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

BOUMAZA MUSTAPHA MOUNIR

AMELLAL YACINE

❖ Introduction

A sorting algorithm is an algorithm that organizes a collection of objects according to a specified order relation. The objects to be sorted are elements of a set with a total order. It is common, for example, to sort integers according to the usual 'less than or equal to' order relation. Sorting algorithms are used in many situations, particularly in more complex algorithms, including some search algorithms like binary search. They can also be used to put data in canonical form or make it more readable for the user. The collection to be sorted is often given in the form of an array to allow direct access to different elements of the collection or in the form of a list, which may be more suitable for certain algorithms and functional programming.

The classification of sorting algorithms is crucial as it helps choose the most suitable algorithm for the given problem, taking into account the constraints imposed by it. The main characteristics that differentiate sorting algorithms, besides their operating principle, are time complexity, space complexity, and stability."

In our work, we focus on these five algorithms:

- **Bubble sort**
- **Gnome sort**
- **Radix sort**
- **QuickSort**
- **Heap Sort**

❖ WORKING SPACE :

Laptop	HP ELITEBOOK
CPU	Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz 1.90 GHz
RAM	8,00 Go
Système	Windows 10 Pro 64 Bit

1. Bubble sort

The bubble sort, or bubble sort algorithm, is a sorting algorithm. It involves repeatedly comparing consecutive elements in an array and swapping them when they are out of order. It gets its name because it quickly moves the largest elements to the end of the array, similar to air bubbles rising quickly to the surface of a liquid.

- **Algorithm :**

```
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;
```

- **Algorithm in C :**

```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;
            }

        }
    }
}
```

- **Analysis and Complexity**

- Best case:

```
void TriBulle (int T[], int n)
{
  int Changement , i, p;
  Changement = 0 ---------- 1
  while (Changement == 0) { ---------- 2
      Changement = 1; ---------- 1
      for (i=1; i<n; i++) { ---------- n-1
          if(T[i] > T[i+1]) { ---------- n-1
              p = T[i]; ---------- 0
              T[i] = T[i+1]; ---------- 0
              T[i+1] = p; ---------- 0
              Changement = 0; ---------- 0
          }
      }
  }
}
```

The array is already sorted, the bubble sort algorithm will perform only one iteration, hence O(n).

$$F1(n)= \sum_{i=1}^{n-1} 1 \quad = \quad n-1 = O(n)$$

In this case we have :

$$F1(n)=1+2+1+n-1+n-1 = 2n+2$$
$$F1(n)=2n+2$$

In asymptotic notation :

F1(n)=O(n)

- **Worst case:**

```
void TriBulle (int T[], int n)
{
  int Changement , i, p;
  Changement = 0 ---------- 1
  while (Changement == 0) { ---------- n
      Changement = 1; ---------- n-1
      for (i=1; i<n; i++) { ---------- (n-1)²
          if(T[i] > T[i+1]) { ---------- (n-1)²
              p = T[i]; ---------- (n-1)²
              T[i] = T[i+1]; ---------- (n-1)²
              T[i+1] = p; ---------- (n-1)²
              Changement = 0; ---------- (n-1)²
          }
      }
  }
}
```

Descending order array, the outer loop will be executed (n-1) times:

$$F2(n)= \sum_{i=1}^{n} \sum_{i=1}^{n-1} 1 = \sum_{i=1}^{n} (n-1) = \frac{1}{2} n * (n-1) = \frac{1}{2} n^2 - \frac{1}{2} n = O(n^2)$$

In this case we have :

$$F2(n)=\sum_{i=1}^{n-1} \sum_{i=1}^{n-1} (6) + (n-1) + (n+1) = 6(n+1)^2 + 2n = 6n^2 - 10n + 6$$

F2(n)= $6n^2$ - 10n + 6

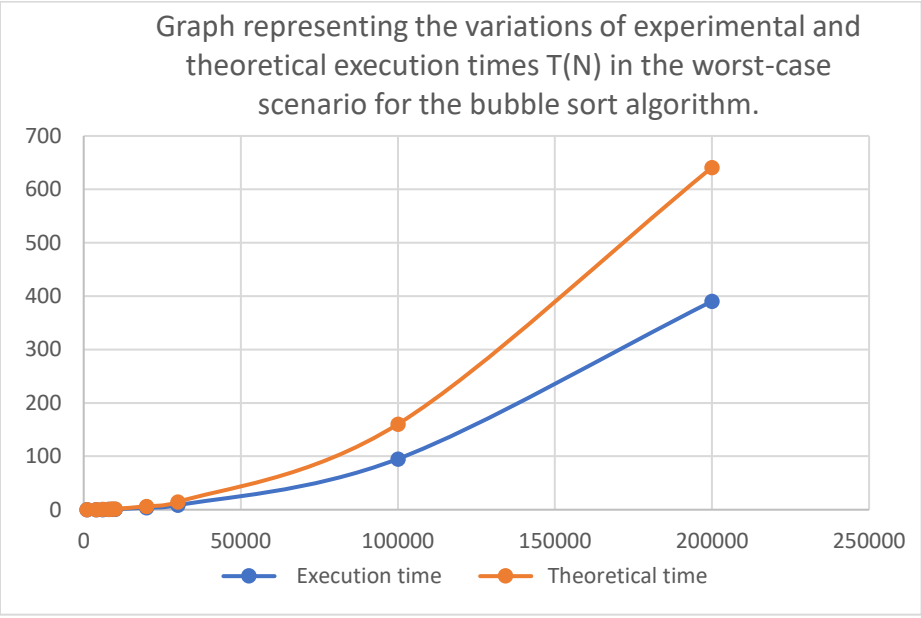
In asymptotic notation :

F2(n)=O(n^2)

- **Experimental Time :**

N	Execution time	Theoretical time
1000	0,016	0,016
4000	0,156	0,256320374
6000	0,327	0,576800961
8000	0,577	1,025495152
9000	0,749	1,297922349
10000	0,951	1,602402947
20000	3,744	6,410145981
30000	8,3	14,42322911
100000	95,146004	160,2643337
200000	390,324005	641,0626771
	dt	2,67112E-09

- **Graphical Representation :**



- **Optimized Algorithm** :

```
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;
```

- **Optimized Algorithm in 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--;
    }
}
```

- **Analysis and Complexity**

- **best case:**

```
void TriBulleOpt (int T[], int n)
{
        int Changement , i, p, m;
        m=n-1; ---------------------------------------------------------- 1
       Changement = 0   ---------------------------------------------------- 1
        while (Changement == 0) {  ------------------------------------- 2
                Changement = 1;  ----------------------------------------- 1
                for (i=1; i<m; i++) {  ---------------------------------- n-1
                        if (T[i] > T[i+1]) { ------------------------------ n-1
                                p = T[i];
                                T[i] = T[i+1];                          0 ( zéro échange)

                                T[i+1] = p;
                                Changement = 0;
                        }
                }
                m=m-1; ------------------------------------------------- 1
}
```

The array is already sorted; the bubble sort algorithm will perform only one iteration, hence O(n).

$$F1(n)= \sum_{i=1}^{n-1} 1 \;=\; n-1 = O(n)$$

In this case we have :

$$F1(n)= 1+1+2+1+ 2(n-1) +1 = 2n+4$$
$$F1(n)= 2n+4$$

In asymptotic notation :

F1(n)=O(n)

worst case:

```
void TriBulleOpt (int T[], int n)
{
        int Changement , i, p, m;
        m=n-1; ---------- 1
       Changement = 0  ---------- 1
        while (Changement == 0) {  ---------- n
                Changement = 1;  ---------- n-1
                for (i=1; i<m; i++) {
                        if (T[i] > T[i+1]) {
                                p = T[i];                    6*n(n-1)/2
                                T[i] = T[i+1];
                                T[i+1] = p;
                                Changement = 0;
                        }
                }
                m=m-1;  ---------- n-1
        }
}
```

Descending order array; the outer loop will be executed (n-1) times, and the inner loop will be executed (n-m) times.

$$F2(n) = 1 + 2 + 3 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{1}{2} n * (n-1) = \frac{1}{2} n^2 - \frac{1}{2} n = O(n^2)$$

In this case we have :

$$F2(n) = 1+1+n+n-1+ 3n(n-1) + n-1 + (n-1) = 3n^2$$
$$F2(n) = 3n^2$$

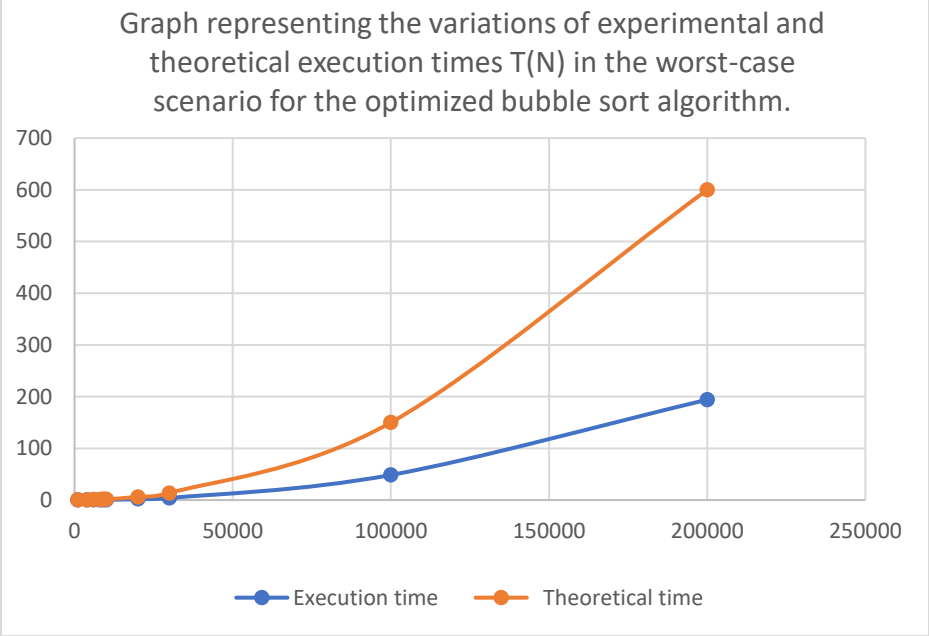
In asymptotic notation :

F2(n)=O(n²)

- **Experimental Time :**

N	Execution time	Theoretical time
1000	0,015	0,015
4000	0,078	0,24
6000	0,187	0,54
8000	0,296	0,96
9000	0,374	1,215
10000	0,453	1,5
20000	1,763	6
30000	3,978	13,5
100000	48,485001	150
200000	194,102005	600
	dt	0,000000005

- **Graphical Representation:**

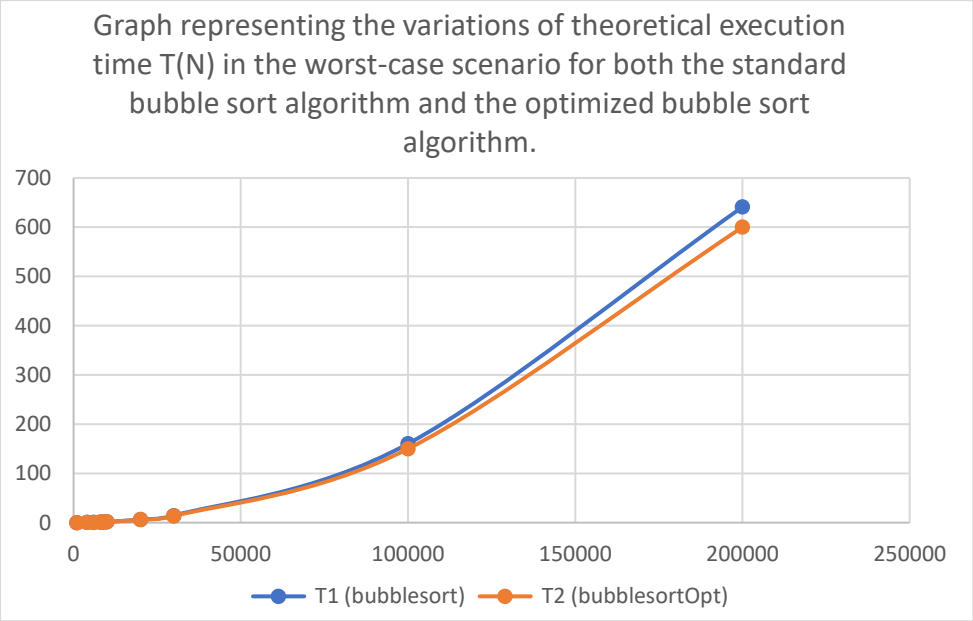


- **Comparison between the two algorithms**

- **Theoretical time:**

N	T1 (bubblesort)	T2 (bubblesortOpt)
1000	0,016	0,015
4000	0,256320374	0,24
6000	0,576800961	0,54
8000	1,025495152	0,96
9000	1,297922349	1,215
10000	1,602402947	1,5
20000	6,410145981	6
30000	14,42322911	13,5
100000	160,2643337	150
200000	641,0626771	600

- **Graphical Representation:**



In terms of execution time, the two algorithms have nearly identical performance, with the latter being slightly faster.

2. Gnome sort

The algorithm is similar to insertion sort, except that instead of directly inserting the element into its correct position, the algorithm performs a series of swaps, similar to a bubble sort. 'Gnome' because it is said to be the method used by Dutch garden gnomes to sort a row of flower pots from the smallest flower to the largest...

- **Algorithm :**

```
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;
```

- **Algorithm in 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--;
        }
    }
}
```

- **Analysis and Complexity**

- **Best case:**

```
void TriGnome (int T[], int n)
{
  int i, p;
  i=1 ---------------------------------------------------------------------- 1
  while (i < n) { ---------------------------------------------------------- n
      if (T[i] <= T[i+1]) ---------------------------------------------- n-1
          i++; ----------------------------------------------------------- n-1
      else {                                                                  } 0 comparasion = 0 échange
          p = T[i];                                                           |
          T[i] = T[i+1];                                                      |
          T[i+1] = p;                                                         |
          if (i == 0) i++;                                                    |
          else i--;                                                           |
          }
      }
}
```

If the array is already sorted, the algorithm will perform n comparisons and thus O(n).

$$F1(n)= \sum_{i=1}^{n-1} 1 \;=\; n-1 \;= O(n)$$

- **worst case:**

```
void TriGnome (int T[], int n)
{
  int i, p;
  i=1  ---------- 1
  while (i < n) { ---------- n²
      if (T[i] <= T[i+1]) ---------- n
          i++;  ---------- 0
      else {
          p = T[i]; ---------- 0
          T[i] = T[i+1]; ---------- 0
          T[i+1] = p; ---------- 0
          if (i == 0) i++; ---------- 0
          else i--; ---------- 0
          }
      }
}
```

In a descending order array, for each element at index i of the array, the algorithm will perform i swaps :

$$F2(n)= \sum_{i=1}^{n} \sum_{i=1}^{n-1} 1 = \sum_{i=1}^{n} (n-1) = \frac{1}{2} n * (n-1) = \frac{1}{2} n^2 - \frac{1}{2} n = O(n^2)$$

In this case we have :

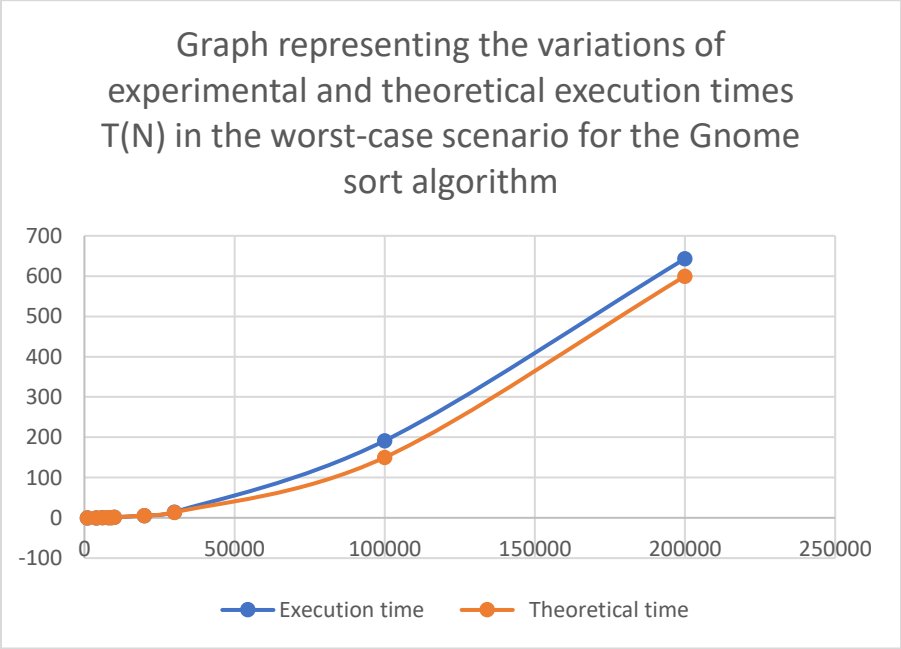
In asymptotic notation :

F2(n)=O(n²)

- **Experimental Time:**

Size	Execution time	Theoretical time
1000	0,015	0,014975015
4000	0,243	0,239900015
6000	0,528	0,539850015
8000	0,957	0,959800015
9000	1	1,0935
10000	2	1,499750015
20000	6	5,999500015
30000	15	13,49925002
100000	192	149,9975
200000	643	599,995
DT :	2,5E-09	

- **<u>Graphical Representation</u>** :



- **conclusion** :

  After studying the algorithm corresponding to the Gnome sort method, we conclude that this method has quadratic complexity in the worst case and linear complexity in the best case. Therefore, it is a slow method and very costly in terms of execution time and memory space. However, it is more efficient than the bubble sort method and less efficient than the optimized bubble sort method.

3. Radix sort

Base sort, or Radix sort, is a sorting algorithm used to arrange elements identified by a unique key. Each key is a string of characters or a number that the base sort arranges in lexicographical order. This algorithm needs to be paired with one or more stable sorting algorithms. In this case, we used the counting sort algorithm.

- **Algorithm :**

```
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 in C:**

```
47
48  int cle(int i, int j){
49
50      int k1 = pow(10, j+1);
51      int k2 = k1/10;
52
53      int t = ((i % k1) - (i % k2))/k2;
54
55      return t;
56  }
```

```
59  void TriAux(int *T, int n , int i)
60  {
61    int *T1=(int *)malloc(n*sizeof(int));
62    int *T2=(int *)malloc(n*sizeof(int));
63    int l=0;
64    for(int j=0;j<n;j++){
65      T1[j]=cle(T[j],i);
66    }
67
68    for(int j=0;j<10;j++)
69      {
70        for(int k=0;k<n;k++)
71        {
72          if (T1[k] == j){
73            T2[l]=T[k];
74            l++;
75          }
76        }
77      }
78      for(int k=0;k<n;k++){
79        T[k]=T2[k];
80      }
81  }
```

```
82  void TriBase(int* T, int n, int k){
83      for(int i=0; i<k; i++){
84          TriAux(T, n, i);
85
86      }
87
88  }
```

- **Analysis and Complexity :**

  The cle function is of order O(1).

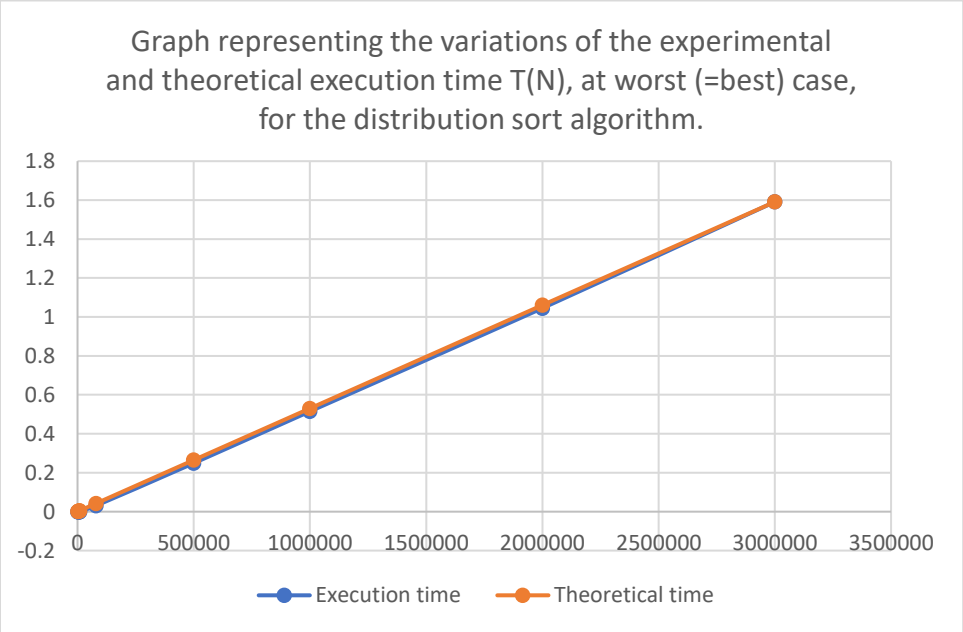
  The TriAux procedure runs in linear time O(n).

  The TriAux procedure will be executed k times, and therefore the theoretical complexity of this algorithm is O(kn).

- **Experimental Time:**

N	Execution time	Theoretical time
1000	0	0,000530333
4000	0	0,002121333
6000	0	0,003182
8000	0	0,004242667
9000	0	0,004773
80000	0,031	0,042426667
500000	0,249	0,265166667
1000000	0,515	0,530333333
2000000	1,045	1,060666667
3000000	1,591	1,591
	dt	1,65729E-08

- **Représentation graphique :**



- **<u>conclusion</u>** :

  After studying the algorithm corresponding to the distribution sort method, we conclude that this method has linear complexity in the worst and best cases. Therefore, it is a fast method and the fastest and most efficient method among the three methods implemented so far.

4. QuickSort

QuickSort or Pivot Sort) is a sorting algorithm invented by C.A.R. Hoare in 1961 and is based on the divide-and-conquer design principle. It is typically used for arrays but can also be adapted for lists. In the case of arrays, it is an in-place sorting algorithm but not stable. The choice of the pivot element can vary from one algorithm to another; in our program, the pivot is set to the first element.

- **Algorithme en C:**

```c
void swap(int *a, int *b)
{
  int t = *a;
  *a = *b;
  *b = t;
}
```

```c
int partition(int arr[], int low, int high)
{
  int pivot = arr[high]; // pivot
  int i = (low - 1);     // Index of smaller element

  for (int j = low; j <= high - 1; j++)
  {
    // If current element is smaller than the pivot
    if (arr[j] < pivot)
    {
      i++; // increment index of smaller element
      swap(&arr[i], &arr[j]);
    }
  }
  swap(&arr[i + 1], &arr[high]);
  return (i + 1);
}
```

```
80  void QuickSort(int arr[], int low, int high)
81  {
82    if (low < high)
83    {
84      /* pi is partitioning index, arr[p] is now
85      at right place */
86      int pi = partition(arr, low, high);
87
88      // Separately sort elements before
89      // partition and after partition
90      QuickSort(arr, low, pi - 1);
91      QuickSort(arr, pi + 1, high);
92    }
93  }
```

- **Analysis and Complexity :**

The complexity of the Partition function is of the order Θ(n). The recurrence equation for the program is:

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

- Worst case : $q = 1$ ou $q = n - 1$

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

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

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

...

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

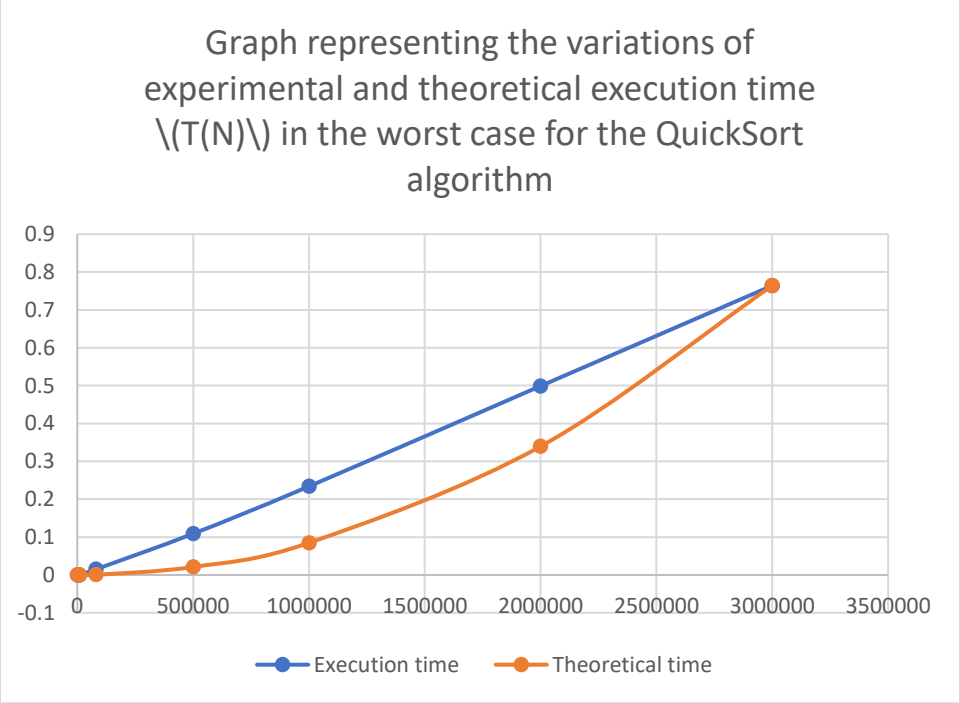
$$T(n) = \mathrm{c}\,.\left(\frac{\mathrm{n(n+1)}}{2}\right)$$

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

- **Experimental Times:**

N	Execution time	Theoretical time
1000	0	8,49737E-08
4000	0	1,35856E-06
6000	0	3,05651E-06
8000	0	5,43357E-06
9000	0	6,87676E-06
80000	0,015	0,000543295
500000	0,109	0,021222258
1000000	0,234	0,084888945
2000000	0,499	0,339555612
3000000	0,764	0,764
	dt	1,69778E-13

- **Représentation graphique :**



- **<u>conclusion</u>** :

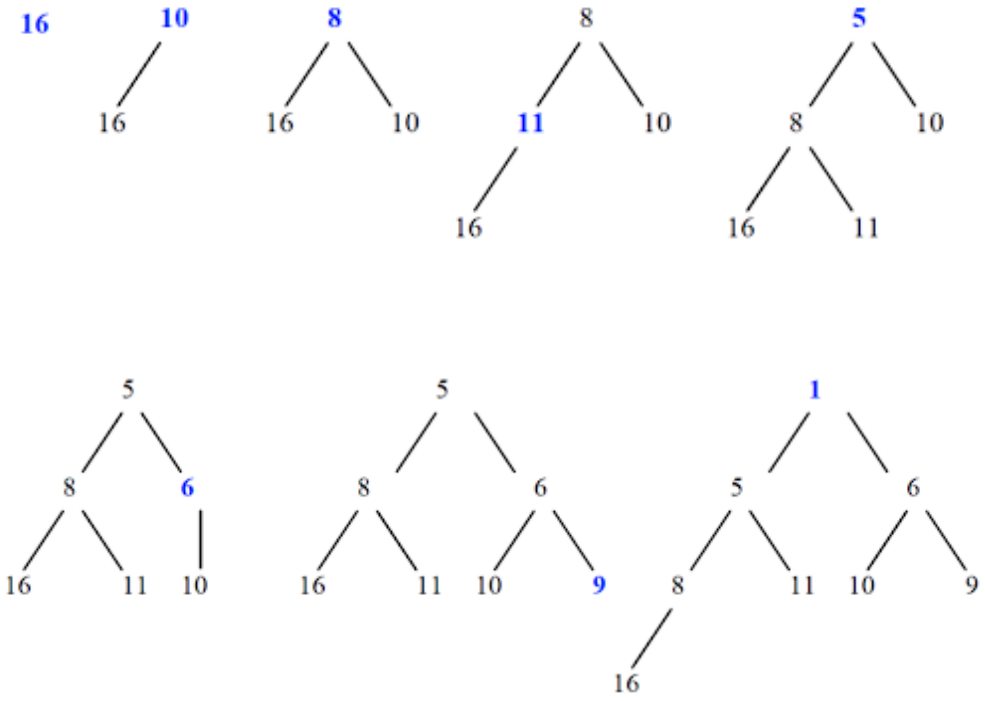
After studying the algorithm corresponding to the QuickSort method, we conclude that this method has a quadratic complexity in the worst case and sub-quadratic complexity in the best case. Therefore, it is a fast method but remains less efficient than the distribution sort method.

5. Heapsort

Heap Sort is a comparison-based sorting algorithm. This algorithm is asymptotically optimal, meaning it is demonstrated that no comparison-based sorting algorithm can achieve better asymptotic complexity. However, Heap Sort is not stable. Its major drawback is its slower performance compared to QuickSort.

5.1. Construction of the binary tree :

Heap sort relies on a specific data structure: the heap. It is a representation of a binary tree in the form of an array. In our case, we are interested in the Min-Heap, where the ordering relation is 'less than or equal to,' and the root of the tree is the smallest element in the array.



- **Algorithme en C :**

```c
void supprimer_min(int T[],int a)
{
    int x;
    x = T[a];
    T[a] = T[0];
    T[0] = x;
}
```

```c
void creer_tas(int T[], int n, int i)
{
    int p = i;
    int gauche = 2 * i + 1; // fils gauche
    int droite = 2 * i + 2; // fils droit
    int x;

    if (gauche < n && T[gauche] < T[p])
    {
        p = gauche;
    }

    if (droite < n && T[droite] < T[p])
    {
        p = droite;
    }

    if (p != i) {
        x = T[i];
        T[i] = T[p];
        T[p] = x;

        creer_tas(T, n, p);
    }
}
```

```
void tri_tas(int T[], int n)
{
    int x;
    int i;
    i = (n/2) - 1;
    while(i>=0)
    {
        creer_tas(T, n, i);
        i--;
    }
    i = n-1;
    while(i>=0)
    {
        supprimer_min(T,i);
        creer_tas(T, i, 0);
        i--;
    }


    //inverser les élements du tableau
    for(i=0;i<n/2;i++)
    {
        x = T[i];
        T[i] = T[n-i-1];
        T[n-i-1] = x;
    }
}
```

Complexity :

The recursive relation for the given program can be expressed as follows:

$$T(n)=2T(2n)+O(n)$$

where \(n \) is the number of elements in the array. This recurrence relation describes the behavior of the `tri_tas` function during both the heap construction phase and the sorting phase.

Explanation:

The heap construction phase, performed by the `creer_tas` function, recursively calls itself for each node in the heap. Each recursive call divides the problem into two subproblems of size n/2, leading to the first part of the recurrence relation: T(n) = 2T(n/2).

Overall, the recurrence relation captures the time complexity of the heap sort algorithm implemented in the provided program, and it is O(n log n) .

- **Total complexity of the heap sort algorithm:**

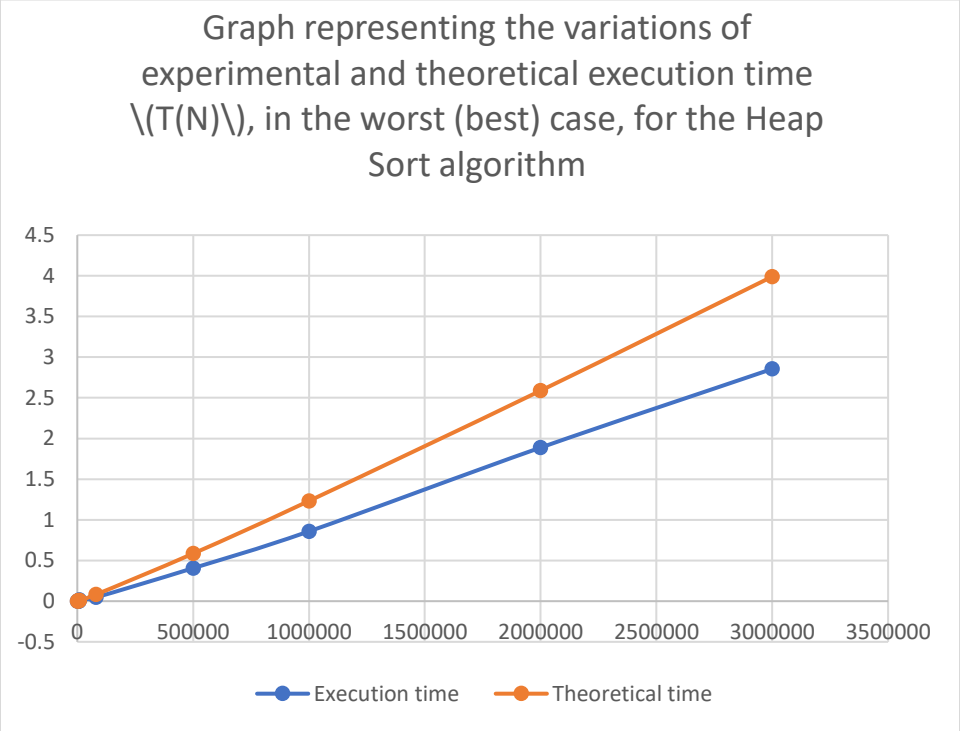
$$T(n) = O(n.\log n) + O(n.\log n) + O(n)$$

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

- **Temps d'exécution :**

N	Execution time	Theoretical time
1000	0	0,000616047
4000	0	0,002958719
6000	0	0,004655039
8000	0	0,006411967
9000	0,015	0,007308
80000	0,047	0,080547595
500000	0,406	0,585139079
1000000	0,858	1,232094387
2000000	1,888	2,58782123
3000000	2,855	3,99021237
	dt	2,05349E-07

- **Représentation graphique :**



- **conclusion** :

After studying the algorithm corresponding to the Heap Sort method, we conclude that this method has sub-quadratic complexity in both the worst and best cases. Therefore, it is a very fast method and is the fastest and least costly among all the methods studied. Thus, it is the most efficient method.

❖ Conclusion

In terms of execution time, the most efficient algorithm is QuickSort, followed by Heap Sort and Distribution Sort. These three algorithms have a theoretical complexity of O(n logn). Bubble Sort is the slowest algorithm, followed by Gnome Sort; both have a complexity of