Critique Sorting Algorithms and Big-O Analysis

Yiqiao Yin
Ph.D. Student

Abstract

This report investigates and surveys a list of sorting algorithms and discusses how sorting algorithms can affect the performance of software production. The analysis focuses on comparison of different sorting algorithms and their time space analysis using big-O notation.

1 Introduction of Algorithms

This section introduces some famous algorithms in computer science and computer programming. Specifically, the paragraphs investigates the procedure of how these algorithms can be implemented to maximize computation power such as processor performance, data retrieval, processing, storage and distribution, and complex calculations management.

Why sorting? Sorting is a fundamental programming technique. It is a computational and mathematical process to rearrange a list of certain objects in a predefined order. Usually the orders are ascending or descending according to certain numerical value. Many literature have contributed to the survey of sorting algorithm (Estivill-Castro and Wood, 1992; Karunanithi et al., 2014; Martin, 1971; Zutshi and Goswami, 2021). Sorting algorithm provides crucial component of algorithmic solutions in today's software computing. It is also imperative to have consistent computing performance and the development sorting algorithms increases the level of elegance of computing software solutions. Throughout the years, the sorting algorithms have been developed by many scholars and sometimes even similar ones have drastically different performance Zutshi and Goswami (2021).

To assess the performance of sorting algorithm, two common perspectives are discussed. They are time and space analysis. The time analysis refers to the time consumption of a certain algorithm while the space analysis refers to the memory consumption of a certain algorithm. As the definition suggests, it is desirable for computer scientists and programmers to prefer algorithms that are fast but also consumes very little memory if those algorithms are viable choices. There are Big-O and Omega to represent the time complexity Zutshi and Goswami (2021). However, in this report, we focus on the Big-O analysis. This report will examine a list of different sorting algorithms and their time and space analysis will also be discussed in order to make sound comparisons Singh et al. (2018); Martin (1971); Yu and Li (2022); Vitanyi (2007).

These enhancements of sorting algorithms provide fundamental building blocks to mitigate the risks in a large-scale programming project, which is extremely important for today's development of complicated tools in Artificial Intelligence.

2 Why It Works

This section discusses why the sorting algorithms such as bubble sort, quick sort, and insertion sort work the way they do.

Almost all sorting algorithms are recursive. This means that there is some sort of for loop in the algorithm and the lists and sub-lists keep increasing or decreasing in certain direction and sometimes according to certain thresholds. This is accomplished in a way to correct a target's value according to the algorithm. The beautify of these algorithms falls on the nature that a certain action (such as swap or change of order) is carried on when certain thresholds trigger the action. Many dynamic algorithms can be designed when nested for loops are used. To check certain sub-lists of the information, the algorithm traverses through the list according to certain direction and predefined comparisons are evaluated before an action is called.

In addition to the design that enables the algorithms to run continuously, another key component to allow sorting algorithms to work well is that eventually it stops. Every sorting algorithms must have a stopping rule. Otherwise there can be an infinite loop created. There cannot be an infinite loop in the code or the program will run a long time until eventually it exhausts all the available memory and crashes. Stopping rule is the second important thing in algorithmic design. A good stopping rule (or sometimes dropping rule) can increase the efficiency of an algorithm drastically.

3 Different Sorting Algorithms

This section provides discussion of a list of famous algorithms in sorting techniques. They include bubble sorts, quick sorts, and insertion sorts. We discuss of they can effectively be used as support to increase production value and computational efficiency of large-scale programming.

The first candidate is Bubble Sort which is the most fundamental sorting algorithm to introduce in the field. The Bubble Sort also has a nickname called Sinking Sort. The algorithm in this strategy compares the two numbers adjacent together throughout the list and changes their order if they are incorrect. This change of order continues until the repetition ends when all the numbers are compared and sorted. For example, the goal is to sort the list $\{5, 1, 4, 2\}$. The algorithm starts by comparing 5 with 1. Since 5 is greater than 1, the order of these two numbers is changed. Then we have $\{1, 5, 4, 2\}$. Then the algorithm arrives at the number 5 and compares 5 with the next number which is 4. Since 5 is greater than 4, then the algorithm changes the order between 5 and 4. Hence, a new list $\{1, 4, 5, 2\}$ is generated. The last pair is 5 and 2, so 2 gets put in front of the number 5 because the numerical value is smaller. Hence, we have $\{1, 4, 2, 5\}$ for one pass. One pass refers to one round from the first number to the last number. After the first pass, some numbers are sorted correctly, however, some are incorrect still. The algorithm does a second pass. The second pass continues to 4 and 2. Then the algorithm swaps their order to obtain $\{1, 2, 4, 5\}$. The algorithm continues to check and change the order if necessary. The algorithm stops in the next round, because there is nothing that needs to be changed and the algorithm finishes. A sample code, in Python, can be seen below to demonstrate this technique.

```python
# Python program for implementation of Bubble Sort
def bubble_sort(some_array):
    n = len(some_array)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if some_array[j] > some_array[j+1]:
                some_array[j], some_array[j+1] = some_array[j+1], some_array[j]
```

Since the algorithm of Bubble Sort checks every pair and it can result in an exploding number of different passes, it is generally not a go-to choice for programmers. This is because the worst-case performance is $\mathcal{O}(n^2)$ comparisons and $\mathcal{O}(n^2)$. Though the best-case scenario the code can finish within $\mathcal{O}(n)$ comparisons and $\mathcal{O}(1)$ swaps. Sometimes, at extreme case, the time complexity can be $\mathcal{O}(n \log n)$ which is extremely slow value.

The next algorithm to be introduced is called Quick Sort. This type of algorithm is an in-place sorting technique. Originally, it was created by a British computer scientist in 1959. His name is Tony Hoare and he published this work in 1961. This type of sorting algorithm is still very common and received a lot of credibility in the field of computer science. Due to its design, it earns a nickname called "divide and conquer". This is because the algorithm select pivot element and sort in between sub-arrays. Sometimes the process can be considered partition-exchange sort. The worst-case scenario there needs to be $\mathcal{O}(n^2)$ where the best-case scenario it is $\mathcal{O}(n \log n)$. A sample python code that executes Quick Sort is presented below.

```python
# This area is for the python script. In this script, the code implements a quick sort algorithm. The algorithm has a helper function and a main function.

# start here:
def helper(some_array, low_value, high_value):

  # choose the value on the right
  pivot_value = some_array[high_value]

  # redefine the running index i
  i = low_value - 1

  # traverse the list and evaluate all elements
  # make comparisons of the item selected with the pivot_value
  for j in range(low_value, high_value):
    if some_array[j] <= pivot_value:
      # if the item triggers the condition,
      # change the order and then redefine i
      i = i + 1

      # change order or replace i with element at j
```

```
      (some_array[i], some_array[j]) = (some_array[j], some_array[i])

  # change or update pivot_value value
  (some_array[i + 1], some_array[high_value]) = (some_array[high_value], some_array[i + 1])

  # output
  return i + 1

# Function to perform quicksort
def quick_sort(some_array, low_value, high_value):
  if low_value < high_value:

    # Find pivot element such that
    # element smaller than pivot are on the left
    # element greater than pivot are on the right
    pi = helper(some_array, low_value, high_value)

    # Recursive call on the left of pivot
    quick_sort(some_array, low_value, pi - 1)

    # Recursive call on the right of pivot
    quick_sort(some_array, pi + 1, high_value)
```

The third candidate is called Insertion Sort. As the name suggests, this type of sorting algorithm takes an element from a list of an array of numbers and insert it inside a previous subset of array. In other words, the algorithm starts with a random array of numbers that is unsorted and then takes one element one by one to compare with the previous subset. For example, consider an list of numbers $\{5, 1, 4, 2\}$. The algorithm starts with number 5 and since there is only one number then nothing changes. Then this number 5 forms a sub-list for comparisons. Then the algorithm moves on to the next number which is 1. Since 1 is less than 5, the algorithm puts 1 on the left side of 5. As of this step, the list becomes $\{1, 5, 4, 2\}$. The algorithm continues and reads 4. The number 4 is less than 5, so it gets to be moved in front of 5. Then the algorithm compares 1 and 4. Since 4 is greater than 1, there is no need to make any additional changes. Now the list becomes $\{1, 4, 5, 2\}$. The last number that has yet to be reorganized is 2. The algorithm reads 2 and iteratively compare the number 2 with the previous sub-list from large to small. Hence, the number 2 gets placed in between 1 and 4. Thus, we have a sorted list $\{1, 2, 4, 5\}$. An example of code, written in python, is presented below.

```
# This area is for python script. The code presents a function for insertion sort algorithms.

# start here:
def insertion_sort(some_array):

    # Traverse through 1 to len(some_array)
    for i in range(1, len(some_array)):

        key = some_array[i]

        # Move elements of some_array[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
```

```
        j = i-1
        while j >= 0 and key < some_array[j] :
                some_array[j + 1] = some_array[j]
                j -= 1
        some_array[j + 1] = key
```

For Insertion Sort, the worst-case performance is the same as above which is $\mathcal{O}(n^2)$ for both comparisons and change of orders. The best-case is also the same which is $\mathcal{O}(n)$ for comparisons and $\mathcal{O}(1)$ for change of order.

4 Big-O Analysis

This section discusses the Big-O notation and provides a high-level overview of how it works in a search-based algorithm. There are different levels of time space complexity. There are also an overview of Big-O notation.

First, let us introduce some definitions. The Big-O notation measures the efficiency of a designed algorithm. This mathematical expression evaluates the speed and time consumption of the algorithm depends on the length of the data. It is a commonly used notation to quickly indicate the complexity of a data structure and algorithmic design. The function is famous at measuring two efficiencies. They are time and space. Both time and space are important performance measure metrics to allow computer programmers to understand how "good" the algorithm is Devi et al. (2011); Emmanuel et al. (2021); Ghent (2020).

The Big-O notation can also be referred to as the upper bound of a certain algorithm. The upper bound of some time consumption function based on the length of the search is considered as the worst-case scenario. It is indeed the worst-case scenario that is informative, because programmers cannot hope to see best-case scenario all the time. That would be a dangerous assumption to make in production. To truly understand how Big-O notation works in a search algorithm, it is easy to demonstrate it with the following example. Consider a search algorithm that tries to find the number 8 from a list of arrays 1 to 8. A simple one is to look for each item one by one. The algorithm starts with the first number. If it is not 8, then the algorithm continues to 2. It continues the search until the algorithm finds the number 8 in the list. A sample code can be seen below.

```
def linear_search(some_array, some_number):
    for i in range(len(some_array)):
        if some_array[i] == some_number:
            return i
    return -1 # while -1 is commonly representing error code or
        ↪ desired result not found
```

Though the linear search is straightforward and it makes perfect sense, it may not be the most efficient search possible. Another potential way is the binary search. Before we introduce binary search, a simple mathematical concept is important to be of aid here. It is called mean value theorem. The theorem states the following. When a function h is continuous on the closed interval $[z_1, z_2]$ and differentiable on the open interval (z_1, z_2), then there exists a point z_3 in the interval (z_1, z_2) such that the derivative of the function $h'(z_3)$ is equal to the function's average rate of change over $[z_1, z_2]$. Here we assume that the numbers z_1, z_2, z_3 are in real line, i.e. $\forall i \in [1, 2, 3], z_i \in \mathbb{R}$. The proof is essentially a two step process. It cuts from the middle and then it makes a comparison.

The idea can be applied here as well in binary sort. In other words, to take the idea and apply it in search algorithm, the algorithm suggests to take a middle point between 1 to 8. We can take 4 as the middle point. Since 4 is less than 8, that means the number 8 must be on the right half of the sub-list. This means there is no purpose of checking the left half of the sub-list and so we omit the left half. For the right remaining half, the algorithm cut it in half again and this time we can use 6 as mid-point. Since 6 is less than 8, that means we can drop 5 and 6. This means we only have 7 and 8 left. The same algorithm continues until we find where 8 is. In this case, the binary search is more efficient, because it eliminates half of the list in each search round. The worst case scenario for binary search is $\mathcal{O}(\log N)$. A sample code is represented below

```
def binary_search(some_array, some_target):
    left = 0
    right = len(some_array) - 1

    while left <= right:
        midpoint = (left + right) // 2
        if some_target < some_array[midpoint]:
            right = midpoint - 1
        elif some_target > some_array[midpoint]:
            left = midpoint + 1
        else:
            return midpoint
    return -1 # while -1 is commonly representing error code or
        ↪ desired result not found
```

4.1 Time Analysis

This section discuss how Big-O analysis supports processing times for each sorting algorithm. Further, it discuss Big-O alternatives to the management of complex algorithms. Support your discussion with scholarly literature.

The Big-O notation, as demonstrated in earlier paragraphs of this section, can be simplified to manage complex algorithms. In other words, for binary search we use $\mathcal{O}(\log N)$ which is less complex than linear search. Many other algorithms that are more complicated can also exist as well and we use Big-O to analyze their level of complexities. A common example can be the quadratic algorithm which takes $\mathcal{O}(N^2)$. The algorithm has nested loops and it suggests to iterate through the entire data in an outer loop. A list of complexities is presented in Table 1.

Table 1: **List of Different Level of Complexities**

Factorial	Exponential	Cubic	Quadratic	$N \times \log N$	Linear	Logarithmic	Constant
$\mathcal{O}(N!)$	$\mathcal{O}(2^N)$	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N \log N)$	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$

The level of complexity accompanying with the efficiency of speed is presented in the Figure 1 and a more detailed graph is in Figure 2.

A more detailed graph can be seen

References

Devi, S. G., Selvam, K., and Rajagopalan, S. (2011). An abstract to calculate big o factors of time and space complexity of machine code.

Figure 1: **Summary of Big-O Complexity Chart**

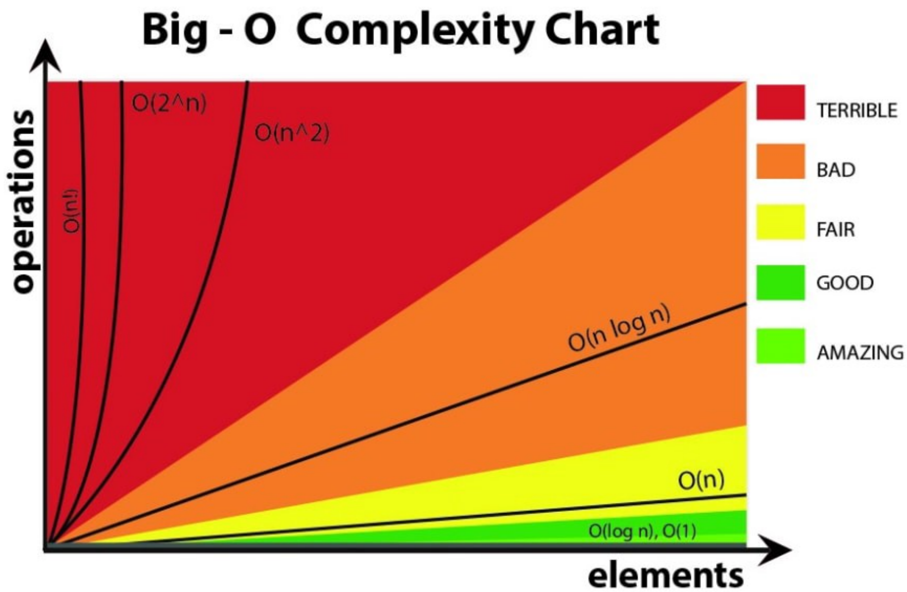
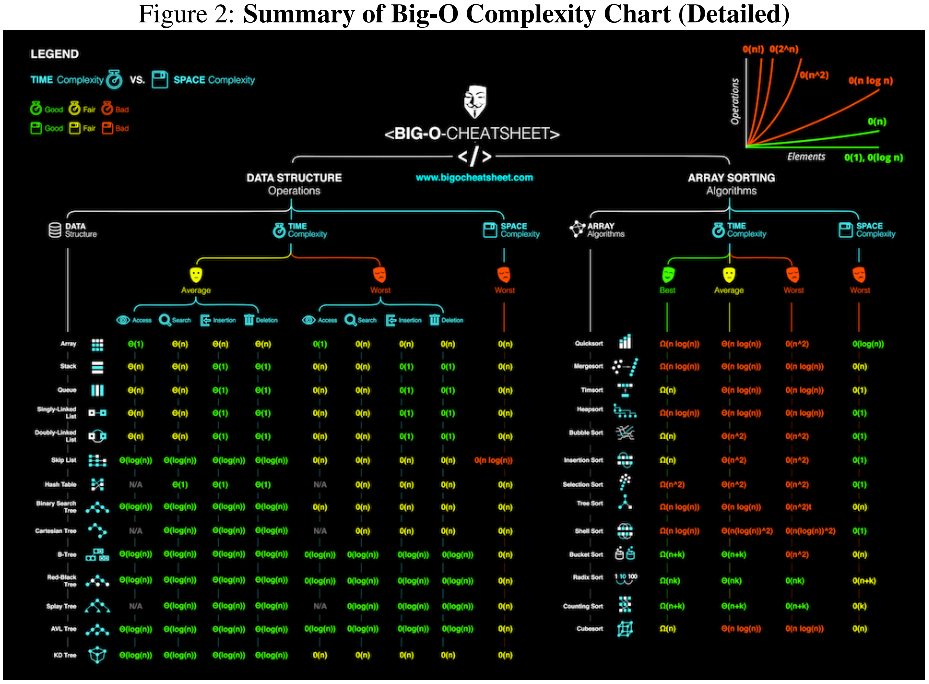


Figure 2: **Summary of Big-O Complexity Chart (Detailed)**



Emmanuel, A. A., Okeyinka, A. E., Adebiyi, M. O., and Asani, E. O. (2021). A note on time and space complexity of rsa and elgamal cryptographic algorithms. *International Journal of Advanced Computer Science and Applications*, 12(7).

Estivill-Castro, V. and Wood, D. (1992). A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)*, 24(4):441–476.

Ghent, K. N. (2020). An introductory survey of computational space complexity.

Karunanithi, A. K. et al. (2014). A survey, discussion and comparison of sorting algorithms. *Department of Computing Science, Umea University*.

Martin, W. A. (1971). Sorting. *ACM Computing Surveys (CSUR)*, 3(4):147–174.

Singh, D. P., Joshi, I., and Choudhary, J. (2018). Survey of gpu based sorting algorithms. *International Journal of Parallel Programming*, 46(6):1017–1034.

Vitanyi, P. (2007). Analysis of sorting algorithms by kolmogorov complexity (a survey). In *Entropy, Search, Complexity*, pages 209–232. Springer.

Yu, T. and Li, W. (2022). A creativity survey of parallel sorting algorithm. *arXiv preprint arXiv:2202.08463*.

Zutshi, A. and Goswami, D. (2021). Systematic review and exploration of new avenues for sorting algorithm. *International Journal of Information Management Data Insights*, 1(2):100042.