A theoretical and experimental comparison of sorting algorithms

Bogdan Iulian Voicu
Faculty of Mathematics and Informatics,
West University Timisoara, Romania
Email: bogdan.voicu04@e-uvt.ro

May 2024

Abstract

This paper presents a comprehensive comparison of various sorting algorithms, encompassing both theoretical analysis and concrete experimentation. The paper examines a set of well-known algorithms. Evaluations encompass time complexity, space complexity, stability, and adaptability. Through analysis and experimentation, this paper aims to provide insights into the strengths and weaknesses of each algorithm, aiding in informed algorithm selection for diverse real-world scenarios.

Contents

1 Introduction

Sorting Algorithms are fundamental to computer science, from organizing databases to facilitating search operations. The problem is selecting the most appropriate sorting algorithm for each scenario: you must account for the size of your database, trends in your data and performance requirements. The code written and used for this paper can be found here. I have also included a copy of this Paper's LaTeX code, sorted alphabetically.

2 The Problem

Our solution involves comparing sorting algorithms systematically. We will:

1. Select a few sorting algorithms, including Bubble Sort, Merge Sort, and Quick Sort.
2. Analyze each algorithm's theoretical properties.
3. Conduct experiments to evaluate performance.
4. Analyze results to identify strengths and weaknesses.
5. Provide use cases in which the chosen algorithm is optimal.

3 Model and Implementation of Problem and Solution

3.1 Problem Model

- **Data:** The problem requires a dataset to be sorted. For the purposes of the experiment, the dataset generated will be of varying size, and in one of three possible orders: Ascending, Descending, or Random.
- **Sorting Algorithms:** A set of algorithms is required for comparison. The chosen Algorithsm are Quicksort, Mergesort, Bubblesort, Inserionsort, Selectionsort, Countingsort, Shakersort, Radixsort and Bogosort.
- **Evaluation Metrics:** Time complexity, Space complexity, and stability and adaptability will be used to evaluate the sorting algorithms.

3.2 Program Implementation

To better understand the sorting algorithms, I have written a program in which the user can input a size range and order for a database to be generated, and the program will return a graph which displays the time taken for the algorithm to finish sorting.

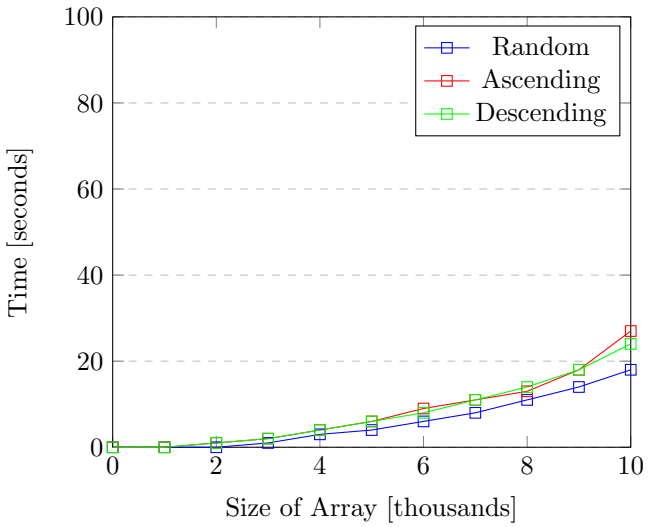
4 Case Studies / Experiment

4.1 Quicksort

Quicksort is a sorting algorithm famous for it's speed. The way it works is quite simple:

- Pick a pivot. This can be any element in the list, but it's ideally in the middle.
- Rearrange the list so that smaller elements come before the pivot and bigger ones come after.
- Repeat for for each subsection, until the entire list has been sorted.

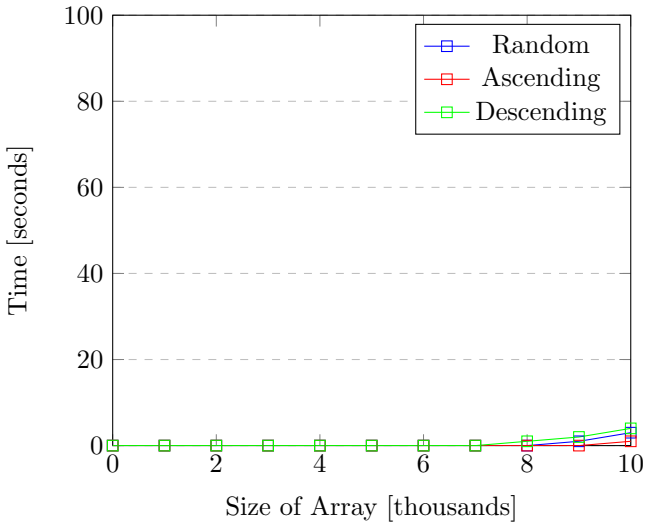
Best	Average	Worst	Space
n log n	n log n	n*n	n log n



4.2 Mergesort

Mergesort, just like Quicksort, uses a technique called divide and conquer. The list is repeatedly divided into two until all the elements are separated individually. Pairs of elements are then compared, placed into order and combined.

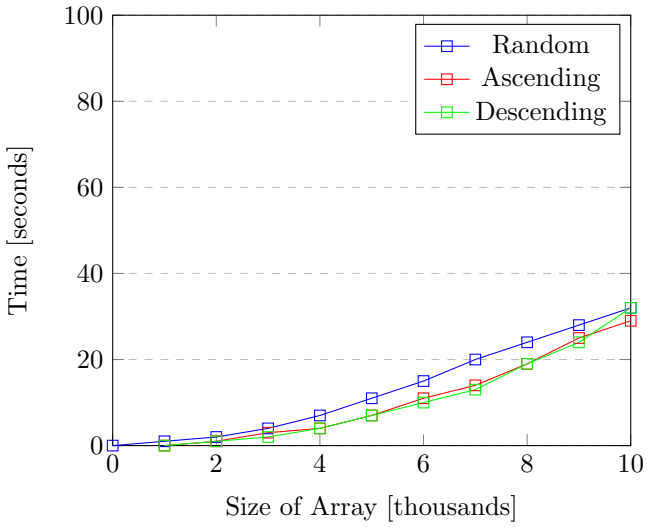
Best	Average	Worst
n log n	n log n	n log n



4.3 Bubblesort

Bubblesort parses an array and compares pairs of elements, swapping them if the first is smaller than the second. It repeats the process until the larger elements "bubble" to the top.

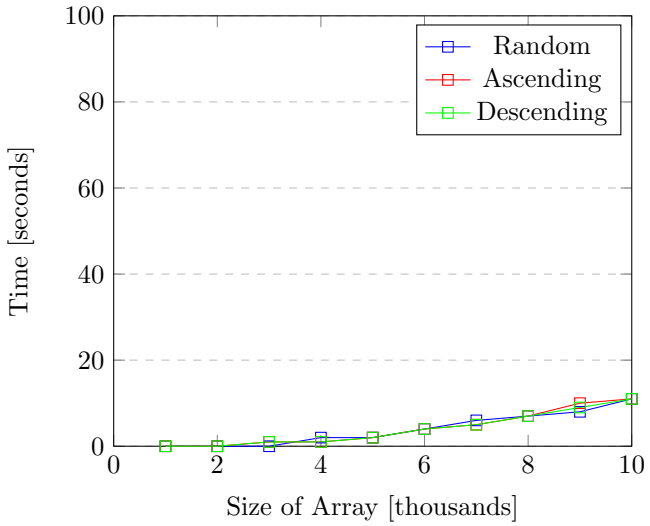
Best	Average	Worst
n	n*n	n*n



4.4 Insertionsort

Insertionsort parses through an array, comparing values in turn, starting with the second value in the list. If this value is greater than the value to the left of it, no changes are made. Otherwise this value is repeatedly moved left until it meets a smaller value. The sort then starts again with the next value.

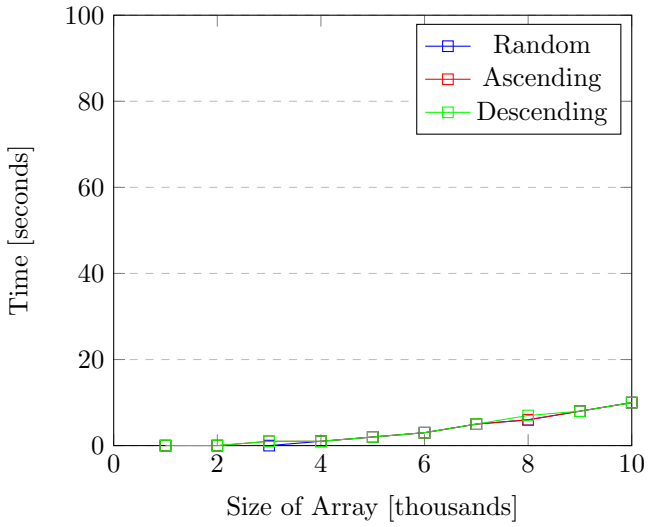
Best	Average	Worst
n	n*n	n*n



4.5 Selectionsort

Selectionsort works by taking an element and bringing it to the front of the array, swapping it with any smaller elements it meets along the way.

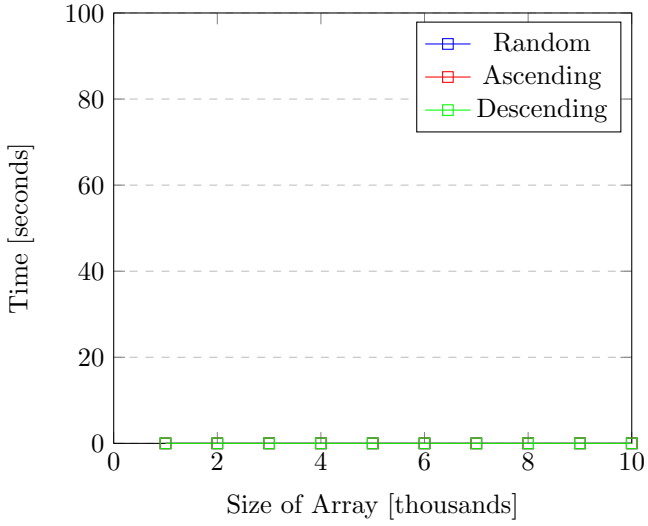
Best	Average	Worst
n*n	n*n	n*n



4.6 Countingsort

Counting Sort counts the occurrences of each unique element in an array and uses this information to sort the elements in ascending order. Because of the way my program has been written, there are only 100 possible integers which can populate the database, and due to that, combined with the efficiency of this algorithm regarding this particular situation, the graph remains flat.

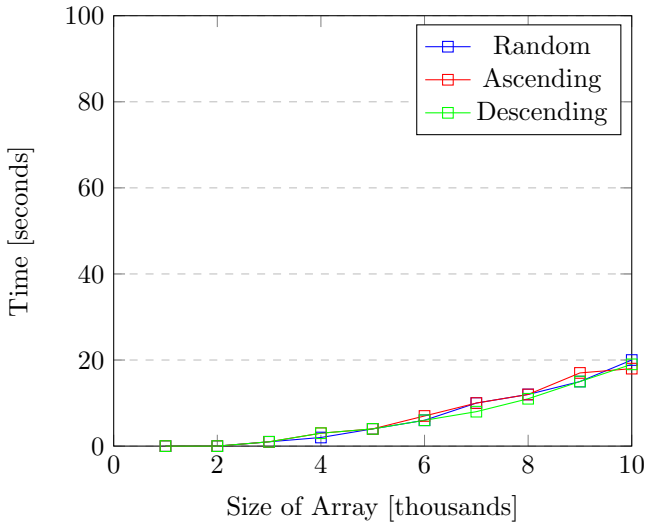
Best	Average	Worst
-	n+ r	n+ r



4.7 Shakersort

Shaker Sort, also known as Cocktail Sort, is a variation of Bubble Sort. It repeatedly traverses the array in both directions, swapping adjacent elements if they are in the wrong order. This bidirectional movement helps to move large elements towards the end of the array and small elements towards the beginning more efficiently than Bubble Sort.

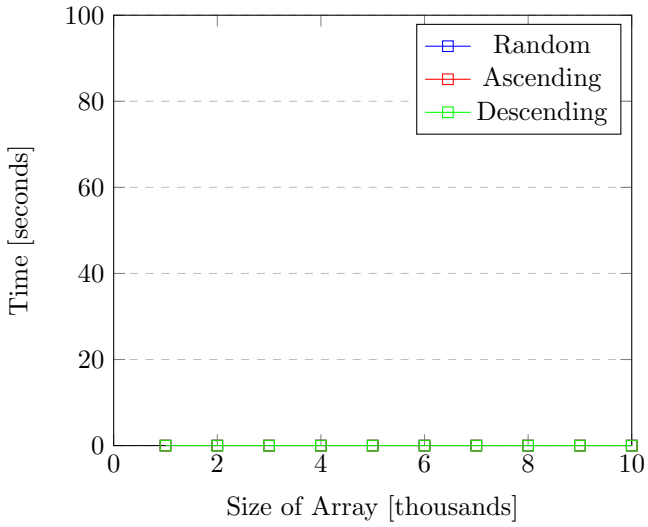
Best	Average	Worst
n	n*n	n*n



4.8 Radixsort

Radix Sort is a non-comparative sorting algorithm that sorts elements based on their digits or characters. It processes the elements digit by digit, from the least significant digit to the most significant digit (or vice versa), distributing them into buckets according to each digit's value. Radix Sort can handle numbers or strings of varying lengths and achieves linear time complexity by using counting sort or bucket sort as its subroutine for each digit.

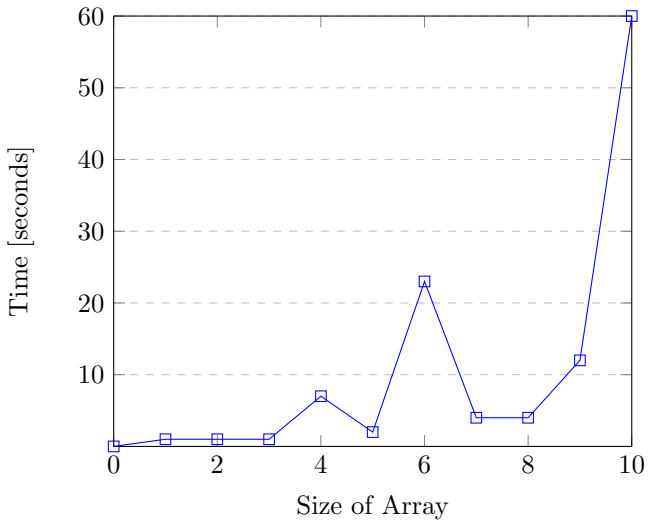
Best	Average	Worst
-	n(k/d)	n(k/d)



4.9 Bogosort

Bogosort repeatedly shuffles the elements of an array randomly and checks if the array is sorted. If not, it continues shuffling and checking until the array becomes sorted.

Best	Average	Worst
1	n*n!	infinite



5 Conclusions

After examining the performance of various sorting algorithms in different scenarios, several conclusions can be drawn regarding their suitability for datasets with specific trends:

1. Trend towards a particular side (Ascending or Descending):
    - For datasets exhibiting such a trend, Insertionsort, Bubblesort (Or it's improved version, Shakersort), and Selectionsort demonstrate efficiency, efficiently handling mostly sorted data with elements predominantly increasing in value.
2. Completely Random Data:
    - In scenarios with randomly ordered data, algorithms like Quicksort and Mergesort are still preferable due to their consistent performance regardless of input order. Their average-case time complexity enables efficient handling of randomness in data distribution.
    - Countingsort and Radixsort are the best choice when the range of values is known and limited.

If one were to believe the multiverse theory, that would imply an universe out there where Bogosort works almost instantly every time, no matter the dataset. Sadly, that is not the universe we find ourselves in, meaning Bogosort is mostly a novelty.

6 References

- Experimental Based Selection of Best Sorting Algorithm
  Profile image of Dharmajee Rao DTVDharmajee Rao DTV
- Introduction to Algorithms, fourth edition
  by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein