Activity No. 14.1	
Algorithm Complexity	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 05/11/2025**
Section: CPE21S4	**Date Submitted: 05/11/2025**
Name(s): Andoy, Francis Nikko	**Instructor:** Engr. Jimlord Quejado
Pulgado, Crishen Luper S.	
Punay, Heidee	
Santiago, David Owen	

A. Output(s) and Observation(s)

Table 14-1. Best- and Worst-Case Analysis using Theoretical Tools

Algorithm	Best case	Worst case	Analysis
Bubble sort	O(n)	O(n^2)	**(Pulgado)**: For the best case, it is O(n) when the array or the elements are already sorted. Hence, it will do only one full pass (n-1) comparisons and 0 swaps since it is already sorted. However, if the elements are in reverse order, it will compare every time until the array is already sorted O(n^2). **(Andoy):** For my analysis here, I've first include the needed libraries like the chrono, ctime, algorithm, and cstdlib, these libraries are essential when creating this activity. Since we want to see the time it completed the sorting process, I don't want to make the array myself since I created 100k elements. That is why I used cstdlib to make a random number that consists of 100k elements. For the best case of bubble sort the array is already sorted that is why when I run the program it finished incredibly fast. Moreover, the bubble sort is best when the array is already sorted, since it will just run through the array. Wherein the worst case is much longer, because the numbers are random a lot of swap and compare was needed for it to be sorted. When the sort has started, we understand why bubble sort is not used much when dealing with large datasets. The algorithm is based on two nested for loops, and hence time complexity is O(n^2). That is, in the case of 100,000 elements, it will be required to make about 10 billion comparisons

			(Punay): In the best case, it is O(n) because the list i already sorted, it only does a check of the elements in the list once, where it does not need to make any swaps anymore. In the worst case, when the list is in reverse order, bubble sort becomes slow because it is gonna keep on comparing and swapping elements repeatedly, which will cause the time to grow quickly as the list gets bigger.
Merge sort	O(n log n)	O (n log n)	**(Pulgado):** The merge sort uses the divide and conquer strategy. So, it will split the array into halves (2T(n/2)) and do the merging process which compares the and joins the elements once per level O(n). So for each level of recursion, the total work for merging all parts is n and the number of levels of splitting is $O(n)$. They have the same time complexity because the time depends on how many times the array gets split and how much work is done when merging. So even if the array is already sorted or unsorted, it still does the same, which divides the array into halves and merges them back, comparing every element. **(Andoy):** For my analysis here, just like in the first algorithm. include the necessary libraries. For this best case to present my insertion sorting function at its best. My code begins with the creation of a 100,000 item array, though not using random numbers, I used a perfectly sorted array. This simply implies that the data array simply contains. It first chooses an element to be inserted such as the 3 and then compares the element it chooses to the one that preceded it, such as the 2. The inner loop of the algorithm verifies whether the last data (2) was more than the next one (3). In this ideal case, this check will never be true since the array is sorted already. This is an indication that the hard work, the while loop, is entirely avoided. The role of this is just merely running through the entire array once, and in effect, merely ensuring that all the elements are already in place. Wherein worst-case code, insertion sort algorithm at its slowest possible rate. My program fills a 100,000 element array, however, it is this time arranged in a perfect reverse order. The data that I produced appears to be. At the beginning of the execution of my insertion sort function, it attempts to insert something, such as 99 into the sorted portion of the array. It is relative to this 99 of the 100s that are already classified in the sorted. Naturally it finds that 99 is smaller. This makes the inner while loop to execute and that was what I wanted to test. All those 100s move one spot to the right just so as to fit. This issue is only magnified when I have to

			insert next in the array with a 1 since my algorithm has already gone to the very end of the array. It must divide that 1 by all the 99,999 elements that preceded it. This huge number of relocations needs billions of transactions. This is as much as my insertionsort program can possibly accomplish. This is precisely what the O(n^2) worst case implies. **(Punay):** In merge sort, the results stayed almost the same in both the best and worst cases. Merge sort splits the list into halves until it can't be split anymore, and then it merges them back, sorted, even if the list is already sorted. It runs in O(nlogn) because the process does not change in the order of the numbers. **(Santiago)**: *Bubble Sort:* The bubble sort's best case is when the array is already sorted, thus, the loop would only need to do one pass/comparison per element. In this case, the Big O notation would only be O(n), since we only need to do one run through the array which depends on its size. Meanwhile, the worst case is when the array is in reverse order, which means the loop would need to compare and swap all of the elements, hence, both outer and inner for loops would be iterated and executed through, thus, its asymptotic notation of $O(n^2)$. We iterate through the loop n*n times. *Merge Sort:* The merge sort has one of the most consistent time complexities due to its best and worst case being O(n log n). This is because of its divide and conquer strategy; it always divides the array into smaller parts before sorting and merging it, which means input size and order of elements are disregarded. Whether the array is in reverse order or already sorted, the algorithm would always divide and conquer then merge first.

Table 14-2. Best- and Worst-Case Analysis using Experimental Tools

	Bubble sort (For input sizes: 1,000, 10,000, and 50,000)	Merge sort (For input sizes: 1,000, 10,000, and 50,000)
Best case	Bubble Sort (Best): 0.002 seconds Bubble Sort (Best): 0.131 seconds Bubble Sort (Best): 3.1 seconds	Merge Sort (Best): 0 seconds Merge Sort (Best): 0.002 seconds Merge Sort (Best): 0.013 seconds
Worst case	Bubble Sort (Worst): 0.004 seconds Bubble Sort (Worst): 0.386 seconds Bubble Sort (Worst): 9.654 seconds	Merge Sort (Worst): 0 seconds Merge Sort (Worst): 0.003 seconds Merge Sort (Worst): 0.012 seconds

Bubble Sort Analysis	Merge Sort Analysis
Pulgado: Based on the given output, it takes much faster to sort the bubble sort and slower for the merge sort in a sorted array. **Santiago:** As seen in the outputs, the experimentation results reflect the results from our theoretical analysis. The best case for the bubble sort is pretty quick since it means that the array is already sorted, thus, the execution time is fairly quick and dependent only on the input size. However, the worst case—where the array is in reverse order—shows the execution time to approximately quadruple compared to the best case. This is also reflected from our theoretical analysis, where the worst case is O(n*n).	**Pulgado**: As for the worst case, the merge sort is much faster than the bubble sort which compliments the theoretical analysis. **Santiago:** The results of the experimentation also reflect the results of our theoretical analysis. As seen in the outputs, the execution time for both cases is approximately the same. This is because the algorithm performs the same steps—divide and conquer then merge—to sort the array, hence the O(n log n) time complexity. So even if the array is already sorted or in reverse order, the merge sort would still finish at roughly the same time. Input size is the only factor that would affect its execution time.

B. Answers to Supplementary Activity

Searching Algorithm	Pseudocode	Time complexity

Linear search	LinearSearch(A, n, key) for i = 0 to n - 1 if A[i] == key return i return -1	Best case: O(1) when element was found at first position Worst case: O(n) element is not found or at the end
Binary search	BinarySearch(A, low, high, key) while low <= high mid = (low + high) / 2 if A[mid] == key return mid else if A[mid] < key low = mid + 1 else high = mid - 1 return -1	Best case: O(1) element found in the middle on first try Worst case: O (log n) is when an array keeps dividing until one element remains

Input Size (N)	**Elapsed time for binary search (worst case)**	**Elapsed time for linear search (worst case)**
100000	Binary Search (Worst Case): 0 µs	Linear Search (Worst Case): 553 µs
10000000	Binary Search (Worst Case): 1 µs	Linear Search (Worst Case): 54980 µs
100000000	Binary Search (Worst Case): 2 µs	Linear Search (Worst Case): 544125 µs

ANALYSIS:

Pulgado:
For both scenarios of worst cases of the searching algorithm, in which the key is not found, we see the results that the binary is much faster than the linear search. In linear search, each element is searched one by one until the end. While the binary search keeps dividing until one element remains. So when the key element is not in the array, the linear search keeps searching each of the thousands of elements, while the binary keeps dividing the thousands of elements which makes it more efficient. Hence, the linear search execution time grows significantly as the n is increasing while the binary search also grows but with minimal increase.

Andoy:
After comparing the two searching algorithms, we can see that the linear search follows the time complexity O(n). This demonstrated a clear and direct relationship between input size, since when the “n” increases the time also increases these 2 are directly proportional linearly. I made sure that I put the target key as the final element in the array, so that I would have the worst case scenario. My results on the tests proved the theory exactly: with the input size of 100, 000 the search took 553 microseconds. When I increased the array size by a factor of a thousand to ten million items, the run time was nearly linear and now the run time became 54,980 microseconds. This linear scaling persisted as I scaled to a hundred million elements which took ten times the time, 544,125 microseconds, of the prior run. This information clearly indicates that the time it takes on my linear search to process increases directly with the number of elements that it has to look through. With these examples my explanation earlier is proven.

Punay:
In this activity, I learned that some algorithms get very slow very quickly when the input becomes large, like bubble sort and linear search. However, merge sort and binary search are fast. This activity helped me understand the importance of choosing the right algorithm for a problem.

Santiago:
This testing activity shows how much faster a binary search algorithm is compared to linear search. To compare them, we specifically tested the worst cases with increasing input sizes from 100,000 to 100,000,000. Using std::chrono, we were able to print the time it took to execute the searches in terms of nanoseconds. As we can see, since the worst case for binary search is only O(log n), it took significantly less time to execute compared to the linear search with the same input sizes. This shows the stark difference between a linear and logarithmic time complexity. The linear search algorithm's execution time, with O(n) as the worst case, increases linearly with the input size. However, in the case of the binary search, there is only a certain input size where it would increase until it plateaus and steadily continues in almost a straight horizontal line.

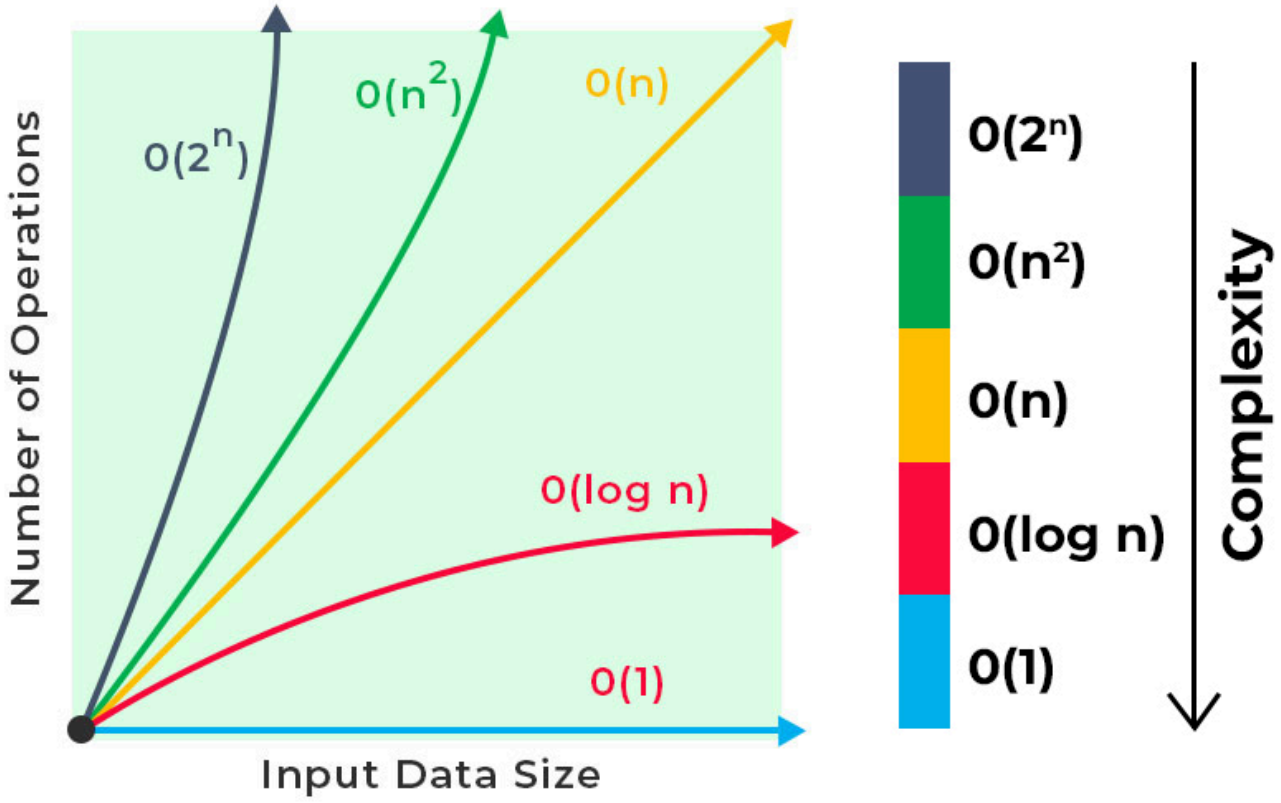

Image source: https://www.geeksforgeeks.org/dsa/what-is-logarithmic-time-complexity/

It perfectly reflects the difference between the graphs of the two functions. O(n) keeps increasing, but O(log n) stops increasing after a certain point and continues on steadily. In conclusion, in similar conditions, the binary search would typically beat the linear search in terms of time complexity—with the caveat being that binary search doesn't work with unsorted sets of data.

C. Conclusion & Lessons Learned

Pulgado:
In conclusion, this activity has taught me the relationship between time complexity and space complexity. It also taught me the factors that may affect the time complexity of the program which are the hardware, ADT'S, compiler, and the coding itself. As the input size increases, the more time it takes to execute the program. With different algorithms comes different results of the time. Hence, we use theoretical analysis to predict the time it takes for the worst, best, and average. Then, we apply the empirical for the exact results of the time complexity.

Andoy:
To conclude, I have learned so much not just how to code, but to also understand how fast and efficient the program is. My experiment has made me realize how crucial the connection between the theoretical time complexity of an algorithm and its practical, real-life performance is, which I have never been made aware of before. It also made me aware of the numerous factors that might influence the execution time of a program, many of which I hadn't stressed out on previously, such as the particular hardware that I am running, what ADTs I use such as an array, what compiler translates my logic and even the idiosyncrasy of the code itself. Therefore, this activity proved the critical collaboration of two forms of analysis. The theoretical analysis, and its worst, best, and average cases, are used to theorize the behavior and scalability of an algorithm as the data size increases, and this is precisely what the time complexity has enabled me to do.

Punay:
In this activity, I learned that some algorithms get very slow very quickly when the input becomes large, like bubble sort and linear search. However, merge sort and binary search are fast. This activity helped me understand the importance of choosing the right algorithm for a problem.

Santiago:
In conclusion, this activity encapsulates the principles of algorithm analysis. Here, I was able to explore and understand the significance of both theoretical and experimental analyses of algorithms. Additionally, I improved my understanding of the different asymptotic analyses of the algorithms I previously encountered. I achieved these by exploring the graphs, the notations, and the results of the experimentations. Based on my learnings, I also realized that active learning through experimentation helps me better understand these complicated principles in this subject. Previously, I never fully understood the reason why bubble sort and merge sort would have their specific Big O notations; and the same can be said with linear and binary searching. However, by tinkering and experimenting with different codes, input sizes, and algorithms, I finally understood them as I compared and contrasted the differences in outputs and structure. I believe that I will be able to apply my learnings here in my future endeavors as I integrate programming and algorithms in my future creations.

D. Assessment Rubric

E. External References