Sorting algorithms analysis

Krisztian Szabo

October 28, 2019

Contents

1 Introduction

Sorting algorithms are important for the field of computer science, as they play a major role in data processing within most, if not all, domains. Therefore, efficient sorting algorithms have always been sought after by researchers and software engineers.

Sorting is the process of arranging a number of items systematically. The result of any sorting process must satisfy two criteria: the output is in increasing order, and the output is a permutation of the input.

In this article we will consider these algorithms: *Merge sort*, *Quicksort*, *Insertion sort*, *Bubble sort*, and *Tree sort*. We will analyze them based on the following criteria:

- Computational complexity in the best, average, and worst cases
- Characteristic operation
- Memory usage
- Recursivity
- Divide and conquer
- Stability: An algorithm is considered stable if the equal elements are left in place. This has implications for sorting several times by different criteria.
- Adaptability: An algorithm is considered adaptable if it takes into consideration the presortedness of the initial list, thus increasing efficiency.

2 Analysis

2.1 Merge sort

Merge sort is an efficient, general-purpose, comparison-based sorting algorithm. It was invented by John von Neumann in 1945. It works in two phases: first it recursively divides the unsorted list into n sub-lists (i.e. each final sub-list is atomic), and secondly it merges these sub-lists two by two, again recursively, producing a sorted sub-list at each step, until finally a single sorted list remains, which is the output.

Characteristics:

- Computational complexity:
  - Best case: $\mathcal{O}(n \log_2 n)$
  - Average case: $\mathcal{O}(n \log_2 n)$
  - Worst case: $\mathcal{O}(n \log_2 n)$
- Characteristic operation: Merge sort is a comparison-based sorting algorithm. As the name implies, this operation takes place on the merging of the sub-lists.
- Memory usage: this algorithm does not operate in-place, because of its recursive nature, and thus it requires $2n$ total memory; n for the initial array, $n/2$ for the second level, $n/4$ for the third, and so forth, essentially reaching $2n$ in total.
- Recursivity: as mentioned above, this algorithm is a recursive algorithm.
- Divide and conquer: like most recursive algorithms, Merge sort does indeed employ a divide and conquer strategy by splitting the list in two and calling itself on the two resulting sub-lists.
- Stability: Merge sort is a stable algorithm.
- Adaptability: Merge sort is not an adaptable algorithm (i.e. it will process the whole list indifferent of its initial state).

2.2 Quicksort

Quicksort is an efficient sorting algorithm originally developed by British computer scientist Tony Hoare in 1959 and published in 1961. It is still in common use to this day. If well implemented, this algorithm can run two or three times faster than its main competitors, Merge sort and Heapsort. It works by picking an element from the list, called the pivot, then moving all elements smaller than the pivot to the left of it, and all the larger ones to the right. It then recursively calls itself on the two sublists to the left and right of the pivot.

Characteristics:

- Computational complexity:
  - Best case: $\mathcal{O}(n \log_2 n)$
  - Average case: $\mathcal{O}(n \log_2 n)$
  - Worst case: $\mathcal{O}(n^2)$
- Characteristic operation: Quicksort is a comparison-based algorithm. This operation happens in the partitioning phase.
- Memory usage: $\log_2 n$ on average, but in certain worst cases it can be as high as n.
- Recursivity: Quicksort is a recursive algorithm.
- Divide and conquer: this algorithm uses a divide and conquer approach.
- Stability: it is not a stable algorithm in its typical implementation.
- Adaptability: Quicksort does not take into consideration the state of the initial list, and is thus not adaptable.

2.3 Insertion sort

Insertion sort is a simple, comparison-based sorting algorithm, that sorts the list by iterating over it, and inserting each element in its appropriate position. It is very inefficient on large data sets, but it does provide a few advantages.

Characteristics:

- Computational complexity:
  - Best case: $\mathcal{O}(n)$
  - Average case: $\mathcal{O}(n^2)$
  - Worst case: $\mathcal{O}(n^2)$
- Characteristic operation: Insertion sort is also a comparison-based algorithm.
- Memory usage: This algorithm works in-place, thus requiring very little additional memory (i.e. just one element's worth of extra memory, used for swapping the place of two elements).
- Recursivity: Insertion sort is an iterative algorithm, consuming the original list one item at a time.
- Divide and conquer: This algorithm does not employ the divide and conquer strategy.
- Stability: Insertion sort is a stable algorithm, leaving equal elements in the relative order they were found in, in the initial list.
- Adaptability: Time complexity decreases the closer each element is to its final sorted position. Thus, if the list is in large part already sorted, very few comparison operations will take place. This fact makes the algorithm adaptable.

2.4 Bubble sort

Bubble sort is a very simple, comparison-based sorting algorithm that works by repeatedly iterating over the input list and swapping the places of adjacent elements if they are out of order. Because it is a very inefficient algorithm, it is not used in practice, but due to its simplicity it is often talked about in educational settings as one of the most naïve approaches to sorting.

Characteristics:

- Computational complexity:
  - Best case: $\mathcal{O}(n)$
  - Average case: $\mathcal{O}(n^2)$
  - Worst case: $\mathcal{O}(n^2)$
- Characteristic operation: Bubble sort is a comparison-based algorithm.
- Memory usage: By working in-place, this algorithm uses very little additional memory, specifically one element's size of extra memory, needed to swap to elements in the list.
- Recursivity: This algorithm works by iterating (repeatedly) over the list, and is thereby not a recursive algorithm.
- Divide and conquer: Bubble sort does not use a divide and conquer strategy.
- Stability: This algorithm is considered stable because it does not alter the order of equal elements.
- Adaptability: Bubble sort is adaptive because it does not perform any alterations to adjacent elements that are in order. Thus, its computational complexity decreases the closer the initial list is to a sorted order.

2.5 Binary tree sort

Binary tree sort is a comparison-based sorting algorithm, that works by adding all the elements of the unsorted list into a balanced binary search tree, and then traversing it in order to obtain the sorted list.

Characteristics:

- Computational complexity:
  - Best case: $\mathcal{O}(n \log_2 n)$
  - Average case: $\mathcal{O}(n \log_2 n)$
  - Worst case: $\mathcal{O}(n \log_2 n)$
- Characteristic operation: This algorithm is comparison-based.
- Memory usage: Tree sort requires n extra memory to build the binary search tree, and thus is inefficient from this point of view.
- Recursivity: While insertion into the tree is done in a recursive way, obtaining the sorted list is a by-product, and thus this approach is not considered recursive in and of itself.
- Divide and conquer: Binary tree sort does not sort by a divide and conquer strategy.
- Stability: The algorithm is stable, leaving equal elements in their relative order.
- Adaptability: Because the algorithm has to insert all the elements into a binary search tree, the initial state of the input list does not matter. Therefore, tree sort is not adaptable.

3 Experimental data

By taking a Monte Carlo approach I ran these five algorithms over large, randomized arrays of integers. The first run consisted of 10,000 iterations over arrays of 10,000 elements. The second was of 20,000 iterations, and arrays of 20,000 elements. My hope was that these large values will pull the average number of operations towards what one would expect mathematically. The results are shown in the following tables:

Algorithm	Complexity	Expected	Actual
Merge sort	$\mathcal{O}(n \log_2 n)$	132,877	133,616
Quicksort	$\mathcal{O}(n \log_2 n)$	132,877	90,905
Insertion sort	$\mathcal{O}(n^2)$	100,000,000	24,998,650
Bubble sort	$\mathcal{O}(n^2)$	100,000,000	49,995,000
Tree sort	$\mathcal{O}(n \log_2 n)$	132,877	154,378

Table 1: The data from 10,000 iterations, over 10,000 elements

Algorithm	Complexity	Expected	Actual
Merge sort	$\mathcal{O}(n \log_2 n)$	285,754	287,232
Quicksort	$\mathcal{O}(n \log_2 n)$	285,754	195,831
Insertion sort	$\mathcal{O}(n^2)$	400,000,000	99,994,499
Bubble sort	$\mathcal{O}(n^2)$	400,000,000	199,990,000
Tree sort	$\mathcal{O}(n \log_2 n)$	285,754	339,232

Table 2: The data from 20,000 iterations, over 20,000 elements

As it can be seen, merge sort got closest to the expected value, then binary tree search, quicksort, bubble sort, and finally insertion sort.