SECTION: MIV
GROUPE: 1

Rapport du Mini projet 1
Étude expérimentale des algorithmes de tri

Module : Algo

Binôme :

- ***Rayan Ibrahim BENATALLAH***
- ***Aya Celyne LAGAB***

Date : 19/11/2022

PARTIE 1:
ETUDE DES MÉTHODES DE TRI

1- Tri par insertion:

1.1 - sa description:

Dans cet algorithme, le principe est de parcourir la liste non triée (a1, a2, ... , an) en la décomposant en deux parties : une partie déjà triée et une partie non triée. La méthode est identique à celle que l'on utilise pour ranger des cartes. L'opération de base consiste à prendre l'élément frontière dans la partie non triée, puis à l'insérer à sa place dans la partie triée (place que l'on recherchera séquentiellement), puis à déplacer la frontière d'une position vers la droite. Ces insertions s'effectuent tant qu'il reste un élément à ranger dans la partie non triée.. L'insertion de l'élément frontière est effectuée par décalages successifs d'une cellule.

1.2 - son algorithme itératif:

```
Proc Tri_insertion (entier T[], entier N)
Début
    entier i,j,temp,stop;

    pour i de 1 à N
    faire
       j=i-1;
       stop=FALSE;
       tant que ((j>=0)&&(stop==FALSE))
       faire
         Si (T[j+1]>T[j])
         alors
            stop=TRUE;
         Fsi
         Sinon
         Dsinon
            temp=T[j];
            T[j]=T[j+1];
            T[j+1]=temp;
            j--;
         Fsinon

       fait
    fait
Fin
```

1.3 - sa complexité temporelle:

1.3.1 meilleur cas:

la complexité meilleur cas sera quand le tableau est déjà trié et ça veut dire le parcour uniquement donc c'est : **O(n)**

1.3.1 pire cas:

La complexité dans le pire cas est : **$O(n_2)$**

1.4 - sa complexité spatiale:

la complexité spatiale de l'algorithme tri insertion sera de : **O(1)**

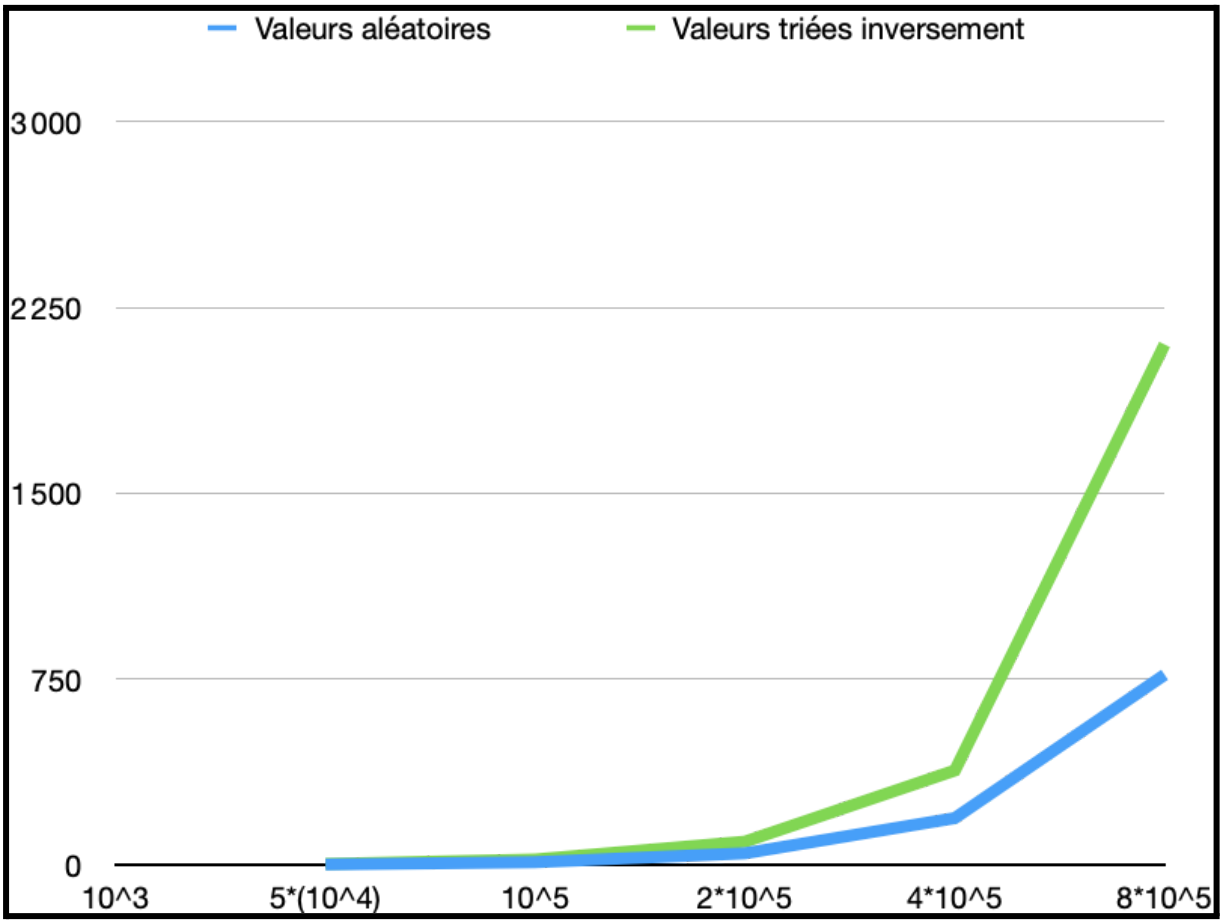
1.5 - son programme en c:

```
double Tri_insertion (int T[], int N)
{
      int i,j,temp,stop;
        double temps;
        clock_t start, stopclock;
        start = clock();
      for (i=1;i<N;i++)
      {
          j=i-1;
          stop=FALSE;
          while ((j>=0)&&(stop==FALSE))
          {
            if (T[j+1]>T[j])
            {
                stop=TRUE;
            }
            else
            {
                temp=T[j];
                T[j]=T[j+1];
                T[j+1]=temp;
                j--;
            }

          }
      }
      stopclock = clock();
      temps = (double)(stopclock - start)/CLOCKS_PER_SEC;
     return temps;
}
```

1.6 -

Tri par insertion	10^3	5*(10^4)	10^5	2*10^5	4*10^5	8*10^5
Valeurs aléatoires	0.002	3	12	48	190	768
Valeurs triées inversement	0.003	5	23	95	382	2100

1.7 - la représentation graphique des variations du temps d'exécution


Valeurs aléatoires
Valeurs triées inversement
3000
2250
1500
750
0
10^3
5*(10^4)
10^5
2*10^5
4*10^5
8*10^5

1.8 - comparaison entre la complexité théorique et expérimentale :

La différence entre la complexité théorique et expérimentale c'est que l'expérimentale est la pratique et en pratique, il est tout à fait possible qu'une solution beaucoup moins complexe s'exécute beaucoup plus rapidement (pour de "petites" entrées). Il est facile d'imaginer un programme qui tourne en O(n) en théorique mais qui nécessite 10^10*n itérations. C'est O(n) mais même une solution O(2^n) pourrait être plus rapide pour un petit n.

.

2- Tri à bulles:

1.1 - sa description:

Le tri à bulles est un algorithme de tri qui consiste à permuter de manière répétée les éléments adjacents s'ils sont dans le mauvais ordre. Cet algorithme n'est pas adapté aux grands ensembles de données car sa complexité temporelle moyenne et dans le pire des cas est assez élevée.

1.2 - son algorithme itératif:

```
Proc Tri_bulle (entier T[], entier N)
Début
   entier i,temp,x;
   entier permut;

   x=N;
   repeter
      x=x-1;
      permut=Faux;

      pour i de 0 à x
      faire
         Si (T[i]>T[i+1])
         alors
            temp=T[i];
            T[i]=T[i+1];
            T[i+1]=temp;
            permut=Vrai;
         Fsi
      fait
   jusqu'à (permut==Vrai);
Fin
```

1.3 - sa complexité temporelle:

1.3.1 meilleur cas:

Le meilleur cas est quand on a un tableau déjà trié, on va parcourir une seule fois donc on aura : **O(n)**

1.3.1 pire cas:

c'est quand le tableau est trié dans un ordre décroissant, à chaque fois on doit effectuer un déplacement, donc l'élément qui se trouve dans la position i va prendre la position n-i, et donc la complexité sera : **O(n^2)**

1.4 - sa complexité spatiale:

Vu qu'on a un tableau des entiers donc la complexité spatiale va être de : 4n , tel que n est la taille du tableau et 4 est le nombres de bits dans la mémoire pour l'entier. donc la valeur de sa complexité spatiale est : **O(n)**

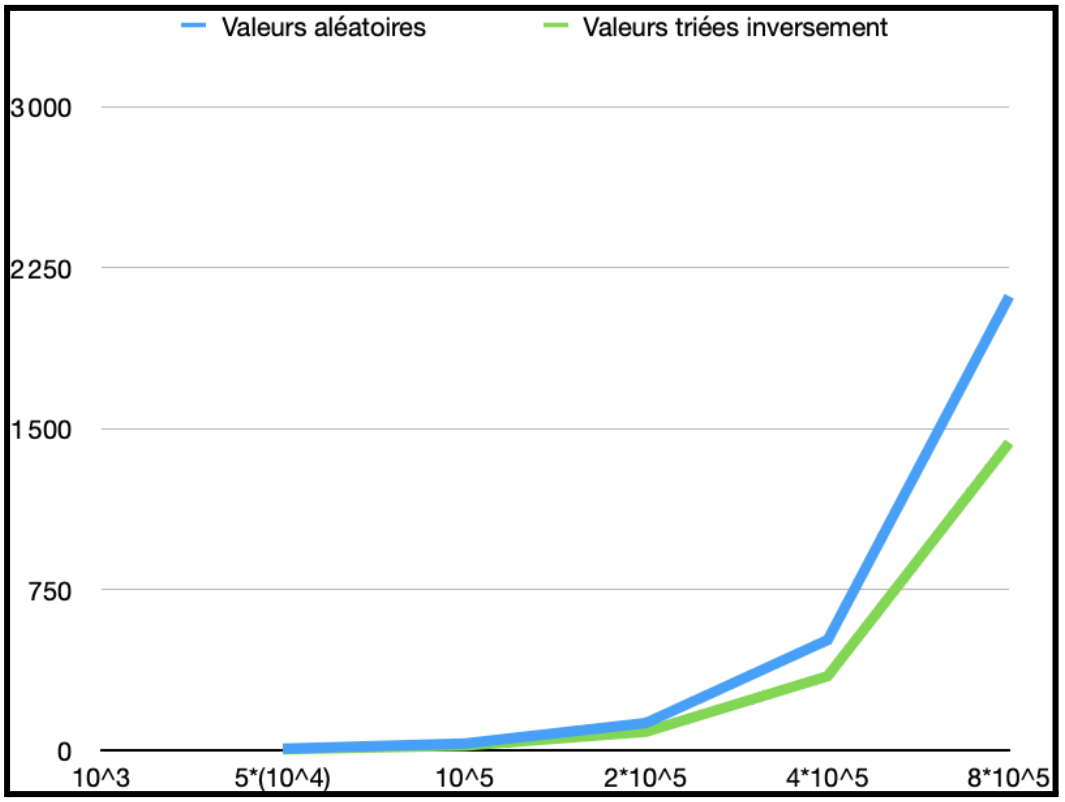
1.5 - son programme en c:

```c
double Tri_bulle (int T[], int N)
{
   int i,temp,x;
   int permut;
   double temps;
   clock_t start, stop;
   x=N;
   start = clock();
   do
   {
        x=x-1;
        permut=FALSE;

        for (i=0;i<x;i++)
        {
            if (T[i]>T[i+1])
            {
                temp=T[i];
                T[i]=T[i+1];
                T[i+1]=temp;
                permut=TRUE;
            }
        }
   }while (permut==TRUE);
   stop = clock();
   temps = (double)(stop-start)/CLOCKS_PER_SEC;
   return temps;
}
```

1.6 -

Tri a bulle	10^3	5*(10^4)	10^5	2*10^5	4*10^5	8*10^5
Valeurs aléatoires	0.002519	8	32	128	515	2117
Valeurs triées inversement	0.003494	5	22	87	346	1435

1.7 - la représentation graphique des variations du temps d'exécution



1.8 - comparaison entre la complexité théorique et expérimentale

3- Tri fusion:

1.1 - sa description:

L'algorithme Tri fusion est un algorithme de tri basé sur le paradigme Diviser et Conquérir. Dans cet algorithme, le tableau est d'abord divisé en deux moitiés égales, puis elles sont combinées de manière triée.

1.2 - son algorithme itératif:

```
Proc fusion(entier tableau[],entier deb1,entier fin1,entier fin2)
    Début
    entier *table1;entier deb2=fin1+1; entier compt1=deb1;entier
compt2=deb2;entier i;
   Allouer(table1)
    Pour i de deb1 à fin1
      Faire
      table1[i-deb1]=tableau[i];
      Fait
    Pour i de deb1 à fin2
      Faire
      Si (compt1==deb2)
        alors
        break;
        Fsi
      Sinon Si (compt2==(fin2+1))
        alors
        tableau[i]=table1[compt1-deb1];
        compt1++;
        Fsi
      Sinon Si (table1[compt1-deb1]<tableau[compt2])
        alors
        tableau[i]=table1[compt1-deb1];
        compt1++;
        Fsi
      Sinon
        Dsinon
        tableau[i]=tableau[compt2];
        compt2++;
        Fsinon
      Fait
    Liberer(table1);
    Fin

Proc tri_fusion_bis(entier tableau[],entier deb,entier fin)
    Début
    Si (deb!=fin)
       alors
      entier milieu=(fin+deb)/2;
      tri_fusion_bis(tableau,deb,milieu);
      tri_fusion_bis(tableau,milieu+1,fin);
      fusion(tableau,deb,milieu,fin);
      Fsi
```

```
        Fin

    Proc tri_fusion(entier tableau[],entier longueur)
      Début
      Si (longueur>0)
          Dsi
          tri_fusion_bis(tableau,0,longueur-1);
          Fsi
      Fin
```

1.3 - sa complexité temporelle:

1.3.1 meilleur cas:

le meilleur cas c'est quand l'algorithme est déjà trié et donc c'est : **O(n*log n)**

1.3.1 pire cas:

sera le même que le meilleur cas ou le cas normal car c'est toujours la même chose à faire **O(n*log n)**

1.4 - sa complexité spatiale:

la complexité spatiale de l'algorithme fusion est 4n donc **O(n)**

1.5 - son programme en c:

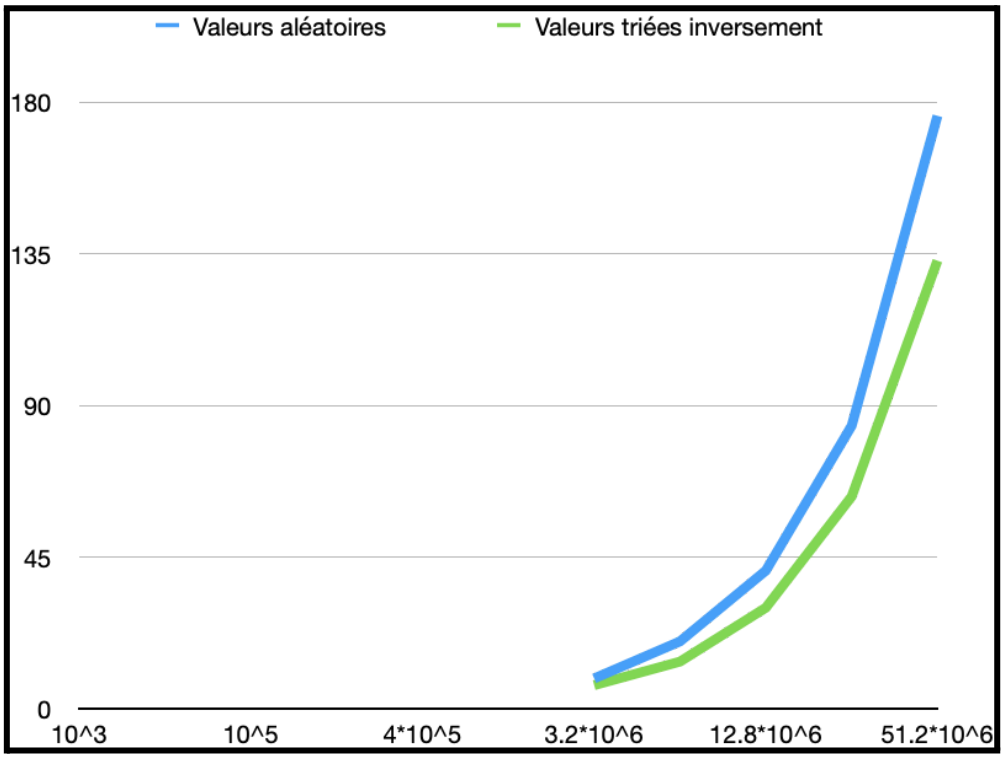
```c
void fusion(int tableau[],int deb1,int fin1,int fin2)
{

        int *table1;
        int deb2=fin1+1;
        int compt1=deb1;
        int compt2=deb2;
        int i;
        table1=malloc((fin1-deb1+1)*sizeof(int));



        for(i=deb1;i<=fin1;i++)
            {
            table1[i-deb1]=tableau[i];
            }


        for(i=deb1;i<=fin2;i++)
            {
            if (compt1==deb2)
                {
                break;
                }
            else if (compt2==(fin2+1))
```

```
                {
                tableau[i]=table1[compt1-deb1];
                compt1++;
                }
            else if (table1[compt1-deb1]<tableau[compt2])
                {
                tableau[i]=table1[compt1-deb1];
                compt1++;
                }
            else
                {
                tableau[i]=tableau[compt2];
                compt2++;
                }
            }
        free(table1);

}


void tri_fusion_bis(int tableau[],int deb,int fin)
        {
        if (deb!=fin)
            {
            int milieu=(fin+deb)/2;
            tri_fusion_bis(tableau,deb,milieu);
            tri_fusion_bis(tableau,milieu+1,fin);
            fusion(tableau,deb,milieu,fin);
            }
        }

double tri_fusion(int tableau[],int longueur)
    {
        double temps;
        clock_t start, stop;
        start = clock();
        if (longueur>0)
            {
            tri_fusion_bis(tableau,0,longueur-1);
            }
        stop = clock();
        temps=(double)(stop-start)/CLOCKS_PER_SEC;
        return temps;
    }
```

1.6 -

Tri fusion	10^3	5*(10^4)	10^5	2*10^5	4*10^5	8*10^5	3.2*10^6	6.4*10^6	12.8*10^6	25.6*10^6	51.2*10^6
Valeurs aléatoires	0.0006	0.14	0.02	0.05	0.11	0.22	9	20	41	84	176
Valeurs triées inversement	0.0003	0.11	0.01	0.04	0.07	0.16	7	14	30	63	133

1.7 - la représentation graphique des variations du temps d'exécution



1.8 - comparaison entre la complexité théorique et expérimentale

4- Tri rapide (quick sort):

1.1 - sa description:

Le processus clé du Tri rapide est une partition. L'objectif des partitions est, étant donné un tableau et un élément x d'un tableau comme pivot, de mettre x à sa position correcte dans un tableau trié et de mettre tous les éléments plus petits (plus petits que x) avant x, et de mettre tous les éléments plus grands (plus grands que x) après x. Tout ceci doit être fait en temps linéaire.

1.2 - son algorithme itératif:

```
Proc Tri_rapide (entier T[], entier N)
{
   entier Binf,Bsup,i,pivot,p_pivot;
   Pile P;
   CreerPile (P);
   Empiler(P,0);
   Empiler (P,N-1);

   Tant que (Pile(P) n'est pas vide)
   {
      Depiler (P, Bsup);
      Depiler (P, Binf);
      pivot=T[Bsup];
      p_pivot=Binf;
      Pour (i de Binf à Bsup-1)
      Faire
         Si (T[i]<=pivot)
         alors
            permut(&T[i],&T[p_pivot]);
            p_pivot++;
         Fsi
      Fait
      permut(&T[p_pivot], &T[Bsup]);
      Si (p_pivot+1 <Bsup)
      alors
         Empiler(P,p.pivot+1);
         Empiler(P,Bsup);
      Fsi
      Si (p_pivot-1>Binf)
      alors
         Empiler(P,Binf);
         Empiler(P,p.pivot-1);

      Fsi
   Fait
Fin
```

1.3 - sa complexité temporelle:

1.3.1 meilleur cas:

meilleur cas c'est quand le tableau est déjà trié et c'est : **O(n*log n)**

1.3.1 pire cas: la complexité du pire cas dans le tri rapide est **O(n2)**

1.4 - sa complexité spatiale: la complexité spatiale dans ce cas sera **O(log n)**

1.5 - son programme en c:

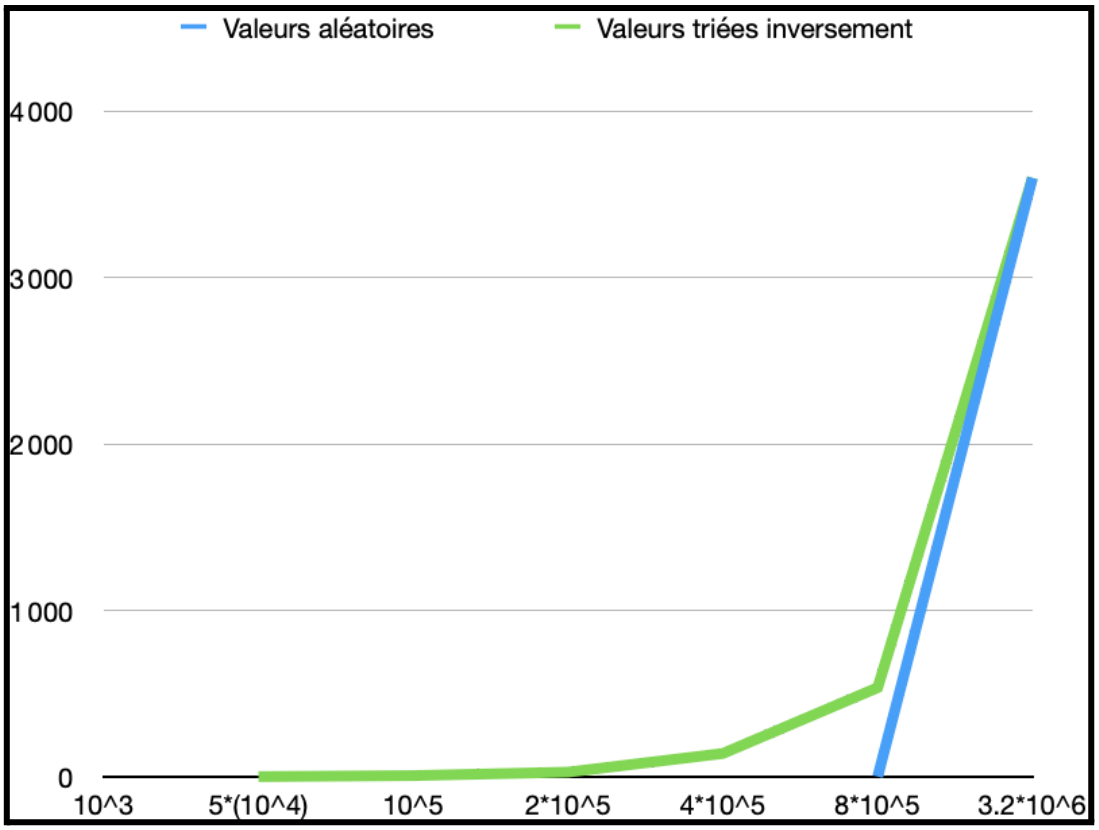
```
double Tri_rapide (int T[], int N)
{
   double temps;
   clock_t start, stop;
   int Binf,Bsup,i,pivot,p_pivot;
   TypePile P;
   start = clock();
   CreerPile (&P);
   Empiler(&P,0);
   Empiler (&P,N-1);

   while (!PileVide(P))
   {
       Depiler (&P, &Bsup);
       Depiler (&P, &Binf);
       pivot=T[Bsup];
       p_pivot=Binf;
       for (i=Binf; i<=Bsup-1; i++)
       {
           if (T[i]<=pivot)
           {
               permut(&T[i],&T[p_pivot]);
               p_pivot++;
           }
       }
       permut(&T[p_pivot], &T[Bsup]);
       if (p_pivot+1 <Bsup)
       {
           Empiler(&P,p_pivot+1);
           Empiler(&P,Bsup);
       }
       if (p_pivot-1>Binf)
       {
           Empiler(&P,Binf);
           Empiler(&P,p_pivot-1);

       }
```

```
    }
    stop = clock();
    temps=(double)(stop-start)/CLOCKS_PER_SEC;

    return temps;
}
```

1.6 -

Tri rapide	10^3	5*(10^4)	10^5	2*10^5	4*10^5	8*10^5	3.2*10^6
Valeurs aléatoires	0.000797	0.028089	0.027118	0.184504	0.6	2	3600
Valeurs triées inversement	0.000797	2	8	29	141	537	3600

1.7 - la représentation graphique des variations du temps d'exécution



1.8 - comparaison entre la complexité théorique et expérimentale

1- Tri par tas (heap sort):

1.1 - sa description:

Le tri par tas est une technique de tri par comparaison basée sur la structure de données du tas binaire. Il est similaire au tri sélectif où nous trouvons d'abord l'élément minimum et le plaçons au début. Répétez le même processus pour les éléments restants.

1.2 - son algorithme itératif:

```
Proc swap(entier *a, entier *b)
Début
   entier temp = *a;
   *a = *b;
   *b = temp;
Fin

 Proc retablir_tas(entier arr[], entier n, int i)
Début
   entier largest = i;
   entier left = 2 * i + 1;
   entier right = 2 * i + 2;

   Si (left < n et arr[left] > arr[largest])
     largest = left;

   Si (right < n et  arr[right] > arr[largest])
     largest = right;

   Si (largest != i) alors
     swap(&arr[i], &arr[largest]);
     retablir_tas(arr, n, largest);
   Fsi
 Fin
//fonction Tri Tas
 Proc faireTas(entier arr[], entier n)
 Début
   Pour  i de n / 2 - 1 à 0; i--)
     retablir_tas(arr, n, i);

   // Tri Tas
   Pour (int i de n - 1 à 0; i--)
   Faire
     swap(arr[0], arr[i]);
     retablir_tas(arr, i, 0);
   Fait
 Fin.
```

1.3 - sa complexité temporelle:

1.3.1 meilleur cas:

la complexité en cas meilleur sera : **O(nlog n)**

1.3.1 pire cas:

la complexité du pire sera la même que le cas meilleur est donc : **O(nlog n)**

1.4 - sa complexité spatiale: la complexité spatiale de l'algorithme sera de : **O(1)**

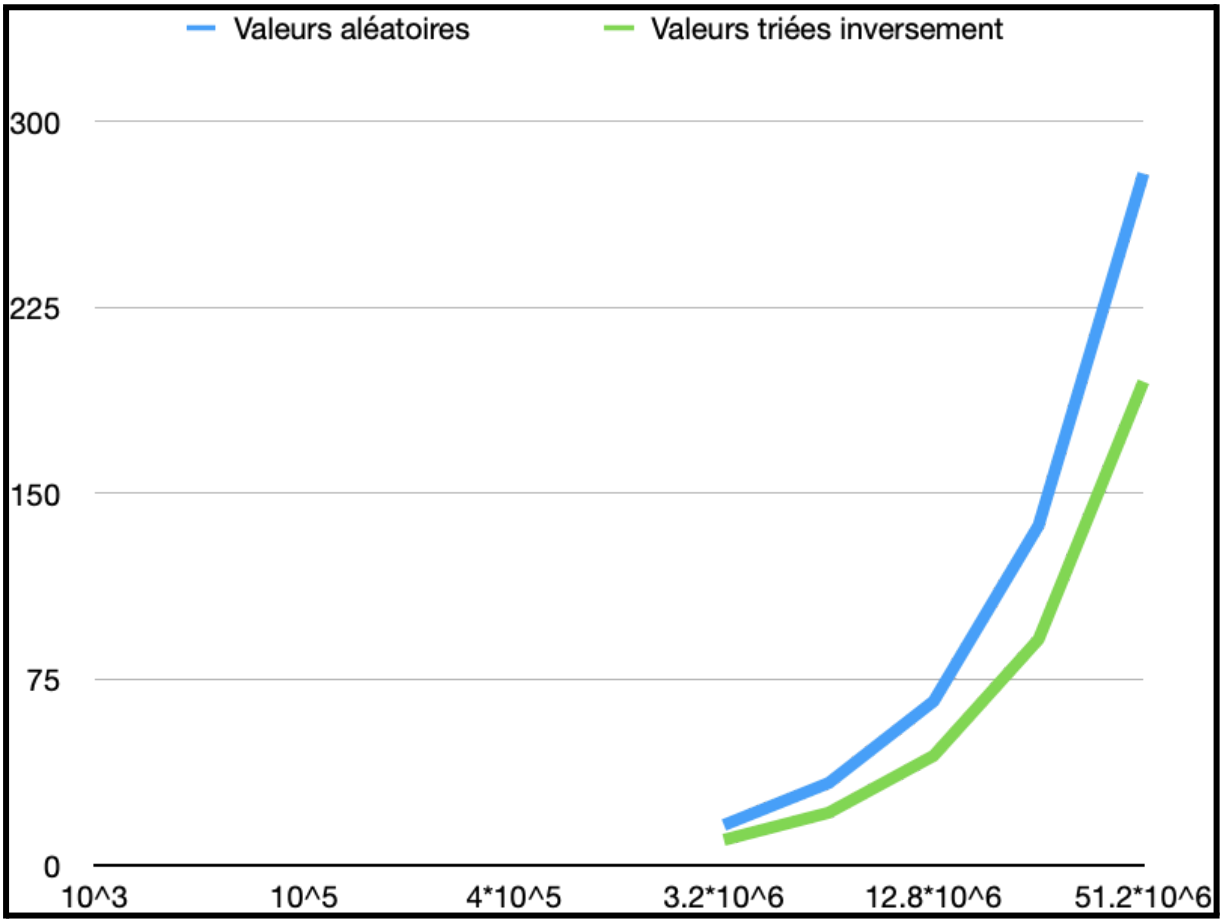
1.5 - son programme en c:

```c
void swap(int *a, int *b)
 {
   int temp = *a;
   *a = *b;
   *b = temp;
 }
 void retablir_tas(int arr[], int n, int i)
 {
   // Find largest among root, left child and right child
   int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
     largest = left;
    if (right < n && arr[right] > arr[largest])
     largest = right;
    // Swap and continue heapifying if root is not largest
   if (largest != i) {
     swap(&arr[i], &arr[largest]);
     retablir_tas(arr, n, largest);
   }
 }
  // Main function to do heap sort
double faireTas(int arr[], int n)
 {
   double temps;
   clock_t start, stop;
   start = clock();
   // Build max heap
   for (int i = n / 2 - 1; i >= 0; i--)
     retablir_tas(arr, n, i);
    // Heap sort
   for (int i = n - 1; i >= 0; i--) {
     swap(&arr[0], &arr[i]);
      // Heapify root element to get highest element at root again
     retablir_tas(arr, i, 0);
   }
   stop =clock();
   temps = (double)(stop-start)/CLOCKS_PER_SEC;
   return temps;
 }
```

1.6 -

Tri par tas	10^3	5*(10^4)	10^5	2*10^5	4*10^5	8*10^5	3.2*10^6	6.4*10^6	12.8*10^6	25.6*10^6	51.2*10^6
Valeurs aléatoires	0.0004	0.019	0.03	0.06	0.13	0.27	16	33	66	137	279
Valeurs triées inversement	0.0003	0.01	0.022	0.04	0.09	0.20	10	21	44	91	195

1.7 - la représentation graphique des variations du temps d'exécution



1.8 - comparaison entre la complexité théorique et expérimentale

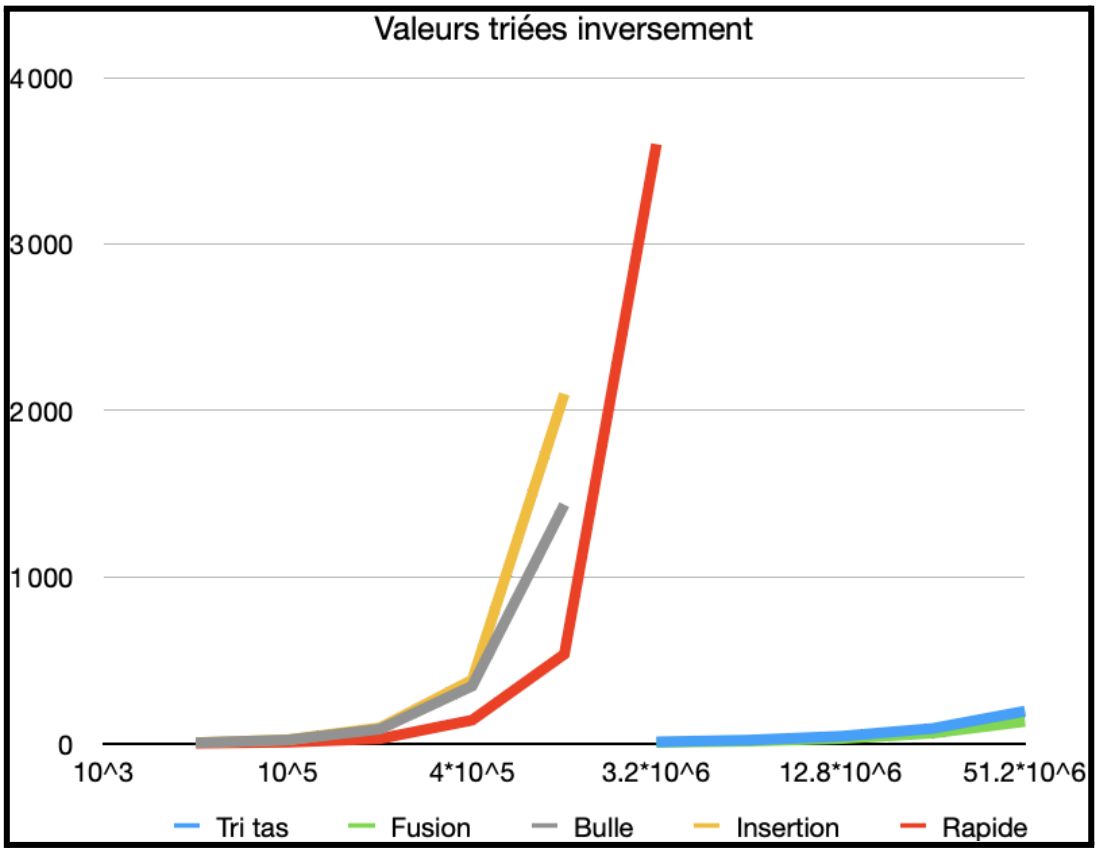
PARTIE 2: COMPARAISON DES MÉTHODES DE TRI

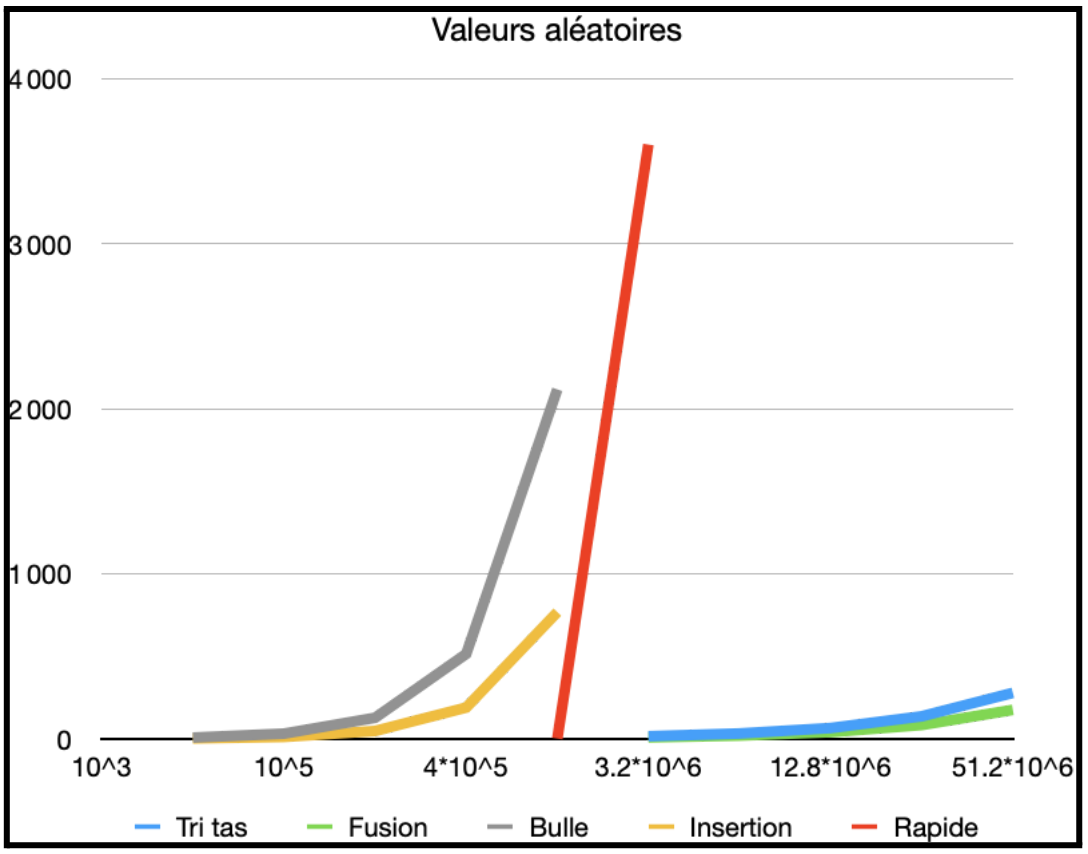
2.1 - représentation graphique des complexité théorique et expérimentale des 5 algorithmes

2.2 - Tableau comparatif:

	Complexité	10^3	5*(10^4)	10^5	2*10^5	4*10^5	8*10^5	3.2*10^6	6.4*10^6	12.8*10^6	25.6*10^6	51.2*10^6
Tri tas	O(nlog n)	0.0003	0.01	0.022	0.04	0.09	0.20	10	21	44	91	195
Fusion	O(n*log n)	0.0003	0.11	0.01	0.04	0.07	0.16	7	14	30	63	133
Bulle	O(n^2)	0.003494	5	22	87	346	1435	/				
Insertion	O(n^2)	0.003	5	23	95	382	2100	/				
Rapide	O(n2)	0.000797	2	8	29	141	537	3600	/			

2.3 - Comparaison graphique entre les 5 algorithmes de tri:





ANNEXE

Fonction pour générer les nombres aléatoirement:

```c
#define NB_MAX  1000
#define NB_MIN  1
void Gen_Tab(int T[],int N)
{
   int i;
   for(i=0; i<N; i++)
   {
       T[i] = rand()%(NB_MAX-NB_MIN)+NB_MIN;
   }
}
```

Fonction main:

```c
int main()
{
   int N,i,j;
   double temps;
   srand(time(NULL));
   printf("DONNEZ UNE TAILLE POUR LE TABLEAU QUE VOUS VOULEZ TRIER:");
   scanf("%d", &N);
   int * T= (int*)malloc(N*sizeof(int));
   /*tableau trié inversement*/
   int * Tr= (int*)malloc(N*sizeof(int));
   /*tableau trié*/
   int * TT= (int*)malloc(N*sizeof(int));
   Gen_Tab(T,N);
   temps = tri(T,N);
   printf("Temps d'execution 1 (Valeurs Aléatoires): %f \n", temps);
   /*construire le tableau des valeurs triées inversement*/
   for (i = N-1, j=0; i >= 0; i--, j++) Tr[j] = T[i];
   /*construire un nouveau tableau trié*/
   for(i=0; i<N; i++) TT[i] = T[i];
   temps = tri(Tr,N);
   printf("\nTemps d'execution 2 (Valeurs triées inversement): %f \n", temps);
   temps= tri(T,N);
   printf("\nTemps d'execution 3 (Valeurs triées): %f", temps);
   printf("\n\n");
   return 0;
}
```