Theoretical and Experimental Comparison of Sorting Algorithms

Victor-Mihai David
Email: victor.david04@e-uvt.ro

May 2024

Abstract

This paper provides a comprehensive analysis of various sorting algorithms, comparing their theoretical complexities and practical performances. The objectives include evaluating the efficiency, scalability, flexibility and suitability of each algorithm for different scenarios. Algorithms such as bubble sort, selection sort, insertion sort, merge sort, quick sort, heap sort, counting sort, radix sort, bucket sort, shell sort, cocktail shaker sort, comb sort, and timsort are evaluated both theoretically and experimentally using example datasets. The study aims to identify the most efficient and versatile sorting algorithms while highlighting the trade-offs in algorithm selection for optimizing system performance.

Contents

1 Introduction

This paper presents a theoretical and experimental comparison of various sorting algorithms, focusing on those covered in the ADS (Algorithms and Data Structures) class and other popular ones. The theoretical analysis will examine the time and space complexity of each algorithm. Then, for the more experimental part, the algorithms will be implemented, tested on couple examples, measuring their practical performance, and also analyzing the final results. The results will help us see their behavior in different scenarios.

1.1 Purpose

When talking about sorting something in our day to day life, we usually don't encounter any problems because we deal with low numbers of elements that need to be sorted. Therefore, this action does not take much time or energy to be done. If we have to do the exact opposite, we want to save as much time possible, so finding out the best way to do it, helped by the computer, is the main goal.

1.2 Practical Example

Let's imagine a scenario: you've got more than a hundred people, and you need to organize them by height. Chances are we will mess up at least once, not even thinking about how exhausted someone can become after doing such tasks. Here is the moment where computers come in. A simple algorithm can sort them in less than a second. But what when dealing as a programmer with lengthy data bases? This comparison will find an answer to all of our question.

1.3 More

Sorting algorithms serve a fundamental role in computer science, facilitating efficient ways of organizing data. Understanding each algorithm, meaning the characteristics and performance of the sorting algorithms is crucial. The analysis will try to explore and compare the theoretical and experimental aspects of the many sorting algorithms, managing to draw conclusions in the end.

2 Theoretical Analysis

We will first begin by analyzing the time and space complexity of each sorting algorithm. By doing this, it provides insight into the fundamentals of the algorithms that we mentioned previously. The complexity analysis is very useful because it helps us understand how each algorithm behaves as the size of the input data increases.
More than that, we will try to make a little analysis from theoretical point of view for each algorithm, in order to get a better understanding for each of our algorithms that are used in this paper. For each sorting algorithm I also provided a linked pseudocode that I find easy to follow.

2.1 Bubble Sort

Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
Pseudocode
Time Complexity: O(n^2) in the worst and average case, O(n) in the best case (when the list is already sorted). Space Complexity: O(1) (in-place algorithm).

2.2 Selection Sort

Selection sort divides the list into two parts: sorted and unsorted. It repeatedly selects the minimum element from the unsorted part and swaps it with the first element of the unsorted part.
Pseudocode
Time Complexity: O(n^2) in all cases (worst, average, and best).
Space Complexity: O(1) (in-place algorithm).

2.3 Insertion Sort

Insertion sort builds the final sorted list one element at a time by repeatedly taking the next element from the unsorted part and inserting it into its correct position in the sorted part.
Pseudocode
Time Complexity: O(n^2) in the worst and average case, O(n) in the best case (when the list is already sorted).
Space Complexity: O(1) (in-place algorithm).

2.4 Merge Sort

Merge sort divides the list into smaller lists, sorts them recursively, and then merges the sorted sublists to produce the final sorted list.
Pseudocode
Time Complexity: O(n log n) in all cases (worst, average, and best).
Space Complexity: O(n) (out-of-place algorithm due to merging process).

2.5 Quick Sort

Quick sort selects a pivot element and partitions the list into two sublists such that elements less than the pivot are placed before it and elements greater than the pivot are placed after it. It then recursively sorts the sublists.
Pseudocode
Time Complexity: O(n log n) in the average and best case, O(n^2) in the worst case (unbalanced partitioning).
Space Complexity: O(log n) to O(n) (depends on the implementation and partitioning strategy).

2.6 Heap Sort

Heap sort builds a max-heap from the list and repeatedly extracts the maximum element from the heap, swapping it with the last element, and then heapifies the reduced heap.
Pseudocode
Time Complexity: O(n log n) in all cases (worst, average, and best).
Space Complexity: O(1) (in-place algorithm).

2.7 Counting Sort

Counting sort assumes that each of the elements in the input array has a key value that is within a specific range. It counts the occurrences of each unique key value and uses arithmetic to determine the position of each key in the output sequence.
Pseudocode
Time Complexity: O(n + k), where n is the number of elements in the input array and k is the range of input.
Space Complexity: O(n + k) (requires additional space for the count array).

2.8 Radix Sort

Radix sort is a non-comparative sorting algorithm that sorts numbers by processing individual digits. It sorts the numbers digit by digit from the least significant digit to the most significant digit using counting sort, bucket sort, or other stable sorting algorithms as subroutines.
Pseudocode
Time Complexity: O(nk), where n is the number of elements in the input array and k is the number of passes or digits in the largest number.
Space Complexity: O(n + k) (requires additional space for auxiliary arrays).

2.9 Bucket Sort

Bucket sort divides the input array into a number of buckets, each capable of holding a range of values. The values from the input array are distributed into

these buckets. Each bucket is then sorted individually, either recursively or using another sorting algorithm.
Pseudocode
Time Complexity: O(n + k), where n is the number of elements in the input array and k is the number of buckets.
Space Complexity: O(n + k) (requires additional space for the buckets).

2.10 Shell Sort

Shell sort is an extension of insertion sort that allows the exchange of items that are far apart. It sorts elements by comparing them with other elements that are a certain distance apart and then gradually reducing the gap between elements to be compared.
Pseudocode
Time Complexity: O($nlogn$) in the best case, worst case O(n^2) for some inputs.
Space Complexity: O(1) (in-place algorithm).

2.11 Cocktail Shaker Sort

Cocktail shaker sort is a variation of bubble sort that sorts the elements by repeatedly stepping through the list in both directions, comparing adjacent elements, and swapping them if they are in the wrong order.
Pseudocode
Time Complexity: O(n^2) in the worst case, but it can perform slightly better than Bubble Sort.
Space Complexity: O(1) (in-place algorithm).

2.12 Comb Sort

Comb sort is an improvement over bubble sort. It eliminates small values at the end of the list efficiently by using a gap value. It starts with a large gap value and shrinks it with each iteration until it becomes 1, similar to the diminishing increment gap sequence in shell sort.
Pseudocode
Time Complexity: O(n^2) in the worst case, but it's an improvement over Bubble Sort.
Space Complexity: O(1) (in-place algorithm).

2.13 Timsort

Timsort is a hybrid sorting algorithm derived from merge sort and insertion sort. It first divides the array into small chunks, then sorts these chunks using insertion sort, and finally merges the sorted chunks using merge sort.
Pseudocode
Time Complexity: O(n) for the best case, O(n log n) for average and worst case.
Space Complexity: O(n).

3 Experimental Analysis

3.1 Test Setup

In order to validate all the theoretical analysis and gain practical insights, we implemented each sorting algorithm and conducted experiments using different sets of test data. The experiments regard the main subject of this paper, measuring the time and space complexity. I will provide the time (in seconds) that took for each sorting algorithm to do the sorting for each test. I also want to mention that the tests are being done on my personal computer (a laptop in my case with these **components**), the program I chose is Python and as an IDE I chose PyCharm. More, by this **link**, you can access the code and the experimental results that were used for this paper. These tests will vary on the data sets and my implementation of the sorting algorithms.

3.1.1 First test

We firstly picked 3 lists([1,2,3,4,5,6,7,8,9,10], [10,9,8,7,6,5,4,3,2,1] and [10,3,2,9,7,5,6,4,1]).

	dataset1	dataset2	dataset3
Bubble Sort	0.000007700	0.000010300	0.000010300
Selection Sort	0.000006900	0.000006500	0.000006300
Insertion Sort	0.000003800	0.000007000	0.000006300
Merge Sort	0.000016500	0.000019700	0.000015900
Quick Sort	0.000010100	0.000011700	0.000008900
Heap Sort	0.000010000	0.000011900	0.000009200
Counting Sort	0.000008600	0.000012300	0.000009600
Radix Sort	0.000016000	0.000020800	0.000018000
Bucket Sort	0.000009200	0.000018800	0.000015300
Shell Sort	0.000004100	0.000008500	0.000008800
Cocktail Shaker Sort	0.000002200	0.000010100	0.000010600
Comb Sort	0.000005700	0.000007600	0.000009200
Timsort	0.000001400	0.000001900	0.000001800

Table 1: Results of the first test

We can see that from all 13 algorithms, Timsort easily was the winner for this test, proving great performance across all tests. Merge Sort and Quick Sort demonstrated okay performance, but their multiple operations kept them behind. Bubble Sort, Selection Sort, and Insertion Sort, showed a good performance for small data sets. Shell Sort, Cocktail Shaker Sort and Comb Sort also showed very competitive times, being an improvement over Insertion Sort, respectively Bubble Sort. Heap Sort, Counting Sort, Radix Sort and Bucket Sort show a slow performance, giving the small length of datasets, Radix Sort peforming the worst across all algorithms.

3.1.2 Second test

After looking to the first test results, we decided to make the length of the lists bigger and we moved to list of 100 elements and 1000 elements. We created 1 ordered list of elements going to 100 and then 2 random lists of 100, respectively 1000 elements.

	dataset4	dataset5	dataset6
Bubble Sort	0.000170100	0.000219400	0.029658700
Selection Sort	0.000136500	0.000129800	0.013439800
Insertion Sort	0.000011600	0.000104200	0.013244400
Merge Sort	0.000118100	0.000126600	0.001509800
Quick Sort	0.000356100	0.000051500	0.000679600
Heap Sort	0.000139800	0.000097400	0.004232100
Counting Sort	0.000033000	0.000024100	0.000166100
Radix Sort	0.000070200	0.000049300	0.000630100
Bucket Sort	0.000041500	0.000038500	0.001221600
Shell Sort	0.000054500	0.000054300	0.000887000
Cocktail Shaker Sort	0.000008100	0.000189000	0.026699400
Comb Sort	0.000056800	0.000073500	0.001364500
Timsort	0.000001700	0.000007000	0.000072700

Table 2: Results of the second test

Again, Timsort performed better in all 3 tests. Bubble Sort and Cocktail Shaker Sort performed the worst during these tests because of the number of comparisons and swaps. Selection sort and Insertion sort have the same problem, but show better results. Shell Sort and Comb Sort have similar problems, but their result are better due to optimisation. Merge Sort and Quick Sort have improved results now comparing to the others, giving to their divide and conquer identity. Heap Sort, Counting Sort, Radix Sort and Bucket Sort show a better performance than before, giving the bigger length of the datasets.

3.1.3 Third test

Going further with our analysis, we will increase the numbers once again, testing 3 new lists of numbers that will help us more into seeing different results. The first one will contain 10^4 numbers in order, the next one 10^4 numbers in reverse order, and the last one 10^4 numbers in random order.
After our third test we can see that timsort is still unbeatable in any case. The other result are still similar to the previous ones, but the super fast algorithms such as counting sort or radix sort are doing very well.

	dataset7	dataset8	dataset9
Bubble Sort	1.818410900	3.880747500	3.032171700
Selection Sort	1.256136400	1.292646600	1.276180200
Insertion Sort	0.000628600	2.565158500	1.266513600
Merge Sort	0.015919900	0.016124100	0.017965500
Quick Sort	0.042164200	0.052556100	0.018005500
Heap Sort	0.017174600	0.017501200	0.020352700
Counting Sort	0.002033600	0.001464400	0.001583200
Radix Sort	0.006141600	0.005659900	0.006003900
Bucket Sort	0.001880600	0.229250500	0.125453000
Shell Sort	0.009896400	0.008530900	0.014203500
Cocktail Shaker Sort	0.000579900	3.992808100	2.528204600
Comb Sort	0.016363100	0.016027800	0.018709700
Timsort	0.000033200	0.000053700	0.000704500

Table 3: Results of the third test

3.1.4 Final test

We will now increase again the length of our list. We will have 3 new lists that will have 10^5, 5*$5 * 10^5$ and respectively 10^6 numbers.

	dataset10	dataset11	dataset12
Bubble Sort	313.977339300	N/A	N/A
Selection Sort	153.766274500	N/A	N/A
Insertion Sort	130.199552000	N/A	N/A
Merge Sort	0.225737400	1.289521600	2.515492400
Quick Sort	1.228448700	28.070323100	111.561292900
Heap Sort	0.243954100	1.382170500	2.969788800
Counting Sort	0.015925600	0.075628600	0.153350400
Radix Sort	0.057107800	0.288076500	0.567064000
Bucket Sort	11.975854000	354.901718100	N/A
Shell Sort	0.159352000	0.941988200	2.052929000
Cocktail Shaker Sort	255.593609700	N/A	N/A
Comb Sort	0.237021000	1.422788900	2.843768000
Timsort	0.007479500	0.038650300	0.078730100

Table 4: Results of the final test

After this final set of test data we can finally go further with our analysis and draw much deeper conversations regarding our complex data gained from our tests.

4 Discussion

During 4 sets of data tests, we managed to get perspective on 12 important datasets by finding out how fast each algorithm is doing the same task.

4.1 Bubble Sort

This was the worst sorting algorithm across all 13. The biggest weakness of this algorithm is the number of useless comparisons that are not being optimized. Even though the algorithm is weak, it performed very well on super short data sets,surpassing super fast sorting algorithms, but its usefulness decreases significantly as the length of the data set grew, reaching the point where it would take too much time and there would be no point in using this algorithm. I would recommend this algorithm to any beginner that needs to work with small data sets.

4.2 Selection Sort and Insertion Sort

The results of these two algorithms are similar to the Bubble Sort ones, but the time lengths of each were shorter for both of them. The two of them have an O(n^2) time complexity, but the difference can be easily seen, Insertion Sort always surpassing the other. An interesting result can be seen with ascending and descending ordered lists, Insertion Sort being one of the fastest when the list is ordered ascending, but in descending cases the Selection Sort is actually ahead of it. Unfortunately, neither of them does a great job with larger data sets, making them good only with short data sets.

4.3 Merge Sort and Quick Sort

Some fast algorithms that deserve their status of being fast. They have done their job well from the start. The problem of them is the large number of recursive calls that can slow a lot their time, especially by slowing the computer in my case. Merge Sort was consistent throughout this analysis keeping up with the rest of the algorithms, outperforming Quick Sort, this can be easily seen from the last 2 dataset tests. Quick Sort needed the number of recursive calls available in my IDE increased in order for it to work without any code adaptation. With datasets smaller than 10^6 elements both to their job, but with larger ones Merge Sort wins by a huge margin, making the pivot weakness of Quick Sort easy to be seen.

4.4 Heap Sort

In terms of performance, Heap Sort did very well across all our tests, showing a consistent performance. Its results show a decrease as the length of our datasets increases, thing that is easily seen in the last 2 tests, it surpassing the 1 second mark. Overall, this algorithm could handle even larger dataset sizes, but at some point its performance would become super slow, due to its in-place sorting nature.

4.5 Counting Sort and Radix Sort

The second and the third fastest algorithms in our analysis. This was expected because of their way of counting each element's occurrences, the two of them maintaining the super-fast algorithms identity. Surprisingly, Radix Sort was a little slower than Counting Sort due to its optimisation with the counting sort algorithm that turned out to be very poor. Another interesting thing is the small speed of both of them in the first test compared to the others, but as the length of the data sets grew, they showed out why both are good in handling larger data sets without any problems. Despite lacking significant optimization, both algorithms performed very well.

4.6 Bucket Sort

An algorithm that didn't have a very outstanding performance. Its best work can easily be seen on ascending ordered lists, and the worst on descending ordered lists. This happens because of the bucket's nature. The lists are being divided into buckets, but this slows down the work by a lot. The sorting algorithm for each bucket is Insertion Sort. After looking to the results of each Bucket Sort and Insertion Sort, we can technically consider Bucket Sort a better variant of Insertion sort, but both of them are slow when talking about datasets with more than 10^5 elements. In the end, this algorithm is decent, but would need better handling to exceed further, not being able to handle more complex operations.

4.7 Shell Sort, Cocktail Shaker Sort and Comb Sort

All 3 of them are variants of other slower algorithms, could be considered as better optimisations of them. Shell Sort and Comb Sort are one of the best algorithms when talking about time management, keeping up very closely with the top three algorithms. In most cases Shell Sort surpassed Comb Sort by a small margin. The difference between them and the original sorting algorithms that they are made from is very visible, both of them being remotely 100 times faster than the primary ones. This thing recommend them as doing a good job with larger datasets, being hand in hand with Merge Sort, even Heap Sort from time perspective. When talking about Cocktail Shaker Sort, this one was the second worst one after Bubble Sort, its originator. It worked well when the list was ordered in ascending order, but in all the other cases it showed the second worst performance in our analysis. It was always behind Bubble Sort as the worst sorting algorithm, needing more optimisation in order to make it better.

4.8 Tim Sort

We saved the best for the end. This algorithm surpassed any other algorithm at any point, making it far superior. If we think about its identity, as being a hybrid algorithm well optimised, it is clear to see why python uses this as the default sorting algorithm. It does a great job always being consistent, so it deserves its spot. The only problem this algorithm could have is that it is harder to completly understand comparing to more simpler ones.

4.9 Bonus: Official Ranking of the Algorithms

After looking at the results we managed to get from this analysis, I think a proper ranking of all 13 sorting algorithms with my implementation of them needs to be done in order to be able to see easier which one of them did the best overall.

Name of the sorting algorithm	Rank
Timsort	1
Counting Sort	2
Radix Sort	3
Shell Sort	4
Merge Sort	5
Comb Sort	6
Heap Sort	7
Quick Sort	8
Bucket Sort	9
Insertion Sort	10
Selection Sort	11
Cocktail Shaker Sort	12
Bubble Sort	13

Table 5: Official Ranking of the Algorithms

5 Conclusions and Future Work

In conclusion, this paper managed to provide a comprehensive theoretical and experimental comparison of sorting algorithms, finding different strengths and weaknesses. Each algorithm is put in the same situation and from there I managed to draw different opinions from it.

The 4 tests that I chose were the most relevant ones that I encountered in the making of this paper, ensuring the reliability of our results. I also want to mention that each test was ran multiple times on the same dataset, because sometimes the output was different due to computer's nature.

All tests tables provide perspective on how each algorithm works when arranging different numbers. The part that really caught my attention was how the algorithms handled sorted and unsorted lists. It was fascinating to see how differently they performed, especially in the first and third tables.

This paper managed to expand my level of knowledge of sorting algorithms, accumulating valuable information about each of them. Understanding them was the hardest task, because this was the real challenge in this paper. Seeing the weaknesses and how different each sorting algorithms is, impressed me a lot.

Looking ahead, future research could explore alternative implementations and adaptations of sorting algorithms to further optimize their performance. Additionally, investigating hybrid algorithms may offer new perspectives for improving sorting efficiency.

References

[CLRS, 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). Introduction to Algorithms. MIT Press.

[Sedgewick, 1998] Sedgewick, R. (1998). Algorithms in C++. Addison-Wesley.

[GeeksforGeeks] GeeksforGeeks website. Available online: `https://www.geeksforgeeks.org/`

[Wikipedia] Wikipedia website. Available online: `https://en.wikipedia.org/`