Mini-Project: Sorting Algorithms

HADJ ARAB Adel BECHAR Walid

January 12, 2025

1 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 an "in-place" approach: the sorted values are stored in the same array as the initial values (which therefore becomes an input-output parameter).

2 Sorting Algorithms

2.1 Bubble Sort

2.1.1 Algorithm

2.1.2 Complexity

Bubble Sort has a worst-case and average-case time complexity of $O(n^2)$, where n is the number of elements to be sorted. The best-case time complexity is $O(n)$ when the array is already sorted. The space complexity is $O(1)$ as it is an in-place sorting algorithm.

2.1.3 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.1.4 Results

2.2 Gnome Sort

2.2.1 Algorithm

2.2.2 Complexity

Gnome Sort has a worst-case and average-case time complexity of $O(n^2)$, where n is the number of elements to be sorted. The best-case time complexity is $O(n)$ when the array is already sorted. The space complexity is $O(1)$ as it is an in-place sorting algorithm.

2.2.3 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.2.4 Results

2.3 Radix Sort

2.3.1 Algorithm

2.3.2 Complexity

Radix Sort has a time complexity of $O(d \cdot (n + k))$, where d is the number of digits in the largest number, n is the number of elements, and k is the range of the digit values. The space complexity is $O(n + k)$.

2.3.3 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.3.4 Results

2.4 Quick Sort

2.4.1 Algorithm

2.4.2 Complexity

Quick Sort has a worst-case time complexity of $O(n^2)$, which occurs when the pivot selection is poor. However, the average-case and best-case time complexity is $O(n \log n)$. The space complexity is $O(\log n)$ due to the recursive stack space.

2.4.3 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.4.4 Results

2.5 Heap Sort

2.5.1 Algorithm

2.5.2 Complexity

Heap Sort has a time complexity of $O(n \log n)$ for all cases (worst, average, and best). The space complexity is $O(1)$ as it is an in-place sorting algorithm.

2.5.3 Experimental Study

To study the real cost of the algorithms, we tested them on arrays of integers of increasing size n filled randomly. Time measurement was done several times (5 times) for a given array size. The values of n chosen were 100, 1000, 10000, and 100000.

2.5.4 Results

3 Conclusion

In this project, we put into practice and tested some sorting algorithms, studied their complexity, and compared theoretical complexity with the evaluation of running cost. The experimental results matched the theoretical expectations, confirming the efficiency of the studied sorting algorithms.