ÉCOLE CENTRALE DE LILLE
Algorithmique Avancée et Programmation

# TEA AAP - TP3 2021

## Complexité

Bruno SOARES ZIMMER

Lille, Haut-de-France
2021

# INTRODUCTION

L'étude de l'ordonnancement des vecteurs est un sujet largement étudié par l'industrie informatique dans le monde entier. Une optimisation du code est de plus en plus recherchée afin d'obtenir un code plus rapide et consommant moins de ressources (1).

L'objectif de ce travail est l'application de deux formes d'ordonnancement des vecteurs, à savoir "Tri Fusion" et "Tri Rapide", ainsi que certaines améliorations de ces codes. Enfin, une analyse entre les méthodes de tri sera effectuée afin de conclure quels sont les avantages de cette méthode de tri.

# 1 Tri Fusion

Cette application va utiliser des vecteurs de grande taille pour vérifier la vitesse et les conditions du programme. Le programme est divisée en trois fonctions:

1) FusionSort: Fonction pour appeler la recherche pour fusion fait sur le tri_fusion avec la liste d'éléments d.

```
7   T_data fusionSort(T_data d, int n) {
8   
9     T_elt * A = d.pElt;
10    tri_fusion(A, 0, n - 1);
11  
12    return d;
13  }
```

2) tri_fusion: Fonction développée en cours avec le professeur pour faire la recherche des données souhaitées de la division au centre de chaque parti déjà divisée.

```
15  void tri_fusion(T_elt t [], int debut, int fin) {
16    int milieu;
17  
18    stats.nbComparisons++;
19    if (debut < fin)
20    {
21      milieu = (debut + fin)/2;
22      tri_fusion(t, debut, milieu);
23      tri_fusion(t, milieu + 1, fin);
24      fusionner(t, debut, milieu, fin);
25    }
26  }
```

3) fusionner: Fonction fondamental pour le fusion sort, déjà étudiée en cours responsable pour faire les comparaisons et la décision d'où faire la division des nombres, on a fait les stats pour faire la somme a chaque opérations et comparisions.

```
29  void fusionner(T_elt t [], int d, int m, int f) {
30    T_elt aux[f - d + 1]; // !! Allocation dynamique sur la pile (standard C99)
31    int i, j, k;
32  
33    memcpy(aux, &t[d], (f - d + 1) * sizeof(T_elt));    // Copie des données à fusionner
34    stats.nbOperations += (f - d + 1);
35  
36    i = 0; j = m - d + 1; k = 0;
37    while (i <= m - d && j <= f - d) {
38      stats.nbComparisons+=2;
39      stats.nbOperations++;
40        if (aux[i] <= aux[j]) {
41          t[d + k++] = aux[i++]; // aux[i] est plus petit : on le place dans t
42        }
43        else {
44          t[d + k++] = aux[j++]; // aux[j] est plus petit : on le place dans t
45        }
46    }
47    stats.nbOperations += (m - d - i > 0) ? m - d - i : 0;
48    for (; i <= m - d; t[d + k++] = aux[i++]); // le reste du tableau gauche
49    stats.nbOperations += (f - d - j > 0) ? f - d - j : 0;
50    for (; j <= f - d; t[d + k++] = aux[j++]); // le reste du tableau droit
51  }
```

Ci-dessous on peut voir les résultats utilisant les modes ordonne, aléatoire et inverse:

| TRI FUSION | | | | |
|---|---|---|---|---|
| Taille | Mode | Nb compar. | Nb opér. | Duree (ms) |
| 25000 | ordonne | 426951 | 709465 | 3 |
| 50000 | ordonne | 903903 | 1518929 | 7 |
| 75000 | ordonne | 1398631 | 2362857 | 11 |
| 100000 | ordonne | 1907807 | 3237857 | 11 |
| 125000 | ordonne | 2391607 | 4112857 | 14 |
| 150000 | ordonne | 2947263 | 5025713 | 13 |
| 175000 | ordonne | 3500167 | 5950713 | 16 |
| 200000 | ordonne | 4015615 | 6875713 | 17 |
| 225000 | ordonne | 4542119 | 7800713 | 20 |
| 250000 | ordonne | 5033215 | 8725713 | 22 |

| TRI FUSION | | | | |
|---|---|---|---|---|
| Taille | Mode | Nb compar. | Nb opér. | Duree (ms) |
| 25000 | aleatoire | 718261 | 709465 | 4 |
| 50000 | aleatoire | 1536799 | 1518929 | 9 |
| 75000 | aleatoire | 2391849 | 2362857 | 13 |
| 100000 | aleatoire | 3272285 | 3237857 | 17 |
| 125000 | aleatoire | 4167791 | 4112857 | 22 |
| 150000 | aleatoire | 5084369 | 5025713 | 27 |
| 175000 | aleatoire | 6010947 | 5950713 | 32 |
| 200000 | aleatoire | 6946101 | 6875713 | 34 |
| 225000 | aleatoire | 7888465 | 7800713 | 40 |
| 250000 | aleatoire | 8838135 | 8725713 | 44 |

| TRI FUSION | | | | |
|---|---|---|---|---|
| Taille | Mode | Nb compar. | Nb opér. | Duree (ms) |
| 25000 | inverse | 407511 | 709465 | 2 |
| 50000 | inverse | 865023 | 1518929 | 4 |
| 75000 | inverse | 1339223 | 2362857 | 6 |
| 100000 | inverse | 1830047 | 3237857 | 9 |
| 125000 | inverse | 2346247 | 4112857 | 13 |
| 150000 | inverse | 2828447 | 5025713 | 15 |
| 175000 | inverse | 3325543 | 5950713 | 19 |
| 200000 | inverse | 3860095 | 6875713 | 22 |
| 225000 | inverse | 4383591 | 7800713 | 20 |
| 250000 | inverse | 4942495 | 8725713 | 24 |

Avec cet analyse, il est possible de conclure que cette méthode de tri est extrêmement rapide, permettant de résoudre des vecteurs de grande taille en quelques millisecondes.

## 2 Tri Rapide

Cette application va utiliser des vecteurs de grande taille pour vérifier la vitesse et les conditions du programme, comme le dernier mais avec un approche un peu différente du précédent. Le programme est divisée en trois fonctions:

1) quickSort: Fonction pour appeler la recherche pour fusion fait sur le Tri_rapide avec la liste d'éléments d.

```
T_data quickSort(T_data d, int n) {
  T_elt * A = d.pElt;

  Tri_rapide(A, 0, n - 1);

  return d;
}
```

2) Tri_rapide: Fonction développée en cours avec le professeur pour faire la recherche des données souhaitées.

```
23  void Tri_rapide( T_elt t[], int debut, int fin) {
24    int iPivot;
25
26    stats.nbComparisons++;
27    if (fin > debut) {
28      //partie pour ameliorer
29      // int random = debut + rand() % (fin - debut);
30      // echanger(t, random, fin);
31
32      //division au millieu pour rechercher
33      iPivot = Partitionner(t, debut, fin);
34      Tri_rapide(t, debut, iPivot - 1);
35      Tri_rapide(t, iPivot + 1, fin);
36    }
37  }
```

3) Partitionner: Fonction fondamental pour le quick sort, déjà étudiée en cours responsable pour faire les comparaisons et la décision d'où faire la partitions des nombres, on a fait les stats pour faire la somme a chaque opérations et comparisons.

```
39  int Partitionner (T_elt t [], int d, int f){
40    int i=d , j=f-1; // On utilise i et j comme « pointeurs » qui se déplacent
41
42    while (i<j) {
43      stats.nbComparisons++;
44      // On déplace i et j jusqu'à trouver des valeurs incohérentes % pivot
45      while ((i<j) && (t[i]<=t[f])) {
46        stats.nbComparisons++;
47        i++ ;
48      }
49      while ((i<j) && (t[j]>t[f])) {
50        stats.nbComparisons++;
51        j-- ;
52      }
53
54      if (i < j) {
55        stats.nbOperations++;
56        echanger(t,i,j);
57        i++ ; j-- ;
58      }
59    }
60    stats.nbComparisons++;
61    if (t[i]<=t[f]) i++ ; // Cf. ci-contre
62
63    stats.nbOperations+=3;
64    echanger(t, i, f) ;
65
66    return i;
```

Ci-dessous on peut voir les résultats utilisant les modes ordonne, aléatoire et inverse:

| QUICK SORT | | | | |
|---|---|---|---|---|
| Taille | Mode | Nb compar. | Nb opér. | Duree (ms) |
| 25000 | ordonne | 442745 | 47310 | 6 |
| 50000 | ordonne | 988209 | 94689 | 13 |
| 75000 | ordonne | 1530632 | 142368 | 14 |
| 100000 | ordonne | 2142235 | 189741 | 11 |
| 125000 | ordonne | 2845814 | 237081 | 13 |
| 150000 | ordonne | 3379475 | 284706 | 14 |
| 175000 | ordonne | 4130564 | 332139 | 16 |
| 200000 | ordonne | 4438951 | 379413 | 18 |
| 225000 | ordonne | 5466667 | 426663 | 22 |
| 250000 | ordonne | 5733095 | 474255 | 22 |

| QUICK SORT | | | | |
|---|---|---|---|---|
| Taille | Mode | Nb compar. | Nb opér. | Duree (ms) |
| 25000 | aleatoire | 405591 | 120174 | 4 |
| 50000 | aleatoire | 880658 | 250972 | 8 |
| 75000 | aleatoire | 1427808 | 383008 | 14 |
| 100000 | aleatoire | 1789716 | 528490 | 18 |
| 125000 | aleatoire | 2308840 | 668660 | 22 |
| 150000 | aleatoire | 2885611 | 807954 | 27 |
| 175000 | aleatoire | 3327857 | 954213 | 31 |
| 200000 | aleatoire | 4066494 | 1089399 | 37 |
| 225000 | aleatoire | 4348620 | 1248311 | 42 |
| 250000 | aleatoire | 5167471 | 1384422 | 47 |

| QUICK SORT | | | | |
|---|---|---|---|---|
| Taille | Mode | Nb compar. | Nb opér. | Duree (ms) |
| 25000 | inverse | 472533 | 59906 | 2 |
| 50000 | inverse | 1039633 | 119922 | 4 |
| 75000 | inverse | 1559280 | 179526 | 6 |
| 100000 | inverse | 2080831 | 239337 | 10 |
| 125000 | inverse | 2708368 | 299532 | 11 |
| 150000 | inverse | 3289590 | 359525 | 14 |
| 175000 | inverse | 3926181 | 419641 | 16 |
| 200000 | inverse | 4512959 | 479076 | 18 |
| 225000 | inverse | 5455379 | 538931 | 21 |
| 250000 | inverse | 5934474 | 599033 | 22 |

## 2.2 Amélioration

Cette méthode de tri est extrêmement longue par rapport aux autres méthodes. Cela est dû au fait que le pivot n'est pas une valeur arbitraire, mais une valeur choisie à l'avance. Pour cette raison, nous irons résoudre ce problème en changeant le Pivot à tout moment.

```
23  void Tri_rapide( T_elt t[], int debut, int fin) {
24      int iPivot;
25
26      stats.nbComparisons++;
27      if (fin > debut) {
28          //partie pour ameliorer
29          int random = debut + rand() % (fin - debut);
30          echanger(t, random, fin);
31
32          //division au millieu pour rechercher
33          iPivot = Partitionner(t, debut, fin);
34          Tri_rapide(t, debut, iPivot - 1);
35          Tri_rapide(t, iPivot + 1, fin);
36      }
37  }
```

## 3 Tri Fusion de Listes

Ce sujet vise à développer une liste chaînée et dans celle-ci à faire d'un tri utilisé une des méthodes déjà développées. On va utiliser les fonctions qu'ils ont déjà été développées dans les cours précédents. On peut voir maintenant les fonctions développées:

1) scrollVector: Pour l'optimisation du code, la scrollVector a été développée pour faire défiler la liste jusqu'à l'élément i(eme), et finalement de retourner l'élément en question:

```
43  T_list scrollVector(T_list l, int i) {
44      int j;
45      T_list aux = l;
46
47      for (j = 0; j < i; j++) {
48          aux = aux->pNext;
49      }
50
51      return aux;
52  }
```

2) fusionnerListe: La même fonction du exercise nombre un mais avec les adaptations pour utiliser la liste:

```
64  void fusionnerListe(T_list l, int d, int m, int f) {
65      T_elt aux[f - d + 1]; // !! Allocation dynamique sur la pile (standard C99)
66      int i, j, k;
67
68      T_list listDebut = scrollVector(l, d);
69
70      k = 0;
71      T_list aux2 = listDebut;
72
73      stats.nbOperations += (j <= f) ? f-d : 0;
74      for(j=d; j<=f; j++){
75          aux[k++] = aux2->data;
76          aux2 = aux2->pNext;
77      }
78
79      i = 0; j = m - d + 1; k = 0;
80      while (i <= m - d && j <= f - d) {
81          stats.nbComparisons+=2;
82          stats.nbOperations++;
83          if (aux[i] <= aux[j]){
84              scrollVector(listDebut, k++)->data = aux[i++]; // aux[i] est plus petit : on le place dans t
85          }
86          else {
87              scrollVector(listDebut, k++)->data = aux[j++]; // aux[j] est plus petit : on le place dans t
88          }
89      }
90    stats.nbOperations += (m - d - i > 0) ? m - d - i : 0;
91      for (; i <= m - d; scrollVector(listDebut, k++)->data = aux[i++]); // le reste du tableau gauche
92    stats.nbOperations += (f - d - j > 0) ? f - d - j : 0;
93      for (; j <= f - d; scrollVector(listDebut, k++)->data = aux[j++]); // le reste du tableau droit
94  }
```

3) tri_fusionListe: Pareil à fusionnerListe, même modifications juste pour avoir le programme avec la  liste:

```
96  void tri_fusionListe(T_elt t [], int debut, int fin) {
97      int milieu;
98
99      stats.nbComparisons++;
100     if (debut < fin)
101     {
102         milieu = (debut + fin)/2;
103         tri_fusion(t, debut, milieu);
104         tri_fusion(t, milieu + 1, fin);
105         fusionnerListe(t, debut, milieu, fin);
106     }
107 }
```

4) Main test: Sur la main du programme de tri Fusion on va utiliser les fonctions déjà développées sur le TEA 2 et pendant les cours 2, pour faire pareil mais avec les listes et les fonctions modifiées pour la liste:

```
12  int main() {
13      T_list test = NULL;
14  //créer la liste
15      test = addNode(10, test);
16      test = addNode(20, test);
17      test = addNode(15, test);
18      test = addNode(10, test);
19      test = addNode(2, test);
20      test = addNode(3, test);
21      test = addNode(5, test);
22      test = addNode(30, test);
23      test = addNode(20, test);
24      test = addNode(100, test);
25      test = addNode(35, test);
26      test = addNode(5, test);
27      test = addNode(0, test);
28      test = addNode(100000, test);
29      NL();
30  //verifier si c'est correct par le print
31      showList(test);
32      NL(); NL();
33  //faire la organisation
34      tri_fusion_liste(test, 0, getSize(test) - 1);
35      showList(test);
36      NL(); NL();
37
38      return 0;
39  }
```

# CONCLUSION

Compte tenu de toutes ces applications et de toutes les comparaisons, il est possible de conclure à l'importance de cette étude pour le secteur des TI. La recherche d'optimisation est une chose actuelle et reste un problème d'actualité, à tout moment les codes déjà développés sont retravaillés afin d'obtenir un résultat plus simple.