Université de Sciences et de la Technologie Houari Boumediene

Faculté Informatique

Rapport Projet 4

TP 4 Algorithmique et complexité

Les algorithmes de tri

Mahmoud Bacha rabah sami G_02

Saadallah Abderahim G_03

M1 RSD 2023/2024

Introduction :

En informatique , le tri des donnés est un processus indispensable car il sert à faciliter et optimiser d'autre taches tel que la recherche des objets et les calculs statistiques.
Plusieurs algorithmes de tri ont été proposé et donc la vrai problématique est le choix d'un algorithme optimal.

Le choix dépend des ressources qu'on dispose , des donnés qu'on veut trié , et surtout du temps d'exécution , on parle ici de la complexité temporelle

Dans cette étude nous présentons une étude de complexité et une comparaison entre les 5 algorithmes suivant :

- Tri par Bulle (Bubble sort)
- Tri Gnome (Gnome sort
- Tri par Base (Radix sort)
- Tri Rapide (Quick sort)
- Tri par Tas (Heap-sort)

Environnement de travail :

- Langage : C , gcc 6.3.0
- CPU: i5 8365U 3.9 GHZ max frequency
- Ram: 8 GB
- OS: windows 11
- Logical: VScode

Pour l'etude expérimental , on utilise des tableaux dynamique de type int *array , pour une utilisation optimisé de la mémoire

Tri par Bulle :

1) Implémentation de l'algorithme :

```
void tri_bulle(int *tableau , int N ){

    int change = 1 ;
    while (change==1)
    {
        change =0;
        for (int i = 0; i < N-1; i++)
        {
            if(tableau[i] > tableau[i+1]){

                permuter(tableau , i , i+1);
                change = 1;
            }
        }
    }
}

void tri_bulle_optimise(int *tableau , int N ){

    int change = 1 ;
    int m = N -1 ;
    while (change==1)
    {
        change =0;
        for (int i = 0; i < m; i++)
        {
            if(tableau[i] > tableau[i+1]){
                permuter(tableau , i , i+1);
                change = 1;
            }
        }
        m--;
    }
}
```

2) Complexité :

- Meilleur cas : $O(N)$

  Dans le Meilleur cas , le tableau est déjà trié .

  La boucle while s'exécute une seule fois car valeur de change va être 0 et donc on ne fait que N-1 itération

  $T(N) = N - 1$

- Pire cas : $O(N^2)$

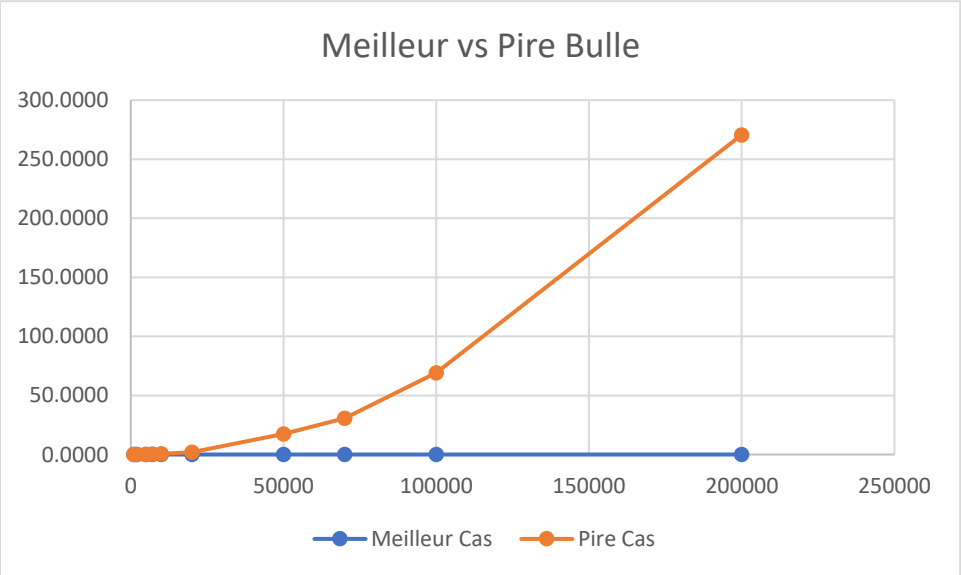
  Dans le pire cas , le tableau est trié dans l'ordre inverse

  - Version non optimisé :
    La boucle while s'exécute N fois et dans chaque itération on exécute la boucle for ayant N – 1 itérations
    $T(N) = N(N-1)$

  - Version optimisé :
    La boucle while s'exécute N fois et le nombre d'itération de la boucle for diminue à chaque fois
    $T(N) = \sum_0^{N-1} i = \frac{N(N-1)}{2}$

Comparaison expérimental :

Taille	1000	2000	5000	7000	10000	20000	50000	70000	100000	200000
Meilleur Cas	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0010
Pire Cas	0.0060	0.0210	0.1400	0.2920	0.6070	1.8950	17.3770	30.6310	69.1200	270.6120



3) Exemple d'exécution :

```
C:\Users\DELL\Desktop\1algo\tp\tp4>tri

 tri par bulle
avant tri
46  45  40  37  36  34  30  27  23  21  17  17  14  13  2
apres tri
2  13  14  17  17  21  23  27  30  34  36  37  40  45  46
```

```
 tri par bulle optimized
avant tri
18  35  37  15  21  36  38  4  26  28  21  17  24  31  43
apres tri
4  15  17  18  21  21  24  26  28  31  35  36  37  38  43
```

Tri Gnome :

1) Implémentation de l'algorithme :

```
    void tri_gnome( int * tableau , int N ){

    int i=0;

    while ( i < N )
    {
        if(i==0) i++;

        if(tableau[i] < tableau[i-1]){
            permuter( tableau , i , i-1);
            i--;
        }else{
            i++;
        }
    }
}
```

2) Complexité :

- Meilleur cas : $O(N)$
  Lorsque le tableau est déjà trié , $tab[i] > tab[i-1]$ et donc i incremente a chaque fois , la boucle while fait N itération

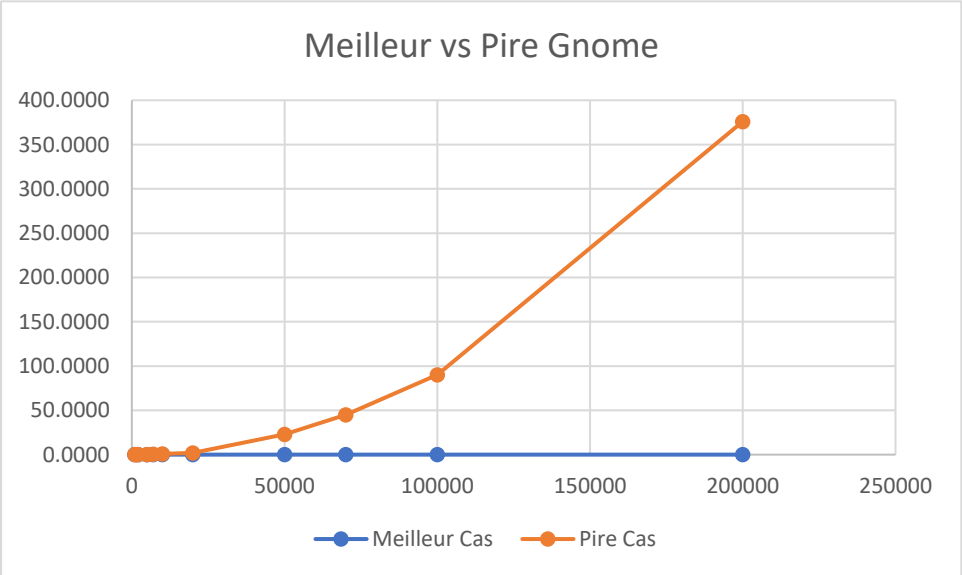
  $T(N) = N$

- Pire cas : $O(N^2)$
  Lorsque le tableau est trié a l'ordre inverse
  pour chaque itération de while , i va être remet a 0 car $tab[i] < tab[i-1]$ et donc à chaque incrémentation on réexécute la boucle while et on diminue i

  $T(N) = \sum_0^{N-1} i = \frac{N(N-1)}{2}$

Comparaison expérimental :

Taille	1000	2000	5000	7000	10000	20000	50000	70000	100000	200000
Meilleur Cas	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0010	0.0000	0.0000	0.0010
Pire Cas	0.0080	0.0290	0.1990	0.3680	0.7590	1.8120	22.8120	45.1210	90.2490	375.9930



3) Exemple d'exécution :

```
 tri gonome :
avant tri
47  38  34  32  27  21  20  19  18  17  8  6  3  2  1
apres tri
1  2  3  6  8  17  18  19  20  21  27  32  34  38  47
```

Tri par Base :

1) Implémentation de l'algorithme :

```
int cle(int number , int digit){

    return (number / digit)% 10;    // tel que digits est 10 100 1000 ...
}

void triaux(int **tab, int N, int digit) {
    int *output = creer_tab_init(N, 0);
    int *count = creer_tab_init(base, 0);  // Array to count the occurrences

    // compter le nombre de digits de tous les cases du tableaux
    for (int i = 0; i < N; ++i) {

        count[cle((*tab)[i],digit)]++;
    }

    // incrementaion
    for (int i = 1; i < base; ++i) {
        count[i] += count[i - 1];
    }


    for (int i = N - 1; i >= 0; --i) {
        output[count[cle((*tab)[i],digit)] - 1] = (*tab)[i];
        count[cle((*tab)[i],digit)]--;
    }

    free(*tab);    // free de la memoire
    *tab = output;
    free(count);
}

void tri_base(int **tab, int N) {
    int maximum = max(*tab, N);

    int digit = 1;
    while (maximum / digit > 0) {
        triaux(tab, N, digit);
        digit *= 10;
    }
}
```

2) Complexité :

D'abord une remarque importante est que cet algorithme a une complexité spatiale $O(\ d + N\ )$
Car la fonction tri aux va créer deux tableaux de taille N et d , le premier pour stocker les valeurs triés et le deuxièmes pour compter le nombre d'apparition de chaque chiffre , dans notre cas base = 10 et donc les chiffres vari de 0 a 9.

La fonction Triaux : $O(N)$

- N itération pour compter le nombre de chiffres de chaque case
- 10 itération pour la somation
- N itération pour le tri selon le chiffre
- $T_1(N) = 2N + 10$

La fonction tri_base : $O(d * N)$

- Le nombre d'appel de la fonction Triaux est déterminer par $d = nombre\ de\ chiffre\ du\ maximum$
  On doit d'abord récupérer le maximum du tableau
- N itération pour trouver le max
- d itération de la fonction triaux
  $T(N) = d(\ 2N + 10)$ +N

Cet algorithme ne se base pas sur des comparaisons et donc l'ordre initial du tableau donnera le même temps d'exécution

Le pire cas existe lorsque le nombre de chiffres du maximum est très grand , dans ce cas, la complexité arrive a $O(N^2)$

Dans notre étude actuelle , $d << << N$, ce qui explique que la complexité est O(N)

3) Exemple d'exécution :

```
 tri distribution :
avant tri
147  202  1  149  289  249  13  130  6  294  128  101  295  110  221
apres tri
1  6  13  101  110  128  130  147  149  202  221  249  289  294  295
```

Tri Rapide :

1) Implémentation de l'algorithme :

```
int partition(int *tab , int deb , int fin ) {
    int pivot = tab[fin];
    int pivindex = deb - 1;          // pivot est le dernier element
    for (int i = deb; i < fin; i++)
    {
        if (tab[i] <= pivot)
        {
            pivindex ++;
            permuter(tab , i, pivindex);        // on permute si tab[i] <= pivot

        }

    }
    permuter(tab , pivindex+1 , fin);
    return pivindex+1 ;

}
void tri_rapide(int *tab , int deb , int fin ){
    if (deb < fin)
    {
        int pindex = partition(tab , deb , fin );
        tri_rapide(tab , deb , pindex -1);      // tableau gauche
        tri_rapide(tab , pindex +1 , fin);      // tableau droite
    }

}
```

2) Complexité :

La complexité dépend du choix du pivot , la complexité est entre $O(N \log N)$ *et* $O(N^2)$

- Meilleur cas : $O(N \log_2 N)$

Lorsque le choix du pivot découpe a chaque appel récursive le tableau en deux sous tableaux de taille $\frac{N}{2}$

dans ce cas , la fonction tri_rapide fait exactement $\log N$ $itérations$

et la fonction partition fait N $itérations$

T(N) = $(N \log N)$

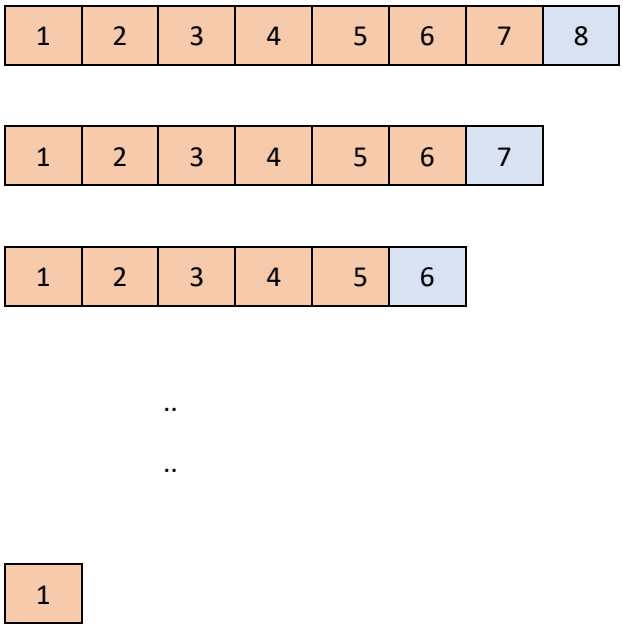
- Pire cas : $O(N^2)$

Si le choix du pivot va mener a une division inéquitable des tableau a chaque appel recursive le tableau a deux tableaux , le premiers ayant taille = 0 et le deuxième ayant taille = N-1

Dans ce cas , $T(N) = 1 + 2 + \cdots + N - 1 + N = \frac{N(N-1)}{2}$

Dans notre implémentation le pivot est le dernier élément du tableau
mais les valeurs du tableau sont prises d'une façon aléatoire donc on peut assumer que le pire ne sera appliqué à chaque itération.

Par contre on peut tester le pire cas en appliquant l'algorithme sur un tableau déjà trié Dans ce cas, a chaque appel tableau gauche de taille N – 1 et tableau droit vide



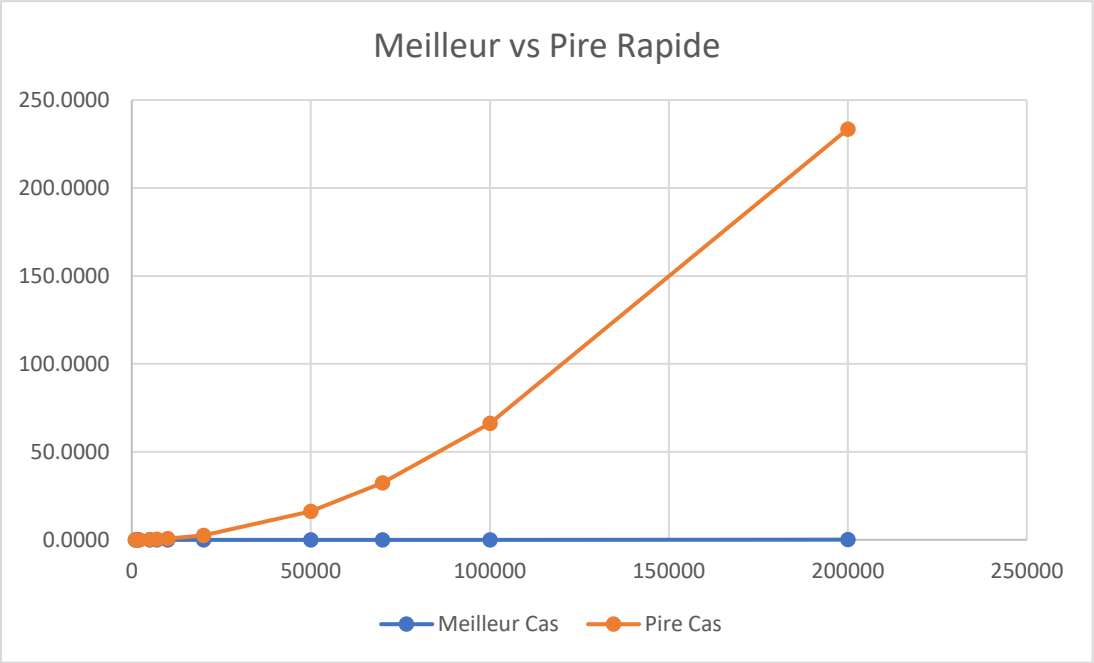
Comparaison expérimental :

Remarque :

Pour taille plus grande que 2000 , il a fallu rajouter l'espace a la pile du programme car le programme crash quand il n'arrive pas a faire autant d'appels récursives
Pour augmenter la taille du stack lors de la compilation

gcc comparer_pire_meilleur.c -o compare_pire_meill -Wl,--stack,104857600

Taille	1000	2000	5000	7000	10000	20000	50000	70000	100000	200000
Meilleur Cas	0.0000	0.0000	0.0010	0.0020	0.0040	0.0050	0.0140	0.0230	0.0390	0.0860
Pire Cas	0.0060	0.0230	0.1710	0.3050	0.6100	2.4870	16.0830	32.2890	66.1370	233.5010



3) Exemple d'exécution :

```
 tri Rapide :
avant tri
28  49  39  39  7  23  16  10  7  1  35  14  1  32  8
apres tri
1  1  7  7  8  10  14  16  23  28  32  35  39  39  49

 tri par tas :
```

Tri par Tas :

1) Implémentation de l'algorithme :

```
void tamiser(int *tab, int n, int i) {
    int largest = i;            //noeud
    int left = 2 * i + 1;    // fils gauche
    int right = 2 * i + 2;   // fils droit

    if (left < n && tab[left] > tab[largest])
        largest = left;

    if (right < n && tab[right] > tab[largest])     // permuter le noeud avec son
plus grand fils
        largest = right;

    if (largest != i) {
        permuter(tab , i , largest);

        tamiser(tab, n, largest);
    }
}

void tri_tas(int *tab, int n) {
    for (int i = n / 2 - 1; i >= 0; i--) // construire max tas
        tamiser(tab, n, i);

    // appliquer la suppression du tas sur le tableau
    for (int i = n - 1; i > 0; i--) {

        permuter( tab , 0 , i);
                                // permuter le max avec le dernier element
        tamiser(tab, i, 0);     // reduire le tas
    }
}
```

2) Complexité : $O(N \log_2 N)$

La complexité ne dépend pas de l'état initial du tableau car pour chaque tableau on doit créer le tas max et ensuite enchainer les suppressions

- Construction de tas max : N/2 itérations
- Suppression de l'élément racine $\log_2 N$ et vu que chaque suppression va se faire N – 1 fois donc $(N-1)(\log_2 N)$

$T(N) = \frac{N}{2} + (N-1)\log N = O(\,N \log N\,)$

3) Exemple d'exécution :

```
 tri par tas :
avant tri
15  1  8  7  23  3  17  31  28  23  27  10  27  27  35
apres tri
1  3  7  8  10  15  17  23  23  27  27  27  28  31  35
```

Etude Expérimentale :

On créer des tableaux de taille

$$[1000,2000,5000,7000,10000,20000,50000,70000,100000,200000]$$

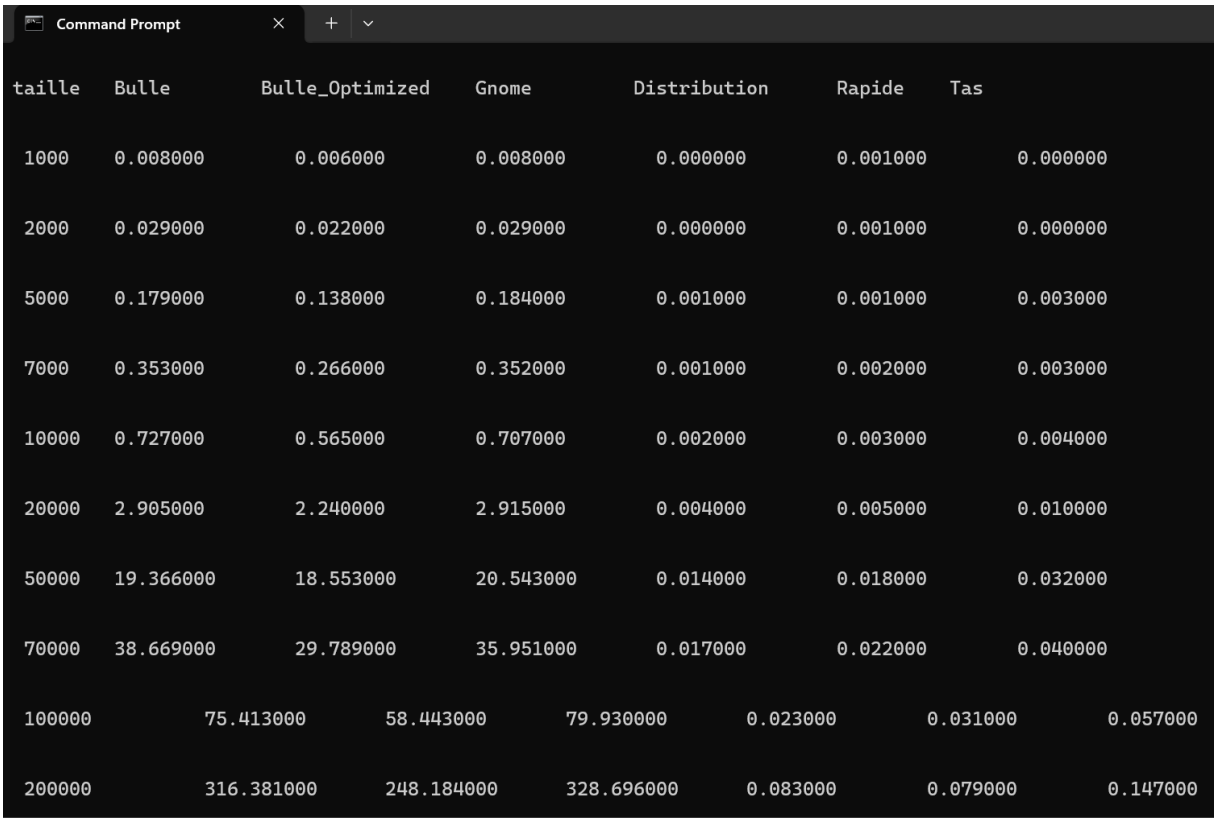
La taille N = 1000000 donne un temps d'exécution énorme (plus **d'une heure**) pour un algorithme de O(N2) et donc on l'avait remplacé par 200000

Les tableaux ont tous des valeurs aléatoires entre 100 et 100000000

On s'assure que le tableau initial est déjà trié en ordre décroissant avant d'appliquer les algorithmes de tri Bulle, Gnome

Pour l'algorithme de tri rapide, on s'assure aussi que le tableau est trié pour que le pivot divise en successions de N – 1

Le tri par base et le tri par tas ne dépend pas de l'ordre de tableau.



Command Prompt

taille	Bulle	Bulle_Optimized	Gnome	Distribution	Rapide	Tas
1000	0.008000	0.006000	0.008000	0.000000	0.001000	0.000000
2000	0.029000	0.022000	0.029000	0.000000	0.001000	0.000000
5000	0.179000	0.138000	0.184000	0.001000	0.001000	0.003000
7000	0.353000	0.266000	0.352000	0.001000	0.002000	0.003000
10000	0.727000	0.565000	0.707000	0.002000	0.003000	0.004000
20000	2.905000	2.240000	2.915000	0.004000	0.005000	0.010000
50000	19.366000	18.553000	20.543000	0.014000	0.018000	0.032000
70000	38.669000	29.789000	35.951000	0.017000	0.022000	0.040000
100000	75.413000	58.443000	79.930000	0.023000	0.031000	0.057000
200000	316.381000	248.184000	328.696000	0.083000	0.079000	0.147000

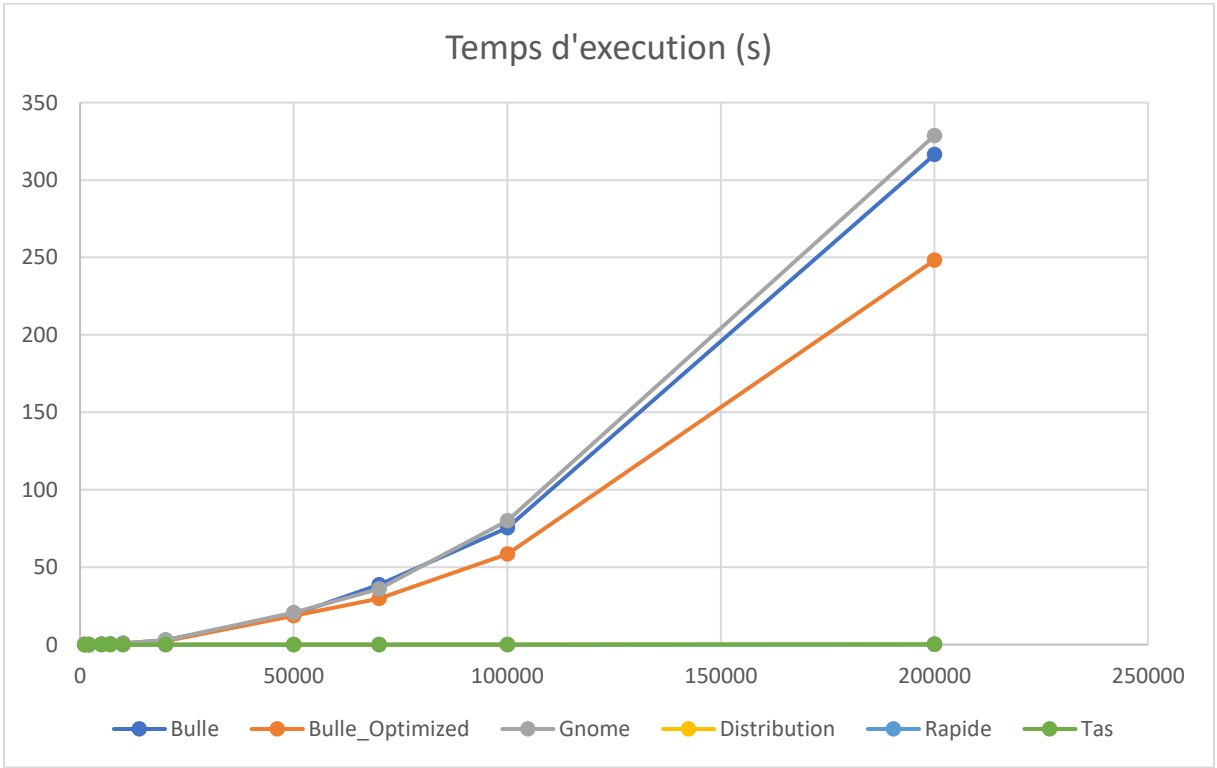
Rappel sur la complexité thorique :

Algorithm	Bulle	Bulle_opt	Gnome	Distribution	Rapide	Tas
Complexity	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(d * N)$	$O(N \log_2 N)$	$O(N \log_2 N)$

Calcul du temps d'execution en secondes :

taille	Bulle	Bulle_Optimized	Gnome	Distribution	Rapide	Tas
1000	0.008	0.006	0.008	0.000	0.001	0.000
2000	0.029	0.022	0.029	0.000	0.001	0.000
5000	0.179	0.138	0.184	0.001	0.001	0.003
7000	0.353	0.266	0.352	0.001	0.002	0.003
10000	0.727	0.565	0.707	0.002	0.003	0.004
20000	2.905	2.240	2.915	0.004	0.005	0.010
50000	19.366	18.553	20.543	0.014	0.018	0.032
70000	38.669	29.789	35.951	0.017	0.022	0.040
100000	75.413	58.443	79.930	0.023	0.031	0.057
200000	316.381	248.184	328.696	0.083	0.079	0.147

Représentation Graphique :



Analyse des résultats expérimental :

- Les tri Bulle, Gnome et Bulle optimisé sont les algorithmes les plus gourmands, avec une complexité quadratique, le temps d'exécution augmente très vite avec une taille plus grande

- Le tri Bulle optimisé est une petite amélioration, car ça diminue le nombre d'itérations mais sans changer l'ordre de complexité

- L'algorithme de tri par distribution est cas spécial, il est très optimal pour trier des entiers avec un nombre de chiffre qui est fixe, mais son inconvenant est que-il créer un tableau supplémentaire
  donc dans un environnement ou la mémoire central est limité, il est déconseillé de l'appliquer.

- Le tri par tas et le tri rapide, sont les deux d'une complexité d'ordre $O(N \log N)$, le temps d'exécution est très minime par rapport au tris quadratiques.

- Le tri rapide est un peu plus performant que tri par tas, mais il le faut un bon choix du pivot

  Les résultats montrent bien la validité des résultats théorique de calcul de complexité
  Les algorithmes ayant complexité $O(N^2)$ sont très gourmand en temps de calcul par rapport aux algorithmes $O(N \log N)$ et donc le passage d'un algorithme de tri d'une complexité quadratique a logarithmique peut optimiser énormément le temps d'exécution.

END 🧒