Raport

Benchmarking Sorting Algorithms

Computational Thinking with Algorithms

G00376368 – Karolina Szafran-Belzowska

Table of contents

[Introduction 3](#_Toc39860663)

[The concept of sorting 3](#_Toc39860664)

[Time and Space Complexity 4](#_Toc39860665)

[Bubble Time complexity 4](#_Toc39860666)

[Project Specification 6](#_Toc39860667)

[References 6](#_Toc39860668)

# 

# Introduction

## The concept of sorting

A sorting algorithm is a methods for reorganizing a large number of items into a specific order, such as alphabetical, highest-to-lowest or shortest-to-longest distance. Sorting algorithms take a list of items as input data, perform specific operations on those lists and deliver ordered arrays as output.[1]

A sorting algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.[2][3]

Choosing the best sorting algorithm is as about knowing what you are sorting as it is about the relative performance of the algorithms.

The output of any sorting algorithm must satisfy two conditions:

1. The output is in non-decreasing order (each element is no smaller than the previous element),
2. And the output is a [permutation](https://en.wikipedia.org/wiki/Permutation) (a reordering of the original elements) of the input.

The input data is often stored in an [array](https://en.wikipedia.org/wiki/Array_data_type), which allows [random access](https://en.wikipedia.org/wiki/Random_access), rather than a list, which only allows [sequential access](https://en.wikipedia.org/wiki/Sequential_access), though many algorithms can be applied to either type of data after suitable modification.

Sorting algorithms are often classified by:

* [**Computational complexity**](https://en.wikipedia.org/wiki/Computational_complexity_theory) ([worst, average and best](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) behavior) in terms of the size of the list (*n*). For typical serial sorting algorithms good behavior is O(*n* log *n*), with parallel sort in O(log2 *n*), and bad behavior is O(*n*2). Ideal behavior for a serial sort is O(*n*), but this is not possible in the average case. Optimal parallel sorting is O(log *n*). [Comparison-based sorting algorithms](https://en.wikipedia.org/wiki/Comparison_sort) need at least Ω(*n* log *n*) comparisons for most inputs.
* [**Computational complexity**](https://en.wikipedia.org/wiki/Computational_complexity_theory)**of swaps** (for "in-place" algorithms).
* [**Memory**](https://en.wikipedia.org/wiki/Memory_(computing))**usage** (and use of other computer resources). Some sorting algorithms are "[in-place](https://en.wikipedia.org/wiki/In-place_algorithm)". Strictly, an in-place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(*n*)) additional memory is considered "in-place".
* **Recursion.** Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
* **Stability:** [stable sorting algorithms](https://en.wikipedia.org/wiki/Sorting_algorithm#Stability) maintain the relative order of records with equal keys (i.e., values).
* Whether or not they are **a**[**comparison sort**](https://en.wikipedia.org/wiki/Comparison_sort)**.** A comparison sort examines the data only by comparing two elements with a comparison operator.
* **General method:** insertion, exchange, selection, merging, *etc.* Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.
* Whether the algorithm is serial or parallel. The remainder of this discussion almost exclusively concentrates upon serial algorithms and assumes serial operation.
* **Adaptability:** Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be [adaptive](https://en.wikipedia.org/wiki/Adaptive_sort).

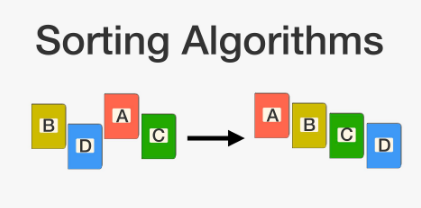


Image 1: Sorting algorithms: <https://brilliant.org/wiki/sorting-algorithms/> on 08/05/2020

## Time and Space Complexity

Best, worst, and average cases of a given [algorithm](https://en.wikipedia.org/wiki/Algorithm) express what the [resource](https://en.wikipedia.org/wiki/Resource_(computer_science)) usage is *at least*, *at most* and *on average*. Usually the resource being considered is running time, for example [time complexity](https://en.wikipedia.org/wiki/Time_complexity), but could also be memory or other resource. **The** **Best case** is the function which performs the minimum number of steps on input data of n elements. **Worst case** is the function which performs the maximum number of steps on input data of size n and **average case** is the function which performs an average number of steps on input data of n elements. Average performance and worst-case performance are the most used in algorithm analysis.

Many algorithms with bad worst-case performance have good average-case performance. For problems we want to solve, this is a good thing. For [cryptography](https://en.wikipedia.org/wiki/Cryptography), this is very bad, we want typical instances of a cryptographic problem to be hard.[4]

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input. To compare standard sorting algorithms on the basis of space, the Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort and Heap Sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.[5]

### Bubble Time complexity

**Worst and Average Case Time Complexity:**O(n\*n). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**Auxiliary Space:** O(1)

Bubble sort takes minimum time (Order of n) when elements are already sorted.

### Selection Sort Complexity

**Time Complexity:** O(n2) as there are two nested loops.

**Auxiliary Space:** O(1)

Selection sort never makes more than O(n) swaps and can be useful when memory write is a costly operation.

### Insertion Sort Complexity

**Time Complexity:** O(n\*2)

**Auxiliary Space:**O(1)

Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

### Merge Sort Complexity

**Time Complexity:**  Sorting arrays on different machines.

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.   
T(n) = 2T(n/2) + 

Time complexity of Merge Sort is  in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

**Auxiliary Space:** O(n)

### Counting Sort Complexity

**Time Complexity:** O(n+k) where n is the number of elements in input array and k is the range of input.  
**Auxiliary Space:** O(n+k)

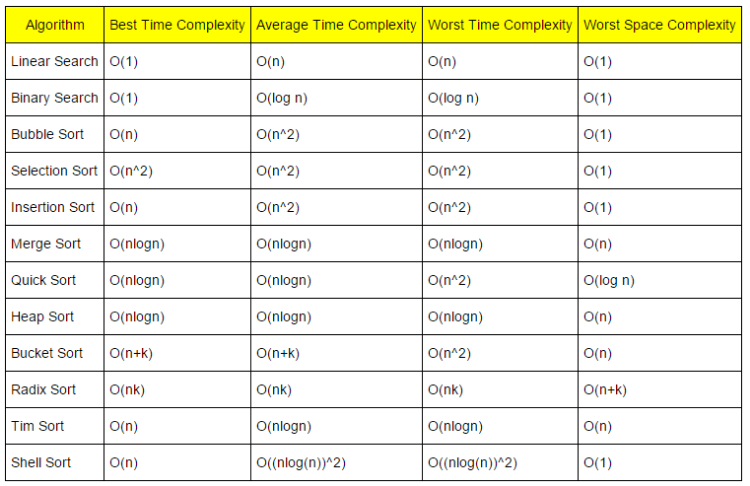


Image 2: Time complexity. Taken from: <https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/> on 08/05/2020

### Big O Complexity

Big O notation is a mathematical notation that describes the [limiting behavior](https://en.wikipedia.org/wiki/Asymptotic_analysis) of a [function](https://en.wikipedia.org/wiki/Function_(mathematics)) when the [argument](https://en.wikipedia.org/wiki/Argument_of_a_function) tends towards a particular value or infinity.[6] Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used by an algorithm.

Below are examples of Big O notation[7][8]:

1. O(1)- describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.



1. O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.



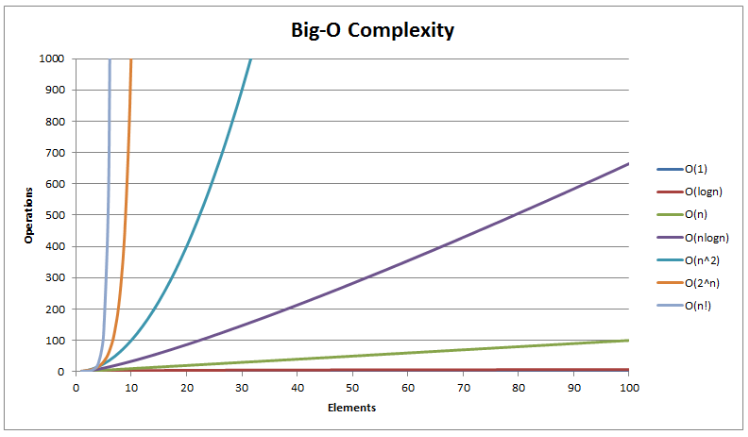
1. O(N2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N3), O(N4) etc.

If the list has two letters in it the program will take four operations to run, but if the list has four trillion letters, it may never finish running.



1. O(2N) denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an O(2N) function is exponential - starting off very shallow, then rising meteorically. An example of an O(2N) function is the recursive calculation of Fibonacci numbers.
2. Logarithms - [**Binary search**](https://en.wikipedia.org/wiki/Binary_search) is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it.  It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set. This type of algorithm is described as **O(log N)**. The bigger the input, the smaller proportion of the actual input your program has to go through.

**Orders of growth**

****

**Image 3: Big O Complexity. Taken from:** <https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/> on 08/05/2020

# In-place algorithm

**In-place algorithm** is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) which transforms input using no auxiliary [data structure](https://en.wikipedia.org/wiki/Data_structure). However a small amount of extra storage space is allowed for auxiliary variables. The input is usually overwritten by the output as the algorithm executes. In-place algorithm updates input sequence only through replacement or swapping of elements. An algorithm which is not in-place is sometimes called **not-in-place** or **out-of-place**.

An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input ‘in-place’. However, a small constant extra space used for variables is allowed.

An example of in-place algorithm.

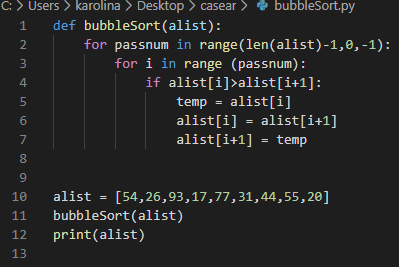
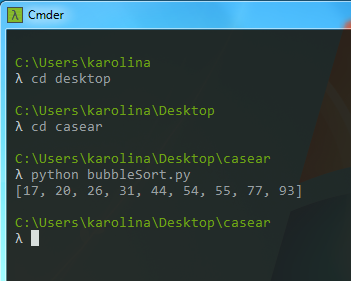


Image 4: Taken from the module Computational Thinking with Algorithms (week 10)

And the result of the code above is:



An example of in-place algorithms: [Bubble sort](http://www.geeksforgeeks.org/bubble-sort/), [Selection Sort](http://www.geeksforgeeks.org/selection-sort/), [Insertion Sort](http://www.geeksforgeeks.org/insertion-sort/), [Heapsort](https://www.geeksforgeeks.org/heap-sort/), QuickSort.

Not In-Place : [Merge Sort](http://www.geeksforgeeks.org/merge-sort/).

# Project Specification

This project contains a Python application which will be used to benchmark five different sorting algorithms. I have implemented and benchmarked five sorting algorithms with random array. I have chosen the fallowing algorithms:

1. A simple comparison-based sort:

* Bubble Sort
* Selection Sort (my choice)
* Insertion Sort(my choice)

1. An efficient comparison-based sort:

* Merge Sort

1. A non-comparison sort:

* Counting Sort

## References

[1] <https://whatis.techtarget.com/definition/sorting-algorithm>

[2] <https://www.studytonight.com/data-structures/introduction-to-sorting>

[3] <https://en.wikipedia.org/wiki/Sorting_algorithm>

[4] <https://en.wikipedia.org/wiki/Best,_worst_and_average_case>

[5] <https://www.geeksforgeeks.org/g-fact-86/>

[6] <https://en.wikipedia.org/wiki/Big_O_notation>

[7] <https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>

[8] <https://medium.com/better-programming/a-gentle-explanation-of-logarithmic-time-complexity-79842728a702>