux[j++];
        }
    }
    stats.nbOperations += (m - d - i > 0) ? m - d - i : 0;
    for (; i <= m - d; t[d + k++] = aux[i++]);
    stats.nbOperations += (m - d - i > 0) ? m - d - i : 0;
    for (; j <= f - d; t[d + k++] = aux[j++]);
}

int compare_ints(const void* a, const void* b){
    int arg1 = *(const int*)a;
    int arg2 = *(const int*)b;
    if (arg1 < arg2) return -1;
    if (arg1 > arg2) return 1;
    return 0;
}

```

## 2.3. Resultats

Les résultats obtenus (en utilisant le même main.c que celui de l'ex6) étaient :

TRI FUSION				
Taille	Mode	Nb compar.	Nb opér.	Duree (ms)
2500	ordonne	29505	43156	0
5000	ordonne	64009	93812	1
7500	ordonne	98865	146240	1
10000	ordonne	138017	202624	1
12500	ordonne	175977	259104	2
15000	ordonne	212729	314980	2
17500	ordonne	251865	373164	2
20000	ordonne	296033	435248	3
22500	ordonne	337625	496044	4
25000	ordonne	376953	555708	4

TRI FUSION				
Taille	Mode	Nb compar.	Nb opér.	Duree (ms)
2500	aleatoire	50187	54065	1
5000	aleatoire	110419	118137	1
7500	aleatoire	174377	185325	2
10000	aleatoire	241055	256268	2
12500	aleatoire	309241	328436	3
15000	aleatoire	378511	400543	4
17500	aleatoire	449289	475031	4
20000	aleatoire	521921	552576	5
22500	aleatoire	594547	629726	6
25000	aleatoire	668107	706803	7

TRI FUSION				
Taille	Mode	Nb compar.	Nb opér.	Duree (ms)
2500	inverse	27305	54309	0
5000	inverse	59609	118617	1
7500	inverse	94753	186117	1
10000	inverse	129217	257233	2
12500	inverse	166257	329733	2
15000	inverse	204505	402233	2
17500	inverse	242601	476965	3
20000	inverse	278433	554465	3
22500	inverse	316841	631965	3
25000	inverse	357513	709465	4

Figura 1: Résultat Tri Fusion dans un Tableau

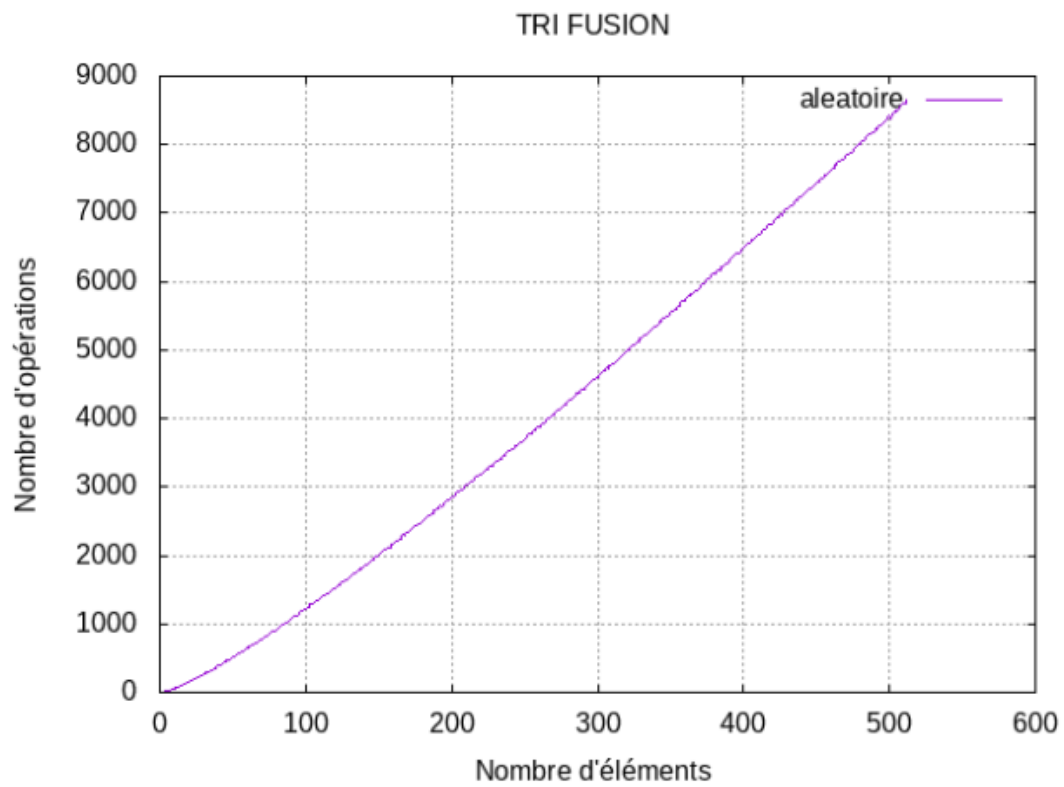


Figura 2: Graphique Tri Fusion dans un Tableau

Avec ces résultats, nous pouvons dire que la Tri Fusion est très efficace et rapide et que sa complexité est  $O(n)$ .



---

## 3. Tri Rapide

Pour l'implémentation de l'algorithme Tri Rapide, le fichier **ex7** de la semaine 3 du cours AAP a été utilisé.

### 3.1. Développement

Les fonctions suivantes ont été implémentées dans le fichier `quick_sort.c` :

- *int Partitionnement (T\_elt t [], int g, int d)*
- *void quick\_sort(T\_elt \*tab, int d, int f)*
- *int comparer(T\_elt e1, T\_elt e2)*
- *void echanger(T\_elt t[], int i1, int i2)*
- *T\_data quickSort(T\_data t, int n)*

Le fonction **Partitionnement** est responsable pour positionner a gauche toute les valeur inférieur a le pivot et a droit, les supérieur ; **quick\_sort** est la fonction qui applique l'algorithme de Tri Rapide en utilisant lui même et la fonction Partitionnement ; **quickSort** est la fonction qui applique le quick\_sort pour une T\_data donnée ;

### 3.2. Code *quick\_sort.c*

```
#include <stdio.h>
#include "test_utils.h"
#include "include/traces.h"

// TODO : placer les compteurs aux endroits appropriés :
// stats.nbOperations ++;
// stats.nbComparisons ++;

int Partitionnement (T_elt t [], int g, int d);
void quick_sort(T_elt *tab, int d, int f);
int comparer(T_elt e1, T_elt e2);
void echanger(T_elt t[], int i1, int i2);

int comparer(T_elt e1, T_elt e2){
    stats.nbComparisons++;
    return e1-e2;
}
void echanger(T_elt t[], int i1, int i2){
    T_elt aux = t[i1];
    t[i1] = t[i2];
    t[i2] = aux;
}
```

---

```

void quick_sort(T_elt *tab , int d, int f){
    int iPivot;
    //int *tab=t.pElt;
    echanger(tab , d + rand() % (f - d), f);
    stats.nbComparisons ++;
    if (f > d) {
        iPivot = Partitionnement(tab , d, f);
        quick_sort(tab , d, iPivot - 1);
        quick_sort(tab , iPivot+1, f);
    }
}

T_data quickSort(T_data t, int n) {
    int *tab=t.pElt;
    quick_sort(tab , 0, n-1);
    return genData(0, tab);
}

int Partitionnement (T_elt t [], int g, int d){
    int pg=g , pd=d-1;
    while (pg<pd) {
        while ( (pg<pd) && (comparer(t[pg],t[d]) <=0) ) {
            pg++ ;
            stats.nbComparisons ++;}
        while ( (pg<pd) && (comparer(t[pd],t[d])>0) ) {
            pd-- ;
            stats.nbComparisons ++;}

        if (pg < pd) {
            echanger(t ,pg ,pd);
            pg++ ; pd-- ;
        }
    }
    stats.nbOperations++;
    if (comparer(t[pg],t[d]) <= 0) pg++ ;

    stats.nbOperations=3+stats.nbOperations;
    echanger(t , pg , d) ;

    return pg ;
}

```

### 3.3. Resultats

Nous avons obtenu les résultats suivants :

QUICK SORT					
Taille	Mode	Nb compar.	Nb opér.	Duree (ms)	
2500	ordonne	6250000	9996	29	
5000	ordonne	25000000	19996	106	
7500	ordonne	56250000	29996	212	
10000	ordonne	100000000	39996	351	
12500	ordonne	156250000	49996	466	
15000	ordonne	225000000	59996	647	
17500	ordonne	306250000	69996	806	
20000	ordonne	400000000	79996	1072	
22500	ordonne	506250000	89996	1431	
25000	ordonne	625000000	99996	1715	

QUICK SORT					
Taille	Mode	Nb compar.	Nb opér.	Duree (ms)	
2500	aleatoire	55883	6692	0	
5000	aleatoire	120433	13296	1	
7500	aleatoire	204844	20012	2	
10000	aleatoire	258928	26624	2	
12500	aleatoire	345798	33488	2	
15000	aleatoire	451914	39988	3	
17500	aleatoire	510770	46648	4	
20000	aleatoire	579139	53364	4	
22500	aleatoire	640566	59940	4	
25000	aleatoire	755794	66596	5	

QUICK SORT					
Taille	Mode	Nb compar.	Nb opér.	Duree (ms)	
2500	inverse	6251249	9996	16	
5000	inverse	25002499	19996	73	
7500	inverse	56253749	29996	163	
10000	inverse	100004999	39996	278	
12500	inverse	156256249	49996	424	
15000	inverse	225007499	59996	629	
17500	inverse	306258749	69996	936	
20000	inverse	400009999	79996	1120	
22500	inverse	506261249	89996	1396	
25000	inverse	625012499	99996	1726	

Figura 3: Résultat Tri Rapide dans un Tableau

Avec ces résultats, nous pouvons dire aussi que la Tri Rapide est très efficace et rapide et que sa complexité est  $O(n \log n)$ .

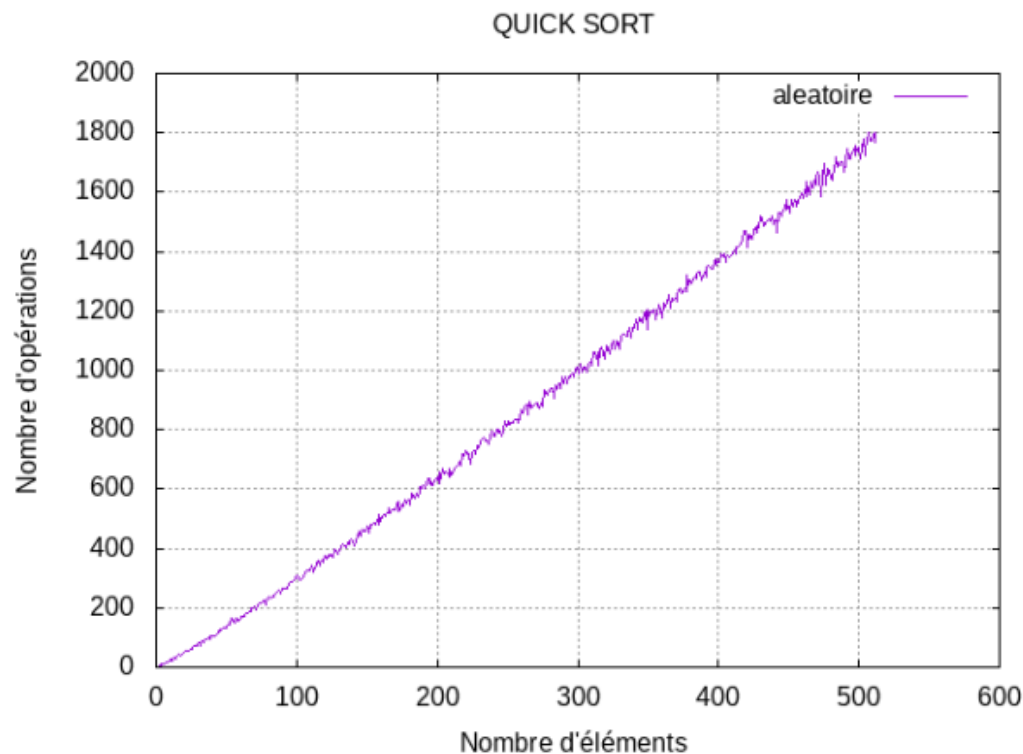


Figura 4: Graphique Tri Rapide dans un Tableau

## 4. Tri Fusion de Listes

Pour cette partie, nous avons utilisé non seulement les codes développés au cours de la troisième semaine, mais aussi les fichiers impliquant la manipulation de listes développés au cours de la deuxième semaine du cours AAP.

### 4.1. Développement

- *T\_list getNode(T\_list l, int i)*
- *void fusionnerList(T\_list l, int d, int m, int f)*
- *T\_list fusionSortList(T\_list l, int n)*

Nous avons **getNode** qui est responsable pour retourner un pointeur pour le i-ème valeur de la liste ; **fusionnerList** qui est responsable pour comparé les élément de 2 listes et les ordonné ; et aussi **fusionSortList** qui est la fonction qui applique le algorithme de Tri Fusion en utilisant lui même et la fonction fusionner ;

### 4.2. Code *main.c*

```
#include <stdio.h>
#include <stdlib.h>
```

---

```

#include "include/traces.h"

#include "test_utils.h"
#include "list.h"
#include "elt.h"
// #include "fusion_sort.c"

// Ajouter ici les prototypes des fonctions      tester
T_list fusionSortList(T_list l, int n);

T_mode m[] = {
    {MODE_TAB_ORDONNE, "ordonne", 0, 1},
    {MODE_TAB_ALEATOIRE, "aleatoire", 0, 1},
    {MODE_TAB_INVERSE, "inverse", 0, 1},
    {MODE_EVAL_X, "x=2.0", 2.0, 0},
    {MODE_TAB_ORDONNE, "ordonne_(x=59)", 59, 0},
    {MODE_TAB_ORDONNE, "hanoi", 1, 0}
};

// mode, label, x, checkOrder

int main(int argc, char *argv[]){
    T_list l = NULL;

    l = addNode(10, 1);
    l = addNode(14, 1);
    l = addNode(9, 1);
    l = addNode(21, 1);
    l = addNode(28, 1);
    l = addNode(91, 1);
    l = addNode(9, 1);
    l = addNode(4, 1);
    l = addNode(551, 1);
    l = addNode(9, 1);

    printf("Avant_sort:_");
    showList(l); NL();

    l = fusionSortList(l, getSizeIte(l));

    printf("Après_sort:_");
    showList(l); NL();
    return 0;
}

```

### 4.3. Code *fusion\_sort\_list.c*

```

#include <string.h>

```

---

```

#include <stdio.h>
#include "test_utils.h"
#include "list.h"

#include "include/traces.h"

// TODO : placer les compteurs aux endroits appropriés :
// stats.nbOperations ++;
// stats.nbComparisons ++;

void fusionnerList(T_list l, int d, int m, int f);
T_list getNode(T_list l, int j);

T_list fusionSortList(T_list l, int n) {
    //n=MAX_ELT;
    if (n==1) return l;
    int m = n/2;
    fusionSortList(l, m);
    fusionSortList(getNode(l,m), (n-m));
    fusionnerList(l, 0, m-1, n-1);
    return l;
}

T_list getNode(T_list l, int pos) {
    int i=0;
    T_list aux = l;
    while (i < pos) {
        aux = aux->pNext;
        i++;
    }
    return aux;
}

void fusionnerList(T_list l, int d, int m, int f) {
    T_elt aux[f - d + 1];
    int i, j, k;
    i = 0; j = m - d + 1; k = 0;
    T_list list_debut = getNode(l, d);
    T_list listAux = list_debut;
    for (j=d; j<=f; j++){
        aux[k++] = listAux->data;
        listAux = listAux->pNext;
    }
    stats.nbOperations += (f - d + 1);

    i = 0; j = m - d + 1; k = 0;
    while (i <= m - d && j <= f - d) {
        if (aux[i] <= aux[j] ) {

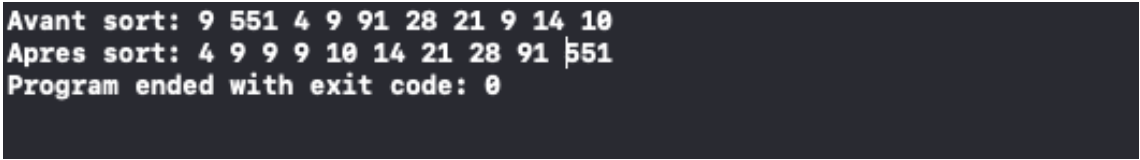
```

---

```
        getNode(list_debut , k++)->data = aux[i++];
    }
    else {
        getNode(list_debut , k++)->data = aux[j++];
    }
}
for (; i <= m - d; getNode(list_debut , k++)->data = aux[i++]);
for (; j <= f - d; getNode(list_debut , k++)->data = aux[j++]);
}
```

#### 4.4. Resultats

Avec l'utilisation du main.c décrit ci-dessus, il est alors possible de vérifier le bon fonctionnement du code développé.



```
Avant sort: 9 551 4 9 91 28 21 9 14 10
Apres sort: 4 9 9 9 10 14 21 28 91 551
Program ended with exit code: 0
```

Figura 5: Results Tri Fusion dans une Liste

---

## 5. Conclusion

A partir des résultats obtenus dans les graphiques et les sorties de chaque méthode Tri, il a été possible d'observer et de comparer leurs efficacités et leurs complexités. En particulier, il est possible de comparer les complexités obtenues avec la fonction qsort qui pour les cas moyens a la même complexité que le Tri Rapide implémenté dans ce TEA.



---

## 6. Références

- AAP - Séance 2
- AAP - Séance 3