Université des Sciences et Technologies Houari Boumediene

Faculté d'Informatique

Master I

Ingénierie de Logiciels

Algorithmique Avancée et Complexité

Mini Projet

Les Algorithmes de tri

ZITOUNI Aymen Abdessalam

NAIT MIHOUB Ayoub

Tables des matières

❖ Introduction

Un algorithme de tri est un algorithme qui permet d'organiser une collection d'objets selon une relation d'ordre déterminée. Les objets à trier sont des éléments d'un ensemble muni d'un ordre total. Il est par exemple fréquent de trier des entiers selon la relation d'ordre usuelle « est inférieur ou égal à ». Les algorithmes de tri sont utilisés dans de très nombreuses situations. Ils sont en particulier utiles à de nombreux algorithmes plus complexes dont certains algorithmes de recherche, comme la recherche dichotomique. Ils peuvent également servir pour mettre des données sous forme canonique ou les rendre plus lisibles pour l'utilisateur. La collection à trier est souvent donnée sous forme de tableau, afin de permettre l'accès direct aux différents éléments de la collection, ou sous forme de liste, ce qui peut se révéler être plus adapté à certains algorithmes et à l'usage de la programmation fonctionnelle.

La classification des algorithmes de tri est très importante, car elle permet de choisir l'algorithme le plus adapté au problème traité, tout en tenant compte des contraintes imposées par celui-ci. Les principales caractéristiques qui permettent de différencier les algorithmes de tri, outre leur principe de fonctionnement, sont la complexité temporelle, la complexité spatiale et le caractère stable.

Dans notre travail, nous nous intéressons à ces cinq algorithmes :

- **Tri à bulles**
- **Tri Gnome**
- **Tri par distribution**
- **Tri rapide**
- **Tri par tas**

❖ Environnement du travail:

Laptop	Dell Latitude 3380
CPU	I5-7200U 2.50 GHz
RAM	8 Go
Système	Windows 10 Pro 64 Bit

1. Tri à bulles

Le tri à bulles ou tri par propagation est un algorithme de tri. Il consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide.

- **Algorithme:**

```
Procédure du TriBulle (E/S: un tableau T[n] d'entiers ; E/n :entier)
Début
    Changement = vrai ; (variable booléenne)
    Tant que (Changement=vrai) faire
        Changement = faux ;
        pour i=1 à n-1 faire
            si (T[i] > T[i+1]) alors
                Permuter(T[i], T[i+1]) ;
                Changement = VRAI;
            Fsi ;
        Fait;
    Fait;
Fin;
```

- **Algorithme en C:**

```
void bulles(int* T, int n){
    int changement = 1;
    while(changement == 1){
        changement = 0;
        for(int i=0; i<n-1; i++){
            if(T[i] > T[i+1]){
                int temp = T[i];
                T[i] = T[i+1];
                T[i+1] = temp;
                changement = 1;
            }

        }
    }
}
```

- **Analyse et complexité**
- Meilleur cas: le tableau est déjà trié, l'algorithme de tri a bulles performera une seule itération et donc O(n).
- Pire cas: tableau décroissant, la boucle extérieure sera exécutée (n-1) fois :

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

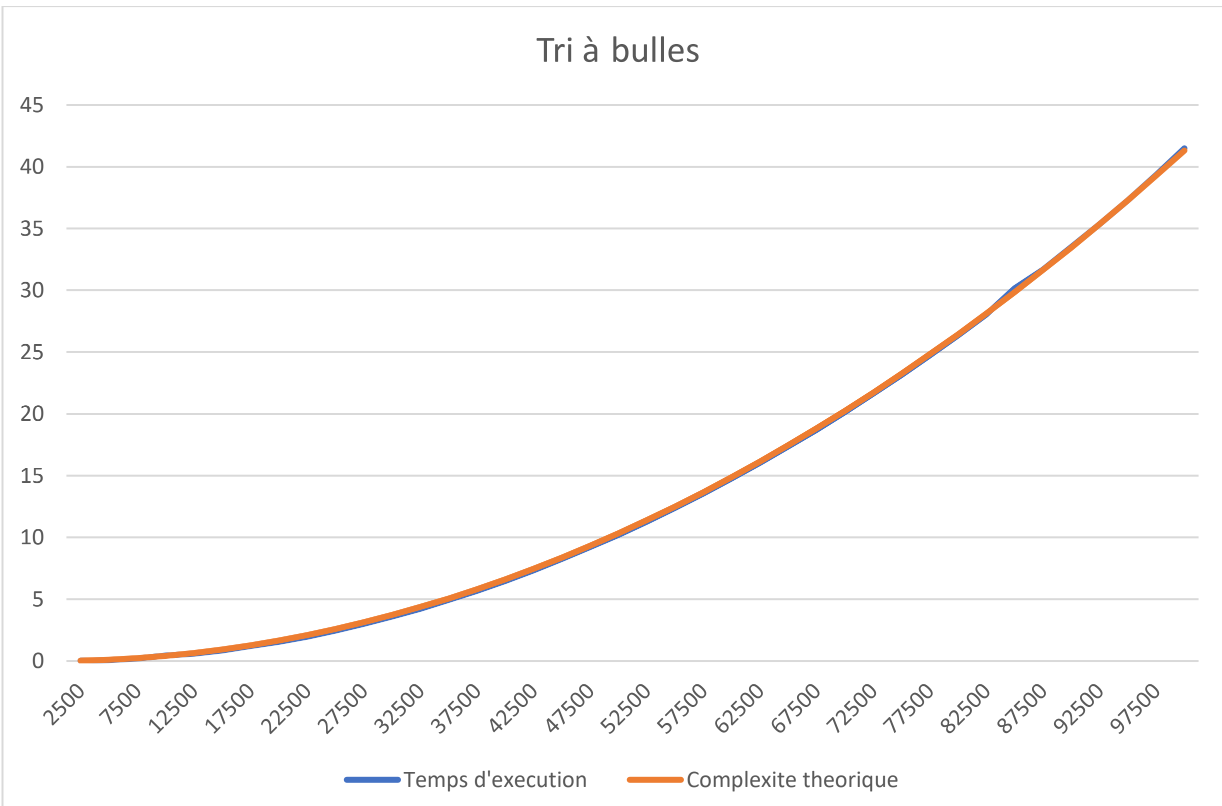
Et donc la complexité théorique de l'algorithme est de l'ordre $O(n^2)$.

- **Temps expérimentaux:**

Taille du tableau	Temps d'exécution (secondes)		
	Cas arbitraire	**Pire cas**	**Meilleur cas**
2500	0.0218	0.03078	0
5000	0.082	0.1188	0
7500	0.1936	0.25785	0
10000	0.428	0.4509	0.0002
12500	0.5788	0.69687	0
15000	0.8572	1.01142	0
17500	1.2034	1.35945	0
20000	1.5482	1.77606	0
22500	1.9734	2.24802	0
25000	2.4606	2.77128	0
27500	2.9892	3.34827	0
30000	3.5904	3.99897	0
32500	4.2212	4.68018	0.0006
35000	4.936	5.42403	0
37500	5.6784	6.22782	0
40000	6.478	7.07724	0
42500	7.3376	8.04006	0
45000	8.256	8.97345	0
47500	9.2292	9.97974	0.0004
50000	10.209	11.64051	0
52500	11.2806	13.65039	0.0006
55000	12.3804	14.41638	0.0002
57500	13.557	16.94088	0.0002
60000	14.7712	16.35849	0
62500	16.0382	17.40177	0
65000	17.3628	18.73287	0.0002
67500	18.7158	20.1582	0.0002
70000	20.1464	21.68964	0.0004
72500	21.6334	23.38821	0.0004
75000	23.1524	24.90021	0
77500	24.7298	26.59581	0.0002
80000	26.3568	28.4472	0

82500	28.0378	30.15063	0.0004
85000	30.1536	32.21019	0.0006
87500	31.6566	33.94683	0.0008
90000	33.5182	35.91729	0.0004
92500	35.3518	37.94229	0.0002
95000	37.2992	40.00239	0.0008
97500	39.3458	42.13917	0
100000	41.4992	44.36262	0.001

- **Représentation graphique**



- **Algorithme optimisé**

```
Procédure du TriBulleOpt (E/S: un tableau T[n] d'entiers ; E/n ;entier) Début
m = n-1 ;
    Changement = vrai ;
    Tant que (Changement=vrai) faire
        Changement □ faux ;
        pour i=1 à m faire
            si (T[i] > T[i+1]) alors
                Permuter(T[i], T[i+1]) ;
                Changement = VRAI;
            Fsi ;
        Fait;
        m = m-1;
    Fait;
Fin;
```

- **Algorithme optimisé en C :**

```c
void bullesOpt(int* T, int n){
    int changement = 1;
    int m = n-1;
    while(changement == 1){
        changement = 0;
        for(int i=0; i<m; i++){
            if(T[i] > T[i+1]){
                int temp = T[i];
                T[i] = T[i+1];
                T[i+1] = temp;
                changement = 1;
            }
        }
        m--;
    }
}
```

- **Analyse et complexité**

- Meilleur cas: le tableau est déjà trié, l'algorithme de tri a bulles performera une seule itération et donc O(n).
- Pire cas: tableau décroissant, la boucle extérieure sera exécutée (n-1) fois, et la boucle intérieure sera exécutée (n-m) fois :

T(n) = $\sum_{i=0}^{n-1} \sum_{j=0}^{n-i} j$ = $\sum_{i=0}^{n-1} (n-i)(n-i+1)/2$ = $O(n^2)$

Et donc la complexité théorique de l'algorithme est de l'ordre $O(n^2)$.

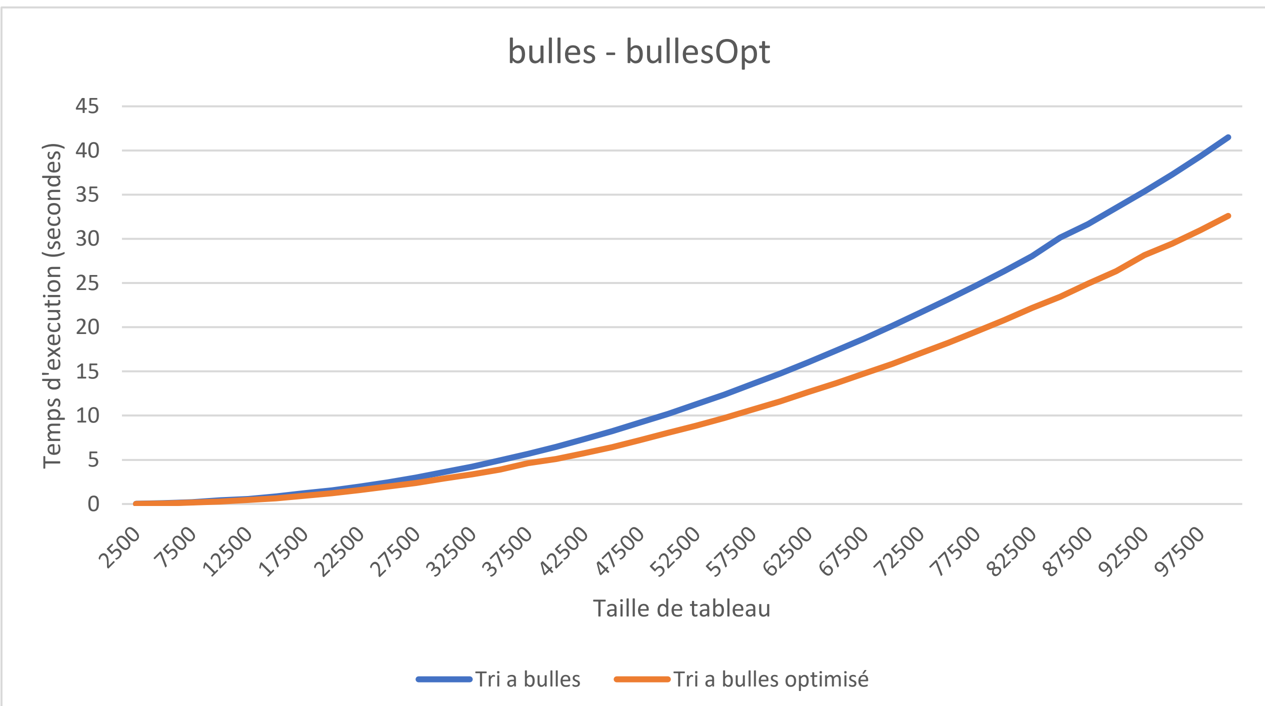
- **Comparaison entre les deux algorithmes**

- **Temps expérimentaux:**

Taille du tableau	Tri a bulles	Tri a bulles optimisé
2500	0.0218s	0.0148s
5000	0.082s	0.0646s
7500	0.1936s	0.1518s
10000	0.428s	0.2818s
12500	0.5788s	0.4574s
15000	0.8572s	0.6554s
17500	1.2034s	0.9134s
20000	1.5482s	1.2236s
22500	1.9734s	1.5636s
25000	2.4606s	1.9702s
27500	2.9892s	2.3822s
30000	3.5904s	2.8672s
32500	4.2212s	3.3514s
35000	4.936s	3.9016s
37500	5.6784s	4.6158s
40000	6.478s	5.088s
42500	7.3376s	5.7314s
45000	8.256s	6.4472s
47500	9.2292s	7.2396s
50000	10.209s	8.0734s
52500	11.2806s	8.871201s
55000	12.3804s	9.731199s
57500	13.557s	10.6632s
60000	14.7712s	11.623s
62500	16.0382s	12.648s
65000	17.3628s	13.6658s
67500	18.7158s	14.7468s
70000	20.1464s	15.8506s
72500	21.6334s	17.0288s
75000	23.1524s	18.2374s

77500	24.7298s	19.4954s
80000	26.3568s	20.794s
82500	28.0378s	22.1672s
85000	30.1536s	23.4678s
87500	31.6566s	24.9342s
90000	33.5182s	26.3474s
92500	35.3518s	28.1756s
95000	37.2992s	29.457s
97500	39.3458s	30.9982s
100000	41.4992s	32.6034s

- **Représentation graphique:**



En termes de temps d’exécution, les deux algorithmes ont des performances presque identiques, le dernier étant légèrement plus rapide.

2. Tri Gnome

L'algorithme est similaire au tri par insertion, sauf que, au lieu d'insérer directement l'élément à sa bonne place, l'algorithme effectue une série de permutations, comme dans un tri bulle.

«Gnome» car il paraît que c'est la méthode utilisée par les nains de jardin hollandais pour trier une rangée de pots de fleurs de la plus petite fleur à la plus grande...

- **Algorithme**

```
Procédure TriGnome (E/S: un tableau T[n] d'entiers ; E/n ;entier)
Début
    i = 1;
    Tant que(i<n) faire
        si (T[i] >= T[i-1]) alors
            i = i + 1;
        Sinon alors
            Permuter(T[i], T[i-1]);
            si (i>1) alors
                i = i -1;
            Fsi;
        Fsi
    Fait;
Fin;
```

- **Algorithme en C:**

```c
void gnome(int* T, int n){
    int i = 1;
    while(i<n){
        if(T[i] >= T[i-1]){
            i++;
        }
        else{
            swap(T, i, i-1);
            if(i>1) i--;
        }
    }
}
```

- **Analyse et complexité**

- Meilleur cas: le tableau est déjà trié, l'algorithme effectuera n comparaisons et donc O(n).
- Pire cas: tableau décroissant, pour chaque élément d'indice i du tableau, l'algorithme performera i permutations :

$$T(n) = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \; O(n^2)$$

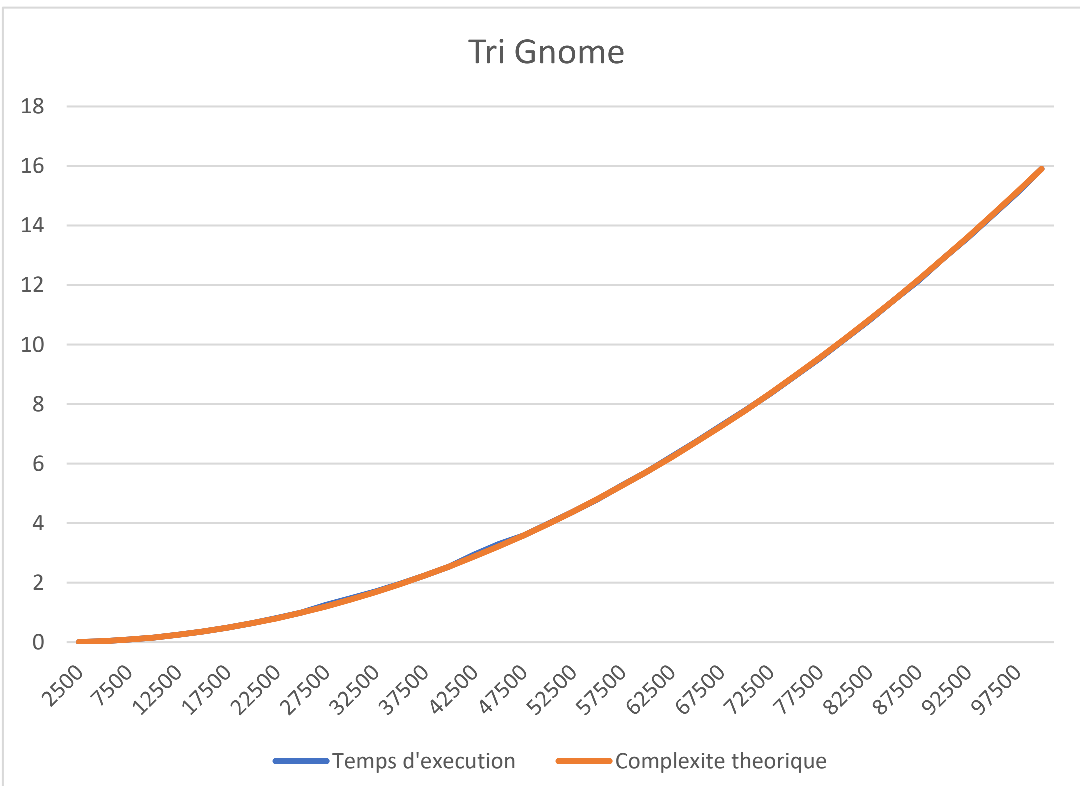
Et donc la complexité théorique de l'algorithme est de l'ordre $O(n^2)$.

- **Temps expérimentaux:**

Taille de tableau	Temps d'exécution (secondes)		
	Cas arbitraire	**Meilleur cas**	**Pire cas**
2500	0.0092	0	0.0264
5000	0.0374	0	0.1166
7500	0.0906	0	0.2654
10000	0.1592	0	0.463
12500	0.2476	0	0.7274
15000	0.3594	0	1.045
17500	0.4882	0	1.4192
20000	0.6374	0	1.857
22500	0.8068	0	2.349
25000	0.9932	0.0002	2.9032
27500	1.2536	0	3.505
30000	1.4826	0.0002	4.1782
32500	1.709	0.0002	4.905
35000	1.967	0	5.6846
37500	2.2364	0.0002	6.5224
40000	2.5388	0.0002	7.4278
42500	2.9416	0.0002	8.4752
45000	3.2918	0.0002	9.4192
47500	3.5854	0	10.5614
50000	3.981	0.0002	11.6168
52500	4.3838	0.0002	12.798
55000	4.8044	0	14.0426
57500	5.275001	0.0004	15.3604
60000	5.7238	0.0002	16.7002
62500	6.225801	0.0004	18.1304
65000	6.732	0.0004	19.6096
67500	7.2762	0.0002	21.186
70000	7.796201	0.0002	22.7438
72500	8.344999	0.0002	24.7218
75000	8.934401	0.0002	26.1854
77500	9.531799	0.0002	28.0204

80000	10.162	0.001	29.8754
82500	10.8024	0	31.7106
85000	11.4848	0.0002	33.6712
87500	12.1402	0	35.6802
90000	12.8822	0.0002	37.768
92500	13.588	0	39.8612
95000	14.3334	0.0004	42.026
97500	15.0918	0	44.3886
100000	15.905	0.0002	46.6768

- **Représentation graphique** :



3. Tri par base

Le tri par base, ou tri Radix, est un algorithme de tri, utilisé pour ordonner des éléments identifiés par une clef unique. Chaque clef est une chaîne de caractères ou un nombre que le tri par base trie selon l'ordre lexicographique. Cet algorithme a besoin d'être couplé avec un ou plusieurs algorithmes de tri stable. dans ce cas, nous avons utilisé l'algorithme de tri dénombrement.

- **Algorithme:**

```
Fonction cle(x, i: Entier)
Debut
    k1 = 1;
    pour j=1 a i+1 faire
        k1 = k1  * 10;
    Fait;
    k2 = k1/10;
    retourner ((x mod k1) - (x mod k2)) / k2;
Fin;

Procédure TriAux (E/S: tableau T[n] d'entiers ; E/n, i ;entier)
    Tableau cles[10] : entier
    Tableau resultat[n] : entier
Début
    pour k=0 a 9 faire
        cles[k] = 0;
    Fait;

    pour j=0 a n-1 faire
        cles[cle(T[j], i)]++;
    Fait;

    pour k=1 a 9 faire
        cles[k] += cles[k-1];
    Fait;

    pour j=n-1 a 0 faire
        resultat[cles[cle(T[j], i)] - 1] = T[j];
        cles[cle(T[j], i)]--;
    Fait;

    pour j=0 a n-1 faire
        T[j] = resultat[j];
    Fait;

Fin;

Procédure TriBase (E/S: tableau T[n] d'entiers ; E/n, k ;entier)
Debut;
    pour i=0 a k-1 faire
        triAux(T, n, i);
    Fait;
Fin;
```

- **Algorithme en C:**

```c
int cle(int i, int j){

    int k1 = pow(10, j+1);
    int k2 = k1/10;

    int t = ((i % k1) - (i % k2))/k2;

    return t;
}

void triAux(int* T, int n, int i){
    int cles[10];
    int *resultat = (int*)malloc(n * sizeof(int));

    for(int k=0; k<10; k++) cles[k] = 0;

    for(int j=0; j<n; j++){
        cles[cle(T[j], i)]++;
    }

    for(int k=1; k<10; k++) cles[k] += cles[k-1];

    for(int j=n-1; j>=0; j--){
        resultat[cles[cle(T[j], i)] - 1] = T[j];
        cles[cle(T[j], i)]--;
    }

    for(int j=0; j<n; j++) T[j] = resultat[j];

}

void TriBase(int* T, int n, int k){
    for(int i=0; i<k; i++){
        triAux(T, n, i);
    }
}
```

- **Analyse et complexité**
    - La fonction Clé est d'ordre $O(1)$.
    - La procédure TriAux s'exécute en un temps linéaire $O(n)$.

    La procédure TriAux sera exécutée k fois et donc la complexité théorique de cet algorithme est $O(kn)$.

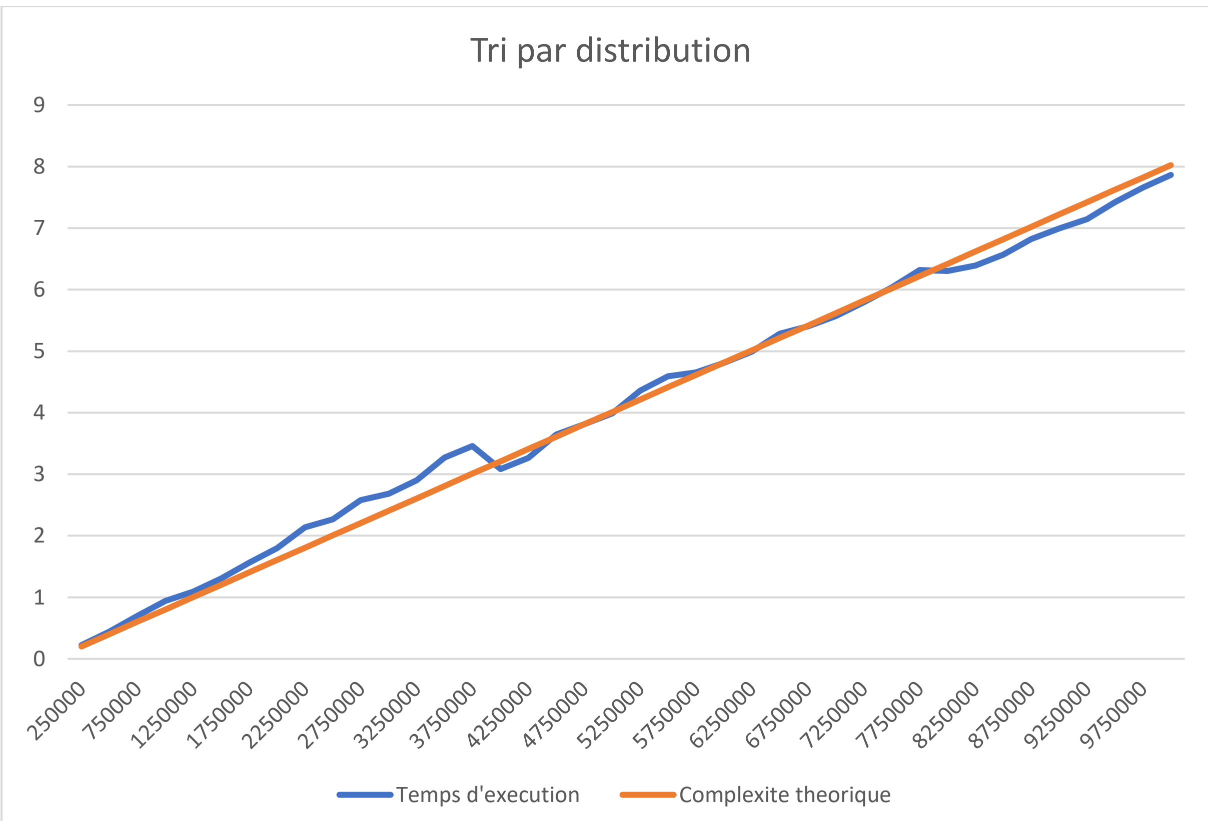
- **Temps expérimentaux :**

Taille de tableau	Temps d'execution (secondes)
250000	0.2232
500000	0.4408
750000	0.698
1000000	0.9394
1250000	1.0898
1500000	1.3064
1750000	1.559
2000000	1.7998
2250000	2.137
2500000	2.2682
2750000	2.581
3000000	2.6856
3250000	2.8994
3500000	3.2712
3750000	3.4588
4000000	3.0822
4250000	3.2682
4500000	3.6464
4750000	3.811
5000000	3.99
5250000	4.3582
5500000	4.5902
5750000	4.652
6000000	4.8114
6250000	4.993
6500000	5.2816
6750000	5.4056
7000000	5.5676
7250000	5.7942
7500000	6.033
7750000	6.3172
8000000	6.3028
8250000	6.3898
8500000	6.5704
8750000	6.8214

9000000	6.9932
9250000	7.1454
9500000	7.4228
9750000	7.6616
10000000	7.864

Les temps d'exécution est beaucoup plus rapide, nous avons donc du utilise des tailles des tables relativement grandes.

- **Représentation graphique :**



4. Tri rapide

Le tri rapide ou tri pivot (en anglais Quicksort) est un algorithme de tri inventé par C.A.R. Hoare en 19613 et fondé sur la méthode de conception diviser pour régner. Il est généralement utilisé sur des tableaux, mais peut aussi être adapté aux listes. Dans le cas des tableaux, c'est un tri en place mais non stable.

Le choix de l'élément pivot varie d'un algorithme a l'autre, dans notre programme **Pivot = 1er Elément**.

- **Algorithme:**

```
Procédure partitionner (E/S: tableau T[n] d'entiers ; E/d, f ;entier)
Debut
    eltPivot = d;
    i = d+1;
    j = f;

    Tant que(i<j) faire
        Tant que (i<=f et T[i]<=T[eltPivot]) faire
            i = i + 1;
        Fait;

        Tant que (j>=d et T[j]>T[eltPivot]) faire
            j = j - 1;
        Fait;

        si(i<j) alors
            echanger(T[i], T[j]);
            j = j - 1;
            i = i + 1;
        Fsi;
    Fait;

    si(i==j et T[j] > T[eltPivot]) alors
        j = j - 1;
    Fsi;

    echanger(T[j], T[eltPivot]);

    return j;
}


Procédure rapide (E/S: tableau T[n] d'entiers ; E/p, r:entier)
Debut
    si(p<r) alors
        q = partitionner(T, p, r);
        rapide(T, p, q-1);
        rapide(T, q+1, r);
    Fsi;
}
```

- **Algorithme en C:**

```c
int partitionner(int* T, int d, int f){
    int eltPivot = d;
    int i = d+1;
    int j = f;

    while(i<j){
        while (i<=f && T[i]<=T[eltPivot]) i++;
        while (j>=d && T[j]>T[eltPivot]) j--;

        if(i<j){
            swap(T, i, j);
            j--;
            i++;
        }
    }

    if(i==j){
        if(T[j] > T[eltPivot]) j--;
    }

    swap(T, j, eltPivot);

    return j;
}


void rapide(int* T, int p, int r){
    int q;
    if(p<r){
        q = partitionner(T, p, r);
        rapide(T, p, q-1);
        rapide(T, q+1, r);
    }
}
```

- **Analyse et complexité**
  - La complexité de la fonction Partitionner est de l'ordre $\Theta(n)$.

  L'équation de récurrence du programme :

$$\begin{cases} T(1) = 0 \\ T(n) = T(q) + T(n-q) + \Theta(\mathrm{n}) \end{cases}$$

  - Pire cas : $q = 1$ ou $q = n - 1$

$$T(n) = T(1) + T(n-1) + \mathrm{c.n}$$

$$T(n) = T(n-1) + \mathrm{c.n}$$

$$T(n) = T(n-2) + \mathrm{c(n-1)} + \mathrm{c.n}$$

$$\dots$$

$$T(n) = T(1) + \mathrm{c}(1 + 2 + 3 + \cdots + \mathrm{n})$$

$$T(n) = \mathrm{c}\,.\left(\frac{\mathrm{n(n+1)}}{2}\right)$$

$$\mathrm{T(n)} = \mathrm{O}(n^2)$$

  - Cas moyen :

$$T(n) \leq T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \mathrm{c.n}$$

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \mathrm{c.n}$$

$$T(n) \leq 2\left(2T\left(\frac{n}{4}\right) + \mathrm{c}\left(\frac{\mathrm{n}}{2}\right)\right) + \mathrm{c.n}$$

$$T(n) \leq 4T\left(\frac{n}{4}\right) + \mathrm{c.n} + \mathrm{c.n}$$

$$\dots$$

$$T(n) \leq k.T\left(\frac{n}{2^k}\right) + k\mathrm{c.n}$$

On pose : $2^k = n$ → $\mathrm{k} = \log n$:

$$T(n) \leq logn.T(1) + c.nlogn$$

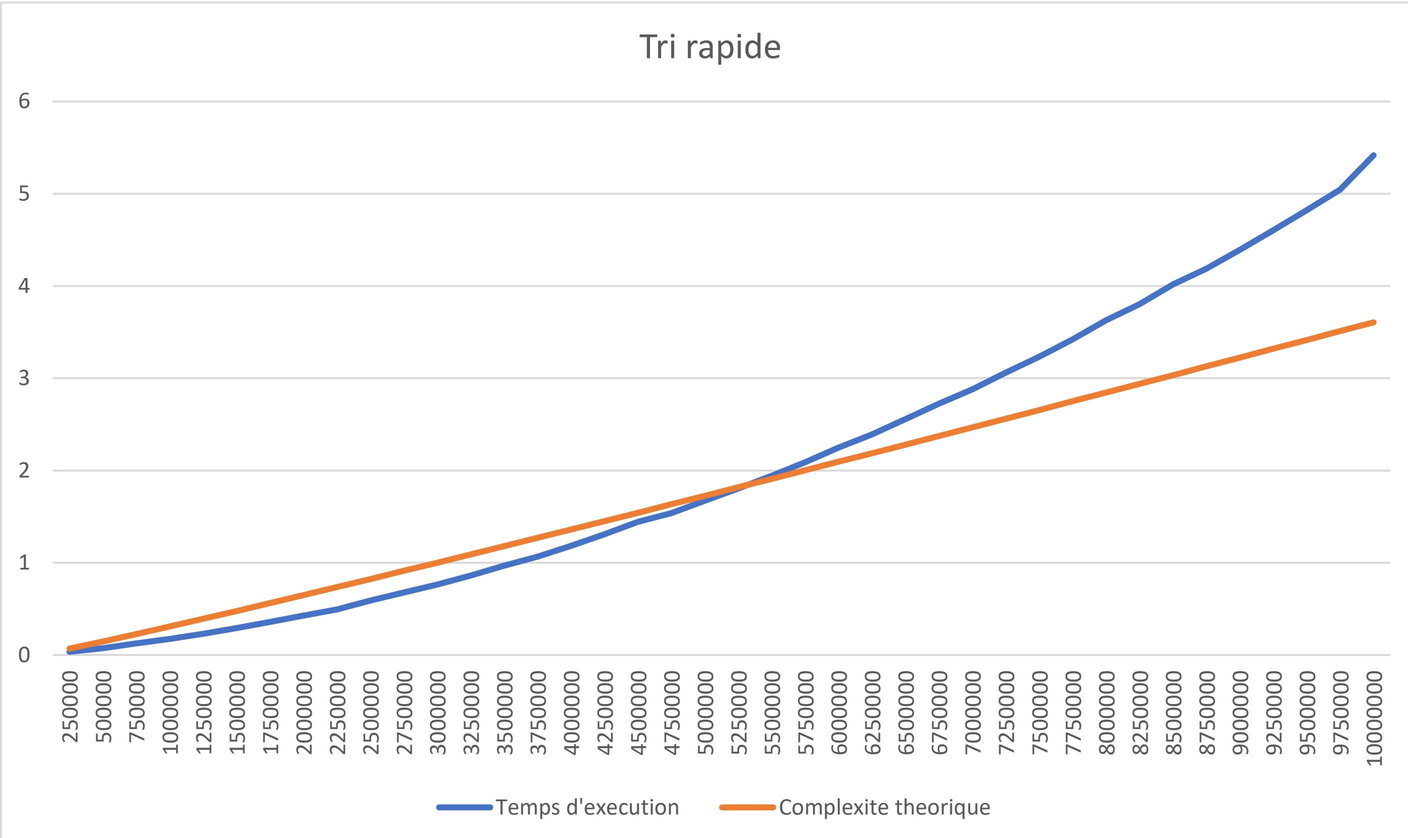
$$T(n) \leq \mathrm{c.nlogn}$$

$$T = O(n\mathrm{logn})$$

- **Temps expérimentaux :**

Taille de tableau	Temps d'exécution (secondes)	
	Cas arbitraire	Pire cas
250000	0.0346	11.44183
500000	0.073	24.15982
750000	0.126	37.3595
1000000	0.1742	50.87197
1250000	0.2304	64.61705
1500000	0.294	78.54749
1750000	0.3596	92.63207
2000000	0.4282	106.8486
2250000	0.494	121.1805
2500000	0.591	135.6149
2750000	0.6766	150.1416
3000000	0.7648	164.752
3250000	0.8628	179.4392
3500000	0.9694	194.1973
3750000	1.0662	209.0212
4000000	1.1824	223.9066
4250000	1.3084	238.8495
4500000	1.4434	253.8465
4750000	1.5376	268.8948
5000000	1.6738	283.9915
5250000	1.8044	299.1343
5500000	1.9434	314.3209
5750000	2.089	329.5494
6000000	2.2472	344.8179
6250000	2.3922	360.1248
6500000	2.556	375.4686
6750000	2.7224	390.8477
7000000	2.8784	406.2609
7250000	3.0614	421.7071
7500000	3.232	437.1849
7750000	3.421	452.6935
8000000	3.629	468.2318
8250000	3.8026	483.7988
8500000	4.0172	499.3937
8750000	4.187	515.0158
9000000	4.3908	530.6641
9250000	4.6048	546.338
9500000	4.822	562.0368
9750000	5.0458	577.7598
10000000	5.4164	593.5064

- **Représentation graphique :**



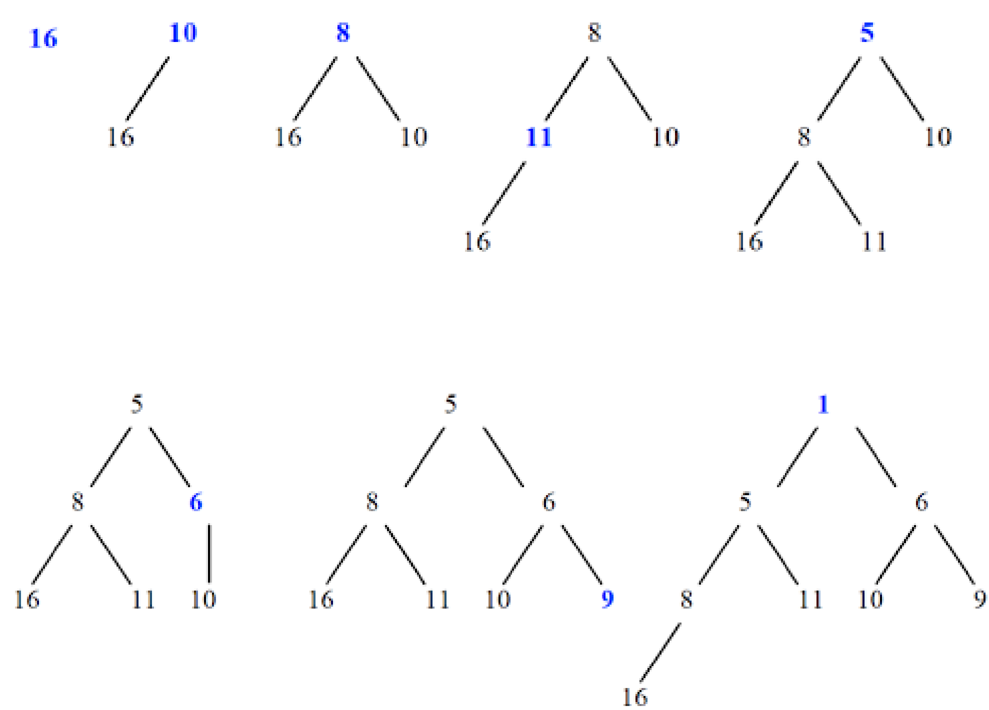
5. Tri par tas

Le tri par tas (Heapsort en anglais) est un algorithme de tri par comparaisons. Cet algorithme est de complexité asymptotiquement optimale, c'est-à-dire que l'on démontre qu'aucun algorithme de tri par comparaison ne peut avoir de complexité asymptotiquement meilleure. Par contre, il n'est pas stable. Son inconvénient majeur est sa lenteur comparée au tri rapide.

5.1. Construction de l’arbre binaire (Min-Heap) :

Le tri par tas se base sur une structure de données particulière : le tas. Il s’agit d’une représentation d’un arbre binaire sous forme de tableau.

Dans notre cas, on a s’intéresse au Tas-min (Min-Heap), c.à.d. la relation d’ordre est ‘’inférieure ou égale’’, la racine de l’arbre étant le plus petit élément du tableau.



- **Algorithme :**

```
Procédure remonter (E/S: tableau T[n] d'entiers ; E/i, n:entier)
Debut
    si(i==1) alors
        retourner;
    Fsi;

    si(T[i] < T[i/2]) alors
        echanger(T[i], T[i/2]);
        remonter(T, i/2, n);
    Fsi;
Fin;

Procédure entasser (E/S: tableau T[n] d'entiers ; E/n:entier)
Debut
    pour i=1 a n-1 faire
        remonter(T, i, n);
    Fait;
Fin;
```

- **Algorithme en C :**

```c
void remonter(int* T, int i, int n){
    if(i==1) return;

    if(T[i] < T[i/2]){
        swap(T, i, i/2);
        remonter(T, i/2, n);
    }
}

void entasser(int* T, int n){
    for(int i=1; i<n; i++){
        remonter(T, i, n);
    }
}
```

- **Complexité**

  Complexité de la procédure Remonter :

$$\begin{cases} T_R(1) = 0 \\ T_R(i) \leq T\left(\frac{i}{2}\right) + 1 \end{cases}$$

$$T_R(i) \leq T\left(\frac{i}{2}\right) + 1$$

$$T_R(i) \leq T\left(\frac{i}{4}\right) + 2$$

$$\ldots$$

$$T_R(i) \leq T\left(\frac{i}{2^k}\right) + k$$

On pose $i = 2^k$:

$$T_R(i) \leq T(1) + \log i$$

$$T_R(i) = O(\log i)$$

Complexité de la procédure Entasser :

$$T_E(n) = \sum_{i=1}^{n} k.\log i$$

$$T_E(n) = k(1\log 1 + 2\log 2 + \cdots + n.\log n)$$

$$k(\log 1 + \log 2 + \cdots + \log n) \leq k(\log n + \log n + \cdots + \log n)$$

$$T_E(n) \leq kn.\log n$$

$$T_E(n) = O(n.\log n)$$

5.2. Tirage du minimum:

Une fois que l'arbre est construit, on tire successivement les éléments de la racine de l'arbre, en les plaçant à la fin du tableau, et on réordonne l'arbre binaire réduit après chaque tirage.

Au bout de l'algorithme, le tableau sera trié dans l'ordre décroissant.

- **Algorithme :**

```
Procédure redescendre (E/S: tableau T[n] d'entiers ; E/i, n:entier)
Debut
    si (2*i+1 >= n) alors retourner Fsi;

    si(T[2*i+1] < T[2*i]) alors
        imin = 2*i + 1;
    sinon alors
        imin = 2*i;
    Fsi;

    si(T[imin] < T[i]) alors
        echanger(T[i], T[imin]);
        redescendre(T, imin, n);
    Fsi;
Fin;


Procédure tri_tas (E/S: tableau T[n] d'entiers ; E/n:entier)
Debut
    entasser(T, n);

    pour i=n-1 a 1 faire
        echanger(T, 1, i);
        redescendre(T, 1, i);
    Fait;

Fsi;
```

- **Algorithme en C :**

```c
void redescendre(int* T, int i, int n){
    int imin;

    if(2*i+1 >= n) return;
    if(T[2*i+1] < T[2*i]) imin = 2*i + 1;
    else imin = 2*i;

    if(T[imin] < T[i])
    {
        swap(T, i, imin);
        redescendre(T, imin, n);
    }
}


void tri_tas(int* T, int n){
    entasser(T, n);

    for(int i=n-1; i>0; i--){
        swap(T, 1, i);
        redescendre(T, 1, i);
    }

}
```

Pour transformer le résultat en ordre croissant il suffit d’inverser le tableau :

```c
void inverser(int* T, int n){
    for(int i=1; i<n/2; i++){
        swap(T, i, n-i);
    }
}
```

- **Complexité**

Complexité de la procédure Redescendre :

$$\begin{cases} T_D(n) = 0 \\ T_D(i) \leq T(2i) + 1 \end{cases}$$

$$T_D(i) \leq T(2i) + 1$$

$$T_D(i) \leq T(4i) + 2$$

$$T_D(i) \leq T(8i) + 3$$

$$\dots$$

$$T_D(i) \leq T\left(2^k i\right) + k$$

On pose $n = \ 2^k\ i\ \rightarrow k = \log n - \log i$

$$T_D(i) \leq \log n - \log i$$

$$T_D(i) = \ O(\log n)$$

Complexité de la procédure Tri_tas :

$$T(n) = \ \sum_{i=1}^{n-1} k. \log n + 1$$

$$T(n) = O(n. \log n)$$

Complexité de la procédure Inverser : $O\left(\frac{n}{2}\right) = \ O(n)$

- **Complexité totale de l'algorithme tri par tas:**

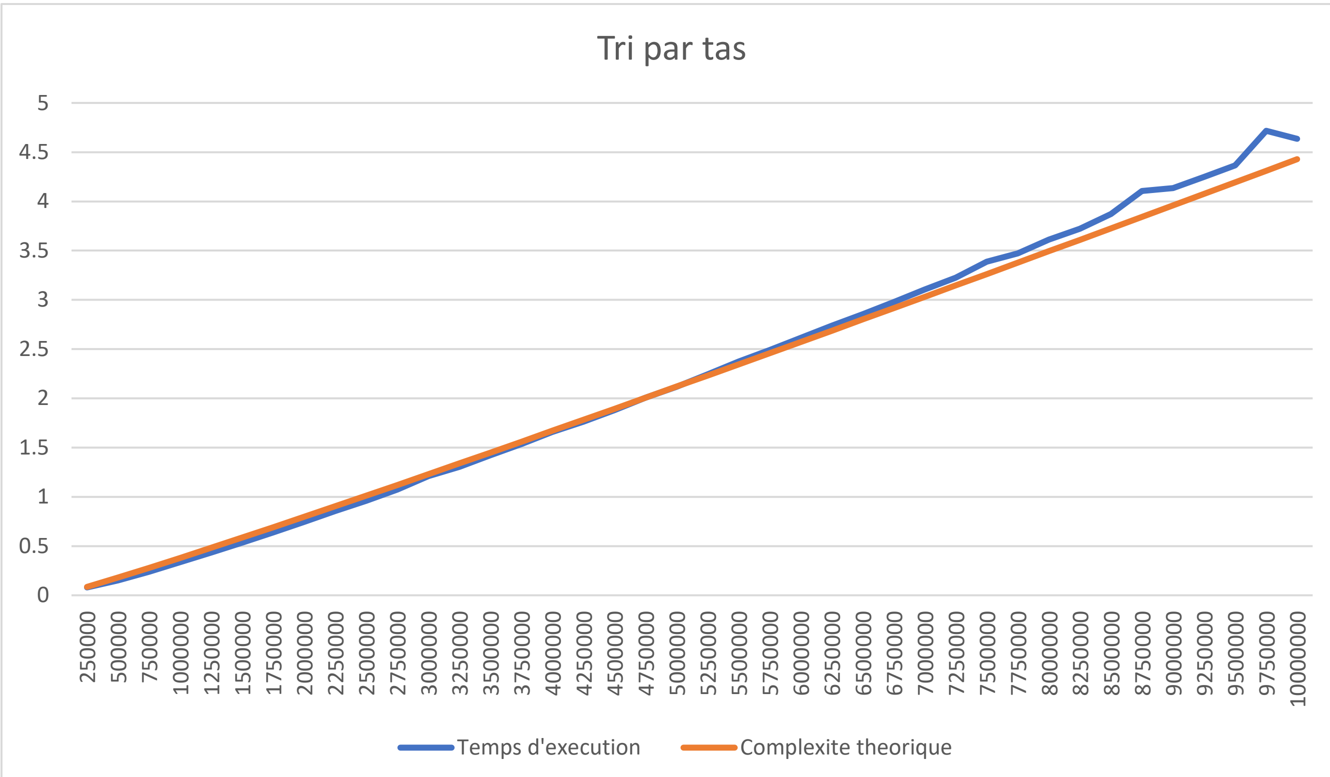
$$T(n) = O(n. \log n) + O(n. \log n) + O(n)$$

$$T(n) = O(n. \log n)$$

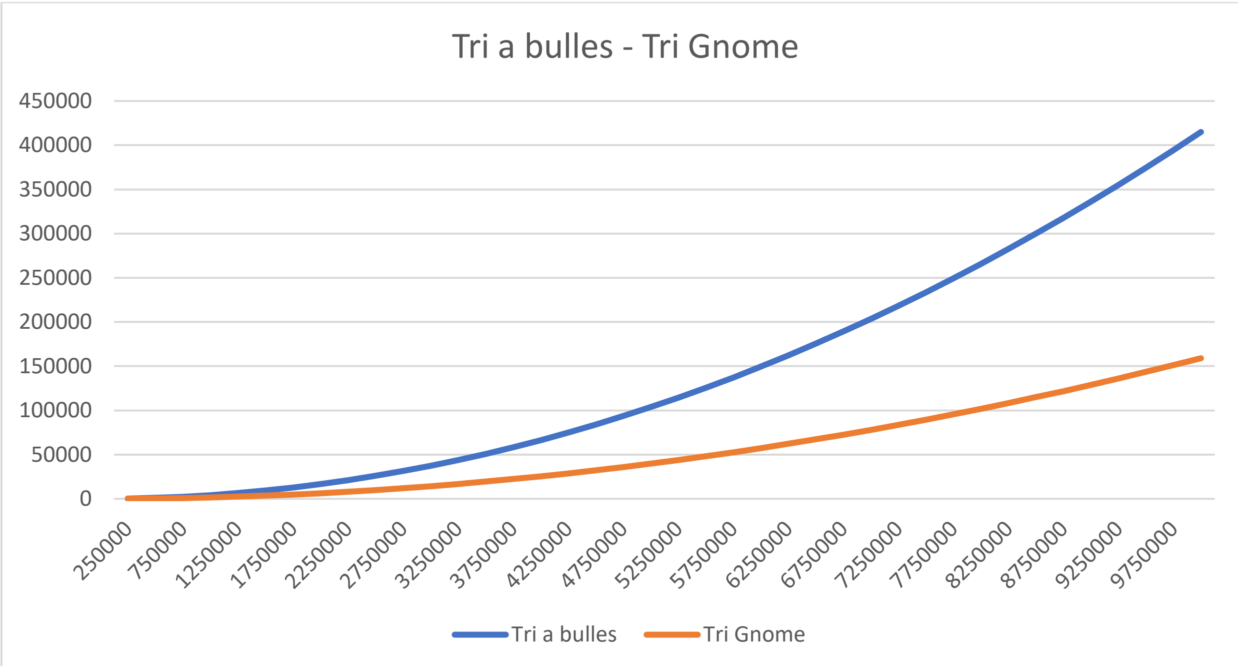
- **Temps d’exécution :**

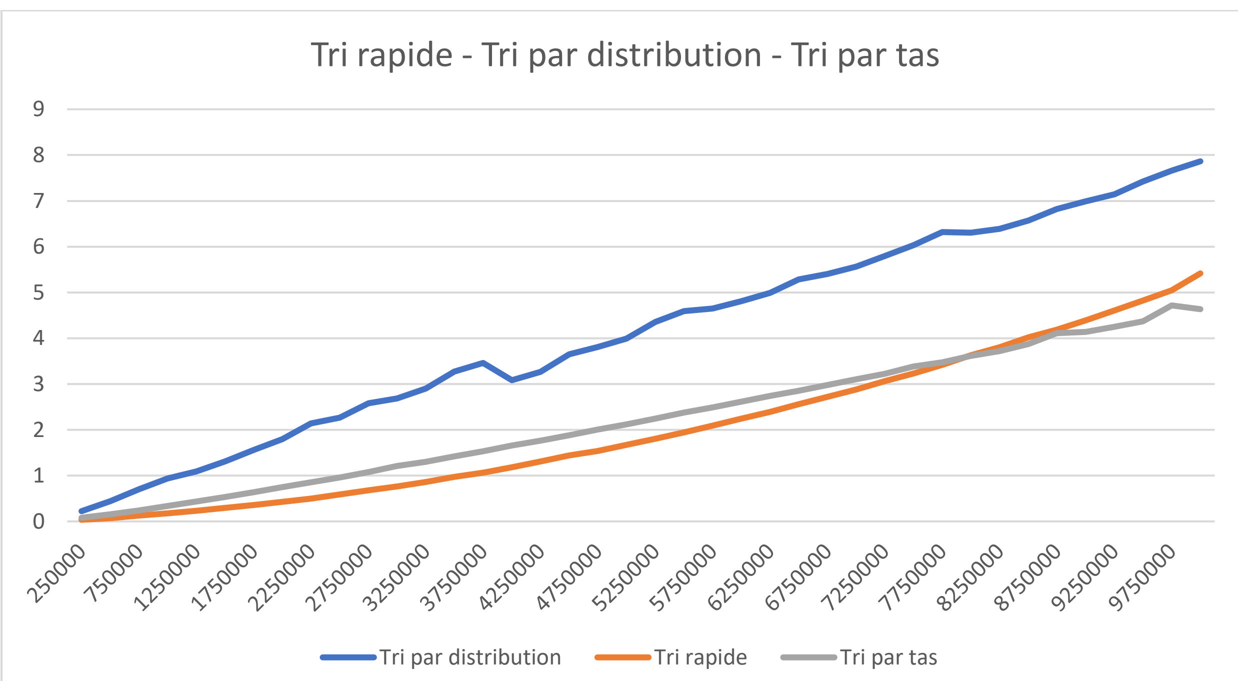
Taille de tableau	Temps d'execution (secondes)
250000	0.0798
500000	0.1516
750000	0.2386
1000000	0.3356
1250000	0.434
1500000	0.5344
1750000	0.6382
2000000	0.7466
2250000	0.8568
2500000	0.9602
2750000	1.0744
3000000	1.2114
3250000	1.3032
3500000	1.422
3750000	1.5346
4000000	1.6574
4250000	1.7636
4500000	1.8814
4750000	2.0064
5000000	2.1162
5250000	2.2446
5500000	2.375
5750000	2.4908
6000000	2.6158
6250000	2.7372
6500000	2.8532
6750000	2.9786
7000000	3.106
7250000	3.225
7500000	3.387
7750000	3.4714
8000000	3.6132
8250000	3.722
8500000	3.873
8750000	4.107
9000000	4.136
9250000	4.2472
9500000	4.3668
9750000	4.717
10000000	4.636

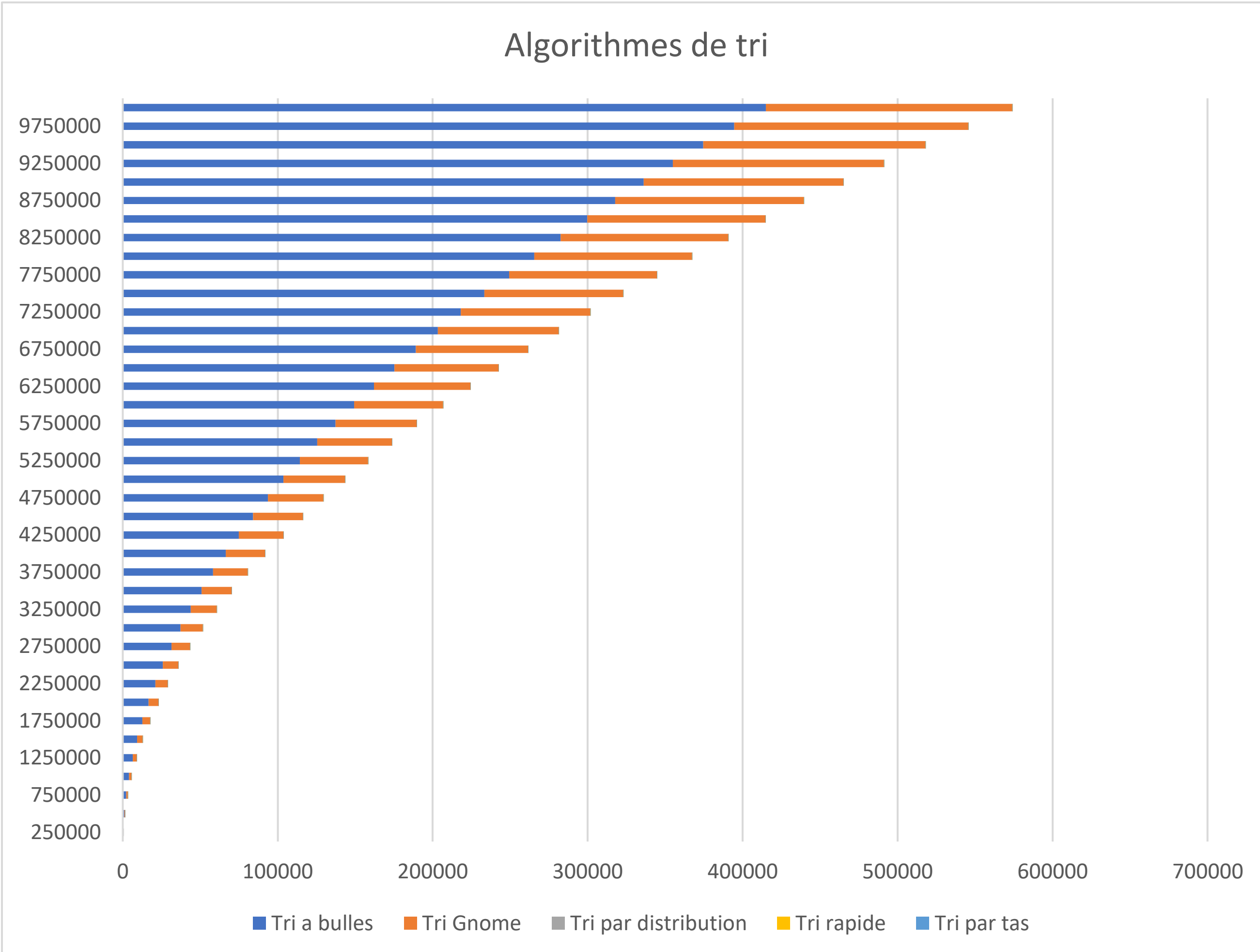
- **Représentation graphique :**



6. Comparaison des algorithmes







❖ Conclusion

En termes de temps d'exécution, l'algorithme le plus performant est le **Tri rapide (Quicksort),** puis viennent le **Tri par tas** et le **Tri par distribution**, ces 3 algorithmes ont une complexité théorique de l'ordre **O(n logn).**

Le **Tri a bulles** est l'algorithme le plus lent, suivi par le **Tri Gnome**, les deux ont une complexité de l'ordre **O(n²).**

Tri rapide > Tri par tas > Tri par distribution > Tri Gnome > Tri à bulles

❖ Références

Wikipédia :

- Algorithmes de tri -- Algorithme de tri — Wikipédia (wikipedia.org)

- Tri a bulles -- Tri à bulles — Wikipédia (wikipedia.org)

- Tri par base -- Tri par base — Wikipédia (wikipedia.org)

- Tri rapide -- Tri rapide — Wikipédia (wikipedia.org)

- Tri par tas -- Tri par tas — Wikipédia (wikipedia.org)

Geeks for Geeks :

- Sorting algorithms -- Sorting Algorithms - GeeksforGeeks

- Gnome sort -- Gnome Sort - GeeksforGeeks

- Radix sort -- Radix Sort - GeeksforGeeks