Analysis of Time Complexity in Sorting Algorithms

Rinaldy Adin - 13521134
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13521134@std.stei.itb.ac.id

***Abstract*—Sorting algorithms rearrange lists into an ordered manner. As with most algorithms, we have to take into account it's efficiency when running in a program, we measure an algorithms efficiency using the concept of time complexity. Calculating time complexity allows us to know and understand the speed of an algorithm relative to the size of its input and express it using big-O notation. This paper analyzes the time complexity of sorting algorithms and collects data on actual algorithm run time. Our findings conclude that divide-and-conquer sorting algorithms are the most effective to use in most use cases, when compared to slower brute-force algorithms and memory hungry non-comparative sorting algorithms.**

***Keywords*—Sorting Algorithm, Time Complexity, big-O.**

I. Introduction

A sorting algorithm is an algorithm that places the elements of a list into a specific order, whether it be in numerical order, lexicographical order and either ascending or descending. As with most algorithms, there is a requirement to optimize the efficiency of algorithms, such as sorting algorithms, to make sure its usage is as fast and efficient as possible. Due to the frequent need for sorting in computer science applications, there are various sorting algorithms with varying efficiencies and techniques used to implement them.

Sorting algorithms use a wide variety of strategies, including brute-force, divide-and-conquer, advanced data-structures, and many more. The most common and popular algorithms use a brute-force strategy, those algorithms include bubble-sort, insertion-sort, and selection-sort, which are some of the most basic sorting algorithms, thus often used to introduce the concept of sorting algorithms to computer science students. More efficient sorting algorithms often use a dividie-and-conquer strategy, with algorithms such as merge-sort and quick-sort being quicker and more efficient than their brute-force counterparts.

With efficiency being an important factor in determining the usefulness of a certain algorithm, computer scientists use the concept of computational complexity to describe a concise and consistent language around the efficiency of algorithms. The complexity of an algorithm can either express the time complexity, expressing how much time an algorithm will take, or the space complexity, expressing how much memory is used when running the algorithm. With the high capabilities of today's machines, time complexity is more often used to gauge the computational complexity of an algorithm than space complexity as space complexity is less relevant when trying to optimize an algorithm. The time complexity of an algorithm can be expressed in terms of the number of operations used by the algorithm when the input has a particular size. Operations are used as metrics in the time complexity of algorithms instead of the actual time taken to run an algorithm due to the difference in time needed for different computers and languages to perform these basic operations. Using this concept we can conclude the efficiency of algorithms and analyze the use-casses in which these algorithms should be used in.

II. Theoretical Basis

A. Time Complexity

The measure of time complexity stems from the number of operations in an algorithm when recieving an input of size n. The operations used to measure time complexity can be the comparison of integers, the addition of integers, the multiplication of integers, the division of integers, or any other basic operation. We express this using the expression $T(n)$.

Since there are many cases for certain algorithm that yield different time complexities, we further derive the time complexity of algorithms as $T_{max}(n)$ to express the maximum time complexity in a worst-case scenario, $T_{avg}(n)$ to express the average time complexity, and $T_{min}(n)$ to express the minimum time complexity in a best-case scenario.

As the actual number of operations done in a specific case when using an algorithm varies due to other factors other than the size of the input, big-O notation is commonly used to simplify the comparison of time complexity between algorithms. Big-O notation defines the growth of a function relative to its input. For example, using big-O notation we can conclude that the function $T_1(n) = 100n^2 + 17n + 4$ is $O(n^2)$ and the function $T_2(n) = n^3$ is $O(n^3)$. With that information, we know that $T_1(n)$ is grows slower thatn $T_2(n)$. The formal definition of big-O notation is "Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that $|f(x)| \leq C|g(x)|$ whenever $x > k$." Intuitively, the definition that $f(x)$ is $O(g(x))$ says that $f(x)$ grows slower that some fixed multiple of $g(x)$ as x grows without bound, with $g(x)$ ideally being as small as possible to be able to compare a fuction's growth effectively.

When applied to computer science, we can calculate the time complexity of an algorithm and then compare it with the time complexity of another by looking at which one grows slower when receiving a bigger input of n. In this context, calculating the big-O of most algorithms in computer science yields the

following complexities.

Table 1, Classification of time complexities

No.	Complexity	Name	Efficiency
1.	$O(1)$	Constant	Excellent
2.	$O(log(n))$	Logarithmic	Excellent
3.	$O(n)$	Linear	Great
4.	$O(n\ log(n))$	n log(n)	Favourable
5.	$O(n^2)$	Quadratic	Slow
6.	$O(n^3)$	Cubic	Slow
7.	$O(a^n)$	Exponential	Extremely slow
8.	$O(n!)$	Factorial	Terrible

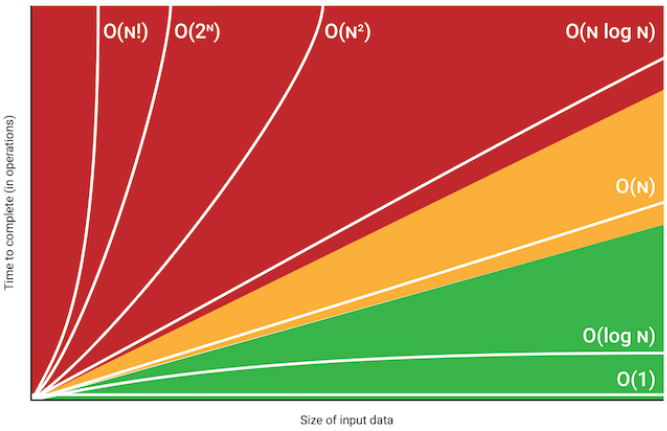

Figure 1, Visualizations of asymptotic time complexity function growth
Source: https://ajakcyer97.medium.com/big-o-time-complexity-graph-simplified-798f3b67877a

B. Bubble Sort

Being the introductory algorithm when learning sorting algorithms, bubble sort uses a brute-force strategy to sort a list into ascending/descending order that can be summarized into these 5 steps:

1. Point to two consecutive values in the array. (Initially, we start by pointing to the array's first two values.)
2. If the two items are out of order (in other words, the left value is greater than the right value), swap them (if they already happen to be in the correct order, do nothing for this step.)
3. Move the two "pointers" one cell to the right.
4. Repeat Steps 1 through 3 until we reach the end of the array, or if we reach the values that have already been sorted. (This will make more sense in the walk-through that follows.) At this point, we have completed our first pass-through of the array. That is, we "passed through" the array by pointing to each of its values until we reached the end.
5. We then move the two pointers back to the first two values of the array, and execute another pass-through of the array by running Steps 1 through 4 again. We keep on executing these pass-throughs until we have a passthrough in which we did not perform any swaps. When this happens, it means our array is fully sorted and our work is done.

To easier visualize the algorithm, the following vizualization shows steps 1 to 4 (one "pass-through" of the list) where these steps will be repeated until we finally get a fully ordered list.

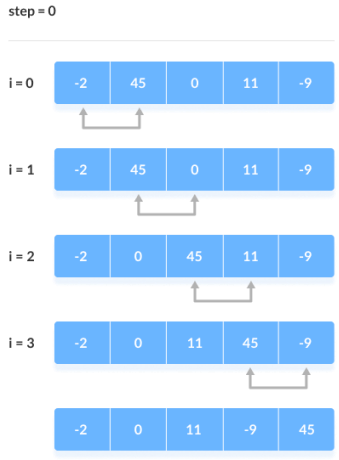

Figure 2, Visualization of a pass-through in bubble sort
Source: https://www.programiz.com/dsa/bubble-sort

C. Selection Sort

Selection sort is another sorting algorithm that uses a brute-force strategy. The algorithm can be summarized into the following steps:

1. We check each cell of the array from left to right to determine which value is least. As we move from cell to cell, we keep track of the lowest value we've encountered so far.
2. Once we've determined which index contains the lowest value, we swap its value with the value we began the pass-through with. This would be index 0 in the first pass-through, index 1 in the second pass-through, and so on.
3. Each pass-through consists of Steps 1 and 2. We repeat the pass-throughs until we reach a pass-through that would start at the end of the array. By this point, the array will have been fully sorted.

To easier visualize the algorithm, the following vizualization shows steps 1 to 2 where these steps will be repeated until we finally get a fully ordered list.

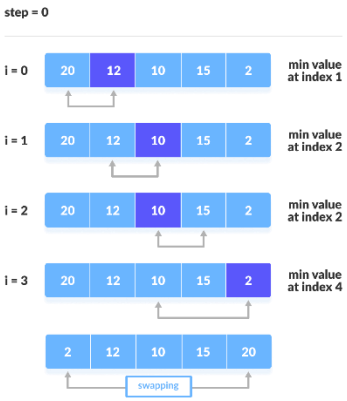

Figure 3, Visualization of a pass-through in selection sort
Source: https://www.programiz.com/dsa/selection-sort

D. Insertion Sort

Selection sort is another sorting algorithm that uses a brute-force strategy. The algorithm can be summarized into the following steps:

1. In the first pass-through, we temporarily remove the value at index 1. In subsequent pass-throughs, we remove the values at the subsequent indexes.
2. We then begin a shifting phase, where we take each value to the left of the gap and compare it to the value in the temporary variable. If the value to the left of the gap is greater than the temporary variable, we shift that value to the right. As we shift values to the right, inherently the gap moves leftward. As soon as we encounter a value that is lower than the temporarily removed value, or we reach the left end of the array, this shifting phase is over.
3. We then insert the temporarily removed value into the current gap
4. Steps 1 through 3 represent a single pass-through. We repeat these passthroughs until the pass-through begins at the final index of the array. By then, the array will have been fully sorted.

To easier visualize the algorithm, the following vizualization shows steps 1 to 3 where these steps will be repeated until we finally get a fully ordered list.

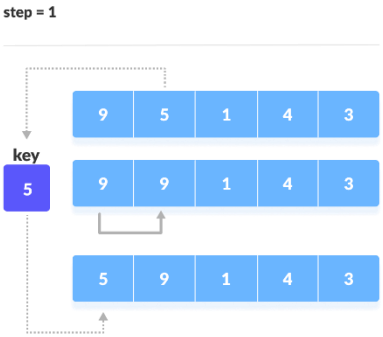

Figure 4, Visualization of a pass-through in insertion sort
Source: https://www.programiz.com/dsa/insertion-sort

E. Counting Sort

Counting sort is a sorting algorithm that uses a simple data structure approach. Compared to the other algorithms we have discussed counting sort is a non-comparative sorting algorithm, meaning that it doesn't compare between two elements to sort it's contents. The algorithm can be summarized into these steps:

1. Find out the maximum element in the given list.
2. Prepare a "count" list, with its elements numbered from 0 to max.
3. Store the count or frequency of every element at their respective indexes in the "count" list.
4. Find the index of each element of the original list in the count list and place the element into the original list.
5. Decrease its count in the count array by one.

To easier visualize the algorithm, the following vizualization shows an example list, its count list, and the resulting sorted list.

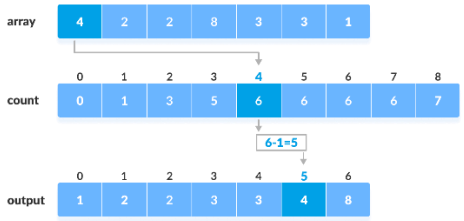

Figure 5, Visualization of counting sort
Source: https://www.programiz.com/dsa/counting-sort

F. Merge Sort

Compared to the sorting algorithms that we haved discussed, merge sort uses a more complex strategy called divide-and-conquer. That is because the algorithm divides the problem, in this case, the list, until finally doing comparisons of its elements until it finally results in a sorted list. The algorithm can be summarized into these steps:

1. Divide the list into halves and record the initial length of the list. Repeat recursively until the recorded lenth is equal to one.
2. Store the element of both halves into two temporary lists, which will only be length one on the lowest divisions, on higher divisions both lists will already be in ascending order.
3. Insert the contents of the two temporary lists into the main list by inserting the lower uninserted elements between the two lists.
4. Repeat step 2 and 3 until the whole list is sorted.

To easier visualize the algorithm, the following vizualization shows how merge sort divides the list by two and then builds the list back up.

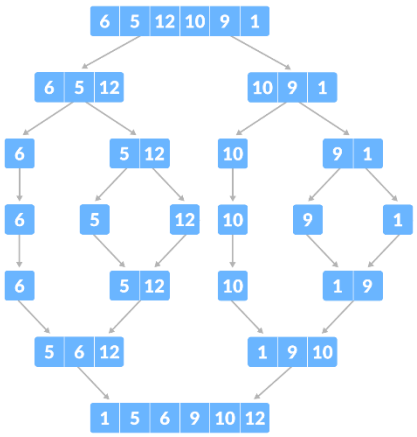

Figure 6, Visualization of merge sort
Source: https://www.programiz.com/dsa/merge-sort

G. Quick Sort

Quick sort also uses divide-and-conquer to sort a lists elements into order. Just like merge sort, quick sort achieves a sorted list by halving the problem until reaching a base case. Quick sort can be summarized into these steps:

1. Choose the rightmost value int the list as a pivotand assign left and right pointers to the array (excluding the pivot).
2. Move the left pointer to the right until it reaches a value that is greater than or equal to the pivot/

3. Then, the right pointer continuously moves one cell to the left until it reaches a value that is less than or equal to the pivot, and then stops. The right pointer will also stop if it reaches the beginning of the array.
4. If the left pointer is to the left of the right pointer, repeat steps 1 until 3. If not, continue to step 5.
5. Swap the pivot with the value the left pointer is pointing to.
6. Treat the subarrays to the left and right of the pivot as their own arrays and recursively repeat steps 1 until 6.
7. When we have a subarray with zero or one element, do nothing.

To easier visualize the algorithm, the following vizualization shows how quick sort rearrange and partitions the list until reaching an ordered list.

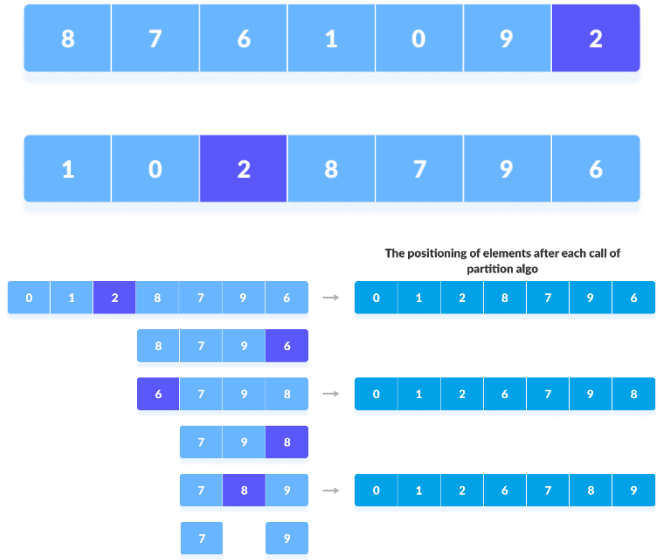

Figure 7, Visualization of quick sort
Source: https://www.programiz.com/dsa/quick-sort

III. METHODOLOGY

In order to test the validity of the resulting big-O calculations, we will use python on jupyter notebook to run each sorting algorithm and analyze the resulting computation time. The analysis is done by running every sorting algorithm with increasing size of the randomized input array, and then recording and plotting the computation time of each run into a graph. If the big-O calculations were correct, the resulting graph should have the same shape as it's asymptotic complexity.

Due to the random nature of our input array, there will often be cases where the input array causes a worst-case scenario to the specific algorithm. To be able to get the desired average computation time, we will repeat the algorithm 20 times and choose the median as the average computation time for the sorting algorithm when sorting an array of length n.

The following is python code to record the computation time of a merge sort algorithm and repeat it 20 times:

```
index = [n for n in range(1,(10**3),1)]
result = []

for n in index:
    data = randomArray(n)
    size = len(data)-1
    t = Timer(lambda: mergeSort(data))
    result.append([t.timeit(number=1)])

for j in range(19):
    i = 0
    for n in index:
        data = randomArray(n)
        size = len(data)-1
        t = Timer(lambda: mergeSort(data))
        ti = t.timeit(number=1)
        result[i].append(ti)
        i += 1

median = []
for res in result:
    res.sort()
    median.append(res[9])
```

After getting the resulting computation time, we will have to compare the results on the plot with its corresponding asymptotic function. Since the plot is mapped to an x-axis of n (the size of the input array) and a y-axis of $T(n)$ (thecomputation time of the algorithm), we will have to calculate the magnitude of the plotted asymptotic function. In order to calculate the magnitude, we first observe the desired $T(n)$ of the asymptotic graph of $O(f(n))$,

$$T(n) = a\, f(n)$$

To get the magnitude a, we will use the computation time $T(n)$ of the input array with the biggest size n.

$$a = \frac{T(n)}{f(n)}$$

The resulting magnitude a will roughly equal the computation time of a single operation in the algorithm. Using a we can plot the asymptotic function that correlates to the time complexity of the algorithm, allowing us to compare our calculated big-O notation with the actual run time of the algorithm.

The following is python code to calculate the magnitude and to create an asymptotic function for $O(n\ logn)$:

```
import math
clock_time = result[-1][9] / (len(result) *
                              math.log2(len(result)))

nlogn = []
for i in range(1, len(result) + 1):
    nlogn.append(i * math.log2(i) * clock_time)
```

IV. BUBBLE SORT TIME COMPLEXITY

Bubble sort works by traversing and swapping elements in the array repeatedly. Each pass-through will take $O(n)$ operations and since our algorithm stops if it does a pass through that did not do any swaps, then we can conclude that in the best-case where the algorithm only does one pass-through, bubble sort will have a time complexity of $O(n)$ and in the average and worst-case where the algorithm will do roughly n pass-throughs, bubble sort will have a time complexity of $O(n^2)$.

The following is python code of the bubble sort algorithm:

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n-1):
        swapped = False
        for j in range(n-1):
            if arr[j] > arr[j + 1]:
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = True;
```

```
        if (not swapped):
            break
```

The time complexity of bubble sort for an array up to length $n = 500$ is plotted in figure 8. When compared to its asymptotic time complexity of $O(n^2)$, since the graphs match, we can conclude that the time complexity of bubble sort is $O(n^2)$.

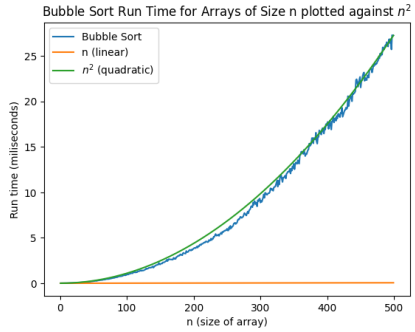

Figure 8, Bubble sort run times for array of size n
Source: writer's analysis

V. Selection Sort Time Complexity

For each index in an array, selection sort will swap its element with the smallest element in the subarray to it's right. Searching for the smallest element will take $O(n)$, when repeated n times, selection sort's time complexity is $O(n\text{^}2)$. For a subarray with n elements, the algorithm will run exactly the same steps everytime, causing it to have no worst-case or best-case, so the algorithm will always run in $O(n^2)$ time.

The following is python code of the selection sort algorithm:

```
def selectionSort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j

        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

The time complexity of selection sort for an array up to length $n = 500$ is plotted in figure 9. When compared to its asymptotic time complexity of $O(n^2)$, since the graphs match, we can conclude that the time complexity of selection sort is $O(n^2)$.

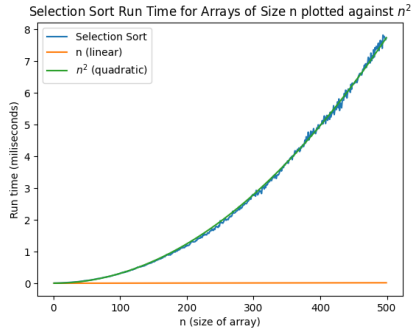

Figure 9, Selection sort run times for array of size n
Source: writer's analysis

VI. Insertion Sort Time Complexity

For each index in an array, inserstion sort will traverse the subarray to its left and swapping every element until it finds the original index's correct spot. Traversing the array will take $O(n)$ time and since it will be repeated for all elements in the array, insertion sort has a time complexity of $O(n^2)$.

The following is python code of the insertion sort algorithm:

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1

        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1

        arr[j + 1] = key
```

The time complexity of insertion sort for an array up to length $n = 500$ is plotted in figure 10. When compared to its asymptotic time complexity of $O(n^2)$, since the graphs match, we can conclude that the time complexity of insertion sort is $O(n^2)$.

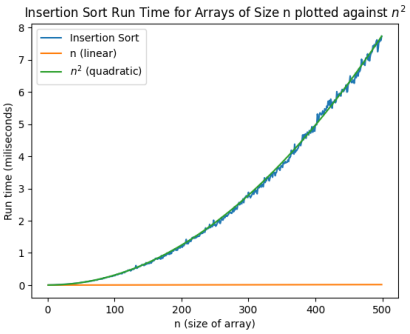

Figure 10, Insertion sort run times for array of size n
Source: writer's analysis

VII. Counting Sort Time Complexity

Since counting sort is non-comparative, the algorithm mostly consists of linear traversals on an array, which takes O(n) time. However, due to the creation and traversal of a count array that has the size k, the maximum value in the original array, the algorithm also does some traversals in $O(k)$ time. Knowing that, we could conclude that counting sort has a time complexity of $O(n + k)$, but since we are interested in the asymptotic function of the algorithm's time complexity, we could also say that counting sort has a complexity of $O(n)$ since the algorithm has linear growth.

The following is python code of the counting sort algorithm:

```
def count_sort(arr):
    max_element = int(max(arr))
    min_element = int(min(arr))
    range_of_elements = max_element - min_element + 1

    count_arr = [0 for _ in range(range_of_elements)]
    output_arr = [0 for _ in range(len(arr))]

    for i in range(0, len(arr)):
        count_arr[arr[i]-min_element] += 1

    for i in range(1, len(count_arr)):
        count_arr[i] += count_arr[i-1]
```

```
    for i in range(len(arr)-1, -1, -1):
        output_arr[count_arr[arr[i] - min_element] - 1]
                                                = arr[i]
        count_arr[arr[i] - min_element] -= 1

    for i in range(0, len(arr)):
        arr[i] = output_arr[i]
```

The time complexity of counting sort for an array up to length $n = 5000$ is plotted in figure 11. When compared to its asymptotic time complexity of $O(n)$, since the graphs match, we can conclude that the time complexity of counting sort is $O(n)$.

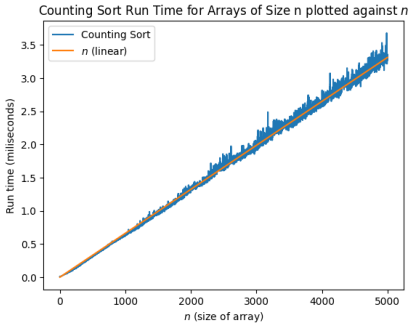

Figure 11, Counting sort run times for array of size n
Source: writer's analysis

VIII. Merge Sort Time Complexity

When analyzing merge sort's divide-and-conquer strategy, we can infer that, for every element in the array, the element will be compared and inserted into a sorted array or subarray $logn$ times. That is because the array is divided into halves to form subarrays and then each subarray's elements will be compared and inserted. Knowing that, we can conclude that merge sort has a time complexity of $O(n\ logn)$.

The following is the python code of the merge sort algorithm:

```
def mergeSort(arr):
    if len(arr) > 1:

        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]

        mergeSort(L)
        mergeSort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

The time complexity of merge sort for an array up to length $n = 1000$ is plotted in figure 12. When compared to its asymptotic time complexity of $O(n\ logn)$, since the graphs match, we can conclude that the time complexity of merge sort is $O(n\ logn)$.

Figure 12, Merge sort run times for array of size n
Source: writer's analysis

VIII. Quick Sort Time Complexity

Similar to merge sort, quick sort halves the array and recursively repeats the operation until it reaches a subarray that has length one or zero. With the same principles as merge sort, we can also find that every element in the array will be compared by the algorithm as much as $logn$ times. So we can conclude that quick sort has a time complexity of $O(n\ logn)$.

The following is the python code of the quick sort algorithm:

```
def partition(array, low, high):
    pivot = array[high]
    i = low - 1

    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])

    (array[i + 1], array[high]) = (array[high], array[i
+ 1])

    return i + 1

def quickSort(array, low, high):
    if low < high:
        pi = partition(array, low, high)

        quickSort(array, low, pi - 1)
        quickSort(array, pi + 1, high)
```

The time complexity of quick sort for an array up to length $n = 1000$ is plotted in figure 13. When compared to its asymptotic time complexity of $O(n\ logn)$, since the graphs match, we can conclude that the time complexity of quick sort is $O(n\ logn)$.

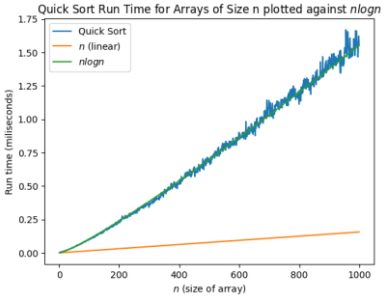

Figure 13, Quick sort run times for array of size n
Source: writer's analysis

IX. Conclusion

From the time complexities that we have calculated and the recorded run time done from our tests, we can confidently say that using the right algorithm is important in optimizing the performance of a program. In our cases, we have found that when doing comparative based sorting, it is best to use the more efficient quick sort and merge sort that have a time complexity of $O(n\,logn)$. Our run time testing also show a significant difference in computation speed between divide-and-conquer algorithms with brute-foce algorithms, such as bubble sort, insertion sort, and selection sort that all have an average time complexity of $O(n^2)$. While counting sort has an average speed that is much more faster than any other algorithm that we have discussed, it's memory usage is much more significant and might affect the program that it is running on.

In most programming languages however, programmers rarely have to consider the sorting algorithm to use as almost all modern languages already have sorting functions that come with it's standard library. These sorting functions often use a mix of sorting algorithms to make it's implementation even faster in the average use case. For example, python's standard sorting function uses Timsort, a mix between merge sort and insertion sort, that has $O(n\,logn)$ average and worst-case time complexity and $O(n)$ best-case time complexity, making it better than all the comparison based algorithms that we have discussed.

VII. Acknowledgment

I wish to show my appreciation to Dr. Fariska Zakhralativa Ruskanda for this assignment, as this assignment has encouraged me into researching deeper about discrete maths and algorithm strategies. I wish to extend my appreciation to the GeeksforGeeks community for providing and maintaining an extensive and accessible archive of computer science knowledge on the internet.

References

[1] Cyer, A. (2022, January 7). Big O — Time Complexity Graph Simplified - Ajak Cyer. Medium. Retrieved December 10, 2022, from https://ajakcyer97.medium.com/big-o-time-complexity-graph-simplified-798f3b67877a

[2] GeeksforGeeks. (2022, December 9). Time Complexities of all Sorting Algorithms. Retrieved December 10, 2022, from https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

[3] Learn Data Structures and Algorithms. (n.d.). Retrieved December 10, 2022, from https://www.programiz.com/dsa

[4] Rosen, K. (2011). Discrete Mathematics and Its Applications Seventh Edition (7th ed.).

[5] McGraw Hill. Wengrow, J. (2020). A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills. Pragmatic Bookshelf.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Desember 2022

Rinaldy Adin 13521134