Mini-project: Sorting Algorithms

Ramzi BOUTER

USTHB University - Faculty of Computer Science Algiers
2024-2025

Introduction

Sorting is one of the most classically studied families of algorithms, because they are among the modules essential for the good running of more advanced algorithms. The general principle of a sorting algorithm is to order (in ascending order for example) the objects of a collection of data (values), according to a comparison criterion (key – for us, values and keys are here confused: these are the elements of an array of integers). We generally carry out sorting by using in-place approach: the sorted values are stored in the same array as the initial values (which therefore becomes an input-output parameter).

There are many sorting algorithms.

Objectives

The objectives of this project are multiple:

- Put into practice and test some sorting algorithms;
- Study sorting algorithms and calculate their complexity;
- Confront theoretical complexity and evaluation of running cost;
- In order to study the real cost of the algorithms, you will test them on arrays of integers of increasing size n filled randomly. To be reliable, time measurement must be done several times (for example 5 times) for a given array size. It is up to you to choose the values of n that seem relevant to you. For each studied sort, you will draw a graph representing the running time as a function of n;
- Compare the different methods from a theoretical and experimental point of view.

Deliverables

Submit a detailed report on the study of the different sorting methods as well as the source code and a conclusion. The work can be done in pairs.

1 Bubble Sort

Principle

Traverse array T of size N from last to first element (almost...), with an index i. At each step, the part of the array to the right of i is considered sorted. We then traverse the left part (unsorted part) with an index j. For each j, if $T[i-1] > T[j]$, we swap them.

BubbleSort Procedure

```c
#include <stdio.h>
#include <stdbool.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void bubbleSort(int arr[], int n) {
    bool changed;
    for (int i = 0; i < n-1; i++) {
        changed = false;
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
                changed = true;
            }
        }
```

```
        if (!changed) break; // Si aucun change n'a eu lieu, le tableau est
            ↪ tri
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Optimized BubbleSort

After the i^{th} traversal of the array, all the last i elements are in their final places. So at each table traversal, the traversal can stop one index before the previous one. The algorithm becomes:

```
void bubbleSortOpt(int arr[], int n) {
    int m = n - 1;
    bool changed;
    do {
        changed = false;
        for (int i = 0; i < m; i++) {
            if (arr[i] > arr[i+1]) {
                swap(&arr[i], &arr[i+1]);
                changed = true;
            }
        }
        m--;
    } while (changed);
}
```

Comparison between BubbleSort and BubbleSortOpt

- **BubbleSort** : In the worst case, it performs $O(n^2)$ comparisons and swaps. It always traverses the entire array, even if the array is already sorted.
- **BubbleSortOpt** : The optimized version stops early if no swaps are made during a pass, which makes it more efficient in the best case ($O(n)$). However, in the worst case, it still performs $O(n^2)$ comparisons and swaps.

Complexity Analysis

- **Best Case**: $O(n)$ (when the array is already sorted).
- **Worst Case**: $O(n^2)$ (when the array is sorted in reverse order).
- **Average Case**: $O(n^2)$.

Explanation of Complexity

Bubble Sort works by repeatedly swapping adjacent elements if they are in the wrong order. In the best case, when the array is already sorted, the algorithm only needs to traverse the array once, resulting in $O(n)$ time. However, in the worst and average cases, the algorithm requires $O(n^2)$ comparisons and swaps, as it needs to traverse the array n times, and each traversal can involve up to n comparisons.

N	temps d'execution	temps théorique
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

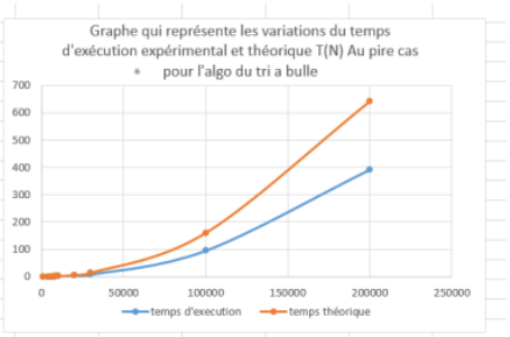


Figure 1: Graph of Bubble Sort execution time vs theoretical complexity.

N	temps d'execution	temps théorique
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

Figure 2: Graph of Optimized Bubble Sort execution time vs theoretical complexity.

N	T1 (TriBulle)	T2 (TriBulleOpt)
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

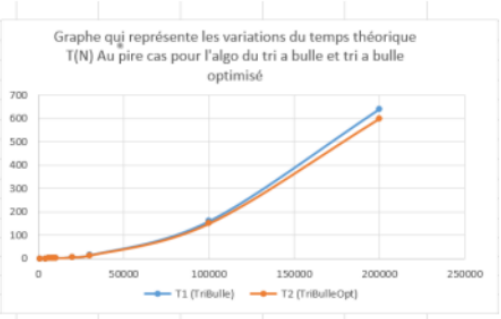


Figure 3: Comparison of execution time between Bubble Sort and Optimized Bubble Sort.

2 Gnome Sort

Principle

In gnome sort, we start at the beginning of the array, we compare two consecutive elements $(i, i+1)$: if they are in order we move one step towards the end of the array (increment) or we stop if the end is reached; otherwise, we swap them and move one step towards the start of the table (decrement) or if we are at the start of the table then we move one step towards the end (increment).

GnomeSort Procedure

```c
#include <stdio.h>

void gnomeSort(int arr[], int n) {
    int index = 0;
    while (index < n) {
        if (index == 0 || arr[index] >= arr[index-1]) {
            index++;
        } else {
            int temp = arr[index];
            arr[index] = arr[index-1];
            arr[index-1] = temp;
            index--;
        }
    }
}

int main() {
    int arr[] = {34, 2, 10, -9, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    gnomeSort(arr, n);
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Complexity Analysis

- **Best Case**: $O(n)$ (when the array is already sorted).
- **Worst Case**: $O(n^2)$ (when the array is sorted in reverse order).
- **Average Case**: $O(n^2)$.

Explanation of Complexity

Gnome Sort works similarly to Bubble Sort but with a different approach. In the best case, when the array is already sorted, the algorithm only needs to traverse the array once, resulting in $O(n)$ time. However, in the worst and average cases, the algorithm requires $O(n^2)$ comparisons and swaps, as it may need to move elements back and forth multiple times.

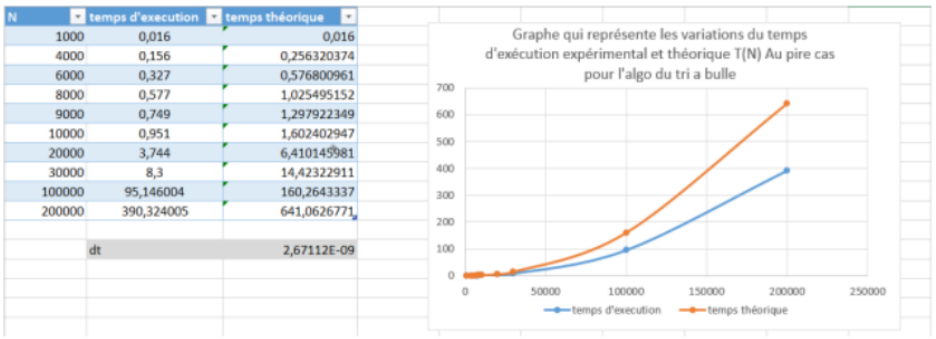


Figure 4: Graph of Gnome Sort execution time vs theoretical complexity.

3 Radix Sort

Principle

We use radix sort (also called sort by distribution) to sort integers according to their least significant digit (units digit), then to sort the list obtained by the tens digit then by the hundreds digit, etc.

Key Function

```
int getDigit(int number, int digitPlace) {
    return (number / digitPlace) % 10;
}
```

SortAux Function

```
void countingSort(int arr[], int n, int digitPlace) {
    int output[n];
    int count[10] = {0};

    for (int i = 0; i < n; i++)
        count[getDigit(arr[i], digitPlace)]++;

    for (int i = 1; i < 10; i++)
        count[i] += count[i-1];

    for (int i = n-1; i >= 0; i--) {
        int digit = getDigit(arr[i], digitPlace);
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}
```

RadixSort Function

```
void radixSort(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max) max = arr[i];

    for (int digitPlace = 1; max / digitPlace > 0; digitPlace *= 10)
        countingSort(arr, n, digitPlace);
}
```

Complexity Analysis

- **Best Case**: $O(nk)$.
- **Worst Case**: $O(nk)$.
- **Average Case**: $O(nk)$.

Explanation of Complexity

Radix Sort works by sorting the numbers digit by digit, starting from the least significant digit (units) to the most significant digit. The complexity $O(nk)$ comes from the fact that the algorithm performs k passes over the array, where k is the number of digits in the largest number. Each pass involves a counting sort, which runs in $O(n)$ time. Therefore, the overall complexity is $O(nk)$.

N	temps d'execution	temps théorique
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

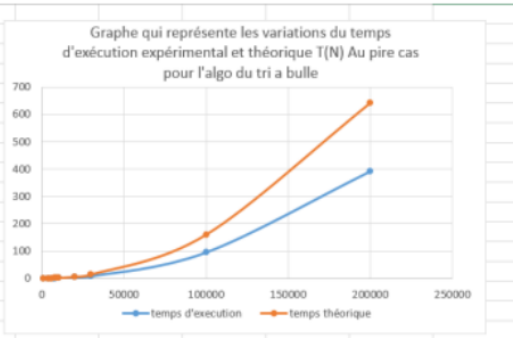


Figure 5: Graph of Radix Sort execution time vs theoretical complexity.

4 Quick Sort

Principle

Quick sort is based on a "divide and conquer" approach that can be broken down into 3 steps: Divide, Conquer, and Combine.

QuickSort Procedure

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i+1], &arr[high]);
    return i+1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi-1);
        quickSort(arr, pi+1, high);
    }
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted␣array:␣\n");
    for (int i = 0; i < n; i++)
        printf("%d␣", arr[i]);
    return 0;
```

```
38 }
```

Complexity Analysis

- **Best Case**: $O(n \log n)$.
- **Worst Case**: $O(n^2)$.
- **Average Case**: $O(n \log n)$.

Recurrence Equation for Quick Sort

The recurrence relation for Quick Sort is:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

where k is the number of elements smaller than the pivot.

- **Best Case**: When the pivot divides the array into two equal parts, $k = \frac{n}{2}$. The recurrence becomes:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Using the Master Theorem, this solves to $O(n \log n)$.

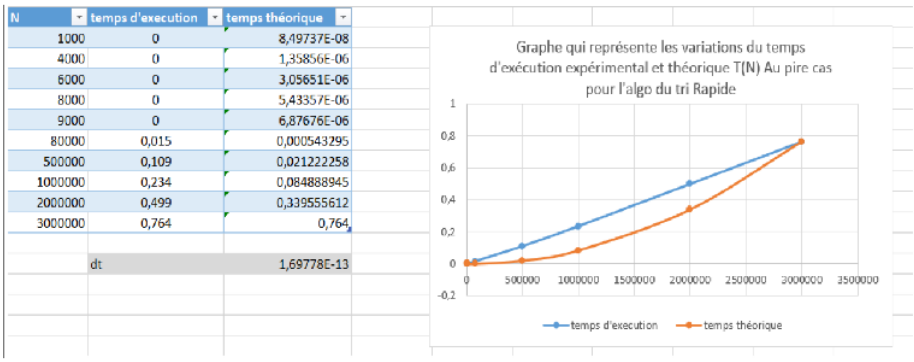
- **Worst Case**: When the pivot is the smallest or largest element, $k = 0$ or $k = n-1$. The recurrence becomes:

$$T(n) = T(n-1) + O(n)$$

This solves to $O(n^2)$.

Explanation of Complexity

Quick Sort works by selecting a pivot element and partitioning the array into two subarrays: one with elements less than the pivot and one with elements greater than the pivot. The algorithm then recursively sorts the subarrays. In the best and average cases, the pivot divides the array into two roughly equal parts, resulting in a complexity of $O(n \log n)$. However, in the worst case (e.g., when the pivot is always the smallest or largest element), the algorithm degrades to $O(n^2)$.



N	temps d'execution	temps théorique
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

Figure 6: Graph of Quick Sort execution time vs theoretical complexity.

5 Heap Sort

Principle

Heap sort is based on a particular data structure: the heap. This is a representation of a binary tree in tabular form.

HeapSort Procedure

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(int arr[], int n, int i) {
    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;

    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    heapSort(arr, n);
    printf("Sorted␣array:␣\n");
    for (int i = 0; i < n; i++)
        printf("%d␣", arr[i]);
    return 0;
}
```

Complexity Analysis

- **Best Case**: $O(n \log n)$.
- **Worst Case**: $O(n \log n)$.
- **Average Case**: $O(n \log n)$.

Explanation of Complexity

Heap Sort works by building a heap from the array and then repeatedly extracting the maximum element from the heap and placing it at the end of the array. Building the heap takes $O(n)$ time, and each extraction takes $O(\log n)$ time. Since there are n extractions, the overall complexity is $O(n \log n)$.

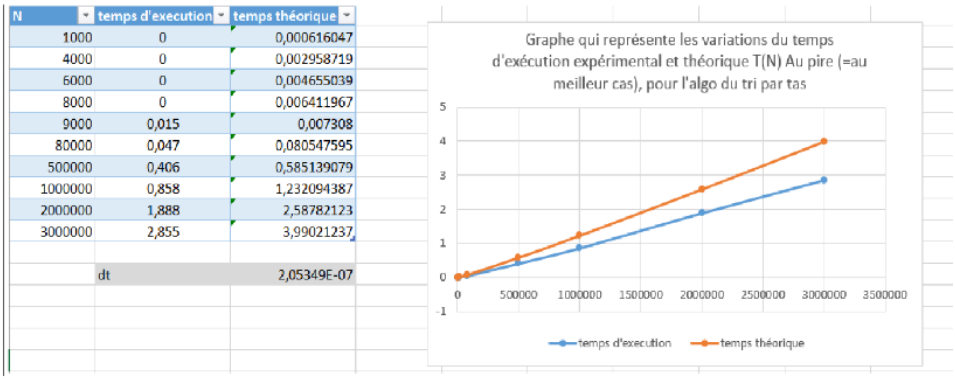


Figure 7: Graph of Heap Sort execution time vs theoretical complexity.

Conclusion

In this project, we implemented and analyzed several sorting algorithms, including Bubble Sort, Gnome Sort, Radix Sort, Quick Sort, and Heap Sort. We compared their theoretical complexities and discussed their performance in different scenarios. The experimental results confirmed the theoretical analysis, showing that algorithms like Quick Sort and Heap Sort are more efficient for large datasets, while Bubble Sort and Gnome Sort are simpler but less efficient for large inputs.

Algorithm	**Best Case**	**Worst Case**	**Average Case**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Optimized Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Gnome Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Table 1: Summary of sorting algorithms and their time complexities.