

TP 01 – Comparison sorting algorithms

Programming language of your choice (to be validated with your TP instructor).

In this TP exercise, we will explore comparison sort algorithms and their complexity. The problem is to sort in ascending order an array T of n integers, whose values are bounded (for example, between 0 and 999). Therefore, the size of the input will be n .

For the experiments, we will plot the computation time as a function of the input size. For each size, we will perform 12 executions and produce three curves :

- a curve of minimum times,
- a curve of maximum times,
- a curve of average time excluding minimum and maximum times.

The theoretical analysis curve will also be included. All graphs must be commented.

Exercise 1.

Lower bound for comparison sort

In this exercise, we will determine an $n \log n$ worst-case lower bound on the number of comparisons of any sorting algorithm based on comparisons between the elements of the array to be sorted. A sorting algorithm takes an array as input and swaps its elements so that the array is sorted in ascending order. A comparison-based sorting algorithm bases its decisions on element comparisons. For each comparison performed between two values in the array, the algorithm can have two different behaviours, depending on the result of the comparison.

For a given size n , a comparison-sorting algorithm can be viewed as a binary tree whose root is the starting point, and whose internal nodes represent the comparisons performed and their two different possible outcomes. Each path from the root to a leaf represents a possible execution of the algorithm.

1. For an array of size n , how many different permutations must the algorithm potentially be able to perform?
2. The number of which elements of the tree is bounded below by this value?
3. Which element of the complexity of the algorithm corresponds to the height of its tree?
4. What is the relationship between the tree's height and its number of leaves?
5. From the previous point, deduce an inequality relating the size n and the worst-case complexity of any correct comparison-sorting algorithm.
6. Finally, bound $n!$ below by the product of its first $\frac{n}{2}$ terms which are all at least $\frac{n}{2}$, or use Stirling's formula approximating $n!$ by $\sqrt{2\pi n}(\frac{n}{e})^n$.

Exercise 2.

Two-thirds sort

The *two-thirds sort* is a recursive sorting algorithm that works in three steps :

- (a) sort the first $2/3$ of the array,
- (b) sort the last $2/3$ of the array,
- (c) sort the first $2/3$ of the array.

If there are at most two elements in the array, sort them directly.

1. Implement the two-thirds sort algorithm.

2. Explain why this algorithm correctly sorts any array.
3. Analyse its worst-case complexity using the Master theorem.
4. Test your algorithm on randomly filled arrays and produce a graph.

Exercise 3.

Merge sort

Merge sort is a recursive algorithm that works in three steps :

- (a) sort the first half of the array using a recursive call,
- (b) sort the second half of the array using a recursive call,
- (c) merge the two sorted parts.

If there are at most two elements in the array, sort them directly.

1. Implement the merge sort algorithm.
2. Analyse its worst-case complexity using the Master theorem.
3. Test your algorithm on randomly filled arrays and produce a graph.

Exercise 4.

Quicksort

This algorithm is the most efficient in practice, and the one that is usually implemented by `sort()` methods. It works by selecting a value of the array, called *pivot*, and then arranging :

- at the beginning of the array all the elements less than the pivot, and
- at the end of the array all the elements greater than the pivot, and
- the pivot between these two parts.

The pivot is chosen to be the first value in the array. To arrange all the elements around the pivot, we start visiting the array from the last element : as long as it is greater than the pivot, we move towards the pivot; then, when a value smaller than the pivot is encountered, we exchange it with the pivot, and continue the visit in the other direction (therefore always towards the pivot). When we meet the pivot, the pivot has found its position.

The algorithm continues by recursively applying the same procedure to both parts.

1. Implement the quicksort algorithm.
2. Analyse its worst-case complexity, which is (surprisingly) the case where the array is already sorted. Test your algorithm to verify this bound. Produce a graph.
3. The average complexity of quicksort is $O(n \log n)$. Test your algorithm on randomly filled arrays to verify this bound and produce a graph.
4. What is the worst-case complexity of quicksort when the pivot at each iteration is the median value of the considered part?
5. The median value of an array can be found in linear time. Implement this pivot method choice, and compare its efficiency on randomly filled arrays.