A Theoretical and Experimental Comparison of Fundamental Sorting Algorithms

Maia Marin

West University of Timișoara

Abstract

This paper is divided into two sections: one concerning general information on how several fundamental sorting algorithms work and behave in terms of time complexity, followed by a more practical analysis of said algorithms in the second section.
In the former, the reader may encounter approachable explanations of how each algorithm is designed and their time complexity (more noticeably so in the case of the divide and conquer algorithms) and is provided with further outside information on how these algorithms may be implemented.
In the latter, as methods of analyzing the algorithms' behavior from an experimental standpoint, I've chosen measuring execution time and memory usage. Results in the form of graphs and tables may be viewed in this regard, along with several observations.

Contents

1 Introduction

The problem considered throughout this paper is sorting in ascending order a given array. Its relevance stems from the ever-increasing human need to make sense of large amounts of data.
There are numerous existing solutions, such as well-known and established algorithms, each with their own strengths and weaknesses, depending on the type of data they are "up against".
Throughout this paper, I plan to analyze several fundamental algorithms, which will serve as a basis for further investigation during, but not limited to, my years as a computer science student.
While the algorithms presented in this paper are so widely known that they are rather used as an introduction to the vast problem of sorting, my contribution lies in the practical comparison presented throughout the second part of the paper.
The reader may first browse throughout the first section of the paper for a brief and approachable explanation on how each algorithm works and its time complexity.
For an experimental comparison of each of the chosen algorithms' execution time and memory usage, the reader may visit the second section of the paper, in which they may find bits of the code used in implementation, along with tables and graphs depicting the results.

2 Algorithms: Presentation and Time Complexity Analysis

Proposed problem: Given an array of numbers $a[1...n]$, sort the numbers in ascending order.

2.1 Insertion Sort

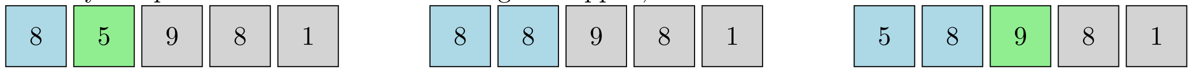
Insertion sort is a iterative algorithm. It works by inserting each element $a[i]$ of the array $a[1...n]$, beginning with the second, in the subarray preceding it such that the subarray a[1...i-1] is going to be already sorted. The elements to be compared are called keys.
The Insertion sort algorithm may be depicted through the following steps:

```
We initialize i, the key's index such that it equal 2 and we iterate from 2
   to n.
  We want to insert the key into the sorted subarray a[1...i-1]. We
    initialize a second index, j, equal to the value i-1.
  As long as j>0 and a[j]>key we do the following:
     If an element with a greater value than the key is found, its value
    gets copied into the next element of the array: a[j+1] = a[j]
     If the condition above is met, the iterator j gets decremented by 1.
  a[j+1], the first element after the already sorted array becomes the new
    key.
```

[Cormen et al.(2022)Cormen, Leiserson, Rivest, and Stein]

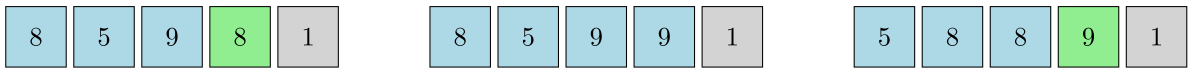
Let's take a concrete example.

- The key is equal to 5. $5 < 8$ so the values get swapped, as illustrated below



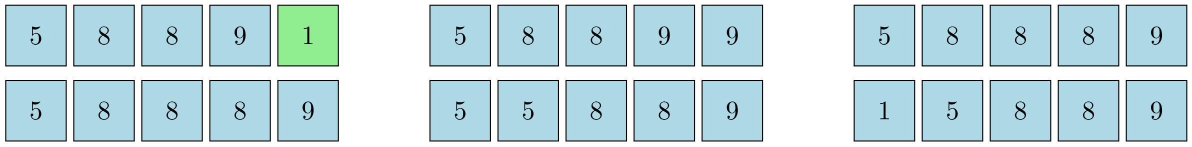
- The key is equal to 9. 9 is greater than all the elements to the left, nothing happens, i gets incremented and 8 becomes the new key.

- The key is equal to 8. $8 < 9$ so the values get swapped.



- The key is equal to 9. 9 is greater than all the elements to the left, nothing happens, i gets incremented and 8 becomes the new key.

- The key is equal to 1. As 1 is less than all the values to the left, a series of values get swapped.

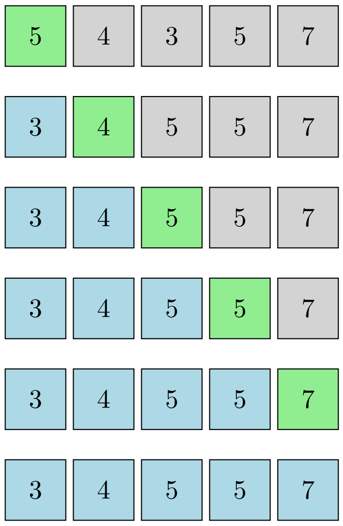


Implementation: An implementation of the Insertion Sort algorithm that illustrates the steps above may be viewed by visiting this link.

Time Complexity: For each $n - 1$ elements, we're going to do at most $n - 1$ comparisons. The time complexity of Insertion Sort is therefore $\Theta(n^2)$.

2.2 Selection Sort

Similarly to Insertion Sort, Selection Sort is an iterative algorithm. Given an array a[1...n], we are going to follow the steps below:



```
For each element in a[1...n]:
  We initialize a variable min
    , equal to the current
    index.
  We iterate through the
    elements to the right of
    our index.
     If a value that's lower
    than the current min is
    found, min gets updated
    with the newfound value.
  After iterating through the
    list, the value at the
    current index gets swapped
     with min.
```

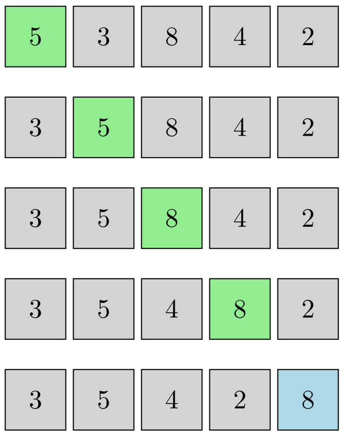
A concrete example may be viewed to the left.

Implementation: An implementation of the Selection Sort algorithm that illustrates the steps above may be viewed by visiting this link.

Time Complexity: For each n elements, we're going to do at most $n-1$ comparisons. The time complexity of Selection Sort is therefore $\Theta(n^2)$.

2.3 Bubble Sort

Similarly to Insertion Sort and Selection Sort, Bubble Sort is an iterative algorithm. Given an array a[1...n], we are going to follow the steps below:



```
For each element a[i] in a
    [1...n]:
  For each element in a[1....(
    length-i)]:
    If a[i] > a[i+1], we swap
    a[i] and a[i+1].
```

To the left we may view a run of the Bubble Sort algorithm.

Implementation: An implementation of the Bubble Sort algorithm that illustrates the steps above may be viewed by visiting this link.

Time Complexity: Similarly to Insertion and Selection Sort, the time complexity of Bubble Sort is $\Theta(n^2)$.. However, it's worth noting that Bubble Sort can be implemented such that the first loop is exited if we encounter no swaps in the first run.
Therefore, in such an implementation, Bubble Sort would have a best-case time complexity of $\Omega(n)$.

2.4 Merge Sort

Merge Sort is a divide and conquer algorithm. This technique allows us to take a complex problem and divide it into smaller and smaller, more aproachable subproblems, then combine each individual result to solve, or "conquer", the original larger problem.
In more exact terms, the algorithm will be designed arround the following steps:

- **Partitioning:** We'll break the sorting problem into smaller sub-problems. A natural way of doing it would be breaking the array $a[0...n]$ evenly in two parts: $a[0...n/2]$, containing the first half of the input data, and $a[n/2+1...n]$, containing the second half.
- **Recursive calls:** We'll recursively call MergeSort to sort the two halves of the array.

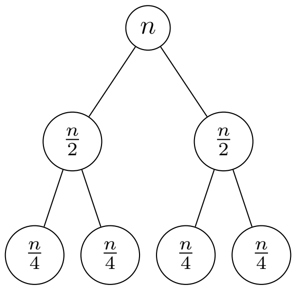
- **Merging:** Now that the subarrays $a[0...n/2]$ and $a[n/2+1...n]$ are both sorted, we'll merge them into a single sorted array. This will be achieved by using three iterators: i for $a[0...n/2]$, j for $a[n/2+1...n]$ and k for updating the elements in the original array. Initially, these iterators will point to the first elements of the three subarrays.
  We'll find the smaller of $a[0...n/2]$ and $a[n/2+1...n]$, update the original array, then move the corresponding iterators (k and either i or j) one step to the right.
  When we reach the end of either of the two subarrays, we'll replace a[k...n] with the remaining elements.

- **Base Case:** When a subarray contains only one number, it is considered already sorted. [Ge(2017)]

Implementation: An implementation of the Merge Sort algorithm that illustrates the steps above may be viewed by visiting this link.

Time Complexity: In order to analyze the time complexity of a recursive algorithm, we need to look at the connection between the execution time of the initial problem and that of the reduced problem. We'll consider:

- the time spent on each level of recursion. The merging step "touches" each element at most one time. In the worse case scenario, when we traverse through two lists of length $n/2$ exhaustively (i.e. we don't reach the end of one list sooner than the other's), the merge function will make $n-1$ comparisons.
  We can see this better examplified in the recursion tree below:



Number of comparisons on each layer:
$1 \cdot (n-1) = 2^0(\frac{n}{2^0} - 1)$
$2 \cdot (n/2-1) = 2^1(\frac{n}{2^1} - 1)$
$4 \cdot (n/4-1) = 2^2(\frac{n}{2^2} - 1)$
...
More generally: $2^i(\frac{n}{2^i} - 1)$, where i is equal to the level of recursion.

Figure 1: Recursion tree, Merge Sort

- the total number of recursion layers. At each recursion step, the sub-problem gets futher divided in two, therefore we'll have a number of log(n) layers of recursion in total and we may express the total number of comparisons through the sum below:

$$\begin{aligned} S &= \sum_{i=0}^{log(n)-1} 2^i(\frac{n}{2^i} - 1) \\ &= \sum_{i=0}^{log(n)-1} (n - 2^i) \\ &= \sum_{i=0}^{log(n)-1} n - \sum_{i=0}^{log(n)-1} 2^i \end{aligned}$$

$$\begin{aligned} \sum_{i=0}^{\log(n)-1} n &= n \cdot (((\log(n) - 1) - 0) + 1) \\ &= n\log(n) \quad (1) \end{aligned}$$

$$\begin{aligned} \sum_{i=0}^{\log(n)-1} 2^i &= 2^{((\log_2(n)-1)+1)} - 1 \\ &= 2^{\log_2(n)} - 1 \\ &= n - 1 \quad (2) \end{aligned}$$

From (1) and (2), we may conclude that $S = n \cdot log(n) + n - 1 \sim n \cdot log(n)$. The time complexity of Merge Sort is therefore $\Theta(n \cdot log(n))$.

2.5 Quick Sort

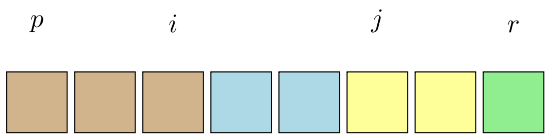
Similarly to Merge Sort, Quick Sort is a divide and conquer algorithm. Let's consider $a[p...r]$ a subarray of $a[0...n]$. The sorting process of $a[p...r]$ may be described through the following steps:

- **Partitioning:** The key step of the Quick Sort algorithm lies in partitioning the sub-array, choosing an element $a[q]$ as pivot and rearranging the array such that $a[p...q-1]$ contains only elements that are less than or equal to $a[q]$, while $a[q+1...r]$ will contain only those elements that are greater or equal to $a[q]$.
  - Typically, the elements that gets chosen as pivot is the rightmost element of the array, in this case $a[r]$.
  - We'll initialize two iterators: i = index of the leftmost element - 1 and $j = 0$.
  - We'll create a loop that iterates through the entire list, as shown in the code below:

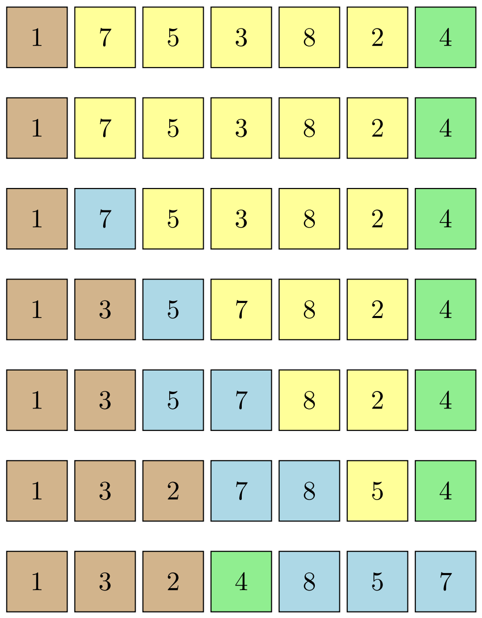
```
    for j in range(leftmost,rightmost):
        # i gets incremented when an element with a value lesser than the pivot's is found
        if list[j] <= pivot:
            i = i+1
            # First element with a greater value than the pivot's gets swapped with new found low element
            (list[i], list[j]) = (list[j], list[i])
    # First element with a greater value than that of the pivot gets swapped with the pivot
    (list[i+1],list[rightmost]) = (list[rightmost],list[i+1])
```

Listing 1: Quick Sort Partitioning

As the loop above runs, each element will fall into exactly one of four regions, some of which may be empty at a given time. They are shown in the figure above as follows:



the elements in the tan area are less or equal to the pivot, the elements in the blue area are greater than the pivot, the status of the elements in the yellow area are unknown, while the green area is occupied by the pivot.
Below I will depict the process of identifying the pivot on a given array:



1. $i = -1, j = 0. a[0] < 4$, therefore i is incremented and we the following swap takes place:$a[0] \leftrightarrow a[0]$
2. $i = 0, j = 1$. $a[1] > 4$, only j gets incremented
3. $i = 0, j = 2$. $a[2] > 4$
4.$i = 0, j = 3$. $a[3] < 4$, therefore i is incremented and the following swap takes place: $a[1] \leftrightarrow a[3]$
5. $i = 1, j = 4$. $a[4] > 4$
6. $i = 1, j = 5$. $a[5] < 4$, therefore i is incremented and the following swap takes place: $a[2] \leftrightarrow a[5]$
7. The for loop is exited and the followin swap takes place: $a[i + 1] \leftrightarrow a[r]$, which in our case is $a[3] \leftrightarrow a[6]$

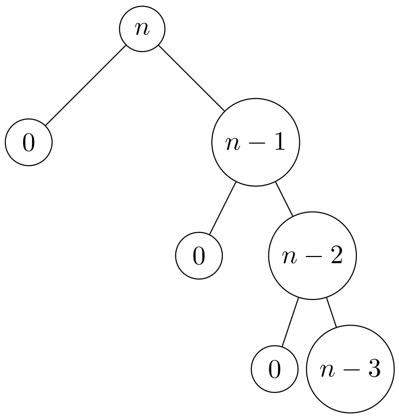
– The Partitioning function will return the index of the pivot, equal to $i + 1$.

- **Dividing:** We'll divide the aray $a[p...r]$ in two subarrays: $a[p...q - 1]$ and $a[q + 1...r]$, where q is the index of the pivot we determined at the previos step.
- **Conquering:** We'll achieve this by calling quicksort recursively to sort each of the subarrays $a[p...q - 1]$ and $a[q + 1...r$.
- **Combining:** The two subarrays are already sorted, no action is needed. All elements in $a[p...q - 1]$ are going to be sorted and less than or equal to $a[q]$and all elements in $a[q + 1...r$ are going to be sorted and greater than or equal to $a[q]$.
  [Cormen et al.(2022)Cormen, Leiserson, Rivest, and Stein]

Implementation: An implementation of the Quick Sort algorithm that illustrates the steps above may be viewed by visiting this link.

Time Complexity: The running time of Quick Sort is greatly dependant on how balanced each partitioning is, which is determined by which element we use as pivot.
If the two sides of the partition are roughly equal in size, the partition is balanced and the algorithm will run in similar time to Merge Sort. However, an unbalanced partition can lead Quick Sort to run as slowly as Insertion Sort.

- **Worst-case partitioning:** Thw worst-case behavior occurs when the partitioning produces one subproblem with $n-1$ elements amd one sub-problem with zero elements.
  Assuming we get such an unbalanced partitioning in each recursive call, we can depict the situation through the recursive tree below:



On each layer of recursion, the problem will get split in two sub-problems of sizes $n-1$ and 0, respectively.
For a problem of size n, we are going to make n-1 comparisons, for a problem of size n-1, we are going to make $((n-1)-1) = n-2$ comparions and so on until we reach a problem of size one, for which we will no one comparison.
Therefore, we may describe the number of comparisons through the sum: $\sum_{i=0}^{n-1} i = \frac{(n-1)\cdot n}{2} \sim n^2$. Therefore, Quick Sort has a worst-case time complexity of O(n^2).

Figure 2: Recursion tree, Quick Sort, Worst-case partitioning

- **Best-case partitioning:** In the most favourable case, the problem is split in two subproblems (one of size $\frac{n-1}{2}$ and one of size $\frac{n-1}{2} - 1$), both lesser in size than $\frac{n}{2}$.
  On each level of recursion, we'll make $2^i(\frac{n}{2^i} - 1)$ comparisons, similarly to Merge Sort, for which I presented the computations in greater detail in the subsection before.
  We can say, therefore, that the best-case time complexity for Quick Sort is $\Omega(n \cdot log(n))$.

3 Experimental Comparison

3.1 Measuring Execution Time

The implementation of the considered algorithms has been provided in the previous section.
The **method used for measuring execution time** may be viewed by visiting this link.

The **types of input** I considered are:

- **Random Order List:** In order to generate a random order list, I used the following code:

```
# Generator function to populate the lists to be sorted with random numbers
def generate_data_set(size):
    for number in range(size):
        yield random.randint(1, 100000)
```

Listing 2: Generate Random Order Lists

Running the sorting algorithms on the generated sets of data has yielded the following results:

	Data Set Size							
Algorithm	**10**	**100**	**1000**	**10000**	**100000**	**1000000**	**10000000**	**100000000**
Selection Sort	0.01 ms	0.18 ms	19.90 ms	1.97 s	4.65 min	–	–	–
Bubble Sort	0.01 ms	0.28 ms	38.40 ms	3.42 s	9.70 min	–	–	–
Insertion Sort	0.01 ms	0.21 ms	16.28 ms	1.63 s	3.89 min	–	–	–
Merge Sort	0.01 ms	0.09 ms	1.05 ms	16.17 ms	206.19 ms	2.58 s	31.73*s*	10.95 min
Quick Sort	0.02 ms	0.12 ms	2.75 ms	13.63 ms	180.24 ms	2.71*s*	1.96 min	-

Table 1: Execution Time Comparison, Random Order Lists

We may see from the graph below that the running time of the iterative algorithms increases significantly faster than that of the divide and conquer algorithms:

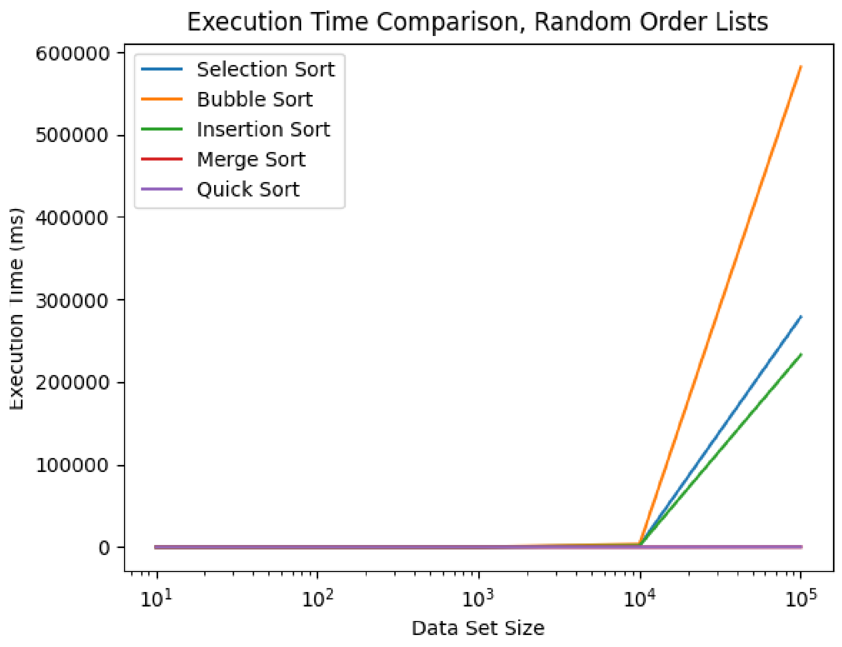


Figure 3: Algorithm Comparison on Random Order Lists

- **Nearly Sorted List:** In order to generate a nearly sorted list, I used the code shown below. The disorder percentage I considered equals 10%.

```python
# Function to generate a nearly sorted list by randomly swapping elements
def generate_data_set(size, disorder_percentage):
    # We firstly generate a sorted list with elements ranging from 1 to size
    sorted_list = list(range(1, size + 1))
    # We compute the number of swaps to be made
    num_disorder = int(size * disorder_percentage)
    for _ in range(num_disorder):
        # We select two random indices
        index1, index2 = random.sample(range(size), 2)
        # We swap the elements found at the indices we previously selected
        sorted_list[index1], sorted_list[index2] = sorted_list[index2],
    sorted_list[index1]
    return sorted_list
```

Listing 3: Generate Nearly Sorted Lists

Running the sorting algorithms on the generated sets of data has yielded the following results:

	Data Set Size							
Algorithm	**10**	**100**	**1000**	**10000**	**100000**	**1000000**	**10000000**	**100000000**
Selection Sort	0.01 ms	0.17 ms	18.7 ms	1.93 s	5.38 min	–	–	–
Bubble Sort	0.01 ms	0.23 ms	26.67 ms	3 s	8.99 min	–	–	–
Insertion Sort	0.01 ms	0.05 ms	4.18 ms	444.27 ms	1.02 min	–	–	–
Merge Sort	0.2 ms	0.17 ms	1.65 ms	22.17 ms	237.86 ms	2.76 s	32.38s	8.39 min
Quick Sort	0.01 ms	0.14 ms	2.56 ms	19.99 ms	436.95 ms	3.24s	1.29 min	13.04 min

Table 2: Execution Time Comparison, Nearly Sorted Lists

- **Reverse Order List:** In order to generate a reverse order list, I used the following code:

```python
# Generating a reverse order list:
def generate_data_set(size):
    sorted_list = list(range(size, 0, -1))
    return sorted_list
```

Listing 4: Generate Reverse Order Lists

Running the sorting algorithms on the generated sets of data has yielded the following results:

	Data Set Size							
Algorithm	**10**	**100**	**1000**	**10000**	**100000**	**1000000**	**10000000**	**100000000**
Selection Sort	0.01 ms	0.19 ms	23.04 ms	1.74 s	3.54 min	–	–	–
Bubble Sort	0.01 ms	0.44 ms	50.93 ms	5.02 s	13.24 min	–	–	–
Insertion Sort	0.01 ms	0.31 ms	37.92 ms	3.92 s	7.34 min	–	–	–
Merge Sort	0.02 ms	0.11 ms	1.21 ms	19.5 ms	195.52 ms	1.98 s	25.05s	9.26 min
Quick Sort	0.01 ms	0.33 ms	20.65 ms	3.01 ms	5.84 min	–	–	-

Table 3: Execution Time Comparison, Reverse Order Lists

Crossing the data gathered for each type of input considered, we obtain the graphs below:

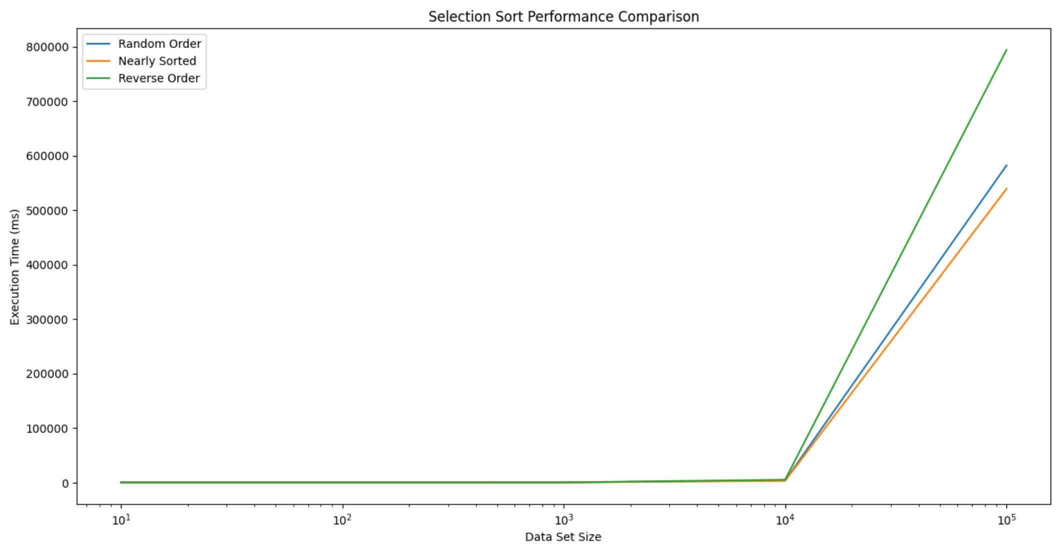


Figure 4: Bubble Sort across several input types

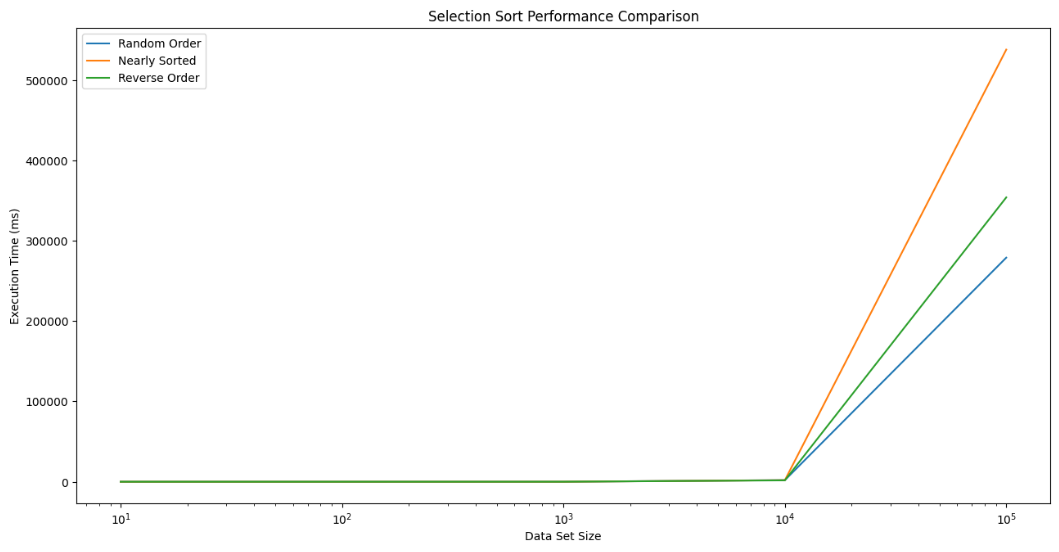


Figure 5: Selection Sort across several input types

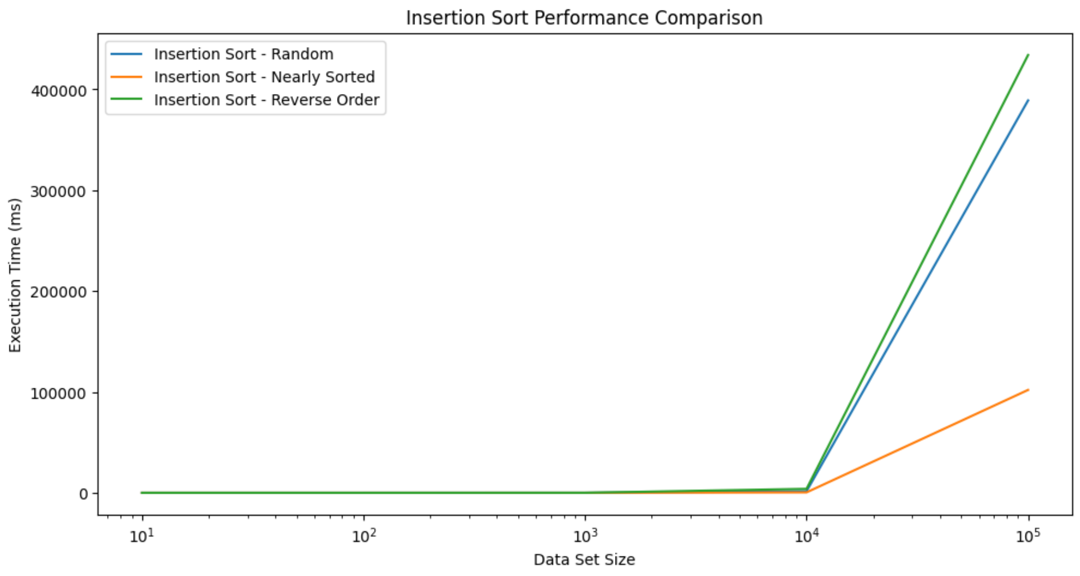


Figure 6: Insertion Sort across several input types

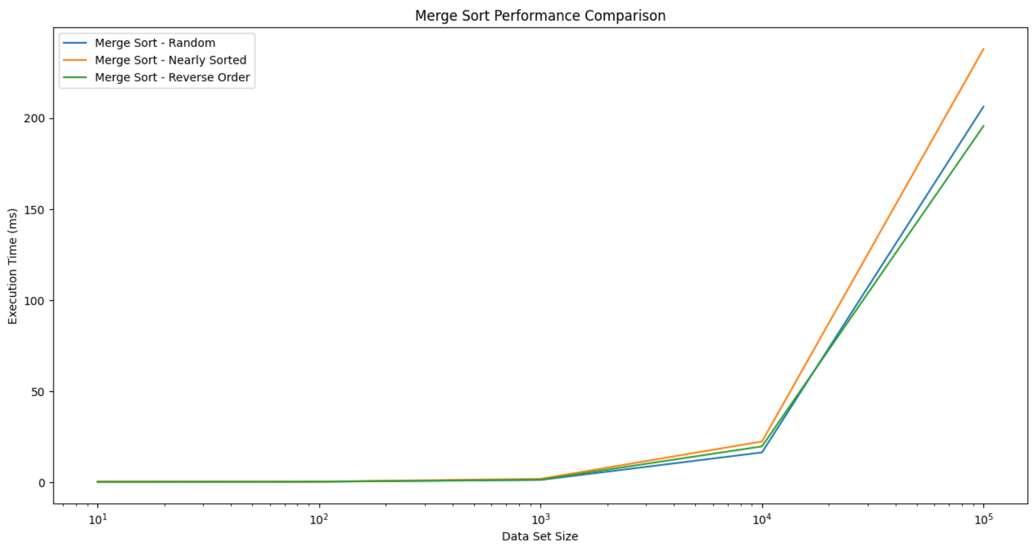


Figure 7: Merge Sort across several input types

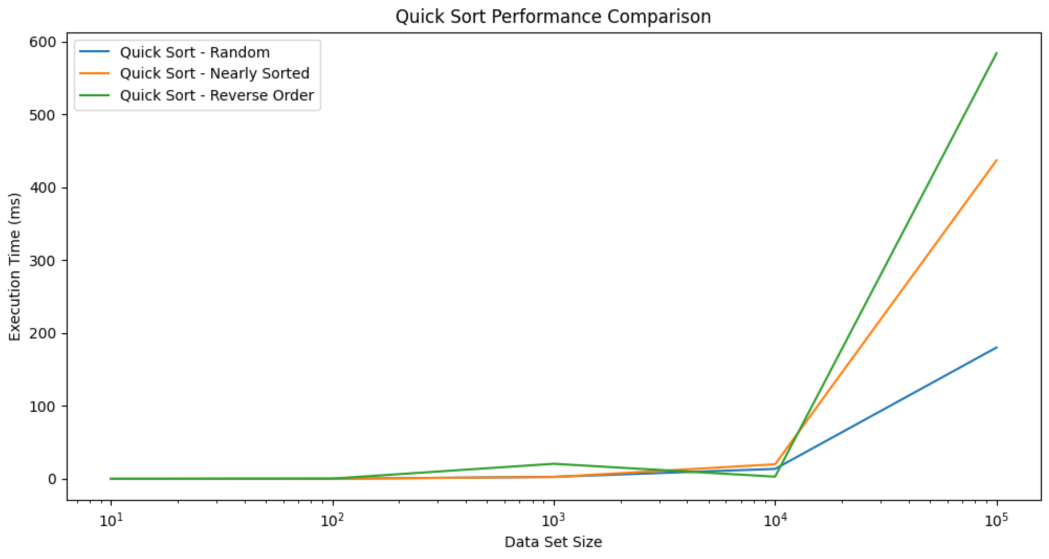


Figure 8: Quick Sort across several input types

While for the majority of the algorithms the differences are minor, we may notice that:

- Bubble Sort behaves as expected, performing best for nearly sorted lists and worst for reverse order lists.
- Selection Sort is slower for nearly sorted lists than for the other two types considered.
- Insertion Sort is significantly quicker for nearly sorted lists.
- Merge Sort has similar running time across all three types considered.
- Quick Sort has similar running time for both nearly sorted and reverse order lists.

3.2 Tracking Memory Usage

In order to track memory usage, I used Memory Profiler, a python module that is able to provide a line-by-line analysis of memory consumption for a given program.
Memory Profile will render a DAT format file, in which each row represent a memory usage measurement taken at regular intervals during the execution of the program.
The code I used in order to parse through the DAT file and compute the overall average memory usage and the maximum memory usage reached at one point during the execution of the program may be viewed by visiting this link.
The results of running Memory Profiler on a program that sorts a random order list of 100000 integers is shown in the table below:

Algorithm:	**Selection Sort**	**Bubble Sort**	**Insertion Sort**	**Merge Sort**	**Quick Sort**
Average Memory Usage	26.96 MB	26.66 MB	27.48 MB	21.49 MB	19.17 MB
Maximum Memory Usage	27.04 MB	26.94 MB	27.55 MB	28.43 MB	27.55 MB
Execution Time	7.91 min	11.73 min	6.31 min	253.52ms	157.94 ms

Table 4: Memory Usage Comparison, Random Order Lists of 100000 elements

4 Related Work

The work this paper is based upon is the book 'Introduction to Algorithms,' written by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
Using this textbook for gathering information on the theoretical aspects of each chosen algorithm, my contributions lie mainly in the experimental part of the paper: implementing the algorithms, methods of measuring execution time and memory usage, and building an experimental comparison of each algorithm's behavior.
While my work is far less comprehensive than the works referenced, it has personal educational value and is something I can use as a point of reference in the future or expand further.

References

[Cormen et al.(2022)Cormen, Leiserson, Rivest, and Stein] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. 2022.

[Ge(2017)] Rong Ge. Course: Design and analysis of algorithms. 2017. Duke University.