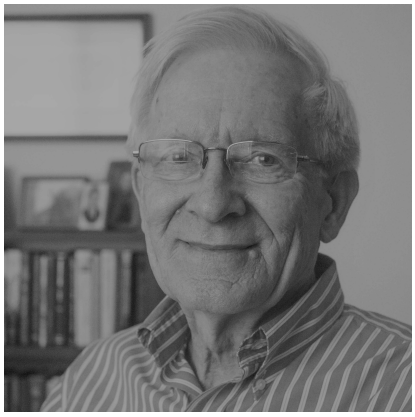
Comparaison de différents algorithmes de tri

Juanfer MERCIER

03 Décembre 2019



David "Dave" Musser est un professeur d'informatique retraité qui a enseigné à l'Institut Rensselaer Polytechnic à Troy, New York. Il est connu pour avoir développé un algorithme de tri appelé "**Introsort**" en 1997 pour la *Standard Template Library* (STL).

Sommaire

1 Quelques petits rappels

1.1 Les algorithmes de tri

> **Selon Wikipedia...**
>
> "Un algorithme de tri est, en informatique ou en mathématiques, un algorithme qui permet d'organiser une collection d'objets selon une relation d'ordre déterminée."

Ainsi, un algorithme de tri permet d'ordonner un ensemble de données selon une relation d'ordre choisi au préalable (Exemple : on peut trier des entiers par ordre décroissant). Il existe de nombreux algorithmes fonctionnels qui permettent d'ordonner un ensemble mais il semble naturel de choisir ceux qui sont les plus performants lorsque l'on fait face à certaines contraintes (très grand nombre de données, utilisation des ressources, ...).

Dans un premier temps, avant de comparer les algorithmes, il faudrait que l'on ait des critères qui nous indique lequel est le plus adapté à la situation.

1.2 Critères de comparaison

- La **complexité temporelle** dans le pire des cas est une mesure de temps utilisé par un algorithme, exprimé comme fonction de la taille de l'entrée à l'aide des notations de Landau O ou Θ. C'est le nombre d'opérations élémentaires effectuées pour trier un ensemble de données.

- La **complexité spatiale** dans le pire des cas est une mesure de la quantité de mémoire utilisé par l'algorithme pour s'exécuter qui peut elle aussi dépendre de taille de l'entrée.

- Un tri est dit **en place** si la complexité spatiale de l'espace supplémentaire pour trier l'ensemble de données est $O(1)$ (ou encore $O(log(n))$).

- Un tri est dit **stable** s'il préserve la disposition initial des éléments que l'ordre considère comme égaux.

1.3 Lien avec les données

> **Selon Wikipedia...**
>
> "La comparaison empirique d'algorithmes n'est pas aisée dans la mesure où beaucoup de paramètres entrent en compte : taille de données, ordre des données, [...]"

L'étude théorique des algorithmes n'est pas suffisante car l'efficacité de ces-derniers dépend aussi de l'ordre des données sur lesquelles on les utilise. Les algorithmes que nous allons considérés par la suite sont les tris : par *insertion*, par *tas*, *compte*, *fusion* et *rapide*. Pour les étudier et les comparer, on se propose de créer plusieurs ensembles de données.

On ordonnera les données toujours par ordre croissant. De plus, on donnera aux données un ordre prédéfini (avant le tri). Les jeux de données seront ordonnés : par ordre croissant de 0 à n, par ordre décroissant de n à 0 (inversé), aléatoirement sans redondance de donnée, aléatoirement avec redondance, aléatoirement mais de façon uniforme sans redondance et aléatoirement de façon uniforme mais avec redondance. D'autre part, on fera varié la taille de ces jeux de données (on aura par exemple n = 10, n = 500, n = 25000).

On utilise un script Python pour générer ces jeux de données :

```
1  import random as rd
2
3  def createOrderedDataAsc(n, a = 0):
4      return [k for k in range(a, n+1)]
5
6  def createOrderedDataDes(n, a = 0):
7      return [k for k in range(n, a-1, -1)]
8
9  def createRandomData(n, a = 0):
10     data = createOrderedDataAsc(n, a)
11     rd.shuffle(data)
12     return data
13
14 def createRandomDataRed(n, a = 0):
15     return [rd.randrange(a, n) for k in range(a, n+1)]
16
17 def createUniformDataRed(n, a = 0):
18     return [int(rd.uniform(a, n)) for k in range(a, n+1)]
19
20 def createUniformData(n, a = 0):
21     dataRed = createUniformDataRed(n, a)
22     red = [0 for k in range(a, n+1)]
23     data = []
24     count = 0
25     for k in dataRed:
26         red[k] += 1
27
28     print(red)
29
30     for k in dataRed:
31         if red[k] == 1 or red[k] == 0:
32             data.append(k)
33         elif red[k] > 1:
34             data.append(k)
35             red[k] = -1
36         else:
37             count += 1
38
39     while count > 0:
40         for k in range(a, n+1):
41             if k not in dataRed:
42                 data.append(k)
43                 count -= 1
44     return data
45
46 def toStr(list):
```

```
47      data = ""
48      for e in list:
49          data += str(e) + "\n"
50      return data
51
52  def createFile(n):
53      file = open("data/oasc{}.txt".format(n), "w")
54      file.write(toStr(createOrderedDataAsc(n)))
55      file.close()
56
57      file = open("data/odes{}.txt".format(n), "w")
58      file.write(toStr(createOrderedDataDes(n)))
59      file.close()
60
61      file = open("data/rand{}.txt".format(n), "w")
62      file.write(toStr(createRandomData(n)))
63      file.close()
64
65      file = open("data/ranR{}.txt".format(n), "w")
66      file.write(toStr(createRandomDataRed(n)))
67      file.close()
68
69      file = open("data/unif{}.txt".format(n), "w")
70      file.write(toStr(createUniformData(n)))
71      file.close()
72
73      file = open("data/uniR{}.txt".format(n), "w")
74      file.write(toStr(createUniformDataRed(n)))
75      file.close()
76
77  k = 10
78  while k <= 100000:
79      createFile(k)
80      k *= 10
81
82  k = 25
83  while k <= 250000:
84      createFile(k)
85      k *= 10
86
87  k = 50
88  while k <= 50000:
89      createFile(k)
90      k *= 10
```

Ces données sont ensuite chargés dans un programme C où sont implémentés les algorithmes de tri :

```c
int main(){
    int a = 0, n = 10;
    long double t = 0;
    unsigned int * Tab;
    char filename[256] = "";

    void (*sortFct[NBFCT])(unsigned int *t,unsigned int d,unsigned int n) = {triInsertion,triFusion,triRapide,triTas,triCompte};
    char *name[NBFCT] = {"insertion", "merge", "quick", "heap", "count"};
    char *dist[NOMS] = {"oasc", "odes", "rand", "ranR", "unif", "uniR"};

    Tab = (unsigned int*)malloc_p(sizeof(unsigned int)*TAILLEMAX);

    while(n < 100000){
        for(int i = 0; i < NBFCT; i++){
            for(int j = 0; j < NOMS; j++){
                initTab(&Tab, n, dist[j]);
                t = clock();
                sortFct[i](Tab,a,n-1);
                t = (clock() - t) / CLOCKS_PER_SEC;
                sprintf(filename, "results/%s_%d.txt", name[i], n);
                saveToFile(filename, t);
            }
        }
        printf("%d", n);
        n *= 10;
    }

    n = 25;

    while(n < 250000){
        for(int i = 0; i < NBFCT; i++){
            for(int j = 0; j < NOMS; j++){
                initTab(&Tab, n, dist[j]);
                t = clock();
                sortFct[i](Tab,a,n-1);
                t = (clock() - t) / CLOCKS_PER_SEC;
                sprintf(filename, "results/%s_%d.txt", name[i], n);
                saveToFile(filename, t);
            }
        }
        printf("%d", n);
        n *= 10;
    }

    n = 50;

    while(n < 500000){
        for(int i = 0; i < NBFCT; i++){
            for(int j = 0; j < NOMS; j++){
                initTab(&Tab, n, dist[j]);
                t = clock();
                sortFct[i](Tab,a,n-1);
                t = (clock() - t) / CLOCKS_PER_SEC;
```

```
                sprintf(filename, "results/%s_%d.txt", name[i], n);
                saveToFile(filename, t);
            }
        }
        printf("%d", n);
        n *= 10;
    }

    free(Tab);
    Tab = NULL;

    return 0;
}
```

Avec les fonctions *initTab* et *saveToFile* telles que :

```
void initTab(unsigned int **t,unsigned int n, char *s){
    char filename[256] = "";
    sprintf(filename, "data/%s%d.txt", s, n);
    FILE *file = fopen_p(filename, "r"); //fopen_p est juste une version "protegee" de fopen
    printf("%p %d", file, n);
    for(int i = 0; i < (int)n && fscanf(file, "%d\n", (*t + i)) == 1; i++);
    fclose(file);
}

void saveToFile(char *s, long double t){
    FILE *file = fopen_p(s, "a"); //fopen_p est juste une version "protegee" de fopen
    fprintf(file, "%Lf\n", t);
    fclose(file);
}
```

2 Performances des tris

2.1 Traitement des données

Pour observer les performances des tris sur nos jeux de données, il nous suffit d'utiliser les résultats écrits dans les fichiers créés par le programme en C. On peut traiter les données à la main ou utiliser le script Python suivant :

```
from matplotlib import pyplot as plt

def loadData(data, a, n):
    while a <= n:
        for i in range(len(data)):
            keys = list(data.keys())
            file1 = open("results/" + keys[i] + '_' + str(a*1) + ".txt", 'r');
            file2 = open("results/" + keys[i] + '_' + str(int(a*2.5)) + ".txt", 'r');
            file3 = open("results/" + keys[i] + '_' + str(int(a*5.0)) + ".txt", 'r');

            data[keys[i]][str(a)] = file1.read().split('\n')
            data[keys[i]][str(int(a*2.5))] = file2.read().split('\n')
```

```
13              data[keys[i]][str(int(a*5.0))] = file3.read().split('\n')
14
15              file1.close()
16              file2.close()
17              file3.close()
18          a *= 10
19
20      return data
21
22  def plotData(a, data, label, filename):
23      labels = ["ordonne", "inverse", "aleatoire sans redondance", "aleatoire avec
        redondance", "uniformement aleatoire sans redonance", "uniformement aleatoire
        avec redonance"]
24
25      for i in range(a, len(labels)):
26          X, Y = [], []
27          for k in range(a, len(data)):
28              v = list((data.keys()))[k]
29              X.append(int(v))
30              Y.append(float(data[v][i]))
31          plt.plot(X, Y)
32      plt.legend(labels[a:])
33      plt.xlabel("Nombre de donnees")
34      plt.ylabel("Temps d'execution de l'algorithme (en s)")
35      plt.title(label)
36      plt.savefig(filename)
37      plt.grid()
38      plt.show()
39
40  def plotComparisons(a, b, data):
41      labels = ["ordonne", "inverse", "aleatoire sans redondance", "aleatoire avec
        redondance", "uniformement aleatoire sans redonance", "uniformement aleatoire
        avec redonance"]
42      tri = ["tri par insertion", "tri rapide", "tri compte", "tri par tas", "tri
        fusion"]
43
44      for i in range(b, len(tri)):
45          for c in range(a, len(data)):
46              keys = list((data.keys()))
47              temp = data[keys[c]]
48              X, Y = [], []
49              for k in range(len(temp)):
50                  v = list((temp.keys()))[k]
51                  X.append(int(v))
52                  Y.append(float(temp[v][i]))
53              plt.plot(X, Y)
54          plt.legend(tri[a:])
55          plt.xlabel("Nombre de donnees")
56          plt.ylabel("Temps d'execution de l'algorithme (en s)")
57          plt.title("Tris sur le jeu de donnees " + labels[i])
58          plt.savefig("plots/" + labels[i] + ".png")
59          plt.grid()
60          plt.show()
61
62  data = {"insertion" : {}, "quick" : {}, "count" : {}, "heap" : {}, "merge" : {}}
63  data = loadData(data, 10, 50000)
```

```
64
65 plotData(0, data["count"], "Tri compte", "plots/count.png")
66 plotData(0, data["heap"], "Tri par tas", "plots/heap.png")
67 plotData(0, data["insertion"], "Tri par insertion", "plots/insertion.png")
68 plotData(0, data["merge"], "Tri fusion", "plots/merge.png")
69 plotData(0, data["quick"], "Tri rapide", "plots/quick.png")
70 #plotData(0, data["introsort"], "Introsort", "plots/intro.png")
71
72 plotComparisons(0, 0, data)
```

Ce script génère plusieurs tracés, on s'intéresse surtout aux tracés suivant :

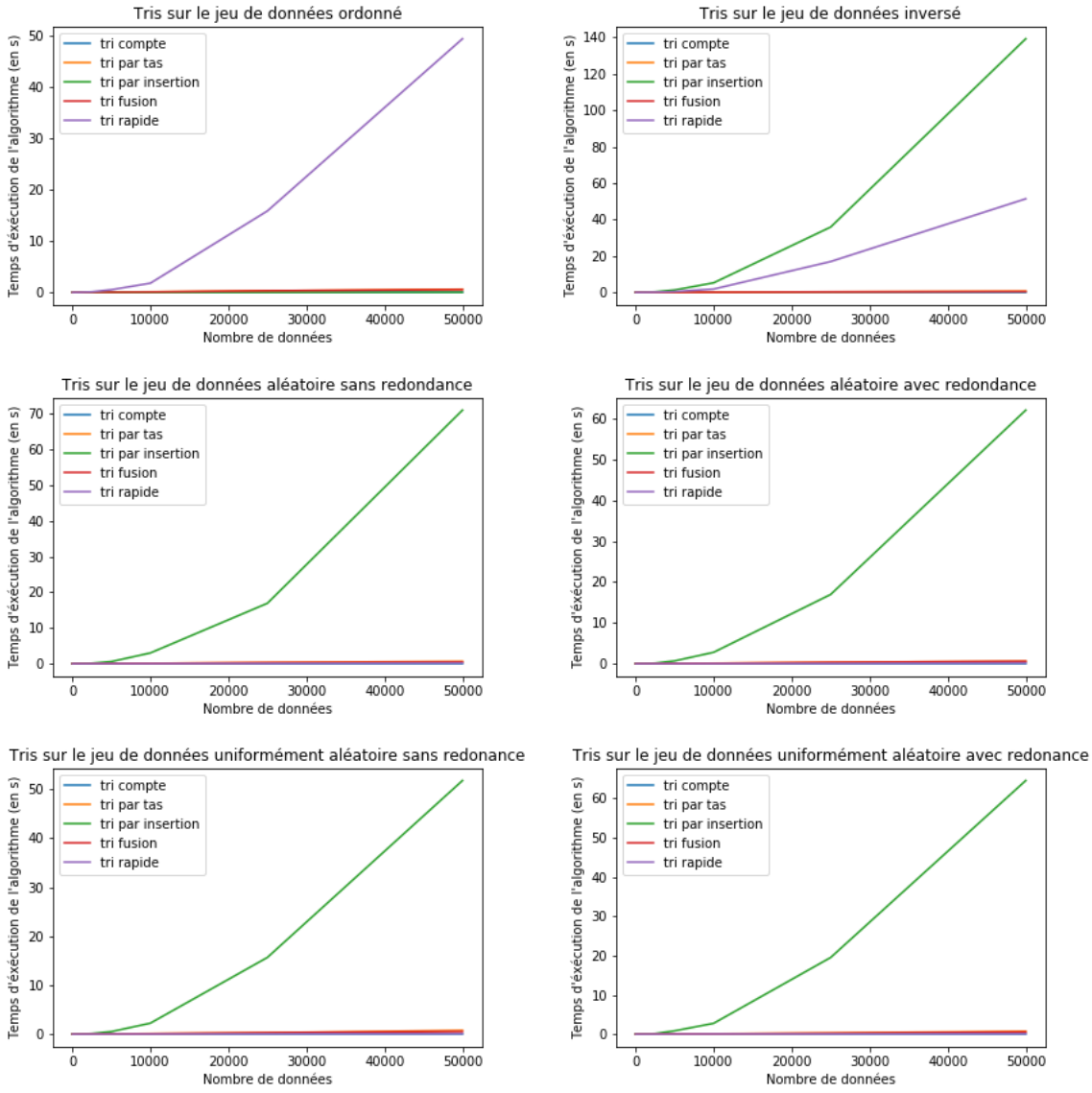


FIGURE 1 – Comparaisons des tris en sur les six jeux de données différents

Pour s'assurer d'avoir de bons résultats on devrait recréer l'expérience plusieurs fois (**N.B :** on aurait pu mesurer le temps de chaque algorithme sur chaque jeu de donnée un certain nombre de fois et ensuite faire une moyenne). On peut aussi regarder les tracés de chacun des tris pour observer dans quel cas ils sont à éviter ou à utiliser :

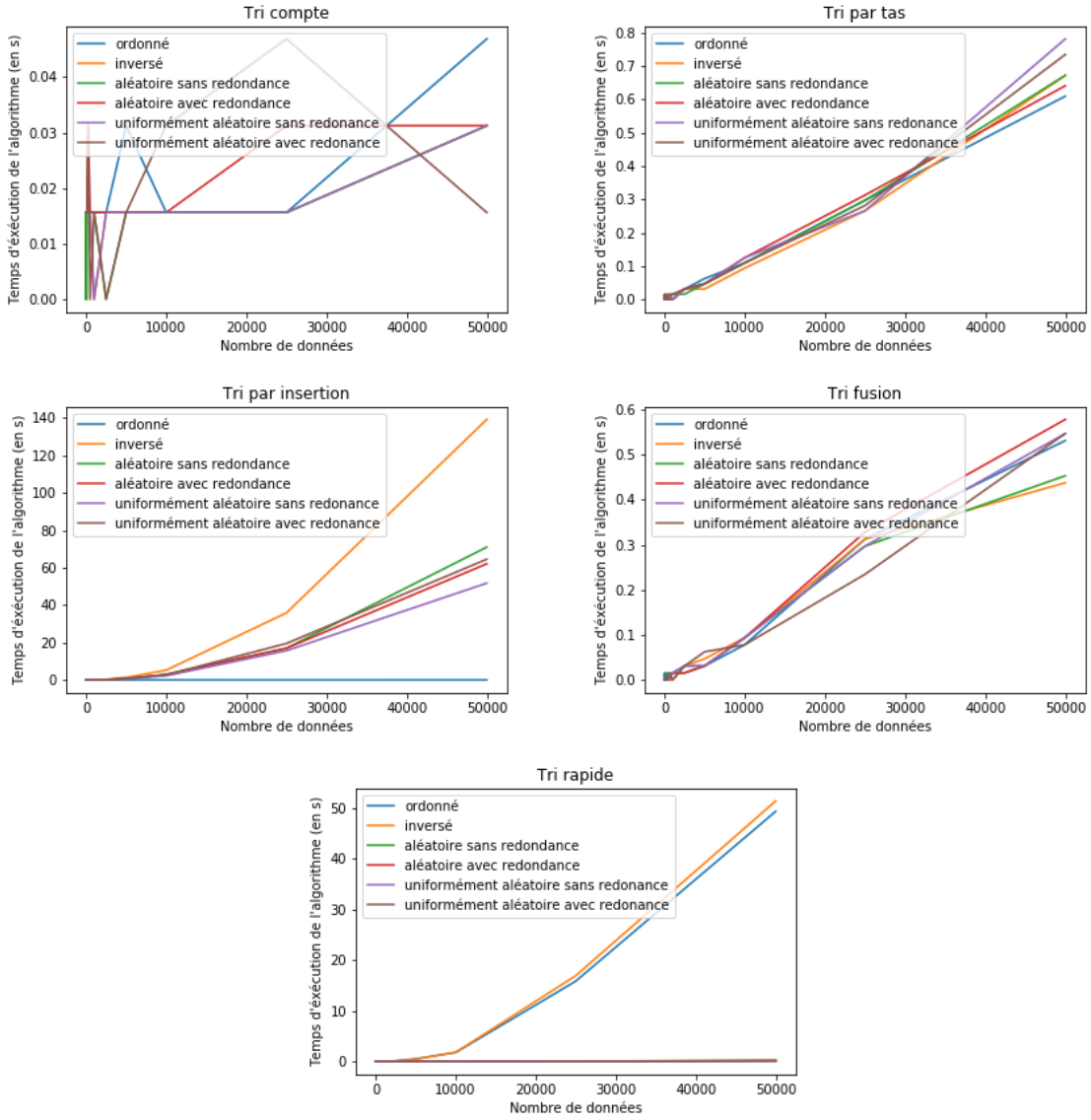


FIGURE 2 – Performances des différents tris

2.2 Complexité temporelle

On remarque grâce aux tracés de la *Figure 1* que parmi les algorithmes étudiés le moins efficace en général est le tri par insertion qui a une complexité dans le pire des cas $O(n^2)$. Cependant, celui-ci est plus rapide que tous les autres tris sur le jeu de données ordonné. C'est notamment le désavantage du tri rapide : il n'est pas efficace sur les jeux de données déjà presque triés.

Les tracés précédents ne permettent pas avec précision de comparer le tri rapide, le tri fusion, le tri compte et le tri par tas. En modifiant le script précédant, on peut obtenir des tracés plus fins en omettant les jeux de données ordonné et inversé (on met de côté le point faible du tri rapide). On a donc :

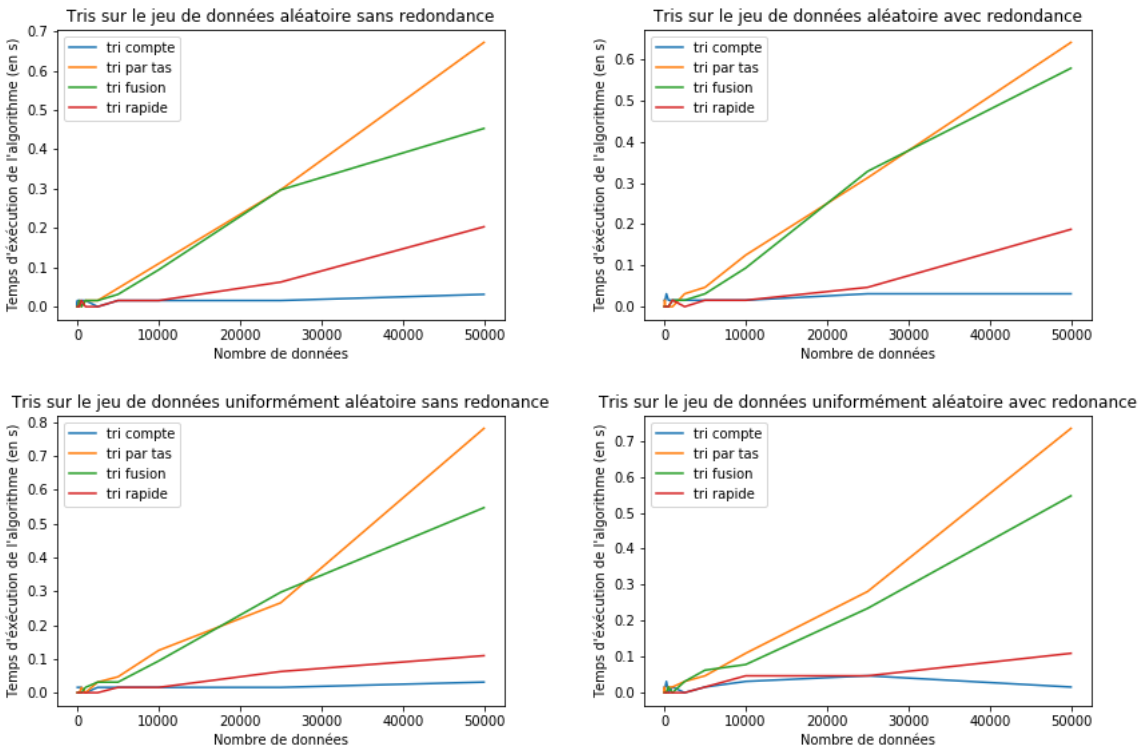


FIGURE 3 – Comparaisons des tris en sur les jeux de données non triés

On remarque que le tri compte est le plus rapide, c'est un algorithme de tri par dénombrement qui s'applique sur des valeurs entières de complexité dans le pire des cas $O(n+k)$. Il est suivi du tri rapide (qui est visiblement plus lent que le tri compte dés que le nombre de données est de 10000 ou plus). Le tri fusion et le tri par tas ont une complexité $O(nlog(n))$ bien qu'empiriquement le tri fusion est légèrement plus rapide en général.

2.3 Complexité spatiale et stabilité

Le tri par insertion et le tri par tas ont une complexité spatiale dans le pire des cas $O(n)$ et ils n'utilisent qu'une quantité constante d'espace additionnel ($O(1)$). Le tri rapide, quant à lui, a une complexité spatiale dans le pire des cas $O(log(n))$ (version Sedgewick) ou $O(n)$ (version "naïve"). Pour finir, le tri fusion a une complexité spatiale dans le pire des cas $O(n)$ et le tri compte a une complexité spatiale dans le pire des cas $O(n+r)$.

Pour résumer, les tris par insertion, par tas et rapide sont en place tandis que les tris fusion et compte ne le sont pas. De plus, le tri rapide et le tri par tas ne sont pas stable contrairement aux autres tris.

3 Limitations et algorithmes "hybrides"

3.1 L'algorithme de tri parfait

Après cette (brève) étude de ces quelques tris on remarque qu'il n'y a pas d'algorithme de tri qui serait efficace pour tous les jeux de données. Le tri par insertion fonctionne bien sur un jeu de donnée trié contrairement au tri rapide qui est moins efficace sur les jeux de données trié (par ordre croissant ou décroissant). Les tri fusion et tri par tas ont l'avantage d'avoir une complexité temporelle dans le pire des cas en $O(nlog(n))$ donc ils sont plus efficaces que le tri rapide sur des jeux de données ordonné mais empiriquement le tri rapide est toujours plus efficace sur les autres jeux (quand il est bien implémenté il peut être deux à trois fois plus rapide que le tri fusion et le tri par tas). Le tri compte (le seul tri sans comparaisons) est le plus efficace parmi les tris étudiés mais il peut difficilement être utilisé sur autre chose que des nombres.

Ainsi, il est nécessaire de penser aux inconvénients lorsqu'on choisi un algorithme de tri. Le tri rapide reste un très bon choix si l'on sait qu'on a très peu de chances de tomber sur des jeux de données ordonnés. David Musser a cependant su pallier au défaut du tri rapide (sa complexité dans le pire des cas en $O(n^2)$) en inventant l'introspective sort.

3.2 L'introspective sort

L'introspective sort est un algorithme de tri "hybride", son principe est assez simple : on utilise un compteur de récursion ! Lorsque la profondeur de récursion dépasse $Klog(n)$ (avec K une constante) on tri le sous-tableau restant avec un algorithme dont la complexité est $O(nlog(n))$ dans tous les cas (tri par tas, tri fusion, ...).

On utilise une implémentation en C++ (avec quelques modifications) de l'introsort trouvé sur le site *geeksforgeeks.org* (voir annexe), cette implémentation utilise en fait 3 autres algorithmes de tri : le tri rapide, le tri par insertion et le tri par tas. La profondeur de récursion maximale de celui-ci est de $2 * log_2(n)$ et sa complexité spatiale dans le pire des cas est $O(log(n))$. En modifiant le script Python correspondant, on obtiens des tracés (voir annexe) qui montrent que l'introsort est plus rapide que tous les autres sur tous les jeux de données.

3.3 Conclusion

Les tris compte, rapide, fusion et par tas sont très performants bien qu'ils présentent des inconvénients. Mais c'est en combinant de la bonne façon ces algorithmes que l'on conçoit des algorithmes encore plus efficaces (bien que plus complexes) qui peuvent être utilisés avec de très grosses données. Dans ce cas-là même les tris moins rapide (comme le tri par insertion) peuvent être utiles.

Annexe A Tracés de l'introsort

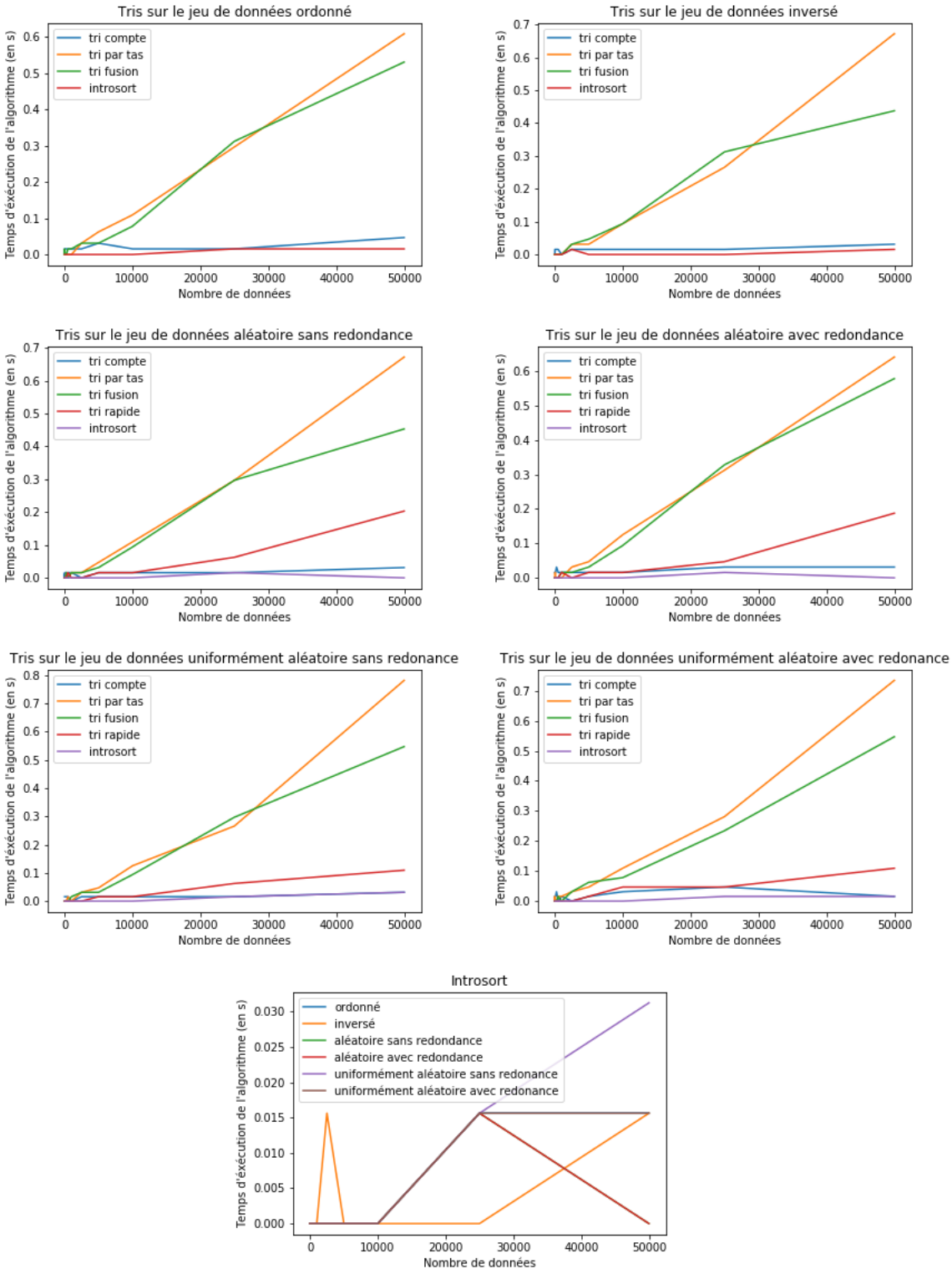


FIGURE 4 – Performances de l'introsort sur les six jeux de données différents

Annexe B Implémentation de l'introsort en C++

```
1 /* A Program to sort the array using Introsort.
2 The most popular C++ STL Algorithm- sort()
3 uses Introsort.
4
5 Source: https://www.geeksforgeeks.org/know-your-sorting-algorithm-set-2-introsort-
      cs-sorting-weapon/
6 */
7
8 #include <time.h>
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <bits/stdc++.h>
12 using namespace std;
13
14 #define TAILLEMAX 10000000
15 #define NOMS 6
16
17 void* malloc_p(unsigned int s){
18     void *p;
19     if((p = malloc(s))) return p;
20     else{perror("Erreur d'allocation"); exit(EXIT_FAILURE);}
21 }
22
23 FILE* fopen_p(const char *path, const char *mode){
24     FILE *file;
25     if((file = fopen(path, mode))) return file;
26     else{perror("Impossible de traiter le fichier"); exit(EXIT_FAILURE);}
27 }
28
29 // A utility function to swap the values pointed by
30 // the two pointers
31 void swapValue(int *a, int *b)
32 {
33     int *temp = a;
34     a = b;
35     b = temp;
36     return;
37 }
38
39 /* Function to sort an array using insertion sort*/
40 void InsertionSort(int arr[], int *begin, int *end)
41 {
42     // Get the left and the right index of the subarray
43     // to be sorted
44     int left = begin - arr;
45     int right = end - arr;
46
47     for (int i = left+1; i <= right; i++)
48     {
49         int key = arr[i];
50         int j = i-1;
51
52     /* Move elements of arr[0..i-1], that are
53         greater than key, to one position ahead
```

```
54          of their current position */
55          while (j >= left && arr[j] > key)
56          {
57              arr[j+1] = arr[j];
58              j = j-1;
59          }
60          arr[j+1] = key;
61  }
62
63  return;
64  }
65
66  // A function to partition the array and return
67  // the partition point
68  int* Partition(int arr[], int low, int high)
69  {
70      int pivot = arr[high]; // pivot
71      int i = (low - 1); // Index of smaller element
72
73      for (int j = low; j <= high- 1; j++)
74      {
75          // If current element is smaller than or
76          // equal to pivot
77          if (arr[j] <= pivot)
78          {
79              // increment index of smaller element
80              i++;
81
82              swap(arr[i], arr[j]);
83          }
84      }
85      swap(arr[i + 1], arr[high]);
86      return (arr + i + 1);
87  }
88
89
90  // A function that find the middle of the
91  // values pointed by the pointers a, b, c
92  // and return that pointer
93  int *MedianOfThree(int * a, int * b, int * c)
94  {
95      if (*a < *b && *b < *c)
96          return (b);
97
98      if (*a < *c && *c <= *b)
99          return (c);
100
101     if (*b <= *a && *a < *c)
102         return (a);
103
104     if (*b < *c && *c <= *a)
105         return (c);
106
107     if (*c <= *a && *a < *b)
108         return (a);
109
```

```
110     if (*c <= *b && *b <= *a)
111         return (b);
112 }
113
114 // A Utility function to perform intro sort
115 void IntrosortUtil(int arr[], int * begin,
116                    int * end, int depthLimit)
117 {
118     // Count the number of elements
119     int size = end - begin;
120
121     // If partition size is low then do insertion sort
122     if (size < 16)
123     {
124         InsertionSort(arr, begin, end);
125         return;
126     }
127
128     // If the depth is zero use heapsort
129     if (depthLimit == 0)
130     {
131         make_heap(begin, end+1);
132         sort_heap(begin, end+1);
133         return;
134     }
135
136     // Else use a median-of-three concept to
137     // find a good pivot
138     int * pivot = MedianOfThree(begin, begin+size/2, end);
139
140     // Swap the values pointed by the two pointers
141     swapValue(pivot, end);
142
143 // Perform Quick Sort
144     int * partitionPoint = Partition(arr, begin-arr, end-arr);
145     IntrosortUtil(arr, begin, partitionPoint-1, depthLimit - 1);
146     IntrosortUtil(arr, partitionPoint + 1, end, depthLimit - 1);
147
148     return;
149 }
150
151 /* Implementation of introsort*/
152 void Introsort(int arr[], int *begin, int *end)
153 {
154     int depthLimit = 2 * log(end-begin);
155
156     // Perform a recursive Introsort
157     IntrosortUtil(arr, begin, end, depthLimit);
158
159     return;
160 }
161
162 // A utility function ot print an array of size n
163 void printArray(int arr[], int n)
164 {
165 for (int i=0; i < n; i++)
```

```
166     printf("%d ", arr[i]);
167 printf("\n");
168 }
169
170 void initTab(int **t, int n, const char *s){
171     char filename[256] = "";
172     sprintf(filename, "data/%s%d.txt", s, n);
173     FILE *file = fopen_p(filename, "r"); //fopen_p est juste une version "protegee"
        de fopen
174     printf("%p %d", file, n);
175     for(int i = 0; i < (int)n && fscanf(file, "%d\n", (*t + i)) == 1; i++);
176     fclose(file);
177 }
178
179 void saveToFile(char *s, long double t){
180     FILE *file = fopen_p(s, "a"); //fopen_p est juste une version "protegee" de
       fopen
181     fprintf(file, "%Lf\n", t);
182     fclose(file);
183 }
184
185 // Driver program to test Introsort
186 int main()
187 {
188     int n = 10;
189     long double t = 0;
190     char filename[256] = "";
191     string dist[NOMS] = {"oasc", "odes", "rand", "ranR", "unif", "uniR"};
192     int *Tab = (int*)malloc_p(sizeof(int)*TAILLEMAX);
193
194     // Pass the array, the pointer to the first element and
195     // the pointer to the last element
196     while(n < 100000){
197         for(int j = 0; j < NOMS; j++){
198             initTab(&Tab, n, dist[j].c_str());
199             t = clock();
200             Introsort(Tab, Tab, Tab+n-1);
201             t = (clock() - t) / CLOCKS_PER_SEC;
202             sprintf(filename, "results/introsort_%d.txt", n);
203             saveToFile(filename, t);
204         }
205         printf("%d", n);
206         n *= 10;
207     }
208
209     n = 25;
210
211     while(n < 250000){
212         for(int j = 0; j < NOMS; j++){
213             initTab(&Tab, n, dist[j].c_str());
214             t = clock();
215             Introsort(Tab, Tab, Tab+n-1);
216             t = (clock() - t) / CLOCKS_PER_SEC;
217             sprintf(filename, "results/introsort_%d.txt", n);
218             saveToFile(filename, t);
219         }
```

```
220         printf("%d", n);
221         n *= 10;
222     }
223 
224     n = 50;
225 
226     while(n < 500000){
227         for(int j = 0; j < NOMS; j++){
228             initTab(&Tab, n, dist[j].c_str());
229             t = clock();
230             Introsort(Tab, Tab, Tab+n-1);
231             t = (clock() - t) / CLOCKS_PER_SEC;
232             sprintf(filename, "results/introsort_%d.txt", n);
233             saveToFile(filename, t);
234         }
235         printf("%d", n);
236         n *= 10;
237     }
238 
239     free(Tab); Tab = NULL;
240 
241     return 0;
242 }
```